

Università degli Studi della Basilicata

Dipartimento di Matematica Informatica ed Economia



CORSO DI LAUREA IN
SCIENZE E TECNOLOGIE INFORMATICHE

Tesi di Laurea Triennale in
SISTEMI OPERATIVI

“ROS – Un sistema operativo per i robot”

Relatore

Prof. Domenico Bloisi

Laureando

Sesta Fabio

Matr. 52295

Anno accademico 2019-2020

A mio padre, Ai miei cari

INDICE

1. Introduzione	1
1.1 Definizione del problema	1
1.2 Motivazioni.....	2
1.3 Obiettivi della tesi.....	3
1.4 Contributi personali	3
1.5 Struttura della tesi.....	3
2. Stato dell'arte	5
2.1 Origine e campi di utilizzo	5
2.2 Robot	9
2.2.1 Componenti.....	11
3. Robot Operating System (ROS).....	14
3.1 Storia.....	15
3.2 Caratteristiche.....	16
3.2.1 Linux/Ubuntu.....	18
3.3 Vantaggi e svantaggi	19
4. Architettura ROS.....	23
4.1 Filesystem level	23
4.1.1 Package	23
4.1.2 Metapackage	24
4.1.3 Package Manifest	24
4.1.4 Message Type Description.....	24
4.1.5 Service Type Definition.....	25
4.2 Computation graph level	25
4.2.1 Master	26

4.2.2	Nodo.....	27
4.2.3	Topic	28
4.2.4	Messaggi	28
4.2.5	Servizi	29
4.2.6	Bag	29
4.3	Community level	30
4.3.1	Distribuzioni	30
4.3.2	Repositories.....	30
4.3.3	ROS Wiki.....	31
4.3.4	ROS Answers.....	31
5.	Robot programming	32
5.1	Build system	32
5.1.1	Catkin.....	32
5.1.2	Workspace.....	33
5.2	Strumenti di sviluppo.....	36
5.3	Nozioni base: C++ in ROS	43
5.3.1	Inizializzazione, avvio ed arresto di un nodo.....	44
5.3.2	Creazione di messaggi	45
5.3.3	Pubblicare e sottoscrivere nei topic	45
5.3.4	Creazione e chiamata di servizi	46
5.3.5	Callback e spinning.....	47
6.	Casi di studio.....	50
6.1	Architettura master-slave e creazione di messaggi.....	50
6.2	Mappare un ambiente con SLAM	54
	Conclusioni.....	61
	Bibliografia e sitografia.....	63

1. Introduzione

Nel corso degli anni, lo sviluppo hardware, inerente al campo della robotica, ha compiuto notevoli progressi ed è tutt'ora in continua evoluzione per via della sua natura interdisciplinare che trova applicazione nei più svariati contesti, dalle case (domotica) all'industria, dagli ospedali (biomedica) ai campi di guerra (militare), fino ad arrivare allo sviluppo di robot umanoidi dalle sembianze umane, dotati di intelligenza artificiale ed in grado di agire autonomamente. Se da un lato questa evoluzione ci permette di disporre di hardware sempre più potente e più accessibile a livello di costi, dall'altro rende più complesso lo sviluppo delle applicazioni di robotica, in quanto bisogna tener conto di tutte le possibilità e circostanze in cui un robot può trovarsi mentre cerca di raggiungere lo scopo per cui è stato programmato. Negli ultimi anni, nella programmazione di applicazioni robotiche, ha acquisito sempre più notorietà ROS (Robot Operating System), che a discapito del nome non è ancora un vero e proprio sistema operativo. Esso, infatti, è definito come meta-sistema operativo, che si appoggia su un sistema operativo "host", quasi sempre una distribuzione di Linux/Ubuntu.

1.1 Definizione del problema

Il software è la parte essenziale, ovvero il cervello artificiale, di un robot e senza di esso non potrebbe svolgere alcuna funzione. Negli anni si è cercato di raggiungere una sempre più elevata standardizzazione del codice, in modo da avere soluzioni già pronte per le applicazioni robotiche, tali da poter essere implementate su diversi robot, senza dover integrare altre linee di codice. In altre parole, il problema principale da risolvere è quello di poter utilizzare lo stesso software su robot diversi, potendoli cambiare a piacimento, senza doversi

preoccupare dell'implementazione. Per poter fare ciò, vi è innanzitutto la necessità di avere uno strato di astrazione hardware, che permetta di scindere il codice dal tipo di componenti e di robot. Dopodiché si rende necessario un framework con il quale poter costruire blocchi di codice ed infine occorre un sistema capace di far comunicare tali blocchi, i quali possono trovarsi anche su macchine differenti, detto anche “middleware”. Un sistema così complesso ed articolato è ROS che negli ultimi anni ha avuto una crescita esponenziale, grazie al sempre più frequente impiego nell'ambito di sviluppo di applicazioni robotiche.

1.2 Motivazioni

Nel settembre 2020, i robot operanti nelle industrie di tutto il mondo sono 2,7 milioni, un numero così elevato, senza contare le tipologie di robot utilizzate negli altri numerosi settori in cui trovano applicazione. Siamo oramai circondati dai robot e ne esistono dunque una vasta gamma con componenti, sensori, attuatori diversi, costruiti da numerosi produttori. Il collante tra tutte queste differenze è ROS, che permette di sviluppare software funzionante su una vasta tipologia di robot, senza dover scendere nei dettagli implementativi di ognuno di essi. Acronimo di Robot Operating System, a discapito del nome non si identifica in un sistema operativo, nel senso stretto del nome. Viene definito, infatti, come meta-sistema operativo che comprende caratteristiche tipiche di un sistema operativo (astrazione dell'hardware sottostante, gestione dei processi, package management, gestione dei dispositivi), elementi tipici di un middleware (fornisce l'infrastruttura per la comunicazione tra processi/macchine differenti) e di un framework (tools di utilità per lo sviluppo, debugging e simulazione).

1.3 Obiettivi della tesi

Lo scopo dell'elaborato è quello di introdurre ed illustrare i concetti base di ROS, ovvero come creare dei sistemi funzionanti, e in che modo le informazioni, sotto forma di messaggi, viaggiano all'interno di essi. Si introducono i 3 livelli concettuali: Filesystem level, Computation Graph level e Community level, che rappresentano l'architettura dei sistemi sviluppati utilizzando il framework ROS. Infine, si vogliono presentare due casi di studio, per mostrare la comunicazione all'interno dell'architettura ROS ed il vantaggio di avere algoritmi standard, testati e funzionanti, forniti dalla larga community, pronti per essere implementati nel codice dell'applicazione. Attraverso i casi di studio, si vuole dimostrare l'efficacia di ROS ed il vantaggio della larga community alle sue spalle.

1.4 Contributi personali

Il contributo che si vuole dare, è quello di effettuare una panoramica dell'ecosistema ROS, spiegandone i concetti base e l'architettura creata da questo framework. Inoltre si rende disponibile una macchina virtuale, già dotata del sistema operativo Ubuntu 20.04 e della versione Desktop-Full di ROS Noetic. Tale macchina virtuale è disponibile al seguente link:

<https://drive.google.com/drive/folders/1YxOnt0ewA4PF1rNKxUB4IRKNUOLJ-W6y?usp=sharing>

1.5 Struttura della tesi

Nel secondo capitolo, viene introdotto il concetto di robotica, l'origine di tale disciplina e le varie sotto-discipline che si sono create grazie ai sempre più

numerosi campi in cui viene utilizzata. Inoltre viene introdotto il concetto di robot, la loro classificazione in base alla “generazione” ed i componenti principali che un robot deve possedere (struttura meccanica, sensori, attuatori, sistema di controllo). Nel terzo capitolo viene introdotto l’argomento principale della tesi, ovvero ROS (Robot Operating System), il quale viene definito come meta-sistema operativo, che ingloba caratteristiche tipiche di un vero e proprio sistema operativo (astrazione dell’hardware sottostante, gestione dei processi, package management, gestione dei dispositivi), unite ad elementi tipici di un middleware (infrastruttura per la comunicazione tra processi/macchine differenti) ed elementi di un framework (tools di utilità per lo sviluppo, debugging e simulazione). Dopo aver elencato vantaggi e svantaggi del suo utilizzo, nel quarto capitolo viene illustrata l’architettura con la quale vengono gestite le informazioni in ROS. Tale struttura si sviluppa su 3 livelli concettuali: Filesystem Level, Computation Graph Level e Community Level. Quest’ultimo è il frutto del contributo della comunità di sviluppatori ROS, la quale è uno dei punti forti di tale framework, motivo per cui è utilizzato sempre con più frequenza. Nel quinto capitolo vengono introdotti i concetti base per programmare applicazioni robotiche con ROS, come il build system Catkin, il concetto di workspace e le varie istruzioni in C++ per creare l’infrastruttura ROS (nodi, topic, servizi ecc.). Nel sesto capitolo, vengono presentati dei casi di studio che mostrano l’interazione e lo scambio di messaggi dei moduli ROS, anche tra macchine differenti, ed una simulazione utile per mappare un ambiente, utilizzando gli algoritmi, in questo caso l’algoritmo SLAM (Simultaneous Localization and Mapping), messi a disposizione dalla community.

2. Stato dell'arte

La robotica è una scienza interdisciplinare che si occupa della progettazione e dello sviluppo di robot. Essa studia e sviluppa metodi che permettano a un robot di eseguire dei compiti specifici riproducendo in modo automatico il lavoro umano. In essa confluiscono approcci di molte discipline, sia di natura umanistica (es: linguistica), sia scientifica (es: elettronica, fisica, informatica, matematica, meccanica).

2.1 Origine e campi di utilizzo

Il primo a parlare di robotica fu Isaac Asimov in un racconto di fantascienza del 1942. Egli è ritenuto il profeta della robotica ed il padre delle tre leggi della robotica e dei robot cosiddetti “positronici”.

Le 3 leggi fondamentali della robotica di Asimov possono essere così sintetizzate:

1. Un robot non può far del male a un essere umano né consentire, restando inoperoso, che un essere umano si trovi in pericolo
2. Un robot deve obbedire agli ordini impartiti dagli esseri umani, a meno che tali ordini non entrino in conflitto con la prima legge
3. Un robot deve proteggere la sua esistenza, a meno che tale protezione non vada in conflitto con la prima o la seconda legge

La robotica è nata come branca dell'ingegneria meccatronica, nella quale convergono tre diverse materie: meccanica, elettronica e informatica. A sua volta essa è una branca dell'ingegneria che si occupa dello studio di sistemi meccanici intelligenti da impiegare in contesti industriali e civili per semplificare il lavoro dell'uomo.

Proprio in virtù della sua natura interdisciplinare, la robotica trova applicazione nei più svariati contesti e questo ha fatto sì che nascessero varie sotto-discipline, anche se molto simili tra di loro. Tra le tante, alcune delle più importanti sono:

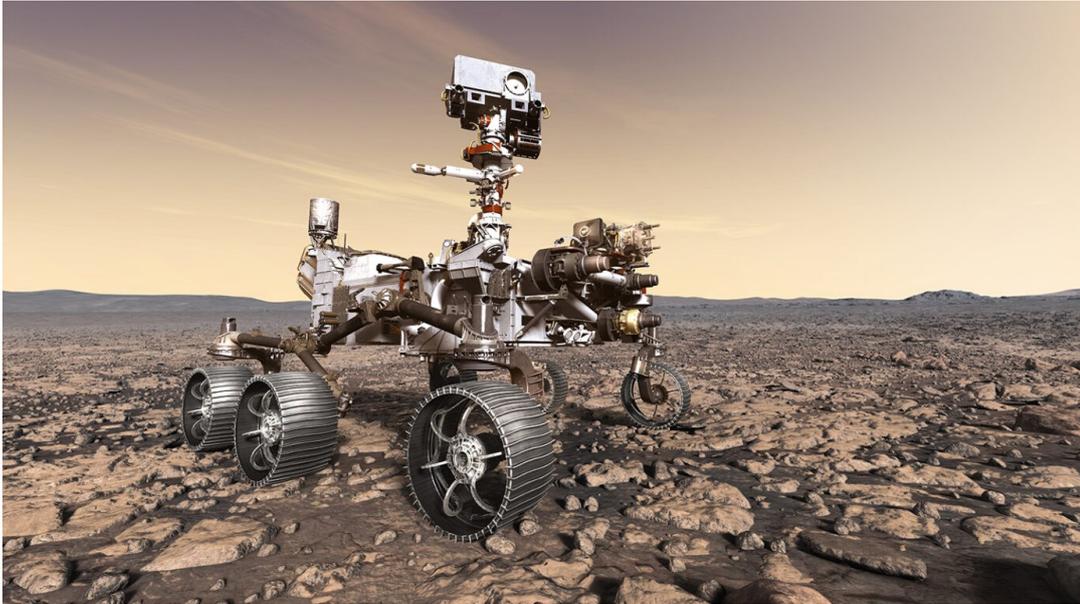
- **Domotica:** ha come obiettivo l'automazione applicata all'ambiente domestico, come la regolazione della temperatura interna all'edificio/stanza, l'accensione e lo spegnimento di luci ed elettrodomestici. Comprende anche tecnologie di aiuto ai portatori di handicap mentali o fisici nella vita quotidiana in casa.



(A sinistra Echo Dot 4 (Amazon Alexa), a destra Google Home)

- **Robotica biomedica:** comprende robot capaci di assistere il chirurgo durante le operazioni, radioterapia robotica, robot telecontrollati con tecnologie dette di telepresenza che permettono al chirurgo di operare a distanza. Rientrano nella categoria anche le sofisticate apparecchiature per analisi biologiche utilizzate nei laboratori.

- **Robotica industriale:** si riferisce a macchine che sostituiscono l'uomo in operazioni faticose e/o ripetitive. Il campo industriale è sicuramente quello in cui i robot hanno trovato maggiore diffusione: il loro impiego nelle catene di montaggio ha permesso alle aziende di abbattere notevolmente i costi accelerando e migliorando la produzione. Quando parliamo di robot industriali facciamo riferimento soprattutto a bracci robotizzati controllati da software, che possono essere utilizzati per il trasporto, lo smistamento e il confezionamento delle merci o per l'assemblaggio e la saldatura di materiali.
- **Robotica militare:** si riferisce in genere a robot utilizzati a scopi ispettivi, attualmente utilizzati, più che altro, con scopi di ricognizione e vigilanza. Un esempio di queste applicazioni sono i droni, veicoli controllati a distanza, che in caso di emergenza possono anche compiere diversi compiti in totale autonomia. Questo permette la ricognizione di campi di guerra, senza mettere a repentaglio vite umane. Un altro esempio sono i robot artificieri, che sono in grado di compiere analisi su un ordigno esplosivo ed eventualmente neutralizzarlo a distanza, riducendo drasticamente i rischi per gli artificieri.
- **Robotica spaziale:** è relativa alle applicazioni e all'impiego di robot fuori dall'atmosfera terrestre. Esempi di questi robot sono le sonde esplorative impiegate in diverse missioni sui pianeti del sistema solare, i bracci manipolatori sulla ISS, utilizzati in sostituzione degli astronauti per alcune attività extraveicolari, ed i più avanzati rover mai costruiti, che nel luglio 2020 hanno intrapreso il loro viaggio verso Marte.



(render del rover Perseverance, crediti: JPL-Caltech/NASA)

- **Robotica terapeutica:** spesso utilizzata per i bambini, può avere un fine puramente ludico, come nel caso dei robot giocattolo oppure un indirizzo educativo/terapeutico. Basti pensare all'utilizzo di robot umanoidi come Buddy, Milo e Nao robot nelle terapie per la cura dell'autismo e dei disturbi dell'apprendimento. Studi recenti dimostrano che interagendo con i robot, i bambini sono più reattivi e propensi all'ascolto, le loro capacità relazionali migliorano e con esse anche il rendimento a scuola.
- **Robotica umanoide:** è in assoluto tra i campi di ricerca più affascinanti. Il suo obiettivo è realizzare robot dalle sembianze umane, dotati di intelligenza artificiale e in grado di agire autonomamente. I robot umanoidi sono stati progettati per essere utilizzati prevalentemente in ambito domestico, ma ne esistono anche altri con finalità educative o con compiti di ricerca e salvataggio. La nazione guida in questo settore è il Giappone, dove si lavora al robot umanoide più avanzato al mondo, Asimo. In Italia,

invece, il robot R1 è stato progettato e realizzato dall'Istituto Italiano di Tecnologia, che da anni sta perfezionando anche iCub, il robot bambino, un umanoide molto sofisticato.



2.2 Robot

L'uso del termine robot, dal ceco “robota”, risale però al 1920 quando lo scrittore Karel Čapek che lo utilizza col significato di “lavoratore”.

Secondo il Robot Institute of America, il robot è un “manipolatore polifunzionale” in grado di eseguire diversi compiti attraverso una serie di movimenti programmati.

Il primo robot dell'era moderna, viene introdotto in ambito industriale solo all'inizio degli anni Sessanta. È un braccio meccanico, chiamato UNIMATE, specializzato nella saldatura delle scocche delle automobili. Inventato dagli ingegneri americani George Devol e Joseph Engelberger, viene acquistato nel 1961 da una fabbrica della General Motors, nel New Jersey.



Al giorno d'oggi, un robot è un'apparecchiatura artificiale che compie determinate azioni in base ai comandi che gli vengono dati e alle sue funzioni, sia in base ad una supervisione diretta dell'uomo, sia autonomamente basandosi su linee guida generali, magari usando processi di intelligenza artificiale; questi compiti tipicamente sono eseguiti al fine di sostituire o coadiuvare l'uomo, come ad esempio nella fabbricazione, costruzione, manipolazione di materiali pesanti e

pericolosi, o in ambienti proibitivi o non compatibili con la condizione umana o semplicemente per liberare l'uomo da impegni.

Ormai largamente diffusi in moltissimi settori del lavoro, nel settembre del 2020 la International Federation of Robotics (IFR) effettua un rapporto in cui afferma ci siano 2,7 milioni di robot industriali che operano nelle fabbriche di tutto il mondo (+12% del 2019).

I robot vengono classificati in base alla “generazione” a cui appartengono:

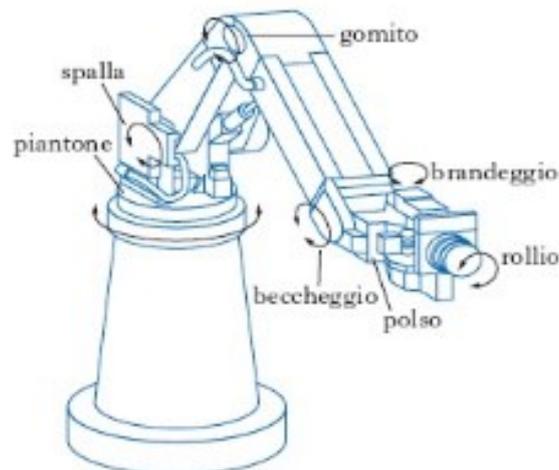
- **1^a generazione:** sono in grado semplicemente di eseguire sequenze prestabilite di operazioni indipendentemente dalla presenza o dall'intervento dell'uomo.
- **2^a generazione:** hanno la capacità di costruire un'immagine (modello interno) del mondo esterno, di correggerla e perfezionarla continuamente. Sono in grado di scegliere la migliore strategia di controllo e di finire ciò che gli è stato programmato, malgrado la presenza di fenomeni di disturbo non prevedibili a priori.
- **3^a generazione:** hanno un'intelligenza artificiale e sono in grado di costruire nuovi algoritmi e di verificarne la coerenza da soli.

2.2.1 Componenti

I robot sono di fatto dei sistemi ibridi complessi, formati da una parte hardware elettronica opportunamente programmata da un software, che regola o controlla una parte meccanica. I componenti principali di un robot possono essere classificati in:

- **Struttura meccanica**

Corrisponde al sistema di movimento e locomozione del robot. Gli organi meccanici, si distinguono tra apparati per compiere operazioni e attività in un posto fisso, oppure apparati in grado di spostarsi. Se volessimo fare un parallelismo con gli organi di movimento degli esseri umani, divideremmo gli organi meccanici in arti superiori (braccia meccaniche, comprese però le estensioni cosiddette *end-effector*, cioè gli attrezzi come pinze e mani robotiche per la manipolazione) e arti inferiori (organi meccanici come ruote, rotelle, sistemi di cinematica, oppure “gambe” meccaniche).



- **Strumenti di percezione**

Tali strumenti sono sensori che consentono al robot di acquisire dati, sia sullo stato interno della struttura meccanica (sensori *propriocettivi*: permettono al robot di “percepire”, per esempio, posizione e velocità), sia sull’ambiente esterno circostante (sensori *esterocettivi*: fanno percepire, per esempio, forza e prossimità e danno al robot una visione artificiale).

- **Strumenti d'azione**

Tali strumenti sono attuatori, dispositivi che convertono dell'energia da una forma a un'altra, in modo che questa agisca nell'ambiente fisico al posto dell'uomo. Essi imprimono il movimento al robot attraverso l'azionamento di motori elettrici, idraulici e talvolta pneumatici.

- **Strumenti di controllo**

Comprendono il software che gestisce le operazioni che deve svolgere il sistema robotico, sulla base della sua struttura meccanica e dei dati forniti dai sensori. Questi dati vengono analizzati ed in base a loro vengono inviati segnali di azionamento agli attuatori.



3. Robot Operating System (ROS)

Un robot così definito dovrebbe essere dotato di connessioni guidate dalla retroazione tra percezione e azione, e non dal controllo umano diretto. Per poter funzionare, prendere decisioni, deve disporre di una “intelligenza artificiale” costituita dal software. Nell’ambito della robotica, la tecnologia ha fatto passi da gigante negli ultimi decenni, richiedendo uno sviluppo software sempre più innovativo. Infatti, per programmare un robot, occorre sviluppare del codice su più livelli, dal software base utilizzato per i driver, a software complesso per ricreare una forma di percezione dell’ambiente circostante, fino ad uno di più alto livello per ricreare una sorta di intelligenza artificiale che reagisca agli input degli strumenti di percezione. L’architettura di controllo del sistema robotico, che deve reagire agli input sensoriali, può essere diversa, ad esempio Deliberative Control o Reactive Control, per citarne alcune. Programmare un robot non è semplice, poiché si hanno risorse limitate, in termini di memoria, processori o energia. Inoltre, basterebbe un minimo errore nel programmare un sensore e la presenza di un minimo ostacolo non segnalato potrebbe essere un gravissimo problema. Per ovviare a queste difficoltà, sono stati sviluppati vari framework per contenere la complessità e facilitare il testing, quindi il rilascio di prototipi. La larga diffusione di robot nell’industria e la necessità di integrarli con altri sistemi di automazione, ha creato forme di programmazione volte alla standardizzazione, per facilitare l’integrazione di sistemi diversi tra loro. Un esempio di tali framework, ed anche uno dei più usati, è ROS (Robot Operating System).



3.1 Storia

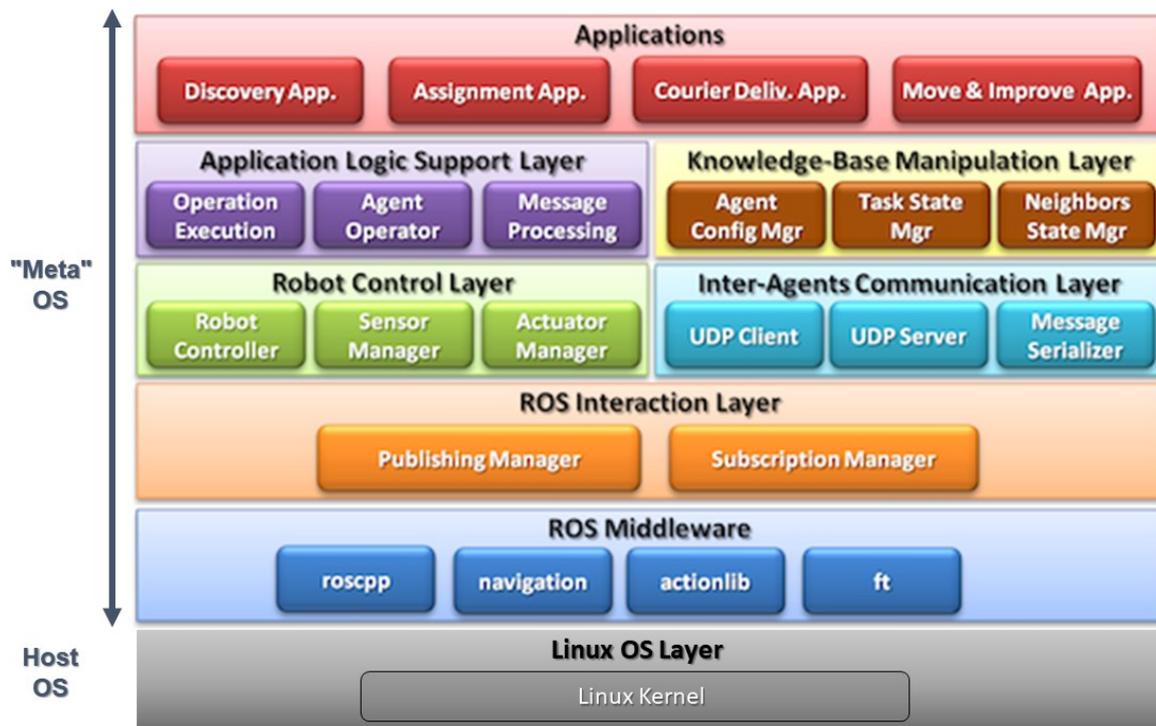
Agli inizi del 2000, Eric Berger e Keenan Wyrobek, due studenti di dottorato del laboratorio di robotica a Stanford, mentre lavoravano con dei robot per svolgere compiti di manipolazione in ambienti umani, notarono che molti dei loro colleghi erano frenati e spaesati, a causa della diversa natura di programmazione della robotica. Iniziarono a creare, quindi, un sistema di base per fornire un punto di partenza, a livello accademico, per la programmazione di robot. Nei loro primi passi verso questo sistema, i due costruirono il PR1 (Personal Robot 1) come prototipo hardware e iniziarono a lavorare sul software basandosi su di esso, prendendo in prestito le migliori pratiche di programmazione da altri framework per software robotici open-source già esistenti. In particolare da switchyard, un sistema che Morgan Quigley, un altro studente di dottorato di Stanford, aveva modellato a sostegno del progetto STAIR (STanford Artificial Intelligence Robot). Mentre cercavano finanziamenti per ulteriori sviluppi, Berger e Wyrobek incontrarono Scott Hassan, che condivise la loro idea di creare un "Linux per la robotica" e li invitò a far parte della sua azienda, Willow Garage. Nel 2007, Willow Garage iniziò a sviluppare il robot PR2 (successore del PR1) ed il software ROS, di cui il primo commit del codice è stato effettuato a SourceForge il 7 novembre dello stesso anno. Nel 2010 viene rilasciata la prima versione stabile, presentante la maggior parte delle caratteristiche odierne. ROS è stato rilasciato sotto licenza BSD (Berkeley Software Distribution) e ciò ha favorito fin da subito il suo sviluppo nel corso degli anni grazie al contributo della comunità di ricerca mondiale ed è divenuto uno dei suoi punti di forza. Varie versioni del software vengono rilasciate negli anni a seguire e nel 2013 l'Open Source Robotics Foundation diventa manutentore di ROS. Grazie alla grande quantità di ricerca svolta in campo industriale, inoltre, nasce ROS Industrial, che può estendere le capacità avanzate di ROS alla produzione. Da qui, ogni anno viene

rilasciata una nuova versione. Nel 2017 l'OSFR cambia il proprio nome in Open Robotics.

Attualmente vi sono due categorie di versioni: ROS1 e ROS2. Le versioni di ROS2 hanno una significativa differenza nelle API, in quanto supportano la programmazione in tempo reale, una più ampia varietà di ambienti informatici e utilizzano una tecnologia più moderna.

3.2 Caratteristiche

ROS è una collezione in continua crescita di librerie software open source e tools progettate al fine di aiutare gli sviluppatori nella realizzazione di applicazioni robot. Acronimo di Robot Operating System, a discapito del nome non si identifica in un sistema operativo, nel senso stretto del nome. Viene definito, infatti, come meta-sistema operativo, inglobando sì le caratteristiche tipiche di un vero e proprio sistema operativo (astrazione dell'hardware sottostante, gestione dei processi, package management, gestione dei dispositivi), ma arricchendolo con elementi tipici di un middleware (fornisce l'infrastruttura per la comunicazione tra processi/macchine differenti), e di un framework (tools di utilità per lo sviluppo, debugging e simulazione). La differenza principale rispetto all'architettura di un sistema operativo tradizionale, è che ROS, in realtà, viene eseguito su un sistema operativo "host". Di seguito viene mostrato uno schema che rappresenta tale architettura:



ROS quindi non è tecnicamente un sistema operativo, ma un framework software (o middleware) che fornisce un ambiente di sviluppo integrato all'applicazione robot.

Il sistema operativo “host” su cui viene eseguito è quasi sempre Linux, o meglio Ubuntu, in quanto ROS opera essenzialmente su piattaforme Unix-Based, anche se esistono altre piattaforme sperimentali, come Windows e MacOS.

Tra le funzionalità fornite da questo meta-sistema operativo, sono incluse: multithreading, controllo del dispositivo di basso livello, gestione dei package e astrazione hardware. Quest’ultima, consente agli sviluppatori di applicazioni robotiche di concentrarsi sul “cosa deve fare il robot” senza preoccuparsi di implementare codice per specifici sensori o attuatori, in quanto è possibile scrivere codice di base, utilizzabile indipendentemente dalla loro marca.

ROS è stato pensato come un sistema multilingue: i moduli ROS possono essere scritti in qualsiasi lingua per la quale esiste una libreria client. I due linguaggi più

comuni sono C++ e Python, ma esistono anche librerie scritte in LISP, Java, Ruby e MATLAB. Inoltre, tutto il software in ROS è organizzato in package che possono essere creati dallo sviluppatore, ma sono anche distribuiti in modo indipendente dalla comunità e ospitati su un repository software. ROS fornisce, quindi, una suite di package guidati dalle comunità di sviluppatori che implementano una serie densa di funzionalità robotiche come la pianificazione della traiettoria, la percezione, la visione, il controllo e la manipolazione.

Il motivo principale della popolarità e dello sviluppo di ROS è proprio la comunità supporto. Gli sviluppatori ROS, presenti in tutto il mondo, creano e mantengono attivamente i package. ROS è stato costruito da zero per incoraggiare lo sviluppo collaborativo di software di robotica. Ad esempio, degli esperti nella mappatura di ambienti interni potrebbero contribuire con un sistema di prim'ordine per la produzione di mappe. Un altro gruppo di sviluppatori potrebbe essere esperto nell'uso delle mappe per navigare. ROS è stato progettato specificamente per gruppi come questi, per collaborare e sviluppare software da poter mettere a disposizione degli altri.

3.2.1 Linux/Ubuntu

Come visto in precedenza, ROS funziona principalmente su un sistema operativo GNU/Linux, che a sua volta è ispirato al sistema operativo Unix. Il kernel Linux è in grado di eseguire il multitasking in sistemi multiutente, inoltre GNU/Linux è gratuito ed open source. Ne esistono molte distribuzioni, le quali fondamentalmente utilizzano il kernel Linux come componente principale, ma hanno differenze nell'interfaccia grafica. Alcune delle distribuzioni Linux più popolari sono Ubuntu, Debian e Fedora.



Ubuntu è la distribuzione Linux più indicata per lavorare con ROS. I vantaggi di questo sistema operativo nella programmazione di applicazioni di robotica sono la sua reattività, la natura leggera e l'alto grado di sicurezza. Al di là di questi fattori, Ubuntu ha un ottimo supporto della comunità e ci sono frequenti rilasci, il che lo rende un sistema operativo aggiornato. Esso ha anche versioni di supporto a lungo termine (LTS), che forniscono supporto agli utenti per un massimo di cinque anni. Un'applicazione robotica può essere eseguita su un sistema operativo che fornisce funzionalità per comunicare con attuatori e sensori del robot. Un sistema operativo basato su Linux può fornire una grande flessibilità per interagire con hardware di basso livello e fornire la possibilità di personalizzare il sistema operativo in base all'applicazione del robot. Questi fattori hanno portato gli sviluppatori di ROS ad attenersi a Ubuntu, il quale è l'unico sistema operativo completamente supportato da ROS.

3.3 Vantaggi e svantaggi

ROS non è l'unico framework per la robotica. Di seguito vengono illustrati alcuni vantaggi e svantaggi nell'utilizzarlo.

L'utilizzo di ROS presenta molti vantaggi, tra cui:

- **Supporto di linguaggi di programmazione di alto livello:** ROS supporta i popolari linguaggi di programmazione utilizzati nella programmazione

dei robot, come C++, Python e Lisp, oltre ad uno sperimentale supporto per linguaggi come C#, Java, Node.js, MATLAB.

- **Algoritmi standard e librerie di terze parti:** ROS implementa algoritmi di robotica popolari come PID, SLAM e pianificatori di percorsi come Dijkstra e AMCL. Gli algoritmi standard riducono il tempo di sviluppo per la prototipazione di un robot. Inoltre ROS integra librerie di terze parti come OpenCV per la visione artificiale e PCL per elaborare i dati e creare tale visione. Queste librerie permettono al programmatore di costruire potenti applicazioni.
- **Numerosi tool:** ROS è pieno di tantissimi strumenti per il debug, la visualizzazione e l'esecuzione di una simulazione, come rqt_gui, RViz e Gazebo. Rviz viene utilizzato per la visualizzazione con telecamere, scanner laser ed altri dispositivi. Gazebo invece è utilizzato per simulare il comportamento dei robot. Questi strumenti rendono lo sviluppo delle applicazioni più semplice.
- **Operabilità inter-piattaforma:** ROS è un “middleware” che consente la comunicazione tra nodi diversi, ad esempio nodi scritti in C++ e nodi scritti in Python. Questo tipo di flessibilità non è disponibile in altri framework.
- **Modularità:** di solito nelle applicazioni robotiche autonome se uno qualsiasi dei thread del codice principale si blocca, l'intera applicazione può interrompersi. In ROS, vengono utilizzati nodi diversi per ogni processo e se un nodo si blocca, il sistema può ancora funzionare. Inoltre, ROS fornisce metodi affidabili per riprendere le operazioni anche se qualsiasi sensore o motore è guasto.

- **Gestione simultanea delle risorse:** in ROS possiamo accedere ai dati utilizzando i topic ROS. Qualsiasi numero di nodi ROS può iscriversi ad essi ed acquisire i dati, per poi elaborarli. Si riduce così la complessità del calcolo e si semplifica l'operazione di debug dell'intero sistema.
- **Standardizzazione e riuso del codice:** ROS garantisce package debuggati e testati di molti algoritmi usati in robotica. Le interfacce ROS per le ultime versioni hardware o algoritmi usciti sono spesso disponibili e ciò permette agli sviluppatori di concentrarsi, nella creazione di un'applicazione, sulle idee e sul testing e non perdere tempo con codice integrativo.
- **Testing:** ROS permette di sviluppare sistemi in cui la parte di controllo a livello hardware sia separata dalla gestione dei meccanismi di più alto livello, e di testare quest'ultimi simulando l'hardware e il software di basso livello. Inoltre ROS offre la possibilità di memorizzare i dati ottenuti dai sensori e in fase di test in un formato che prende il nome "bag", per poi visualizzarli, analizzarli e riutilizzarli più e più volte nello stesso processo o in processi differenti. Testato fisicamente sul robot, o su un simulatore o vengano utilizzati dei dati di tipo "bag", il software non richiede integrazioni o modifiche al codice perché tutte le modalità forniscono uno stesso interfacciamento.
- **Comunità attiva:** In ROS, la comunità di supporto è attiva ed esiste un portale web per gestire le richieste di supporto da parte degli utenti. Perciò ogni libreria di ROS è verificata. Inoltre, la comunità ROS ha una crescita costante di sviluppatori in tutto il mondo.

Alcuni degli svantaggi e delle difficoltà, motivo per cui non viene utilizzato ROS, sono i seguenti:

- **Difficoltà nell'apprendimento:** ROS può essere difficile da imparare. Ha una curva di apprendimento ripida e gli sviluppatori dovrebbero acquisire familiarità con molti nuovi concetti per ottenere vantaggi dal framework ROS.
- **Difficoltà nella modellazione del robot:** la modellazione del robot in ROS viene eseguita utilizzando URDF, ovvero una descrizione del robot basata su XML. In V REP, invece, possiamo costruire direttamente il modello di robot 3D nella stessa GUI. Imparare a modellare un robot in ROS richiederà molto tempo e anche la creazione utilizzando i tag URDF richiede molto tempo rispetto ad altri simulatori.
- **Ritardo nei messaggi:** ROS ha molti nodi che dialogano tra loro attraverso la rete. Generalmente ciò provoca un ritardo molto variabile nella ricezione dei messaggi. Di conseguenza, si ha un grande ritardo di controllo per assicurarsi che tutti i messaggi arrivino e non è possibile reagire velocemente agli eventi.
- **Cambio di piattaforma complesso:** se, per una ragione o per l'altra, si volesse spostare l'applicativo ROS su un'altra piattaforma robotica, come OROCOS o YARP, ciò sarebbe molto complesso.

4. Architettura ROS

Il framework ROS permette di creare un'architettura a grafo orientato, dove le informazioni vengono processate nei nodi e la comunicazione tra di essi può avvenire in maniera asincrona, attraverso l'uso di topic sui quali possono pubblicare messaggi o riceverne tramite sottoscrizione, o in maniera sincrona con la chiamata di servizi, simili a RPC (Remote Procedure Call). Strutturalmente ROS si sviluppa su 3 livelli concettuali: Filesystem Level, Computation Graph Level e Community Level.

4.1 Filesystem level

Il Filesystem level riguarda principalmente le risorse ROS presenti fisicamente sul disco. Esso include package, metapackage, package manifest, tipi di messaggi "Message(msg) type" e tipi di servizi "Service(srv) type".

4.1.1 Package

Il software in ROS è organizzato in package, i quali possono contenere nodi ROS, librerie indipendenti, file di configurazione, software di terze parti ed altro ancora. L'obiettivo di questi package è fornire funzionalità facili da utilizzare, in modo che il software possa essere facilmente riutilizzato. La filosofia di ROS è che questi package siano funzionali, utili per l'utente, ma non troppo da renderli pesanti e difficili da utilizzare.

4.1.2 Metapackage

I metapackage sono particolari package che non contengono alcun test, codice, file o altri elementi che si trovano solitamente nei package (se non il solo *package.xml*). Un metapackage fa semplicemente riferimento a uno o più package correlati dalle stesse dipendenze, per essere utilizzati in un'applicazione specifica. La maggior parte dei metapackage in ROS ha preso il posto dei precedenti stack di rosbuild.

4.1.3 Package Manifest

Un package manifest è un file XML denominato *package.xml*, il quale fornisce tutti i metadati riguardanti un package: nome, versione, descrizione, licenza, dipendenze. Deve essere presente in tutti i package compatibili con il sistema di build Catkin.

4.1.4 Message Type Description

Una descrizione del tipo di messaggio indica che tipo di valori pubblica un nodo ROS. Le descrizioni dei messaggi sono archiviate in file *.msg* nella sottodirectory *msg* di un package ROS. Una descrizione del messaggio è un elenco di definizione di campi dati e/o di costanti su righe separate. I campi sono i dati inviati all'interno del messaggio e sono costituiti da un tipo (primitivo o composto) e da un nome, separati da uno spazio (ad es. *int32 x*). Le costanti definiscono valori utili che possono essere utilizzati per interpretare i campi dati e la loro definizione è come una descrizione di campo, tranne per il fatto che assegna anche un valore. Infatti, sono costituite da un tipo, da un nome e da un valore, questi ultimi due separati da un segno "=" (ad es. *int32 x=123*). I tipi di messaggio utilizzano convenzioni

di denominazione standard in ROS, come: <nome_package> + / + <nome_file > (ad es. *std_msgs/msg/String.msg* ha il tipo di messaggio *std_msgs/String*).

4.1.5 Service Type Definition

Simile alla descrizione del tipo di messaggio, una definizione del tipo di servizio indica il tipo di dato del messaggio scambiato nell'uso del servizio. Indica il tipo di dato del messaggio scambiato nell'uso dei servizi di ROS. Poiché i servizi sono utilizzati in un modello di comunicazione richiesta/risposta, la definizione del servizio è suddivisa in due parti: una relativa alla richiesta ed una alla risposta. Una definizione di un servizio è costituita, dunque, da queste due parti separate da una linea del tipo "---". Tali definizioni sono memorizzate in file *.srv* salvati nella sottodirectory *srv* di un package. Un esempio di semplice definizione di un servizio, che richiede una stringa e risponde con un'altra stringa, è il seguente:

```
string richiesta
---
string risposta
```

Così come i topic, il tipo di servizio utilizza le convenzioni di denominazione standard in ROS, dato da: <nome_package> + / + <nome_file>. (ad es. *my_srvs/srv/PolledImage.srv* ha il tipo di servizio *my_srvs/PolledImage*)

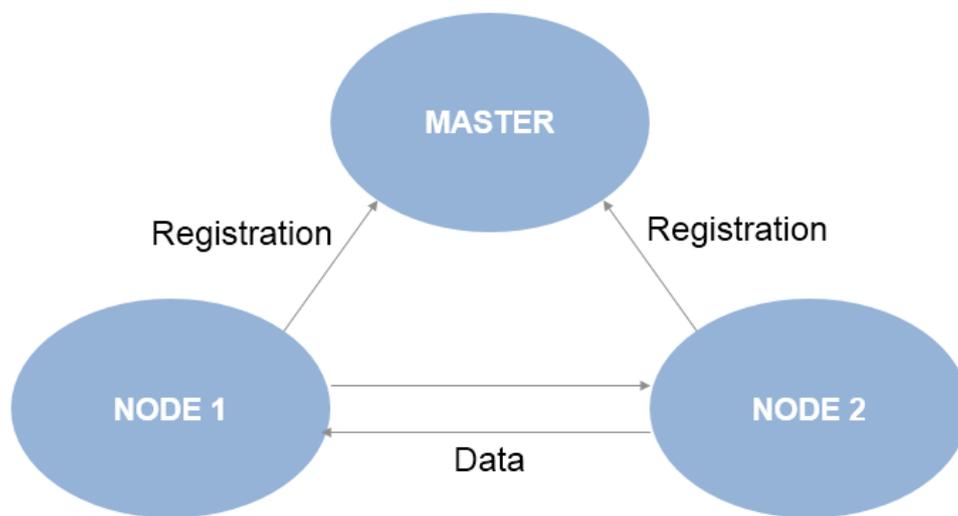
4.2 Computation graph level

Il Computation Graph è la rete peer-to-peer descritta da tutti i processi attivi in ROS che elaborano dati assieme. L'utilizzo di una topologia peer-to-peer permette di evitare carichi di traffico di messaggi eccessivi, ma richiede un name service per rendere possibile ai nodi di contattarsi a vicenda nella rete. Fanno parte del

Computation Graph Level: Master, Parameter Server, Nodi, Topic e Servizi (per lo scambio di messaggi rispettivamente asincroni e sincroni), Bag. Di seguito vengono discussi uno ad uno.

4.2.1 Master

In un sistema basato su ROS, il Master è un server centralizzato XML-RPC che offre ai nodi un servizio di registrazione e di naming, similmente all'informazione data da un server DNS. Permette infatti al singolo nodo di contattarne un secondo attraverso una metodologia peer-to-peer. Tiene inoltre traccia, per ogni singolo topic, dei relativi publisher e subscriber e allo stesso modo gestisce i servizi. Il master ha un well-known URI, in modo da essere accessibile da tutti i nodi.



4.2.1.1 Parameter Server

Il Master offre anche il *Parameter Server*, in cui l'utente può memorizzare vari parametri o valori e tutti i nodi possono accedervi. L'utente può anche impostare la privacy del parametro. Se è un parametro pubblico, tutti i nodi hanno accesso;

se è privato, solo un nodo specifico può accedere al parametro. Il Parameter Server è implementato attraverso XMLRPC ed offre, dunque, un servizio di memorizzazione e consultazione di parametri a runtime ai nodi che ne richiedono i servizi attraverso un'API di rete. Esso offre il vantaggio a tutti i tool di sviluppo di poter analizzare in qualsiasi momento la configurazione dello stato del sistema e modificarne i parametri se necessario.

4.2.2 Nodo

Un nodo è un processo che utilizza le API ROS per eseguire calcoli. I nodi presenti nel Computation Graph possono comunicare tra loro utilizzando dei Topic, servizi RPC o il Parameter Server. ROS è progettato per essere modulare e fine-grained, ovvero un sistema che comprende numerosi nodi, interpretabili come moduli software che si occupano di gestire un aspetto del comportamento del robot, come ad esempio la parte decisionale, il movimento, l'azionamento dei motori ed altro ancora. Tale sistema, grazie al carico computazionale ripartito tra i vari nodi di cui è costituito, ha il vantaggio di una maggiore tolleranza agli errori, poiché il crash di un singolo nodo non arresta tutto il sistema. La complessità del codice è ridotta rispetto ai sistemi monolitici ed i dettagli di implementazione sono ben nascosti, poiché i nodi espongono un'API minima al resto del grafico e le implementazioni alternative, anche in altri linguaggi di programmazione, possono essere facilmente sostituite. Ogni nodo in esecuzione dispone di quello che viene definito “**graph resource name**”, un nome che lo identifica univocamente nel sistema, e di un tipo che semplifica il processo di indirizzamento di un nodo eseguibile all'interno del filesystem. Tali tipi sono detti “**package resource names**” e sono costituiti dalla concatenazione del nome del package e del file eseguibile del nodo. Ogni nodo, inoltre, presenta un URI, che corrisponde alla

concatenazione *host:porta* del server XMLRPC attivo, il quale viene utilizzato per negoziare la connessione tra nodi e comunicare col Master.

4.2.3 Topic

I topic costituiscono il mezzo di comunicazione asincrono e unidirezionale per lo scambio di messaggi tra nodi, secondo una semantica di tipo publish/subscribe. Possono essere visti come bus che hanno un nome, in cui i nodi ROS possono inviare messaggi. Per ogni Topic ci possono più publisher e subscriber concorrenti. Un singolo nodo può, inoltre, pubblicare e/o sottoscrivere a più Topic. In generale, i nodi non sanno con chi stanno comunicando, disaccoppiando così la produzione dell'informazione (publisher) dal suo consumo (subscriber). Ogni topic è fortemente tipizzato dal tipo di messaggio che può essere pubblicato su di esso ed i nodi "subscribers" possono ricevere i messaggi solo se esso corrisponde. Ciò determina che all'interno del Topic sia possibile scrivere o leggere un solo tipo di messaggio.

4.2.4 Messaggi

La comunicazione tra nodi avviene tramite lo scambio di messaggi sui topic. In ROS un messaggio è una struttura dati fortemente tipizzata. Sono supportati i tipi primitivi standard (interi, numeri a virgola mobile, booleani, ecc.), così come gli array di tipi primitivi. I messaggi possono anche includere strutture e array annidati arbitrariamente (come per le *struct* in C). Come visto nella definizione di messaggi, la struttura di un messaggio è descritta da un semplice file di testo con estensione *.msg* contenuto nella sottocartella *msg* del package. Tale file è costituito da due parti, ovvero campi e costanti, che rispettivamente sono i dati

spediti all'interno del messaggio e i valori numerici utili a interpretare il significato dei campi.

4.2.5 Servizi

I servizi rappresentano la comunicazione sincrona tra nodi, secondo una semantica di tipo richiesta / risposta, ed implementano in ROS funzionalità di RPC (Remote Procedure Call). Nella rete si hanno quindi nodi che svolgono la funzione di service provider e nodi client. Il nodo che mette a disposizione un servizio è detto “nodo Server” ed il nodo che lo utilizza è detto “nodo Client”. Il nodo server mette a disposizione il servizio e invia risposte ai nodi client, che effettuano le richieste per utilizzarlo. I servizi, come visto nella definizione del loro tipo, sono definiti da una coppia di messaggi, uno per la richiesta e uno per la risposta, in un file con estensione `.srv`, contenuto nella sottocartella `srv` del package.

4.2.6 Bag

Sono file con estensione **.bag**, utilizzati in ROS per memorizzare dei messaggi. Possono essere creati utilizzando il tool `rosbag`, il quale si iscrive a uno o più topic e memorizza in un file i messaggi in forma serializzata. Questo meccanismo può essere utile per effettuare il logging nelle comunicazioni topic, per salvare e riprodurre i topic ed anche per registrare i dati da un robot per elaborarli successivamente.

4.3 Community level

Comprende tutte le risorse che permettono a comunità separate di ricercatori di poter interagire scambiando software. Di seguito vengono elencate le risorse più importanti.

4.3.1 Distribuzioni

Una distribuzione ROS è un insieme di package della stessa versione, simili alle distribuzioni di Linux/Ubuntu. Lo scopo delle distribuzioni ROS è di consentire agli sviluppatori di lavorare su una base di codice relativamente stabile. Le versioni ROS possono essere incompatibili con altre versioni e vengono spesso indicate con il nome in codice anziché con il numero di versione. Attualmente vengono rilasciati due tipi di distribuzioni: ROS1 e ROS2. Queste ultime si differenziano per il supporto allo sviluppo real-time.

L'ultima versione di ROS1 è **Noetic Ninjemys**, rilasciata il 23 maggio 2020.



4.3.2 Repositories

ROS fa affidamento su una rete di archivi di codice, in cui diverse istituzioni possono sviluppare e rilasciare i propri componenti software per robot. Per massimizzare la partecipazione della comunità, gli utenti e gli sviluppatori di tutto

il mondo sono incoraggiati mantenere i propri archivi di package ROS. Ogni repository può essere gestito e concesso in licenza come desiderato dal rispettivo manutentore, che mantiene la proprietà e il controllo diretto sul codice.

4.3.3 ROS Wiki

È il forum principale per documentarsi su ROS. Chiunque può registrarsi e contribuire alla crescita della comunità scrivendo tutorial, offrendo aiuto, correzioni, update e altro ancora. Il wiki ROS comprende, tra le tante cose, tutorial su come installare ROS e iniziare a programmare.

4.3.4 ROS Answers

È il sito utilizzato dalla community, per effettuare domande inerenti a ROS e ricevere risposte.

5. Robot programming

In questo capitolo si vogliono introdurre le nozioni base per programmare con ROS, quali il build system di ROS (Catkin), gli strumenti utili da riga di comando e le istruzioni base della POO in C++ per programmare con ROS.

5.1 Build system

Per lo sviluppo di singoli progetti software, gli strumenti esistenti come Autotools, CMake e i sistemi di compilazione inclusi negli IDE tendono ad essere sufficienti. Tuttavia, questi strumenti possono essere difficili da utilizzare da soli con ecosistemi di codice grandi, complessi, principalmente a causa dell'enorme numero di dipendenze, della complessa organizzazione del codice e delle regole di compilazione personalizzate che un particolare target potrebbe avere. Poiché questi strumenti sono molto generici e progettati per essere utilizzati dagli sviluppatori di software, tendono anche ad essere difficili da utilizzare da coloro che non hanno un background di sviluppo software.

ROS è una raccolta molto ampia di package e molti di essi dipendono l'uno dall'altro, utilizzano vari linguaggi di programmazione, strumenti e convenzioni di organizzazione del codice. Per questo motivo, il processo di creazione di una destinazione in un package potrebbe essere completamente diverso dal modo in cui viene creata un'altra destinazione.

5.1.1 Catkin

Catkin è il sistema di build ufficiale di ROS ed è il successore del sistema di compilazione ROS originale, *roscpp*.

Il nome Catkin, in italiano “amento”, deriva dall’infiorescenza che si trova sugli alberi di salice ed è un riferimento a Willow Garage, dove è stato creato.

Catkin è stato progettato per essere più convenzionale di rosbuid, consentendo una migliore distribuzione dei package, un migliore supporto per la compilazione incrociata e una migliore portabilità. In poche parole, mira a rendere più semplice la creazione e l'esecuzione del codice ROS utilizzando strumenti e convenzioni per semplificare il processo.

Esso combina macro di CMake e script in Python per fornire alcune funzionalità oltre al normale flusso di lavoro di CMake. CMake è una famiglia di strumenti open source e multiplatforma progettata per creare, testare e confezionare software.

Una delle filosofie alla base di ROS è ridurre al minimo il numero di strumenti ROS specifici necessari per creare, gestire e utilizzare package ROS e cercare sempre di rimandare a strumenti ben consolidati, ampiamente utilizzati, di terze parti, open source (ad esempio usando libtinyxml invece di scrivere un parser XML personalizzato).

Catkin è indipendente dall'ecosistema ROS e può essere utilizzato anche su progetti non ROS.

5.1.2 Workspace

I package Catkin possono essere creati come progetti autonomi, nello stesso modo in cui possono essere creati i normali progetti Cmake, ma Catkin fornisce anche il concetto di workspaces (spazi di lavoro), dove è possibile creare più package insieme in un’unica volta. Uno spazio di lavoro Catkin è una cartella in cui si può modificare, creare e installare package catkin ed è definito nella REP 128.

Per creare uno workspace, si utilizza il comando *catkin_init_workspace* per inizializzare un nuovo spazio di lavoro ROS. Se non lo si inizializza, non è possibile creare e compilare correttamente i package.

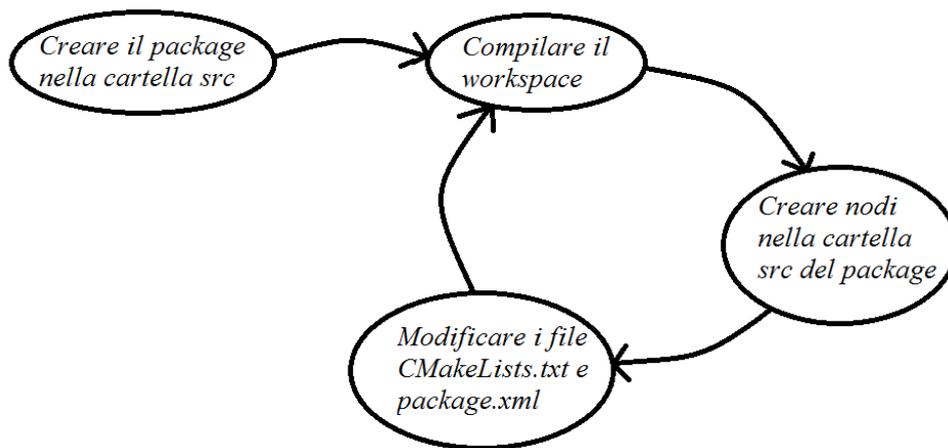
Esso può contenere fino a quattro spazi (cartelle) diversi, ciascuno dei quali ha un ruolo diverso nel processo dello sviluppo software:

- **src:** la cartella src all'interno della cartella dello spazio di lavoro catkin contiene il codice sorgente dei package catkin. Qui si può estrarre, controllare o clonare il codice sorgente per i package che si desidera creare. I package ROS compilano e creano un eseguibile solo quando si trovano in questa cartella. Quando eseguiamo il comando *catkin_make* dalla cartella dell'area di lavoro, esso controlla all'interno della cartella src e compila ogni package. La cartella src contiene un collegamento simbolico al file CMakeLists.txt di “top-level”. Questo file viene richiamato da CMake durante la configurazione dei progetti catkin.
- **build:** la cartella build è il punto in cui CMake viene invocato per creare i package Catkin. CMake e Catkin conservano qui le informazioni sulla cache e altri file intermedi: quando eseguiamo il comando *catkin_make* dallo spazio di lavoro ROS, vengono creati alcuni file build e CMake all'interno della cartella build. Questi file cache aiutano a prevenire la ricostruzione di tutti i pacchetti quando si riesegue il comando *catkin_make*; ad esempio, se si creano cinque package e poi si aggiunge un nuovo package alla cartella src, solo il nuovo package verrà compilato durante il successivo comando *catkin_make*. Ciò è dovuto a quei file di cache all'interno della cartella build.
- **devel:** quando eseguiamo il comando *catkin_make*, ogni pacchetto viene compilato e, se il processo di compilazione ha esito positivo, viene creato l'eseguibile di destinazione. L'eseguibile è archiviato nella cartella devel,

che contiene file di script di shell per aggiungere lo spazio di lavoro corrente al percorso dello spazio di lavoro ROS. La cartella `devel` è il luogo in cui vengono posizionati i “target” prima di essere installati. Ciò fornisce un utile ambiente di test e sviluppo che non richiede la fase di installazione.

- **install**: dopo aver creato localmente il/gli eseguibile/i di destinazione, lo/li si può installare con il comando `catkin_make install`. Deve essere eseguito dalla cartella dell'area di lavoro ROS. Dopodiché apparirà la cartella `install` nell'area di lavoro. Questa cartella conserva i file di destinazione dell'installazione. Quando lanciamo un eseguibile, esso viene preso dalla cartella di installazione.

È possibile riassumere il processo di build di un package e la creazione di nodi ROS con il seguente automa a stati finiti:



I comandi utilizzati per creare e compilare i package, verranno discussi nel prossimo paragrafo.

5.2 Strumenti di sviluppo

I command line tools, sono strumenti di sviluppo utili sia per navigare nel workspace di ROS, sia per avere informazioni riguardo package, messaggi, topic o servizi.

Poiché Catkin è stato menzionato in precedenza, vale la pena di riportare alcuni comandi utili per lavorare con il nostro workspace:

- *catkin_init_workspace* : per inizializzare lo spazio di lavoro e generare le sottocartelle principali
- *catkin_create_pkg* <nome_pkg> [dipendenze] : per creare un package catkin e specificare le eventuali dipendenze
- *catkin_make* : per compilare i package presenti nella cartella src
- *catkin_make install* : per installare i package

Tra i comandi più importanti di ROS abbiamo:

- **roscore**

Prima di eseguire qualsiasi nodo ROS, occorre digitare questo comando. È buona norma utilizzarlo sempre all'apertura del primo terminale.

Esso avvia tre diversi strumenti:

- Il *master*, che gestisce il servizio di registrazione e denominazione dei nodi
- Il *parameter server*, che contiene i parametri chiave/valore dei dati
- Il nodo *rosout*, che aggrega i messaggi di debug da tutti gli altri nodi

L'utilizzo del comando è molto semplice, basta digitare: *roscore*

```
fbj@fbj-VirtualBox:~$ roscore
... logging to /home/fbj/.ros/log/108f634c-729c-11eb-a503-1b35ad94a22f/roslaunch-fbj-
VirtualBox-6085.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://fbj-VirtualBox:35201/
ros_comm version 1.15.9

SUMMARY
=====

PARAMETERS
* /rostdistro: noetic
* /rosversion: 1.15.9

NODES

auto-starting new master
process[master]: started with pid [6095]
ROS_MASTER_URI=http://fbj-VirtualBox:11311/

setting /run_id to 108f634c-729c-11eb-a503-1b35ad94a22f
process[rosout-1]: started with pid [6105]
started core service [/rosout]
```

- **roscd**

Consente di eseguire il comando `cd` direttamente su un package, uno stack in base al nome, senza doverne conoscere il percorso specifico. È utilizzato come di seguito:

- *roscd <nome_package>*

- **roscd**

Il comando `roscd` esplora tutti gli aspetti di un nodo ROS. Tipici utilizzi di questo comando sono:

- *roscd list* : per avere una lista dei nodi attivi
- *roscd ping <nodo>* : per testare la connettività di un nodo

- *roscnode info nodo_1 [nodo_2 ...]* : per avere informazioni sui nodi specificati

```
fbj@fbj-VirtualBox:~$ roscnode info /turtlesim
-----
Node [/turtlesim]
Publications:
* /rosout [rosgraph_msgs/Log]
* /turtle1/color_sensor [turtlesim/Color]
* /turtle1/pose [turtlesim/Pose]

Subscriptions:
* /turtle1/cmd_vel [unknown type]

Services:
* /clear
* /kill
* /reset
* /spawn
* /turtle1/set_pen
* /turtle1/teleport_absolute
* /turtle1/teleport_relative
* /turtlesim/get_loggers
* /turtlesim/set_logger_level

contacting node http://fbj-VirtualBox:41379/ ...
Pid: 4354
Connections:
* topic: /rosout
  * to: /rosout
  * direction: outbound (43861 - 127.0.0.1:37252) [23]
  * transport: TCPROS
```

- **rostopic**

Il comando *rostopic* fornisce informazioni sui topic pubblicati e/o sottoscritti nel sistema. Questo comando è molto utile per elencare i topic, stampare i dati dei topic e pubblicare dati sui topic. Alcuni utilizzi sono:

- *rostopic info <topic>* : per avere informazioni su un determinato topic attivo
- *rostopic list* : per avere una lista dei topic pubblicati e/o sottoscritti

```
fbj@fbj-VirtualBox:~$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

- *rostopic type <topic>* : per vedere il tipo di messaggio pubblicato da un topic

- *rostopic echo <topic>* : per “ascoltare” e stampare i dati pubblicati da un topic
 - *rostopic pub <topic> <tipo_messaggio> <dati>*: per pubblicare dati su un topic
- **rosmg**
- Permette di visualizzare le informazioni sui tipi di messaggio ROS. Viene utilizzato spesso per l’introspezione del codice e per capire il messaggio com’è composto, oppure per capire quale tipo di dato aspettarsi, nel caso in cui si voglia lavorare con determinati messaggi. Viene utilizzato nel seguente modo:
- *rosmg show <tipo_messaggio>* : visualizza i campi in un tipo di messaggio ROS

```
fbj@fbj-VirtualBox:~$ rosmg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

- **rosparam**

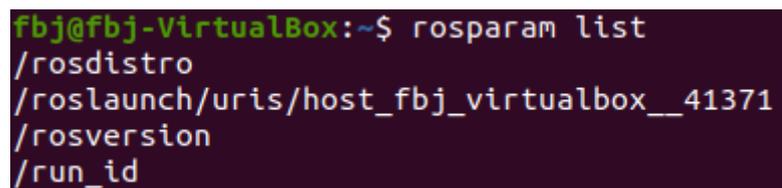
Consente di ottenere e impostare i valori del parameter server dal terminale utilizzando testo con codifica YAML. Un esempio di file YAML è il seguente:



```
1 rosdistro: 'noetic'
2
3 '
4 roslaunch:
5   uris:
6     host_fbj_virtualbox__42285: http://fbj-VirtualBox:42285/
7 rosversion: '1.15.9'
8
9 '
10 run_id: ef4017a4-720a-11eb-af79-19826cc60d67
11 turtlesim:
12   background_b: 200
13   background_g: 255
14   background_r: 0
```

Il comando rosparam viene tipicamente utilizzato nei seguenti modi:

- `rosparam get <parametro>` : stampa il valore del parametro
- `rosparam set <parametro> <valore>` : setta il parametro al valore specificato
- `rosparam list` : elenca tutti i parametri attivi



```
fbj@fbj-VirtualBox:~$ rosparam list
/rosdistro
/roslaunch/uris/host_fbj_virtualbox__41371
/rosversion
/run_id
```

- **rossrv**

È utilizzato soprattutto per visualizzare la definizione della struttura dei dati di un servizio (srv). Tipici utilizzi sono:

- *rossrv list* : elenca tutti i servizi
- *rossrv show <servizio>* : mostra la descrizione di un servizio

```
fbj@fbj-VirtualBox:~$ rossrv show turtlesim/Spawn
float32 x
float32 y
float32 theta
string name
---
string name
```

- **rosservice**

Implementa una varietà di comandi che consentono di scoprire quali servizi sono attualmente attivi e quali nodi li mettono a disposizione. Inoltre, permette di ottenere informazioni specifiche su un servizio, come il suo tipo, l'URI e gli argomenti, utili per poterlo chiamare da riga di comando.

Alcuni utilizzi sono:

- *rosservice list* : elenca tutti i servizi attivi
- *rosservice args <servizio>* : elenca gli argomenti necessari per un servizio
- *rosservice call <servizio> <argomenti>* : chiama un servizio per usufruirne
- *rosservice info <servizio>* : stampa le informazioni specifiche di un servizio

```
fbj@fbj-VirtualBox:~$ rosservice info /turtle1/teleport_relative
Node: /turtlesim
URI: rosrpc://fbj-VirtualBox:47675
Type: turtlesim/TeleportRelative
Args: linear angular
```

- **rosvag**

È uno strumento utilizzato per eseguire varie operazioni sui ROS bag, come la riproduzione e la registrazione di messaggi su determinati topic. Tipici utilizzi sono:

- *rosvag record* <topic1> [<topic2> ...] : registra il contenuto dei topic specificati
- *rosvag play* <bag > : riproduce il contenuto di un ROS bag

- **rosrun**

È uno strumento utilizzato per avviare un nodo ROS. Viene utilizzato nel seguente modo:

- *rosrun* <package> <nodo> : avvia il nodo specificato

- **roslaunch**

Permette di avviare una serie di nodi da un unico file di configurazione XML (con estensione .launch) e permette inoltre l'avvio di nodi su macchine remote. Il vantaggio di questo comando è che permette di avviare più nodi con un solo comando ed in aggiunta, se il master non è attivo, viene effettuata la chiamata a roscore. Poiché c'è il rischio di instabilità, dovuta alla creazione di più ROS Master, gli sviluppatori suggeriscono di non abusarne. Un tipico utilizzo di questo comando è:

- *roslaunch* <package> <file.launch> : esegue il file .launch specificato

Un esempio di file con estensione *.launch* è il seguente:

```
<launch>
  <group ns="turtlesim1">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <group ns="turtlesim2">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <node pkg="turtlesim" name="mimic" type="mimic">
    <remap from="input" to="turtlesim1/turtle1"/>
    <remap from="output" to="turtlesim2/turtle1"/>
  </node>
</launch>
```

Esistono anche tool grafici molto utili, come *rqt_graph*, che mostra un grafo delle interazioni tra nodi e topic, e *rqt_plot*, che traccia i dati numerici di un determinato topic nel tempo. Il grafo mostrato da *rqt_graph* è simile al seguente:



5.3 Nozioni base: C++ in ROS

Una libreria ROS è una raccolta di codice che facilita il lavoro del programmatore. La libreria **roscpp** è un'implementazione C++ di ROS, che consente ai programmatori di interfacciarsi rapidamente con topic, servizi, parametri ROS e molto altro. È la libreria più utilizzata ed è progettata per essere quella con le più alte prestazioni per ROS.

5.3.1 Inizializzazione, avvio ed arresto di un nodo

L'inizializzazione di un nodo roscpp avviene tramite una chiamata alla funzione `ros::init()`. Tale funzione fornisce gli argomenti dalla riga di comando a ROS, consente di dare un nome al nodo e specificare altre opzioni. Viene utilizzato come segue:

```
ros::init(argc, argv, <nome_nodo>, ros::init_options::<opzione>);
```

dove `argc`, `argv` sono gli argomenti presi dalla riga di comando.

Il modo più comune per avviare un nodo roscpp è creare un oggetto di tipo `ros::NodeHandle`, con la seguente istruzione:

```
ros::NodeHandle nh;
```

Quando viene creato il primo `ros::NodeHandle`, esso chiamerà `ros::start()`, e quando l'ultimo `ros::NodeHandle` verrà distrutto, chiamerà `ros::shutdown()`.

Per arrestare un nodo è possibile chiamare la funzione `ros::shutdown()`. Questo cancellerà tutti i topic aperti, le pubblicazioni, le chiamate di servizio e i server di servizio. Roscpp installa anche un gestore SIGINT, il quale rileva se l'utente digita "Ctrl + C" dalla riga di comando ed arresta automaticamente il nodo. Il metodo migliore per controllare se un nodo è arrestato correttamente è la funzione `ros::ok()`. Una volta che `ros::ok()` restituisce `false`, il nodo ha terminato l'arresto. Tale funzione viene di solito utilizzata in un ciclo while come segue:

```
while (ros::ok()) {  
    //codice  
}
```

5.3.2 Creazione di messaggi

La libreria `roscpp` prende i file `.msg` e genera per essi il codice sorgente C++. Grazie a questo nel codice della nostra applicazione ROS, possiamo creare un messaggio, utilizzando l'istruzione:

```
nome_pacchetto::tipo_messaggio
```

Inoltre, i file di intestazione (`.h`) vengono generati con lo stesso nome del nome del file `.msg`. Pertanto, se si vuole includere nel codice un messaggio di tipo `std_msgs/String` bisogna usare le seguenti istruzioni:

```
#include<std_msgs/String.h>

std_msgs::String stringa;
```

5.3.3 Pubblicare e sottoscrivere nei topic

Per pubblicare su un topic è necessario prima creare un `NodeHandle`, il quale con il metodo `advertise()` può creare un `Publisher`. Quest'ultimo è l'istanza che realmente pubblica su un topic e per farlo utilizza il metodo `publish()`. Immaginando di voler pubblicare un messaggio di tipo `std_msgs/String`, bisognerà utilizzare le seguenti istruzioni:

```
ros::NodeHandle nh;
ros::Publisher pub = nh.advertise<std_msgs::String>(<nome_topic>, <lungh_coda>);
std_msgs::String stringa;
stringa.data = "Hello world!";
pub.publish(stringa);
```

Il metodo `advertise()` prende in input il nome del topic su cui si vuole pubblicare e la lunghezza della coda, ovvero il numero massimo di messaggi accodati da consegnare ai sottoscrittori.

La sottoscrizione a un topic avviene anch'essa utilizzando un NodeHandle, con il quale si può creare un Subscriber utilizzando il metodo `subscribe()`. Per sottoscrivere un nodo ad un topic si usano le seguenti istruzioni:

```
ros::NodeHandle nh;  
ros::Subscriber sub = nh.subscribe(<nome_topic>, <lungh_coda>, <func_callback>);
```

La funzione `subscribe()` prende in input il nome del topic al quale sottoscrivere il nodo, una funzione di callback nella quale processa i messaggi ricevuti ed la lunghezza della coda dove mantiene i messaggi. La lunghezza della coda è molto importante, perché se arrivassero troppi messaggi velocemente, non passerebbero tutti nella funzione di callback e quindi molti andrebbero eliminati senza essere processati.

5.3.4 Creazione e chiamata di servizi

I servizi ROS sono definiti da file `.srv`, i quali contengono un messaggio di richiesta e un messaggio di risposta. `roscpp` converte questi file `.srv` in codice sorgente C++ e crea tre classi differenti: una per il servizio, una per i messaggi di richiesta (`Request`) e una per i messaggi di risposta (`Response`). La classe `Request` fornisce l'input al servizio, mentre la classe `Response` viene restituita al client come output del servizio. Il servizio, di per sé, è una funzione che ha per parametri una request e una response e dopo varie operazioni, restituisce un booleano (`true`, appena completato). Un servizio può essere creato in due modi differenti, ma per semplicità si continuerà ad utilizzare un NodeHandle. Un esempio è il seguente:

```
//Creazione di un servizio  
ros::NodeHandle nh;  
ros::ServiceServer servizio = nh.advertiseService(<nome_servizio>, <func_callback>);
```

La funzione `advertiseService()` ha come argomenti il nome del servizio e la funzione di callback invocata quando viene richiesto il servizio.

La funzione di callback prende in input il tipo della richiesta e della risposta forniti ad `advertiseService()` e restituisce un booleano (*true*, se la chiamata è riuscita e l'oggetto risposta è stato riempito con i dati necessari, *false* se la chiamata non è riuscita e l'oggetto risposta non verrà inviato al chiamante).

Per effettuare la chiamata ad un servizio si utilizza un `NodeHandle` per creare un `ServiceClient`.

```
//Creazione di un client per chiamare il servizio
ros::NodeHandle nh;
ros::ServiceClient client = nh.serviceClient<tipo_servizio>(<nome_servizio>, <persistente>);
```

La funzione `serviceClient()` ha come parametri il nome del servizio al quale si vuole fare richiesta ed un parametro booleano per avere o meno una connessione persistente con esso. Il `ServiceClient` effettuerà la chiamata al servizio tramite la funzione `call()`, la quale ha come parametro il nome del servizio da chiamare.

5.3.5 Callback e spinning

La libreria `roscpp` effettua vari lavori sui thread, ma non li espone mai alla nostra applicazione ROS. Questo vuol dire che senza un opportuno intervento dell'utente, le funzioni di callback presenti nell'applicazione non verranno mai chiamate. Per far sì che esse siano chiamate, ci sono i cosiddetti Spinning. Esistono principalmente due tipi di Spinning:

- Spinning a **thread singolo**

È la versione più semplice e più comune di spinning. Si può utilizzare in due modi diversi:

- ***ros::spin()***

```
ros::init(.....);  
ros::NodeHandle nh;  
ros::Subscriber sub = nh.subscribe(....);  
.....  
ros::spin();
```

Tutti i callback degli utenti verranno chiamati da `ros::spin()`, fin quando il nodo sarà attivo.

- ***ros::spinOnce()***

```
ros::init(.....)  
.....  
ros::Rate r(10); // 10 hz  
while (....)  
{  
    .....  
    ros::spinOnce();  
    r.sleep();  
}
```

Si imposta un `ros::Rate` per temporizzare la chiamata di `ros::spinOnce()`, la quale chiamerà tutti i callback in attesa di volta in volta.

- **Spinning multi-thread**

`roscpp` fornisce un supporto integrato per la chiamata di callback da più thread. Anche in questo caso abbiamo due opzioni:

- **`ros::MultiThreadedSpinner`**

```
ros::MultiThreadedSpinner spinner(4); // Usa 4 thread  
spinner.spin();
```

`MultiThreadedSpinner` è uno spinner “bloccante”, simile a `ros::spin()`. È possibile specificare un numero di thread nel suo costruttore, ma se non specificato (o impostato su 0), utilizza un thread per ogni core della CPU.

- **ros::AsyncSpinner**

```
ros::AsyncSpinner spinner(4); // Usa 4 thread  
spinner.start();
```

AsyncSpinner, invece di essere uno spinner “bloccante”, ha a disposizione le chiamate *start()* e *stop()*.

6. Casi di studio

In questo capitolo si vogliono presentare due casi di studio. Il primo riguarda la creazione di un messaggio ROS personalizzato e la creazione di un'architettura master-slave tra due macchine virtuali differenti, dove nella prima viene creato un nodo master, il quale pubblica il messaggio personalizzato su un topic, e nella seconda viene creato un nodo slave, che riceve i messaggi pubblicati su di esso. Il secondo riguarda una simulazione di un ambiente virtuale e di un robot che mappa tale ambiente, sfruttando l'algoritmo SLAM (Simultaneous Localization And Mapping) messo a disposizione dalla community ROS.

6.1 Architettura master-slave e creazione di messaggi

In ROS lo scambio di informazioni avviene attraverso messaggi. Questo permette di avere un'astrazione hardware dalla macchina su cui sono eseguiti i nodi. È possibile definire un tipo di messaggio personalizzato in file *.msg*, nella sottocartella *msg* presente nella cartella *src* di un package. Per questo caso di studio, è stato creato un package rinominato “caso_studio” ed un file “Messaggio.msg” contenente come unico dato una stringa, ovvero contenente la sola istruzione: *string messaggio*

Per creare un nuovo messaggio occorre solamente modificare il file *CMakeLists.txt*, aggiungendo alla sezione dedicata ai messaggi il nome del file, in questo caso “Messaggio.msg”. Infine ricompilare il workspace con il comando *catkin_make*.

Per creare un'architettura master-slave tra le due macchine virtuali e poter comunicare grazie allo stesso ROS Master, nei file *.bashrc* delle rispettive sono state inserite le informazioni riguardanti il *ROS_IP* ed il *ROS_MASTER_URI*.

Per quanto riguarda la macchina master:

```
export ROS_IP=192.168.1.135
export ROS_MASTER_URI=http://192.168.1.135:11311
```

Per la macchina slave, invece:

```
export ROS_IP=192.168.1.136
export ROS_MASTER_URI=http://192.168.1.135:11311
```

Come si può notare dalle istruzioni precedenti, per permettere la comunicazione tra macchine differenti, il `ROS_MASTER_URI` è lo stesso. Vengono poi creati dei nodi sulle due macchine, uno per la pubblicazione di messaggi ed un altro per la ricezione.

Sulla macchina master viene creato il *nodo_master*, il quale pubblica su un topic, denominato *topic_condiviso*, il messaggio creato precedentemente. Di seguito è riportato il codice C++:

```
#include "ros/ros.h"
#include "caso_studio/Messaggio.h"
#include <sstream>

int main(int argc, char **argv){
    ros::init(argc, argv, "nodo_master");
    ros::NodeHandle nh;
    ros::Publisher pub = nh.advertise<caso_studio::Messaggio>("topic_condiviso", 1000);
    ros::Rate loop_rate(1);
    int count = 1;
    while (ros::ok()){
        caso_studio::Messaggio messaggio;
        std::stringstream ss;
        ss << "Messaggio #" << count;
        messaggio.messaggio = ss.str();
        ROS_INFO("%s", messaggio.messaggio.c_str());
        pub.publish(messaggio);
        ros::spinOnce();
        loop_rate.sleep();
        ++count;
    }
    return 0;
}
```

Il *nodo_master* pubblicherà un messaggio ogni secondo sul *topic_condiviso*.

Sulla macchina slave viene creato il *nodo_slave*, il quale riceve i messaggi personalizzati dal topic denominato *topic_condiviso*. Di seguito è riportato il codice C++:

```
#include "ros/ros.h"
#include "caso_studio/Messaggio.h"

void callback(const caso_studio::Messaggio::ConstPtr& messaggio)
{
    ROS_INFO("Messaggio ricevuto: %s", messaggio->messaggio.c_str());
}

int main(int argc, char **argv){
    ros::init(argc, argv, "nodo_slave");
    ros::NodeHandle nh;
    ros::Subscriber sub = nh.subscribe("topic_condiviso", 1000, callback);
    ros::spin();
    return 0;
}
```

Il *nodo_slave* è sottoscritto al *topic_condiviso* e riceverà i messaggi pubblicati su di esso. La funzione di callback, in questo caso, ha il solo compito di stampare il messaggio ricevuto.

Ora non resta altro che far scambiare i messaggi tra le due macchine. Per prima cosa occorre avviare il ROS Master sulla macchina virtuale “master”, lanciando il comando *roscore*. In seguito avviamo il *nodo_slave* sulla macchina virtuale “slave” per attendere l’arrivo di messaggi ed il *nodo_master* sulla macchina “master” per iniziare a pubblicare messaggi.

Lo scenario sarà il seguente, lato master:

```
fbj@fbj-VirtualBox:~$ rosrund caso_studio pubblicatore
[ INFO] [1615322852.026659309]: Messaggio #1
[ INFO] [1615322853.042795716]: Messaggio #2
[ INFO] [1615322854.040742510]: Messaggio #3
[ INFO] [1615322855.027289719]: Messaggio #4
[ INFO] [1615322856.045068499]: Messaggio #5
[ INFO] [1615322857.029449633]: Messaggio #6
[ INFO] [1615322858.027711945]: Messaggio #7
[ INFO] [1615322859.027906599]: Messaggio #8
[ INFO] [1615322860.039723200]: Messaggio #9
[ INFO] [1615322861.027173182]: Messaggio #10
```

Lato slave, avremo:

```
utente@utente-VirtualBox:~/catkin_ws$ rosrund caso_studio sottoscrittore
[ INFO] [1615323115.123984257]: Messaggio ricevuto: Messaggio #2
[ INFO] [1615323116.122291142]: Messaggio ricevuto: Messaggio #3
[ INFO] [1615323117.121036338]: Messaggio ricevuto: Messaggio #4
[ INFO] [1615323118.119973769]: Messaggio ricevuto: Messaggio #5
[ INFO] [1615323119.117894436]: Messaggio ricevuto: Messaggio #6
[ INFO] [1615323120.125571556]: Messaggio ricevuto: Messaggio #7
[ INFO] [1615323121.115194108]: Messaggio ricevuto: Messaggio #8
[ INFO] [1615323122.119487961]: Messaggio ricevuto: Messaggio #9
[ INFO] [1615323123.113505187]: Messaggio ricevuto: Messaggio #10
```

È possibile inoltre, su una delle due macchine, visualizzare i nodi ed i topic attivi su entrambe le macchine, poiché il ROS Master è lo stesso:

```
utente@utente-VirtualBox:~/catkin_ws$ rosnod list
/nodo_master
/nodo_slave
/rosout
```

```
utente@utente-VirtualBox:~/catkin_ws$ rostopic list
/rosout
/rosout_agg
/topic_condiviso
```

Si possono notare i nodi di entrambe le macchine, *nodo_master* e *nodo_slave*, ed il topic utilizzato per lo scambio di messaggi, ovvero *topic_condiviso*.

6.2 Mappare un ambiente con SLAM

Di seguito si vuole riportare un semplice esempio dell'utilizzo di ROS. In particolare, l'utilizzo del simulatore Gazebo e del visualizzatore 2D/3D Rviz per mappare l'ambiente circostante al robot (viene utilizzato il TurtleBot3 Burger). Lo scopo di tale esempio è mostrare come la larga community di ROS metta a disposizione delle librerie già pronte per l'uso, tali che anche un programmatore con poca esperienza nel campo della robotica, possa svolgere attività notevoli come creare una mappa dell'ambiente circostante ad un robot.

Per iniziare, bisogna installare i package relativi a TurtleBot3 nella cartella *src* del catkin workspace. I comandi utilizzati sono i seguenti:

```
$ cd ~/catkin_ws/src/
```

```
$ git clone -b noetic-devel https://github.com/ROBOTIS-GIT/turtlebot3.git
```

```
$ git clone -b noetic-devel https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
```

```
$ git clone -b noetic-devel https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
```

```
$ cd ~/catkin_ws && catkin_make
```

Il package *turtlebot3_simulations*, contiene degli ambienti di simulazione già pronti per TurtleBot3, tra cui un appartamento, che verrà utilizzato per l'esempio.

Poiché esistono diversi tipi di TurtleBot3, occorre specificare quale si vuole utilizzare.

In questo esempio, si vuole utilizzare il TurtleBot3 Burger e possiamo fare ciò inserendo nel file *~/.bashrc* il seguente comando:

```
export TURTLEBOT3_MODEL=burger
```

Per mappare l'ambiente, utilizzeremo il package *turtlebot_slam*. Prima di fare ciò occorre installare il package *gmapping*:

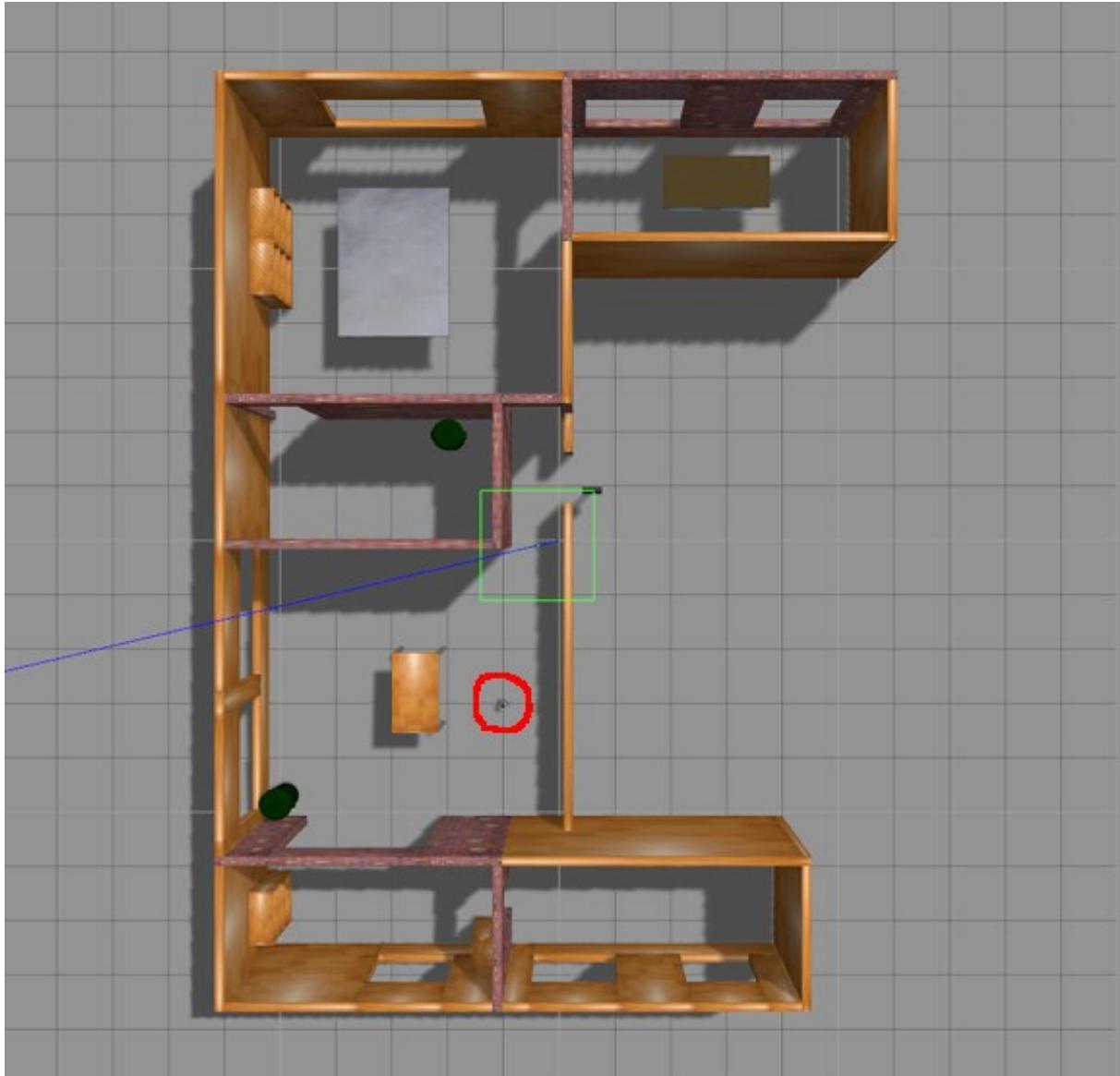
```
$ sudo apt install ros-noetic-slam-gmapping
```

Tale package contiene un nodo ROS chiamato *slam_gmapping* che fornisce l'algoritmo SLAM (Simultaneous Localization and Mapping) basato sullo scanner laser del robot. Utilizzando tale nodo, è possibile creare una mappa 2D dell'ambiente (ad es. la pianta di un edificio) dal laser e dai dati della posizione di un robot.

Per l'esempio discusso, occorre, innanzitutto, avviare la simulazione in Gazebo del Turtlebot3 (per l'esempio, come già detto, si userà "TurtleBot3 House" come ambiente virtuale). Per fare ciò si utilizza il comando:

```
$ roslaunch turtlebot3_gazebo turtlebot3_house.launch
```

Tale comando avvierà Gazebo con la simulazione di una casa e del TurtleBot3 all'interno di essa.

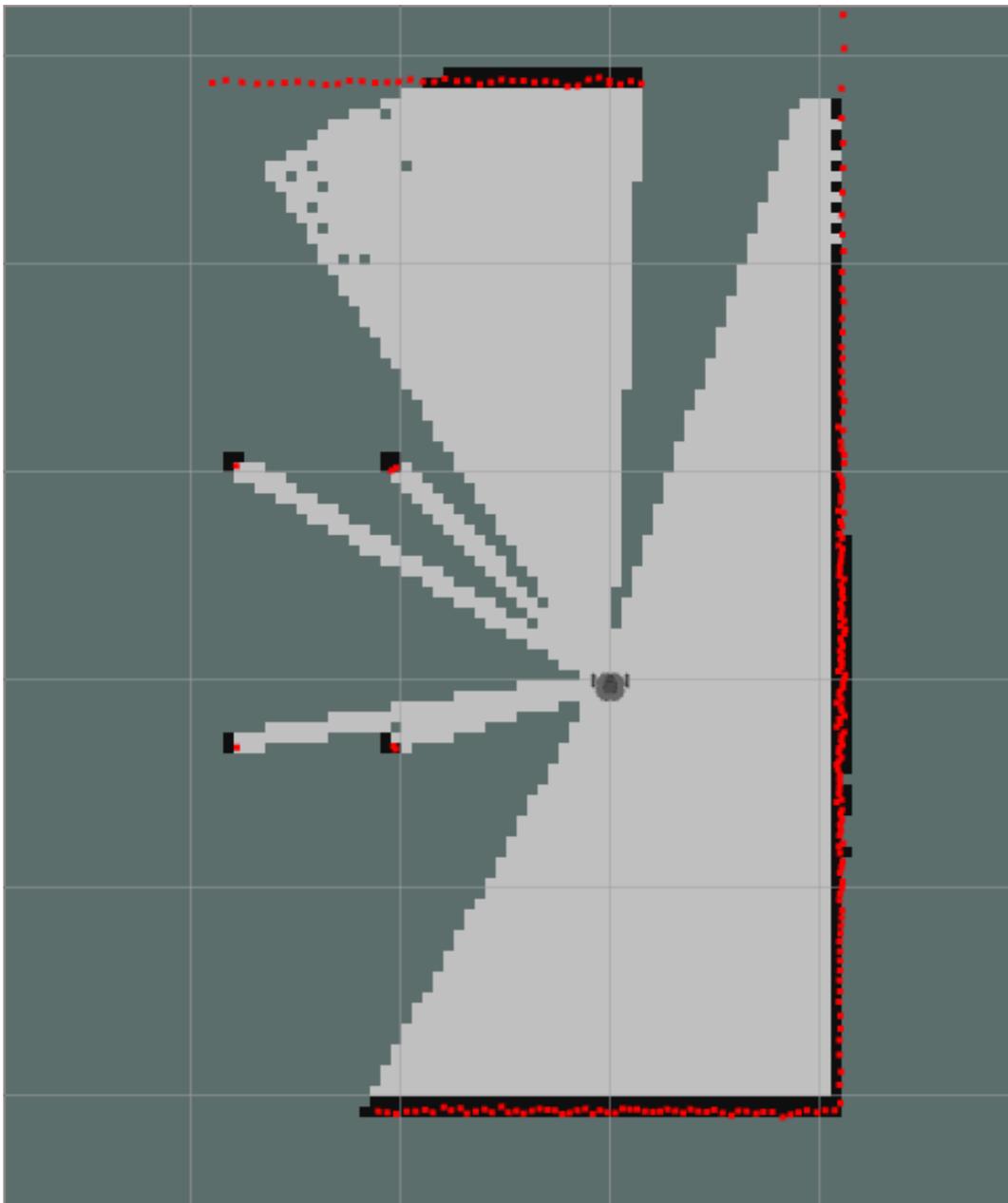


(In rosso è cerchiato il TurtleBot3)

Per iniziare a creare la mappa dell'ambiente circostante al robot, si utilizza il package *turtlebot3_slam* ed in questo caso, useremo il metodo "gmapping". In un nuovo terminale, si avvia Rviz per la visualizzazione 2D e la creazione della planimetria della casa, utilizzando il seguente comando:

```
$ roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
```

Possiamo quindi vedere la mappa disegnata dal robot usando ciò che è in grado di rilevare con il laser dalla sua posizione iniziale.



Possiamo distinguere così: il sensore laser (rosso), l'area esplorata e libera (grigio chiaro), l'area non ancora esplorata o troppo distante dal laser (grigio scuro) e gli ostacoli (nero).

È possibile, inoltre, configurare dei parametri di gmapping per ottimizzare le prestazioni SLAM, in base all'ambiente in cui il robot deve operare. Tali parametri sono definiti nel file:

```
turtlebot3_slam/config/gmapping_params.yaml
```

Uno dei parametri interessanti è *map_update_interval*, che scandisce gli intervalli con cui la mappa viene aggiornata. Più il valore di tale parametro è piccolo, più frequente è l'aggiornamento della mappa. Tuttavia, impostarlo su un valore troppo piccolo richiede una maggiore potenza di elaborazione per il calcolo della mappa.

Per completare la mappa, occorre far esplorare al nostro robot lo spazio circostante. Il package *turtlebot3* mette a disposizione il nodo *turtlebot3_teleop*, il quale ci permette di utilizzare la nostra tastiera per guidare TurtleBot3 nella direzione voluta. Per fare ciò, in un nuovo terminale, occorre lanciare il comando:

```
$ rosrun turtlebot3_teleop turtlebot3_teleop_key
```

Nel terminale verrà visualizzato il seguente schema:

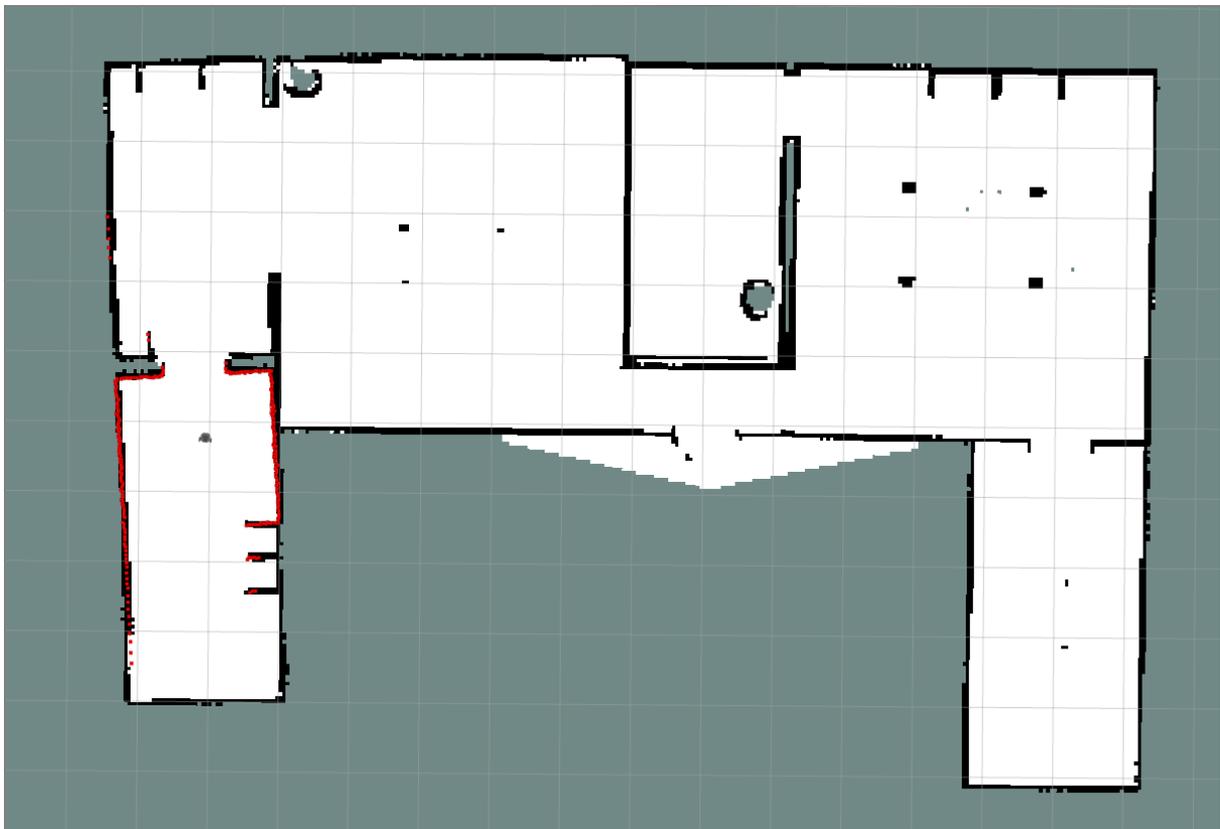
```
Control Your TurtleBot3!
-----
Moving around:
   w
 a   s   d
   x

w/x : increase/decrease linear velocity (Burger : ~ 0.22, Waffle and Waffle Pi :
~ 0.26)
a/d : increase/decrease angular velocity (Burger : ~ 2.84, Waffle and Waffle Pi
: ~ 1.82)

space key, s : force stop
```

Premendo i tasti *w* e *x* aumenta/diminuisce la velocità lineare (va avanti o indietro), *a* e *d* aumenta/diminuisce la velocità angolare (gira a destra o sinistra) ed *s* o *barra spaziatrice* le velocità vengono resettate a 0 (il robot si ferma).

Possiamo così guidare il robot ed esplorare il resto della casa presente nella simulazione.

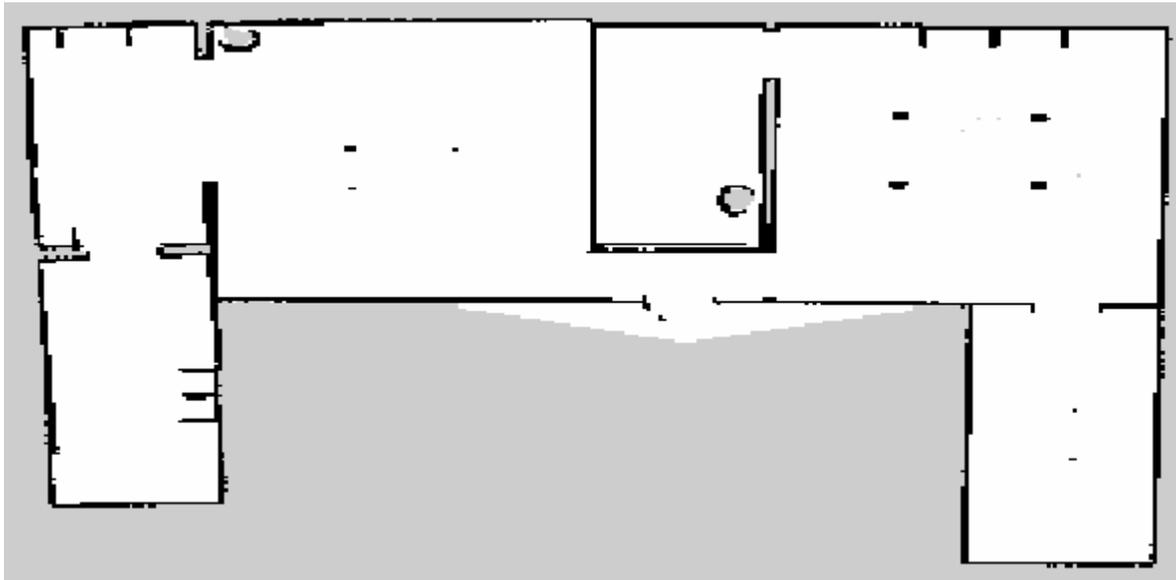


Dopo aver creato una mappa completa dell'area desiderata, ROS ci permette di salvare i dati relativi ad essa. Possiamo utilizzare il nodo *map_saver*, presente all'interno del pacchetto *map_server*, per creare file di mappe. In un nuovo terminale, utilizziamo il comando:

```
$ rosrun map_server map_saver
```

Questo creerà i file *map.pgm* e *map.yaml*, rispettivamente l'immagine della mappa ed un file contenente i parametri della mappa.

Il risultato della mappa finale sarà il seguente:



Si possono distinguere tre aree di colore diverso: l'area bianca è un'area libera, ovvero priva di collisioni, quella nera è inaccessibile, ovvero è presente un ostacolo, e quella grigia rappresenta l'area sconosciuta, ovvero quella non perlustrata dal robot.

Tale mappa può essere utilizzata in futuro per la navigazione di un robot reale in un ambiente simile a quello ricreato nella simulazione con Gazebo.

Conclusioni

Lo scopo della tesi è stato introdurre il lettore nel campo della robotica e presentare ROS come framework per lo sviluppo di applicazioni. Abbiamo visto come ROS comprenda caratteristiche tipiche di un sistema operativo, come l'astrazione dell'hardware sottostante, grazie alla comunicazione tra messaggi, elementi tipici di un middleware per la comunicazione tra processi/macchine differenti, grazie al Master ed al suo `ROS_MASTER_URI`, ed infine di un framework, grazie al build system Catkin ed agli strumenti di sviluppo per creare nuovi moduli software, ovvero nuovi nodi. Nel quarto capitolo vengono introdotti i concetti chiave per lo sviluppo di applicazioni di robotica con l'utilizzo di ROS: il build system di ROS, ovvero Catkin, il quale mira a rendere più semplice la creazione e l'esecuzione del codice ROS utilizzando strumenti e convenzioni per semplificare il processo, alcuni dei command line tools di ROS ed alcune delle funzioni base per sviluppare con ROS in C++, come la creazione di nodi, topic, servizi ed altro ancora. Nell'ultimo capitolo viene mostrato come il framework ROS permetta di creare un'architettura, in questo caso di tipo master-slave, e di far comunicare sistemi presenti su macchine differenti. Viene illustrata la creazione di due sistemi, ognuno su una macchina diversa, che comunicano attraverso la condivisione del cosiddetto `ROS_MASTER_URI`. Si vuole, inoltre, rafforzare il concetto di come ROS e la sua sempre più attiva e numerosa comunità, mettano a disposizione algoritmi standard e librerie già testate e funzionanti che permettono di facilitare molto il lavoro degli sviluppatori, anche alle prime armi, fornendo un semplice esempio di come creare la mappa di un ambiente virtuale, a partire dalla simulazione di un robot in tale ambiente. In conclusione, abbiamo visto la comunicazione e lo scambio di informazioni all'interno dell'architettura ROS, anche tra macchine differenti, ed il vantaggio di avere algoritmi standard, testati e funzionanti, forniti dalla larga community,

pronti per essere implementati nel codice dell'applicazione. L'utilizzo di ROS porta enormi vantaggi nello sviluppo di applicazioni robotiche, tuttavia, il fatto di non sostituire un vero e proprio sistema operativo, ma anzi di lavorarci ad un livello superiore, ed il fatto di funzionare principalmente su sistemi Linux limita fortemente il suo utilizzo. Per il futuro, la comunità ROS sta lavorando allo sviluppo della nuova versione ROS2, che oltre ad integrare il supporto real-time (assente in ROS1), tenterà di sbarcare definitivamente sui sistemi Windows e MacOS, con versioni stabili e non più sperimentali.

Bibliografia e sitografia

- [1] *A Gentle Introduction to ROS, CreateSpace Independent Publishing Platform, 2013* - Jason M. O’Kane
- [2] *Hands-On ROS for Robotics Programming: Program highly autonomous and AI-capable mobile robots powered by ROS - Packt Publishing - 2020*
- Bernardo Ronquillo Japón
- [3] *Learning Robotics using Python: Design, simulate, program, and prototype an autonomous mobile robot using ROS, OpenCV, PCL, and Python - 2nd Edition - Packt Publishing - 2018* – Lentin Joseph
- [4] *Mastering ROS for Robotics Programming: Design, build, and simulate complex robots using the Robot Operating System – 2nd Edition - Packt Publishing - 2018* - Lentin Joseph, Jonathan Cacace
- [5] *Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy - Apress - 2018* – Lentin Joseph
- [6] *Robotica Modellistica, pianificazione e controllo - 3^a ed. – McGrawHill - 2008* - Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, Giuseppe Oriolo
- [7] *International Federation of Robotics: <https://ifr.org>*
- [8] *ROS Wiki: <http://wiki.ros.org>*
- [9] *ROS: <https://www.ros.org>*
- [10] *TurtleBot3: <https://emanual.robotis.com>*