ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea in
Ingegneria Informatica

LA PIATTAFORMA ROS PER LO SVILUPPO DI APPLICAZIONI ROBOTICHE

Elaborata nel corso di: Sistemi Operativi

Tesi di Laurea di:
DAVIDE LEARDINI

Relatore:
Prof. ALESSANDRO RICCI

ANNO ACCADEMICO 2013-2014
SESSIONE III

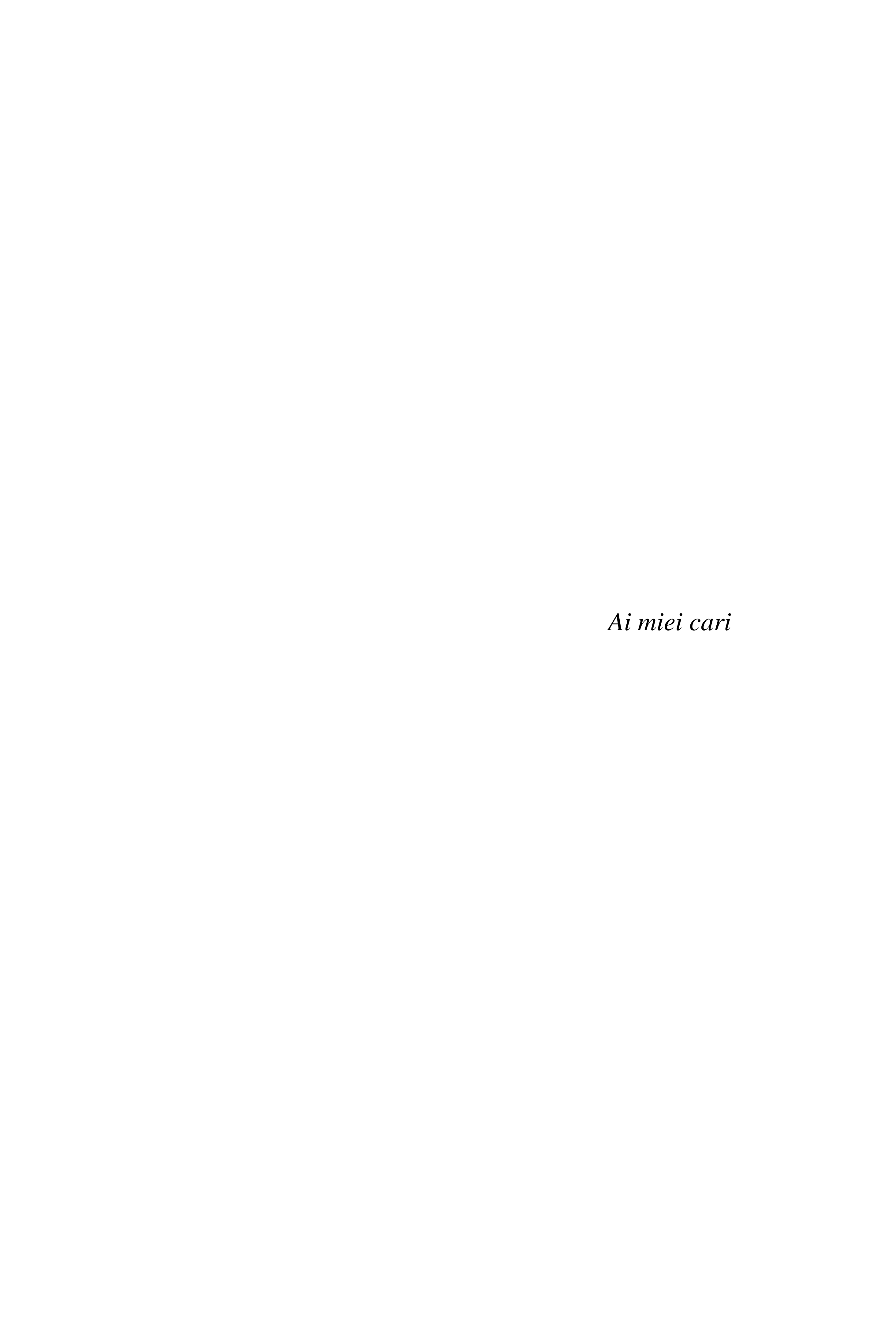
Parole chiave

ROS

Robot

Robotica

Piattaforma di Sviluppo



Ai miei cari

Indice

Prefazione **xi**

1 Introduzione alla Robotica **1**

1.1 Robotica, Breve Definizione 1

1.2 Robot 3

1.2.1 Corpo 4

1.2.2 Sensori 5

1.2.3 Effettori/Attuatori 5

1.2.4 Controllori 7

2 ROS, Robot Operating System **8**

2.1 Introduzione a ROS 8

2.2 Storia 9

2.3 ROS, Caratteristiche 10

2.4 ROS, Perchè Usarlo? 12

2.5 Architettura 15

2.5.1 Filesystem Level 16

2.5.1.1 Packages 16

2.5.1.2 Metapackages 17

2.5.1.3 Package Manifest 17

2.5.1.4 Stacks 17

2.5.1.5 Message (msg) Type . . . 18

2.5.1.6 Service (msg) Type . . . 19

2.5.2 Computation Graph Level 19
2.5.2.1 Nodi 20
2.5.2.2 Master 21
2.5.2.3 Messaggi 22
2.5.2.4 Topic 22
2.5.2.5 Servizi 28
2.5.2.6 Bag 32
2.5.3 Community Level 33
2.5.3.1 Distribuzioni 33
2.5.3.2 Repositories 34
2.5.3.3 ROS Wiki 34

3 Un Semplice Caso di Studio 35
3.1 Simulazione Robot Tramite Stage . . . 35
3.2 Implementiamo il Movimento via Topic . 44
3.3 Dalla Simulazione al Caso Reale 53

4 Conclusioni 54

Bibliografia e Sitografia 56

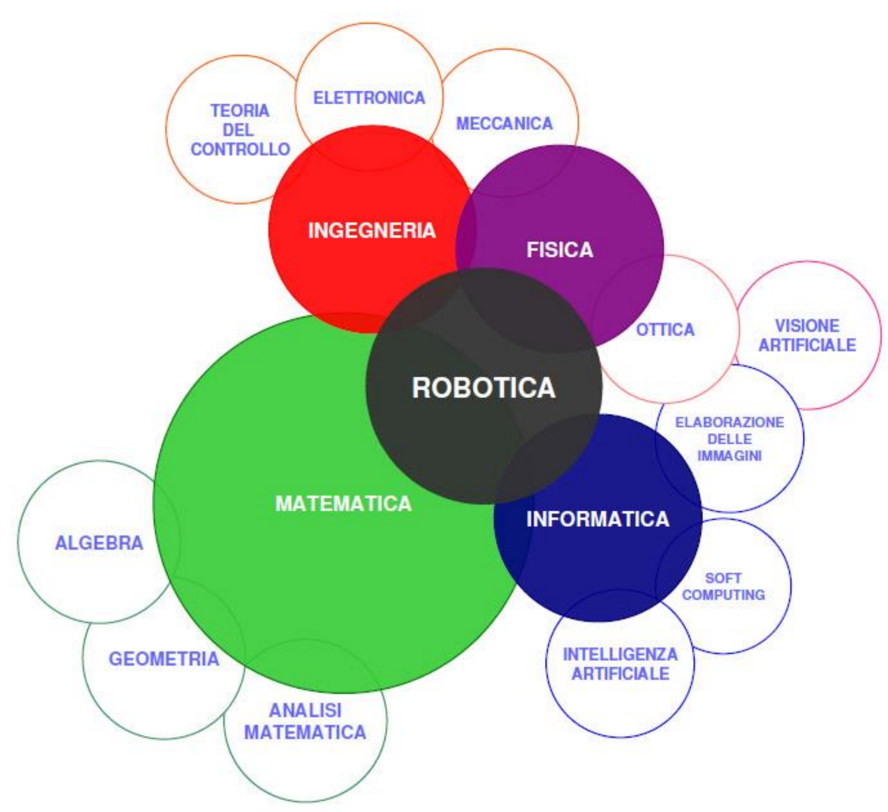
PREFAZIONE

Lo sviluppo hardware nel campo della robotica ha raggiunto negli ultimi anni livelli impressionanti ed è in continua crescita, e di pari passo si è espansa l'eterogeneità delle forme che può assumere, dalle tipologie basate su movimento a terra ai droni volanti, fino a forme più sofisticate di robot umanoidi che cercano di emularne il comportamento. Se da un lato ora possiamo disporre di hardware sempre più potente ed efficiente a costi sempre minori, dall'altro programmare il comportamento che un robot deve tenere nelle svariate circostanze in cui può imbattersi, nel poter portare a compimento il proprio obbiettivo, risulta essere sempre più complesso. Dopo una breve introduzione alla robotica e alle difficoltà che deve affrontare e una panoramica sui robot, cosa siano e come siano strutturati, fulcro della tesi sarà l'esposizione delle caratteristiche principali di ROS, Robot Operating System, come piattaforma di sviluppo software nel campo della robotica , e si concluderà con un semplice caso di studio in cui ne verrà messo in mostra concretamente l'utilizzo.

Capitolo 1
Introduzione alla Robotica

1.1 Robotica, Breve Definizione

La Robotica è la branca dell'ingegneria che si occupa della progettazione, dello sviluppo e del funzionamento dei robot, dei sistemi informatici per l'elaborazione dei dati provenienti dai sensori e del loro utilizzo per delineare il comportamento da tenere al fine di eseguire compiti specifici. Il termine fu coniato nel 1920 dallo scrittore - scienziato ceco Karel Capek e ha il significato di " lavoro forzato ", e in effetti il fine ultimo di un robot è quello di riprodurre in qualche forma il lavoro umano. Un Robot deve affrontare tutte le difficoltà provenienti dall'interazione col mondo reale. Una semplice salita può modificarne la velocità e la percezione della distanza percorsa, un leggero urto può condizionarne la traiettoria irrimediabilmente. Cosi, pur avendola delineata come branca dell'ingegneria, sono molteplici le discipline con cui deve confrontarsi. In primis vengono richieste conoscenze dei modelli matematici e della fisica, soprattutto per quanto concerne tutta la parte di acquisizione e elaborazione dei dati, essenziale per potersi rapportare con l'ambiente con cui si interagisce.



Successivamente l'informatica è la disciplina cardine per tutta la parte computazionale, e in primo luogo per quello che riguarda il controllo. Come deve reagire il robot ai dati sensoriali? Nel caso debba eseguire necessariamente piu azioni, quale deve essere l'ordine? Sono diverse le architetture di controllo sviluppate al fine di ricreare una sorta di " intelligenza " interna al robot, ed evidentemente il campo presenta una complessità non affrontabile dal singolo programmatore. Nel seguito della tesi verrà esposta una breve descrizione della struttura tipica di un robot, e si entrerà nel dettaglio del framework ROS come utile strumento nello sviluppo software e test di applicazioni robotiche.

1.2 Robot

Definire cosa è un robot e cosa non lo è può risultare a volte triviale. Con la parola “robot” si vuole indicare un sistema autonomo operante nel mondo fisico, quindi con una struttura materiale, che percepisce l’ambiente che lo circonda e che interagisce con esso, attuando procedure al fine di perseguire l’ obiettivo per cui è stato creato. Questa definizione contiene concetti molto importanti che distinguono quello che vogliamo considerare robot da una semplice macchina. Prima di tutto abbiamo chiarito che deve essere un sistema *autonomo*. Deve perciò poter operare sulla base di decisioni prese per proprio conto: pur potendo ricevere input da persone, non devono essere controllati da essi al fine di raggiungere il proprio scopo. Un robot deve avere una *struttura fisica* con cui interagire col mondo esterno e affrontarne le sfide. Un robot che esiste solo all’interno di un computer è solo una simulazione che non affronta la vera complessità del mondo reale. E’ necessario che *percepisca* l’ambiente in cui si ritrova ad operare. Deve montare di conseguenza sensori di vario tipo al fine di reperire informazioni riguardo il mondo circostante, al fine di poter rispondere in maniera adeguata agli ostacoli catturandone la presenza nel proprio percorso. Differentemente a un robot simulato siamo noi a dover fornire informazioni e conoscenze, e lavora di conseguenza utilizzando dati ricavati “per magia”. Un robot deve poter *agire in risposta* agli input derivanti dai sensori, montando quelli che vengono definiti attuatori e affrontare il mondo circostante nelle numerevoli forme in cui può presentarsi. Infine è necessario che tutto quello che compia abbiamo come fine ultimo il perseguire un obiettivo. Qui si entra nel campo dell’intelligenza di un robot: avere un sistema autonomo in grado di captare gli stimoli provenienti dall’esterno ma che agisse in maniera del tutto casuale sarebbe inutile.

Quello che lo differezia è il poter svolgere un task predefinito , il tutto in maniera autonoma, la cui difficoltà può essere semplice come “ non bloccarti” o di grado sempre più complesso come “percorri ogni singoli centimetro di una stanza nella maniera più efficente e veloce possibile”. Abbiamo finora descritto quali caratteristiche generali debba avere una macchina al fine di poter essere considerata un robot, ma per poter esistere deve presentare quattro elementi fisici che vanno poi a rispecchiarsi nel relativo hardware: un corpo fisico, sensori, effettori / attuatori e infine i controllori. Andremo a descriverli brevemente a seguito.

1.2.1 Corpo

L’essere dotato di un corpo fisico è ovviamente necessario per esistere e interagire nel mondo reale. Tuttavia lo porta a dover rispettare tutte le leggi fisiche che esso impone, non potendo disporre dei vantaggi presenti in simulazione. Non può cambiare le proprie dimensioni, così come non può essere in due posti contemporaneamente. Per potersi muovere deve sfruttare i propri attuatori, e aumentare o diminuire di velocità richiede tempo ed è influenzato dalla topologia del mondo circostante. Inoltre disporre di un corpo implica la probabile eventualità di incorrere in ostacoli e quindi di dover essere coscenti di cosa vi è attorno. Infine il corpo impone al robot tutta una serie di limitazioni, derivanti dalla forma e struttura del corpo stesso, che vanno a influire sui movimenti, l’interazione con oggetti o altri robot, il percepire l’esterno etc.

1.2.2 Sensori

Si identificano sotto il nome di sensori tutti i dispositivi fisici montati su un robot in grado di ottenere informazioni dall'ambiente circostante e permettere al robot stesso di essere " coscente " di ciò che lo circonda. Il tipo di sensore montato dipende dal tipo di informazione che viene richiesta al fine di poter portare a compimento il proprio task. Si dividono essenzialmente in sensori passivi e attivi. I primi si limitano a strumenti che rilevano la proprietà fisica da misurare, mentre quelli attivi generano un segnale e sfruttano la relativa interazione con l'ambiente circostante come proprietà da misurare. L'uso di entrambi permette quindi al robot di essere a conoscenza del proprio stato, ossia di una descrizione di sè stesso in un certo dato momento, nel qual caso si parla di *stato interno* , e dell'ambiente circostante in cui deve operare, e qui si parla di *stato esterno.* Avere a disposizione uno stato interno risulta importante in quanto dà la possibilità di memorizzare informazioni sul mondo in un modello o rappresentazione. Una rappresentazione, sia essa di uno stato interno o del mondo esterno, può assumere una grande varietà di forme e può essere utilizzata in svariati modi da un robot. Ad esempio se l'obiettivo fosse percorrere un labirinto, un robot potrebbe creare una rappresentazione del percorso fatto basata sulle distanze esatte percorse, oppure sulla sequenza di mosse eseguite in determinati punti (gira a destra al primo angolo). Più è complessa la rappresentazione, maggiore è il costo in memoria utilizzata.

1.2.3 Effettori / Attuatori

Per effettore si intende ogni dispositivo fisico che ha un impatto sull'ambiente reale su cui il robot deve operare. Sono essenzialmente

la controparte meccanica di quello che in biologia rappresenta una gamba, un braccio, un dito. Il meccanismo che rende possibile ad un effettore di compiere un’azione viene definito attuatore, e tale termine include dispositivi di diffenti tipi di tecnologie:

- **Motori elettrici**: i più comuni attuatori adoperati per facilità d’uso, affidabilità e varietà di grandezze e forme in cui sono reperibili, forniti di elettricità al giusto voltaggio permettono di convertire l’energia elettromagnetica in energia cinetica producendo movimento.

- **Dispositivi idraulici**: consistono in attuatori il cui funzionamento è basato sulla pressione di un fluido. Alla sua variazione, l’attuatore inizia a muoversi. Potenti e precisi, hanno il difetto di avere dimensioni generalmente elevate e di essere potenzialmente pericolosi se non bene progettati.

- **Dispositivi pneumatici**: hanno le stesse caratteristiche di funzionamento dei dispositivi idraulici, ma anzichè essere basati sulla pressione di un fluido utilizzano la pressione dell’aria.

- **Materiali foto-reattivi**: materiali che producono un lavoro fisico in risposta all’ammontare della quantità di luce attorno a loro. Il lavoro generato e generalmente molto contenuto, e vengono utilizzati di conseguenza in robot di piccole dimensioni.

- **Materiali chimico-reattivi**: materiali che reagiscono in presenza di una sostanza chimica, come ad esempio particolari fibbre in grado di accorciarsi in presenza di soluzioni acidi e allungarsi con soluzioni basiche.

- **Materiali reattivi termicamente**: tutti i materiali che hanno una reazione di qualche tipo in presenza di un cambiamento di temperatura nell'ambiente circostante.

- **Materiali piezoelettrici**: materiali costituiti generalemente da cristalli in grado di creare cariche elettriche se premuti.

1.2.4 Controllori

Sono la componente hardware e software che permettono al robot di essere autonomo, processando gli input provenienti dai sensori o altre informazioni come quelle contenute in memoria, decidere quale azione intraprendere e di conseguenza quali effettori mettere in movimento. Costituiscono quindi la controparte robotica di quello che è essenzialmente il cervello e il sistema nervoso. Generalmente su un robot coesistono più controllori che operano su gruppi differenti di sensori e effettori in modo tale da poter , nello stesso istante, processare più informazioni e prendere decisioni differenti sulle azione che le varie parti del robot devono intraprendere.

.

Capitolo 2
ROS, Robot Operating System

2.1 Introduzione a ROS

Abbiamo brevemento introdotto in cosa consista un robot, quali siano le sue caratteritische fisiche, il suo hardware, ma d'altronde per poter funzionare, prendere decisioni, deve disporre di un'intelligenza artificiale costituita dal software. Di fronte al continuo progesso della tecnologia nell'ambito della robotica e ai crescenti obiettivi sempre più audaci che essa si pone, lo sviluppo in ambito software è divenuto non meno difficoltoso e non triviale. Programmare un robot richiede codice su più livelli, da software più basilare lato driver , a forme molto complesse dedite a ricreare una forma di percezione del mondo circostante, fino a un più alto livello in cui si va a ricreare il comportamento voluto, ossia che il robot abbia un'intelligenza artificiale interna in grado di reagire agli input derivanti dai sensori secondo la metodologia di controllo più adatta al caso specifico, Deliberative Control, Reactive Control, Behaviour-Based Control per citarne alcune. Programmare un robot richiede abbandonare la sicurezza dello sviluppo software classico e immergersi nel mondo fisico e nella sua casualità, nel mondo asincrono dove la presenza di un minimo ostacolo può deviare la traiettoria, dove l'unica consapevolezza di ciò che ci circonda è data dai sensori e un minimo errore può avere grandi ripercussioni, dove le risorse in termini di memoria, processori, energia di alimentazione sono limitate. Programmare un robot è una sfida che va ben oltre alle capacità del

singolo ricercatore. D'innzanzi alle sopra citate difficoltà sono molteplici i framework realizzati dalle varie comunità di ricercatori per ovviare ai problemi dello sviluppo software lato robotica, contenerne la complessità e facilitare la rapidità di rilascio di prototipi per il testing. Un framework viene progettato con lo scopo di aiutare lo sviluppo di un determinato tipo di software, ne consegue che non esista "il miglior framework " ma che ognuno sia il frutto di tradeoff e priorità decise nel proprio ciclo di vita per far fronte a determinati problemi. Nel seguito analizzeremo nel dettaglio ROS, progettato per fronteggiare lo sviluppo dei cosiddetti " large-scale service robot ", orientato a una programmazione parallela e general purpose.

2.2 Storia

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

subito il suo sviluppo nel corso degli anni grazie al contributo della comunità di ricerca mondiale ed è divenuto uno dei suoi punti di forza.

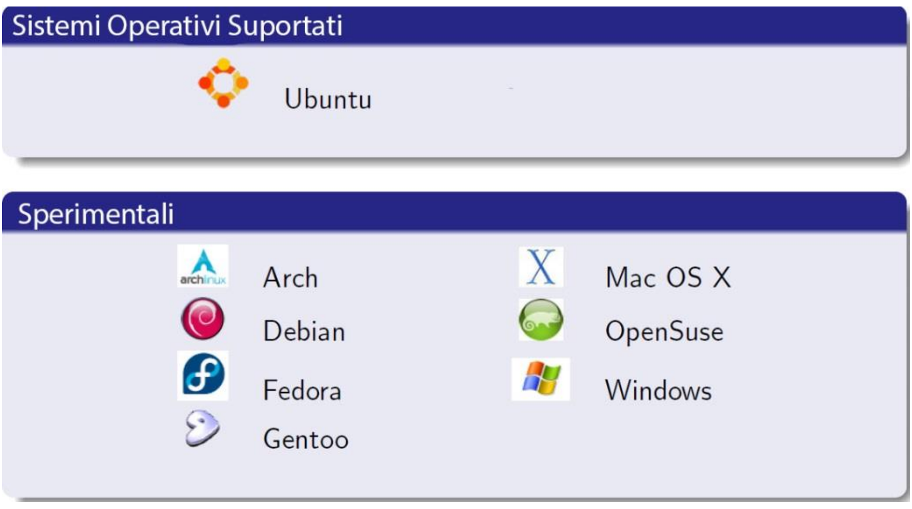
2.3 ROS, Caratteristiche

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

What Makes the Difference…?

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

ROS opera essenzialmente su piattaforme Unix-Based, anche se sono numerevoli le piattaforme sperimentali.



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

ROS client libraries

Main client libraries:

- Python
- C++
- Lisp

Experimental client libraries:

- Java (with Android support)
- Lua

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

2.4 ROS, Perchè Usarlo?

Ci si potrebbe probabilmente chiedere perchè usare ROS e quali vantaggi avremmo per il nostro software. D’altronde per quanto la curva di apprendimento possa essere bassa, diventare abili nell’uso di un nuovo framework richiede sempre tempo, tempo ben investito o meno? In realtà sono numerevoli le situazioni in cui l’uso di ROS

apporta un effettivo grande contributo, situazioni che in cui ci si scontra di continuo nella programmazione di software per robot.

- **Sistemi Distribuiti**

  Numerevoli robot ai giorni nostri contano su software basato sulla cooperazione di molti processi spesso nemmeno risiedenti sulla stessa macchina, ma diffusi su più computer differenti. Alcuni possono integrare in sè più computer, ognuno dei quali controlla un gruppo di sensori o attuatori. Altri possono operare con un singolo computer dovente gestire operazioni complesse con alto rischio di incorrere in errori, e seguendo un algoritmo divide et impera suddividere il proprio software in più parti cooperanti tra loro per raggiungere l'obiettivo comune. Uno stesso task può essere raggiunto con la coordinazione di più robot che devono agire all'unisono per risolvere il problema. In tutti i suddetti casi coesiste la necessità di poter comunicare tra processi, siano essi insiti nello stesso robot o suddivisi tra differenti, e il middleware garantito da ROS permette ciò tramite due meccanismi primari, topic per una comunicazione asincrona, servizi per la tipologia sincrona, ognuno dei quali verrà discusso piu avanti.

- **Standardizzazione e riuso del codice**

  Il rapido progresso della robotica ha portato allo sviluppo di algoritmi sempre più complessi ed efficaci per ovviare ai tipici problemi quali ad esempio la navigazione , il mapping di una zona, la scelta del percorso migliore. Di pari passo avere un'implementazione stabile di tali algoritmi senza dover ogni

volta si cambi sistema dover integrare con altro codice o reimplementare risulterebbe utile. Ciò è garantito da ROS e dalla grande comunità di ricerca che lo sostiene alle spalle. ROS garantisce packages debuggati e testati di molti algoritmi usati in robotica. Inoltre la sua interfaccia di comunicazione attraverso scambio di messaggi sta divenendo una standard de facto nell'interoperabilità del software in ambito robotica, e interfacce ROS per le ultime versioni hardware o algoritmi usciti sono spesso disponibili. Tutto ciò permette agli sviluppatori di concentrarsi nella creazione di un'applicazione sulle idee e il testing e non perdere tempo con codice integrativo, perchè le librerie necessarie risultano già disponibili, stabili e debuggate.

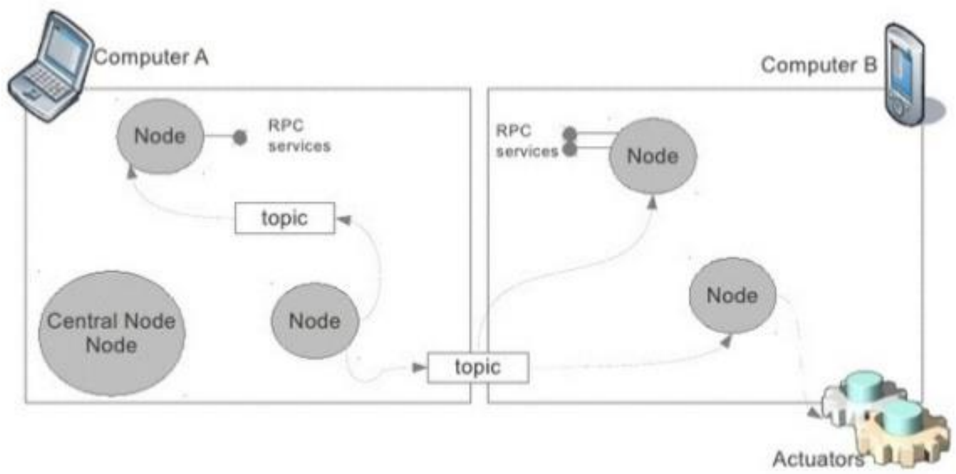
- **Testing**

Il testing su software per applicazioni nell'ambito della robotica richiede più tempo ed è più incline alla presenza di errori. Per di più non sempre è disponibile fisicamente il robot e anche se fosse accessibile spesso il processo di testing risulta macchinoso e lento. ROS permette invece 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 riutilizarli 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

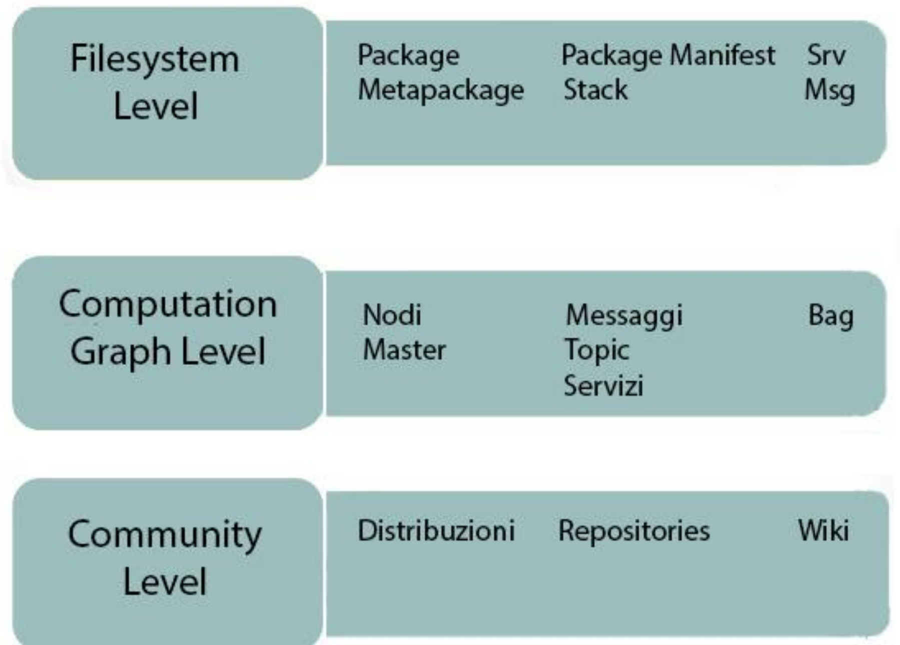
integrazioni o modifiche al codice perchè tutte le modalità forniscono uno stesso interfacciamento.

2.5 Architettura

ROS è basato su un'architettura a grafo dove il processamento avviene nei nodi, che comunicano tra loro attraverso lo scambio di messaggi in maniera asincrona attraverso l'uso di topic ai quali possono sottoscrivere e/o sui quali possono pubblicare, e in maniera sincrona con la chiamata di servizi, simili a RPC.



Strutturalmente ROS si sviluppa su 3 livelli concettuali, Filesystem Level, Computational Level e Community Level, di cui andremo ad esaminare gli elementi costitutivi e il loro ruolo nell'architettura .



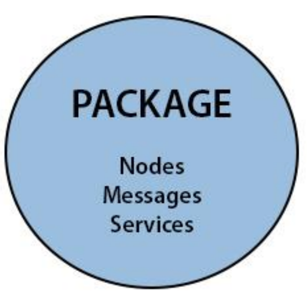
2.5.1 Filesystem Level

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

2.5.1.1 Packages

In ROS il software viene organizzato in packages, che in sostanza costituiscono dei moduli. Ogni package può contenere un nodo ROS, un file di configurazione, un dataset, una libreria, software di terze parti etc. Lo scopo dei packages è ottenere una maniera efficace per il

riuso del codice, venendo strutturati in modo tale da risultare funzionali ma non troppo pesanti e difficili da utilizzare.



2.5.1.2 Metapackages

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

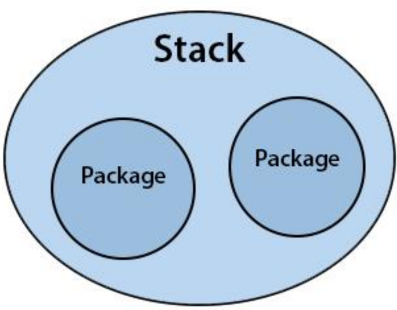
2.5.1.3 Package Manifest

Fornisce tutti i metadati riguardanti un package, nome, versione descrizione, licenza, dipendenze.

2.5.1.4 Stacks

Sono collezioni di packages con il comune scopo di svolgere un determinato task. Così a titolo di esempio il “ Navigation Stack “ è lo stack contenente tutti i package necessari al movimento del robot. Ogni Stack ha associata una versione e può dichiarare dipendenze da

altri Stack. Sono in ROS il meccanismo primario per la distribuzione del software.



2.5.1.5 Message (msg) Type

Indica il tipo di dato del messaggio, che determina di conseguenza il contenuto del messaggio stesso. E' strutturato come una lista di campi, uno per linea, ognuno dei quali è definito dalla concatenazione di un tipo di dato built-in (come bool, string, float64 etc) e nome del campo. Come esempio si riporta la struttura del messaggio geometry_msgs/Twist, che verrà utilizzato più volte nell'esempio applicativo a fine tesi:

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

Sia "linear" che "angular" risultano essere campi composti il cui tipo di dato è geometry_msgs/Vector3. In generale un campo composto è semplicemente la combinazione di uno o più sotto-campi, ognuno dei quali può essere a sua volta un altro campo composto o un campo semplice con tipo di dato built-in.

2.5.1.6 Service (msg) Type

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

2.5.2 Computation Graph Level

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

Computation Graph Level

Nodi
Master

Messaggi
Topic
Servizi

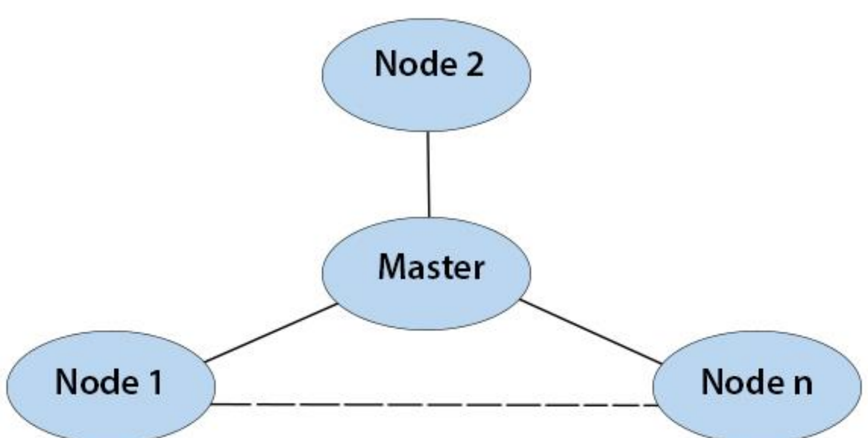
Bag

2.5.2.1 Nodi

Un nodo è un processo che compie una qualsiasi attività computazionale all’interno del sistema ROS. Essendo ROS progettato per essere modulare e fine-grained, tipicamente un sistema basato su di esso comprende numerosi nodi, e in tal contesto sono interpretabili come moduli software, ognuno dei quali incaricato di gestire un aspetto del comportamento del robot, come ad esempio la parte decisionale, il movimento, l’azionamento dei motori etc. Un sistema il cui carico computazionale venga ripartito tra i vari nodi di cui è costituito ha innanzitutto il vantaggio di una maggiore tolleranza agli errori, potendo gestire il crash del singolo nodo. La complessità del codice è ridotta se confrontata coi sistemi monolitici e i dettagli implementativi sono nascosti in quanti i singoli nodi offrono un’interfaccia composta da una API minimale. Ogni nodo in esecuzione dispone di quello che viene definito “ graph resource name “, un nome che lo identifica unicamente al resto del sistema, e di un tipo che semplifica il processo di indirizzamento di un nodo eseguibile all’interno del filesystem. Tutti i tipi sono trattati come “ package resource names”, costituiti dalla concatenazione del nome del package e del file eseguibile del nodo.

2.5.2.2 Master

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



2.5.2.3 Messaggi

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

2.5.2.4 Topic

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

Master. Il master ha altresì un well-known URI accessibile da tutti i nodi. La procedura sequenziale che permette a due nodi della rete di poter scambiare messaggi attraverso un topic si riassume nei seguenti passaggi:

1. Il publisher si registra al Master attraverso XMLRPC. Invia informazioni riguardo a ciò che andrà a pubblicare, tipo dei messaggi, nome del topic e il proprio URI.

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

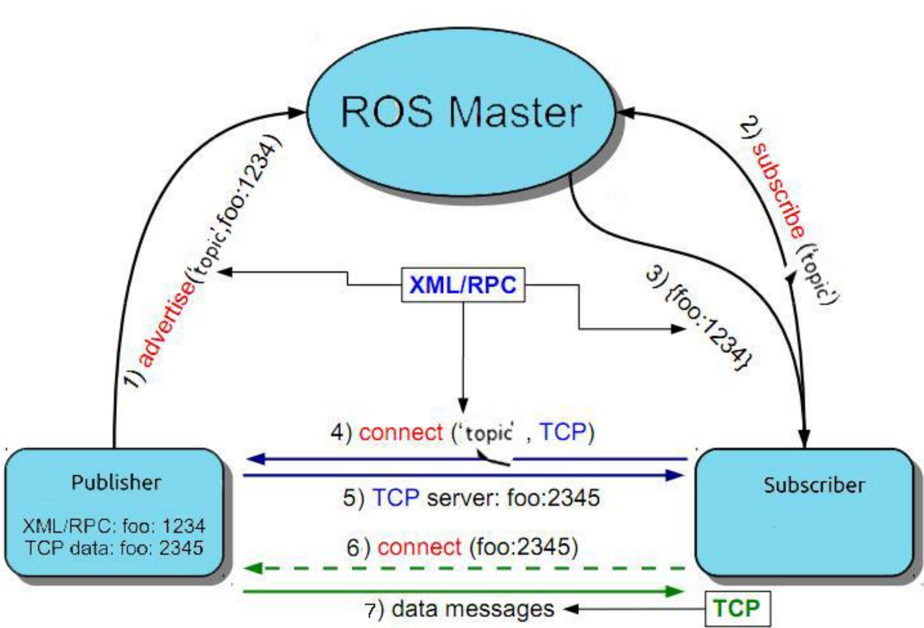
Il Master mantiene una tabella in cui risiedono tutti i dati sia per i subscriber che per i publisher.

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

5. Il publisher invia al subscriber la configurazione adatta al tipo di protocollo di trasporto scelto. Nel caso TCP ad esempio indirizzo IP e numero di porta.

6. Il subscriber si connette separatamente al publisher secondo il protocollo scelto.

7. I successivi messaggi pubblicati dal publisher verranno inviati al subscriber attraverso il canale aperto. Stabilita la connessione non è più richiesto il coinvolgimento del Master che in nessun momento è coinvolto nel flusso di dati scambiati nel topic.



Il lavoro di pubblicare messaggi è eseguito da un oggetto della classe ros::Publisher, che viene così creato

```
ros::Publisher pub = node_handle.advertise<message_type>(
topic_name, queue_size);
```

dove :

- Node_handle è un oggetto della classe ros::NodeHandle, una cui istanza generalmente viene creata a inizio programma.
- Message_type è il tipo del messaggio che vogliamo pubblicare
- Topic_name è il nome del topic su cui pubblicare.
- Queue_size è la grandezza della coda dei messaggi in uscita.

Per pubblicare il messaggio è sufficiente poi utilizzare il metodo publish(msg) dell'oggetto ros::Publisher. Invocare la funzione publish() serializza il messaggio in un buffer, il quale successivamente viene messo in coda. Tale coda viene processata il prima possibile in maniera asincrona da uno dei thread interni di roscpp, il package contenente le librerie per l'interfacciamento coi topic e service. Il messaggio serializzato contenuto è spostato, dal thread , per ogni subscriber connesso al topic in una coda ad esso associata. Il numero di messaggi che questo secondo set di code può contenere prima che i vecchi siano scartati è definito dal parametro queue_size della funzione advertise() di cui abbiamo parlato in precedenza. Infine i dati serializzati sono inviati sulle connessioni degli subscriber.

Diametralmente dall'altro lato della comunicazione avremo un nodo subscriber. La differenza importante rispetto a un publisher consiste nel fatto che il nodo subscriber non è a conoscenza di quando il messaggio effettivamente arriverà. Per fronteggiare ciò, va implementato il codice che risponda ai messaggi in entrata in una funzione di callback, che verrà chiamata per ogni messaggio in arrivo. Tipicamente una funzione di callback ha la seguente signature:

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

. . .

}
```

I parametri "package_name" e "type_name" sono gli stessi del publisher, in quanto si riferiscono alla classe del messaggio per il topic a cui miriamo sottoscrivere. Il corpo della funzione di callback ha accesso a tutti i campi dei messaggi in arrivo, e può decidere liberamente se utilizzare, scartare o memorizzare i dati contenuti. Il ritorno della chiamata è void in quanto è ROS a gestire la chiamata alla funzione. I Messaggi in arrivo vengono messi in una coda fino a quando ROS ha la possibilità di eseguire la callback. La grandezza della coda viene definita nella costruzione del subscriber, che andiamo ora ad analizzare.

Per creare un ros::Subscriber si utilizza la seguente signature

```
ros::Subscriber sub = node_handle.subscribe(topic_name,

queue_size, pointer_to_callback_function);
```

dove:

- Topic_name è il nome del topic in format stringa al quale vogliamo sottoscrivere

- Queue_size è appunto l'integer che va a definire la grandezza dell coda in cui alloggiare i messaggi non ancora processati. In caso di arrivo di un nuovo messaggio e coda piena, il più vecchio viene scartato.

- Pointer_to_callback_function è il puntatore alla funzione di callback

Non è menzionato il tipo del messaggio in quanto il compilatore lo recupera dal tipo di dato del puntatore a callback che gli viene fornito.ROS esegue le chiamate di callback solo quando lato user viene esplicitamente detto di farlo. Ciò avviene utilizzando due differenti funzioni:

- ros::spinOnce() ordina a ROS di eseguire tutte le callback pendenti da tutte le sottoscrizioni del nodo e ritorna il controllo .
- ros::spin() ordina a ROS di attendere e processare callback finchè il nodo risulta attivo, entrando in un loop simile a quello che si otterrebbe da

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

In conclusione, usare i topic risulta molto efficiente in situazioni in cui si ha un flusso continuo di dati, come quelli derivanti dai sensori o parametri relativi allo stato del robot.

2.5.2.5 Servizi

I servizi rappresentano il mezzo di comunicazione sincrono secondo una semantica di tipo request / reply, implementando in ROS funzionalità di RPC. Andranno cosi a definirsi nella rete nodi che svolgeranno funzione di service provider e nodi client. Esistono funzioni per verificare la presenza di un service provider nella rete . Il nodo client invierà dei dati che prenderanno il nome di request a un nodo server, aspettando la risposta di quest'ultimo. Il nodo server una volta ricevuta, processerà la request del client a cui inviere come risposta dei dati col nome di response. Un nodo client che voglia usufruire di un servizio inizializza un oggetto di tipo ros::ServiceClient che ha il compito di inoltrare la chiamata, e la cui signature è

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

dove:

- Service_type è il service data type utilizzato nella comunicazione

- Service_name è il nome passato come stringa del servizio che si vuole chiamare

Successivamente si creano gli oggetti Request e Response

```
package_name::service_type::Request

package_name::service_type::Response
```

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

La chiamata al servizio

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

effettua infine il lavoro di localizzare il nodo server, trasmettere la Request, attendere l'arrivo della risposta e magazzinarne i dati ricevuti nei campi dell'oggetto Response creato in precedenza. Il ritorno è di tipo booleano indicandoci se la chiamata ha avuto successo o meno. Lato server, come accadeva per i subscriber, ogni servizio che il nodo offre deve essere associato a una funzione di callback con signature

```
bool function_name(

package_name::service_type::Request &req),

package_name::service_type::Response &resp)

){

...

}
```

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

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

```
ros::ServiceServer server = node_handle.advertiseService(

service_name,

pointer_to_callback_function

);
```

dove:

- Service_name è il nome in formato stringa del servizio che si va ad offrire.

- Pointer_to_callback_function è il puntatore alla callback prima definita

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

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

1. Registrazione del servizio nel Master.

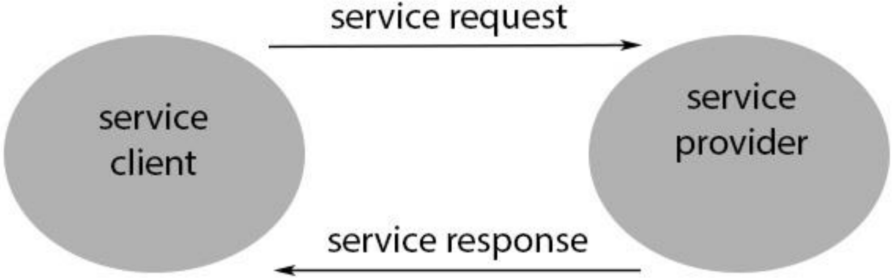
2. Richiesta di un determinato servizio al Master da parte di un client.

3. Creazione di una connessione TCP/IP al servizio da parte del Client.

4. Scambio tra Client e servizio di un Connection Header, il quale contiene importanti metadata sulla connessione che andrà a crearsi.

5. Invio del messaggio di request serializzato da parte del Client.

6. Invio del messaggio di response serializzato da parte del servizio.



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

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

2.5.2.6 Bag

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

2.5.3 Community Level

Comprende tutte le risorse che permettono a comunità separate di ricercatori di poter interagire scambiando software.

2.5.3.1 Distribuzioni

Sono collezioni di stacks versionati pronti all'installazione. Fino ad ora sono state otto le distribuzioni rilasciate, delle quali l'ultima è ROS Indigo Igloo.

ROS Hydro Medusa
Septeber 4, 2013

ROS Indigo Igloo
July 22, 2014

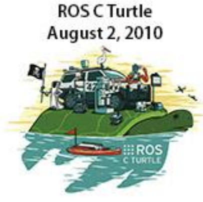
ROS Electric Emys
August 30, 2011

ROS Fuerte Turtle
April 23, 2012

ROS Groovy Galapagos
December 31, 2012

ROS box Turtle
March 2, 2010

ROS C Turtle
August 2, 2010



Ros Diamondback
March 2,2011

2.5.3.2 Repositories

Per ROS è stato scelto un modello di repository federato del codice: anzichè utilizzare un medesimo server centrale in cui depositare il codice, ogni gruppo di ricercatori può creare il proprio repository di ROS sui propri server mantenendo il pieno controllo e tutti i diritti sul codice ivi depositato. Decidendo poi di renderlo pubblico, otterrebbero tutti i riconoscimenti e i crediti per i propri conseguimenti, e i feedback tecnici utili al miglioramento come in ogni progetto software open-source. Questa metodologia di condivione del codice ha enfatizzato e massimizzato la partecipazione della comunità internazionale al progetto, rendendo ROS tra i framework piu utilizzati a livello mondiali nell'ambito dello sviluppo di applicazioni per robot, dalle più semplice ai grandi sistemi di automazione industriali.

2.5.3.3 ROS Wiki

E il forum princiale per documentarsi su ROS. Dopo una veloce registrazione, si può contribuire alla crescita della comunità scrivendo tutorial, offrendo aiuto, correzioni, update.

Capitolo 3
Un Semplice Caso di Studio

3.1 Simulazione Robot Tramite Stage

In seguito verrà proposto un esempio applicativo di utilizzo di topic per trasmettere informazioni e permettere il movimento casuale di un robot in un ambiente con ostacoli. Verrà utilizzato a tal proposito Stage. Stage è un tool di simulazione che permette di simulare robot muniti di sensori in una mappa in due dimensioni. Essendo stato progettato nell'ambito della ricerca per i sistemi multi-agente autonomi, fornisce, dei molti dispositivi emulati, un modello computazionale piuttosto semplice. Tuttavia pur non simulandoli con grande fedeltà, sono numerosi i sensori che permette di utilizzare a bordo dei robot, laser, bouncer, dispositivi a infrarossi fra questi. Stage è implementato attraverso lo Stage Stack, il quale contiene il nodo Stageros che fornisce le funzionalità del simulatore utilizzando ROS. Ricrea l'ambiente definito in un file di estensione .world, che contiene tutti i dettagli implementativi su sensori utilizzati, struttura del robot, composizione della mappa e ostacoli. Il codice del .world file che verrà utilizzato al fine della simulazione è il seguente:

```
define block model
(
  size [0.5 0.5 0.5]
  gui_nose 0
)
```

```
define robot_laser ranger
(
   Sensor
  (
   range [ 0.0  5.0 ]
   fov 40.25
   samples 1081
   )
  color "green"
  size [ 10 10 20 ]
)

define robot position
(
  size [0.35 0.35 0.25]
  origin [-0.05 0 0 0]
  gui_nose 1
  drive "diff"
  robot_laser(pose [ 0.050 0.000 0 0.000 ])
)

define floorplan model
(
  color "gray30"
  boundary 1
  gui_nose 0
  gui_grid 0
  gui_outline 0
  gripper_return 0
  fiducial_return 0
```

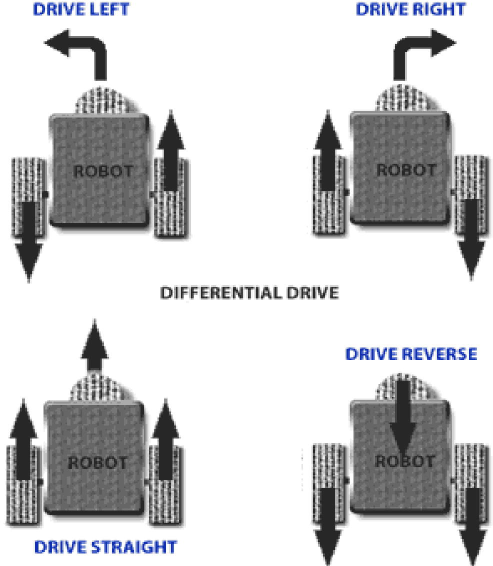
```
  laser_return 1
)

resolution 0.02
interval_sim 100

window
(
  size [ 745.000 448.000 ]
  show_data 1
  rotate [ 0.000 -1.560 ]
  scale 28.806
)

floorplan
(
  name "test_map"
  bitmap "autolab.png"
  size [54.0 58.7 0.5]
  pose [ 0 0  0 90.000 ]
)

robot ( pose [ 5 -5 0 180.000 ] name "bot" color "red" )
```

Senza analizzare nel dettaglio il codice in ogni singola riga, viene caricata la mappa “autolab.png” e posizionato nel punto voluto il robot, descritto da:

```
define robot position
(
  size [0.35 0.35 0.25]
```

```
 origin [-0.05 0 0 0]
 gui_nose 1
 drive "diff"
 robot_laser(pose [ 0.050 0.000 0 0.000 ])
)
```

Sarà quindi un differential wheeled robot, ossia la classica tipologia che monta per il movimento due ruote, ciascuna su uno dei due lati del corpo, permettendo alla base mobile di curvare variando la velocità relativa di rotazione delle singole ruote.



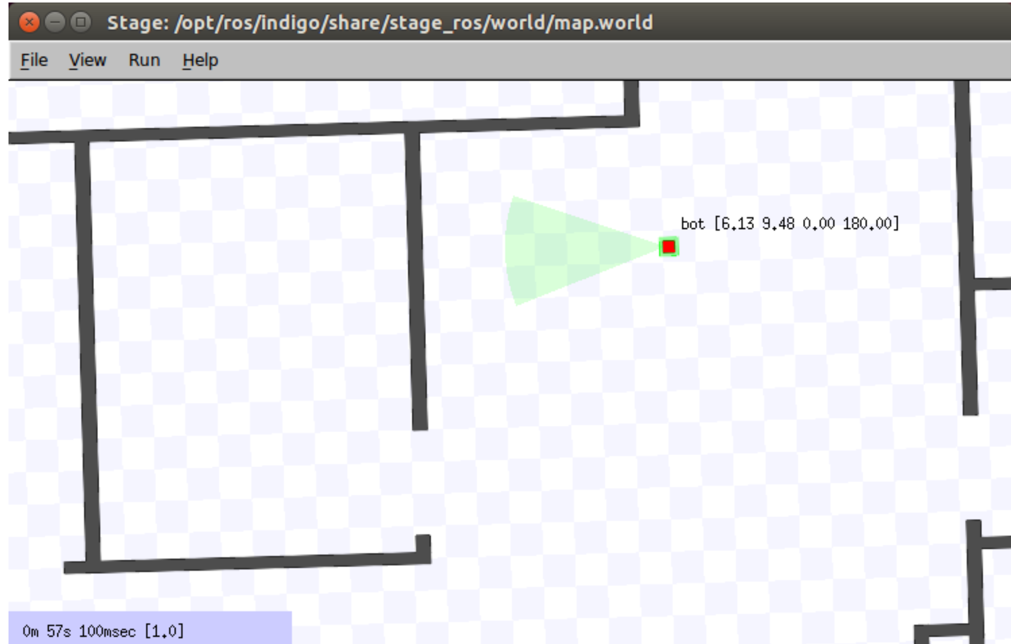
Inoltre monterà in testa un laser di 5m di range per individuare la presenza di ostacoli onde evitarli, le cui caratteristiche sono descritte da:

```
define robot_laser ranger
(
  Sensor
 (
  range [ 0.0  5.0 ]
  fov 40.25
  samples 1081
  )
 color "green"
 size [ 10 10 20 ]
)
```

Avviando il simulatore tramite riga di comando

```
rosrun stage_ros stageros $(rospack find stage_ros)/world/map.world
```

si aprirà la nostra mappa con al centro il robot emulato come in figura:



In realtà quello che accade a livello di codice è la creazione del nodo Stageros, che a sua volta sottoscrive a/ pubblica su diversi topic, visualizzabili attraverso il comando

```
rostopic list
```

che mostra la seguente lista:

```
ubuntu-tesi@ubuntutesi-VirtualBox: ~
ubuntu-tesi@ubuntutesi-VirtualBox:~$ rostopic list
/base_pose_ground_truth
/base_scan
/clock
/cmd_vel
/odom
/rosout
/rosout_agg
/tf
ubuntu-tesi@ubuntutesi-VirtualBox:~$
```

Nel dettaglio quello che Stageros esegue a livello di topic è:

- sottoscrivere il topic cmd_vel da cui riceverà informazioni di comando per la velocità lineare e angolare del robot sottoforma di messaggi di tipo geometry_msgs/Twist di struttura:

```
geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
        float64 x
        float64 y
        float64 z
```

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

- Pubblicare informazioni odometriche sullo stato del robot sui topic /odom e /base_pose_ground_truth attraverso messaggi di tipo nav_msgs/Odometry con struttura:

```
Header header
  uint32 seq
  time stamp
  string frame_id
  string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
```

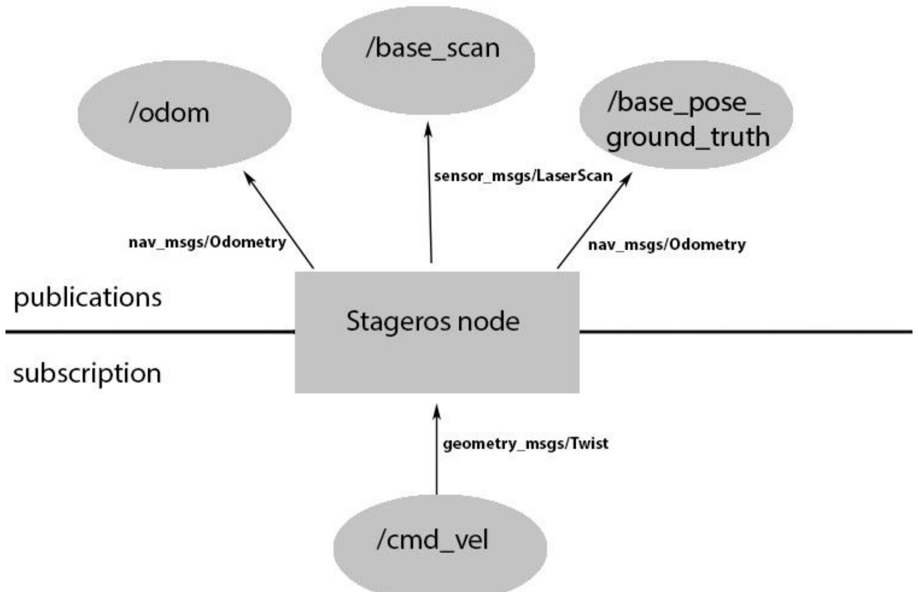
```
geometry_msgs/Twist twist
  geometry_msgs/Vector3 linear
    float64 x
    float64 y
    float64 z
  geometry_msgs/Vector3 angular
    float64 x
    float64 y
    float64 z
float64[36] covariance
```

- Pubblicare i dati ottenuti dalle scansioni del laser sul topic base_scan attraverso messaggi di tipo sensor_msgs/LaserScan con struttura :

```
Header header
float32 angle_min         # angolo iniziale di scan [rad]
float32 angle_max         # angolo finale di scan [rad]
float32 angle_increment   # distanza angolare [rad]
float32 time_increment    # tempo tra le misurazioni [s]
float32 scan_time         # tempo tra ogni scan [s]
float32 range_min         # range minimo [m]
float32 range_max         # range massimo [m]
float32[] ranges          # range dei dati [m]
```

```
float32[] intensities    # intensità del dato
```

Ne consegue che Stageros presenterà la seguente configurazione:



Andremo in seguito a creare un nodo che permetterà al robot simulato di muoversi e un nodo che pubblicherà semplicemente le informazioni odometriche di base, velocità lineare e velocità angolare, sul topic /rosout, lo "standard output" di ROS per visualizzare messaggi di logging su console.

3.2 Implementiamo il Movimento via Topic

Affinchè il robot simulato possa muoversi nella mappa in cui è stato posizionato, avendo cura di evitare gli ostacoli, è necessario instaurare una comunicazione tra il nodo stageros e un nodo che andremo a creare (e che chiameremo per comodità Walker) che si sviluppi nel seguente modo:

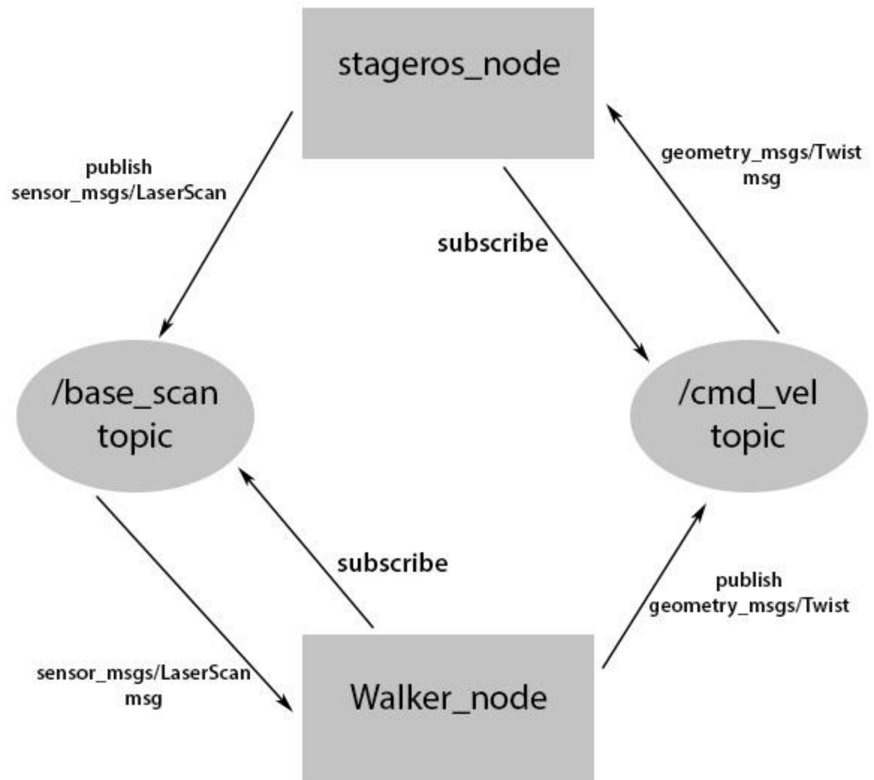
1. Walker sottoscrive il topic /base_scan.

2. Stageros pubblica i dati metrici provenienti dalle scansioni laser sul topic /base_scan sottoforma di messaggio sensor_msgs/LaserScan

3. Walker riceve i dati, li processa e ricava velocità lineare e angolare da impartire

4. Walker pubblica velocità lineare e angolare sul topic /cmd_vel (al quale Stageros è sottoscritto) sottoforma di messaggio geometry_msgs/Twist.

5. Stageros riceve le informazioni e il robot in simulazione inizializza il movimento.



Viene proposta ora di seguito il codice del nodo Walker, che verrà poi commentato nelle sue parti:

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "sensor_msgs/LaserScan.h"
#include <cstdlib>
#include <ctime>
#define FORWARD_SPEED 1
#define ROTATE_SPEED 1.4
#define REVERSAL_ROTATE_SPEED -1.4
```

```
class Walker {

 public:

  Walker(ros::NodeHandle& nh) {
   commandPub=nh.advertise<geometry_msgs::Twist>("cmd_vel",1);
   laserSub=nh.subscribe("base_scan",1,&Walker::commandCallback,
   this);
  };

 void move(double linearVel, double angularVel) {

   geometry_msgs::Twist msg;
   msg.linear.x = linearVel;
   msg.angular.z = angularVel;
   commandPub.publish(msg);

  };



 void commandCallback(const sensor_msgs::LaserScan::ConstPtr&
        msg) {

          float closestRange = msg->ranges[0];
           for( int x = 0; x< msg->ranges.size(); x++){
             if(msg->ranges[x] < closestRange){
               closestRange              =              msg->ranges[x];
```

```
          }
        }
        ROS_INFO_STREAM("Range: " << closestRange);
        if (closestRange < 2){
          srand(time(NULL));
          int random;
          random = rand() % 2 + 1;
          if (random > 1){
             move(0, ROTATE_SPEED);
          }else{
            move(0, -REVERSAL_ROTATE_SPEED);
          }
        }else{
          move(FORWARD_SPEED, 0);
          return;
        }
    };

    ros::Publisher commandPub;
    ros::Subscriber laserSub;

};

int main(int argc, char **argv) {

  ros::init(argc, argv, "Walker");
  ros::NodeHandle n;
  Walker walker(n);
  ros::spin();
```

```
 return 0;

}
```

Partiamo analizzando il main. Dopo aver, attraverso l'invocazione di

```
 ros::init(argc, argv, "Walker");
 ros::NodeHandle n;
```

inizializzato il nodo e assegnatogli il nome "Walker" (che deve essere univoco all'interno di ROS) , creiamo l'oggetto "Walker" che, come si evidenzia dal costruttore

```
 public:

  Walker(ros::NodeHandle& nh) {
   commandPub=nh.advertise<geometry_msgs::Twist>("cmd_vel",1);
   laserSub=nh.subscribe("base_scan",1,&Walker::commandCallback,
   this);
  };
```

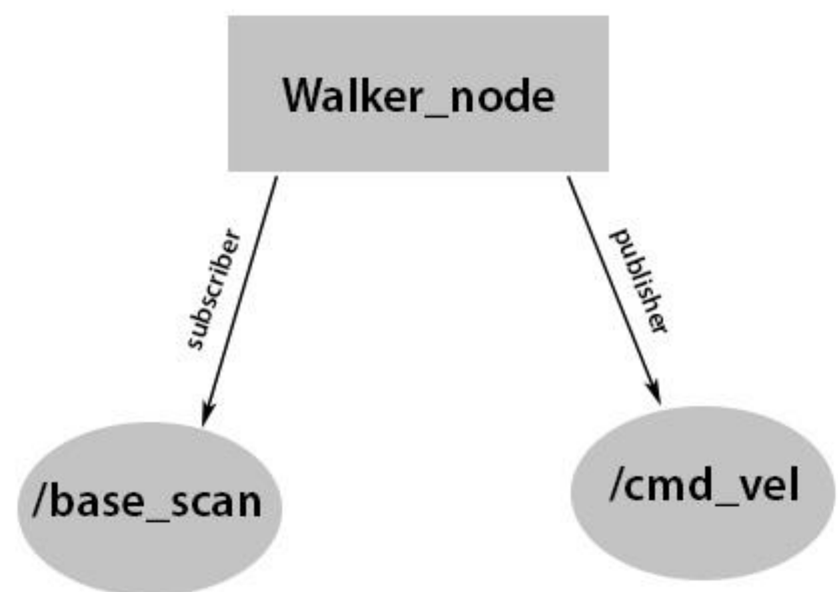
Esegue due chiamate di funzione:

```
commandPub=nh.advertise<geometry_msgs::Twist>("cmd_vel",1);
```

Connette il nostro nodo al ROS Master indicando che pubblicherà messaggi di tipo geometry_msg/Twist nel topic /cmd_vel e che avrà come grandezza della coda dei messaggi in uscita un solo messaggio, garantendoci che verrà pubblicato sempre l'ultimo messaggio generato.

```
laserSub=nh.subscribe("base_scan",1,&Walker::commandCallback,
   this);
```

Sottoscrive il nostro nodo al topic /base_scan e imposta come funzione di callback il nostro metodo di classe commandCallback passato come puntatore, che verrà chiamata ogniqualvolta arriverà un nuovo messaggio da quel topic. A questo punto il nostro nodo è sottoscritto al topic /base_scan dove arriveranno le informazioni relative al puntatore laser, e pubblica i comandi di velocità nel topic /cmd_vel.



Invocando

```
ros::spin();
```

Mandiamo il nostro nodo in un loop infinito. Fino allo shutdown del nodo, ordiniamo a ROS di attendere e processare tutte le callback da tutti i nodi sottoscritti, nel nostro caso /base_scan. Quindi ad ogni

nuovo messaggio in arrivo, verrà lanciata il nostro metodo di callback di cui il codice:

```
void commandCallback(const sensor_msgs::LaserScan::ConstPtr&
        msg) {

          float closestRange = msg->ranges[0];
           for( int x = 0; x< msg->ranges.size(); x++){
             if(msg->ranges[x] < closestRange){
               closestRange = msg->ranges[x];
             }
          }
          ROS_INFO_STREAM("Range: " << closestRange);
          if (closestRange < 2){
            srand(time(NULL));
            int random;
            random = rand() % 2 + 1;
            if (random > 1){
               move(0, ROTATE_SPEED);
            }else{
              move(0, -REVERSAL_ROTATE_SPEED);
            }
         }else{
          move(FORWARD_SPEED, 0);
          return;
        }
    };
```

La nostra callback eseguirà un ciclo for dove analizzerà il messaggio LaserScan, in particolar modo analizzerà ogni punto della scansione e

memorizzerà in una variabile quello di distanza inferiore. Dopodichè se tale valore risulterà inferiore a una certa soglia, verrà impartito il comando di ruotare casualmente a destra o sinistra, in caso contrario di continuare ad avanzare dritto. I comandi per la rotazione o il proseguimento sono inoltrati attraverso la funzione move() di cui il codice:

```
void move(double linearVel, double angularVel) {

  geometry_msgs::Twist msg;
  msg.linear.x = linearVel;
  msg.angular.z = angularVel;
  commandPub.publish(msg);

};
```

Tale funzione compila un messaggio di tipo geometry_msgs/Twist con informazioni riguardo la velocità lineare o angolare a seconda che si voglia far proseguire diritto o far curvare il robot simulato, e lo pubblica sul topic /cmd_vel dove Stage_ros è in ascolto. Il risultato finale sarà il movimento randomico all'interno della mappa del nostro robot simulato, che proseguirà dritto innanzi a se fino a quando il laser non rileverà un ostacolo in un range inferiore ai due metri , dopodichè casualmente ruoterà a destra o sinistra finchè il laser accerterà che non ci siano ostacoli inferiori a tale soglia e infine proseguirà il cammino.

Stage: /opt/ros/indigo/share/stage_ros/world/map.world

File View Run Help

0m 31s 000msec [1.0]

Stage: /opt/ros/indigo/share/stage_ros/world/map.world

File View Run Help

1m 55s 700msec [1.0] [PAUSED]

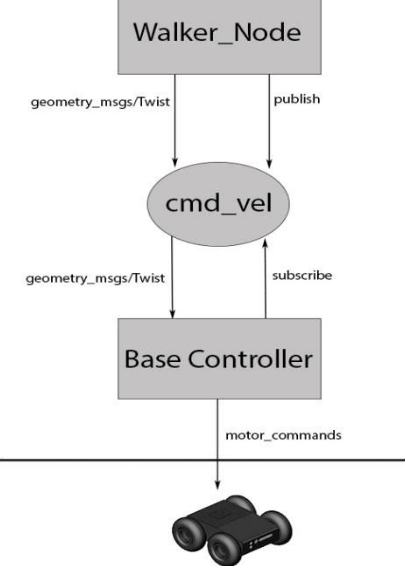
Stage: /opt/ros/indigo/share/stage_ros/world/map.world

File View Run Help

1m 58s 200msec [1.0] [PAUSED]

3.3 Dalla Simulazione al Caso Reale

Stage emula le caratteristiche di un robot, dal movimento ai sensori, creando un nodo che sottoscriva e pubblichi sui topic necessari in base alla configurazione data al robot stesso, e mostrando a video la simulazione. In un caso di esempio reale è necessario invece interfacciarsi al robot fisico. Mantenendo il codice redatto inalterato, per poterlo utilizzare per il movimento reale del robot deve essere costruito quello che in terminologia ROS prende il nome di Base Controller. Un base controller non è altro che un nodo di interfacciamento che comunica direttamente con l'elettronica del robot, e in particolare con la base mobile. La sua funzione consiste quindi nel convertire i messaggi geometry_msgs/Twist relativi alla velocità, avendo sottoscritto il topic cmd_vel, in comandi motore da inviare alla base mobile. Essendo strettamente dipendente dal tipo di piattaforma robotica utilizzata, ROS non ne fornisce una forma standardizzata. La configurazione che ne risulterebbe sarebbe la seguente:



Capitolo 4
Conclusioni

Lo scopo della tesi è stato introdurre il lettore nel campo della robotica e presentare ROS come framework per lo sviluppo di applicazioni. ROS non è di certo il framework migliore in assoluto, poichè tutte le difficoltà e i casi differenti che un campo così complesso come la robotica offre non possono essere risolti certamente con un'unica soluzione, nè la sua architettura risulta sempre la più adatta ai differenti casi. Tuttavia è ormai uno standard de facto all'interno della comunità di sviluppo di applicazioni robotiche. Perchè tutto questo successo, e perchè non usare altri framework quali OROCOS, YARP o Microsoft Robotics Studio? A prescindere da tutti gli aspetti strutturali precedentemente presentati, il primo punto forte di ROS è il fatto di aver avuto come obbiettivo la riusabilità del codice, combattendo la tendenza a rendere algoritmi e driver potenzialmente riutilizzabili in altre applicazioni difficili da estrarre dal contesto originale. In che modo? Incoraggiandone lo sviluppo in librerie standalone prive di dipendendenze da ROS, contenenti virtualmente tutta la complessità, e creando solo piccoli eseguibili che ne espongano le funzionalità a ROS stesso. Inoltre l'essere stato rilasciato sotto licenza BSD, quindi totalmente free e open – source, ha certamente contribuito alla crescita della forte comunità di ricercatori che può vantare alle spalle, e che ne costituisce un altro grande punto di forza. Grazie ad essa ROS è in continuo sviluppo, col rilascio di sempre nuove distribuzioni, e sono sempre maggiori di numero le piattaforme robotiche a supportarlo.

Parlando tuttavia dei limiti che ROS impone, è necessario citare che non sostituisce un sistema operativo ma vi lavora a un livello superiore. Ciò implica la difficoltà di importarlo su sistemi di piccole dimensioni, dove la disponibilità in termini di memoria e risorse è strettamente limitata. Inoltre il fatto di funzionare su sistemi Linux – based lo taglia dalla corposa fetta di mercato operante su sistemi Windows. Per il futuro , la vera sfida di ROS risulterà così essere lo sviluppo di una versione adatta ai sistemi embedded .

Bibliografia e Sitografia

[1] *A Gentle Introduction to ROS* , Jason M. O'Kane

[2] *Is ROS Good for Robotics?*
http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6213236

[3] *ROS by Examples INDIGO – Vol 1,* R. Patrick Goebel

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

[5] *ROS Wiki*, http://wiki.ros.org/it

[6] *Sito Ufficiale*, http://www.ros.org

[7] *Stage*,
http://playerstage.sourceforge.net/index.php?src=stage

[8] *The Robotics Primer* , Maja J Mataric,