INTEGRAZIONE DI ROBOEARTH SU ROS PER LA CONDIVISIONE DI CONOSCENZA E AZIONI PER ROBOT

FLAVIO MARCATO

Integration of Roboearth in ROS for the sharing of knowledge and robot actions

Ingegneria Magistrale in Informatica

11 Dicembre 2012 – versione 1.0

Flavio Marcato: *Integrazione di Roboearth su ROS per la condivisione di conoscenza e azioni per robot,* Integration of Roboearth in ROS for the sharing of knowledge and robot actions, © 11 Dicembre 2012

Keep it simple, stupid!

— The Archlinux community

O al problema c'è soluzione e quindi è inutile preoccuparsi.
O al problema non c'è soluzione e quindi è inutile preoccuparsi.

— Archimede

SOMMARIO

Ad oggi risulta indispensabile l'intervento dell'uomo nella gestione delle attività dei robot. Alcuni recenti progetti fanno tuttavia pensare ad un trend che sempre più minimizza lo sforzo umano nelle procedure di robot-learning. Grazie alle moderne tecnologie informatiche, ora i robot sono in grado di connettersi autonomamente ad Internet per cercare e condividere informazioni di diversa natura, attraverso opportune risorse pensate esclusivamente a questo scopo. Tali risorse sono attualmente raccolte in un progetto innovativo, chiamato Roboearth. Nel nostro lavoro abbiamo implementato dei moduli software sfruttando le sole interfacce web di Roboearth, realizzando attività di identificazione oggetti, `reasoning` legato alle azioni ed esplorazione dell'ambiente. Il tutto focalizzato a metodi basati sul *riuso* di codice, sfruttamento della connessione alla rete, portabilità e condivisione dati, critici in campi inerenti e non nell'automazione industriale.

ABSTRACT

The common sense tends to associate the robot activities along with the human intervention for the movement and interations in the world. Today the trends show us that the man is going to diminish or even disappear from the robot-learning processes. Thanks to the Internet and ad hoc reasources for the scope, the robots are now able to search contents, learn actions and enviroments almost automatically, without the human hand. These resources are studyed and implemented by an innovative project, called Roboearth. We worked integrating Roboearth in a usable framework, trying to emphatise the reuse of code, the internet connection capabilities, the portability and data-sharing features to actually go beyond the present *offline* systems.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth

RINGRAZIAMENTI

Ai miei genitori, che mi hanno sostenuto in questi anni di studio.

Alla mia ragazza, che ha ben contribuito alla mia motivazione.

Al mio relatore, prof. Enrico Pagello.

Al dott. Antonello, che mi ha assistito in fasi critiche di questo lavoro.

A tutti coloro che leggeranno questa tesi.

INDICE

ELENCO DELLE FIGURE

ELENCO DELLE TABELLE

LISTINGS

ACRONIMI

API Application Programming Interface

OWL Ontology Web Language

RDF Resource Description Framework

ROS Robot Operating System

DRY Don't Repeat Yourself!

YAML YAML Ain't Markup Language

JSON JavaScript Object Notation

SeRQL Sesame RDF Query Language

ROI Region Of Interest

Parte I

CONTESTO E FRAMEWORK

1 INTRODUZIONE

In questo lavoro abbiamo studiato e testato *Roboearth*, un sistema online per la condivisione di informazioni tra robot.

Il progetto Roboearth ripensa il ruolo umano nell'interazione tra e con i robot, rendendo quest'ultimi autonomi per quanto riguarda il riconoscimento dell'ambiente e l'apprendimento di azioni.

Nella prima parte della tesi ci siamo concentrati nello studio dell'architettura del software, nonché al *testing* delle funzionalità dello stack ufficiale su ROS Fuerte.

Roboearth può essere definito un sito a stile *Wiki*, progettato specificamente per robot. Il suo funzionamento si basa su opportune Application Programming Interface (API) a cui il robot può accedere al fine di comunicare con una risorsa remota centralizzata. Le possibili azioni sono semplici: si tratta di eseguire *download* o *upload* di informazioni relative ai particolari *task* in esecuzione.

Per Wiki si intende una risorsa web collaborativa che genera informazione grazie ad una comunità di utenti, ciascuno dotato di privilegi di lettura o scrittura di contenuti.

Questo protocollo offre un semplice tramite per *condividere* informazione e conoscenza tra robot, tagliando fuori qualunque intervento da parte dell'uomo.

Figura 1: L'idea di Roboearth.

Roboearth fa parte di un progetto di ricerca presso lo *Swiss Federal Institute of Technology*, a Zurigo.

CONTENUTI: Questa parte affronta le principali funzionalità di Roboearth, coprendo le fasi iniziali di impostazione dell'ambiente di

lavoro, i tutorial e un esempio di task reale effettuato in laboratorio. In particolare i concetti saranno strutturati come segue:

- Descrizione dello stack roboearth
- Passi preliminari
- Simulazione di riconoscimento di oggetti reali in laboratorio

1.1 ARCHITETTURA A TRE LIVELLI

L'obiettivo del progetto Roboearth consiste nel generare conoscenza basata su sistemi di robot, permettendo a quest'ultimi di identificare oggetti, migliorare la qualità delle loro azioni e di adattarsi il più possibile all'ambiente che li circonda. Il tutto senza essere stati progettati *ad hoc* per tali mansioni e sfruttando il *riuso* di dati relativi a oggetti, azioni e mappe. Ancora oggi i robot non sono in grado di capire e adattarsi in ambienti non strutturati in quanto basati su sistemi progettati per prevedere ogni possibile sistuazione nel mondo. In condizioni come queste ogni azione deve essere progettata in anticipo mentre i sistemi di base sono tenuti a ricostruire il loro modello di mondo dagli stimoli dei vari sensori installati sulla macchina. Roboearth utilizza la rete internet per creare una base di dati distribuita a cui si possa accedere in lettura o scrittura da qualunque robot e in ogni momento. A regime, grazie alla mole di conoscenza condivisa, Roboearth vanta il potenziale di poter garantire un potente miglioramento alle capacità di *sensoring*, *action* e *learning* attuali. L'idea è quella di permettere lo sviluppo di sistemi che beneficino dall'esperienza di altri robot, basandosi su un linguaggio standard.

Per ambienti non strutturati intendiamo ambienti anche complessi tipici della vita di tutti i giorni, fuori dal laboratorio.

Roboearth si sviluppa in tre livelli: livello dei dati (a), livello gestionale (b) e livello robot (c).

1.1.1 *Livello dei dati*

L'applicazione lato server rappresenta il *core* di Roboearth descrivibile in due parti, di cui la prima fondamentale:

- La base di dati principale
- L'interfaccia web

La base di dati è un contenitore capiente e accessibile tramite API pronte all'uso mentre le risorse a cui è possibile accedere sono tre: le *action recipes*, gli *objects* e gli *enviroments*, rilasciati ciascuno in diversi formati.

Le API essenzialmente mettono a disposizione delle interfacce che permettono allo sviluppatore di interrogare la base di dati al fine di ottenere le risorse di interesse. Il tutto è poi accompagnato da un *middleware* che si preoccupa di tradurre le *query* semantiche per il sistema interno di Roboearth.

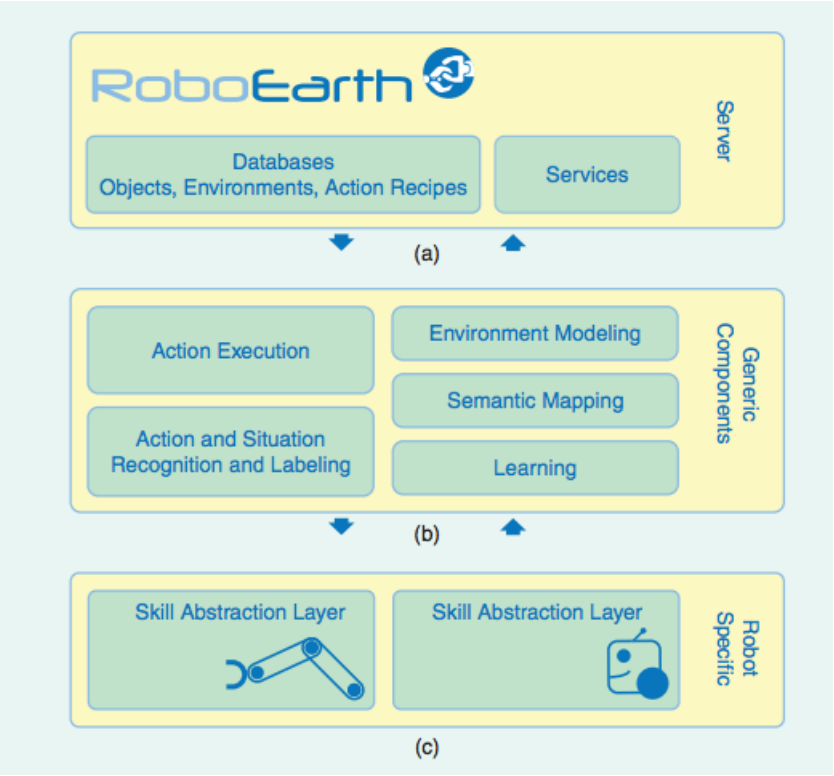

Figura 2: Architettura a tre livelli.

RISORSA	FORMATO
Action recipes	File RDF (OWL)
Objects	modelli CAD, point cloud e immagini .png
Enviroments	file compressi contententi immagini e coordinate

Tabella 1: Struttura dei dati.

ACTION RECIPES: DETTAGLIO Le informazioni legate alle azioni sono organizzate in maniera ricorsiva e si sviluppa in varie dimensioni:

- ogni azione comprende delle subazioni che ne determinano le dipendenze. Ogni subazione è a sua volta un'azione e comprende in genere comandi di alto livello come "Serve a drink", segnalando infine le formule atomiche richieste.
- ogni azione è inoltre catalogata in una collezione, che aiuta il motore di ricerca interno durante la sua esecuzione.
- le azioni sono infine raggruppate in blocchi, che farciti di vincoli di ordinamento, determinano il corretto susseguirsi degli stessi.

OBJECTS: DETTAGLIO La base di dati memorizza informazioni riguardo il tipo di oggetto, le dimensioni, lo stato corrente o altre pro-

prietà di interesse. La figura 3 mostra un esempio di modello in un'i-

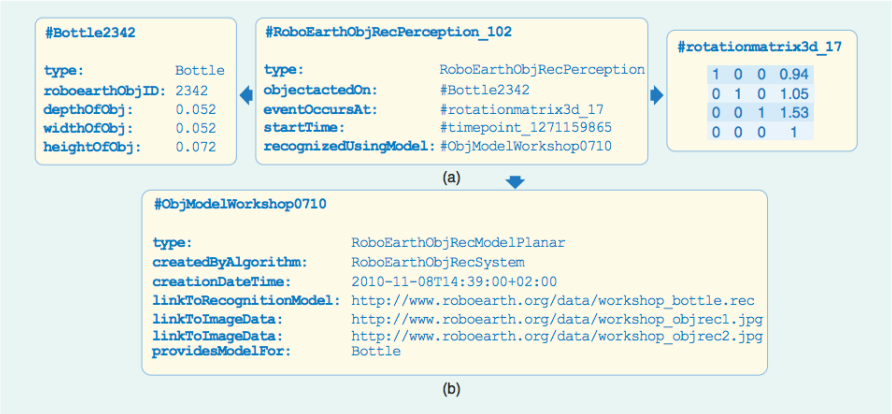

Figura 3: Schema di riconoscimento di oggetto.

potetica situazione di riconoscimento. In questo scenario il software di Roboearth si trova in una fase intermedia di elaborazione e in particolare la presenza dell'intermediario *RoboearthObjRecPerception* sta a significare che *Bottle2342* è stato effettivamente riconosciuto. Ulteriori informazioni sono ricavabili dai vari attributi e link allegati.

ENVIROMENTS: DETTAGLIO La rappresentazione dei dati di localizzazione fuziona in maniera simile agli objects. In questo caso abbiamo a disposizione un importante attributo chiamato *linkToMapFile*, che punta ad un file binario .map che meglio descrive l'ambiente corrente. Un esempio di rappresentazione possiamo trovarlo in figura 4 dove l'istanza di percezione riesce a indicare sia la posizione relativa nel *container*, sia un timestamp sull'ultima rilevazione dell'oggetto.

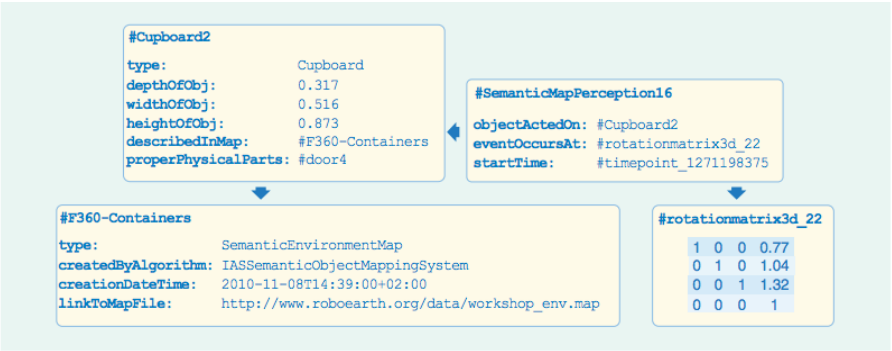

Figura 4: Rappresentazione di ambiente.

L'interfaccia web può essere trovata all'indirizzo `http://api.roboearth.org/`.

Previa registrazione è possibile testare le funzionalità di Roboearth ed esplorare i vari modelli presenti nella base di dati. Viene inoltre fornita un chiave che andrà utilizzata nelle fasi di autenticazione a livello di codice, in tutte le fasi di inserimento o modifica di dati.

1.1.2 *Livello gestionale*

A questo livello intermedio Roboearth offre tutta una serie di funzionalità atte a tradurre i modelli astratti del livello superiore in veri e propri comandi per il robot. In tal senso poniamo particolare accento sulle azioni. Un'ipotetica azione richiede la risoluzione delle situazioni seguenti:

- Identificare un set di requisiti relativi a un certa *macro*
- Determinare il corretto ordine delle subazioni
- Costruire una corrispondenza tra le capacità percettive del robot e le action recipe
- Eseguire le azioni valide

Mentre le prime due possono essere risolte lato server, le ultime due richiedono dei moduli aggiuntivi, implementati al secondo livello, che garantiscano un collegamento permanente tra azione e percezione. Questo passaggio è decisamente delicato in quanto è necessaria una forte standardizzazione nella rappresentazione dei dati, obiettivo tutt'oggi ancora da raggiungere.

Ulteriore accento lo poniamo su una futura funzionalità: la gestione dei fallimenti. Secondo lo stato dell'arte il server di Roboearth procederà con l'esplorazione di alberi di dipendenze alternativi a quello che ha generato il fallimento prevedendo l'intervento da parte dell'utente in caso di eccezione critica. Il sistema insomma promette un miglioramento sostanziale alla robustezza delle applicazioni.

1.1.3 *Livello Robot*

L'ultimo livello non è ad oggi direttamente supportato dal progetto Roboearth ma stando agli articoli gli sviluppatori potranno contare su una serie di interfacce specifiche per il modello e l'hardware di base del robot, probabilmente appoggiandosi a framwork molto diffusi come Robot Operating System (ROS).

1.2 LO STACK ROBOEARTH SU ROS

Il framework ROS si pone tra il secondo e il terzo livello dell'architettura di Roboearth. Tramite comodi servizi pronti all'uso *roboearth* semplifica la comunicazione tra l'applicazione dello sviluppatore e la base di dati online. Lo stack ufficiale è composto da tre pacchetti:

Uno stack di ROS è fondamentalmente una collezione di pacchetti, come una libreria.

- roboearth
- knowrob

- ccny_vision

Tutti i principali servizi sono invece riportati in figura 5.

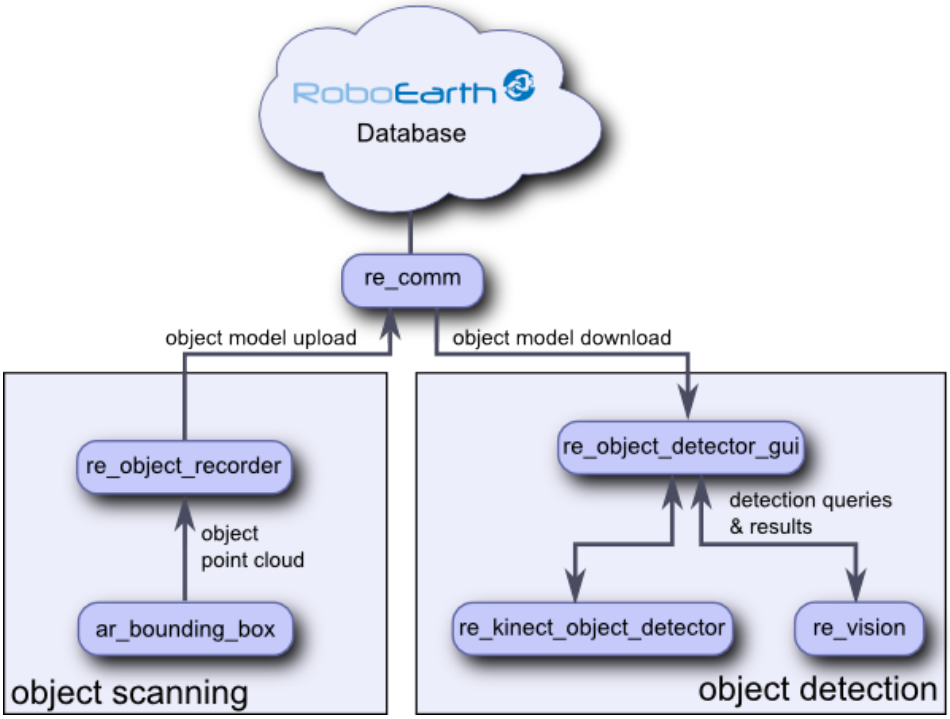

Figura 5: I pacchetti dello stack roboearth.

RE_COMM: modulo di comunicazione principale, specifico di Roboearth.

RE_MSGS: gestione messaggistica, specifico di Roboearth.

RE_ONTOLOGY: gestione ontologia, specifico di Roboearth.

AR_BOUNDING_BOX: modulo di scansione oggetti, object scanning.

RE_OBJECT_RECORDER: GUI di supporto, object scanning.

RE_VISION: modulo di riconoscimento oggetti, object detection.

RE_KINECT_OBJECT_DETECTO: supporto alla kinect per il riconoscimento, object detection.

RE_OBJECT_DETECTOR_GUI: GUI di supporto, object detection.

1.3 PASSI PRELIMINARI

1.3.1 *Porting di Roboearth su ROS Fuerte*

Nelle prime fasi di lavoro abbiamo riscontrato problemi in fase di compilazione dello stack. Appurata come una *known issue* ci siamo

attivati per sistemare i sorgenti. I cambiamenti più sostanziali sono stati applicati ai percorsi degli *header* contenuti in alcuni pacchetti di prefisso *re*, nonché alla definizione di alcune dipendenze critiche nei *manifest.xml*. Per tutti i dettagli abbiamo costruito un file .diff, allegato in appendice.

Il problema è ben sentito dalla comunità di ROS, così come dimostra una diretta ricerca su ROS Answers (`http://goo.gl/lmkRG`).

1.3.2 *Istruzioni di installazione*

Una volta studiato e portato a termine il porting dello stack roboearth su ROS Fuerte abbiamo preparato il una versione ad hoc del file di installazione "roboearth.rosinstall".

I passi seguenti sono stati testati in Ubuntu 12.04 Precise Pangolin su ROS Fuerte.

I sorgenti sono reperibili su Github: `https://github.com/flavius476/roboearth`

PREPARAZIONE WORKSPACE

```
sudo apt-get install python-rosinstall
sudo apt-get install python-rosdep
mkdir -p ~/Works && cd ~/Works
mkdir -p ~/Works/workspace
# Il .rosintall comprende il nostro porting a Fuerte
rosinstall . /opt/ros/fuerte http://db.tt/4eeX9En6
source ~/Works/setup.bash
```

CONFIGURAZIONE .BASHRC

```
# da aggiungere a ~/.bashrc
source /opt/ros/fuerte/setup.bash
source ~/Works/setup.bash
export ROS\_PACKAGE_PATH=~/Works/${ROS_PACKAGE_PATH}
export Ros\_WORKSPACE=~/Works/workspace
export JAVA\_HOME=/usr/lib/jvm/java-6-openjdk
```

INSTALLAZIONE DELLE DIPENDENZE

```
sudo apt-get install ros-fuerte-ias-common \
ros-fuerte-simulator-gazebo ros-fuerte-vision-opencv \
ros-fuerte-octomap-mapping \
ros-fuerte-client-rosjava-jni
sudo apt-get install swi-prolog libgstreamer0.10-dev \
libkml0 libjson-glib-dev
rosdep install knowrob_actions
```

COMPILAZIONE DELLO STACK

```
rosmake roboearth --pre-clean
```

Riportiamo ulteriori dettagli sul porting e sul file di installazione in Appendice A.

1.3.3 *Driver necessari per l'uso della Kinect*

Nelle esperienze descritte nel capitolo 2 utilizzeremo una Kinect per gestire i descrittori di oggetti su Roboearth. Tutte le funzionalità necessarie sono state reperite grazie a tre stack di ROS: *openni_camera*, *openni_kinect* e *openni_launch*.

INSTALLAZIONE DRIVER

```
sudo apt-get install ros-fuerte-openni-camera
sudo apt-get install ros-fuerte-openni-kinect
sudo apt-get install ros-fuerte-openni-launch
```

AGGIORNAMENTO RVIZ (OPZIONALE)

```
rosdep install rviz
rosmake rviz --pre-clean
```

1.3.4 *Porting dello stack NXT su ROS Fuerte*

Utilizzeremo il robot NXT Mindstorm nel progetto Dream, centrale nel lavoro di tesi, come *testdrive*.

Abbiamo anche in questo caso apportato alcune modifiche allo stack di ROS, datato e incombatibile con Fuerte. Abbiamo dunque applicato dei cambiamenti traendo ispirazione dalla patch al Turtlebot del 2012, ridefinendo alcune strutture di supporto riguardo la gestione dei messaggi.

Tutti i dettagli sono reperibili in appendice A.2

2 IN LABORATORIO

2.1 ESPERIENZA N.1

Nella prima esperienza abbiamo testato le funzionalità di scansione di Roboearth registrando un oggetto casuale con una kinect presente in laboratorio e cercando di costruire un modello compatibile con lo stack.

Per concludere il lavoro abbiamo seguito i seguenti passi:

Figura 6: In laboratorio.

SETUP: utilizzando il template relativo al pacchetto re_comm_recorder abbiamo fatto attenzione a rispettare i parametri dei marker pari a 8cm per lato, con consistenza di colore nero sufficiente per le forme. Abbiamo poi posizionato la Kinect su un supporto improvvisato in laboratorio a circa 60cm dall'obiettivo, ispezionando la qualità dell'immagine con rviz.

INIZIALIZZAZIONE: abbiamo lanciato i vari servizi necessari, seguendo l'ordine seguente.

Listing 1: Inizializzazione Esperienza n.1

```
# driver openni
roslaunch openni_launch openni.launch
```

```
# supporto kinect e marker AR
roslaunch ar_bounding_box ar_kinect.launch

# comunicazione server Roboearth
rosrun re_comm run

# interfaccia utente
rosrun re_object_recorder record_gui
```

RILEVAZIONE: abbiamo ripetuto più rilevazioni con diversi oggetti, scegliendo infine un'elegante tazza porta penne, presente in laboratorio. Riportiamo il risultato finale in figura 7.

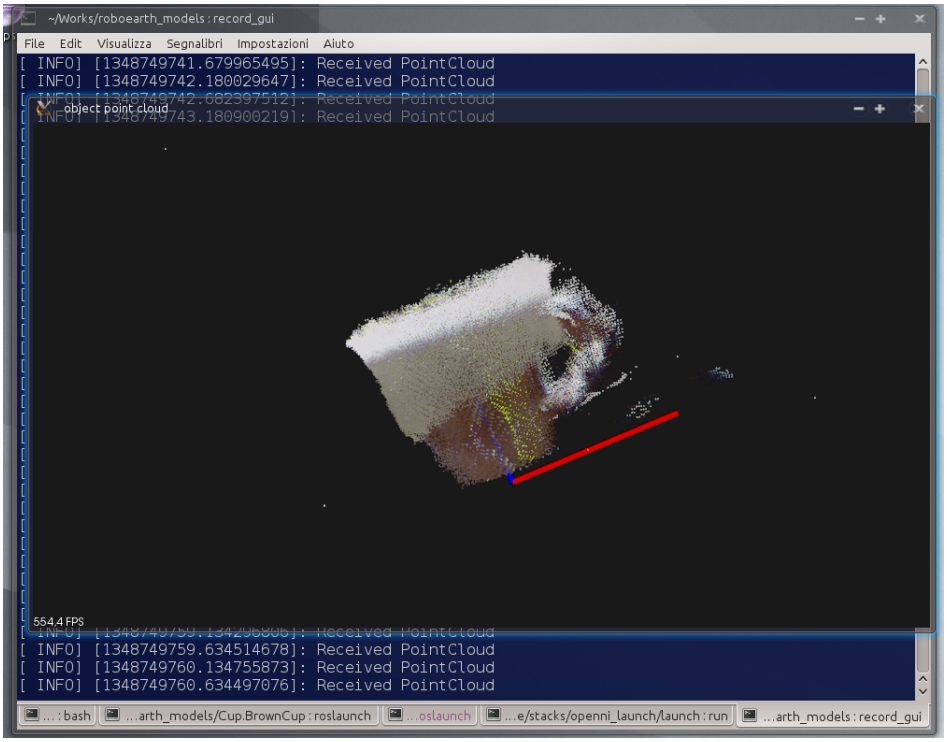

Figura 7: Esperienza di scansione.

NOTE: Le rilevazioni di pointcloud non sono avvenute in maniera ottimale a causa di un malfunzionamento del driver openni in corrispondenza del parametro *depht_registration*.

2.2 ESPERIENZA N.2

Nella seconda esperienza abbiamo testato le le funzionalità di riconoscimento scaricando un modello da Roboearth, utilizzando nuovamente la kinect in dotazione.

Per concludere il lavoro abbiamo seguito i seguenti passi:

SETUP: utilizzando il template relativo al pacchetto *re_comm_recorder* abbiamo fatto attenzione a rispettare i parametri dei marker pari a 8cm per lato, con consistenza di colore nero sufficiente per le forme. Abbiamo poi posizionato la Kinect su un supporto improvvisato in laboratorio a circa 60cm dall'obiettivo, ispezionando la qualità dell'immagine con rviz.

INIZIALIZZAZIONE: abbiamo lanciato i vari servizi necessari, seguendo l'ordine seguente.

Listing 2: Inizializzazione Esperienza n.2

```
# driver openni
roslaunch openni_launch openni.launch

# comunicazione server Roboearth
rosrun re_comm run

# attivazione algoritmo di rilevazione
rosrun re_kinect_object_detector re_kinect

# interfaccia utente
rosrun re_object_detector_gui detect
```

RICONOSCIMENTO: Questa esperienza non si è purtroppo conclusa a buon fine a causa dell'inacessibilità dei servizi lato server e quindi della ricezione di modelli CAD. Ciò ha reso dunque impossibile l'esecuzione degli ultimi passi.

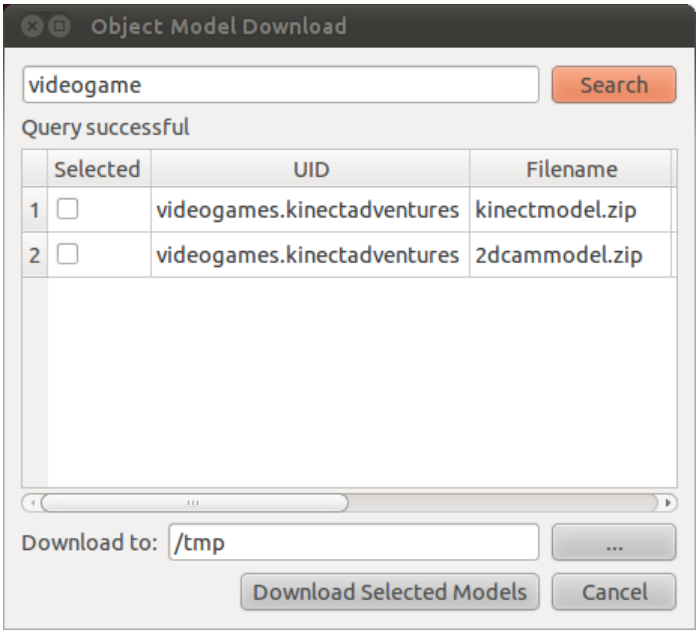

Figura 8: Esperienza di rilevazione.

Parte II

IL PROGETTO DREAM

3

FRONT END

Lo studio delle potenzialità di Roboearth ha ispirato il lavoro principale di questa tesi. Gli attuali limiti dei robot riguardo la loro architettura, sistema *harware* e linguaggio di programmazione tendono a creare grosse barriere in termini di comunicazione tra macchine di diversa natura. Per questo l'astrazione e l'utilizzo di concetti di più alto livello permettono di affrontare un dato problema da un'ampia prospettiva. In termini di realizzazione pratica si tratta dunque di potenziare le capacità di *learning* e condivisione di concetti tra robot, stimolando alcune tipiche proprietà di un pacchetto software moderno.

È quindi fondamentale una conoscenza del dominio, o stato dell'arte, dell'attuale situazione nella robotica autonoma: in questo senso Roboearth desta molto interesse, soprattutto da parte degli sviluppatori, in quanto si parla spesso di *feature* che sono state raccolte e tenute in grande considerazione nello sviluppo del pacchetto Dream, in questa tesi. Stiamo parlando in definitiva di parole chiave come: *portabilità*, *estensibilità*, *robustezza* e *modularità* del *software*.

In questa parte verranno esposti i passi percorsi durante la progettazione e lo sviluppo del pacchetto Dream nonché del relativo *testdrive*. Cominceremo dagli obiettivi e requisiti per poi evidenziare le varie scelte che si sono susseguite durante l'implementazione.

3.1 DEFINIZIONE DEL PRODOTTO

In questa tesi abbiamo progettato e realizzato un pacchetto software che possiamo definire un "gestore di azioni per robot" ispirato ai comuni gestori di pacchetti nei sistemi Unix, dotato di funzioni come la risoluzione delle dipendenze e lo sfruttamento di conoscenza condivisa tramite il core di Roboearth. Tutte le caratteristiche, i dettagli, le fasi di lavoro e le scelte fatte nel percorso di realizzazione sono sviluppate in due fasi: iniziale (front end) e finale (back end).

3.2 OBIETTIVI

Come obiettivo ultimo ci siamo posti lo studio, la progettazione e lo sviluppo di un pacchetto software pensato per gli sviluppatori di applicazioni per robot. In particolare abbiamo posto molta attenzione sul ruolo delle azioni e sull'uso di strumenti già riconosciuti dalla community. Abbiamo quindi stilato una lista di proprietà che il prodotto finale dovrà supportare:

- Spinta integrazione con ROS

- Indipendenza del pacchetto
- Sfruttamento di Roboearth

3.2.1 *Integrazione con ROS: dettaglio*

Quando si parla di sviluppo su robot non si può non citare ROS:

"Robot Operating System (ROS) is a software framework for robot software development, providing operating system-like functionality on a heterogenous computer cluster. ROS was originally developed in 2007 under the name switchyard by the Stanford Artificial Intelligence Laboratory in support of the Stanford AI Robot project. As of 2008, development continues primarily at Willow Garage, a robotics research institute/incubator, with more than twenty institutions collaborating in a federated development model."[1]

Questo framework è ad oggi in forte crescita e ricco di *feedback positivi* da parte della comunità di sviluppatori, per questo è stato analizzato e posto come base di partenza per l'intero lavoro. Tra i vari vantaggi nell'uso di ROS possiamo trovare:

ASTRAZIONE DELL'HARDWARE Ogni funzionalità è identificata da un nodo e ogni nodo può comunicare tramite semplice messaggi. Il sistema mette a disposizione una serie di utili funzioni che facilitano lo sviluppo di driver e azioni di basso livello per robot e giunti.

MANUTENIBILITÀ Grazie al supporto della community i problemi possono essere rapidamente identificati e risolti, a disposizione troviamo la wiki ufficiale[2] e la piattaforma Q/A[3] ricca di utenti di ogni livello.

Con ROS Answers l'utente sa precisamente se l'aiuto proviene da una fonte autorevole o meno, grazie al sistema StackExchange.

PORTABILITÀ ORIZZONTALE ROS garantisce supporto per vari modelli di robot e tale numero è in continua crescita.

PORTABILITÀ VERTICALE I linguaggi supportati sono C++ e Python, entrambi *cross-platform* e dunque usabili in qualunque sistema operativo.

ESTENSIBILITÀ la struttura stessa del framework, basata su pacchetti e stack, stimola fortemente lo sviluppo di software modulare e dunque ordinato, manutenibile ed estendibile. Insomma questa proprietà apporta un buon contributo a tutti i vantaggi sopradescritti.

1 Tratto da Wikipedia: http://en.wikipedia.org/wiki/Robot_Operating_System
2 ROS Wiki: http://www.ros.org/wiki/
3 ROS Answers: http://answers.ros.org/questions/

3.2.2 *Indipendenza del pacchetto: dettaglio*

Con indipedenza del pacchetto intendiamo un'unità software *self-contained*: il prodotto sarà facilmente installabile e utilizzabile da uno sviluppatore ROS come un semplice servizio. Inoltre sarà semplice agganciare qualunque tipo di pacchetto aggiuntivo che gestisca funzioni specifiche lato robot come, per esempio, il driver di basso livello. Abbiamo fortemente voluto un prodotto sia interessante che utile nella pratica, per questo evidenzieremo tale caratteristica nel capitolo 4, quando parleremo dell'implementazione: il pacchetto finale maneggerà l'informazione accettando e restituendo definizioni di azioni comprensibili da qualunque robot ROS-compatibile.

3.2.3 *Sfruttamento di Roboearth: dettaglio*

Non è raro che all'interno di un laboratorio farcito di menti geniali vengano rilasciate nuove ed interessanti idee. La vera sfida sta però nella condivisione della conoscenza: nel mondo della robotica autonoma ogni modello di macchina tende ad avere solo programmi di supporto ad hoc, garantendo grande controllo ma scarso riuso. È inoltre per superare il frequente fenomeno di "reinvenzione della ruota" che abbiamo osservato attentamente Roboearth: un agente terzo in grado di raccogliere tutte le informazioni elaborate in un dato lavoro per poi metterle a disposizione del mondo, grazie alla forza della Rete. Abbiamo studiato insomma un modo per applicare l'idea del progetto Roboearth, in un pacchetto software reale, costruendo uno strumento che offra agli sviluppatori un'interfaccia semplice a una base di dati già pronta ed estendibile, piuttosto che spingere loro a crearne una nuova ad ogni iterazione[4].

3.3 REQUISITI E ASSUNZIONI

Identificato lo scopo ultimo e le *feature* da includere nel prodotto elenchiamo i requisiti che sono risultati necessari alla stesura del codice e nell'esecuzione dei test in laboratorio. Abbiamo dunque definito ROS come framework principale e Roboearth come risorsa remota a cui attingere per le operazioni di download e upload di dati.

Possiamo riassumere nei seguenti punti le azioni che abbiamo intrapreso per rendere possibile la fase di implementazione:

CONFIGURAZIONE SISTEMA OPERATIVO Sebbene la scelta della piattaforma di base sia spesso di carattere soggettivo, nel nostro caso abbiamo avuto ben poca libertà in quanto ROS è supportato attualmente per la sola distribuzione Ubuntu Linux, men-

4 Vedi principio Don't Repeat Yourself! (DRY)
http://en.wikipedia.org/wiki/Don't_repeat_yourself

TIPO DI AZIONE	DESCRIZIONE	ESEGUIBILE
Driver	Funzione interna, di basso livello	SI
Atomica	Tipicamente insierite in un driver di comando	SI
Macroazione	Definita da un termine più generico	NO

Tabella 2: Tipi di Azione

tre risulta "sperimentale" in altri sistemi come OSX, Windows e in altre distribuzioni come Debian e Archlinux. In definitiva è altamente consigliabile attenersi al prodotto più supportato, per non incorrere in problemi tutt'altro che banali in fase di installazione.

COMPATIBILITÀ DELLO STACK Roboearth è implementato tramite stack su ROS Fuerte, così come vari altri supporti relativi a robot fisici. Abbiamo tuttavia illustrato nel capitolo 1 il necessario (e ingente) lavoro di porting. Per tutti i dettagli rimandiamo all'apposita Appendice A.

IMPLEMENTAZIONE FUNZIONI DI BASE La gestione delle *azioni* è stato un focus assolutamente centrale: attenendoci allo stato dell'arte abbiamo definito tre insiemi di azioni, come illustrato nella tabella 2.

Snippet di esempio sono riportati in Appendice.

Un esempio di azione atomica può essere, relativamente al movimento, una funzione come *straightToCone()*. Gestita da un modulo driver tale azione consiste in genere da una composizione di azioni di basso livello direttamente eseguibili dal robot. Un esempio di macroazione può essere invece una funzione come *serveADrink*. Intuitivamente tale azione consisterebbe in una composizione di subazioni come *getTheDrink()*, *moveTheArm()*, *releaseTheDrink()*.

In laboratorio assumiamo che il robot sia in grado di svolgere le azioni atomiche. In tal modo simuliamo un ipotetico sviluppatore totalmente indipendente dal nostro pacchetto e che dunque abbia la libertà di definire i vari *behaviour* di basso livello.

CONNESSIONE ALLA RISORSA REMOTA Essendo una base di dati online la connettività alla rete risulta fondamentale per il pieno sfruttamento del software. In casi di emergenza abbiamo comunque garantito un modulo offline che, sebbene non goda della stessa potenzialità della versione completa, permette al pacchetto di funzionare.

3.4 AVVIO E SCELTE PRATICHE

In relazione ai requisiti stilati nella sezione precedente riportiamo qui le varie scelte fatte in termini di configurazione hardware, software, formato dei dati e linguaggi di programmazione.

3.4.1 *Configurazione Harware*

Entrambe le fasi di progettazione ed implementazione sono state cominciate e portate a termine con un Macbook Air da 13 pollici dalle seguenti caratteristiche:

- Processore Intel Core i7 da 1.8 GHz
- Memoria RAM da 4GB DDR3 e 1333 MHz
- Grafica Intel HD Graphics 3000 da 384 MB
- Archivio SSD da 256 GB

Per garantire il proseguimento dei lavori più agevole possibile segnaliamo l'importanza del processore. Critico soprattutto nelle fasi di compilazione è consigliabile adottare almeno un dual core da 1.3 GHz in modo da sfruttare un buon parallellismo.

Il robot scelto come *testdrive* è l'NXT Mindstorm, fornito dal laboratorio di robotica[5] presso la stessa Università di Padova. La scelta è dovuta a due motivi: primo, il robot è ufficialmente supportato da ROS (porting a parte), secondo, l'istrinseca proprietà di scambio dei pezzi e dunque dei sensori simula bene il concetto di portabilità orizzontale: una funzione di *obstacle detection* può variare di molto da un NXT equipaggiato di infrarossi da uno dotato di telecamera. In questo senso il pacchetto software è stato progettato per essere funzionante in qualunque configurazione.

I sensori e attuatori presenti in questo modello sono:

- Ultrasonico, supportato da messaggi di tipo Range
- Webcam, supportata da moduli OpenCV per la visione
- Due motori, supportati da messaggi di tipo Twist

Il robot è autonomo e funziona a batterie la cui rispettiva longevità si aggira sulle 2/3 ore. Computer e webcam necessitano ciascuno di un cavo USB per lo scambio di comandi e la trasmissione del flusso video rispettivamente.

5 IAS-LAb: http://robotics.dei.unipd.it

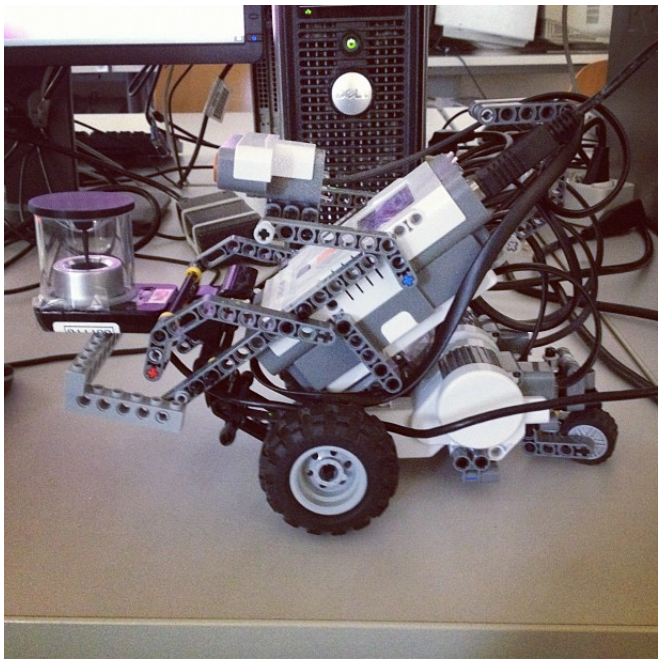

Figura 9: Il robot NXT Mindstorm: Ufo Robot.

NOTE: tutti i moduli NXT sono stati sviluppati sui pacchetti messi a disposizione da ROS portati a Fuerte in occasione di questo lavoro, in particolare:

- `nxt_ros`
- `nxt`
- `nxt_apps`
- `nxt_robots`

Riportiamo ulteriori dettagli sulla configurazione dei motori e le porte utilizzate in Appendice B.

3.4.2 *Configurazione Software*

In accordo coi requisiti del progetto abbiamo lavorato con:

SISTEMA OPERATIVO: Ubuntu 12.04 LTS.
`Nota:` Ubuntu è stato installato in dual boot con OSX. L'alternativa in macchina virtuale è stata scartata in quanto presentava difficolà nell'uso di connessioni USB.

FRAMEWORK: ROS Fuerte, versione stabile odierna. L'installazione del framework è risultata agevole tramite la wiki e il gestore pacchetti *apt-get*. È infatti presente il repository ufficiale della community. In laboratorio abbiamo utilizzato il pacchetto completo *ros-desktop-full*.

LINGUAGGI DI PROGRAMMAZIONE: Python 2.7.2, C++. Utilizzati entrambi per evidenziare ancora una volta la proprietà di portabilità del software, nodi scritti in linguaggi differenti saranno in grado di comunicare grazie al sistema di messaggi messo a disposizione dal framework.

Una buona presentazione su Python: http://goo.gl/XUCed

STRUMENTI: Durante il lavoro di sviluppo abbiamo configurato una cartella condivisa su Dropbox e successivamente una repo su GIT (https://github.com/flavius476/Tesi-Magistrale).
Sia per i sorgenti, che per la tesi in LaTeX abbiamo mantenuto un backup in tempo reale e disposto un canale comune tra sviluppatore e tutor per tutti i feedback necessari. Abbiamo poi utilizzato Sublime-Text[6] come editor universale.

3.4.3 *Formato dei Dati*

I principali formati di dato con i quali il progetto scambia informazione sono tre: YAML Ain't Markup Language (YAML), JavaScript Object Notation (JSON) e OWL.

YAML è un tipo di serializzazione che rappresenta i dati sottoforma di dizionario o liste di dizionari. Importante concetto da evidenziare è la gerarchia dei dati, rappresentata tramite tabulazioni o spazi, tutti inseriti in in file ".yaml".
Lo YAML è particolarmente comodo in quanto facilmente leggibile e modificabile, sia da mano umana che da script automatizzati.
Un esempio di file YAML lo riportiamo in Appendice A.3.

Ulteriori esempi saranno riportati in dettaglio nel prossimo capitolo.

JSON è un altro tipo di serializzazione molto comune nell'interscambio di informazione tra applicazioni web. Lo scopo è lo stesso dello YAML, rappresentare i dati in maniera facilmente leggibile e snella. I vincoli di appartenenza e gerarchia sono però espressi tramite le parentesi graffe, piuttosto che dalle tabulazioni.
Un esempio di file JSON lo riportiamo in appendice A.3.

OWL è una famiglia di linguaggi per la rappresentazione e descrizione di ontologie, ovvero conoscenza descritta come insieme di concetti dotati di dominio. Nella pratica OWL è un'estensione dello standard Resource Description Framework (RDF) [7] che aggiunge il concetto di classi e sottoclassi, ovvero raggruppamenti di oggetti che condividono caratteristiche comuni.
Un esempio di file RDF lo riportiamo in Appendice A.3.

6 Preferito dall'autore: http://www.sublimetext.com
7 W3C: http://www.w3.org/RDF/

4

BACK END

Chiariti i requisiti e le prime scelte progettuali descriviamo ora il cuore del lavoro svolto: lo sviluppo del pacchetto Dream e del relativo *testdrive*. Dream è completamente integrato in ROS ed è dunque organizzato in *nodi*, ovvero moduli indipendenti che comunicano tra di loro tramite messaggi. Nelle sezioni che seguono descriveremo tutti i nodi ROS che compongono il pacchetto da noi sviluppato:

- Il client, richiedente delle azioni
- Il server, gestore delle richieste
- Il modulo di comunicazione con Roboearth
- Il driver del robot

Il pacchetto è diviso in due parti principali: il *core*, etichettato "Dream" e il testdrive, che comprende i moduli di supporto per il robot NXT Mindstorm. Le due parti sono totalmente indipendenti l'una dall'altra ma comunicano tra di loro tramite particolari messaggi. Identifichiamo a tal proposito due possibili azioni:

- Sottoscrizione: messa in ascolto di messaggi in ingresso
- Pubblicazione: scrittura e invio di messaggi

Nel nostro caso parliamo di messaggi contenenti stringhe il cui formato è supportato da String, incluso in *std_msgs*. La comunicazione è basata sul classico modello client-server, con supporto di ack per la conferma di ricezione. Parliamo di un sistema basato su canali rumorosi senza gestione degli errori, caratterizzato da saltuarie perdite di messaggi. Segnaliamo dunque che per evitare evenutali blocchi dell'iterazione abbiamo riadattato il modello, sostituendo l'invio semplice con un "burst" di messaggi (ack o dati), il cui numero è aggiustato empiricamente a seconda della situazione.

Abbiamo utilizzato un tipo di messaggio direttamente supportato `http://goo.gl/BU6pF`

In definitiva abbiamo costruito dei canali di comunicazione cercando di enfatizzare il più possibile la modularità del software: ogni nodo può infatti comunicare con altri nodi e accedere ai dati di cui ha bisogno sottoscrivendosi al canale appropriato, gestendo poi la risposta tramite *parsing* della stringa in entrata. Con questa filosofia abbiamo dunque progettato il modulo di test, simulando uno sviluppatore esterno che desideri usufruire dei servizi offerti dal modulo Dream. In totale abbiamo registrato tre canali:

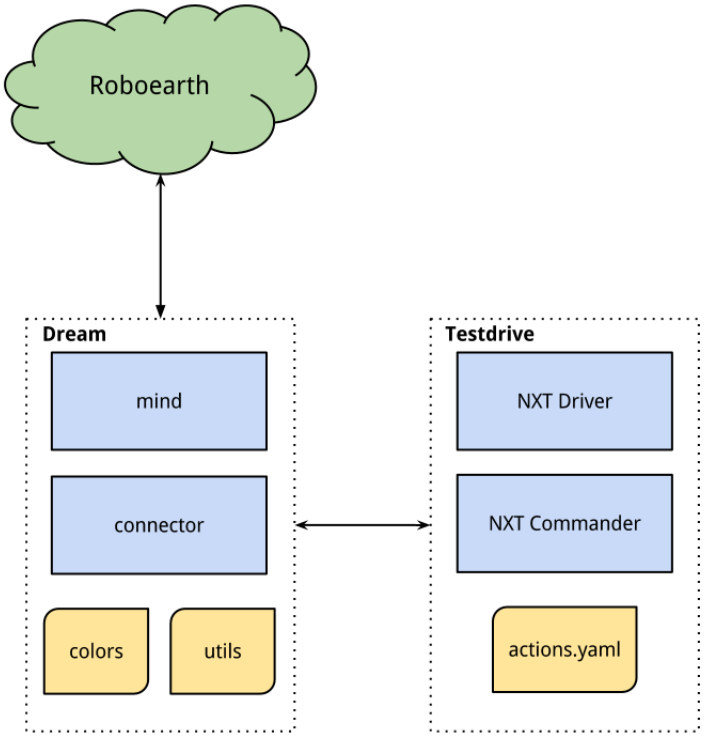

Figura 10: Architettura del pacchetto Dream.

ACTION REQUEST sottoscritto dal nodo *mind.py*, questo canale contiene la richiesta di azione in formato stringa, per esempio, tramite il nome o una parola chiave.

ACTION RESPONSE pubblicato dal nodo *mind.py*, questo canale contiene la risposta di azione in formato stringa, dopo essere stata elaborata dallo stesso nodo.

MANAGER RESPONSE sottoscritto dal nodo *mind.py*, questo canale aiuta la sincronizzazione dei messaggi tra robot e Dream, sbloccando l'invio di risposte dopo l'effettiva esecuzione di precedenti comandi.

4.1 IL CORE

Abbiamo etichettato il core del progetto con "Dream". Interamente scritto in Python, questo pacchetto è composto da due moduli principali più due *utility*.

4.1.1 *Il nodo Mind*

Mind.py è centrale nel funzionamento di tutto il sistema: attivabile da riga di comando con le opzioni "-i" e "-r", si preoccupa di inizia-

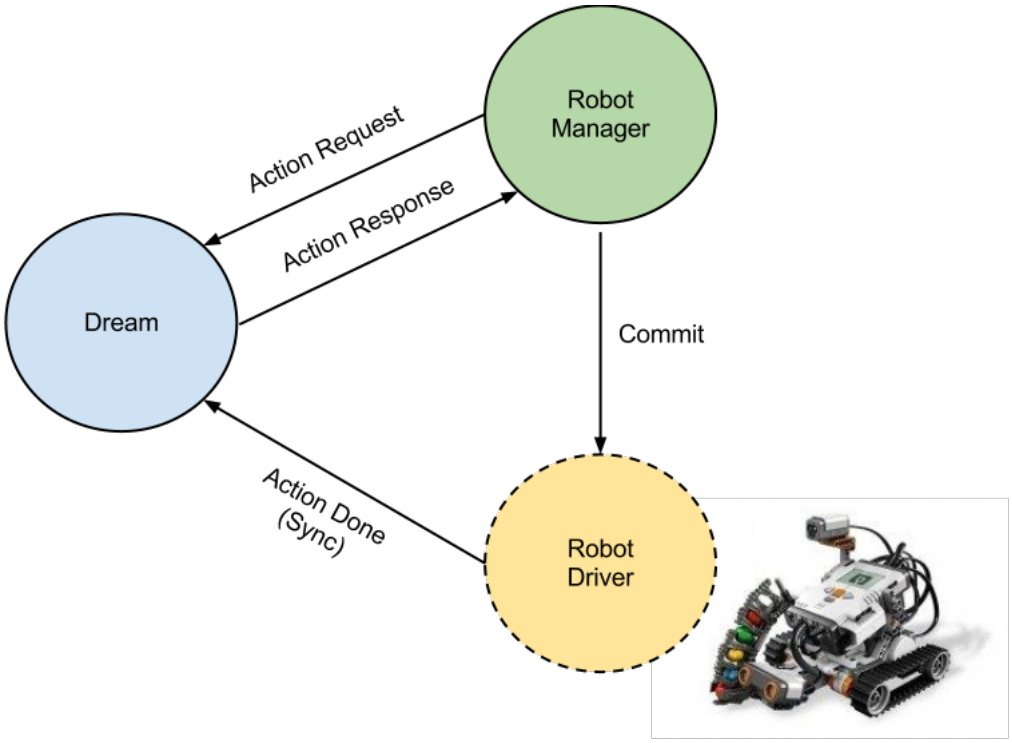

Figura 11: Messaggi tra nodi.

lizzare i canali di comunicazione e gestirli tramite due chiamate di callback: *mainCallback* e *managerCallback*.

Listing 3: Dream: Esempio di invocazione

```
rosrun Dream mind.py -i ~/Recipes/robot_actions.yaml -r
```

L'opzione "-i" indica a Dream dove trovare lo YAML relativo alle azioni del robot, "-r" invece attiva la funzionalità del connector. In definitiva le funzioni svolte da *Mind.py* sono le seguenti:

MAINCALLBACK riceve le richieste dal canale "action_request", carica lo YAML in memoria e invoca un'elaborazione ricorsiva. Nel caso base viene lanciata una *checkKnownAction* preventiva sull'azione in ingresso: se tale azione è presente nel descrittore delle azioni viene inoltrato il comando di esecuzione, altrimenti si fa intervenire la funzione *connect_to_roboearth*, specificata in *Connector.py*. Nel caso medio la richiesta di azione è una macro non conosciuta dal robot e che dunque viene decodificata da Roboearth come insieme di subazioni. Per ognuna delle subazioni si ripete il processo sino ad ottenere una sequenza di azioni atomiche conosciute e dunque eseguibili dal robot. È infine possibile che la richiesta venga respinta, nel caso in cui venga trovata anche solo un'azione atomica non specificata nell'apposito descrittore YAML.

MANAGERCALLBACK riceve degli ack dal nodo robot (nel nostro caso, dal testdrive) e si preoccupa di settare alcune costanti per la sincronizzazione delle rispose. In particolare:

- `action_sync` è un semaforo che segnala l'avvenuta esecuzione dell'azione inoltrata, se impostato a 1 sblocca l'elaborazione di ulteriori richieste in coda, altrimenti mette in attesa il nodo corrente.
- `tryals` è un intero che rappresenta il numero di messaggi o ack spediti per singolo invio. Il valore di questa costante è aggiustato empiricamente per garantire la ricezione dei messaggi (che talvolta vanno persi) senza sovraccaricare il canale.

4.1.2 *Il nodo Connector*

Connector.py è il modulo che si preoccupa di comunicare con Roboearth tramite le apposite API. Il procedimento si articola in due fasi:

1. Viene sottoposta l'interrogazione tramite una connessione di tipo `post`, allegando al *payload* un pacchetto contenente la stringa serializzata in JSON. Di fondamentale importanza è la costruzione dell'interrogazione, che nel nostro caso consiste in SeRQL. In tutte le richieste ricorriamo ad una clausola `select source`, imposta da Roboearth, filtrando i risultati in base a un preciso campo. Nelle azioni maneggiate dal pacchetto Dream abbiamo utilizzato il soggetto "rdfs:label" come identificativo. La ricerca è effettuata tramite l'operatore `like`, che verifica l'occorrenza dell'etichetta richiesta.

   Listing 4: Esempio di interrogazione SeRQL

   ```
   SELECT source FROM CONTEXT source {x} rdfs:label {L}
       where L LIKE "*manouver*"
   ```

   Il risultato potrà comprendere più di un'azione e nel nostro caso il discriminante è scelto in base alla fattibilità: il robot esegue la prima azione che ha registrato nel proprio YAML.

2. Avvenuta la ricezione il nodo procede con il parsing della risposta JSON, estrapolando l'azione in OWL. Ciò viene fatto agilmente grazie alla libreria *rdflib*, la quale mette a disposizione utili funzioni a questo scopo sia in lettura che scrittura. Un'importante azione svolta a questo livello riguarda però l'ordinamento delle subazioni che nativamente sono trattate come membri di un insieme e dunque disposte casualmente. Nel nostro caso

l'ordine determina l'identità della macroazione e per questo abbiamo `scelto` di implementare i vincoli di ordinamento tramite la proprietà "owl:Annotation".

Ulteriori dettagli a questo proposito sono riportati in sezione 5.5.

Per completezza riportiamo lo snippet più significativo di *connector.py* in Appendice B.2.

4.1.3 *Le action_utils*

action_utils è uno script che raccoglie una serie di funzioni il cui scopo risponde alla necessità di identificare la fattibilità di una determinata azione e all'inoltro del comando di esecuzione. Parliamo di due funzioni principali: *checkKnownAction* e *getAction*.

CHECKKNOWNACTION riceve due parametri obbligatori più uno facoltativo: lo YAML che descrive il robot, una stringa rappresentante la richiesta e il campo di ricerca. La ricerca è appunto effettuata in due fasi: nella prima si estrapolano le parole chiave dallo YAML mentre nella seconda si verifica che l'azione sia in qualche modo contenuta nella lista risultante. Il terzo parametro gioca un ruolo di supporto: il motore di ricerca attraverserà più stadi di validazione lavorando su una serie di parole chiave come l'ID, il nome oppure i "tag". Tutto ciò serve a irrobustire il sistema nel caso in cui le azioni atomiche non siano invocate con il loro identificativo. Il valore di ritorno finale è un booleano: True o False.

GETACTION Simile alla funzione precedente, differisce solamente nel valore di ritorno che consiste in una stringa corrispondente all'azione richiesta. Anche in questo caso supportiamo una ricerca tramite parole chiave come identificativo, nome o tag.

Facciamo presente che l'utilizzo dei tag si verifica frequentemente in quanto le *action recipe* scaricate da Roboearth spesso sono definite in modo generale e dunque potenzialmente diverso dalle particolari descrizioni nel nostro YAML.

NOTE: abbiamo inserito una leggera ridondanza nelle due funzioni precedenti. Il file YAML è infatti scansionato due volte. Adottando questa scelta siamo consapevoli di contravvenire ai puristi dell'algoritmica ma facciamo presente due considerazioni:

1. Favoriamo la leggibilità del codice, posta in primo piano poichè in fase di studio

2. *getAction* è spesso invocata dopo una verifica positiva di *checkKnownAction*, con conseguente sicurezza di trovare l'elemento cercato e successiva uscita dal ciclo di iterazione.

3. trattiamo con file di dimensioni molto modeste e decisamente poco influenti sulla fluidità complessiva

Possiamo dunque dire che abbiamo creato una ridondanza di fattore $1 < x < 2$ rispetto alla soluzione ottimale in termini di numero di accessi in memoria, su una memoria molto limitata. In definitiva un costo che ci siamo sentiti di pagare.

Riportiamo infine lo snippet di dettaglio in Appendice B.3.

4.2 IL TESTDRIVE

Per testare il funzionamento del core abbiamo scelto di sviluppare un *testdrive* su robot NXT Mindstorm.

La modularità intrinseca del dispositivo *lego* ci ha permesso di simulare bene l'esigenza di portabilità orizzontale: pur cambiando radicalmente la configurazione dei sensori Dream sarà infatti in grado di elaborare le richieste e inoltrare le risposte.

Abbiamo sviluppato in tutto due moduli di test:

`nxt_iaslab` è composto da due nodi principali: RobotCommander e RobotManager. Il primo è sviluppato in coerenza con la configurazione dei sensori, attivando le rispettive funzionalità e settando i parametri necessari a livello di driver; il secondo si occupa esclusivamente della comunicazione con agenti esterni. In relazione all'hardware in dotazione (esposto nella sezione 3.4.1), abbiamo attivato la sottoscrizione al canale "ultrasonic_sensor" per l'ultrasonico e la pubblicazione su "cmd_vel" per i motori. In via sperimentale abbiamo poi agganciato il topic "Odometry", per aggiustare con più precisione l'attività di manovra del robot. Il manager è si articola in due funzioni: la prima raccoglie la richiesta di azione dall'utente, la seconda riceve ed invia messaggi con Dream.

`CameraVision` è aggiunge la funzionalità di visione al robot. Nella pratica si tratta di un nodo extra che si preoccupa di utilizzare *OpenCV* per catturare ed elaborare le immagini percepite da una webcam omnidirezionale. In particolare lo strumento funziona in campo verde provvisto di quadrettatura bianca, magari con qualche ostacolo.

In laboratorio abbiamo messo assieme tutti i nodi e i pacchetti presentati nelle sezioni precedenti per poi sottomettere alcune interrogazioni e studiarne i risultati. L'attività si è svolta in alcune fasi:

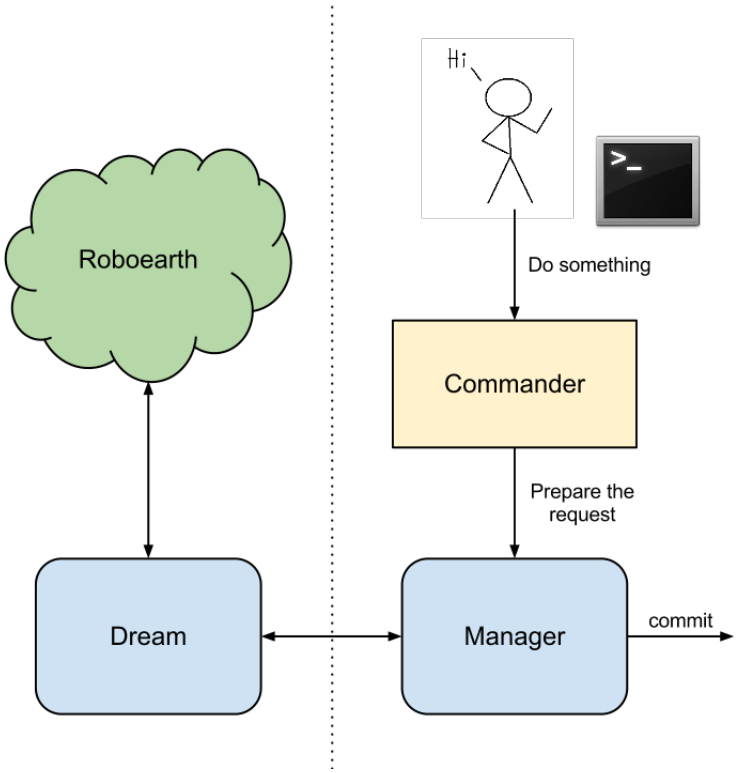

Figura 12: Il pacchetto dal punto di vista dell'utente.

- Abbiamo inizializzato i tre nodi principali in tre diversi terminali, avviandoli secondo questo ordine: *roscore*, *Dream*, *RobotManager*.
- Abbiamo poi inviato le richieste di azione tramite un quarto terminale.
- Abbiamo provato due serie di interrogazioni, la prima già presente in Roboearth, la seconda inserita da noi prima dell'esperienza e appositamente per l'occasione.
- Abbiamo ripetuto l'operazione per i due casi di studio in esame: "un semplice Ufo Robot" e "CameraVision".

In conclusione mettiamo l'accento sulla forma finale del lavoro. Dream è dunque un pacchetto indipendente, integrato con ROS, che una volta installato mette a disposizione un servizio per la lettura di risorse remote come Roboearth. Può essere infine inserito in un progetto qualunque come dipendenza esterna attraverso la sola gestione dei canali "action_request", "action_response" e facoltativamente "manager_response".

Riportiamo i dettagli sui casi di studio e sui risultati conseguiti nel prossimo capitolo (Capitolo 5).

5 CASI DI STUDIO

In questo capitolo riportiamo in dettaglio due esperienze di laboratorio con annessi risultati cercando di evidenziare le difficoltà riscontrate e gli eventuali sviluppi futuri.

5.1 CASO N.1: UN SEMPLICE UFO ROBOT

Nella prima configurazione abbiamo utilizzato un robot NXT Mindstorm dotato di ultrasonico. Come semplice scopo ci siamo posti il superamento di un ostacolo in terreno casuale attraverso il solo uso del sensore equipaggiato. Le fasi di lavoro sono state le seguenti:

- Abbiamo montato il robot con il solo sensore ultrasonico.
- Abbiamo configurato la cartella di lavoro con i relativi *setup.sh*.
- Abbiamo studiato la struttura dei messaggi necessari ai nodi.
- Abbiamo implementato le chiamate relative ai topic *Range* e *Odometry*.

Riportiamo alcuni dettagli sulla configurazione del sensore e dei motori:

RANGE: Abbiamo posizionato il sensore ultrasonico nella parte anteriore del robot, in modo che la distanza rilevata permettesse l'identificazione di eventuali ostacoli frontali. In questo caso il parametro critico è stato proprio *range*, contenuto nei messaggi di topic *Range*. Valutazioni empiriche ci hanno portato in definitiva a scegliere il valore 2.55, in accordo con la velocià media del robot al fine di garantire tempi di risposta ragionevoli.

COMMAND VELOCITY: Il robot NXT richiede un minimo di iniziaizzazione per quanto riguarda i valori di accelerazione e posizionamento delle ruote. Questo topic si occupa proprio di questo, attraverso un messaggio di tipo *Twist* e due parametri: angular e linear. Anche in questo caso abbiamo adottato valutazioni empiriche che ci hanno portato a settare entrambi i parametri a 1.0.

ODOMETRY: Questo topic ha lo scopo di identificare l'orientazione corrente del robot. Usato in via sperimentale, tale informazione è data dal valore di un parametro denominato z. Riportiamo l'idea di base in Figura 13.

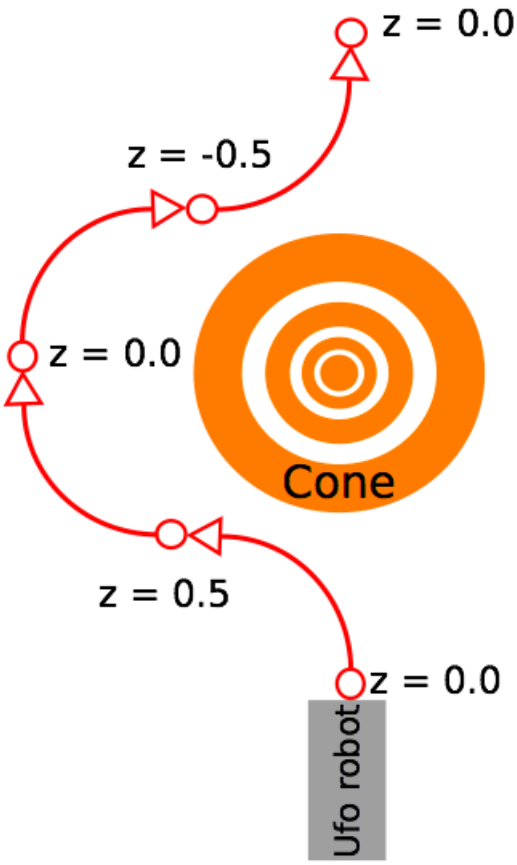

Figura 13: NXT: Procedura di schivata con Odometry.

In questa esperienza abbiamo utilizzato Dream per raccogliere la macroazione di schivata. In termini di driver abbiamo invece implementato le azioni atomiche di movimento (diritto e virata) e la manovra di schivata. Il termine "sperimentale" associato all'uso del topic Odometry è giustificato dall'accuratezza del robot durante l'esperimento: sebbene il valore di z rispetto al punto di partenza sia stato decisamente utile per correggere vari parametri, saltuari tempi di risposta eccessivamente lunghi hanno sporcato il risultato finale rendendo l'azione di movimento talvolta poco fluida.

5.2 CASO N.2: CAMERAVISION

Nella seconda configurazione abbiamo aggiunto una webcam omnidirezionale al nostro NXT. Con il driver in allegato abbiamo sfruttato due funzionalità in più: la conoscenza di una mappa a scacchiera e le informazioni fornite dal modulo di visione.

Abbiamo dunque apportato delle modifiche strutturali al robot spostando l'ultrasonico sopra il dispositivo NXT e montando la teleca-

mera sulla parte frontale, il più possibile parallela rispetto al terreno. Riportiamo la configurazione finale in figura 9.

La differenza fondamentale di questo caso sta nell'ambiente di movimento e nell'approccio del robot con gli ostacoli: la mappa è divisa in celle delineate da quadri bianchi ben rilevabili dal sensore. Gli ostacoli sono al solito rilevati tramite il topic Range, mentre un nodo aggiuntivo chiamato CameraVision si preoccupa di elaborare le informazioni della telecamera per migliorare l'accuratezza dell'orientazione e la direzione del robot.

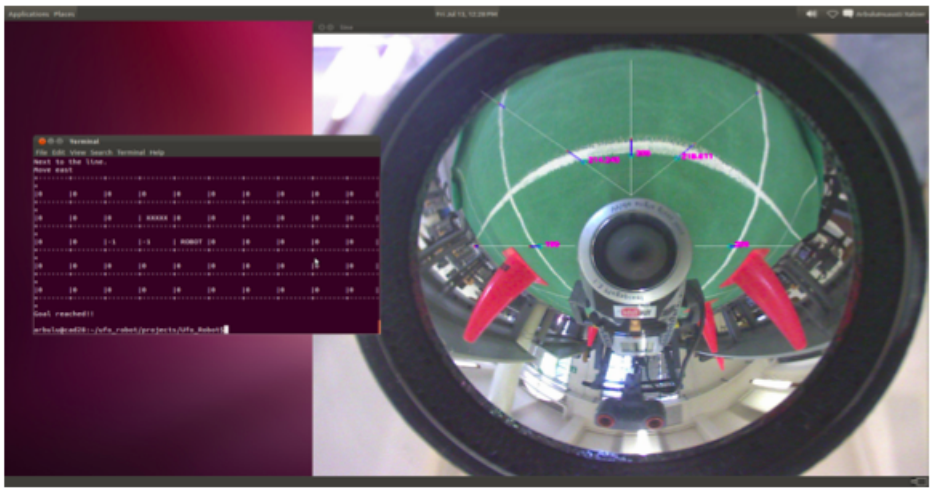

Figura 14: NXT: maschera di rilevazione tramite webcam

Come riportato in figura 14 abbiamo impostato 5 raggi per altrettante Region Of Interest (ROI). Da queste CameraVision si preoccupa di rilevare, attraverso una funzione appropriata, la distanza in pixel tra la linea bianca più vicina e l'origine del raggio. La combinazione di tutte le distanze permette al driver di inviare comandi di virata al robot in modo da rimanere in una zona il più possibile centrale rispetto alla cella.

Abbiamo impostato dunque un sistema di messaggi interno tra CameraVision e il driver per trasmettere valori utili alle funzioni sopradescritte.

FRONT_DISTANCE contiene un float a 32 bit e corrisponde virtualmente al fascio verticale della figura. Il dato del messaggio misura la distanza tra il robot e la cella successiva, utilizzato nella funzione di manovra per registrare l'avvenuto passaggio dalla cella corrente alla successiva.

ANGLE_DISTANCE contiene un float a 32 bit e corrisponde virtualmente ai due fasci obliqui nella figura. Il dato del messaggio è utilizzato nella funzione di manovra per aggiustare la direzione del robot per correggere gli slittamenti delle ruote, per esempio, in una semplice azione di *straight*.

HORIZONTAL_DISTANCE contiene un float a 32 bit e corrisponde virtualmente ai due fasci orizzontali della figura. Il dato del messaggio è utilizzato, come nel caso precedente, per aggiustare la direzione del robot nella fase di avvicinamento al centro di una cella.

Il pacchetto Dream è stato invocato per raccogliere la macroazione di movimento da un punto di inizio ad un punto di arrivo, mentre al driver è stato delegato l'onere di eseguire le azioni atomiche di movimento da una cella all'altra.

5.3 RISULTATI

Abbiamo raccolto i risultati del lavoro sottoponendo interrogazioni sia inerenti che non inerenti ai casi di studio.

La prima sottoposta è stata la macroazione "serve a drink", caricata per un robot umanoide con varie definizioni sull'ambiente e sull'utilizzo di attuatori come un braccio meccanico. Aspettandoci il fallimento da parte del nostro NXT il software non si è infatti smentito restituendo un messaggio di non fattibilità dell'azione nel suo complesso.

La seconda sottoposta è stata la macroazione "manouver", basata sul movimento, sull'individuazione di ostacoli e certamente fattibile. Nel caso di studio numero 1, relativo all'NXT dotato di solo ultrasonico, abbiamo constatato una corretta ricezione ed esecuzione l'intera macro. Nel caso di studio numero 2, relativo all'NXT dotato di webcam, abbiamo riscontrato alcuni problemi circa la lettura dell'interrogazione. Ecco perchè abbiamo inserito un hotfix che descriveremo nella sezione 5.4. Al secondo tentativo Dream è stato in grado di trasmettere la corretta action recipe che, riconosciuta dal robot, è stata portata a termine con successo da un driver intrinsicamente diverso, ma con definizione YAML molto simile al caso di studio n.1.

5.4 HOTFIX

Dalle prime fasi di test ci siamo accorti dell'importanza della fase di stesura dell'azione in formato OWL. È importante che l'azione sia specificata in modo corretto per un più coerente funzionamento rispetto agli obiettivi di portabilità: il rischio è che stringhe malscritte portino instabilità in robot leggermente diversi dal modello usato nel testdrive. Alla luce di questo abbiamo sviluppato un piccolo aggiornamento a Dream, sulle `action_utils`, che si preoccupa di semplificare queste problematiche e irrobustire il sistema. In breve abbiamo predisposto alcune funzioni aggiuntive che massaggiano l'interrogazione, sistemando la sintassi o cercando dei sinonimi.

In particolare abbiamo implementato tre tipi di utilità:

- action_strip rimuove spazi e utilizza il formato maiuscole *wordWord*
- action_fill sostituisce spazi con *underscore*
- action_synm cerca sinonimi della parola in ingresso, in lingua inglese.
- action_roll raccoglie le tre funzioni precendenti e le applica iterativamente alla richiesta in input. Supportanto i campi di ricerca aggiuntivi come "tag" e "tips".

La ricerca sperimentale dei sinonimi è resa possibile dal modulo Python "enchant"

Riportiamo lo snippet in Appendice B.3.

5.5 SVILUPPI FUTURI

Come ogni modulo software Dream è sicuramente bisognoso di estensioni e azioni di raffinamento. Il modulo di *reasoning* potrebbe per esempio essere reso più furbo, aggiungendo il supporto di una base di dati serverless come SQLite per registrare lo storico delle richieste e adattarne le interrogazioni: attualmente infatti non c'è una vera e propria gestione dei conflitti in caso di macroazioni omonime. Per quanto riguarda il funzionamento, viene infatti eseguita la prima disponiibile (che può non essere la migliore). In questo caso abbiamo scelto di aspettare in quanto questa funzionalità è tra le promesse lato server di Roboearth.

Test più estesi, magari su modelli di robot completamente diversi sarebbero interessanti per valutare ancora una volta l'orizzontalità del pacchetto. Purtroppo tale opzione è ostacolata dal costo dei macchinari che necessitano di driver molto sofisticati e che dunque poco si prestano a test rischiosi.

Una sintassi ad hoc per i vincoli di ordinamento su OWL è in via di sviluppo ma purtroppo ancora non supportata dai parser utilizzati in questa tesi[1]. Migrare dalla gestione attuale ad uno standard riconosciuto (una volta ritenuto stabile) è sicuramente un miglioramento da tenere in considerazione.

In aggiunta al modello nodo a nodo, ROS mette a disposizione dei servizi con gestione dei messaggi ed ack completamente ottimizzata. Tale opzione è stata scartata in questa occasione in quanto piuttosto complessa e spesso rinnovata dalle frequenti release del framework. Sarebbe tuttavia un'opzione interessante in vista di un miglioramento dei tempi di risposta.

1 Il parser nominato è uno dei più grandi e robusti della comunità Python: https://github.com/RDFLib

5.6 CONCLUSIONI

In questa tesi abbiamo presentato lo studio, la progettazione e lo sviluppo di Dream: un nodo ROS che mette a disposizione una serie di canali per una facile comunicazione con il core del progetto Roboearth. Tramite Dream siamo in grado di comunicare con una grande base di dati nella quale sono memorizzate *action recipes* scritte in OWL. Abbiamo dimostrato come il valore aggiunto nell'utilizzo della Rete consista nell'arricchimento di conoscenza su robot che siano in grado di svolgere autonomamente dei passi base, ma che non siano stati progettati per un preciso task. Mentre i risultati in laboratorio sono stati incoraggianti ci aspettiamo futuri aggiornamenti al pacchetto software per migliorare le facoltà di reasoning in particolare nella risoluzione dei conflitti.

Parte III

APPENDICE

In appendice riportiamo materiale aggiuntivo riguardo il lavoro svolto. Includiamo complementi all'installazione e snippet di approfondimento.

A

COMPLEMENTI ALL'INSTALLAZIONE

A.1 PORTING DI ROBOEARTH SU ROS FUERTE: DETTAGLIO

Una volta ultimato il lavoro di *porting* abbiamo registrato un file .diff segnalando il lavoro agli sviluppatori, in attesa di una patch ufficiale.

Un file diff mette in evidenza le differenze tra due progetti.

LINK AL FILE DIFF: http://db.tt/o1LrA7s0.

LINK AL FILE ROSINSTALL: http://db.tt/4eeX9En6.

Listing 5: Calcolo del file .diff

```
#!/bin/bash
# getdiff.sh
# esclude:
#   - righe e spazi vuoti
#   - file nascosti
#   - cartelle non interessanti
#   - commenti

diff -Bwur --exclude '.*' --exclude bin --exclude workspace \
-I '/\*(?>(?:(?>[^*]+)|\*(?!/))*)\*/' \
Roboearth_official/stacks/roboearth \
Roboearth_merged/stacks/roboearth
```

A.2 PORTING DELLO STACK NXT SU ROS FUERTE: DETTAGLIO

Particolare attenzione è stata dedicata al pacchetto *nxt_msgs* in quanto rilevato critico per il funzionamento dei sensori del robot. Alleghiamo una traccia del ".diff" con i cambiamenti effettuati.

DIFFERENZE SOSTAZIALI: siamo intervenuti principalmente nei sorgenti relativi alla gestione dei messaggi in NXT. In particolare "Accelerometer", "Color", "Gyro", "Contact", "JoinCommand" e "Range". Tutti condividono una descrizione in Python molto simile per quanto riguarda la struttura di supporto dati, sotto *roslib*. Tale libreria è stata aggiornata in seguito al passaggio da ROS Electric a ROS Fuerte, corrompendo alcuni riferimenti utilizzati nei moduli in oggetto. Abbiamo rilevato un problema simile (e risolto) riguardo al Turtlebot, sempre sotto ROS: A questo ci siamo ispirati per rilasciare la patch al pacchetto aggiungendo il supporto alla libreria *genpy* e alla relativa struttura.

Le modifiche in codice sono state le seguenti, ripetute per tutti i pacchetti sopra elencati:

Listing 6: Patch al pacchetto NXT

```
+ import genpy.message # Flavio
...
- _struct_I = roslib.message.struct_I
+ _struct_I = genpy.message.struct_I # Flavio
```

A.3 FORMATO DEI DATI: ESEMPI

Per il corretto funzionamento del robot è necessario dichiarare alcuni parametri riguardo le porte utilizzate e il loro scopo. Nel nostro caso utilizziamo due slot per i motori e uno per il sensore ultrasonico. In caso di webcam invece non è necessario aggiornare nulla in quanto trattata come dispositivo extra disconnesso dalle porte principali.

Listing 7: Esempio di file YAML: configurazione di NXT Mindstorm

```
nxt_robot:
  - type: motor
    name: r_wheel_joint
    port: PORT_A
    desired_frequency: 20.0

  - type: motor
    name: l_wheel_joint
    port: PORT_B
    desired_frequency: 20.0

  - type: ultrasonic
    frame_id: ultrasonic_link
    name: ultrasonic_sensor
    port: PORT_4
    spread_angle: 0.2
    min_range: 0.01
    max_range: 2.5
    desired_frequency: 5.0
```

Roboearth accetta e ritorna i dati in formato JSON ovvero impacchettati in dizionari le cui chiavi sono specificate nella documentazione ufficiale: http://api.roboearth.org/documentation/dev. Riportiamo qui una risposta di Roboearth ricavata da una chiamata di tipo GET.

Listing 8: Esempio di file JSON: test su Roboearth

```
[
    {
        "rating": "1",
        "modified_by": "uforobot",
        "description": "A first test for our macro action.",
        "author": "uforobot",
        "recipes": [
            {
                "timestamp": 1353682691192,
                "recipe": "
                <?xml version=\"1.0\"?>
                <!DOCTYPE rdf:RDF
                [ <!ENTITY roboearth_path 'http://ias.cs.tum.
                    edu/kb/'>
                        ...
                ]
                </rdf:RDF>"
            }
        ],
        "id": "robotics.dei.unipd.manouver"
    }
]
```

Le azioni sono descritte e memorizzate formalmente in file .rdf includendo il linguaggio OWL per rappresentare gli insiemi di azioni e subazioni. Di seguito riportiamo un estratto di azione scaricata tramite l'interfaccia web di roboearth al solito indirizzo http://api.roboearth.org/.

Listing 9: Esempio di file RDF

```
<?xml version="1.0"?>
<!-- Header -->
<!DOCTYPE rdf:RDF [
    <!ENTITY roboearth_path 'http://ias.cs.tum.edu/kb/'>
    <!ENTITY kb "http://ias.cs.tum.edu/kb/" >
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY owl2xml "http://www.w3.org/2006/12/owl2-xml#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#
        " >
    <!ENTITY knowrob "http://ias.cs.tum.edu/kb/knowrob.owl#"
        >
    <!ENTITY roboearth "http://www.roboearth.org/kb/roboearth
        .owl#" >
]>

<!-- Action Recipe -->
<rdf:RDF xmlns="http://www.roboearth.org/kb/roboearth.owl#"
     xml:base="http://www.roboearth.org/kb/roboearth.owl"
     xmlns:owl2xml="http://www.w3.org/2006/12/owl2-xml#"
     xmlns:roboearth="http://www.roboearth.org/kb/roboearth.
         owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:knowrob="http://ias.cs.tum.edu/kb/knowrob.owl#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:owl="http://www.w3.org/2002/07/owl#">
    <owl:Ontology rdf:about="">
      <owl:imports rdf:resource="&roboearth_path;roboearth.
          owl"/>
    </owl:Ontology>

    <owl:Class rdf:about="http://robotics.dei.unipd.it/
        actions#manouver">
        <rdfs:label>manouver</rdfs:label>
        <rdfs:comment>The manouver class.</rdfs:comment>
        <rdfs:subClassOf>
            <owl:Class>
                <owl:intersectionOf rdf:parseType="Collection
                    ">
                    <owl:Restriction>
                        <owl:onProperty rdf:resource="&
                            knowrob;subAction"/>
```

```
                        <owl:Annotation rdf:resource="1"/>
                        <owl:someValuesFrom rdf:resource="#
                            straightToObstacle"/>
                    </owl:Restriction>
                    <owl:Restriction>
                        <owl:onProperty rdf:resource="&
                            knowrob;subAction"/>
                        <owl:Annotation rdf:resource="2"/>
                        <owl:someValuesFrom rdf:resource="#
                            dodgeTheObstacle"/>
                    </owl:Restriction>
                    <owl:Restriction>
                        <owl:onProperty rdf:resource="&
                            knowrob;subAction"/>
                        <owl:Annotation rdf:resource="3"/>
                        <owl:someValuesFrom rdf:resource="#
                            goToGoal"/>
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </rdfs:subClassOf>
    </owl:Class>

</rdf:RDF>
```

B

SNIPPET DI APPROFONDIMENTO

B.1 MIND.PY: MAINCALLBACK

Riportiamo il callback principale del nodo *mind.py*, per la descrizione completa si faccia riferimento alla sezione 4.1.1.

Listing 10: Callback principale del nodo mind

```
def mainCallback(data):
  # Inizializzazione e attesa del semaforo
  global ACTION_SYNC
  response = "None"
  while ACTION_SYNC == 0:
      print "+ Waiting ACTION_SYNC"
      rospy.sleep(1)
  print
  # Lettura messaggio e caso base
  rospy.loginfo(rospy.get_name() + "\n+ Requested action: %s",
      data.data)
  requested_action = str(data.data)
  status = checkKnownAction(robot_actions, requested_action)
  # Pulizia interrogazione
  if not status:
      requested_action = action_roll(robot_actions,
          requested_action)
  found_action = getAction(robot_actions, requested_action,
      field='name')
  if found_action is not None:
      print "+ Action found: " + bcolors.OKGREEN + str(
          found_action) + bcolors.ENDC
      response = found_action
  else:
      print
      print bcolors.HEADER + "+ Initializing remote request" +
          bcolors.ENDC
      # Messa in campo di Roboearth
      macro = connect_to_roboearth(requested_action)
      if macro is not None:
          print macro
          for action in macro:
              data.data = action
              # Per ogni subazione, richiama la ricorsione
              mainCallback(data)
          sys.exit(0)
  if status:
      ACTION_SYNC = 0
  print "+ Forwarding action request... "
```

```
    # Pubblicazione finale
    pub = rospy.Publisher('action_response', String)
    for i in range(0, TRYALS):
        pub.publish(response)
        rospy.sleep(1)
```

B.2 CONNECTOR.PY: CONNECTO_TO_ROBOEARTH

Il nodo `connector.py` è responsabile della comunicazione con Roboearth. Le sue attività consistono nell'interrogare la base di dati remota e ritornare i risultati in formato di semplice lista. Per tutti i dettagli riferirsi alla sezione 4.1.2

Listing 11: Funzione principale del nodo connector

```
def connect_to_roboearth(query=DEFAULT_QUERY):
    # Preparazione della connessione
    print "+ Connecting to roboearh..."
    url = "http://api.roboearth.org/api/recipe"
    query_wrapper = "SELECT source FROM CONTEXT source {x} rdfs
        :label {L} where L LIKE \"*%s*\"" % query
    print query_wrapper
    payload = { "query" : query_wrapper }
    headers = {'content-type': 'application/json'}
    # Tentativo di connessione POST
    try:
        r = requests.post(url, data=json.dumps(payload),
            headers=headers)
        r.raise_for_status()
    except RequestException:
        print "+ We noticed the following error: %s. Exiting."
            % r.status_code
        return None
    # Ricezione del risultato
    print "+ Data received, parsing the rdf..."
    recipe = r.json[0]['recipes'][0]['recipe']
    # Parsing e riordinamento
    g = rdflib.Graph()
    g.parse(data=recipe, format="application/rdf+xml")
    s = g.serialize(format='n3')
    result = get_subactions(g, "owl:Annotation" in s)
    print "Done."
    return [r[1] for r in result]
```

B.3 ACTION_UTILS.PY: DETTAGLIO

Questo modulo mette a disposizione una serie di utili funzioni per l'irrobustimento dell'interrogazione, in particolare si occupa della cor-

rezione di eventuali errori di battitura o di sintassi. Per ulteriori dettagli riferirsi alla sezione 4.1.3.

Listing 12: Utils in dettaglio

```
STANDARD_ACTION = "straight"
""" Una ricerca Pythonica """
def checkKnownAction(robot_actions, requested_action=
    STANDARD_ACTION, field='name'):
    keywords = [action[field].split(" ") for action in
        robot_actions]
    status = any([requested_action in key for key in keywords])
    printlog("+ Robot knowledge about the requested action: ",
        status)
    return status

""" Estrapola la risorsa richiesta dalla lista di azioni """
def getAction(robot_actions, digested_action, field):
    for action in robot_actions:
        if digested_action in action[field]:
            return action['name']
    try:
        index = next(index for (index, d) in enumerate(
            robot_actions) if (digested_action in d[field]))
    except StopIteration:
        return None
    return robot_actions[index]['name']
```

Listing 13: Hotfix alle utils

```
lower = lambda s: s[:1].lower() + s[1:] if s else ''

def action_strip(action):
    print "+ Stripping action request..."
    action = lower(action.title().replace(" ", ""))
    return action

def action_fill(action):
    print "+ Filling action request..."
    action = action.replace(" ", "_")
    return action

def action_synm(action):
    print "+ Rerolling action request..."
    action_words = action.split()
    d = enchant.Dict(LANGUAGE)
    action_words = [choice(d.suggest(word)) for word in
        action_words]
    action = action_strip("".join(action_words))
    return action

ACTION_DIGEST = {
```

```
    1: action_strip,
    2: action_fill,
    3: action_synm,
}
ACTION_RESEARCH = {
    1: 'name',
    2: 'tags',
    3: 'tips',
}
""" ACTION_RESEARCH e ACTION_DIGEST massaggiano la
    action_request per irrobustire la validazione """
def action_roll(robot_actions, requested_action):
    for idx, key in enumerate(ACTION_RESEARCH):
        field = ACTION_RESEARCH[key]
        print
        print bcolors.OKBLUE + "+ field: " + field + bcolors.
            ENDC
        for idx, key in enumerate(ACTION_DIGEST):
            digested_action = ACTION_DIGEST[key](
                requested_action)
            print "+ Action result: %s" % digested_action
            print "+ Done"
            status = checkKnownAction(robot_actions,
                digested_action, field=field)
            if status:
                return digested_action
    return requested_action
```

B.4 DRIVER NXT MINDSTORM: CASO DI STUDIO N.1

Durante i *testdrive* abbiamo equipaggiato l'NXT con un sensore ultrasonico e una webcam. Di seguito riportiamo i costruttori del modulo driver con la rispettiva inizializzazione dei messaggi.

Listing 14: Costruttore n.1

```
RobotDriver::RobotDriver():
  l_scale_(1.0), a_scale_(1.0), range(2.55)
{
  vel_pub_ = n.advertise<geometry_msgs::Twist>("cmd_vel", 1000)
      ;
  ultrasonic = n.subscribe(
      "ultrasonic_sensor",
      1000,
      &RobotDriver::ultrasonicCallback,
      this
  );
}
```

Listing 15: Supporti del driver 1

```
/*
 *  Avanti dritto
 *  @param duration: durata di moto
 */
void RobotDriver::straightForTime(ros::Duration duration){
    ros::Time begin = ros::Time::now();
    ros::Time endTime = begin + duration;
    ros::Rate loop_rate(10);
    while ( ros::Time::now() <= endTime ) {
        publish( 0.0, 0.12);
        ros::spinOnce();
    }
    stop();
}

/*
 *  Avanti dritto
 *  @param dinstance: distanza di moto (odometry)
 */
void RobotDriver::straightFor( double distance ) {
  std::cout << "Straight for " << distance << std::endl;
  double startX = this->position[0];
  double difference = 0.0;
  while (difference <= distance ) {
      difference = this->position[0] - startX;
      readOdometry();
      publish(0, 0.1);
      ros::spinOnce();
  }
}

/*
 *  Esempio di virata: sinistra
 */
void RobotDriver::left90InPlace() {
  std::cout << "Turn left 90Â°: " << std::endl;
  double startRotation = this->orientation[2];
  double rotation = 0.0;
  while (rotation <= 0.5) {
      readOdometry();
      rotation = this->orientation[2] - startRotation;
      publish(0.5, 0.1);
      ros::spinOnce();
  }
}
```

B.4.1 *Esempi di azioni atomiche*

Due esempi pratici di ciò che intendiamo con “azioni atomiche“. Entrambe utilizzano utilità di basso livello messe a disposizione dal driver.

Listing 16: Azioni atomiche: caso di studio 1

```
/*
 *  Con lettura da ultrasonico
 */
void RobotDriver::straightToCone( double distance ) {
  std::cout << "Straight to the cone." << std::endl;
  while (range >= distance) {
    std::cout << range << std::endl ;
    readUltrasonic();
    publish(0,0.1) ;
    ros::spinOnce();
  }
}

/*
 *  Utilizzo azioni di basso livello del driver
 */
void RobotDriver::dodgeTheObstacle( double distance ) {
  robot.left90InPlace();
  robot.straightFor(0.02);
  robot.right90InPlace();
  robot.straightFor(0.15);
  robot.right90InPlace();
  robot.left90InPlace();
}
```

B.5 DRIVER NXT MINDSTORM: CASO DI STUDIO N.2

Listing 17: Costruttore n.2

```
RobotDriver::RobotDriver():
  l_scale_(1.0), a_scale_(1.0), range(2.55),
  whiteLine(false), estimatedNorth(0.5), estimatedSouth(-0.5),
  estimatedEast(0.0), estimatedWest(1.0), robotOrientation(2),
  timeForRight90(ros::Duration(1.705)),
  timeForLeft90(ros::Duration(1.80))
{

  /*
   * Variabili utili per il supporto della mappa
   */
  position[0] = position[1] = position[2] = 0.0;
  orientation[0] = orientation[1] = 0.0;
  orientation[2] = orientation[3] = 0.0;

  /*
   * Inizializzazione messaggi principali
   */
  vel_pub_=n.advertise<geometry_msgs::Twist>("cmd_vel", 1000);
  ultrasonic=n.subscribe(
```

```
        "ultrasonic_sensor",
        1000,
        &RobotCommander::ultrasonicCallback,
        this
    );
    direction_line=n.subscribe(
        "horizontal_difference",
        1000,
        &RobotCommander::directionLineCallback,
        this
    );
    orientation_line=n.subscribe(
        "angle_difference",
        1000,
        &RobotCommander::orientationLineCallback,
        this
    );
}
```

Listing 18: Supporti del driver (estratto)

```
/*
 * Gestione fasci orizzontali
 */
void RobotCommander::correctDirectionCamera() {
  waitDirectionLine();
  ros::Rate loopRate(20);
  while ( directionLine != RC_OK ) {
    if ( directionLine == RC_MUST_LEFT ) {
      // A sinistra
      publish( 0.1, 0.1 );
    } else {
      // A destra
      publish( -0.1, 0.1 );
    }
    ros::spinOnce();
    loopRate.sleep();
    stop();
    loopRate.sleep();
  }
  stop();
  stop();
}

/*
 * Utilizza il fascio verticale per identificare
 * la prossima linea bianca (proximity)
 */
void RobotCommander::straightToLineCamera() {
  waitProximityLine();
  ros::Rate loopRate(20);
```

```
    publish(0, 0.1);
    ros::spinOnce();
    publish(0, 0.1);
    ros::spinOnce();
    while ( proximityLine != 1.0 ) {
      publish(0, 0.1);
      ros::spinOnce();
      loopRate.sleep();
    }
    std::cout << "Next to the line." << std::endl;
    stop();
}

/*
 * Uso combinato di angle, front e horizontal distance
 */
void move(RobotDriver &robot){
  std::cout << "+ Manouvering... " << std::endl;
  robot.correctOrientationCamera();
  robot.straightToLineCamera();
  while(robot.inTheCenterCamera() != 1)
  {
    robot.straightForTime(ros::Duration(0.3));
    robot.correctDirectionCamera();
  }
}
```

Listing 19: Azioni atomiche: caso di studio 1

```
/*
 * La manovra include l'uso di ultrasoico e telecamera
 */
void straightToCone(RobotDriver &robot){
  while (robot.checkObstacle()) {
    std::cout << "+ Manouvering... " << std::endl;
    move(robot)
  }
}

/*
 * La mappa \'e memorizzata in una matrice quadrata
 * che calcola il prossimo passo per scartare l'ostacolo
 * ----------------------------------------------------
 * Questa azione atomica ha un nome diverso rispetto
 * al caso n.1, ma \'e registrata con gli stessi tag
 */
void dodgeTheCone(RobotDriver &robot){
  Map mappa;
  mappa.emptyMap();
  while (!mappa.isRobotAtGoal() && step < 5) {
    int direzione = mappa.getNextStepA();
```

```
      cout << "Step " << step++ << ": " << direzione << endl;
      robot.setRobotOrientation(direzione);
      mappa.calculatePath();
      robot.move();
    }
  }
```

BIBLIOGRAFIA

[1] Jon Bentley. *Programming Pearls*. Addison–Wesley, Boston, MA, USA, seconda edizione, 1999.

[2] Giuseppe Caravita. Il welfare nelle mani dei robot. *Nóva, Sole 24 Ore*, pp. 46–47, Luglio 2012. http://nova.ilsole24ore.com.

[3] Likeddatatools Community. Introducing linked data and the semantic web, 2012. http://www.linkeddatatools.com/semantic-web-basics.

[4] ROS Community. Ros wiki, 2012. http://www.ros.org/wiki/.

[5] Luca dello Iacovo. Connessioni robotiche. *Nóva, Sole 24 Ore*, pp. 45–46, Febbraio 2011. http://nova.ilsole24ore.com.

[6] Willow Garage. Xml robot description format, 2011. http://www.ros.org/wiki/urdf/XML.

[7] W3C OWL Working Group. Web ontology language document overview, 2009.

[8] Donald E. Knuth. Computer Programming as an Art. *Communications of the ACM*, 17(12):667–673, December 1974.

[9] Trung Ngo Lam, Haeyeon Lee, Katsuhiro Mayama, e Makoto Mizukawa. Evaluation of Commonsense Knowledge for Intuitive Robotic Service. *IEEE International Conference on Robotics and Automation*, Maggio 2012.

[10] Daniel Di Marco, Moritz Tenorth, Kai Häussermann, Oliver Zweigle, e Paul Levi. Roboearth action recipe execution, 2010.

[11] Ian Sommerville. *Software Engineering*. Addison-Wesley, Boston, MA, USA, quarta edizione, 1992.

[12] Moritz Tenorth, Alexander Perzylo, Reinhard Lafrenz, e Michael Beetz. The roboearth language: Representing and exchanging knowledge about actions, objects, and environments. 2011.

[13] Markus Waibel, Michael Beetz, Javier Civera, Raffaello D'Andrea, Jos Elfring, Dorian Gálvez-López, Rob Janssen, J.M.M. Montiel, Alexander Perzylo, Bjorn Schießle, Moritz Tenorth, Oliver Zweigle, René van de Molengraft, e Kai Häussermann. Roboearth. *IEEE Robotics & Automation Magazine*, Giugno 2011.

DICHIARAZIONE

Il lavoro descritto in questo documento è stato svolto dall'autore stesso come tesi di laurea Magistrale in Ingegneria Informatica.

Padova, 11 Dicembre 2012

Flavio Marcato