View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Padua@thesis

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Triennale in Ingegneria Informatica

"Sviluppo di un software per il controllo di un robot di telepresenza via Brain-Computer Interface"

Laureando

Gloria Beraldo

Relatore

Prof. Emanuele Menegatti

Co-relatore

Dr. Luca Tonin

Dr. Marco Carraro

Anno Accademico 2014/2015

Alla mia famiglia che mi ha
sempre sostenuto durante questi anni.

Gloria Beraldo

Indice

Abstract

Migliorare, almeno in parte, la qualità di vita di molte persone affette da gravi disabilità, a volte paralizzate e costrette a rimanere a letto, mi ha spinto a condurre l'attività sperimentale presentata in questa tesi. Diverse infatti sono le patologie che colpiscono il canale neuromuscolare: sclerosi laterale amiotrofica (SLA), lesioni al tronco encefalico, al cervello, al midollo, paralisi cerebrale, distrofia muscolare, sclerosi multipla, ictus e numerose altre malattie che danneggiano il controllo neuronale dei muscoli e i muscoli stessi. Molte delle persone affette da queste patologie non sono autosufficienti, la loro vita sociale è minima o assente e nei casi più gravi sono *locked-in*, ovvero sono coscienti ma non possono muoversi e comunicare a causa della completa paralisi di tutti i muscoli volontari del corpo. Per permettere a tali soggetti di interagire con familiari e amici e partecipare alle loro attività, anche a distanza, ho focalizzato questa tesi sull'integrazione di una Brain-Computer Interface (BCI) di tipo non invasivo e un robot di telepresenza, in particolare un TurtleBot2, con l'aggiunta di una webcam necessaria per il collegamento video. La BCI è una tecnologia predisposta a controllare dei dispositivi esterni non manualmente ma attraverso il pensiero, senza la partecipazione effettiva di nervi periferici e muscoli.
L'utente è così in grado di guidare il robot solamente tramite la sua attività cerebrale senza nessun movimento e contemporaneamente osservare lo spazio circostante mediante stream video. Infine il software sviluppato è stato testato in un esperimento presso il laboratorio di Sistemi Autonomi Intelligenti IAS-Lab dell'Università di Padova.

Citazioni

«È quando le aspettative sono ridotte a zero che si apprezza veramente ciò che si ha.»
Stephen Hawking

«Quando sei così, non hai alternative.»
Ambrogio Fogar

«È difficile accettarsi quando non sei più quello di prima: ogni impulso è una frustata, ogni desiderio una ferita, nelle mie condizioni di vita devi chiedere aiuto anche per grattarti il naso. »
Ambrogio Fogar

«La voce è importante.»
Stephen Hawking

Stephen William Hawking (Oxford, 8 gennaio 1942) è un fisico, matematico, cosmologo e astrofisico britannico, fra i più importanti e conosciuti del mondo, noto soprattutto per i suoi studi sui buchi neri e l'origine dell'universo.
Affetto dal morbo di Gherig, rara patologia neurodegenerativa che distrugge le cellule nervose e che provoca la paralisi progressiva dei muscoli motori volontari, da molti anni bloccato su una sedia a rotelle, è incapace d'emettere suoni con la voce se non attraverso un sintetizzatore collegato a un computer.
Immobilizzato, prigioniero del corpo malato, da tempo ha perso anche l'uso delle mani e può comunicare con il mondo solo con il battito di una palpebra.

Ambrogio Antonio Fogar (Milano, 13 agosto 1941 - Milano, 24 agosto 2005) è stato un navigatore, esploratore, scrittore e conduttore televisivo italiano. Dal settembre del '92, quando la sua jeep si ribaltò sulla pista del raid Parigi-Mosca-Pechino, Fogar rimase imprigionato in un corpo immobile, doveva essere assistito, lavato, vestito, pettinato e imboccato, l'unica terapia per le lesioni al midollo subite era accettarsi, lottare, evitare il peso dei ricordi e la disperazione per qualche abbandono.

Capitolo 1

Introduzione

Diverse sono le patologie che coinvolgono il canale neuromuscolare, responsabile della comunicazione tra cervello e muscoli: sclerosi laterale amiotrofica (SLA), lesioni al tronco encefalico, al cervello, al midollo, paralisi cerebrale, distrofia muscolare, sclerosi multipla, ictus e numerose altre malattie che danneggiano il controllo neuronale dei muscoli e i muscoli stessi. La maggior parte dei soggetti colpiti da queste patologie hanno perso completamente il controllo dei muscoli, nei casi peggiori anche quello degli occhi e della respirazione, fino ad essere completamente *locked-in*, incapaci di comunicare in nessun modo. [1]
Infatti il canale neuromuscolare è di estrema importanza: ogni volta che si vuole compiere un certo movimento, il cervello, attraverso i percorsi neuromuscolari, comunica ai muscoli quali azioni sta per compiere.
Tuttavia la scienza moderna può permettere a questi individui, anche quelli *locked-in*, di vivere più a lungo e in maniera migliore.
Per loro esistono tre alternative possibili: [1]

- rinforzare e tentare di sfruttare le capacità dei muscoli sani. Per esempio i pazienti che hanno avuto lesioni al tronco encefalico possono usare il movimento degli occhi per rispondere a semplici domande, dare comandi o lavorare con un programma di *word processing*.

- ristabilire le funzioni nei percorsi neuromuscolari che controllano i muscoli. Nei pazienti con lesioni al midollo, tramite l'elettromiografia (EMG) dei muscoli sopra il livello delle lesioni, è possibile controllare direttamente la stimolazione degli arti paralizzati e ristabilire il loro movimento.

- ripristinare le funzioni motorie, fornendo un nuovo canale di comunicazione non motoria e di controllo, la Brain-Computer Interface (BCI), attraverso il quale è possibile inviare messaggi e comandi al mondo esterno.

1.1 Cos'è la Brain-Computer Interface (BCI)

L'interfaccia BCI è un sistema che permette la connessione diretta tra il cervello umano e un computer [2]. Si tratta di un ambito che ha affascinato l'uomo e i ricercatori di informatica, neuroscienze e ingegneria biomedica a partire dagli anni '90 [3]:

tramite l'EEG (elettroencefalogramma) si possono riconoscere determinati compiti mentali effettuati dalle persone, capire il loro intento e cosa vogliono comunicare senza usare nervi periferici e muscoli. Questa concezione, apparsa in diversi film e fiction, negli ultimi anni è stata approfondita a livello scientifico fino a convogliare nella nascita della BCI [1].
Con il termine BCI si intende infatti la tecnologia predisposta a controllare dei dispositivi esterni non manualmente, ma attraverso il pensiero senza la partecipazione effettiva di nervi periferici e muscoli [1, 4].
La BCI è quindi in grado di tradurre mediante tecniche di *machine learning* e *signal processing* l'attività cerebrale registrata e poi fornire in output l'intento dell'utente, per esempio muovere una sedia a rotelle o selezionare una lettera su una tastiera virtuale [3].
Esistono due grandi famiglie di BCI, non invasiva e invasiva [4]:

- **invasiva:** utilizza degli elettrodi impiantati all'interno della corteccia del cervello mediante intervento chirurgico. Traduce quindi l'intento dell'utente a partire dall'azione dei neuroni registrata all'interno della corteccia cerebrale e sulla superficie.
  La BCI di tipo invasivo fornisce sicuramente dei segnali più intensi rispetto a quella non invasiva, ma può comportare dei rischi clinici, come le infezioni causate dall'usura degli elettrodi.

- **non invasiva:** registra l'attività cerebrale senza intervenire chirurgicamente all'interno del cervello e permette un controllo e una comunicazione basilare alle persone con disabilità. Tuttavia recentemente i ricercatori hanno dimostrato che una BCI non invasiva basata sui segnali EEG, può fornire un controllo 3D paragonabile a quello ottenuto con una di tipo invasivo.
  Per monitorare l'attività cerebrale esistono una varietà di metodi [3]:

  – **tramite elettroencefalogramma (EEG)**: si posizionano degli elettrodi sullo scalpo da cui si ricavano dei segnali elettrici. Essi rappresentano l'attività di milioni di neuroni, infatti a causa della complessa geometria elettrica e spaziale del cervello e della variabilità dell'attività cerebrale, è davvero molto difficile ricavare informazioni dettagliate su un singolo neurone o di una piccola area del cervello [1].

  Tuttavia l'attività neuronale produce non solo segnali di tipo elettrico, ma anche magnetico e metabolico, che possono essere usati dalla BCI [3].

  – **magnetoencefalografia (MEG)** per registrare i campi magnetici
  – l'attività metabolica del cervello si riflette sul flusso del sangue che può essere osservato mediante **tomografia a emissione di positroni (PET)**, **risonanza magnetica funzionale (fMRI)** e **immagini ottiche**.

  Tuttavia queste tecniche richiedono degli strumenti sofisticati che possono operare solo in specifiche strutture.
  Al contrario la modalità dell'EEG è più economica e più pratica per le applicazioni che utilizzano la BCI [3].

1.2 Applicazioni della BCI

La motivazione principale dei ricercatori e sviluppatori della BCI fu quella di accrescere in primis le capacità delle persone disabili e migliorare la loro interazione con l'esterno [2]. Infatti, dato che la BCI non richiede la partecipazione effettiva di nessun muscolo e nervo periferico, può costituire davvero una chance per coloro che sono affetti da SLA, lesioni al tronco encefalico, al cervello, al midollo, paralisi cerebrale, distrofia muscolare, sclerosi multipla e numerose altre malattie che danneggiano il controllo dei muscoli [1].
Grazie al continuo processo di ricerca della BCI le potenziali applicazioni pratiche stanno aumentando e possono essere ricondotte a quattro aree principali: *Communication and control, Motor substitution, Entertainment e Motor recovery* [5].

- **Communication and control:** attraverso l'uso della BCI si ha la potenzialità di permettere agli individui anche con gravi disabilità di comunicare con altre persone e di controllare il loro ambiente. Le principali funzioni di comunicazione possono consistere nell'inviare/ricevere email, chattare con il telefono e navigare in internet. Riporto di seguito alcuni degli esperimenti di questa macro-area, eseguiti negli ultimi anni [5].

  - **uso della BCI come supporto ai *spelling device*:** si tratta di dispositivi che vengono utilizzati principalmente da coloro che soffrono di SLA e basati sulla modulazione delle onde cerebrali [5]. In un sistema di *spelling* viene presentato sullo schermo del computer una matrice contenente le lettere dell'alfabeto inglese, i numeri decimali e un carattere per spaziare le parole. Nella versione classica, in Figura 1.1, si tratta di una matrice 6x6 [3, 5].

Figura 1.1: P300 speller: matrice 6x6 contenente le lettere dell'alfabeto inglese, i numeri decimali e un carattere per spaziare le parole. Si sfrutta la componente delle onde cerebrali "P300", particolarmente adatta per la BCI, che si presenta a causa della reazione di una persona allo stimolo a circa 300 ms [6].

In essa le righe e le colonne vengono illuminate in maniera intermittente, mentre al soggetto viene richiesto di mantenere l'attenzione su un simbolo particolare (per esempio per mantenere la concentrazione può contare il numero di accensioni del simbolo interessato). Si può osservare, al momento della scelta, una variazione del potenziale in corrispondenza della

zona parietale ed occipitale del cranio. Il rilevamento, l'analisi e la classificazione di tali segnali sono compiti della BCI.
Come si può intuire, il limite di questo sistema è la capacità di mantenere la concentrazione per molto tempo. Tuttavia lo stesso soggetto può migliorare la sua abilità ed efficienza con l'esercizio [3, 5, 6].

– **uso della BCI per la navigazione in Internet:** nella prima implementazione detta "Descartes" (1999) veniva mostrata ai pazienti che soffrivano di SLA una finestra principale del browser e da lì potevano collegarsi ad una delle pagine predefinite memorizzate, vedi Figura 1.2 A [5]. La pagina selezionata a sua volta conteneva in ordine alfabetico i

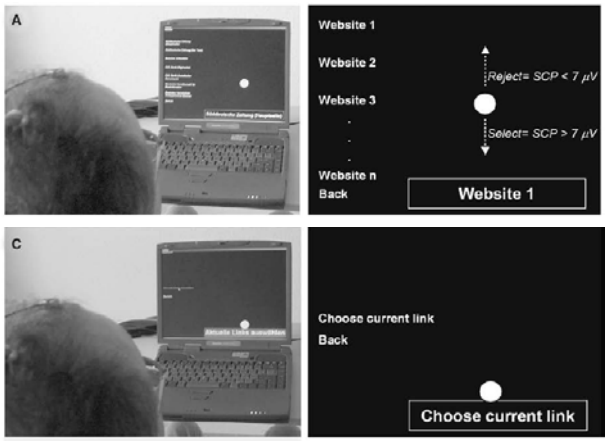

Figura 1.2: Prima implementazione di un BCI browser risalente al 1999 detta "Descartes": ai pazienti che soffrivano di SLA veniva mostrata una finestra principale dal quale potevano collegarsi a delle pagine predefinite. Ogni pagina selezionata conteneva altri possibili link in ordine alfabetico [8].

possibili link che potevano essere raggiunti, come delle foglie di un albero, in Figura 1.2 C. In seguito ci fu una seconda versione "Nessi", [5] in Figura

Figura 1.3: Seconda versione di un BCI browser nominata "Nessi": viene illustrato come avviene la ricerca a partire dalla pagina principale di Google mediante tastiera virtuale [7].

1.3, in cui si migliorò la grafica e si aggiunse una tastiera virtuale.
Grazie all'uso della BCI la selezione dei link fu facilitata [5].

– **BCI come dispositivo di assistenza**: la BCI può essere riconosciuta come una speciale tecnologia di assistenza nell'ambito dell'ICT (Information and Comunication Technologies), ovvero tutte le apparecchiature che sono in grado di aiutare le persone a ricevere, inviare e produrre informazioni in diverse forme. Esse comprendono infatti i dispositivi per vedere, ascoltare, leggere, scrivere, telefonare e avvisare [5].

– **hybrid Brain-Computer Interface (hBCI):** molte persone con gravi disabilità per varie ragioni devono utilizzare dei dispositivi di assistenza (*assistive devices*, AD) per permettere loro ogni giorno di comunicare e divertirsi.
Esistono diversi tipi di AD da semplici *switches* connessi ad un *controller* remoto fino a complessi sensori attaccati al computer in grado di rilevare il movimento degli occhi. Esempi di dispositivi di assistenza sono illustrati in Figura 1.4. Tuttavia questi dispositivi funzionano correttamente solo

Figura 1.4: Esempi di *assistive devices AD*: in alcune situazioni può essere un' alternativa agli AD. Nel 2010 infatti è stata sviluppata un "hybrid BCI (hBCI)" che combina gli AD con la BCI.

dopo essere adattati sulla specifica persona che deve utilizzarli e ci sono casi in cui l'effetto degli AD non è adeguato, spesso a causa di stanchezza del paziente. In tali situazioni la BCI può essere una buona alternativa, in quanto usa il segnale del cervello per il controllo senza la necessità di movimento [9]. I ricercatori dunque nel 2010 hanno sviluppato una "hybrid BCI (hBCI)" che combina gli esistenti dispositivi con la BCI [5]. Un utente può decidere di utilizzare normalmente lo specifico AD, senza sfruttare la BCI. Tramite l'hBCI si può stabilire quindi quali dei due canali offre un segnale più affidabile, si può passare da un canale all'altro bilanciando la fatica e la stanchezza dei pazienti, migliorando la velocità, l'accuratezza e in genere la performance del sistema [5, 9].

- **Motor substitution:** l'uso della BCI permette alle persone colpite da lesioni al midollo spinale, affetti da paralisi motoria e perdita funzionale delle dita e delle mani, di accrescere le proprie capacità di afferrare oggetti. Inoltre le persone disabili possono interagire con amici e familiari sfruttando la telepresenza [5].

– **Accrescere la capacità di afferrare oggetti:** In Italia si stima che ci siano 85000 persone colpite da lesioni al midollo spinale, con un ritmo di 2400 nuovi casi all'anno e 7 al giorno[1]. In Europa, secondo uno studio risalente al 2012, 330000 cittadini europei presentano ferite al midollo[2], un numero elevato di questi é dotato delle minime funzioni motorie e sono costretti a dipendere dai loro operatori socio-sanitari, peggiorando così la loro qualità di vita. L'unica opportunità per tali soggetti è la Functional Electrical Stimulation (FES) ovvero la stimolazione elettrica funzionale che tenta di ripristinare, tramite eccitazione diretta di strutture nervose, le funzioni motorie perse, attivando indirettamente il muscolo. Tuttavia FES non è in grado di restituire al corpo il naturale movimento, ma può aiutare le persone paralizzate a vivere in modo migliore.
I pioneri, Heidelberg e Graz, hanno connesso una BCI a un sistema FES, permettendo alle persone paralizzate di controllare i loro muscoli con la mente. I soggetti dovevano indossare una cuffia con degli elettrodi da cui si ricavavano i segnali elettrici come si vede in Figura 1.5. Grazie

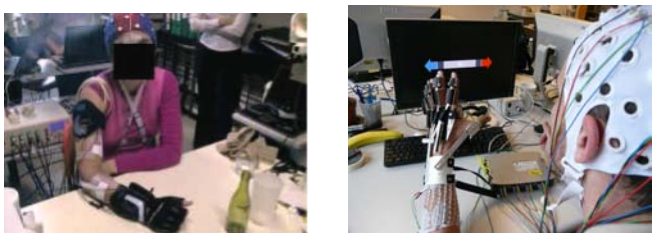

Figura 1.5: Utenti che stanno indossando il set completo durante un esperimento [10].

alla BCI, Heidelberg e Graz hanno dimostrato che i pazienti affetti da una completa paralisi motoria e da una perdita funzionale delle dita e mani, riuscivano a completare una serie di compiti, in cui veniva chiesto loro di afferrare degli oggetti, tramite l'immaginazione dei movimenti dei piedi [5].

– **BCI come telepresenza e assistenza negli spostamenti:** attraverso il cervello e in seguito mediante l'analisi cerebrale svolta dalla BCI si riesce anche a controllare una sedia a rotelle o a guidare un robot mobile equipaggiato con dei sensori per riconoscere gli ostacoli, vedi Figura 1.6 [5]. Lo scopo della telepresenza è infatti quello di permettere alle persone disabili di interagire con gli amici e i familiari e partecipare alle loro attività, anche a distanza, tramite uno schermo, un microfono e una webcam [11].

[1] http://www.faiponline.it/page.asp?menu1=6&menu2=14&menu3=8&menu4=&notizia=360&page=1
[2] http://www.eurostemcell.org/it/factsheet/lesioni-del-midollo-spinale-come-potrebbero-aiutare-le-cellule-staminali

Figura 1.6: Telepresenza: grazie all'analisi dell'attività cerebrale svolta dalla BCI si riesce a controllare una sedia a rotelle o a guidare un robot mobile [12].

- **Entertainment:** questa area è tipicamente riconosciuta come a bassa priorità nello sviluppo della BCI. In questo contesto la BCI può facilitare per esempio la ricerca all'interno di un album fotografico o di una playlist musicale mediante il controllo basato sull'umore. Infatti questi sistemi possono offrire opportunità agli utenti di esprimere il loro stato d'animo e i desideri più rapidamente ed espressivamente rispetto all'uso della scrittura [5]. Per esempio un'applicazione degli ultimi anni è l'utilizzo della BCI per dipingere, offrendo così agli utilizzatori la possibilità di esprimere la loro creatività.
  In seguito riporto alcuni sviluppi di questa macro-area:

  – **Gioco:** i giochi che possono essere controllati mediante l'uso della BCI, come si vede in Figura 1.7, permettono alle persone gravemente disabili un minimo di divertimento e, a volte, un miglioramento della loro qualità di vita [5]. I giochi possono essere competitivi (richiedono veloci

Figura 1.7: BCI game: due soggetti, i cui comandi mentali sono determinati dalla BCI, mentre giocano a un videogame. Essi indossano una cuffia con degli elettrodi da cui si ricavano i segnali elettrici [13].

    comandi) o strategici (di solito più lenti). Un esempio è Pacman, in cui si richiede all'utente di immaginare di girare a destra e a sinistra: Pacman cambia così direzione appena viene riconosciuto il comando mentale, altrimenti si muove fino a raggiungere i muri dove si ferma [3]. Questi giochi usano diversi protocolli della BCI: alcuni si basano sui comandi mentali per controllare degli aspetti del gioco, altri ricavano dagli EEG delle informazioni che consentono di adattarne la dinamica.

  - **Realtà virtuale:** Diversi prototipi sono stati in grado di navigare in ambienti virtuali sfruttando l'attività cerebrale registrata dagli EEG elettrodi posizionati sullo scalpo [5]. Alcuni partecipanti sani hanno potuto esplorare spazi virtuali e manipolare oggetti virtuali, mentre i pazienti con lesioni al midollo spinale sono stati in grado di controllare una sedia a rotelle che si muoveva attraverso una strada virtuale. Questi esperimenti, che comportano uno sforzo mentale, possono essere utili per allenare i soggetti e migliorare le loro performance [5].
  - **Cercare una canzone e creazione di una playlist:** con la nascita del formato di compressione mp3 ci fu un boom di dispositivi che permettono di ascoltare musica, grazie ai quali si possono creare delle playlist in cui ognuno può inserire una collezione di brani musicali. Grazie alla BCI si possono creare playlist basate sullo stato d'animo dell'utente e delle canzoni all'interno del dispositivo [5].
  - **Ricerca di foto all'interno di un album:** spesso capita, mentre si cerca una particolare immagine, di selezionare delle foto che non sono conformi a quelle che si aspettava di vedere.
    Così i ricercatori, dopo aver focalizzato l'attenzione su che tipo di foto le persone conservano e come le organizzano, hanno provato a combinare la BCI con delle tecniche di ricerca di immagini [5]. L'utente seleziona una determinata immagine e le tecniche di ricerca provvedono ad individuarne alcune simili basandosi anche sull'analisi cerebrale [5].
    Queste procedure possono essere sfruttate per stimolare la memoria ed eventualmente il soggetto può intervenire manifestando cosa vorrebbe vedere [5].

- **Motor recovery:** i danni motori in seguito a incidenti o ictus sono la principale causa di disabilità [5]. Il recupero delle funzioni delle mani e braccia è cruciale nelle attività di ogni giorno, ma spesso variabile e incompleto. La riabilitazione non è sempre efficace, infatti una percentuale che varia dal 30 al 60% dei pazienti non è in grado di utilizzare effettivamente le braccia.
  Attualmente le neuroscienze, basate sulla riabilitazione, cercano di stimolare le funzioni motorie sfruttando i processi cognitivi per riorganizzare la materia plastica non lesionata, dopo un ictus o incidente, e per acquisire nuove abilità nell'organismo. La maggior parte delle tecniche riabilitative si basa sulla mobilitazione attiva e passiva: l'articolazione del paziente viene sottoposta ad una serie di movimenti regolari ripetitivi, a volte con l'aiuto di un terapista, con lo scopo di ripristinare un adeguato livello di funzionalità dell'arto. Negli ultimi anni in alcuni centri specializzati in possesso della BCI, queste attività furono integrate con esercizi in ambienti virtuali [5]. Essi prevedono *feedback* per accelerare le capacità di apprendimento con l'utilizzo di dispositivi di assistenza e di robot, sono utili per una terapia di lunga durata e rappresentano una nuova opportunità per i pazienti. Infatti questi esperimenti sono basati sull'abilità dei pazienti di eseguire azioni, che richiedono l'uso di mani e braccia e anche attività motoria, attraverso l'immaginazione con assenza di movimenti reali. Questo approccio permette, dopo una fase di allenamento, di migliorare

le performance motorie e di produrre dei cambiamenti alla plasticità corticale. Infatti allenamenti di tale genere hanno portato ad esiti positivi in persone che hanno subito ictus, con miglioramenti nell'uso delle mani e nella concentrazione mentale. Inoltre questo metodo costituisce una valida alternativa quando l'allenamento fisico non è possibile [5].

1.3 Scopo della tesi

La riduzione di mobilità nell'esperienza di molte persone ha un profondo impatto sull'indipendenza personale, l'attività sociale e l'autostima. Lo testimoniano le parole di Stephen Hawking e di Ambrogio Fogar riportate nelle citazioni iniziali: si apprezza ciò che si ha, non si hanno alternative, è difficile accettarsi.
Per questi motivi il mio intento è stato quello di svolgere un'attività sperimentale per la tesi, mirata a migliorare almeno in parte la qualità della vita di diverse persone che soffrono di gravi disabilità, costrette a rimanere a letto oppure a dipendere dai loro operatori socio-sanitari.
Ho così deciso che lo scopo della mia tesi fosse lo sviluppo di un software per il controllo di un robot di telepresenza via Brain-Computer Interface (BCI). In particolare mi sono focalizzata sull'introduzione di un sistema innovativo, finora non ancora sperimentato presso l' Università di Padova, che prevede l'integrazione di un TurtleBot2 con una BCI di tipo non invasivo.
L'idea alla base era quello di permettere al soggetto, soprattutto se in stato *locked-in* o paralizzato, di poter interagire con i suoi amici o parenti, che si trovano in altre stanze. Sul robot infatti viene posizionata una webcam che riprende l'ambiente circostante e a sua volta l'utente può vedere lo streaming da un computer.
Inoltre l'utente è in grado di controllare il movimento del robot usando solamente i segnali provenienti dal cervello umano, senza la partecipazione di nervi periferici e muscoli.
Tuttavia l'obiettivo della mio lavoro non era quello di studiare, analizzare l'attività cerebrale, processare i segnali elettrici e classificarli, ma l'implementazione del codice necessario a far comunicare la BCI con l'utente e il robot. Quest'ultimo oltre a ricevere i comandi dell'utente è anche in grado di riconoscere la presenza di ostacoli e di fermarsi in caso di perdita di connessione.
L'utente quando è pronto a iniziare invia un comando di `start`, mentre per sospendere la navigazione e bloccare il robot quello di `stop`, come si vede in Figura 1.8.
Per completare il mio progetto ho testato presso il laboratorio di Sistemi Autonomi Intelligenti IAS-Lab[3] dell' Università di Padova il software che ho sviluppato. Come goal intermedio mi sono prefissata di riuscire a guidare il robot da una stanza all'altra senza l'uso della BCI, fornendo i comandi da tastiera e osservando gli spostamenti nell'ambiente circostante mediante collegamento video.
Infine ho condotto l'esperimento completo utilizzando la BCI per dare i comandi di movimento al robot in base alla mia attività cerebrale.

[3] `http://robotics.dei.unipd.it`

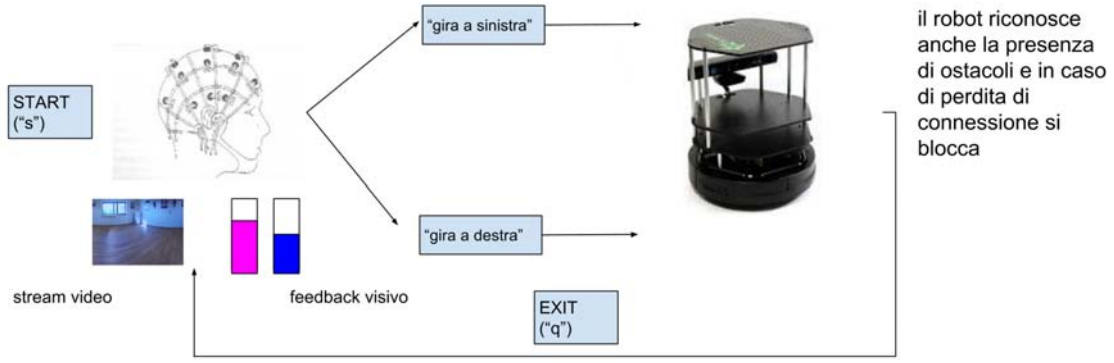

Figura 1.8: Integrazione della BCI con un robot di telepresenza: l'utente inizialmente invia un comando di `start`. Poi può guidare il robot nell'ambiente circostante, scegliendo tra due classi di comandi "gira a sinistra" e "gira a destra". Il robot si comporta di conseguenza, inoltre evita gli ostacoli e quando perde la connessione si blocca. Il loop continua fino a quando l'utente non invia il comando di `stop`. Infine sullo schermo del suo computer il soggetto vede un *feedback* che mostra gli input inviati e uno stream video dell'ambiente di fronte al robot.

1.4 Struttura della tesi

La struttura della tesi riflette il mio percorso sperimentale. Nel capitolo 2 vengono illustrate le metodologie e gli strumenti che ho approfondito inizialmente, in particolare presento la Struttura della BCI in sezione 2.1 che comprende: l'utente (2.1.1), l'acquisizione (2.1.2), il *pre-processing* (2.1.3), l'estrazione delle *feature* (2.1.4), la classificazione (2.1.5) e il *feedback* (2.1.6). In sezione 2.2 introduco Robot Operating System (ROS) che ho dovuto usare per lo sviluppo del codice e descrivo il robot (sezione 2.3) che ho utilizzato nell'esperimento. Infine in sezione 2.4 è presente una descrizione dell'infrastruttura di telepresenza necessaria per creare il collegamento video con cui l'utente può osservare l'avanzamento del robot nell'ambiente circostante.
Il capitolo 3 è dedicato all'analisi del codice che ho sviluppato: il trasmettitore BCI (sezione 3.1), il ricevitore BCI (sezione 3.2), il software di navigazione (sezione 3.3) e il tester di connessione (sezione 3.4). Il codice completo è presente rispettivamente nelle appendici A,B,C,D.
Il capitolo 4 presenta l'esperimento che ho svolto presso il laboratorio di Sistemi Autonomi Intelligenti IAS-Lab: la descrizione dell'esperimento (sezione 4.1), i risultati (sezione 4.2) e conclusioni e lavoro futuro (sezione 4.3) in cui propongo degli spunti per migliorare ed espandere la mia applicazione.

Capitolo 2

Metodologia

2.1 Struttura della BCI

La generale architettura della BCI è costituita da diversi blocchi come si vede in Figura 2.1: l'utente, l'acquisizione dei segnali elettrici, il *pre-processing*, l'estrazione delle *feature*, la classificazione e il *feedback* [1, 3, 9]. L'attività cerebrale viene regi-

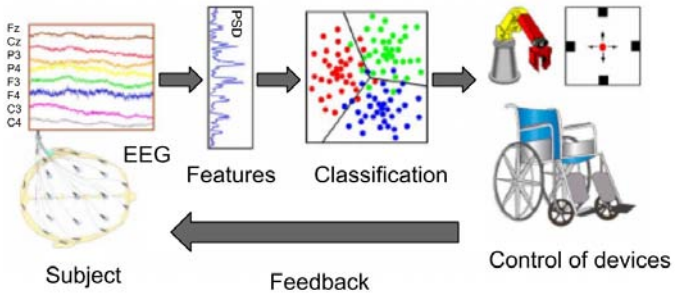

Figura 2.1: La principale architettura della BCI: inizialmente si acquisiscono i segnali elettrici, successivamente vengono processati per estrarre specifiche *feature*. Tali *feature* riflettono l'intento dell'utente e vengono tradotte in comandi per il controllo di uno specifico dispositivo (per esempio un programma di *word processing*, una sedia a rotelle o neuroprotesi). Alla fine all'utente viene restituito un *feedback*, la parte più essenziale del BCI loop, che gli permette di migliorare e acquisire la giusta strategia di controllo [1, 3]

strata mediante degli strumenti. Questi segnali sono inizialmente processati e puliti dal rumore per estrarre alcune rilevanti *feature*, poi vengono passati come input ad alcuni modelli matematici. Tali modelli calcolano, dopo una fase di allenamento, gli appropriati comandi mentali per il controllo del corrispondente dispositivo. Infine un *feedback* visivo o di altro tipo come stimolazione tattile, informa il soggetto riguardo alla sua performance [3].

2.1.1 Utente

L'utente, la persona che utilizza la BCI, è l'elemento fondamentale di questo sistema. Infatti a seconda dell'analisi della sua attività cerebrale si ha uno specifico controllo della BCI.
All'utente viene assegnato un compito mentale detto *task*, che deve essere semplice da eseguire e deve generare segnali cerebrali significativi per garantire la ripetibilità dell'esperimento e motivarlo [15]. I *task* cambiano tipologia a seconda del segnale di controllo che si vuole estrarre e dell'applicazione a cui si è interessati.
Un particolare tipo di BCI è quella basata sul *Motor Imagery* (MI), ovvero sfrutta la correlazione tra i segnali elettrici cerebrali e l'immaginazione dei movimenti motori [1]. MI può essere definito come uno stato dinamico in cui la rappresentazione di una specifica attività motoria è internamente provata dal soggetto, ma non prevede alcun movimento esterno [14]. All'utente viene richiesto di immaginare mentalmente di eseguire dei movimenti, ma senza effettivamente muoversi, permettendo così l'attivazione delle regioni del cervello corrispondenti. I *task* di immaginazione motoria possono essere di diversi tipi: aprire o chiudere mano destra e sinistra e movimenti di entrambi i piedi [14].
Tali *task* mentali sono tradotti dalla BCI basata sul *Motor Imagery* in azioni e attività di ogni giorno per attuatori esterni, per esempio: raggiungere e afferrare una tazza o altri oggetti, girare una pagina di un libro, usare un programma di *word processing* e guidare una sedia a rotelle [5, 14].

2.1.2 Acquisizione

Per registrare i segnali elettrici provenienti dal cervello, si usano degli elettrodi posizionati sullo scalpo, quindi non si considera l'attività cerebrale dei singoli neuroni, ma soltanto di popolazioni di neuroni in superficie [2, 16]. Spesso gli elettrodi sono attaccati a delle cuffie, un esempio in Figura 2.2, che ne facilitano il posizionamento e ne impediscono il movimento durante la registrazione. Di solito si usano dei *wet electrodes*, sopra dei quali bisogna applicare una pasta. Tuttavia sono stati inventati anche dei *dry electrodes* che non richiedono l'uso del gel [5].

Figura 2.2: Cuffia con gli elettrodi per l'acquisizione dei segnali elettrici provenienti dal cervello.

Nel mio esperimento l'attività cerebrale viene acquisita attraverso 16 canali attivi EEG secondo la configurazione internazionale 10-20 illustrata in Figura 2.3: Fz, FC3, FC1, FCz, FC2, FC4, C3, C1, Cz, C2, C4, CP3, CP1, CPz, CP2 e CP4 [14].

La continua fluttuazione della normale attività cerebrale induce tra vari punti del

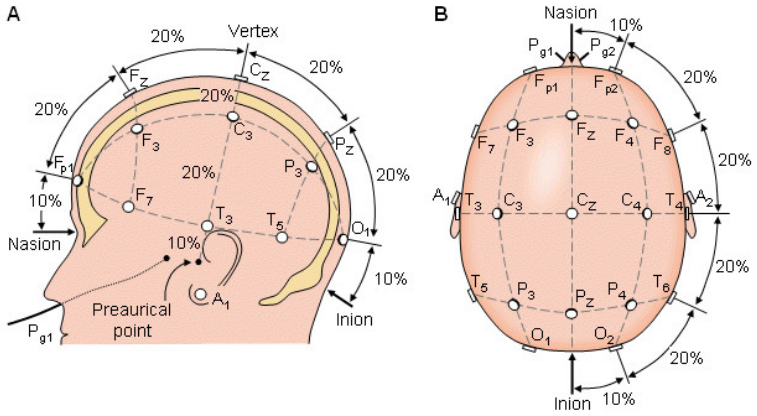

Figura 2.3: Posizionamento elettrodi secondo sistema internazionale 10-20 [17].

cuoio capelluto piccole differenze di potenziale elettrico (microvolt) che vengono amplificate e registrate. Si ottiene in questo modo un tracciato che segna per ciascun elettrodo le variazioni del voltaggio nel tempo.
Solitamente, del segnale elettroencefalografico vengono studiati con analisi in frequenza i ritmi base: delta (0.1-3.9 Hz), theta (4-7.9 Hz), alfa (8-13.9 Hz), beta (14-30 Hz) e gamma (30-42 Hz). Si cerca infatti in questi ritmi delle alterazioni rispetto allo stato di normalità [6].

2.1.3 Pre-processing

Osservando una registrazione dell'elettroencefalogramma si nota la presenza del rumore di sottofondo, disturbi di natura elettrica. Sono quindi necessarie delle operazioni di *pre-processing* ovvero di pulizia, ottimizzazione e correzione del segnale, con lo scopo di rimuovere il rumore ed evidenziare l'informazione trasportata, massimizzando l'SNR (Signal to Noise Ratio) [1, 6]. Inoltre, per aumentare l'accuratezza sarebbero necessarie diverse misurazioni a seguito dello stesso stimolo [1].
Un esempio di rumore è rappresentato dai cosiddetti artefatti. Si tratta di picchi di tensione provocati dal movimento delle sopracciglia, dei bulbi oculari, della testa e della mascella. Per la rimozione degli artefatti esistono delle tecniche di *machine learning* [1, 6] che sono oltre lo scopo della tesi.

2.1.4 Estrazione delle feature

Una volta ottenuto un segnale pulito si estraggono delle *feature*, ovvero dei parametri rilevanti nell'analisi dei segnali su scala temporale, delle frequenze e tramite approcci tempo-frequenza. Esse possono evidenziare alcuni eventi come l'attivazione del ritmo sinaptico nella corteccia sensomotoria che produce un ritmo mu o beta [1]. La conoscenza di tali fatti può guidare la BCI nella fase di classificazione. Infatti una buona selezione delle *feature* costituisce la chiave del successo per gli algoritmi di classificazione. Tuttavia altre volte rappresentano dei parametri autoregressivi

che sono correlati con l'attività del cervello, ma non necessariamente esprimono un specifico evento [1].

2.1.5 Classificazione

Una delle più importanti fasi nell'architettura della BCI è quella di classificazione, che consiste nell'identificare le intenzioni del soggetto a partire dal segnale registrato e dalle *feature* estratte precedentemente [1, 3]. Per completare tali operazioni si sfruttano delle tecniche di *machine learning* (classificatori, reti neurali o Support Vector Machines (SVM)) [6]. Tramite la classificazione si ottengono le informazioni necessarie per il controllo di un dispositivo esterno: i comandi possono essere continui (movimento di un cursore) o discreti (selezione di una lettera) [1, 3]. Ognuno di essi deve essere il più possibile indipendente: per esempio la scelta del movimento di un cursore non dipende da nessun altro evento [1].
Nel mio esperimento si sono utilizzati dei classificatori bayesiani: l'output di classificazione è rappresentato da due probabilità che descrivono se il comando dell'utente produca un movimento verso destra o verso sinistra. La più alta tra le due determina l'esito.

2.1.6 Feedback

Il soggetto è in grado di valutare la sua performance mediante dei *feedback*, che gli permettono di acquisire la giusta strategia di controllo dello strumento e di effettuare delle scelte più rapide per raggiungere l'obiettivo [1, 3]. Il *feedback* risulta così la parte più essenziale dell'architettura della BCI perché porta l'utente a migliorare le sue prestazioni [1]. Infatti esso deve sapere cosa fare, conoscere l'effetto delle sue scelte precedenti mediante dei *feedback* e di conseguenza nelle prove successive fornire i giusti comandi.
A seconda dell'applicazione i *feedback* possono essere di diverso tipo: caratteri usati per comporre parole, output audio o grafici che rappresentano l'attività del cervello, comandi motori, movimenti di un robot o di una sedia rotelle, spostamento del cursore nella direzione scelta come nel mio caso [3, 4, 5].

2.2 Robot Operating System (ROS)

ROS è una piattaforma software nel campo della robotica, utile strumento nella progettazione e test di applicazioni robotiche [18, 19, 20, 21, 22]. Lo sviluppo di ROS ebbe ufficialmente inizio nel 2007 sotto il nome di Switchyard presso lo Standford Artificial Intelligence Laboratory a supporto del progetto Standford Artificial Robot (STAIR) e del programma Personal Robotics (PR). Nel 2008 il progetto continuò presso Willow Garage, importante ente di ricerca che si interessa alla robotica da anni, e dove venne rinominato con Robot Operating System (ROS). Nel 2013 venne rilasciata la prima versione con molte delle caratteristiche odierne e diventò un sofware open source. Nello stesso anno ROS è passato alla Open Source Robotics Foundation(OSRF), che tuttora lo mantiene.

Esistono diverse distribuzioni: ROS Box Turtle, ROS C Turtle, ROS Diamondback, ROS Electric Emys, ROS Fuerte Turtle, ROS Groovy Galapagos, ROS Hydro Medusa e quella che ho utilizzato ROS Indigo Igloo.

2.2.1 Che cos'è ROS

ROS è innanzitutto un *framework* di programmazione open source che mette a disposizione strumenti e librerie utili per aiutare gli sviluppatori software nello sviluppo di applicazioni robotiche a partire dalla scrittura fino all'esecuzione e al *debugging* del codice.
ROS è quindi anche una collezione di librerie e di *tools*, tra i più popolari quelli di simulazione, navigazione, SLAM (Simultaneous Localization and Mapping), percezione e poi quelli che permettono l'astrazione di dispositivi hardware a basso livello come joystick, GPS, Controller e Laser e sonar.
Oltre ad essere un *framework*, ROS presenta alcune caratteristiche di un sistema operativo (gestione di processi, di pacchetti e delle loro dipendenze, a basso livello di dispositivi) e allo stesso tempo è un *middleware* perché permette la comunicazione tra processi/macchine diverse.
Infine costituisce un' architettura distribuita in cui è possibile gestire in maniera asincrona un insieme di moduli software. Questi ultimi posso essere in `C++`, `Python` e `Lisp`.
Le caratteristiche e funzioni di ROS, vengono riassunte in Figura 2.4.

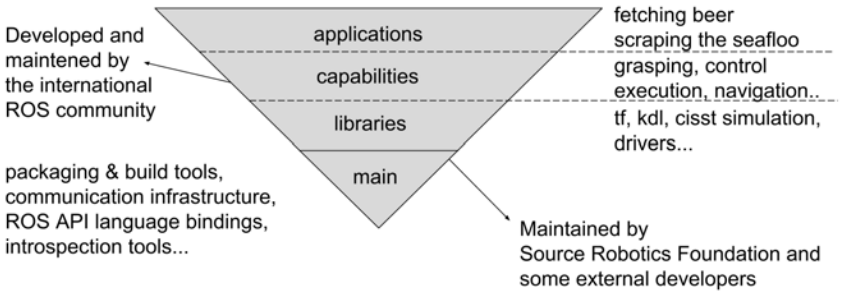

Figura 2.4: Le principali caratteristiche e funzioni del sistema ROS, suddivise in livelli di sviluppo: il primo livello fornisce pacchetti, strumenti per lo sviluppo del codice, infrastrutture che permettono la comunicazione. Il secondo contiene le librerie come tf, kdl, cisst simulation. Il terzo è riservato alle capacità di controllo, di esecuzione, navigazione e infine l'ultimo alle applicazioni.

2.2.2 Che cosa non è ROS

ROS non deve essere confuso con un sistema operativo, anche se come elencato in precedenza presenta alcune similitudini, viene definito piuttosto un meta sistema operativo.
Nella tabella in Figura 2.5 vengono illustrate le principali caratteristiche e differenze

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

Figura 2.5: Confronto tra un sistema operativo e ROS.

tra un tipico sistema operativo e ROS.
Il sistema operativo attualmente supportato per ROS è Ubuntu, anche se esistono diverse piattaforme sperimentali: Arch, Debian, Fedora, Gentoo, Mac OS X, OpenSuse, Windows.
Inoltre ROS non costituisce un linguaggio di programmazione, ma integra del codice scritto in `C++`, `Python` e `Lisp`. Esistono delle librerie sperimentali per Java e Lua.
Non è un ambiente di sviluppo IDE, anche se fornisce *tools* per lo sviluppo, *debugging* e simulazione.
Infine non è una vera e propria architettura real time, tuttavia è possibile integrare ROS con codice *realtime* e renderlo un *framework realtime*.

2.2.3 Obiettivi e vantaggi ROS

ROS permette il riutilizzo del codice anche con sviluppo di nuovi robot o algoritmi, in modo che gli sviluppatori non debbano reimplementare nuovamente gli applicativi precedentemente scritti. Uno degli obiettivi di ROS è infatti quello di garantire una piattaforma software compatibile con la maggior parte dei robot anche con caratteristiche molto differenti tra loro. Quindi nonostante il continuo sviluppo nel campo della robotica ROS garantisce il riciclo del codice esistente.
Gli sviluppatori poi, grazie alle librerie e gli strumenti forniti da ROS, possono focalizzarsi solamente sull'applicazione senza badare alla scrittura di codice integrativo.
ROS è flessibile, è in grado di adattarsi alle esigenze dell'utente ed è leggero. Non a

caso uno degli obiettivi di ROS era la leggerezza, per permettere la sua integrazione anche con altri *framework*.
ROS agevola inoltre la fase di *testing*, infatti permette di simulare hardware e software di basso livello e fornisce la possibilità di salvare i dati ottenuti dai sensori per poi analizzarli.

2.2.4 ROS software

In ROS il software viene organizzato in `package`, l'unità di gestione del codice ROS. Ogni `package` può contenere più eseguibili (`nodi`) ROS.

- **Comunicazione:** la comunicazione rappresenta uno degli elementi più significativi del sistema ROS: l'architettura ROS è un grafo in cui i diversi nodi comunicano tra loro. La possibilità di permettere un facile scambio di messaggi tra i nodi rende possibile la creazione di un programma *multi-threading*, tralasciando le problematiche relative alla politica di sincronizzazione e comunicazione tra i vari processi.
  Esistono due tipi di comunicazione:

  - **Publisher/subscriber:** modalità di comunicazione asincrona. La scrittura di un messaggio avviene su di un `topic` messo a disposizione dal `roscore` (collezione di nodi e programmi che costituiscono i prerequisiti del sistema ROS. Il `roscore` deve essere lanciato per prima di qualsiasi nodo e avvia il `ROS Master`, `ROS Parameter Server` e il `rosout logging node`). Tutti i nodi che desiderano ricevere tale messaggio possono richiederlo al `roscore`.

  - **Service:** modalità di comunicazione sincrona secondo la semantica *request/reply*. Un nodo invia una richiesta a tutti i nodi in grado di soddisfarla. Da questi nodi riceverà una risposta.

- **Messaggi (msg):** un messaggio è una struttura di dati con diversi campi. Esistono diversi tipi di messaggi a seconda del loro contenuto, usato per memorizzare delle informazioni che i nodi scambiano.
  Ogni messaggio è strutturato come una lista di tipi (sono supportati tutti i tipi standard primitivi integer, floating point, boolean, array e costanti) e dei nomi. Ogni messaggio viene memorizzato in un file .msg contenuto all'interno di una sottocartella nel `package`.

- **Nodi:** ROS è progettato per essere modulare permettendo una maggior tolleranza agli errori e quindi un sistema è costituito da tanti nodi ognuno che gestisce una particolare funzione. ROS è strutturato intorno ad un nodo `master` che permette ai vari nodi di essere a conoscenza della presenza di altri nodi e quindi di comunicare tramite *peer-to-peer*. In questo modo la codifica a basso livello dell'indirizzo del computer a cui sono destinati i dati viene mascherato dai nodi. Inoltre il `master` tiene traccia per ognuno dei `topic`, dei `publisher`

e `subscriber` e gestisce i `servizi`.
I nodi usano una libreria (`ROS client library`) per la comunicazione con gli altri nodi, attraverso linguaggio `C++` (`roscpp`) o `Python` (`rospy`).
Uno dei vantaggi di ROS è che i nodi possono anche trovarsi su sistemi differenti.

- **Topic:** rappresentano un mezzo di comunicazione asincrono per lo scambio di messaggi secondo la semantica *publish/subscribe*. Un nodo invia un messaggio per pubblicarlo su un `topic`. Un nodo che è interessato ad un certo tipo di messaggio si sottoscriverà a quel `topic`. Per uno stesso `topic` possono esserci più `publisher` e più `subscriber`. Inoltre un nodo può pubblicare e/o sottoscriversi a più `topic`. In generale i `publisher` e `subscriber` non sono consapevoli dell'esistenza degli altri.
  Ogni `topic` è fortemente legato al tipo di messaggio che viene pubblicato e i nodi possono ricevere messaggi solo di quel tipo. Quindi all'interno di un `topic` è possibile scrivere o leggere solo un tipo di messaggio.
  L'idea alla base di questo meccanismo è di disaccoppiare la produzione dell'informazione da quella di consumo. Il `topic` quindi può essere considerato come un bus che ha un nome, a cui ognuno può collegarsi per ricevere o inviare dei messaggi.

2.2.5 Publisher/Subscriber

Per pubblicare un messaggio bisogna definire un oggetto della classe `ros::Publisher`

```
ros::Publisher name = node_handle.advertise<package_name::
message_type>(topic_name,buffer_size);
```

name è il nome che si da all'oggetto di tipo `ros::Publisher`
node_handle è un oggetto della classe `ros::NodeHandle` di cui un'istanza viene creato a inizio programma
package_name nome del pacchetto
message_type tipo di messaggio che si desidera pubblicare
topic_name nome in formato stringa del topic su cui pubblicare
buffer_size è la dimensione del buffer in cui vengono contenuti i messaggi.

Per pubblicare il messaggio è necessario invocare il metodo `publish(msg)` sull'oggetto creato precedentemente:

```
name.publish(msg);
```

Il messaggio rimarrà all'interno del buffer, la cui dimensione è stata definita alla creazione del `Publisher`, fino a quando non verrà processato da uno dei *thread* di `roscpp`, il `package` che contiene le librerie relative ai `topic` e ai `service`. In seguito il messaggio verrà spostato dal *thread* sul buffer di ogni `Subscriber` associato al

topic.

In parallelo ad un Publisher ci sarà un nodo Subscriber. Però a differenza del Publisher, il Subscriber non può sapere quando un messaggio arriverà. Per questo si utilizza una funzione di callback, chiamata per ogni messaggio in arrivo.

```
void function_name(const package_name::type_name &msg){...}
```

function_name è il nome della funzione di callback
package_name il nome del pacchetto ed è lo stesso del Publisher.
type_name è il nome del file che memorizza il messaggio ed è lo stesso del Publisher.
msg è il nome della *reference* che contiene il messaggio
All'interno della funzione di callback si ha accesso a tutti i campi dei messaggi in arrivo e si decide quali memorizzare o scartare. Il valore di ritorno della callback è void perchè è ROS che si occupa di gestire la chiamata.

Per creare un Subscriber, bisogna definire un oggetto della classe ros::Subscriber

```
ros::Subscriber name =
   nodle_handle.subscribe<package_name::message_type>
(topic_name,buffer_size, pointer_to_callback_function);
```

name è il nome che si da all'oggetto di tipo ros::Subscriber
node_handle è un' istanza della classe ros::NodeHandle che viene creata a inizio programma
package_name nome del pacchetto che contiene il nodo interessato
message_type tipo di messaggio che si desidera sottoscrivere
topic_name nome in formato stringa del topic al quale si vuole sottoscrivere
buffer_size è la dimensione del buffer in cui vengono memorizzati i messaggi non ancora processati.
pointer_to_callback_function è il puntatore alla funzione di callback definita precedentemente.

ROS esegue le callback solo quando gli viene detto esplicitamente di farlo mediante le funzioni:
ros::spinOnce() esegue tutte le callback pendenti da tutte le sottoscrizioni del nodo e ritorna il controllo
ros::spin() ordina a ROS di attendere e processare tutte le callback fino a quando il nodo è attivo. Ovvero è equivalente a :

```
while(ros::ok()){
    ros::spinOnce();
}
```

È molto utile usare questo tipo di comunicazione in situazioni con continuo flusso di dati, come per fornire comandi o parametri relativi allo stato di robot od ottenere informazioni derivanti dai sensori.

2.3 Robot

Il robot che ho utilizzato è un TurtleBot 2 visibile in Figura 2.6 [23, 24, 25].

Figura 2.6: TurtleBot 2 visto lateralmente e posteriormente.

TurtleBot 2 è una piattaforma robotica progettata per l'istruzione e la ricerca nel campo della robotica. Inoltre può essere un potente strumento utile nell'insegnamento di ROS, citato nella sezione precedente 2.2, e può rappresentare un supporto per lo sviluppo di questa tecnologia di avanguardia.
È dotato di sensori 3D che permettono di tracciare mappe e di navigare in ambienti interni.
Viene riconosciuto anche come l'hardware più economico sul mercato che permette di ottenere determinate performance.
Le principali caratteristiche del robot sono:

- **base Kobuki:** la base Yujin Kobuki è una piattaforma progettata specificamente per il TurtleBot. Tale base include una batteria di 2200 mAh Li -Ion che può essere sostituita con una batteria 4400 mAh Li-Ion per un utilizzo prolungato.

- **Gyro:** TurtleBot 2 è dotato inoltre di un giroscopio calibrato in fabbrica, il quale agevola l'uso e fornisce un migliore input ai sensori a basse velocità per la stima dello stato del robot e garantendo un *feedback* di posizione. Ha anche un *encoder* con una risoluzione di 1000x.

Per la rilevazione degli ostacoli il TurtleBot 2 è stato equipaggiato con un sensore

Figura 2.7: sensore SOKUIKI: sensore laser con un campo visivo di 240 gradi e un raggio massimo di 5600 mm [28].

Scanning range finder (SOKUIKI sensor), come si vede in Figura 2.7 [26, 27].
Si tratta di un sensore laser per scansionare l'ambiente circostante, misura fino a 4 metri di distanza e ha un campo visivo di 240 gradi con un raggio massimo di 5600 mm e risoluzione angolare di 0.36 gradi. Il principio della misurazione è basato sul calcolo della differenza di fase, grazie alla quale è possibile ottenere misure stabili con la minima influenza del colore e della riflettanza dell'oggetto.

2.4 Infrastruttura di telepresenza

All'interno del mio esperimento per guidare il robot di telepresenza è necessario un collegamento video che mostri al soggetto la scena davanti al robot, permettendogli così di controllarlo anche essendo in un'altra stanza. Una webcam viene posizionata sopra al robot per riprendere l'avanzamento e i movimenti nell'ambiente circostante. Contemporaneamente l'utente vede lo stream video sul suo computer e può decidere quali comandi dare al robot.
Per migliorare le prestazioni e non essere dipendente dalla rete ho utilizzato come software per lo stream video GStreamer, al posto di Skype (soluzione alternativa, ma più semplice).
GStreamer è una piattaforma software modulare per creare applicazioni multimediali basato sul concetto di *pipeline* (tubatura) in grado di collegare diversi elementi tra di loro in serie [29, 30, 31, 32]. Ognuno di questi componenti è contenuto in un *plugin*. Lo scopo di GStreamer è infatti quello di permettere la creazione di applicazioni multimediali sfruttando *plugin* già esistenti.
Per formare una *pipeline* i vari componenti sono collegati gli uni con gli altri tramite dei connettori *pad*, che si distinguono a seconda del tipo di dati e di connessione. Come si vede in Figura 2.8, dal punto di vista dei pad gli elementi si dividono in:

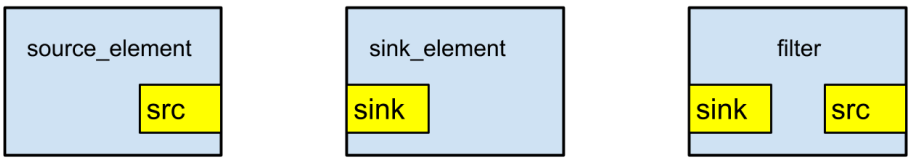

Figura 2.8: GStreamer: suddivisone degli elementi in base ai pad.

- *source* prelevano dati da fonti esterne come file, altri programmi, connessioni di rete. Hanno una uscita *source pad*.
- *sink* (pozzo), elemento finale che non ha uscite. Inviano dati provenienti dal programma a componenti esterne come file, altri programmi, connessioni di rete. Hanno un ingresso *sink pad*.

- *filter* permette di codificare/decodificare o di comprimere/decomprimere i dati provenienti dall'elemento precedente in modo che siano compatibili con altri parti del programma. Infatti possono essere collegati solo ad altri elementi e non comunicano con l'esterno. L'unico loro scopo è quello di modificare, convertire ed elaborare dati. Hanno almeno un *source pad* e un *sink pad*.

Dal punto di vista applicativo una *pipeline* si crea mediante il comando *gst-launch* e i vari elementi vengono divisi tra di loro mediante punti esclamativi (!). Inoltre è possibile settare alcune particolari proprietà e opzioni degli elementi che sono separate dagli spazi.
Nel caso di network streaming, come nel mio esperimento, la comunicazione avviene tra due host: un *sender* che invia dati e un *receiver* che li riceve. Il *sender* acquisisce il video, lo comprime, lo divide in più pezzi che invia grazie al livello del trasporto. Il *receiver* riceve i pacchetti, secondo i protocolli del livello del trasporto, li riassembla, li decomprime e visualizza sullo schermo il video.

Sender

Listing 2.1: sender

```
gst-launch v4l2src
   device=DEVICE!video/x − raw − yuv, width =WIDTH,
   height=HEIGHT, framerate =FPS ! ffmpegcolorspace !
   jpegenc ! multipartmux ! tcpserversink host=IPport =PORT
```

- `v4l2src`: indica che la source è una webcam
- `device`: posizione del dispositivo, di default assume il valore "/dev/video0"
- `video/x-raw-yuv`: è un *caps filter* per catturare l'immagine alla più alta risoluzione possibile del corrispondente device.
- `width`: larghezza della finestra del video che trasmette
- `height`: altezza della finestra del video che trasmette
- `framerate`: setta la frequenza dei frame
- `ffmpegcolorspace`: converte i frames del video in base ad una vasta gamma di formati cromatici
- `jpegenc`: è un elemento che codifica nel formato JPEG
- `multipartmux`: serve per generare un *multipart stream* costituito da diversi frames sottoforma di JPEG.
- `tcpserversink`: fa in modo che l'uscita vada in un server tcp che ha le proprietà elencate in seguito
- `host`: indirizzo IP del *receiver*

- `port`: porta

Receiver

Listing 2.2: receiver

```
gst-launch tcpclientsrc host=IPport =PORT !
   multipartdemux ! jpegdec ! autovideosink
```

- `tcpclientsrc`: la sorgente questa volta è un client tcp che ha le proprietà elencate in seguito
- `host`: indirizzo IP del *sender*
- `port`: porta
- `multipartdemux`: a partire da un *multipart stream* ricava i diversi frames di tipo JPEG
- `jpegdec`: è un elemento che decodifica dal formato JPEG
- `autovideosink`: visualizza lo stream video sullo schermo

Capitolo 3

Codice sviluppato

Questa sezione è dedicata alla presentazione e alla descrizione del codice.
Inizialmente ho sviluppato degli applicativi in C++: il **trasmettitore BCI** e il **ricevitore BCI**. Il **trasmettitore BCI** invia dei comandi al **ricevitore BCI**, secondo il modello *client-server* basato sul protocollo UDP. Contrariamente al TCP, UDP è un protocollo di comunicazione non orientato alla connessione, che quindi garantisce una maggior velocità, ma una minor affidabilità in reti molto grandi e trafficate. Ho scelto il protocollo UDP per la mia applicazione, in quanto la rete interna presente non ha una quantità di traffico significativa ed è di dimensioni piccole che rendono l'affidabilità di UDP molto elevata. Inoltre non si ha la necessità di creare uno stream, alla base del TCP, dato che il **trasmettitore BCI** invia solamente un byte, quindi un *datagram* di dimensione fissa, che contiene il comando scelto dall'utente.
Il **trasmettitore BCI** simula il ruolo della BCI: i comandi scelti dall'utente sono di stato (`start, exit`) o di controllo (`left, right`). L'utente preme sulla tastiera ''`s`'' per iniziare, ''`q`'' per fermarsi e infine le frecce sinistra e destra per muoversi. Questi input vengono poi codificati e inviati all'interno di un pacchetto al **ricevitore BCI**:

- START=1
- EXIT=2
- LEFT=11
- RIGHT=12

È stato necessario il **trasmettitore BCI** per la fase di test iniziale in cui ho pilotato il robot con la tastiera.
Il **ricevitore BCI** riceve i pacchetti e stampa il corrispondente contenuto su terminale. Tuttavia nella fase successiva ho modificato tale ricevitore, integrandolo con ROS.
Innanzitutto ho creato un messaggio di tipo `BCIMessage`: è costituito da due campi descritti da variabili `int8`, `type` che indica se si tratta di un comando di stato (0) o di controllo (1) e `value` che memorizza la corrispondente codifica descritta precedentemente.

```
int8 type //assume valori 0 o 1
int8 value  //assume valori 1, 2, 3, 11, 12
```

Il **ricevitore BCI** è quindi diventato un nodo ROS, appena riceve un messaggio, lo processa e setta adeguatamente i campi del `BCIMessage`. Infine pubblica il messaggio su `topic ‘‘/bci_message’’`.
Il mio progetto comprende, oltre al **ricevitore BCI**, altri due nodi ROS illustrati in Figura 3.1: uno per la **navigazione** e l'altro per il **test di connessione**.
Il **software di navigazione** si occupa di comunicare al robot i comandi di stato e di

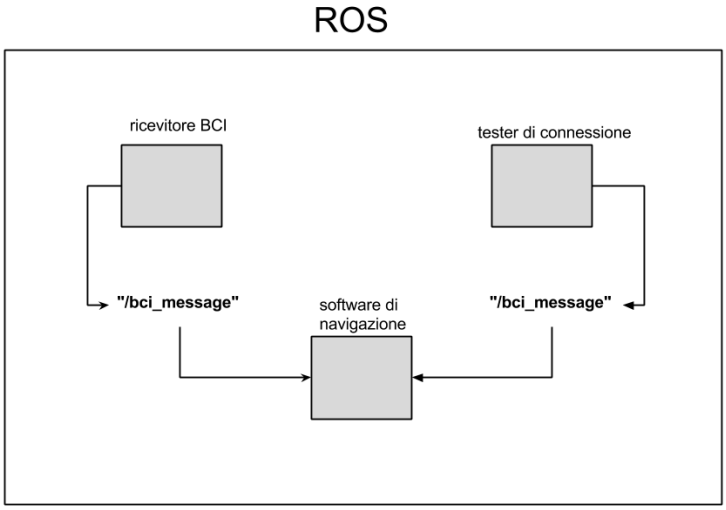

Figura 3.1: Nodi ROS presenti all'interno del mio progetto: il **ricevitore BCI** comunica con il software di navigazione su `topic ‘‘/bci_message’’` e lo stesso viene usato anche dal **tester di connessione**. Le frecce entranti rappresentano l'azione di scrittura su `topic` da parte dei nodi relativi al **ricevitore BCI e tester di connessione**. Invece le frecce uscenti indicano la lettura dal `topic` da parte del nodo riguardante il **software di navigazione**.

controllo, letti da un `subscriber` sul `topic ‘‘/bci_message’’`. Appena riceve `start` il robot inizia a muoversi dritto con una velocità lineare costante fino a quando non arriva un comando di `stop`, a quel punto sta in attesa di ricevere nuovamente uno `start`. I comandi `left` e `right` fanno ruotare il robot di 45° verso sinistra (senso antiorario) o destra (senso orario). Nel percorso il robot può incontrare degli ostacoli, quindi per rendere la navigazione più sicura il **software di navigazione** permette la ricezione di messaggi dal bumper e dal laser. Quando il laser segnala la presenza di un ostacolo il robot ruota in senso orario di un angolo casuale e poi riprende il moto con velocità lineare costante. Invece nel caso di ostacoli bassi, rilevati dalla pressione e dal rilascio del bumper, il robot indietreggia e ruota di un angolo casuale per poi muoversi dritto con la velocità precedente e costante.
Il **software di navigazione** comunica anche con il terzo nodo ROS **tester di connessione**. Quest'ultimo serve per verificare la connettività, è utile infatti nel

caso in cui la connessione tra la BCI , o per il test del **trasmettitore BCI**, e il computer posizionato sul robot venga persa. In questo caso il robot non deve più ricevere comandi, ma solamente fermarsi dato che non c'è più comunicazione con la BCI. Quindi il **tester di connessione** invoca la shell e richiama il comando di `ping`, leggendone l'output corrispondente.

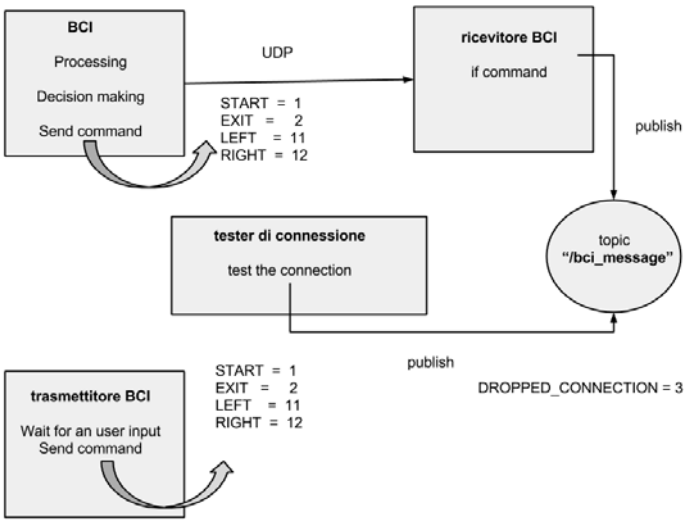

Figura 3.2: Struttura del codice sviluppato lato publisher: la BCI invia i comandi di stato e di controllo al ricevitore BCI. Quest'ultimo appena ha un comando lo pubblica su `topic ‘‘/bci_message’’`. Contemporaneamente il **tester di connessione** verifica la connessione. Nella fase di test iniziale ho utilizzato il **trasmettitore BCI**.

Quando la connessione cade pubblica su `topic ‘‘/bci_message’’` un messaggio di stato `BCIMessage` con `value` settato a 3. Il **software di navigazione** verifica tramite un `subscriber` la presenza sul `topic ‘‘/bci_message’’` di un `BCIMessage`

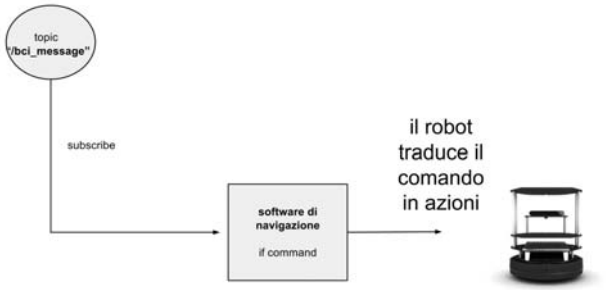

Figura 3.3: Struttura del codice sviluppato lato subscriber: il **software di navigazione** legge dal `topic ‘‘/bci_message’’`. Appena è presente un comando lo invia al robot che lo traduce in azioni

.

dovuto alla perdita di connessione e in caso affermativo stoppa il robot.
La struttura del codice sviluppato e la codifica dei comandi inviati tra i vari nodi viene riassunta nelle Figure 3.2 e 3.3.
Di seguito analizzo più nello specifico il **trasmettitore BCI**, il **ricevitore BCI**, il **software di navigazione** e il **tester di connessione** riportando degli estratti. Il codice completo è presente nelle appendici: il **trasmettitore BCI** (appendice A), il **ricevitore BCI** (appendice B), il **software di navigazione** (appendice C) e **tester di connessione** (appendice D).

3.1 Analisi del trasmettitore BCI

Il **trasmettitore BCI** svolge la funzione di *client*, che invia dei pacchetti UDP al **ricevitore BCI**. Ho creato perciò alla riga 58 un socket di tipo SOCK_DGRAM.

```
56 bzero(&sender,sizeof(sender));
57 //creazione del socket
58 sock= socket(AF_INET, SOCK_DGRAM, 0);
59 if (sock < 0) error("Socket failed");
60 sender.sin_family = AF_INET;
61 sender.sin_port = htons(atoi(argv[2]));
62 inet_pton(AF_INET,argv[1],&sender.sin_addr);
63 fcntl(sock,F_SETFL,O_NONBLOCK); //setto il socket di
     tipo non bloccante
```

L'indirizzo dell'host e della porta vengono passati come parametro da terminale (riga 52) e associati al corrispondente socket nelle righe 61 e 62.

```
51 if (argc != 3) {
52 std::cerr<<"Usage:"<<argv[0]<<" <host> <port> "
     <<std::endl;
53 exit(1);
54 }
```

Alla riga 63 ho settato il socket di tipo non bloccante in modo che la funzione `sendto` richiamata successivamente per inviare il *datagram*, non attenda nel caso in cui il buffer di trasmissione `buf` sia pieno, ma ritorni semplicemente -1.
Il cuore del programma inizia alla riga 66: vi sono due while annidati.

```
66  while(!quit) {  //itera finchè l'utente non digita q
67  std::cout<<"Where do you want to go?"<<std::endl;
68  while(waitcommand) {
69  ch=get_key_pressed();   //memorizzo il carattere
      digitato dall'utente in ch e ne faccio il parsing
70  switch(ch){//appena ho il comando setto a false
      waitcommand perché non devo più attendere e lo
      memorizzo in buf
71  case START:
72  waitcommand=false;
```

```
73  buf[0]=START_C;
74  break;
75  case RIGHT_ARROW:
76  waitcommand=false;
77  buf[0]=RIGHT;
78  break;
79  case LEFT_ARROW:
80  waitcommand=false;
81  buf[0]=LEFT;
82  break;
83  case EXIT:
84  waitcommand=false;
85  buf[0]=EXIT_C;
86  quit = true;
87  break;
88  //i due pass servono per mangiare dei caratteri
       aggiuntivi quando l'utente preme freccia sinistra o
       destra
89  case PASS:
90  case PASS1:
91  break;
92  default:
93  std::cout<<"The command sent is wrong: you must use
94  only left or right arrows. To quite press q" <<
       std::endl;
95  std::cout<<"Where do you want to go?"<<std::endl;
96  break;
97  }
98  }
```

Le condizioni di uscita dei while si basano su due variabili booleane:
- `quit` viene settata a true quando l'utente digita ''q'' e quindi termina il programma (riga 86)
-`waitcommand` indica se si sta aspettando che l'utente prema un tasto, appena si ha il comando viene settata a false. Il comando digitato dall'utente viene memorizzato in `ch` (riga 69) e si ottiene richiamando la funzione `get_key_pressed`. I comandi validi che possono essere scelti dall'utente sono i seguenti:

- START serve per avviare il robot e corrisponde al tasto ''s'' (START). Esso viene codificato dal **trasmettitore BCI** con START_C pari a 1.
- RIGHT_ARROW invia il comando di movimento a destra e si ottiene premendo la freccia destra. Viene codificato con RIGHT pari a 12.
- LEFT_ARROW invia il comando di movimento a sinistra e si ottiene premendo la freccia sinistra. Viene codificato con LEFT pari a 11.
- EXIT serve per fermare il robot e corrisponde al tasto ''q'' (EXIT). Viene codificato con EXIT_C pari a 2.

Qualsiasi altra scelta dell'utente non comporta nessun azione perché ritenuta sbagliata. In realtà quando decide di inviare i comandi di RIGHT_ARROW e LEFT_ARROW, premendo le frecce sulla tastiera, la funzione `get_key_pressed` riconosce tre carat-

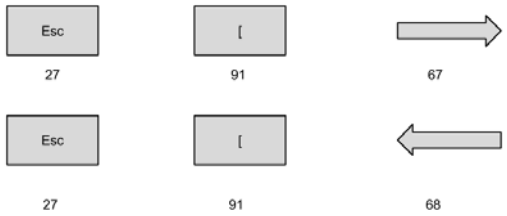

Figura 3.4: **Trasmettitore BCI**: quando l'utente invia i comandi RIGHT_ARROW e LEFT_ARROW la funzione get_key_pressed riconosce rispettivamente i caratteri "Esc" "[" 67 o 68.

teri distinti: "Esc", "[" e 67 o 68, illustrati in Figura 3.4. Dato che tutti questi tre caratteri sono necessari per dare comando di RIGHT e LEFT, nello switch sono stati aggiunti due casi PASS e PASS1 (righe 89-90) per non considerare "Esc" e il carattere "[" aggiuntivi.
Alla riga 99 si invia il comando scelto dall'utente e contenuto in `buf` tramite la funzione `sendto`, trattandosi infatti del protocollo UDP non si può usare direttamente la `read`.

```
99  n=sendto(sock,buf,1,0,(const struct sockaddr
       *)&sender,length);   //invio i comandi al
       BCI_receiver
100 if (n < 0) {
101 error("Sendto failed");
102 } else {
103 count++;     //ho inviato un comando, aumento il
       conteggio
104 std::cout<<"-  "<<count<<" Datagram sent to
       BCI_receiver"<<std::endl;
105 }
106 waitcommand=true; //appena inviati i comandi risetto a
       true waitcommand in attesa del prossimo comando
107 }
108 close(sock);//chiusura del socket
109 return 0;
110 }
```

`sendto` restituisce il numero di byte inviati, memorizzati in n. In caso di successo (riga 103) aumento `count`, variabile che conta il numero di *datagram* inviati, e risetto a true `waitcommand` perchè il **trasmettitore BCI** sarà in attesa nuovamente di un comando dell'utente (riga 106).
Quando l'utente decide di premere **''q''**, il while più esterno termina e alla riga 108 chiudo il socket.

3.2 Analisi del ricevitore BCI

Ho riportato nell' appendice B il **ricevitore BCI** già integrato con ROS.

```
37 ros::init(argc,argv,"server_robot");    //inizializza le
     librerie ROS
38
39 ros::NodeHandle nh("~");  //punto di acceso per
     comunicare con il resto del sistema ROS
40
41 ros::Publisher
     bci_chatter=nh.advertise<bci_to_ros::BCIMessage>
42 ("/bci_message",1); //definizione del publisher che
     pubblica su topic
     "/bci_message"ros::init(argc,argv,"server_robot");
     //inizializza le librerie ROS
43
44 ros::NodeHandle nh("~");  //punto di acceso per
     comunicare con il resto del sistema ROS
45
46 ros::Publisher
     bci_chatter=nh.advertise<bci_to_ros::BCIMessage>
47 ("/bci_message",1); //definizione del publisher che
     pubblica su topic "/bci_message"
```

Alla riga 37 chiamo infatti la funzione `ros::init` per inizializzare le librerie di ROS, come ultimo parametro vi è `server_robot` che costituisce il nome del nodo. L'oggetto `ros::NodeHandle` (riga 39) rappresenta il principale meccanismo con cui il mio applicativo interagisce con ROS. Quando viene richiamato registra il mio nodo nel sistema di ROS. ("~") indica che il nodo `server_robot` è privato.
Alla riga 41 ho definito il `publisher bci_chatter` che pubblica messaggi di tipo `BCIMessage` sul `topic` ‘‘/bci_message’’.
Il **ricevitore BCI** svolge la funzione di *server*, che riceve i pacchetti UDP dal **trasmettitore BCI**.

```
74 // creo socket, e assegno l'indirizzo tramite bind
75 sock=socket(AF_INET, SOCK_DGRAM, 0);
76 if (sock < 0) error("socket error");
77 length = sizeof(receiver);
78 bzero(&receiver,length);
79 receiver.sin_family=AF_INET;
80 receiver.sin_addr.s_addr=htonl(INADDR_ANY);
81 receiver.sin_port=htons(port);
82 if (bind(sock,(struct sockaddr *)&receiver,length)<0)
83 error("binding error");
84 sender_len = sizeof(struct sockaddr_in);
85 bzero(&buf1,BLEN1);
```

In modo analogo al **trasmettitore BCI** ho creato un socket di tipo SOCK_DGRAM alla riga 75. Questa volta la porta viene settata nel `launch` riportato nel box seguente 3.1, insieme ai parametri `start_c, exit_c, left, right, bci_type_state e bci_type_control` e acquisiti tramite la funzione `getParam` (righe 58-64).

Listing 3.1: porzione launch del ricevitore BCI

```
<launch>
<arg name="port" default="24000"/>
<arg name="start_c" default="1"/>
<arg name="exit_c" default="2"/>
<arg name="left" default="11"/>
<arg name="right" default="12"/>
<arg name="bci_type_state" default="0"/>
<arg name="bci_type_control" default="1"/>
```

```
57 // setto i parametri che vengono definiti nel launch
     corrispondente
58 nh.getParam("port",port);
59 nh.getParam("start_c",start_c);
60 nh.getParam("exit_c",exit_c);
61 nh.getParam("left",left_c);
62 nh.getParam("right",right_c);
63 nh.getParam("bci_type_state",bci_state);
64 nh.getParam("bci_type_control",bci_control);
```

Trattandosi di un *server* UDP è necessario la `bind` (riga 82) per associare al socket la porta e l'indirizzo del *client* con cui comunicare.

```
86 while (ros::ok()) {
87 //attendo un messaggio dal BCI sender o dalla BCI
88 n = recvfrom(sock,buf1,BLEN1,0,(struct sockaddr
     *)&sender,&sender_len);
89 if (n < 0) error("recvfrom");
90 else
91 {  //ho ricevuto un messaggio dal BCI sender o dalla BCI
92 icommand=static_cast<int>(command[0] & 0xFF);
93 bci_to_ros::BCIMessage msg; //creo messaggio di tipo
     BCIMessage
94 //ho ricevuto il messaggio, lo processo e setto i
     corrispondenti campi del messaggio BCIMessage
95 if(icommand == start_c) {
96 ROS_INFO("User sent start_command\n");
97 msg.type=bci_state;
98 msg.value=start_c;
99 } else if(icommand == exit_c) {
100 ROS_INFO("User sent exit_command\n");
```

```
101 msg.type=bci_state;
102 msg.value=exit_c;
103 } else if(icommand == left_c){
104 ROS_INFO("User sent left_command\n");
105 msg.type=bci_control;
106 msg.value=left_c;
107 }else if(icommand == right_c){
108 ROS_INFO("User sent right_command\n");
109 msg.type=bci_control;
110 msg.value=right_c;
111 }
112
113 bci_chatter.publish(msg);   //ora pubblico il messaggio
      precedentemente dichiarato
114
115 ros::spinOnce();
116
117 loop_rate.sleep();  //lo sospendo per evitare di
      pubblicare troppi messaggi ad ogni ciclo
118 }
119 bzero(&buf1,BLEN1);
120 }
121
122 ros::shutdown();
```

Specularmente al **trasmettitore BCI** alla riga 86 vi è presente un while che continua fino a quando l'utente non preme *Ctrl + c* facendo terminare il programma (riga 122). In questo loop il **ricevitore BCI** attende un *datagram* dal **trasmettitore BCI** e lo memorizza in `buf1` tramite la funzione `recvfrom` (riga 88). Essa restituisce il numero di byte ricevuti che vengono memorizzati in n.
Per assicurarmi di considerare i bit più significativi maschero `command` (puntatore char definito alla riga 53) con 0xFF e poi faccio un cast per memorizzarlo in `icommand`, variabile intera definita alla riga 54. A partire dalla riga 95 fino a 111 processo il comando ricevuto, contenuto in `icommand`, e setto i campi del `BCIMessage` definito alla riga 93. Come anticipato precedentemente, i comandi possono essere di:
- stato (`bci_state`): `start_c` e `exit_c`
- controllo (`bci_control`): `left_c` e `right_c`.
Infine alla riga 113 pubblico il messaggio `msg` di tipo `BCIMessage`. In questo caso chiamare la funzione `ros::spinOnce` alla riga 115 non sarebbe necessaria perché non sono presenti `callback`. Sull'oggetto `loop` definito alla riga 44 chiamo la funzione `sleep` (riga 117) che sospende il nodo per il tempo necessario a mantenere la frequenza di ciclo di 10 Hz e quindi fa in modo che il **ricevitore BCI** non pubblichi troppo velocemente i `BCIMessage`.

3.3 Analisi del software di navigazione

Per rendere più leggibile e pulito il codice del **software di navigazione** ho preferito scrivere un `basic_navigation.h` in cui riporto le firme delle funzioni utilizzate e il loro funzionamento.

Listing 3.2: basic_navigation.h

```
1 /*Restituisce un numero casuale compreso tra due estremi
    passati come parametro
2 @param double estremo inferiore dell'intervallo, double
    estremo superiore dell'intervallo
3 @return double
4 */
5
6 double rand_float( double low, double high );
7
8
9 /*Pubblica un messaggio di tipo Twist relativo alla
    velocità, a seguito di un movimento in avanti o
    indietro per un determinato tempo
10 @param constante double tempo per eseguire il movimento,
    int segno (+1=avanti, -1=indietro)
11 */
12
13 void go_straight(const double & time, int sign);
14
15 /*Pubblica un messaggio di tipo Twist relativo alla
    velocità, a seguito di un movimento rotatorio in
    senso orario o antiorario per un determinato tempo
16 @param constante double tempo per eseguire il movimento,
    int segno (+1= antiorario, -1=orario)
17 */
18
19 void turn(const double & rotation_time, int sign);
20
21 /* Callback che verifica l'arrivo di un messaggio della
    BCI: può essere di start, di stop, di avviso che è
    caduta la connesione, oppure di movimento a destra o
    sinistra
22 @param BCIMessage::ConstPtr puntatore per scrivere
    all'interno del campo del messaggio
23 */
24 void bci_callback(const
    bci_to_ros::BCIMessage::ConstPtr& bci_);
25
26
```

```
27 /* Callback che verifica la distanza rispetto a tre
      angoli fissati determinata dal laser
28 @param constante  LaserDistances::ConstPtr puntatore per
      scrivere all'interno del messaggio
29 */
30
31 void distances_callback(const
      basic_navigation::LaserDistances::ConstPtr &las_dist);
32
33
34 /*Callback che verifica se il bumper viene premuto e
      rilasciato
35 @param constante kobuki_msgs::BumperEventConstPtr
      puntatore per scrivere all'interno del messaggio
36 */
37
38 void bumperEventCB(const
      kobuki_msgs::BumperEventConstPtr msg);
```

Alla riga 12 del **software di navigazione** ho incluso *basic_ navigation.h.*

```
130 ros::init(argc,argv,"Navigation_node"); //inizializza le
      librerie ROS
131
132 ros::NodeHandle p; //punto di acceso per comunicare con
      il resto del sistema ROS
133 ros::NodeHandle nh("~"); //punto di acceso per
      comunicare con il resto del sistema ROS
134
135 srand (time(NULL));
```

Il **sofware di navigazione** è come il **ricevitore BCI** un nodo ROS, quindi alla riga 130 chiamo la funzione `ros::init` che ha come ultimo parametro `Navigation_node`. Vengono creati due oggetti `ros::NodeHandle` (riga 132 e 133), di cui uno privato. In questo nodo ho la necessità di avere anche un `nodle_handle` pubblico p (riga 132) per la presenza dei `subscribers`. Alla riga 135 `strand(time(NULL))` serve per inizializzare il generatore di numeri casuali, che viene richiamato all'interno della funzione `rand_float` (riga 42 `rand`).

```
139 //setto i parametri che vengono definiti nel launch
      corrispondente oppure valore di default equivale al
      terzo parametro
140 nh.param<double>("linear", linear, 0);
141 nh.param<double>("angular", angular, 0);
142 nh.param<double>("min_dist", min_distance, 0.5);
```

Questa volta il settaggio dei parametri (righe 140-142) può avvenire o tramite `launch` riportato nel box 3.3 o mediante il metodo `param` dell'oggetto `nh`. In quest'ulti-

mo caso le variabili corrispondenti `linear, angular, min_dist` assumono il valore passato come terzo parametro del metodo `param`.

- `linear` rappresenta la velocità lineare in m/s del robot.
- `angular` rappresenta la velocità angolare in rad/s del robot. Inizialmente il robot non ruota.
- `min_distance` la minima distanza in m a cui il robot può avvicinarsi ai vari oggetti presenti, altrimenti vengono dichiarati ostacoli.

Listing 3.3: porzione launch del software di navigazione

```
<launch>
<arg name="linear" default="0.1"/>
<arg name="angular" default="0"/>
<arg name="min_distance" default="0.25"/>
```

```
144 velocity.angular.z = angular;
145 velocity.angular.y = 0;
146 velocity.angular.x = 0;
147 velocity.linear.x = linear;
148 velocity.linear.y = 0;
149 velocity.linear.z = 0;
```

Ho completato quindi (righe 144-149) i campi del messaggio `velocity` di tipo `Twist` (descritto nel Listing 3.4) definito alla riga 26.

Listing 3.4: geometry_msgs

```
geometry_msgs/Vector3 linear
float64 x
float64 y
float64 z
geometry_msgs/Vector3 angular
float64 x
float64 y
float64 z
```

```
153 //Publishers
154 vel_pub = p.advertise<geometry_msgs::Twist
155 ("/cmd_vel_mux/input/navi",1);
156 blink_publisher = p.advertise< kobuki_msgs::Led
    >("commands/led1", 10);
```

Alle righe 154-156 vi è la definizione dei `publishers` che pubblicano rispettivamente messaggi di tipo `Twist` sulla velocità (righe 154-155) e di tipo `Led` per il controllo del led della base Kobuki del robot (riga 156). I messaggi di tipo `Twist` e `Led` sono riportati rispettivamente nei Listing 3.4 e 3.5.

Listing 3.5: Led.msg

```
# Sends a command for controlling the a LED.
# Typically the first LED is always reserved to denote
# the state - the remainder will be controllable.
uint8 BLACK   = 0
uint8 GREEN   = 1
uint8 ORANGE  = 2
uint8 RED     = 3
# For kobuki there are only two controllable LED's.
uint8 value
```

```
158 //Subscribers: uno per il laser, uno per il bumper e uno
      per la ricezione di messaggi dalla BCI
159 ros::Subscriber dist_sub =
      p.subscribe<basic_navigation::LaserDistances>
160 ("/distances", 1, distances_callback);
161 ros::Subscriber bumper_event_sub =
      p.subscribe("/mobile_base/events/bumper", 10,
      bumperEventCB);
162 ros::Subscriber
      bci_sub=p.subscribe<bci_to_ros::BCIMessage>
163 ("/bci_message",1,bci_callback);
164
165 //Per essere sicura che il laser sia attivo, sleep per 4
      secondi
166 ros::Duration(4).sleep();
```

Invece per leggere i messaggi di tipo `BCIMessage` dal `topic` ‘‘`/bci_message`’’ questa volta definisco un `subscriber` (righe 162-163) che utilizza la `callback` `bci_callback`. Non è l'unico `subscriber` infatti il **software di navigazione** riceve messaggi anche dal laser (di tipo `LaserDistances`, riportato nel box seguente 3.6,su `topic` ‘‘`/distances`’’)

Listing 3.6: LaserDistances.msg

```
float32[] distances
```

e dal bumper (su `topic` ‘‘`/mobile_base/events/bumper`’’).
Per assicurarmi che il laser sia attivo, richiamo alla riga 166 il metodo `sleep` che sospende l'esecuzione del corrispondente *thread* per quattro secondi.
Prima di procedere con l'analisi del main, descrivo più nel dettaglio le varie funzioni e `callback` che vengono richiamate in seguito:

```
41 double rand_float( double low, double high ) {
42 return ( ( double )rand() * ( high - low ) ) / ( double
     )RAND_MAX + low;
43 }
44
45 void go_straight(const double & time, int sign)
```

```
46 {
47 ros::Time now = ros::Time::now();
48 while((ros::Time::now() - now).sec < time )
49 {
50 //ROS_INFO_STREAM("actual - now: " << ros::Time::now() -
     now);
51 velocity.linear.x = sign * linear;
52 velocity.angular.z = 0;
53 vel_pub.publish(velocity);
54 ros::spinOnce();
55 }
56 }
57
58 void turn(const double & rotation_time, int sign) // 4.5
     seconds, 90°
59 {
60 ros::Time now = ros::Time::now();
61 while((ros::Time::now() - now).sec < rotation_time )
62 {
63 //ROS_INFO_STREAM("actual - now: " << ros::Time::now() -
     now);
64 velocity.linear.x = 0;
65 velocity.angular.z = sign * M_PI / 8;
66 vel_pub.publish(velocity);
67 ros::spinOnce();
68 }
69 }
70
71 void bci_callback(const
     bci_to_ros::BCIMessage::ConstPtr& bci_)
72 {
73
74 if(bci_->type==BCI_TYPE_STATE) {
75 switch(bci_->value) {
76 case START:
77 ROS_INFO("[StateMsg] - Robot has BCI control");
78 bci_control=true; bci_msg=false;
79 break;
80 case STOP:
81 ROS_INFO("[StateMsg] - pause-Robot stops");
82 bci_control=false;
83 break;
84 case DROPPED_CONNECTION:
85 ROS_INFO("[StateMsg] - connecion-Robot stops");
86 bci_control=false;
87 break;
```

```
88 default:
89 break;
90 }
91
92 } else if (bci_->type==BCI_TYPE_CONTROL) {
93 ROS_INFO_STREAM("[ControlMsg] - Received BCI_message: "
      << bci_->value);
94 mov_val = bci_->value;
95 bci_msg=true;       //set local state
96 }
97 }
98
99 void distances_callback(const
      basic_navigation::LaserDistances::ConstPtr &las_dist)
100 {
101 actual_distances[0] = las_dist->distances[0];
102 actual_distances[1] = las_dist->distances[1];
103 actual_distances[2] = las_dist->distances[2];
104 }
105
106 void bumperEventCB(const
      kobuki_msgs::BumperEventConstPtr msg)
107 {
108 // Preparing LED message
109 kobuki_msgs::LedPtr led_msg_ptr;
110 led_msg_ptr.reset(new kobuki_msgs::Led());
111
112 if (msg->state == kobuki_msgs::BumperEvent::PRESSED)
113 {
114 ROS_INFO_STREAM("Bumper pressed. Turning LED on.");
115 led_msg_ptr->value = kobuki_msgs::Led::GREEN;
116 blink_publisher.publish(led_msg_ptr);
117 ground_obstacle = true;
118 }
119 else // kobuki_msgs::BumperEvent::RELEASED
120 {
121 ROS_INFO_STREAM("Bumper released. Turning LED off.");
122 led_msg_ptr->value = kobuki_msgs::Led::BLACK;
123 blink_publisher.publish(led_msg_ptr);
124 ground_obstacle = false;
125 }
126 }
```

- `go_straight(const double & time, int sign)` il corpo è costituito da un ciclo che itera per `time` secondi (riga 48). L'effetto è di settare i campi del messaggio `velocity` di tipo `Twist`, ponendo una velocità lineare positiva (mo-

vimento in avanti del robot) o negativa (movimento indietro del robot) e quella angolare a 0 (righe 51-52) . Infine pubblico il messaggio.

- `turn(const double & rotation_time, int sign)` speculare al metodo `go_straight` è presente un ciclo che itera per `rotation_time` secondi (riga 60). Si settano i campi del messaggio `velocity` di tipo `Twist` questa volta con velocità lineare nulla e angolare di $\pi/8$ rad/s con valore positivo (verso antiorario), negativo (verso orario) (righe 64-65). Infine pubblico il messaggio (riga 67).

- `bci_callback(const bci_to_ros::BCIMessage::ConstPtr& bci_)` controlla l'arrivo di messaggi `BCIMessage` che possono essere di stato (riga 74 `BCI_TYPE_STATE`) o di controllo (riga 92 `BCI_TYPE_CONTROL`). A sua volta a seconda del `value` del `BCIMessage` setto i flag `bci_control` e `bci_msg`. `bci_control` viene modificato solo con l'arrivo di un `BCIMessage` di stato: inizialmente assume il valore false, ma appena arriva un comando di `start` diventa true. Se invece il messaggio segnala un comando di `stop` o la perdita di connessione `bci_control` torna ad essere false.
  `bci_msg` segnala l'arrivo di un `BCIMessage` di controllo e il corrispondente `value` viene memorizzato all'interno della variabile `mov_val`. Quando si riceve un comando di `start` si setta nuovamente a false `bci_msg` per azzerare, nel caso fosse rimasto a true, la presenza di comandi di controllo (riga 78).

- `distances_callback( const basic_navigation::LaserDistances:: ConstPtr &last_dist)` all'arrivo di un messaggio di tipo `LaserDistances` ne memorizzo i valori delle distanze rilevate dal laser rispetto a tre angoli fissati, nell'array `actual_distances` (righe 101-103).

- `bumperEventCB (const kobuki_msgs::BumperEventConstPtr msg)` verifica la presenza di ostacoli a terra rilevati dal bumper. Inizialmente definisco un puntatore ad un messaggio di tipo `Led` (righe 109-110). Per capire lo stato del bumper (premuto o rilasciato) si deve valutare il campo `state` del messaggio di tipo `BumperEvent` descritto nel box seguente 3.7.

Listing 3.7: BumperEvent.msg

```
# Provides a bumper event.
# This message is generated whenever a
   particular bumper is pressed or released.
# Note that, despite bumper field on SensorState
   messages, state field is not a
# bitmask, but the new state of a single sensor.
# bumper
uint8 LEFT   = 0
uint8 CENTER = 1
uint8 RIGHT  = 2
# state
uint8 RELEASED = 0
uint8 PRESSED  = 1
```

```
uint8 bumper
uint8 state
```

Se il bumper è premuto `PRESSED` (riga 112) setto `value` del `Led.msg` a GREEN e ne pubblico il messaggio; inoltre il flag `ground_obstacle` assume valore true, perché si considera che ci sia un ostacolo (righe 112-118). Al contrario quando il bumper viene rilasciato, il `value` del `Led.msg` diventa BLACK e ne pubblico il messaggio; `ground_obstacle` ora assume il valore false perché non è più premuto (righe 119-125).

Il cuore dell'applicativo inizia alla riga 168 con il while che termina solamente digitando `Ctrl+c`.

```
168 while(ros::ok())
169 {
170 ros::spinOnce();
171 //controllo che ci sia un messaggio BCIMessage di
      controllo
172 if(bci_control)
173 {
174 vel_pub.publish(velocity);
175
176 if(bci_msg)
177 {
178 //verifico la direzione e richiamo la funzione di
      rotazione
179 if(mov_val==LEFT)
180 {
181 ROS_INFO("Next direction: <---\n");
182 turn(STANDARD_ROTATION,1);  //in senso antiorario
183 }
184 else if(mov_val==RIGHT)
185 {
186 ROS_INFO("Next direction: --->\n");
187 turn(STANDARD_ROTATION,-1); //in senso orario
188 }
189 bci_msg=false;  //ho processato il messaggio quindi
      setto a false bci_msg
190 velocity.linear.x = linear; //risetto il moto sull'asse
      x di velocità linear
191 velocity.angular.z = 0; // e pongo a 0 la velocità
      angolare sull'asse z
192 vel_pub.publish(velocity);
193 ros::spinOnce();
194 }
195 else
196 {
```

```
197
198 /*il laser verifica la distanza sui tre assi e se è
      minore di metà della minima distanza possibile (1 e
      3) o della minima distanza possibile (2) stabilisce
      la presenza di un ostacolo
199 */
200 if(actual_distances[0] < min_distance / 2 ||
201 actual_distances[1] < min_distance ||
202 actual_distances[2] < min_distance / 2)
203 {
204 std::cout << "Obstacle Detected!" << std::endl;
205
206 double rotation_time = rand_float(3, 7); //numero
      casuale tra 3 e 7, 4.5 corrisponde a 90°
207 std::cout << "rotation_time " << rotation_time <<
      std::endl;
208 turn(rotation_time,-1); //ruota in senso orario
209
210 velocity.linear.x = linear; //risetto il moto sull'asse
      x di velocità linear
211 velocity.angular.z = 0;    // e pongo a 0 la velocità
      angolare sull'asse z
212 vel_pub.publish(velocity);
213 ros::spinOnce();
214 }
215
216 //verifica la presenza di ostacoli bassi al livello del
      terreno che vengono rilevati dal bumper
217 if (ground_obstacle)
218 {
219 std::cout << "Obstacle on the ground!" << std::endl;
220 double reverse_time = 2;
221 go_straight(reverse_time,-1);// va indietro per due
      secondi
222
223 double rotation_time = rand_float(3, 7); //numero
      casuale tra 3 e 7, 4.5 corrisponde a 90°
224 std::cout << "rotation_time " << rotation_time <<
      std::endl;
225 turn(rotation_time,-1); //ruota in senso orario
226
227 velocity.linear.x = linear; //risetto il moto sull'asse
      x di velocità linear
228 velocity.angular.z = 0;     // e pongo a 0 la velocità
      angolare sull'asse z
229 vel_pub.publish(velocity);
```

```
230 ros::spinOnce();
231 }
232 }
233 }
234
235 ros::spinOnce();
236 ros::Duration(0.05).sleep();    // sleep per 50 ms in
      modo di evitare di pubblicare troppi messaggi ad ogni
      ciclo
237
238 }
```

Inizialmente richiamo `ros::spinOnce()` per eseguire tutte le `callback` e quindi verificare la presenza di messaggi sui diversi `topic`.
La priorità tra i messaggi è per quelli di tipo `BCIMessage` quindi alla riga 172 verifico il flag `bci_control`. Se assume il valore false significa che non è ancora stato inviato il comando di `start` dall'utente e quindi non succede niente. Invece in caso contrario, a sua volta verifico il flag `bci_msg` (riga 176), che viene settato a true appena si riceve un messaggio `BCIMessage` di controllo. La direzione scelta dall'utente può essere sinistra (righe 179-183) e quindi ruoto, tramite la funzione `turn`, in senso antiorario oppure destra in senso orario (righe 184-188). In entrambi i casi la rotazione avviene di un angolo di circa 45°, perciò dal momento che il modulo della velocità angolare di rotazione è di $\pi/8$ rad/s, il tempo necessario per ruotare (`STANDARD_ROTATION`) sarebbe di 2 s. Tuttavia, dopo una verifica sperimentale si è scelto di assegnare a `STANDARD_ROTATION` il valore 2.25 s.
Una volta processato il messaggio setto nuovamente a false il flag `bci_msg`, in attesa del prossimo messaggio di controllo.
Il robot, terminata la rotazione, deve continuare il suo moto con velocità lineare costante e quindi ridefinisco i campi del messaggio `velocity` (riga 190-191) e poi lo pubblico (riga 192). Se invece, `bci_control` assume il valore true, ma `bci_msg` è false; significa che l'utente ha inviato il comando di `start` però non ha mandato messaggi `BCIMessage` di controllo. In questo caso il robot si muove sempre dritto e può variare il suo moto solo con l'arrivo di messaggi dal laser e dal bumper.
Per motivi sperimentali, nell'ampio campo visivo del laser (240°) si è deciso di fissare tre angoli, rispetto ai quali il laser ne calcola la distanza, vedi Figura 3.5. Tutti e tre devono trovarsi di fronte al robot, infatti ai fini dell'esperimento devono permettere di rilevare degli ostacoli durante il suo avanzamento. Quindi si verificano le condizioni:

$$distanza1 < distanza\_minima/2$$

$$distanza2 < distanza\_minima$$

$$distanza3 < distanza\_minima/2$$

Affinché un ostacolo sia rilevato tutte e tre devono essere false (righe 200-202). Si ruota quindi tramite la funzione `turn` di un tempo casuale (riga 206) in senso orario (riga 208). Quando il bumper rileva un ostacolo a terra il flag `ground_obstacle` viene settato a true (riga 217). Il robot indietreggia per due secondi (riga 221),

Figura 3.5: Gli angoli fissati all'interno del campo visivo di 240° sono delimitati dalle semirette blu. Si verifica che la distanza 1 e 3 sono minori della metà della minima distanza e la distanza 2 è minore della minima distanza. Per rappresentare le distanze ho usato dei trattini rossi.

richiamando la funzione `go_straight`. A seguito il robot ruota tramite la funzione `turn` di un tempo casuale (riga 223) in senso orario (riga 225).
In entrambi i casi, terminata la rotazione, il robot deve continuare il suo moto con velocità lineare costante e quindi ridefinisco i campi del messaggio `velocity` (righe 210-211, 227-228) e poi lo pubblico (righe 212, 229). Infine alle riga 236 sospendo per 50 ms in modo da evitare la pubblicazione di troppi messaggi ad ogni ciclo.

3.4 Analisi del tester di connessione

Il **tester di connessione** è il terzo nodo ROS privato, quindi come i precedenti viene inizializzato alla riga 17 e ha come ultimo parametro di `ros::init connection`.

```
21 ros::Publisher
       bci_chatter=nh.advertise<bci_to_ros::BCIMessage>
22 ("/bci_message",1); //definizione del publisher che
       pubblica su topic "/bci_message"
```

Alla riga 21 ho definito il `publisher bci_chatter` che pubblica messaggi di tipo `BCIMessage` su `topic` ‘‘/bci_message’’.

```
26 //setto i parametri che vengono definiti nel launch
       corrispondente
27 nh.getParam("bci_type_state",bci_state);
28 std::cout<<"bci_type_state : "<<bci_state<<std::endl;
29 nh.getParam("dropped_connection", dropped_connection_c);
30 std::cout<<"dropped_connection :
       "<<dropped_connection_c<<std::endl;
31 nh.getParam("address",address);
32 std::cout<<"address : "<<address<<std::endl;
```

Il passaggio dei parametri avviene mediante `launch` riportato nel box seguente 3.8 e poi acquisiti nel main tramite la funzione `getParam` (righe 27-32):
- `bci_state` codifica da assegnare al `type` del `BCIMessage` quando cade la connessione
`dropped_connection_c` codifica da assegnare al `value` del `BCIMessage` quando cade la connessione
- `address` rappresenta l'indirizzo ip del computer di cui voglio verificare la connettività.

Listing 3.8: porzione launch del tester di connessione

```
<launch>
<arg name="bci_type_state" default="0"/>
<arg name="dropped_connection" default="3"/>
<arg name="address" default="192.168.202.231"/>
```

```
34 while(ros::ok())
35 {
36 bci_to_ros::BCIMessage msg; //creo messaggio di tipo
     BCIMessage
37 //in ping_command ne scrivo la corrispondente stringa
     che costituisce il comando
38 sprintf(ping_command, "ping -c 1 -W 3 %s | grep 'time='
     | wc -l",address.c_str());
39 FILE * ptr_out = popen(ping_command,"r"); //invoca la
     shell e richiama il comando di ping, restituisce un
     standard  I/O stream
40
41 char * ptr_buff = fgets(buffer, sizeof(buffer),
     ptr_out); //legge dallo stream e ne memorizza il
     contenuto in buffer, quando
42 //l'operazione avviene con successo fgets restituisce la
     stringa letta
43 pclose(ptr_out);
44
45 result = atoi(ptr_buff); //converto la stringa in
     intero, con le opzioni scelte result contiene 0 o 1
46
47 if (result == 0) //ovvero non c'è più connessione, setto
     i corrispondenti campi del messaggio BCIMessage
48 {
49 ROS_INFO("Connection dropped");
50 msg.type=bci_state;
51 msg.value=dropped_connection_c;
52 }
53 bci_chatter.publish(msg);   //ora pubblico il messaggio
     precedentemente dichiarato
54 ros::spinOnce();
```

```
55 ros::Duration(1).sleep();   //sleep per 1 s per evitare
     di pubblicare troppi messaggi ad ogni ciclo
56 }
57 ros::shutdown();
```

Alla riga 34 c'è un while che continua fino a quando l'utente non preme `Ctrl+ c` (riga 57): ad ogni iterazione invoca la shell, richiamando, tramite funzione `popen`(riga 39), il comando di `ping` memorizzato in `ping_command` (riga 38). Il `ping` invia diversi pacchetti ICMP all'indirizzo specificato, rispettando le opzioni aggiuntive:

- -c 1 si ferma dopo l'invio di un pacchetto `ECHO_REQUEST`, il `ping` attende 1 pacchetto `ECHO_REQUEST` fino alla scadenza del timeout.
- -W 3 specifica quanti secondi aspettare. Questa opzione riguarda il timeout in assenza di tutte le risposte, altrimenti il `ping` attende per due RTTs.
- `| grep 'time=' | wc -l` ricerca nell'output del `ping` la parola time e conta il numero di righe che la contengono.

Dal momento che nel mio caso invio un solo pacchetto, quando richiamo la funzione `popen` restituisce sotto forma di standard I/O stream 0 o 1, che corrispondono alle righe con la parola `time` nel return del `ping`. Tale output a sua volta viene memorizzato in un puntatore a FILE `ptr_out` (riga 39). Lo stream viene letto tramite la funzione `fgets`; se l'operazione avviene con successo `fgets` restituisce la stringa letta contenuta in `ptr_buff` (riga 41). Dal momento che `ptr_buff` è un puntatore a char, converto l'output in intero (come detto prima può essere 0 o 1) e lo memorizzo in `result` (riga 45). Se `result` assume il valore 0 allora la connessione è caduta, perché il `ping` non ha inviato o non ha ricevuto alcun pacchetto. In questo caso setto come precisato prima i campi `type` e `value` del `BCIMessage` (righe 47-52).
Infine alla riga 53 pubblico il messaggio `msg` di tipo `BCIMessage`. In questo caso chiamare la funzione `ros::spinOnce` alla riga 54 non sarebbe necessaria perché non sono presenti `callback`. Infine alla riga 55 sospendo per 1 s in modo che il **tester di connessione** non pubblichi troppi `BCIMessage` ad ogni ciclo.

Capitolo 4

Esperimento finale

In questo capitolo descrivo l'esperimento conclusivo del mio progetto, eseguito presso il laboratorio di Sistemi Autonomi Intelligenti IAS-Lab dell'Università di Padova. Inoltre propongo alcuni step successivi per migliorare ed espandere la mia applicazione.

4.1 Descrizione dell'esperimento

L'obiettivo dell'esperimento è stato testare la comunicazione tra una BCI di tipo non invasivo e un robot di telepresenza. In particolare tale esperimento, nel mio percorso sperimentale, mi ha permesso di verificare, dal punto di vista applicativo, il software sviluppato.
Nello specifico l'utente è doveva essere in grado, una volta inviato il comando di `start`, di controllare e di guidare il robot mediante dei *task* di immaginazione motoria. Inoltre, secondo la mia implementazione il robot doveva riconoscere eventuali ostacoli, quindi evitarli, ma non ha avuto la possibilità di decidere autonomamente la direzione verso cui muoversi. Infatti solamente il soggetto ha avuto il pieno controllo nella guida del robot nell'ambiente circostante. Tuttavia per maggior sicurezza, in caso di perdita di connessione con la BCI il robot ha doveva fermarsi.
Dal punto di vista metodologico, ho utilizzato una BCI basata sul *Motor Imagery (MI)*, perciò all'utente è stato richiesto, per controllare il robot, di immaginare mentalmente di muovere la mano destra e sinistra o entrambi i piedi e contemporaneamente è stata registrata la sua attività cerebrale attraverso i 16 canali EEG. Poi, a seguito di una fase di *preprocessing* e di estrazione delle *feature*, i *task* mentali sono stati tradotti rispettivamente nei comandi "gira a sinistra" e "gira a destra". La determinazione della classe corrispondente (una verso sinistra, l'altra verso destra) è avvenuta mediante dei classificatori bayesiani. Infine secondo l'architettura della BCI descritta al capitolo 2, l'intenzione del soggetto è stata manifestata tramite un *feedback* visivo. Due barre, una viola a sinistra e una azzurra a destra in Figura 4.1, a seconda della variazione di colorazione, hanno tradotto lo stato mentale dell'utente. Appena hanno raggiunto una certa soglia di decisione, un altro *feedback* discreto è stato fornito in output dalla BCI: la corrispettiva barra si è riempita completamente, con lo scopo di indicare che è stato rilevato un comando. Oltre alla BCI, ho utilizzato un robot di telepresenza descritto in Sezione 2.3. Come specificato in tale

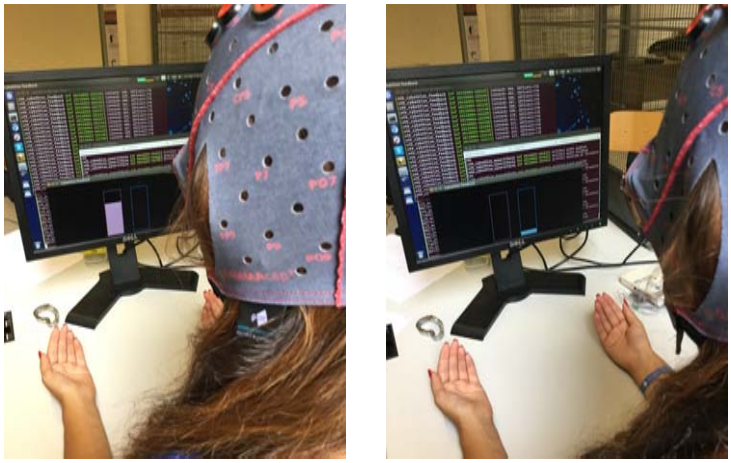

Figura 4.1: Feedback: due barre, a sinistra (viola) e a destra (azzurra), si sono colorate mentre il soggetto ha immaginato il corrispondente *task* mentale. Appena si è raggiunta una certa soglia di decisione, una delle due si è riempita completamente per indicare che è stato rilevato un comando.

Figura 4.2: Il robot è stato dotato di telecamera, lo stream è stato visto dall'utente direttamente sul suo computer. Inoltre ha potuto dare un comando "gira a sinistra" o "gira a destra" e ha ricevuto il *feedback*.

Sezione, sopra al robot ho posizionato una webcam che ne ha ripreso l'avanzamento e i movimenti nell'ambiente circostante. L'utente contemporaneamente vedeva lo stream sul suo computer tramite collegamento video e ha potuto volontariamente scegliere una tra le due classi di comandi, come si può vedere in Figura 4.2.
Il sistema ha implementato la seguente navigazione. Il robot si muove in automatico, solo dopo aver ricevuto lo `start`, ad una velocità costante ed evita gli ostacoli quando è necessario. Non appena riceve un comando mentale dell'utente determinato dalla BCI, ruota rispettivamente a destra (senso orario) o a sinistra (senso antiorario) di un angolo circa di 45°. Quando il laser segnala la presenza di un ostacolo ruota in senso orario di un angolo casuale e poi riprende il moto con velocità lineare costante. Invece nel caso di ostacoli bassi, rilevati dalla pressione e dal rilascio del bumper, il robot indietreggia e ruota di un angolo casuale per poi muoversi dritto con la velocità precedente e costante. Appena l'utente è stanco oppure vuole fermare il robot è sufficiente che invii il comando di `stop`.
In caso in cui si ricevano due comandi simultanei dovuti uno al riconoscimento di

un ostacolo e l'altro proveniente dalla BCI, ha precedenza quello scelto dal soggetto. Infatti appena arriva un comando che contiene la decisione dell'utente, il robot interrompe la navigazione in automatico.
L'obiettivo della mia tesi è stato quello di teleoperare personalmente (visibile in Figura 4.3), attraverso l'uso della BCI, il robot. Pertanto, poiché il classificatore della BCI non era stato allenato sulle mie onde cerebrali, precedentemente alla prova, ho sostenuto una fase di *training* (visibile in Figura 4.4), della durata di 8 minuti in 4 sessioni separate da pause di 2 minuti.

Figura 4.3: Soggetto dell'esperimento: io stessa mentre sto indossando la caratteristica cuffia con già integrati gli elettrodi per registrare l'attività cerebrale.

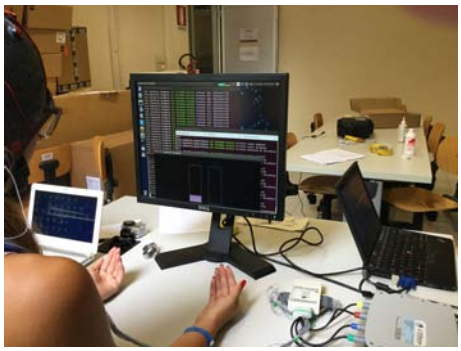

Figura 4.4: Fase di training: ho dovuto immaginare di aprire e chiudere la mano destra e sinistra, ma senza compiere effettivamente il movimento. Dopo aver analizzato la mia attività cerebrale, processato i segnali e successivamente classificati, è stata verificata la correttezza del modello: mediante l'immaginazione del movimento richiesto dovevo inviare il comando corretto.

L'esperimento si è svolto in un ambiente di laboratorio, un naturale spazio di lavoro con dei corridoi e tre stanze diverse. In Figura 4.5 è visibile una di queste stanze. Lo spazio conteneva ostacoli come tavoli, sedie, armadi, persone. Il soggetto è seduto di fronte a un monitor all'interno di una stanza e nell'ambiente circostante si trovava il robot. Durante la prova il soggetto ha guidato il robot attraverso un percorso scelto da lui.
Per verificare la fattibilità dell'esperimento inizialmente ho fatto dei test in cui ho guidato il robot manualmente, ovvero fornendo i comandi da tastiera (freccia sinistra e destra).

Figura 4.5: Robot che si muoveva nell'ambiente circostante. Sono presenti diversi ostacoli come tavoli, sedie, armadi, persone.

4.2 Risultati

L'integrazione tra BCI e il robot di telepresenza è avvenuta con successo. Il robot è stato in grado di ricevere i comandi dell'utente determinati dalla BCI.
Una volta che il cursore si colorava completamente il comando è stato inviato immediatamente al robot, che a sua volta lo ha implementato.
Correttamente e tempestivamente il robot ha iniziato a muoversi dopo l'invio di uno `start` e si è fermato in seguito a uno `stop`.
La verifica della connessione tra BCI e il computer posizionato sopra al robot è stata utile a bloccare il robot stesso nel caso effettivamente vi fosse stata assenza di rete.
Inoltre appena il robot incontrava un ostacolo durante la navigazione, per esempio il muro o le sedie, ha ruotato di 45° e ha proseguito dritto in quella direzione. Si è comportato in maniera analoga, dopo aver indietreggiato, quando il bumper è stato premuto e rilasciato, come nel caso in cui ha incontrato le gambe dei tavoli.

4.3 Conclusioni e lavoro futuro

Con l'implementazione e il test di tutto il codice sviluppato, si è dimostrato la possibilità di controllare un robot di telepresenza via Brain-Computer Interface.
Uno dei risultati fondamentali è stato quello di riuscire a far comunicare la BCI con il robot. In particolare grazie al **ricevitore BCI**, essa è in grado di trasmettere al robot i comandi inviati dall'utente e a sua volta vengono differenziati a seconda del tipo (`stato` e `controllo`). Poi il robot si comporta di conseguenza, traducendo gli input ricevuti in azioni "gira a sinistra", "gira a destra", "avanza" e "fermati".
Come riportato nell'introduzione il mio obiettivo principale era quello di riuscire a integrare una BCI non invasiva con un robot di telepresenza, quindi mi sono occupata principalmente dell'invio, ricezione e codifica dei comandi inviati dall'utente. Perciò per quanto riguarda il movimento del robot, il **software di navigazione** presenta alcuni aspetti che dovrebbero essere perfezionati, da quanto è emerso duranti gli esperimenti. La guida del robot con comandi da tastiera, ovvero senza l'uso della BCI, è riuscita in un tempo sufficientemente limitato. Nel corso dell' espe-

rimento completo, in cui ho utilizzato la BCI per fornire i comandi di movimento in base alla mia attività cerebrale, la navigazione del robot ha mostrato dei limiti come impossibilità di determinare l'angolo di rotazione del robot in caso di ostacolo rilevato e di correggere questo angolo durante la fase di rotazione.
Il passo successivo del mio progetto sarebbe quello di rendere la navigazione semi-intelligente per facilitare il movimento del robot e velocizzare la sua capacità di evitare gli ostacoli.
Nella precisione il robot, una volta rilevato un ostacolo, non dovrebbe ruotare di un angolo casuale, ma essere in grado di calcolare quello corretto per assicurarsi il cammino di fronte a lui libero. Quindi bisognerebbe passare ad un sistema BCI *shared-control* che permette all'utente di focalizzare l'attenzione solo sulla destinazione finale del robot ignorando i problemi a basso livello relativi alla navigazione (evitare gli ostacoli). Al soggetto sembrerà di avere il pieno controllo, anche se il robot semi-intelligente lo aiuterebbe nelle situazioni problematiche. In questo modo tutte le persone potrebbero portare a termine la guida del robot, anche lungo percorsi complessi, in poco tempo e con meno comandi [11].
Inoltre nel caso in cui l'utente invii un comando che porterebbe il robot troppo vicino ai muri, quest'ultimo non dovrebbe eseguirlo.
Per realizzare queste soluzioni, è necessario sfruttare delle tecniche di odometria che consentono di stimare la posizione del robot e garantiscono delle performance migliori.
Un'altra osservazione riguarda la rotazione del robot, essa dovrebbe immediatamente interrompersi, senza concludersi per permettere in ogni istante all'utente di inviare comandi (e poi essere implementati) anche quando il robot sta girando. In questo modo si eviterebbe di perdere alcuni input che effettivamente vengono trasmessi dal soggetto e ricevuti dal **ricevitore BCI**, ma non sono eseguiti dal robot perché impegnato a ruotare.
Un altro aspetto positivo del mio progetto è il **tester di connessione** infatti dal punto di vista della sicurezza permette di bloccare il robot in assenza di rete. Si evita così che il robot si muova autonomamente nell'ambiente circostante senza essere controllato dalla BCI, aspetto fondamentale per applicazioni future.
Per quanto riguarda l'infrastruttura di telepresenza, il soggetto è stato in grado, tramite stream video, di osservare cosa c'era davanti al robot, anche se si trovava in posizioni differenti. Tuttavia, in una futura versione, integrerei il collegamento video con l'audio permettendo all'utente non solo di vedere amici e parenti in altre stanze, ma anche di comunicare con loro mediante un microfono. Ciò è supportato da Gstreamer, infatti è sufficiente aggiungere alla *pipeline* un ulteriore elemento che si occupa dell'audio.
In conclusione, soggetti con gravi disabilità, paralizzati e *locked-in* possono effettivamente sfruttare l'interfaccia BCI come canale di comunicazione alternativo. Infatti la BCI permette in diverse applicazioni un buon controllo, per esempio, come dimostra questa tesi, su un robot mobile.
È bene osservare che per ora si tratta di una tecnologia costosa, usata per lo più in laboratorio e che per un utilizzo quotidiano, a mio parere, richiederebbe delle modifiche. Per evitare che queste persone si sentano diverse, sarebbe opportuno creare delle cuffie, con elettrodi già integrati, più estetiche e fare in modo che non

solo computer ma anche smartphone, tablet e smartwatch possano essere collegati alla BCI.
Le possibilità di approfondire la ricerca e di proseguire tale progetto innovativo per l'Università di Padova sono quindi numerose.
Ciò che è importante sottolineare è che la BCI può costituire una chance per molte persone affette da gravi disabilità, con patologie che colpiscono il canale neuromuscolare e che spesso sono paralizzate e costrette a rimanere a letto.

Appendice A

Il trasmettitore BCI

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4 #include <arpa/inet.h>
5 #include <netdb.h>
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <unistd.h>
9 #include <string.h>
10 #include <unistd.h>
11 #include <fcntl.h>
12 #include <iostream>
13 #include <string>
14 #include <sstream>
15 #include <assert.h>
16 #include <unistd.h>
17 #include <curses.h>
18 #include <termios.h>
19
20 #define LEFT    11;
21 #define RIGHT   12;
22 #define EXIT_C  2;
23 #define START_C 1;
24
25 const int BUFLEN=1;
26
27 //costanti che indicano valore decimale dei tasti
     premuti dall'utente
28 const int LEFT_ARROW=68; //carattere freccia sinistra
29 const int RIGHT_ARROW=67; //carattere freccia destra
30 const int START=115; //carattere s
31 const int EXIT=113; //carattere q
32 const int PASS=27;  //esc
```

```
33 const int PASS1=91; //[
34
35 //funzione che restituisce un intero corrispondente al
     tasto premuto da tastiera
36 int get_key_pressed();
37
38 /* Funzione che viene richiamata quando si riscontra un
     errore e ne stampa il messaggio di errore passato per
     parametro, poi esce*/
39 void error(const char *);
40
41 int main(int argc, char *argv[])
42 {
43 int sock, n, count;
44 unsigned int length;
45 struct sockaddr_in sender;
46 bool waitcommand=true;   //variabile booleana che indica
     se sta aspettando che l'utente prema un tasto
47 bool quit = false;
48 char ch;
49 char buf[BUFLEN] = {0};  //buffer che memorizza i
     comandi che vengono inviati
50
51 if (argc != 3) {
52 std::cerr<<"Usage:"<<argv[0]<<" <host> <port> "
     <<std::endl;
53 exit(1);
54 }
55 count=0;
56 bzero(&sender,sizeof(sender));
57 //creazione del socket
58 sock= socket(AF_INET, SOCK_DGRAM, 0);
59 if (sock < 0) error("Socket failed");
60 sender.sin_family = AF_INET;
61 sender.sin_port = htons(atoi(argv[2]));
62 inet_pton(AF_INET,argv[1],&sender.sin_addr);
63 fcntl(sock,F_SETFL,O_NONBLOCK); //setto il socket di
     tipo non bloccante
64 length=sizeof(struct sockaddr_in);
65
66 while(!quit) {  //itera finchè l'utente non digita q
67 std::cout<<"Where do you want to go?"<<std::endl;
68 while(waitcommand) {
69 ch=get_key_pressed();   //memorizzo il carattere
     digitato dall'utente in ch e ne faccio il parsing
```

```
70 switch(ch){//appena ho il comando setto a false
     waitcommand perché non devo più attendere e lo
     memorizzo in buf
71 case START:
72 waitcommand=false;
73 buf[0]=START_C;
74 break;
75 case RIGHT_ARROW:
76 waitcommand=false;
77 buf[0]=RIGHT;
78 break;
79 case LEFT_ARROW:
80 waitcommand=false;
81 buf[0]=LEFT;
82 break;
83 case EXIT:
84 waitcommand=false;
85 buf[0]=EXIT_C;
86 quit = true;
87 break;
88 //i due pass servono per mangiare dei caratteri
     aggiuntivi quando l'utente preme freccia sinistra o
     destra
89 case PASS:
90 case PASS1:
91 break;
92 default:
93 std::cout<<"The command sent is wrong: you must use
94 only left or right arrows. To quite press q" <<
     std::endl;
95 std::cout<<"Where do you want to go?"<<std::endl;
96 break;
97 }
98 }
99 n=sendto(sock,buf,1,0,(const struct sockaddr
     *)&sender,length);   //invio i comandi al BCI_receiver
100 if (n < 0) {
101 error("Sendto failed");
102 } else {
103 count++;    //ho inviato un comando, aumento il conteggio
104 std::cout<<"-  "<<count<<" Datagram sent to
     BCI_receiver"<<std::endl;
105 }
106 waitcommand=true; //appena inviati i comandi risetto a
     true waitcommand in attesa del prossimo comando
107 }
```

```
108 close(sock);//chiusura del socket
109 return 0;
110 }
111
112 void error(const char *msg)
113 {
114 std::cerr<<msg;
115 exit(0);
116 }
117
118 int get_key_pressed() {
119 int c = 0;
120 struct termios org_opts, new_opts;
121 int res = 0;
122 //-----  store old settings -----------
123 res = tcgetattr(STDIN_FILENO, &org_opts);
124 assert(res == 0);
125 //---- set new terminal parms --------
126 memcpy(&new_opts, &org_opts, sizeof(new_opts));
127 new_opts.c_lflag &=
128 ~(ICANON | ECHO | ECHOE | ECHOK | ECHONL | ECHOPRT |
      ECHOKE |
129 ICRNL);
130 tcsetattr(STDIN_FILENO, TCSANOW, &new_opts);
131 c = getchar();
132 //------  restore old settings ---------
133 res = tcsetattr(STDIN_FILENO, TCSANOW, &org_opts);
134 assert(res == 0);
135 return c;
136 }
```

Appendice B

Il ricevitore BCI

```
1 #include <sys/types.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <string.h>
7 #include <netdb.h>
8 #include <stdio.h>
9 #include <iostream>
10 #include <std_msgs/String.h>
11 //include per ROS
12 #include <ros/ros.h>
13 #include <bci_to_ros/BCIMessage.h>
14 // include per abilitare Ctrl+ c
15 #include <csignal>
16 #include <sstream>
17
18 const int BLEN1=256;
19
20 /* Funzione che viene richiamata quando si riscontra un
     errore e ne stampa il messaggio di errore passato per
     parametro,
21 poi esce*/
22 void error(const char *msg)
23 {
24 std::cerr<<msg;
25 exit(0);
26 }
27
28 /* Funzione che viene richiamata quando l'utente preme
     Ctrl +c, fa terminare il programma dopo aver stampato
     exit*/
29 void signalHandler(int signum)
```

```
30 {
31 std::cout<<std::endl<<"Exit"<<std::endl;
32 exit(signum);
33 }
34
35 int main(int argc, char ** argv)
36 {
37 ros::init(argc,argv,"server_robot");    //inizializza le
     librerie ROS
38
39 ros::NodeHandle nh("~");  //punto di acceso per
     comunicare con il resto del sistema ROS
40
41 ros::Publisher
     bci_chatter=nh.advertise<bci_to_ros::BCIMessage>
42 ("/bci_message",1); //definizione del publisher che
     pubblica su topic "/bci_message"
43
44 ros::Rate loop_rate(10);    //specifica la frequenza di
     ciclo ovvero 10 Hz
45
46 signal(SIGINT, signalHandler);
47
48 int sock, length, n;
49 socklen_t sender_len;
50 struct sockaddr_in receiver;
51 struct sockaddr_in sender;
52 char buf1[BLEN1];
53 char *command;
54 int icommand;
55 int left_c,right_c,port,start_c,exit_c, bci_state,
     bci_control;
56
57 // setto i parametri che vengono definiti nel launch
     corrispondente
58 nh.getParam("port",port);
59 nh.getParam("start_c",start_c);
60 nh.getParam("exit_c",exit_c);
61 nh.getParam("left",left_c);
62 nh.getParam("right",right_c);
63 nh.getParam("bci_type_state",bci_state);
64 nh.getParam("bci_type_control",bci_control);
65
66 std::cout<<"port : "<<port<<std::endl;
67 std::cout<< "LEFT: " << left_c<<std::endl;
68 std::cout<< "RIGHT: " << right_c<<std::endl;
```

```
69 std::cout<<"START: " << start_c<<std::endl;
70 std::cout<<"EXIT: " << exit_c<<std::endl;
71 std::cout<<"bci_type_state : "<<bci_state<<std::endl;
72 std::cout<<"bci_type_control : "<<bci_control<<std::endl;
73
74 // creo socket, e assegno l'indirizzo tramite bind
75 sock=socket(AF_INET, SOCK_DGRAM, 0);
76 if (sock < 0) error("socket error");
77 length = sizeof(receiver);
78 bzero(&receiver,length);
79 receiver.sin_family=AF_INET;
80 receiver.sin_addr.s_addr=htonl(INADDR_ANY);
81 receiver.sin_port=htons(port);
82 if (bind(sock,(struct sockaddr *)&receiver,length)<0)
83 error("binding error");
84 sender_len = sizeof(struct sockaddr_in);
85 bzero(&buf1,BLEN1);
86 while (ros::ok()) {
87 //attendo un messaggio dal BCI sender o dalla BCI
88 n = recvfrom(sock,buf1,BLEN1,0,(struct sockaddr
     *)&sender,&sender_len);
89 if (n < 0) error("recvfrom");
90 else
91 {  //ho ricevuto un messaggio dal BCI sender o dalla BCI
92 icommand=static_cast<int>(command[0] & 0xFF);
93 bci_to_ros::BCIMessage msg; //creo messaggio di tipo
     BCIMessage
94 //ho ricevuto il messaggio, lo processo e setto i
     corrispondenti campi del messaggio BCIMessage
95 if(icommand == start_c) {
96 ROS_INFO("User sent start_command\n");
97 msg.type=bci_state;
98 msg.value=start_c;
99 } else if(icommand == exit_c) {
100 ROS_INFO("User sent exit_command\n");
101 msg.type=bci_state;
102 msg.value=exit_c;
103 } else if(icommand == left_c){
104 ROS_INFO("User sent left_command\n");
105 msg.type=bci_control;
106 msg.value=left_c;
107 }else if(icommand == right_c){
108 ROS_INFO("User sent right_command\n");
109 msg.type=bci_control;
110 msg.value=right_c;
111 }
```

```
112
113 bci_chatter.publish(msg);   //ora pubblico il messaggio
      precedentemente dichiarato
114
115 ros::spinOnce();
116
117 loop_rate.sleep();  //lo sospendo per evitare di
      pubblicare troppi messaggi ad ogni ciclo
118 }
119 bzero(&buf1,BLEN1);
120 }
121
122 ros::shutdown();
123 return 0;
124 }
```

Appendice C

Il software di navigazione

```
1 #include <tf/tf.h>
2 #include <iostream>
3 #include <fstream>
4 // include per ROS
5 #include <ros/ros.h>
6 #include <kobuki_msgs/BumperEvent.h>
7 #include <kobuki_msgs/Led.h>
8 #include <geometry_msgs/Twist.h>
9 #include <basic_navigation/LaserDistances.h>
10 #include<bci_to_ros/BCIMessage.h>
11 //include per le spiegazioni delle callback e delle
     funzioni richiamate
12 #include <basic_navigation.h>
13
14 #define EPSILON M_PI / 20
15 #define STANDARD_ROTATION 2.25  //corrisponde ad una
     rotazione di 45°
16 //comandi
17 #define LEFT 11
18 #define RIGHT 12
19 #define START 1
20 #define STOP 2
21 #define DROPPED_CONNECTION 3
22
23 #define BCI_TYPE_STATE      0
24 #define BCI_TYPE_CONTROL    1
25
26 geometry_msgs::Twist velocity;
27 ros::Publisher vel_pub;
28 ros::Publisher blink_publisher;
29
30 bool bci_msg = false;
31 int mov_val;
```

```
32
33 bool bci_control=false;
34
35 double linear = 0.2;
36 double angular = 0;
37
38 double actual_distances[3] = {5., 5., 5.};
39 bool ground_obstacle = false;
40
41 double rand_float( double low, double high ) {
42 return ( ( double )rand() * ( high - low ) ) / ( double
     )RAND_MAX + low;
43 }
44
45 void go_straight(const double & time, int sign)
46 {
47 ros::Time now = ros::Time::now();
48 while((ros::Time::now() - now).sec < time )
49 {
50 //ROS_INFO_STREAM("actual - now: " << ros::Time::now() -
     now);
51 velocity.linear.x = sign * linear;
52 velocity.angular.z = 0;
53 vel_pub.publish(velocity);
54 ros::spinOnce();
55 }
56 }
57
58 void turn(const double & rotation_time, int sign) // 4.5
     seconds, 90°
59 {
60 ros::Time now = ros::Time::now();
61 while((ros::Time::now() - now).sec < rotation_time )
62 {
63 //ROS_INFO_STREAM("actual - now: " << ros::Time::now() -
     now);
64 velocity.linear.x = 0;
65 velocity.angular.z = sign * M_PI / 8;
66 vel_pub.publish(velocity);
67 ros::spinOnce();
68 }
69 }
70
71 void bci_callback(const
     bci_to_ros::BCIMessage::ConstPtr& bci_)
72 {
```

```
73
74 if(bci_->type==BCI_TYPE_STATE) {
75 switch(bci_->value) {
76 case START:
77 ROS_INFO("[StateMsg] - Robot has BCI control");
78 bci_control=true; bci_msg=false;
79 break;
80 case STOP:
81 ROS_INFO("[StateMsg] - pause-Robot stops");
82 bci_control=false;
83 break;
84 case DROPPED_CONNECTION:
85 ROS_INFO("[StateMsg] - connecion-Robot stops");
86 bci_control=false;
87 break;
88 default:
89 break;
90 }
91
92 } else if (bci_->type==BCI_TYPE_CONTROL) {
93 ROS_INFO_STREAM("[ControlMsg] - Received BCI_message: "
     << bci_->value);
94 mov_val = bci_->value;
95 bci_msg=true;      //set local state
96 }
97 }
98
99 void distances_callback(const
     basic_navigation::LaserDistances::ConstPtr &las_dist)
100 {
101 actual_distances[0] = las_dist->distances[0];
102 actual_distances[1] = las_dist->distances[1];
103 actual_distances[2] = las_dist->distances[2];
104 }
105
106 void bumperEventCB(const
     kobuki_msgs::BumperEventConstPtr msg)
107 {
108 // Preparing LED message
109 kobuki_msgs::LedPtr led_msg_ptr;
110 led_msg_ptr.reset(new kobuki_msgs::Led());
111
112 if (msg->state == kobuki_msgs::BumperEvent::PRESSED)
113 {
114 ROS_INFO_STREAM("Bumper pressed. Turning LED on.");
115 led_msg_ptr->value = kobuki_msgs::Led::GREEN;
```

```
116 blink_publisher.publish(led_msg_ptr);
117 ground_obstacle = true;
118 }
119 else // kobuki_msgs::BumperEvent::RELEASED
120 {
121 ROS_INFO_STREAM("Bumper released. Turning LED off.");
122 led_msg_ptr->value = kobuki_msgs::Led::BLACK;
123 blink_publisher.publish(led_msg_ptr);
124 ground_obstacle = false;
125 }
126 }
127
128 int main(int argc,char** argv)
129 {
130 ros::init(argc,argv,"Navigation_node"); //inizializza le
     librerie ROS
131
132 ros::NodeHandle p; //punto di acceso per comunicare con
     il resto del sistema ROS
133 ros::NodeHandle nh("~"); //punto di acceso per
     comunicare con il resto del sistema ROS
134
135 srand (time(NULL));
136
137 double min_distance;
138
139 //setto i parametri che vengono definiti nel launch
     corrispondente oppure valore di default equivale al
     terzo parametro
140 nh.param<double>("linear", linear, 0);
141 nh.param<double>("angular", angular, 0);
142 nh.param<double>("min_dist", min_distance, 0.5);
143
144 velocity.angular.z = angular;
145 velocity.angular.y = 0;
146 velocity.angular.x = 0;
147 velocity.linear.x = linear;
148 velocity.linear.y = 0;
149 velocity.linear.z = 0;
150
151 std::cout << "EPSILON: " << EPSILON << std::endl;
152
153 //Publishers
154 vel_pub = p.advertise<geometry_msgs::Twist
155 ("/cmd_vel_mux/input/navi",1);
```

```
156 blink_publisher = p.advertise< kobuki_msgs::Led
    >("commands/led1", 10);
157
158 //Subscribers: uno per il laser, uno per il bumper e uno
    per la ricezione di messaggi dalla BCI
159 ros::Subscriber dist_sub =
    p.subscribe<basic_navigation::LaserDistances>
160 ("/distances", 1, distances_callback);
161 ros::Subscriber bumper_event_sub =
    p.subscribe("/mobile_base/events/bumper", 10,
    bumperEventCB);
162 ros::Subscriber
    bci_sub=p.subscribe<bci_to_ros::BCIMessage>
163 ("/bci_message",1,bci_callback);
164
165 //Per essere sicura che il laser sia attivo, sleep per 4
    secondi
166 ros::Duration(4).sleep();
167
168 while(ros::ok())
169 {
170 ros::spinOnce();
171 //controllo che ci sia un messaggio BCIMessage di
    controllo
172 if(bci_control)
173 {
174 vel_pub.publish(velocity);
175
176 if(bci_msg)
177 {
178 //verifico la direzione e richiamo la funzione di
    rotazione
179 if(mov_val==LEFT)
180 {
181 ROS_INFO("Next direction: <---\n");
182 turn(STANDARD_ROTATION,1);  //in senso antiorario
183 }
184 else if(mov_val==RIGHT)
185 {
186 ROS_INFO("Next direction: --->\n");
187 turn(STANDARD_ROTATION,-1); //in senso orario
188 }
189 bci_msg=false;  //ho processato il messaggio quindi
    setto a false bci_msg
190 velocity.linear.x = linear; //risetto il moto sull'asse
    x di velocità linear
```

```
191 velocity.angular.z = 0; // e pongo a 0 la velocità
      angolare sull'asse z
192 vel_pub.publish(velocity);
193 ros::spinOnce();
194 }
195 else
196 {
197
198 /*il laser verifica la distanza sui tre assi e se è
      minore di metà della minima distanza possibile (1 e
      3) o della minima distanza possibile (2) stabilisce
      la presenza di un ostacolo
199 */
200 if(actual_distances[0] < min_distance / 2 ||
201 actual_distances[1] < min_distance ||
202 actual_distances[2] < min_distance / 2)
203 {
204 std::cout << "Obstacle Detected!" << std::endl;
205
206 double rotation_time = rand_float(3, 7); //numero
      casuale tra 3 e 7, 4.5 corrisponde a 90°
207 std::cout << "rotation_time " << rotation_time <<
      std::endl;
208 turn(rotation_time,-1); //ruota in senso orario
209
210 velocity.linear.x = linear; //risetto il moto sull'asse
      x di velocità linear
211 velocity.angular.z = 0;    // e pongo a 0 la velocità
      angolare sull'asse z
212 vel_pub.publish(velocity);
213 ros::spinOnce();
214 }
215
216 //verifica la presenza di ostacoli bassi al livello del
      terreno che vengono rilevati dal bumper
217 if (ground_obstacle)
218 {
219 std::cout << "Obstacle on the ground!" << std::endl;
220 double reverse_time = 2;
221 go_straight(reverse_time,-1);// va indietro per due
      secondi
222
223 double rotation_time = rand_float(3, 7); //numero
      casuale tra 3 e 7, 4.5 corrisponde a 90°
224 std::cout << "rotation_time " << rotation_time <<
      std::endl;
```

```
225 turn(rotation_time,-1); //ruota in senso orario
226
227 velocity.linear.x = linear; //risetto il moto sull'asse
      x di velocità linear
228 velocity.angular.z = 0;     // e pongo a 0 la velocità
      angolare sull'asse z
229 vel_pub.publish(velocity);
230 ros::spinOnce();
231 }
232 }
233 }
234
235 ros::spinOnce();
236 ros::Duration(0.05).sleep();    // sleep per 50 ms in
      modo di evitare di pubblicare troppi messaggi ad ogni
      ciclo
237
238 }
239
240
241 ROS_INFO_STREAM("CTRL+C for shutdown...");
242 ros::waitForShutdown();
243
244 return 0;
245 }
```

Appendice D

Il tester di connessione

```
1 #include <stdio.h>
2 #include <iostream>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <fstream>
6 #include <unistd.h>
7 //include per ROS
8 #include <ros/ros.h>
9 #include <bci_to_ros/BCIMessage.h>
10
11 int main(int argc, char ** argv)
12 {
13 int dropped_connection_c,bci_state,result;
14 std::string address;
15 char ping_command[70];
16 char buffer[1024];
17 ros::init(argc,argv,"connection");  //inizializza le
     librerie ROS
18
19 ros::NodeHandle nh("~"); //punto di acceso per
     comunicare con il resto del sistema ROS
20
21 ros::Publisher
     bci_chatter=nh.advertise<bci_to_ros::BCIMessage>
22 ("/bci_message",1); //definizione del publisher che
     pubblica su topic "/bci_message"
23
24 ros::Rate loop_rate(10);    //specifica la frequenza di
     ciclo ovvero 10 Hz
25
26 //setto i parametri che vengono definiti nel launch
     corrispondente
27 nh.getParam("bci_type_state",bci_state);
```

```
28 std::cout<<"bci_type_state : "<<bci_state<<std::endl;
29 nh.getParam("dropped_connection", dropped_connection_c);
30 std::cout<<"dropped_connection :
     "<<dropped_connection_c<<std::endl;
31 nh.getParam("address",address);
32 std::cout<<"address : "<<address<<std::endl;
33
34 while(ros::ok())
35 {
36 bci_to_ros::BCIMessage msg; //creo messaggio di tipo
     BCIMessage
37 //in ping_command ne scrivo la corrispondente stringa
     che costituisce il comando
38 sprintf(ping_command, "ping -c 1 -W 3 %s | grep 'time='
     | wc -l",address.c_str());
39 FILE * ptr_out = popen(ping_command,"r"); //invoca la
     shell e richiama il comando di ping, restituisce un
     standard  I/O stream
40
41 char * ptr_buff = fgets(buffer, sizeof(buffer),
     ptr_out); //legge dallo stream e ne memorizza il
     contenuto in buffer, quando
42 //l'operazione avviene con successo fgets restituisce la
     stringa letta
43 pclose(ptr_out);
44
45 result = atoi(ptr_buff); //converto la stringa in
     intero, con le opzioni scelte result contiene 0 o 1
46
47 if (result == 0) //ovvero non c'è più connessione, setto
     i corrispondenti campi del messaggio BCIMessage
48 {
49 ROS_INFO("Connection dropped");
50 msg.type=bci_state;
51 msg.value=dropped_connection_c;
52 }
53 bci_chatter.publish(msg);   //ora pubblico il messaggio
     precedentemente dichiarato
54 ros::spinOnce();
55 ros::Duration(1).sleep();   //sleep per 1 s per evitare
     di pubblicare troppi messaggi ad ogni ciclo
56 }
57 ros::shutdown();
58 return 0;
59 }
```

Bibliografia

[1] Jonathan R. Wolpaw, Niels Birbaumer, Dennis J. McFarland, Gert Pfurtscheller, Theresa M. Vaughan, Brain-computer interfaces for communication and control, *Clinical Neurophysiology* , vol. 113, pp. 767-791, 2002.

[2] Gernot R. Müller-Putz, Reinhold Scherer, Gert Pfurtscheller and Rüdiger Rupp, Brain-computer interfaces for control of neuroprostheses: from synchronous to asynchronous mode of operation, *Biomedizinische Technik*, vol. 51, no. 2, pp. 57-63, 2006.

[3] José del R. Millán, Pierre W. Ferrez, Ferran Galán, Eileen Lew, Ricardo Chavarriaga, Non-Invasive Brain-Actuated Interaction,*Advances in Brain, Vision, and Artificial Intelligence*, pp. 438-447. LCNS 4729, Springer, 2007.

[4] Dennis J. McFarland and Jonathan R. Wolpaw, Brain-Computer Interface Operation of Robotic and Prosthetic Devices,*Computer 41*, 10, pp.48-52, 2008.

[5] J. d. R. Millán, R. Rupp, G. R. Müller-Putz, R. Murray-Smith, C.Giugliemma, M. Tangermann, C. Vidaurre, F. Cincotti, A. Kübler, R. Leeb, C. Neuper, K.-R. Müller and D. Mattia, Combining brain-computer interfaces and assistive technologies: state-of-the-art and challenges, *Front. Neurosci.* 4: 161.10.3389/fnins.2010.00161, 2010.

[6] http://www.brainfactor.it/?p=1852

[7] Michael Bensch, Ahmed A. Karim, Jürgen Mellinger, Thilo Hinterberger, Michael Tangermann, Martin Bogdan, Wolfgang Rosenstiel, and Niels Birbaumer, Nessi: An EEG-Controlled Web Browser for Severely Paralyzed Patients, *Comput. Intell. Neurosci.*, 5, Article ID 71863. doi:10.1155/2007/71863.10.1155/2007/71863, 2007.

[8] Adil Mehmood Khan,Brain-Computer Interfaces for Communication in Paralysis: A Clinical Experimental Approach.ppt

[9] Gernot R. Müller-Putz, Christian Breitwieser, Febo Cincotti, Robert Leeb, Martijn Schreuder, Francesco Leotta, Michele Tavella, Luigi Bianchi, Alex Kreilinger , Andrew Ramsay, Martin Rohm, Max Sagebaum, Luca Tonin, Christa Neuper and José del. R. Millán, Tools for brain-computer interaction: a general concept for a hybrid BCI, *Frontiers Neuroinformatics* 5: doi.10.3389/fnins.2011.00039, 2011.

[10] E. Rocon, J.A. Gallego, L. Barrios, A.R. Victoria, J. Ibánez, D. Farina, F. Negro, J.L. Dideriksen, S. Conforto T. D'Alessio, G. Severini, J.M. Belda-Lois, L.Z. Popovic, G Grimaldi, M. Manto, J.L. Pons, Multimodal BCI-Mediated FES Suppression of Pathological Tremor, *Engineering in Medicine and Biology Society (EMBC)*, Annual International Conference of the IEEE, pp. 337-3340, 2010.

[11] Luca Tonin, Robert Leeb, Michele Tavella, Serafeim Perdikis, José del R. Millán, The role of shared-control in BCI-based telepresence, *IEEE International Conference on Systems, Man, and Cybernetics*, 2010.

[12] http://polymorph.units.it/what_is_polymorph/#/state-of-the-art

[13] http://neurogadget.com/2012/03/05/cebit-bci-demo-intendix-soci-lets-you-control-pc-games-with-98-accuracy/3679

[14] Robert Leeb, Serafeim Perdikis, Luca Tonin, Andrea Biasiucci, Michele Tavella, Marco Creatura, Alberto Molina, Abdul Al-Khodairy, Tom Carlson, José d.R. Millán, Transferring brain-computer interfaces beyond the laboratory:Successful application control for motor-disabled users,*Artif Intell Med.*, 59:121-32. doi:10.1016/j.artmed.2013.08.004, 2013.

[15] Andrea Cinetto, Brain-Computer Interface basata su P300: stato dell'arte e sviluppo di un nuovo sistema di analisi single trial, Tesi specialistica, Università degli studi di Padova, 2010.

[16] José del R. Millán, Frédéric Renkens, Josep Mouriño, Wulfram Gerstner, Noninvasive Brain-ActuatedControl of a Mobile Robot by Human EEG, *Biomedical Engineering, IEEE Transactions on*, vol. 51, no. 6, pp. 1026-1033, 2004.

[17] http://www.bem.fi/book/13/13.htm

[18] Jason M.O'Kane, A Gentle Introduction to ROS, ed.2, 2014.

[19] http://www.ros.org

[20] http://wiki.ros.org

[21] http://www.jitrc.com/downloads/ppt_reports/ROS_tut_jit.pdf

[22] https://robohow.eu/_media/meetings/first-integration-workshop/ros-tutorial.pdf

[23] http://www.turtlebot.com/

[24] http://kobuki.yujinrobot.com/home-en/about/reference-platforms/turtlebot-2/

[25] http://www.clearpathrobotics.com/turtlebot_2/features/

[26] https://www.hokuyo-aut.jp/02sensor/07scanner/urg_04lx_ug01.html

[27] https://www.hokuyo-aut.jp/02sensor/07scanner/download/products/urg-04lx-ug01/

[28] http://effistore.effidence.com/capteurs-mesures/sensors/scanners-lasers/hokuyo-urg-04lx-ug01.html

[29] https://it.wikipedia.org/wiki/GStreamer

[30] http://www.z25.org/static/_rd_/videostreaming_intro_plab/

[31] http://docs.gstreamer.com/display/GstSDK/Basic+tutorial+10

[32] http://wiki.oz9aec.net/index.php/Gstreamer_cheat_sheet

Ringraziamenti

Ringrazio il Prof. Emanuele Menegatti per avermi accolto come tesista, per la Sua disponibilità e per avermi dato la possibilità di lavorare in un laboratorio di ricerca;

Ringrazio il Dr. Luca Tonin per avermi aiutata a realizzare il mio progetto, per i preziosi suggerimenti, critiche e osservazioni e per aver risposto ad ogni mia domanda con precisione;

Ringrazio il Dr. Marco Carraro per avermi aiutata in laboratorio ogni qualvolta ne avessi avuto bisogno e con pazienza avermi insegnato ad usare nuovi strumenti;

Ringrazio tutte le persone del laboratorio di Sistemi Autonomi Intelligenti IAS-Lab dell' Università di Padova per avermi accolto e per la loro cordialità e gentilezza;

Ringrazio i miei genitori per essermi sempre stati vicini e per avermi sostenuto nelle mie scelte;

Infine ringrazio mia sorella e Riccardo di essermi stati a fianco sia nei momenti difficili sia in quelli felici.