ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
SEDE DI CESENA
SECONDA FACOLTÀ DI INGEGNERIA CON SEDE A CESENA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Tecnologie per la collaborazione sociale tra robot Mindstorm: LeJOS & ReSpecT

Elaborato in
Sistemi Distribuiti

Relatore
Andrea Omicini

Presentata da
Cornel Moisuc

Sessione III°
Anno Accademico 2010-2011

A mia madre

Sommario

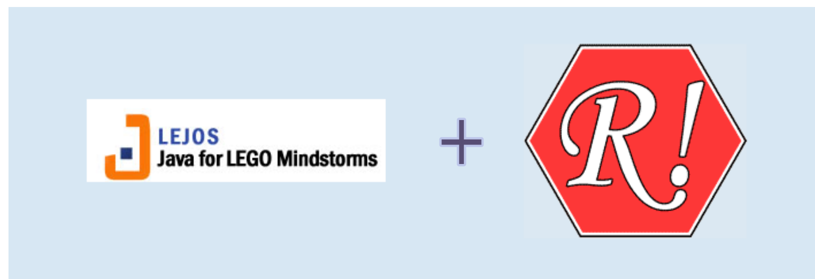

Figura 1: Logo LeJOS e ReSpecT

Obiettivo

Questo elaborato ha l'obiettivo di sperimentare nuove tecnologie per la collaborazione sociale basata sul linguaggio di coordinazione ReSpecT tra robot LEGO Mindstorms NXT. Si valutano i vantaggi dell'integrazione di tale collaborazione con LeJOS, piuttosto che con le API iCommand. Il benchmark viene realizzato confrontando le prestazioni dei vari approcci in alcuni scenari già discussi in lavori precedenti.

Contenuti

Nel capitolo 1 si fa un'introduzione al mondo della robotica.

Nel capitolo 2 si descrive la tecnologia LEGO Mindstorms.

Nel capitolo 3 si espone in sintesi la coordinazione attraverso ReSpecT.

Nel capitolo 4 si descrivono le due tecnologie a confronto: iCommand e LeJOS NXJ.

Nel capitolo 5, che rappresenta il cuore di questo lavoro, viene descritto il framework svilppato per attuare il passaggio da iCommand a LeJOS NXJ. Vengono poi descritte le soluzioni proposte per i case study.

Nel capitolo 6 si fanno delle considerazioni conclusive riguardo ai vantaggi della nuova tecnologia e ad eventuali miglioramenti.

Indice

Indice i

1 Introduzione al mondo della robotica 1

1.1 Scenario 1

1.2 Intelligenza Artificiale 2

1.3 I robot 4

1.4 Robotica 6

1.4.1 Le leggi della robotica 6

1.4.2 Applicazioni della robotica 7

2 Tecnologia LEGO Mindstorms 9

2.1 Panoramica 9

2.2 Il brick NXT 11

2.3 I sensori 12

2.4 I servomotori 14

3 ReSpecT 17

3.1 Linguaggio di coordinazione 17

3.2 Tuple Space 18

3.3 Tuple Center 19

4 Dalle API iCommand a LeJOS 21

4.1 Le API iCommand 22

4.2 LeJOS NXJ 22

5 Case studies con LeJOS 27

5.1 Framework 27

5.2 Case study: Free casual roaming 32

5.2.1 Algoritmo 32
5.2.2 Risultati 36
5.3 Case study: Come this way 36
5.3.1 Algoritmo 37
5.3.2 Risultati 41
5.4 Case study: Help me 42
5.4.1 Algoritmo 42
5.4.2 Risultati 47
5.5 Case study: Listen to me 47
5.5.1 Algoritmo 47
5.5.2 Risultati 50

6 Conclusioni **51**
6.1 La fase di sviluppo 51
6.2 Analisi delle prestazioni, limiti e miglioramenti 52

Bibliografia **53**

Capitolo 1

Introduzione al mondo della robotica

1.1 Scenario

“ Quando considero i tuoi cieli, che sono opera delle tue dita, la luna e le stelle che tu hai disposte, che cosa è l’uomo, perché te ne ricordi, e il figlio dell’uomo, perché lo visiti? Eppure tu lo hai fatto di poco inferiore a DIO, e lo hai coronato di gloria e di onore. Lo hai fatto regnare sulle opere delle tue mani e hai posto ogni cosa sotto i suoi piedi! ”

Salmo 8:3-6 [1]

Il termine robot[1] è stato introdotto nel linguaggio comune universale dallo scrittore Karel Čapek[2], il quale lo usò nella sua opera R.U.R. [11]. Nel suo dramma fantascientifico Čapek narra come i robot vengono costruiti nella fabbrica Rossum. Si tratta di esseri artificiali fatti di materia organica, più precisamente androidi[3], con capacità di pensare. Queste creature inizialmente sembrano felici di poter lavorare per gli uomini, ma una loro successiva ribellione conduce alla estinzione della razza umana. Dopo aver finalizzato il manoscritto, Čapek si è reso conto di aver realizzato una ver-

[1] il termine *robot* deriva dal ceco *robota* che significa *lavoro pesante, forzato*

[2] Karel Čapek (9 gennaio 1890 - Praga, 26 dicembre 1938) è stato un giornalista, scrittore e drammaturgo ceco

[3] *androide* - essere artificiale con sembianze umane. Il termine deriva dal greco *anèr, andròs*, 'uomo'

sione moderna della leggenda ebraica del *golem*. Secondo questa, un golem era un gigante di argilla che poteva essere usato come servo, impiegato per svolgere lavori pesanti e come difensore del popolo ebraico dai suoi persecutori. Esso poteva essere evocato attraverso le arti magiche. Era dotato di una straordinaria forza e resistenza ed eseguiva alla lettera gli ordini del suo creatore, di cui diventava una specie di schiavo. Tuttavia era incapace di pensare, di parlare e di provare qualsiasi tipo di emozione, perché era privo di un'anima e nessuna magia fatta dall'uomo sarebbe stata in grado di fornirgliela.

Al giorno d'oggi un robot rappresenta una macchina costruita per eseguire azioni stabilite da comandi diretti dell'uomo o in maniera autonoma usando l'intelligenza artificiale[4]. Le sue azioni hanno l'utilità di assistere o addirittura sostituire l'uomo in lavori diversi, come ad esempio nella fabbricazione, costruzione, manipolazione di materiali pesanti e pericolosi, in ambienti proibitivi o non compatibili con la condizione umana, oppure semplicemente per liberare l'uomo da impegni. Per questo motivo i robot vengono sfruttati nei più svariati contesti lavorativi e si integrano fra loro e con l'uomo, creando una catena di produzione sempre più efficace.

Viene spontaneo chiedersi quanto sia vantaggioso l'utilizzo di un sistema con più robot specializzati per eseguire ognuno una micro-attività per il conseguimento di un obiettivo comune e se la realizzazione di tali infrastrutture sia di facile implementazione. Per rispondere a questo tipo di quesiti, sono nate branche della robotica[5], che studiano le modalità d'interazione, comunicazione e coordinazione fra robot.

Il presente lavoro si concentra sull'analisi di scenari che riguardano la coordinazione sociale fra robot. In particolare, vengono ripresi alcuni casi di studio di lavori precedenti e viene valutata l'efficienza delle soluzioni proposte.

1.2 Intelligenza Artificiale

Gli sviluppi che interessano la nascita dell'intelligenza artificiale avvengono attorno alla metà del Novecento per opera di Alan Turing[6], che oltre al modello ideale di *calcolatore automatico universale* (la macchina di Tu-

[4] *intelligenza artificiale* - vedi sezione 1.2

[5] *robotica* - vedi sezione 1.4

[6] Alan Mathison Turing (Londra, 23 giugno 1912 - Wilmslow, 7 giugno 1954) è stato un matematico, logico e crittanalista britannico, considerato uno dei padri dell'informatica e uno dei più grandi matematici del XX secolo.

ring), propose il *gioco dell'imitazione*[7], ossia un paradigma per stabilire se una macchina è 'intelligente'. Nel suo noto articolo *Computing Machinery and Intelligence* (1950), egli suggeriva di porre un osservatore di fronte a due telescriventi. Una delle due è comandata da un uomo, l'altra da un calcolatore programmato in modo da fingere di essere una persona umana. Quando non si riuscirà a distinguere il calcolatore dall'interlocutore umano, allora si potrà dire che il calcolatore è 'intelligente'.

Il termine Intelligenza Artificiale (IA) venne proposto nel 1956 da John McCarthy[8], in occasione di uno storico incontro organizzato presso il Darthmouth College[9]. L'IA si propone di indagare i meccanismi che sono alla base della cognizione umana, in primo luogo il ragionamento logico-matematico, la capacità di risolvere problemi e la comprensione del linguaggio naturale, con il fine dichiarato di riprodurli per mezzo di elaboratori elettronici sufficientemente potenti. Questa indagine incomincia col determinare cosa significhi pensare. Ironicamente, nonostante tutti siano d'accordo che gli esseri umani sono intelligenti, nessuno è ancora riuscito a dare una definizione soddisfacente di intelligenza. Proprio a causa di ciò, lo studio dell'IA si divide in due correnti:

- la prima, detta **intelligenza artificiale forte**, ritiene che un computer correttamente programmato possa essere veramente dotato di una intelligenza pura, non distinguibile in nessun senso importante dall'intelligenza umana. L'idea alla base di questa teoria è il concetto che risale al filosofo empirista inglese Thomas Hobbes[10], il quale sosteneva che ragionare non è nient'altro che calcolare: la mente umana sarebbe dunque il prodotto di un complesso insieme di calcoli eseguiti dal cervello

- la seconda, detta **intelligenza artificiale debole**, sostiene che un computer non sarà mai in grado di eguagliare la mente umana, ma potrà solo arrivare a simulare alcuni processi cognitivi umani, senza riuscire a riprodurli nella loro totale complessità. Il calcolatore non

[7] noto anche come *il test di Turing*

[8] John McCarthy (Boston, 4 settembre 1927 - Stanford, 24 ottobre 2011), è stato un informatico statunitense che ha vinto il Premio Turing nel 1971 per i suoi contributi nel campo dell'IA. Tra le altre cose ha ideato il linguaggio logico Lisp.

[9] è un'università a Hanover nel New Hampshire, Stati Uniti d'America che mantiene il nome di College per motivi storici e nostalgici

[10] Thomas Hobbes (Malmesbury, 5 aprile 1588 - Hardwick Hall, 4 dicembre 1679) è stato un filosofo britannico, autore del volume di filosofia politica intitolato *Leviatano* (1651)

è visto come l'analogo di un cervello biologico, ma come uno strumento che può essere di grande aiuto nella comprensione dei processi cognitivi.

Uno dei principali campi di applicazione dell'intelligenza artificiale è certamente il campo della robotica[11].

1.3 I robot

In base alle capacità possedute, i robot si possono dividere in tre categorie:

1. **Robot di prima generazione:** sono in grado semplicemente di eseguire sequenze prestabilite di operazioni indipendentemente dalla presenza o dall'intervento dell'uomo.

2. **Robot di seconda generazione:** hanno la capacità di costruire un' immagine (modello interno) del mondo esterno, di correggerla e perfezionarla continuamente. Inoltre, sono in grado di scegliere la migliore strategia di controllo e di finire ciò che gli è stato programmato, malgrado la presenza di fenomeni di disturbo non prevedibili a priori. Il NXT di LEGO Mindstorms fa parte di questa categoria.

3. **Robot di terza generazione:** hanno un'intelligenza artificiale e perciò sono capaci di costruire nuovi algoritmi e di verificarne la coerenza.

I robot sono costituiti da una serie di componenti diverse, così definite: *parti meccaniche, sistema di trasmissione, sistema di attuazione, elettronica di controllo e rilevazione, sistema di calcolo e controllo, software comportamentale.*

- **le componenti meccaniche** individuano le parti strutturali della macchina. Queste vengono a loro volta suddivise in due categorie: link e giunti. I primi costituiscono il corpo del robot, i secondi le articolazioni mobili. Due sono i tipi di giunti base: rotativi e lineari. Alla progettazione della meccanica di un robot contribuiscono specifiche prestazionali ed operative, determinate in base alla applicazione cui esso è destinato. A seconda della combinazione di giunti e di link, si potrà parlare di diverse categorie di robot quali: *seriale, seriale diramato, parallelo, ibrido.*

[11] *robotica* - vedi sezione 1.4

- **il sistema di trasmissione** si occupa di trasmettere l'energia prodotta da un sistema di attuazione alle articolazioni del robot. Esistono diverse tecnologie tramite le quali tale trasmissione può essere determinata: diretta (il motore è sull'asse del giunto), a cinghia, a cavi ecc.

- **il sistema di attuazione** è la sorgente di energia che fornisce movimento al robot. Gli attuatori si distinguono in due classi: lineari e rotativi. I primi possono agire in senso alternato lungo una linea di movimento, i secondi ruotare attorno un asse. Inoltre, l'attuazione si distingue sia per classificazione fisica del mezzo di trasmissione dell'energia (elettrica, idraulica, pneumatica) che per specifico principio di trasduzione.

- **l'elettronica** di un robot è forse la componente con maggiori varianti possibili. Essa si distingue in due parti: la componente di acquisizione e quella di attuazione. La prima serve a raccogliere informazioni ambientali (quali la posizione del robot, forze di contatto, accelerazioni, visione, temperatura, ecc.), la seconda a trasformare le indicazioni del sistema di controllo in segnali elettrici idonei a pilotare i motori. I sistemi di attuazione (driver) sfruttano una relazione diretta tra una variabile elettrica controllabile e l'azione meccanica da esercitare. Nel caso dei motori elettrici, generalmente si controlla la relazione tra la tensione-corrente presentata al motore e la coppia/velocità di esercizio.

- **il controllo** di un robot coordina i segnali elettronici rilevati dal sistema di percezione al fine di produrre segnali di movimento da trasdurre in opportuni comandi elettrici per i motori. Sebbene originariamente siano stati creati anche meccanismi di controllo meccanici, due categorie di controlli sono attualmente in uso: *controlli elettronici* e *controlli digitali*. La tendenza, sia per questioni di costo che di affidabilità, è comunque nel digitalizzare tutti i controlli.

- **il software** comportamentale definisce il compito e le relazioni del robot con l'ambiente circostante. Diverse famiglie di software comportamentali sono disponibili e si differenziano prevalentemente in base al campo di applicazione. Alcuni esempi includono: inseguimento di traiettorie preimpostate (automazione industriale) ovvero fornite in tempo reale (teleguida, teleoperazione), rilevazione di parametri (controllo qualità, ispezione), analisi semantica dei dati rilevati

(sorveglianza, navigazione autonoma), inseguimento di fattori di merito (ausili tecnologici, extender). Inoltre il software comportamentale include un'ampia famiglia di soluzioni analitiche/numeriche per risolvere specifici problemi di movimento: cinematica diretta, cinematica inversa, pianificazione di traiettorie.

1.4 Robotica

La robotica è una scienza che studia i comportamenti degli esseri intelligenti e cerca di sviluppare delle metodologie che permettano ad una macchina (robot), dotata di opportuni dispositivi (sensori e attuatori) atti a percepire l'ambiente circostante e ad interagire con esso, di eseguire dei compiti specifici. È una disciplina relativamente nuova, che affonda le sue radici nell'antico desiderio dell'uomo di costruire strumenti che possano liberarlo da compiti troppo faticosi, noiosi o pericolosi. La scienza robotica trova applicazioni in molteplici contesti. Questo ha fatto sì che nascessero varie sotto-discipline fra le quali però raramente esiste una netta linea di demarcazione.

1.4.1 Le leggi della robotica

Le leggi della robotica sono un insieme di principi scritti da Isaac Asimov[12], ai quali sono soggetti molti robot dei suoi racconti. Ai tre enunciati iniziali[13] è stata aggiunto in seguito un quarto[14], la legge zero[15]. Insieme regolano il comportamento dei robot, al fine di salvaguardare l'incolumità dell'umanità. Le leggi sono le seguenti (le parti in corsivo indicano le modifiche in seguito all'aggiunta della legge zero):

- 0. Un robot non può recare danno all'umanità, né può permettere che, a causa del proprio mancato intervento, l'umanità riceva danno.

- 1. Un robot non può recar danno a un essere umano né può permettere che, a causa del proprio mancato intervento, un essere umano riceva danno. *Purché questo non contrasti con la Legge Zero.*

[12] Isaac Asimov (Petroviči, 2 gennaio 1920 – New York, 6 aprile 1992) è stato un biochimico e scrittore statunitense di origine russa.

[13] le tre leggi iniziali sono state introdotte per la prima volta insieme nel racconto *Circolo vizioso* [3]

[14] la quarta legge è stata introdotta nel libro *I robot e l'Impero* [4]

[15] una legge con numero più basso ha priorità rispetto a una con numero maggiore

- 2. Un robot deve obbedire agli ordini impartiti dagli esseri umani, purché tali ordini non contravvengano *alla Legge Zero e* alla Prima Legge.

- 3. Un robot deve proteggere la propria esistenza, purché questa autodifesa non contrasti con *la Legge Zero,* la Prima Legge e la Seconda Legge.

1.4.2 Applicazioni della robotica

La robotica prende piede nella società negli anni '70 come supporto alla produzione industriale. In quel periodo l'ambiente in cui operava il robot e l'ambiente in cui operava l'uomo erano completamente separati, al fine di garantire i margini di sicurezza agli operatori necessari. I primi robot, inoltre, operavano in ambienti completamente strutturati, ovvero dove le posizioni di tutti gli elementi con cui il robot doveva interagire erano note a priori. Successivi sviluppi della robotica, come la visione artificiale, hanno poi consentito di ridurre i vincoli imposti nell'ambiente. Successivamente la robustezza e i fattori di sicurezza per gli operatori collegati all'impiego dei robot hanno indotto i ricercatori a progettare nuovi sistemi, detti di teleoperazione, in cui i robot siano capaci di trasportare in ambienti remoti e/o ostili le capacità di azione e destrezza di un operatore umano. L'avvento della teleoperazione ha introdotto nel paradigma di azione dei robot una profonda trasformazione, dato che l'operatore doveva operare nello stesso spazio fisico del robot con cui interagiva.

Negli anni '90 gli ambienti virtuali hanno ulteriormente ridotto questa distanza tra operatore e robot, immaginando e realizzando una serie di dispositivi robotici, detti *interfacce afferenti*, tramite i quali l'operatore risultava in grado di interagire con l'ambiente virtuale e percepirne stimoli fisici. L'uso di questi dispositivi (aptici, tattili, termici) richiedeva che l'operatore indossasse fisicamente una componente, o tutto il meccanismo, in grado di trasmettere percezioni tramite aree di contatto. Gli ambienti virtuali arricchiscono quindi la capacità di condivisione dell'ambiente tra sistema robotico e uomo, con la capacità di condivisione delle esperienze.

Un ulteriore sviluppo di questi sistemi viene dall'intelligenza artificiale. Ad oggi, infatti, un'ampia serie di sistemi robotici consentono di avere in comune con l'operatore non solo l'interazione in termini di esperienza, ma anche in termini di intenzione. È possibile suddividere le aree di ricerca ed applicazione della robotica in quattro grandi categorie:

- **robot industriali**: i primi in ordine di applicazione, hanno oggi un vastissimo impiego in tutti i settori della produzione. Essi infatti consentono di ridurre costi e tempi di produzione e di poter modulare i costi in base ai volumi di produzione necessari. In aggiunta, miglioramenti qualitativi dei prodotti e la maggior sicurezza per il personale danno forti motivazioni a questo tipo di produzione. Nel campo della robotica industriale esistono manipolatori per decine di applicazioni differenti: *verniciatura, assemblaggio, pulizia, ispezione, saldatura, montaggio, movimentazione, taglio, controllo qualità, rilevamento, ecc.* Il numero di applicazioni sembra destinato ad aumentare con il tempo e l'autonomia raggiunta da questi dispositivi.

- **robot sociali**: individuano una nuova applicazione della robotica destinata ad essere nel futuro uno strumento di interazione sociale. Eliminata la barriera della sicurezza, tramite una serie accurata di norme e di certificazioni, i robot possono entrare a far parte della vita sociale. Rispetto ai computer questi ultimi sembrano avere maggiori potenzialità espressive nel fatto che possono integrare le capacità multimodali con movimento e gestualità ai primi negati. Ad oggi lo sviluppo dei robot sociali è ancora limitato.

- **robot ludici e domestici**: sono ormai diffusi i robot per il gioco come il Sony Aibò, Mitsubishi Wakamaru, i LEGO Mindstorms, i robot per il cinema (l'animatronica[16]) e per l'assistenza agli anziani (domotica[17]).

- **robot orientati all'interazione artistica, lo sport e la musica**: sono robot di più recente ricerca.

[16]per *animatronica* si intende l'utilizzo di componenti elettronici e robotici per dare autonomia di movimento a soggetti, specialmente pupazzi meccanici.

[17]la *domotica* è la scienza interdisciplinare che si occupa dello studio delle tecnologie atte a migliorare la qualità della vita nella casa e più in generale negli ambienti antropizzati. Il termine domotica deriva dal latino *domus* che significa 'casa'.

Capitolo 2

Tecnologia LEGO Mindstorms

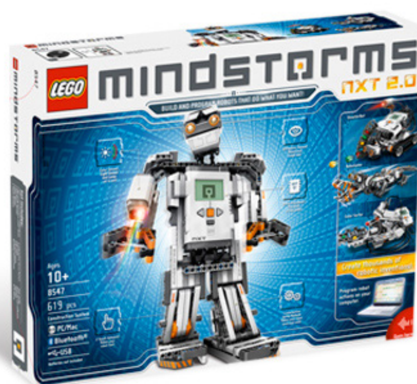

Figura 2.1: Il pacchetto LEGO Mindstorms

2.1 Panoramica

Il progetto Mindstorms è nato alla fine degli anni '80 al MIT[1] di Boston nell'ambito di una ricerca di orientamento costruttiva per avviare i più giovani al computer, integrando l'informatica a prodotti ad essi dedicati. Partito da questa idea, le potenzialità di questo prodotto hanno permesso che i Mindstorms fossero sfruttati anche in ambito di ricerca ed in settori molto più

[1] Massachusetts Institute of Technology

professionali, tanto che sarebbe sbagliato considerarli dei semplici giocattoli. Infatti LEGO Mindstorms può essere usato per costruire un modello di sistema integrato con parti elettromeccaniche controllate da computer e praticamente tutti i tipi di sistemi integrati elettromeccanici esistenti nella vita reale (come gli elevatori o i robots industriali) possono essere modellati con i Mindstorms.

LEGO Mindstorms combina mattoncini programmabili con motori elettrici, sensori, mattoncini LEGO, pezzi di LEGO Technic (come ingranaggi, assi e parti pneumatiche) per costruire robot e altri sistemi automatici e/o interattivi.

La prima generazione di LEGO Mindstorms era costruita intorno ad un mattone programmabile conosciuto come RCX. Esso conteneva un microcontroller Renesas H8/300 come CPU interna e veniva programmato scaricando il programma (che poteva essere scritto in vari linguaggi di programmazione) da un PC o da un Macintosh sulla sua RAM attraverso una speciale interfaccia ad infrarossi. La seconda generazione di LEGO Mindstorms è il Mindstorms NXT, che è stato rilasciato ad agosto 2006. Il kit comprende tre servomotori (molto più grandi di quelli contenuti nella precedente confezione), un sensore tattile, un sensore luminoso, un nuovo sensore sonoro, un sensore di prossimità (a ultrasuoni) e un nuovo mattoncino intelligente NXT. Oltre alle migliorie apportate ai componenti del kit, il mattoncino intelligente NXT è stato dotato di porta Bluetooth la quale può essere utilizzata non solo per caricare eventuali applicazioni sul robot, ma anche per comandare la macchina da remoto.

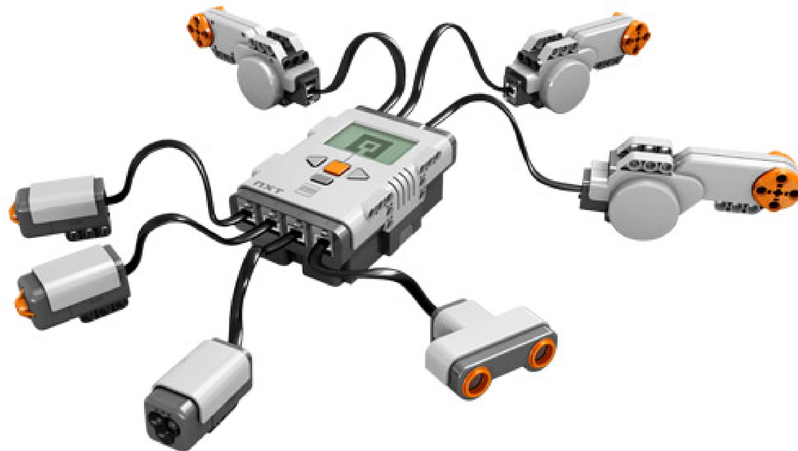

Figura 2.2: Panoramica NXT con sensori e motori connessi

2.2 Il brick NXT

Figura 2.3: Il brick NXT

Il brick NXT (figura 2.3) è il cervello dei robot Mindstorms. È un mattoncino computerizzato ed intelligente che rende possibile l'esecuzione di tutti i compiti che un robot deve eseguire. Le caratteristiche tecniche sono le seguenti:

- **microcontrollore**: a 32-bit ARM7
- **coprocessore** 8 bit Atmel ATmega48 (classe AVR: è un RISC a 8 bit) a 8 MHz, con 4k flash e 512 byte RAM
- **memoria flash** 256KB, 64KB RAM
- **interfaccia Bluetooth**: v2.0+EDR (chipset CSR BlueCore 4 version 2, clock a 26 MHz, con propri buffer RAM e firmware stack Bluelab 3.2) velocità teorica massima 0,46 Mbit/sec (per trasferire il software o per controllare il robot da remoto)
- **porta USB**: v1.1 a piena velocità (12 Mbit/s)
- **display LCD**: bianco e nero da 100x64 pixel
- **speaker**: mono 8 bit fino a 16 KHz
- **porte di input**: 4 riservate ai sensori
- **porte di output**: 3 riservate ai motori
- **alimentazione**: con 6 batterie AA (1.5V) oppure tramite la batteria ricaricabile al litio della casa.

2.3 I sensori

I vari tipi di sensori che LEGO Mindstorms supporta:

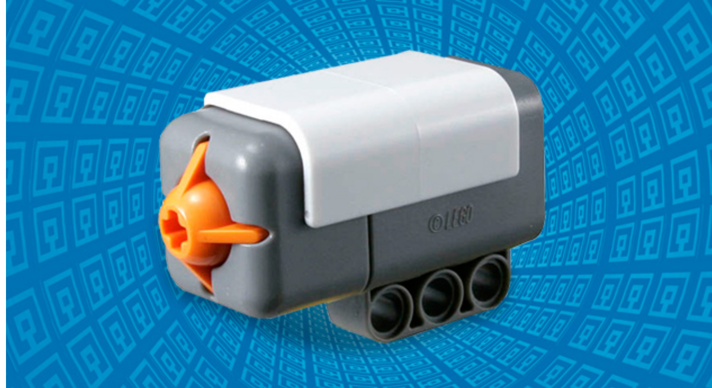

Figura 2.4: Il sensore tattile

- **sensore tattile** (figura 2.4): fornisce al robot il senso del tatto, rilevando la pressione del pulsante posto all'esterno. Può essere utilizzato per sapere se il robot ha qualche oggetto fra le 'mani' da catturare.

Figura 2.5: Il sensore sonoro

- **sensore sonoro** (figura 2.5): è capace di catturare i decibel dei rumori nell'ambiente circostante. Può misurare livelli di intensità sonora fino a 90 dB circa. Per rendere più semplice la lettura, tali misurazioni

vengono espresse in percentuale, ad esempio: 4-5% rumore all'interno di un soggiorno silenzioso, 5-10% suono di conversazione nelle vicinanze, 10-30% suono di conversazione vicino al microfono o musica a normale livello, 30-100% persone che gridano o musica ad alto livello.

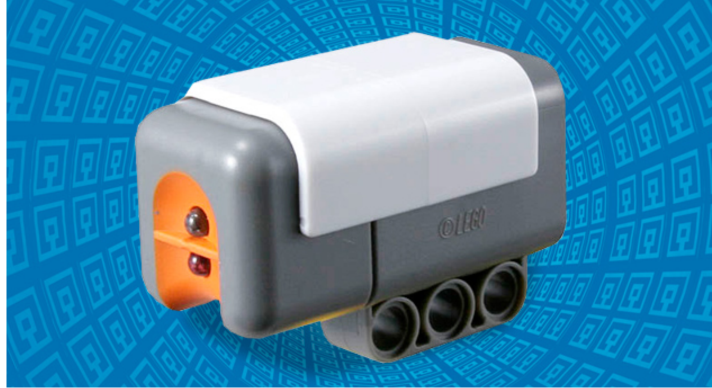

Figura 2.6: Il sensore luminoso

- **sensore luminoso** (figura 2.6): è uno dei due sensori che danno la visione al robot (l'altro è il sensore ad ultrasuoni). Esso percepisce l'intensità luminosa ambientale o di un oggetto (vede il colore come una scala di grigi, il risultato è una tonalità che va dal chiaro allo scuro). È dotato anche di una luce led rossa che, riflettendo sulla superficie, ne fa assimilare la reale luminosità.

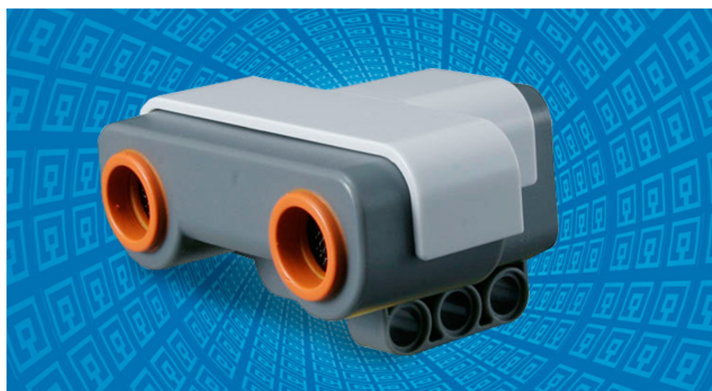

Figura 2.7: Il sensore ad ultrasuoni

- **sensore ad ultrasuoni** (figura 2.7): è forse il sensore più usato nella costruzione di robot. Grazie ad esso il robot è in grado di vedere e scoprire oggetti, lo si può usare per evitare ostacoli, misurare distanze o rilevare dei movimenti. Le distanze sono espresse in centimetri o in pollici, e vanno da un massimo di 255 a un minimo di 0 centimetri con una precisione di +/- 3 centimetri. Per il suo funzionamento utilizza lo stesso principio scientifico usato dai pipistrelli: misura la distanza calcolando il tempo impiegato da un'onda sonora a colpire un oggetto e ritornare indietro, come un eco. Il sensore è più sensibile a oggetti grandi con superfici dure, mentre lo è meno con oggetti realizzati in stoffa o curvati (come una palla) oppure molto sottili e piccoli.

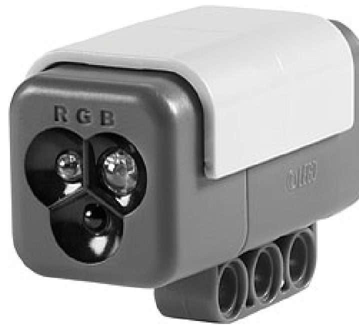

Figura 2.8: Il sensore di colore

- **sensore di colore** (figura 2.8): tipo di sensore più preciso, della serie HiTech, simile a quello di luce, che però riesce a riconoscere direttamente i colori invece di vedere solo gradazioni di grigio

2.4 I servomotori

Il servomotore (figura 2.9) è responsabile degli spostamenti, o in generale dei movimenti, della macchina. Ha al suo interno un sensore di rotazione per misurare sia la velocità che la distanza, valori che vengono trasmessi al processore dell'NXT. Vi è la possibilità di sincronizzare in maniera accurata i vari servomotori per poterli fare lavorare assieme.

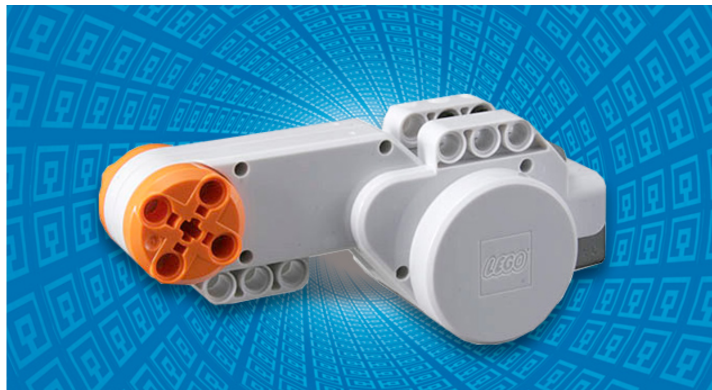

Figura 2.9: Il servomotore

Capitolo 3

ReSpecT

Il linguaggio ReSpecT[1] è un linguaggio logico per la coordinazione[2] nei sistemi multi-agente. ReSpecT è fondato su un modello di coordinazione che prevede *tuple centres*[3] come primaria astrazione per progettare e sviluppare un mezzo di coordinazione general-purpose. Il comportamento dei tuple centres è programmabile attraverso il linguaggio logico di primo livello ReSpecT.

3.1 Linguaggio di coordinazione

Un modello di coordinazione può essere visto come la colla che lega più attività assieme. Tale modello fornisce una struttura sulla quale entità attive ed indipendenti, chiamate agenti, possono svolgere i propri compiti. Le questioni di cui un modello di coordinazione dovrebbe occuparsi sono la creazione, la distruzione e la collocazione spaziale degli agenti, nonché la distribuzione e la sincronizzazione delle loro azioni. Ma quali sono i componenti di un sistema di coordinazione?

- **Entità**, le cui interazioni sono gestite e controllate dal modello. Sono anche chiamate *entità coordinabili*, possono essere processi unix-like, threads, oggetti concorrenti o anche utenti.

[1] Reaction Specification Tuples. Per una più dettagliata descrizione vedi [6], [5], [7]
[2] *linguaggio di coordinazione* - vedi sezione 3.1
[3] *tuple center* - vedi sezione 3.3

- **Media**, astrazione che regola e rende possibili le interazioni fra gli agenti. È il cuore attorno al quale i componenti del sistema vengono organizzati. Alcuni esempi sono i semaphors, i monitors o i tuple spaces.

- **Regole di coordinazione**, le quali definiscono il comportamento del media in risposta alle interazioni. Tali regole vengono utilizzate dagli agenti per capire come poter agire sul media di coordinazione e quali primitive poter utilizzare (es. in/out/rd...). Definiscono anche la struttura delle informazioni che verranno 'manipolate' all'interno del media e dai vari agenti (es. tuples, elementi XML, oggetti Java, ecc.).

Modelli di coordinazione di questo genere sono solitamente sviluppati attorno al concetto di spazio di dati condiviso. Tale spazio non è altro che un 'contenitore indirizzabile' di strutture dati. Se più processi autonomi stanno collaborando in una stessa attività, essi non possono comunicare fra loro in maniera diretta, ma solamente affidandosi a tale spazio condiviso. Tali modelli, detti anche modelli data-driven, sembrano essere migliori per applicazioni il cui numero di entità che devono collaborare non è conosciuto a priori .

3.2 Tuple Space

Il tuple space è uno spazio di calcolo globale, all'interno del quale vengono elaborate delle strutture dati chiamate tuple. Le tuple sono una collezione ordinata, potenzialmente eterogenea, di informazioni e rappresentano il linguaggio di comunicazione del nostro meta modello. Esse hanno vita esattamente all'interno del tuple space, sono ugualmente accessibili dagli agenti, ma non sono collegate a nessuno di esse. L'accesso ad una tupla avviene in maniera associativa (controllando il loro contenuto o struttura) piuttosto che per indirizzamento o naming. Il linguaggio di coordinazione, tramite il quale inserire e recuperare tuple dal tuple space, è **Linda**. Esso utilizza come linguaggio di comunicazione i seguenti elementi:

- **tuple**: es. persona(nome,Davide), done(yes)

- **template** (o **anti-tuple**): identificano un set, una classe di tuple; es. persona(nome, X), done(Y)

- un **meccanismo di tuple matching**: tramite esso si cerca di avere un match fra una tupla ed un template. Il match con una tupla avviene se: il numero dei campi è uguale, tipi, valori e lunghezza delle stringhe del template corrispondono a quelli della tupla, tipi e lunghezza dei campi formali corrispondono a quelli dei corrispondenti campi della tupla

Linda è stato concepito con un set di primitive veramente ristretto, ma nello stesso tempo completo:

- **in(TT)**: legge e toglie dal tuple space la tupla che fa match con il template TT. Se più di una tupla fa match con il template, ne viene scelta una in maniera non deterministica. Se non è presente alcuna tupla facente match con il template proposto, l'operazione viene sospesa fino a quando non viene trovata la tupla opportuna

- **out(T)**: inserisce nel tuple space la tupla T

- **rd(TT)**: legge dal tuple space, senza rimuoverla, la tupla che fa match con il template TT. Nel caso in cui non vengano trovate tuple facenti match con il template o, al contrario, ci siano più di una tupla corretta, il comportamento rimane identico a quello precedentemente descritto per la primitiva in(TT).

Caratteristica di Linda, oltre alla coordinazione basata sul matching informativo, è quella di agevolare la comunicazione e la collaborazione trasversale fra gli agenti. Due agenti, che per comunicare non hanno la necessità di essere nello stesso ambiente nel medesimo momento, costituiscono un esempio di disaccoppiamento temporale e spaziale.

3.3 Tuple Center

Nota la struttura tuple space precedentemente descritta, come risolvere il problema di eseguire un'attività dopo che, ad esempio, un certo numero di agenti ha fatto richiesta per una determinata risorsa? Più in generale, cosa serve per basare la coordinazione su eventi che vengono scatenati piuttosto che sul cambiamento o meno delle informazioni? È evidente che, con il modello precedentemente illustrato (tuple space), il quale basa il meccanismo di coordinazione unicamente sulla presenza o assenza di dati, non è possibile realizzare tutti i pattern di coordinazione control-driven (es. event-driven). Ciò che è stato fatto è aggiungere uno strato di controllo control-driven su

quello già esistente di Linda, facendo bene attenzione a non aggiungere ulteriori primitive. Viene così introdotto il concetto di tuple center (centro di tuple): tuple space unito ad un automa a stati finiti che effettua computazioni event-driven. Possiamo immaginare che ogni operazione interna al tuple centre (in entrata o in uscita) generi un evento che viene catturato dall'infrastruttura. L'automa a stati finiti interno al tuple-centre effettua le computazioni necessarie al fine di eseguire particolari azioni (che possono eventualmente modificare lo stato interno del tuple centre) in base al tipo di evento catturato. Insieme con l'introduzione del concetto di tuple centre nasce ReSpecT (Reaction Specification Tuples) come linguaggio di programmazione di tuple center. ReSpecT può essere visto come estensione di Linda e offre la possibilità di specificare reazioni: azioni da eseguire in base al verificarsi di particolari eventi definiti nella sintassi del linguaggio, che modificano lo stato del tuple centre. Le reazioni in ReSpecT sono del tipo *reaction(E,G,R)*, dove:

- **E** rappresenta il template con il quale l'evento che viene a verificarsi deve fare match per poter far scattare la reaction corrispondente
- **G** è la guardia (possono esserne specificate anche più di una) e rappresenta la condizione che deve essere verificata dopo che l'evento E è stato catturato
- **R** rappresenta la lista delle reazioni che devono essere eseguite, in termini di codice, dopo che la reazione è scattata

Ciò che mancava inizialmente a ReSpecT era la possibilità di specificare, in termini di reazioni, predicati di coordinamento relativi ad eventi ambientali riguardanti risorse esterne. Le numerose modifiche ed estensioni al linguaggio ReSpecT hanno portato verso la cosiddetta Situatedness.

Capitolo 4

Dalle API iCommand a LeJOS

Le due tesi precedenti[1] hanno cercato di sfruttare la tecnologia ReSpecT Situated per realizzare un sistema di collaborazione tra robot Mindstorm. Tale intento è stato portato a compimento con risultati soddisfacenti, sebbene non ottimali. Infatti l'approccio ivi proposto presentava intrinsecamente dei limiti dovuti alla tecnologia scelta per l'implementazione: le API iCommand. Questo package Java consente di controllare un robot NXT attraverso una connessione bluetooth; utilizza il firmware standard LEGO Mindstorms NXT per trasmettere comandi scritti in Java da computer al robot. Inizialmente[2] iCommand è stato rilasciato per permettere agli utenti di cominciare a programmare il NXT in Java mentre LeJOS NXJ era ancora in fase di sviluppo. Nelle note di rilascio della versione LeJOS NXJ 0.8[3] si legge: "*iCommand is dead! Long live iCommand. PC control is now included in pccomm.jar and the lejos.nxt.remote package*"[4] Ad oggi LeJOS NXJ è disponibile nella versione 0.9.1[5]. Il presente lavoro si prefigge di trasportare i lavori precedenti da iCommand a LeJOS NXJ indicando i vantaggi di questa 'migrazione'.

[1]per lavori precedenti ci si riferisce alle tesi di laurea [2] e [10]

[2]la prima versione di iCommand risale al 2006

[3]LeJOS NXJ v 0.8 - rilasciata il 22 maggio 2009

[4]letteralmente: *iCommand è morto! Lunga vita a iCommand. Il controllo da PC è incluso nel pccomm.jar e nel package lejos.nxt.remote*

[5]LeJOS NXJ v 0.9.1 - rilasciata il 6 febbraio 2012

4.1 Le API iCommand

Le API iCommand rappresentano un framework che permette di controllare il NXT utilizzando il linguaggio Java e una connessione Bluetooth. Utilizza il firmware standard del Lego NXT per ricevere comandi dal codice Java presente su un computer. Per questo motivo il flusso di controllo di un'applicazione basata interamente su iCommand è costantemente sotto la guida della parte in esecuzione sul computer che esegue i comandi da remoto. Dalla versione 0.8 di LeJOS NXJ, iCommand è stato assimilato nel *pccomm.jar* e nel package *lejos.nxt.remote*, che insieme consentono a LeJOS NXJ di attuare lo stesso flusso di controllo di iCommand.

4.2 LeJOS NXJ

LeJOS[6][8] è una piccola Java Virtual Machine. Nel 2006 è stata portata[7] sul LEGO NXT.

LeJOS NXJ[9] è un firmware che sostituisce quello standard LEGO del NXT. Esso include tutte le classi dell'API NXJ (per il NXT) e anche gli strumenti necessari a caricare il codice sul NXT (tool per PC).

LeJOS NXJ supporta:

- un linguaggio object oriented (Java)
- preemptive threads (tasks)
- arrays, anche multi-dimensionali
- recursion
- synchronization
- exceptions
- tipi di dato Java (including float, long, and String)
- quasi le intere classi java.lang, java.util and java.io
- una ben documentata Robotics API (utilizzata dalla parte su computer)

[6]pronuncia simile allo spagnolo *lejos, 'lontano'*

[7]in informatica, la portabilità di un componente software è un adattamento o una modifica del componente, volto a consentire il suo utilizzo in un ambiente di esecuzione diverso da quello originale

Segue la descrizione di alcune classi utilizzate nel presente lavoro.
Le API per pc includono:

- `lejos.pc.comm.NXTComm`
  interfaccia la cui implementazione permette una comunicazione di basso livello con il NXT. Alcuni metodi utilizzati:
  - `NXTInfo[] search(java.lang.String name)throws NXTCommException`
    cerca dispositivi NXT via USB, Bluetooth o entrambi. Il parametro name può essere null. Un metodo analogo include un secondo parametro che indica il tipo di ricerca (USB, Bluetooth)
  - `boolean open(NXTInfo nxt)throws NXTCommException`
    si connette a un NXT indicato dall'oggetto nxt restituito da una search() o creato con name e address
  - `java.io.OutputStream getOutputStream()`
    ritorna un OutputStream per la scrittura di uno stream di dati al NXT su questa connessione
  - `java.io.InputStream getInputStream()`
    ritorna un InputStream per la scrittura di uno stream di dati al NXT su questa connessione

Le API per NXT includono:

- `lejos.nxt.comm.Bluetooth`
  classe che permette comunicazioni Bluetooth mediante connessioni in entrata o in uscita. Alcuni metodi utilizzati:
  - `public static RemoteDevice getKnownDevice(String fName)`
    restituisce un device incluso nella lista interna dei BC4-Chip Device conosciuti (quelli che sono stati accoppiati in precedenza)
  - `public static BTConnection waitForConnection()`
    aspetta una connessione in ingresso. Utilizza il PIN di default (in generale 0000, per il NXT 1234);
- `lejos.nxt.comm.NXTConnection`
  classe per connessioni generiche LeJOS NXT. Fornisce accesso ai metodi standard read/write. Supporta operazioni di I/O in modalità asincrona (utilizzata per connessioni Bluetooth e RS-485) o sincrona (utilizzata per connessioni USB) per l'effettiva scrittura/lettura dei buffer di basso livello. Alcuni metodi utilizzati:

- `public DataInputStream openDataInputStream()`
  restituisce un DataInputStream per questa connessione
- `public DataOutputStream openDataOutputStream()`
  restituisce un DataOutputStream per questa connessione
- `public void close()`
  Chiude la connessione. Svuota il buffer (flush) di ogni output ancora da trasmettere. Informa il lato remoto che la connessione è chiusa e libera le risorse allocate.

- `public interface SensorPortListener`
  interfaccia per monitorare variazioni nel valore rilevato da un sensore (luminoso, tattile, ecc.) su una porta SensorPort.
  - `void stateChanged(SensorPort aSource,int aOldValue, int aNewValue)`
    metodo invocato quando varia il valore raw del sensore attaccato alla porta

- `lejos.nxt.LightSensor`
  questa classe viene utilizzata per leggere i valori di un sensore luminoso LEGO NXT. Il sensore può essere calibrato a valori grandi o piccoli
  - `public int getLightValue()`
    restituisce la luminosità calibrata e normalizzata della luce rilevata. Il valore è compreso tra 0 e 100%, con 0=buio e 100=luce intensa del sole
  - `public int getNormalizedLightValue()`
    restituisce la luminosità normalizzata raw della luce rilevata. Il valore è compreso tra 0 e 1023, con 0=buio e 1023=luce del sole. Tipicamente i sensori NXT rilevano valori nell'intervallo 145 - 890.

- `lejos.nxt.UltrasonicSensor`
  astrazione di un sensore ultrasonico NXT. Il sensore può funzionare in quatto modalità: off (spento), continuous (verifica periodicamente se un oggetto è rilevato), ping (manda un singolo ping e rileva fino a 8 oggetti) and capture (aspetta segnali da altri device ultrasonici).
  - `public int getDistance()`
    rileva la distanza fino a un oggetto o 255 se non viene rilevato nessun oggetto o avviene un errore. Per assicurarsi che i dati

rilevati siano validi, questo metodo potrebbe dover aspettare un piccolo intervallo di tempo finché i dati relativi alla distanza sono disponibili

- `lejos.nxt.TouchSensor`
  astrazione di un sensore tattile NXT. Funziona anche con i sensori tattili del RCX.
  - `public boolean isPressed()`
    verifica se il sensore è premuto
- `lejos.nxt.SoundSensor`
  astrazione di un sensore sonoro per il NXT
  - `public int readValue()`
    legge il valore corrente del sensore
- `lejos.nxt.LCD`
  scrive testo e grafici sul display LCD
  - `public static void clear(int y)`
    pulisce una riga del display LCD
  - `public static void drawInt(int i, int x, int y)`
    mostra sul display LCD un intero alle coordinate x,y
  - `public static void drawString(String str, int x, int y)`
    mostra sul display LCD una stringa alle coordinate x,y
  - `public static void refresh()`
    aggiorna il display LCD. Deve essere invocato ogni volta in seguito a una drawInt(), drawString(), ecc.
- `lejos.nxt.Button`
  astrazione di un pulsante del NXT
  - `public final void waitForPressAndRelease()`
    aspetta finché il pulsante è stato rilasciato
- `lejos.nxt.Motor`
  astrazione di un motore del NXT di cui ne permette il controllo

Capitolo 5

Case studies con LeJOS

5.1 Framework

Per implementare i case studies con LeJOS si è sviluppato un framework specifico. La sua architettura logica contiene i seguenti componenti: un **server** in esecuzione su PC il cui compito è gestire i vari client NXT che si vogliono connettere al sistema, vari **thread** che gestiscono le specifiche applicazioni a cui i robot partecipano (coordinando il tutto con ReSpecT) e vari **client** NXT che eseguono moduli specifici. La tecnologia utilizzata per la comunicazione tra server e client è Bluetooth.

Più in dettaglio il ruolo dei componenti del sistema è il seguente:

- Il **server** introdotto rappresenta una specie di middleware. Esso accetta le richieste di connessione dei client e li associa come componenti al contesto di esecuzione di un'applicazione (scenario). L'implementazione concreta del loop del server è la seguente:

```
37   // server loop
38   while (clients < 2) {// true se si prevede un numero
         illimitato di client
39     // inquire for connections
40     devicesInfo = nxtComm.search(null, NXTCommFactory.
           BLUETOOTH);
41     log("NXTComm bluetooth devices search executed.");
42
43     for (int i = 0; i < devicesInfo.length; i++) {
44       if (isAvailable(devicesInfo[i])) {
45         // connect
46         log("Found new available device named \"" +
               devicesInfo[i].name + "\". Connecting.");
```

```
47            nxtComm.open(devicesInfo[i]);
48            dos = new DataOutputStream(nxtComm.
                 getOutputStream());
49            dis = new DataInputStream(nxtComm.
                 getInputStream());
50
51            // get program name
52            log("Connected to \"" + devicesInfo[i].name +
                 "\". Getting program name.");
53            String progName = BT.receiveStringValue(dis,
                 logWindow);
54
55            if (progName.equals("CTW")) {
56              int pos = programPos("CTW");
57              if (pos >= 0) {
58                // add this client to program manager
59                programs.get(pos).addClient(nxtComm, dis,
                     dos);
60              } else {
61                // create new thread to manage the program
62                log("Have to start new thread.");
63                CTWpc p = new CTWpc(nxtComm, dis, dos,
                     logWindow);
64                programs.add(p);
65                p.run();
66              }
67            } else {
68              // here are the other case studies...
69            }
70          } else {
71            log("Device ignored \"" + devicesInfo[i].name
                 + "\".");
72          }
73        }
74
75        // server waits a little time before inquiring
             bluetooth again
76        Thread.sleep(1000);
77      }// loop END
```

Ciascuna applicazione viene gestita sul PC da un thread dedicato che estende una classe base astratta LejosProgram. Questa classe incapsula le informazioni generiche (nome programma, costanti, ecc.) e le funzionalità comuni a tutte le applicazioni (inizializzazione di ReSpecT, ecc.). Di seguito si riporta parte della sua implementazione:

```
11  public abstract class LejosProgram implements Runnable {
12
13    protected String progName;
14    protected int max_clients;
15    protected int now_clients;
16    protected boolean inhibit;
```

```
17    protected NXTmonitor logger;
18
19    protected NXTComm[] nxtComm;
20    protected DataInputStream[] dis;
21    protected DataOutputStream[] dos;
22
23    protected String tcIdName;
24    protected TupleCentreId tcId;
25    protected IManagementContext tc_mc;
26
27    public LejosProgram(String progName, NXTComm nxtComm,
          DataInputStream dis,
28        DataOutputStream dos, NXTmonitor logger, int
            max_clients) {
29      this.progName = progName;
30      this.max_clients = max_clients;
31      this.now_clients = 1;
32      this.inhibit = false;
33      this.logger = logger;
34
35      this.nxtComm = new NXTComm[max_clients];
36      this.dis = new DataInputStream[max_clients];
37      this.dos = new DataOutputStream[max_clients];
38
39      this.nxtComm[0] = nxtComm;
40      this.dis[0] = dis;
41      this.dos[0] = dos;
42
43      // start ReSpecT engine and resources
44      try {
45        tcIdName = this.progName+"@localhost";
46        log("tcIdName=" + tcIdName);
47        tcId = new TupleCentreId(tcIdName);
48        RespectTCContainer.createRespectTC(tcId, 100);
49        tc_mc = RespectTCContainer.getManagementContext(
            tcId);
50      } catch (Exception e) {
51        e.printStackTrace();
52      }
53    }
54
55    public abstract void execute() throws Exception;
56
57    public void stop() {
58      this.inhibit = true;
59      System.exit(0);
60    }
61
62    public void addClient(NXTComm nxtComm, DataInputStream
           dis,
63        DataOutputStream dos) {
64      this.nxtComm[now_clients] = nxtComm;
65      this.dis[now_clients] = dis;
66      this.dos[now_clients] = dos;
67      now_clients++;
68    }
```

```
69
70   public String getName() {
71     return this.progName;
72   }
73
74   public void run() {
75     try {
76       execute();
77     } catch (Exception e) {
78       e.printStackTrace();
79     }
80   }
81
82   public void log(String text) {
83     logger.log("[" + this.progName + "] " + text);
84   }
85 }
```

- Sui **client** NXT che si connettono al sistema è in esecuzione un *piano*, un insieme di azioni da eseguire. A ciascun scenario analizzato corrisponde un piano programmato per il NXT. Le istruzioni sono scritte nel linguaggio LeJOS NXJ e vengono prima compilati e in seguito caricati sul NXT. Le classi rappresentative dei specifici scenari estendono una classe base BrickProgram, che incapsula le informazioni comuni come nome del programma, identificativo del server, orientamento iniziale, etc. Le singole specializzazioni inizializzano i sensori necessari e implementano il flusso logico per l'esecuzione dell'applicazione relativa. Di seguito si riporta parte dell'implementazione della classe BrickProgram:

```
14 public abstract class BrickProgram {
15
16   protected final String masterName = "RAINFOXxp";
17   protected final String pName;
18   protected DataInputStream dis;
19   protected DataOutputStream dos;
20   protected Direction orientation;
21   protected DifferentialPilot pilot;
22   protected LightSensor lightSensor;
23   protected UltrasonicSensor sonicSensor;
24   protected SoundSensor soundSensor;
25
26   protected SensorPort lightPort = SensorPort.S1;
27   protected SensorPort sonicPort = SensorPort.S2;
28   protected SensorPort soundPort = SensorPort.S3;
29
30   protected int WHITE_LIGHT = 890;
31   protected int BLACK_LIGHT = 145;
32   public enum Direction{LEFT,UP,RIGHT,DOWN};
```

```
33
34	public BrickProgram(String pName){
35	  this.pName = pName;
36	}
37
38	public abstract void run() throws Exception;
39
40	public void initLightSensor(BrickProgram brick){
41	  lightSensor = new LightSensor(lightPort);
42	  lightPort.addSensorPortListener(new
	      LightPortListener(brick, WHITE_LIGHT,
	      BLACK_LIGHT));
43	}
44
45	public void initUltrasonicSensor(BrickProgram brick){
46	  sonicSensor = new UltrasonicSensor(sonicPort);
47	  sonicPort.addSensorPortListener(new
	      SonicPortListener(brick));
48	}
49
50	public void initSoundSensor(BrickProgram brick){
51	  soundSensor = new SoundSensor(soundPort);
52	  soundPort.addSensorPortListener(new
	      SoundPortListener(brick));
53	}
54
55	public void initPilot(Direction initialDirection){
56	  //pilot = new DifferentialPilot(2.25f, 5.5f, Motor.A
	      , Motor.C);
57	  pilot = new DifferentialPilot(MoveController.
	      WHEEL_SIZE_NXT2, 5.5f, Motor.A, Motor.C);
58	  setOrientation(initialDirection);
59	}
60	//...
```

È necessario specificare che per ciascuna applicazione in parte viene utilizzato un *protocollo di comunicazione a stati* implementato ad-hoc. Questo è dovuto anche al fatto che LeJOS supporta l'invio attraverso Bluetooth di soli tipi java base: int, string, ecc. Un esempio di funzionamento del protocollo è il seguente. Nello stato iniziale il client NXT deve inviare al server una stringa che rappresenta il nome dell'applicazione a cui intende partecipare. Il server in questo stato iniziale si aspetta di ricevere questa stringa e non fa procedere l'esecuzione dell'applicazione del robot finché non riceve tale stringa. In un altro stato in cui ad esempio il client NXT si aspetta di ricevere un segnale prima di poter procedere e riceve dal server un intero, il NXT sa già cosa rappresenta quel intero in base allo stato in cui si trova l'applicazione a cui partecipa.

- La coordinazione viene garantita da ReSpecT. L'engine ReSpecT viene

affiancato al server e ciascuna applicazione avrà il suo tuple center dedicato. Una applicazione inizia con la connessione dei vari client. Il thread che gestisce l'applicazione crea il tuple center e in base allo stato dell'applicazione scrive (oppure legge) tuple dal tuple center, scatenando eventuali reazioni. I vari client comunicano con il thread che gestisce l'applicazione, il quale in base al segnale ricevuto modifica il centro di tuple dedicato.

- Inoltre, il framework dispone anche di una libreria di funzioni utili a snellire lo sviluppo di futuri programmi. Contiene wrapper per I/O LeJOS , segnalazione errori, logging e anche alcune funzioni per la comunicazione Bluetooth.

5.2 Case study: Free casual roaming

Questo scenario prevede un robot in *free casual roaming*: un robot che deve navigare casualmente in un'area evitando gli ostacoli, integrando quindi due componenti principali: *movimento* e *collision avoidance*.

5.2.1 Algoritmo

Data l'autonomia del robot in questione, risulta che una iniziale soluzione semplificata LeJOS a questo primo case study non necessita di coordinazione ReSpecT. Si sceglie di utilizzare il package `Behavior` di LeJOS NXJ, API che consente di specificare in termini di comportamento il funzionamento del robot.

Il modello di controllo orientato al comportamento permette l'incapsulamento di ciascun *behavior*[1] in una struttura facilmente comprensibile. Questo approccio garantisce una più facile manutenzione del codice, la sua riutilizzabilità, la separazione di compiti indipendenti in moduli.

I concetti fondamentali del *behavior programming* implementato in LeJOS NXJ sono molto semplici:

- solo un behavior può essere attivo e avere il controllo del robot in un determinato istante di tempo
- ciascun behavior ha una priorità predefinita

[1] *behavior*, letteralmente *comportamento*, indica un insieme di azioni orientate al raggiungimento di uno scopo prestabilito, come ad esempio: muoversi, evitare gli ostacoli, monitorare il proprio stato, ecc.

- ciascun behavior è capace di stabilire se è il momento di assumere il controllo
- il behavior attivo in un determinato istante di tempo è quello con priorità maggiore fra tutti i behavior che potrebbero assumere il controllo

L'API Behavior è composta di una sola interfaccia a di una sola classe. L'interfaccia `Behavior` definisce tre metodi publici che ne riassumono il funzionamento. Ciascun compito che il robot deve eseguire può essere definito in una classe individuale di tipo Behavior. Una volta che sono stati creati, i vari behavior sono passati ad un `Arbitrator` che regola la loro esecuzione.

La struttura dell'API è la seguente:

- `lejos.subsumption.Behavior`
  - `boolean takeControl()`
    Ritorna un valore booleano che indica se questo behavior può diventare attivo.
  - `void action()`
    Viene eseguito quando un behavior è attivo. Di conseguenza un behavior è attivo per tutto il tempo che questo metodo è in esecuzione. Il metodo finisce l'esecuzione quando il compito ad esso assegnato finisce o quando viene invocato il metodo suppress(), lasciando comunque il robot in uno stato *safe* per il prossimo behavior.
  - `void suppress()`
    Termina immediatamente l'esecuzione del codice del metodo action()
- `lejos.subsumption.Arbitrator`
  - `public Arbitrator(Behavior[] behaviors, boolean returnWhenInactive)`
    Costruttore di un Arbitrator.
    behaviors: la priorità di un behavior è il suo indice nell'array
    returnWhenInactive: se true, il programma esce quando non c'è nessun behavior che vuole assumere il controllo. Altrimenti il programma rimane in esecuzione finché viene fermato attraverso la pressione dei pulsanti Enter/Escape del NXT
  - `public void start()`
    Fa partire il sistema di arbitraggio.

L'utilizzo della classe Arbitrator è di facile comprensione. Quando un oggetto Arbitrator viene creato, gli viene passato un array di oggetti Behavior. Successivamente viene invocato il suo metodo start() che fa partire il suo processo di arbitraggio, decidendo quale behavior diventerà attivo. Infatti, l'Arbitrator invoca il metodo takeControl() di ciascun oggetto Behavior, a cominciare dall'oggetto con il più grande indice nell'array e scalando in ordine decrescente fino a trovare un behavior che vuole assumere il controllo. Se la priorità di quest'ultimo è maggiore di quella del behavior correntemente attivo, allora il behavior attivo viene soppresso e viene invocato il metodo action() del nuovo behavior. Come risultato si ha che tra eventuali behavior che vogliono il controllo, solo quello con priorità maggiore diventerà attivo.

Fatte queste premesse, l'implementazione risulta di facile attuazione. Per prima cosa si implementano i due behavior relativi a *movimento* e *collision avoidance*.

```
6  public class Movement implements Behavior {
7    private boolean suppressed = false;
8
9    public boolean takeControl() {
10     return true;
11   }
12
13   public void suppress() {
14     suppressed = true;
15   }
16
17   public void action() {
18     suppressed = false;
19
20     //muove il robot avanti
21     Motor.A.forward();
22     Motor.C.forward();
23
24     //avanza finchè non viene sopresso
25     //(ossia finchè non viene rilevato un ostacolo)
26     while (!suppressed)
27       Thread.yield();
28
29     //arresta il robot
30     Motor.A.stop();
31     Motor.C.stop();
32   }
33 }
```

La classe Movement rappresenta un behavior che ha il compito di muovere il robot in avanti finché l'esecuzione di questo behavior viene fermata. Tale evento avviene nel caso della rilevazione di un ostacolo, compito assegnato alla classe CollisionAvoidance di seguito.

```
 6 public class CollisionAvoidance implements Behavior {
 7   private TouchSensor touch;
 8   private UltrasonicSensor sonar;
 9   private boolean suppressed = false;
10
11   public CollisionAvoidance(UltrasonicSensor sonar,
       TouchSensor touch) {
12     this.sonar = sonar;
13     this.touch = touch;
14   }
15
16   public boolean takeControl() {
17     return touch.isPressed() || sonar.getDistance() < 25;
18   }
19
20   public void suppress() {
21     suppressed = true;
22   }
23
24   public void action() {
25     suppressed = false;
26
27     //gira il robot a 180°
28     Motor.A.rotate(-180, true);
29     Motor.C.rotate(-360, true);
30
31     //aspetta di finire la rotazione o essere sopresso
32     while (Motor.C.isMoving() && !suppressed)
33       Thread.yield();
34
35     //arresta il robot
36     Motor.A.stop();
37     Motor.C.stop();
38   }
39 }
```

Finalmente la classe principale RoamingRobot che raggruppa il tutto.

```
 9 public class RoamingRobot {
10   public static void main(String[] args) {
11     //allocazione dei sensori di tatto e ultrasonico
12     UltrasonicSensor sonar = new UltrasonicSensor(SensorPort.
         S1);
13     TouchSensor touch = new TouchSensor(SensorPort.S2);
14
15     //creazione dei behavior e inserimento in un array
16     Behavior b1 = new Movement();
17     Behavior b2 = new CollisionAvoidance(sonar, touch);
18     Behavior[] bArray = { b1, b2 };
19
20     //creazione del Arbitrator ed esecuzione
21     Arbitrator arby = new Arbitrator(bArray);
22     arby.start();
```

```
23     }
24 }
```

5.2.2 Risultati

Per quanto riguarda questo case study, per la natura stessa dello scenario, l'approccio più conveniente (facendo uso di LeJOS) è quello di sviluppare un firmware autonomo sul NXT. La programmazione orientata al comportamento è principalmente utilizzata per robot autonomi che lavorano in modo indipendente. Ovviamente, se si dovessero coordinare tra di loro più robot in free casual roaming, questa soluzione diventa inefficiente, in quanto non permette la comunicazione diretta tra device diversi, ma garantisce solamente il funzionamento del robot come singola entità nell'ambiente, non inserito in un sistema più esteso. Infatti, per poter implementare la coordinazione bisognerebbe sviluppare altri behavior. Sapendo che i vari behavior sono ordinati in base alla precedenza, chi potrebbe garantire che un robot abbia precedenza su un altro?

Quindi la soluzione migliore per questo case study specifico implica l'utilizzo delle API Behavior. Tale scelta è giustificata dalla ricerca della soluzione più semplice, più potente e specifica possibile, anche se è leggermente più dispendiosa in termini di tempo. Altrettanto importante è lo sviluppo di codice facilmente mantenibile e riutilizzabile, obiettivo più facile da raggiungere vista la modularità dei singoli behavior che possono essere aggiunti e rimossi arbitrariamente in quanto essenzialmente indipendenti.

D'altra parte, se dovessimo coordinare più robot in free roaming, lo scenario cambia e la soluzione migliore implica l'utilizzo di ReSpecT, in un'applicazione che integra un piano in esecuzione sul NXT con un modulo eseguito su PC e dotato delle funzionalità garantite da ReSpecT.

5.3 Case study: Come this way

Questo case study prevede l'utilizzo di due robot: il primo (*leader*) segue un percorso su una griglia da un punto A ad un punto B e il secondo (*follower*) lo insegue passo-passo.

5.3.1 Algoritmo

La soluzione proposta per questo scenario utilizza il framework[2] sviluppato. Il server accetta la richiesta del primo NXT, chiede l'applicazione di cui fa parte e fa partire un nuovo thread che gestisca il rispettivo scenario. Quando arriva il secondo NXT , il server accetta la richiesta e, dopo aver chiesto l'applicazione di cui fa parte, lo associa al thread già in esecuzione per questo scenario. Questo preambolo è comune a tutti i scenari analizzati.

Il thread su PC che gestisce questo case study è implementato nella classe CTWpc. La fase che gestisce l'avvio dell'applicazione è implementata nel modo seguente:

```
30  public void execute() throws Exception {
31    log("CTW manager: getting first role...");
32    // get first role
33    roles[0] = BT.receiveStringValue(dis[0], logger);
34    log("First client connected: " + roles[0]);
35
36    // send wait
37    BT.sendInt(dos[0], 2, logger);
38    log("wait sent.");
39
40    // update TC
41    sendNewClientTuple(roles[0]);
42
43    log("Waiting for second client.");
44    while (now_clients < 2 && !inhibit) {
45      Thread.sleep(100);
46    }
47
48    // get second role
49    roles[1] = BT.receiveStringValue(dis[1], logger);
50    log("Second client connected: " + roles[1]);
51    sendNewClientTuple(roles[1]);
52
53    sem.wait();
54    if (!inhibit) {
55      for (int i = 0; i < roles.length; i++) {
56        if (roles[i].equals("leader"))
57          leaderId = i;
58        else if (roles[i].equals("follower"))
59          followerId = i;
60        else
61          log("Error @ roles");
62      }
63      // if valid send start to both clients
64      if (leaderId != -1 && followerId != -1) {
65        BT.sendInt(dos[0], 1, logger);
66        BT.sendInt(dos[1], 1, logger);
```

[2] vedi sezione 5.1

```
67        } else {
68          inhibit = true;
69          log("Cannot proceed. No leader, or no follower
              available. Stopping.");
70        }
71      }
72      int ready = 0;
73      ready = BT.receiveInt(dis[followerId], logger);
74      if (ready == 1) {
75        sendReadyTuple();
76      } else {
77        log("Error recieving pre-ready signal from follower");
78        inhibit = true;
79      }
80      //...
```

La fase iniziale consiste nell'inserimento dei due NXT nel contesto dell'applicazione, uno in seguito all'altro, indifferentemente dall'ordine di arrivo. La controparte sui robot è implementata nella classe CTWbrick:

```
61        LCD.clear();
62        selection = mainMenu.select();
63
64        if (selection == 0){ // Launch program
65
66          //choose role
67          role = roleMenu.select();
68
69          //connect via BT to PC
70          LCD2.cleanWrite(0, "Connecting BT...");
71          RemoteDevice btrd = Bluetooth.getKnownDevice(
              masterName);
72          if (btrd == null) Signals.signalError(1, "No masterTC
              ");
73
74          NXTConnection btc = Bluetooth.waitForConnection();//
              Bluetooth.connect(btrd);
75          if (btc == null) Signals.signalError(1, "Conn. fail")
              ;
76
77          dis = btc.openDataInputStream();
78          dos = btc.openDataOutputStream();
79
80          //communicate program to PC: CTW
81          LCD2.cleanWrite(0, "Connection OK.");
82          LCD2.cleanWrite(1, "Sending program");
83          BT.sendStringValues(dos, new String[]{this.pName}, 2)
              ;
84
85          //communicate role to PC: leader/follower
86          LCD2.cleanWrite(1, "Program sent.");
87          LCD2.cleanWrite(2, "Sending role...");
88          BT.sendStringValues(dos, new String[]{rolesItems[role
              ]}, 3);
```

```
89
90        //wait until PC says to start
91        LCD2.cleanWrite(2, "Role sent.");
92        LCD2.cleanWrite(3, "Waiting start...");
93        int go = dis.readInt();
94        if (go==1){           // ok1 signal
95          LCD2.cleanWrite(3, "Start received.");
96        } else if (go==2){     // can't start yet
97          LCD2.cleanWrite(3, "Wait for 2nd NXT");
98          go = dis.readInt();
99          if (go==1){        // ok2 signal
100           LCD2.cleanWrite(3, "Start recieved.");
101         } else {
102           Signals.signalProtocolError(4, "Unknown go2");
103         }
104       } else {
105         Signals.signalProtocolError(4, "Unknown go1");
106       }
107       LCD2.cleanWrite(0, "executing "+pName);
108       //...
```

Quando i robot sono pronti, si entra nella fase principale dell'applicazione che consiste in un loop. Il robot leader si muove in una nuova posizione, si ferma e invia le nuove coordinate al thread su PC, il quale le inserisce nel tuple center. A questo punto nel tuple center scatta una reazione che permette l'avanzamento dell'applicazione. Il thread su PC invia la nuova posizione al robot follower e aspetta che questo la raggiunga. Quando il robot follower è pronto, invia un segnale al thread su PC che inserisce una tupla adatta nel tuple center. Nel tuple center scatta un'altra reazione che ha come risultato finale l'invio di un segnale al robot leader che può procedere nella successiva posizione. Tutto ciò si ripete finché il leader raggiunge la posizione finale. L'implementazione relativa alla parte principale su PC è la seguente (sempre nella classe CTWpc):

```
80    //...
81    // loop START
82    while (!inhibit) {
83      int[] nextPos = new int[2];
84      nextPos[0] = BT.receiveInt(dis[leaderId], logger);
85      nextPos[1] = BT.receiveInt(dis[leaderId], logger);
86      if (nextPos[0] == -1 || nextPos[1] == -1) {
87        log("Finished execution.");
88        BT.sendInt(dos[followerId], nextPos[0], logger);
89        BT.sendInt(dos[followerId], nextPos[1], logger);
90        inhibit = true;
91        break;
92      }
93      log("Leader next pos: X=" + nextPos[0] + " Y=" +
           nextPos[1]);
94      sendNextPositionTuple(nextPos);
```

```
95
96          sem.wait();
97          BT.sendInt(dos[followerId], nextPos[0], logger);
98          BT.sendInt(dos[followerId], nextPos[1], logger);
99
100         ready = BT.receiveInt(dis[followerId], logger);
101         if (ready == 1) {
102           sendReadyTuple();
103         } else {
104           log("Error recieving ready signal from follower");
105           inhibit = true;
106           break;
107         }
108
109         sem.wait();
110         BT.sendInt(dos[leaderId], 1, logger);
111
112       } // loop END
113       //...
```

La controparte sui robot è implementata sempre nella classe CTWbrick e comprende i due stub relativi a leader e follower:

```
108         //...
109         //LOOP START
110         while (true){
111
112           if (role==0){ //leader stub
113
114             //select next way point (x,y)
115             LCD2.cleanWrite(1, "Calc next pos...");
116             if (!generateNextPosition()){
117               LCD2.cleanWrite(1, "At destination.");
118               LCD2.cleanWrite(2, "Notifying end...");
119               BT.sendIntValues(dos, new int[]{-1,-1}, 4);
120               LCD2.cleanWrite(2, "End notified.");
121               break;
122             }
123             LCD2.cleanWrite(1, "Next position:");
124             LCD2.cleanWrite(2, "X = ");
125             LCD.drawInt(nextX, 2, 5, 2);
126             LCD2.cleanWrite(3, "Y = ");
127             LCD.drawInt(nextY, 2, 13, 3);
128             LCD.refresh();
129
130             //move to new position
131             LCD2.cleanWrite(4, "Moving...");
132             moveToNextposition();
133
134             //communicate to TC new position
135             LCD2.cleanWrite(4, "Sending pos...");
136             BT.sendIntValues(dos, new int[]{nextX,nextY}, 4);
137
138             //wait for TC signal to continue
```

```
139                 LCD2.cleanWrite(4, "Waiting ACK...");
140                 int[] ack = BT.receiveIntValues(dis, 1, 4);
141                 if (ack[0]!=1){
142                   Signals.signalProtocolError(4, "Unknwn ACK");
143                 }
144
145               } else {  //follower stub
146
147                 // communicate ready
148                 LCD2.cleanWrite(1, "Sending ready...");
149                 BT.sendIntValues(dos, new int[]{1}, 2);
150                 LCD2.cleanWrite(1, "Ready sent.");
151
152                 //wait next way point (x,y)
153                 LCD2.cleanWrite(2, "Wait next pos...");
154                 int[] nextPos = BT.receiveIntValues(dis, 2, 3);
155
156                 nextX = nextPos[0];
157                 nextY = nextPos[1];
158                 if (nextX==-1){
159                   LCD2.cleanWrite(2, "At destination.");
160                   break;
161                 }
162                 LCD2.cleanWrite(1, "Next position:");
163                 LCD2.cleanWrite(2, "X = ");
164                 LCD.drawInt(nextX, 2, 5, 2);
165                 LCD2.cleanWrite(3, "Y = ");
166                 LCD.drawInt(nextY, 2, 13, 3);
167                 LCD.refresh();
168
169                 //move to new position
170                 LCD2.cleanWrite(4, "Moving...");
171                 moveToNextposition();
172               }
173             }//LOOP END
174             //...
```

Per ultimo si chiudono le connessioni e si liberano le risorse.

5.3.2 Risultati

Questo scenario è stato portato a termine con successo. Le principali difficoltà incontrate sono relative alla comunicazione via Bluetooth. LeJOS NXJ ha permesso di gestire facilmente il riconoscimento delle linee (con il sensore luminoso) permettendo ai singoli robot di seguire i singoli passi fino alla posizione finale.

5.4 Case study: Help me

Questo case study prevede l'utilizzo di due robot: il primo (*leader*) trasporta un oggetto in una zona franca A, da dove il secondo (*follower*) lo preleva e lo trasporta in una seconda zona franca B (inaccessibile al primo).

5.4.1 Algoritmo

La soluzione proposta per questo scenario utilizza il framework[3] sviluppato. Il preambolo è lo stesso del case study precedente. Il server accetta la richiesta del primo NXT, chiede l'applicazione di cui fa parte e fa partire un nuovo thread che gestisca il rispettivo scenario. Quando arriva il secondo NXT, il server accetta la richiesta e, dopo aver chiesto l'applicazione di cui fa parte, lo associa al thread già in esecuzione per questo scenario.

Il thread su PC che gestisce questo case study è implementato nella classe HMpc. La fase che gestisce l'avvio dell'applicazione è implementata nel modo seguente:

```
30   public void execute() throws Exception {
31     log("HM manager: getting first role...");
32     // get first role
33     roles[0] = BT.receiveStringValue(dis[0], logger);
34     log("First client connected: " + roles[0]);
35
36     // send wait
37     BT.sendInt(dos[0], 2, logger);
38     log("wait sent.");
39
40     // update TC
41     sendNewClientTuple(roles[0]);
42
43     log("Waiting for second client.");
44     while (now_clients < 2 && !inhibit) {
45       Thread.sleep(100);
46     }
47
48     // get second role
49     roles[1] = BT.receiveStringValue(dis[1], logger);
50     log("Second client connected: " + roles[1]);
51     sendNewClientTuple(roles[1]);
52
53     sem.wait();
54     if (!inhibit) {
55       for (int i = 0; i < roles.length; i++) {
56         if (roles[i].equals("leader"))
57           leaderId = i;
```

[3]vedi sezione 5.1

```
58          else if (roles[i].equals("follower"))
59            followerId = i;
60          else
61            log("Error @ roles");
62        }
63        // if valid send start to both clients
64        if (leaderId != -1 && followerId != -1) {
65          BT.sendInt(dos[0], 1, logger);
66          BT.sendInt(dos[1], 1, logger);
67        } else {
68          inhibit = true;
69          log("Cannot proceed. No leader, or no follower
              available. Stopping.");
70        }
71      }
72      //...
```

Essenzialmente la fase iniziale consiste nell'inserimento dei due NXT nel contesto dell'applicazione, uno in seguito all'altro, indifferentemente dall'ordine di arrivo. La controparte sui robot è implementata nella classe HMbrick:

```
63        LCD.clear();
64        selection = mainMenu.select();
65
66        if (selection == 0){ // Launch program
67
68          //choose role
69          role = roleMenu.select();
70
71          //connect via BT to PC
72          LCD2.cleanWrite(0, "Connecting BT...");
73          RemoteDevice btrd = Bluetooth.getKnownDevice(
              masterName);
74          if (btrd == null) Signals.signalError(1, "No masterTC
              ");
75
76          NXTConnection btc = Bluetooth.waitForConnection();//
              Bluetooth.connect(btrd);
77          if (btc == null) Signals.signalError(1, "Conn. fail")
              ;
78
79          dis = btc.openDataInputStream();
80          dos = btc.openDataOutputStream();
81
82          //communicate program to PC: HM
83          LCD2.cleanWrite(0, "Connection OK.");
84          LCD2.cleanWrite(1, "Sending program");
85          BT.sendStringValues(dos, new String[]{this.pName}, 2)
              ;
86
87          //communicate role to PC: leader/follower
88          LCD2.cleanWrite(1, "Program sent.");
```

```
89         LCD2.cleanWrite(2, "Sending role...");
90         BT.sendStringValues(dos, new String[]{rolesItems[role
              ]}, 3);
91 
92         //wait until PC says to start
93         LCD2.cleanWrite(2, "Role sent.");
94         LCD2.cleanWrite(3, "Waiting start...");
95         int go = dis.readInt();
96         if (go==1){          // ok1 signal
97           LCD2.cleanWrite(3, "Start received.");
98         } else if (go==2){    // can't start yet
99           LCD2.cleanWrite(3, "Wait for 2nd NXT");
100          go = dis.readInt();
101          if (go==1){        // ok2 signal
102            LCD2.cleanWrite(3, "Start recieved.");
103          } else {
104            Signals.signalProtocolError(4, "Unknown go2");
105          }
106        } else {
107          Signals.signalProtocolError(4, "Unknown go1");
108        }
109        LCD2.cleanWrite(0, "executing "+pName);
```

Una volta che i robot sono pronti, si entra nella fase principale dell'applicazione. Il robot leader raccoglie l'oggetto, segue la linea fino alla fine, deposita l'oggetto e invia un segnale al thread su PC informandolo che ha completato il proprio compito. Il thread su PC inserisce una tupla indicativa nel tuple center. A questo punto nel tuple center scatta una reazione che permette l'avanzamento dell'applicazione. Il robot follower riceve il segnale che gli da il via per andare a raccogliere l'oggetto. Una volta trovato e raccolto l'oggetto, il follower ricerca la linea e la segue fino alla fine, arrivando con l'oggetto nella zona franca B di destinazione.

L'implementazione relativa alla parte principale su PC è relativamente breve (sempre nella classe HMpc):

```
72     //...
73     // MAIN SECTION START
74     int leaderOK = BT.receiveInt(dis[leaderId], logger);
75     if (leaderOK==1){
76       sendReadyTuple();
77     } else {
78       log("Errore nella ricezione del messaggio dal leader");
79     }
80 
81     sem.wait();
82     BT.sendInt(dos[followerId], 1, logger);
83     // MAIN SECTION END
84     //...
```

La controparte sui robot è implementata sempre nella classe HMbrick e comprende i due stub relativi a leader e follower:

```
111        // PLAN START
112        if (role == 0) { // leader stub
113
114          // pickup object
115          Claw.grab();
116
117          // follow line to area of exchange
118          followLine();
119
120          // release object
121          Claw.release();
122
123          // send done signal to PC
124          LCD2.cleanWrite(4, "Sending OK...");
125          BT.sendIntValues(dos, new int[] { 1 }, 1);
126
127          // go back home
128          pilot.rotate(180);
129          followLine();
130
131        } else { // follower stub
132
133          // wait for go signal
134          LCD2.cleanWrite(2, "Waiting for PC...");
135          go = BT.receiveInt(dis);
136
137          // follow line to area of exchange
138          followLine();
139
140          // find object
141          detectAndReachObject();
142
143          // pickup object
144          Claw.grab();
145
146          //find line
147          goBackToLine();
148
149          // go back home
150          followLine();
151        }
```

L'implementazione delle procedure di rilevazione della linea da seguire, della rilevazione dell'oggetto da trasportare e del ritorno alla base è la seguente:

```
198  private void goBackToLine() {
199    pilot.rotate(180);
200    pilot.travel(nowDist);
```

```
201      pilot.rotate(-this.nowDeg);
202    }
203
204    private boolean findLineNearAndGo() {
205      int degLeft = -90, degRight = 90, degNow = 0, degStep =
            5;
206      int blackWhiteThreshold = 45;
207
208      // turn left
209      degNow = degLeft;
210      pilot.rotate(degNow);
211
212      // ping each 5 degrees
213      for (; degNow < degRight; degNow += degStep) {
214        // if black line found go forward and return true
215        if (lightSensor.readValue() < blackWhiteThreshold) {
216          pilot.forward();
217          return true;
218        }
219        pilot.rotate(degStep);
220      }
221
222      // else rotate to initial heading and return false
223      pilot.rotate(-degNow);
224      return false;
225    }
226
227    private void followLine() throws InterruptedException {
228      int blackWhiteThreshold = 45;
229      boolean finished = false;
230      lightSensor.setFloodlight(true);
231      while (!finished) {
232        if (lightSensor.readValue() > blackWhiteThreshold) {
233          // robot has lost line
234          pilot.stop();
235          if (!findLineNearAndGo()) {
236            // line has ended
237            finished = true;
238          }
239        }
240        Thread.sleep(10);
241      }
242      Thread.sleep(1000);
243    }
244
245    private void detectAndReachObject() {
246      int degLeft = -90, degRight = 90, degNow = 0, degStep =
            5;
247      int nearestIndex = 0;
248      int[] pings = new int[(Math.abs(degLeft - degRight)) /
            degStep];
249
250      // turn left
251      degNow = degLeft;
252      pilot.rotate(degNow);
253
```

```
254     // ping each 5 degrees to have a half star panoramic view
255     for (; degNow < degRight; degNow += degStep) {
256       this.sonicSensor.ping();
257       pings[degNow / degStep] = this.sonicSensor.getDistance
              ();
258       pilot.rotate(degStep);
259     }
260     pilot.rotate(-degNow);
261
262     // choose the nearest result assuming its the object
263     for (int i = 0; i < pings.length; i++) {
264       if (pings[i] < pings[nearestIndex]) {
265         nearestIndex = i;
266       }
267     }
268     // memorize the distance and the angle
269     this.nowDist = pings[nearestIndex];
270     this.nowDeg = degNow;
271
272     // reach the object
273     pilot.rotate(nowDeg);
274     pilot.travel(nowDist);
275   }
```

Per ultimo si chiudono le connessioni e si liberano le risorse.

5.4.2 Risultati

Questo case study ha avuto relativo successo. È stato superato il limite del lavoro precedente relativo al polling sul sensore luminoso, ma rimane comunque il limite relativo all'accuratezza nella rilevazione dell'oggetto da raccogliere.

5.5 Case study: Listen to me

Questo case study prevede l'utilizzo di due robot: il primo (*leader*) cerca un oggetto in un'arena, lo raccoglie e lo trasporta in una zona franca A. Poi avvisa il secondo (*follower*) con un segnale acustico. Il secondo riceve il segnale e attivandosi raccoglie l'oggetto dalla zona A e lo trasporta nella zona B (anche questa volta inaccessibile al primo robot).

5.5.1 Algoritmo

La soluzione proposta per questo scenario utilizza il framework[4].

[4] vedi sezione 5.1

Il preambolo per la connessione è lo stesso utilizzato negli altri case studies. Il server accetta la richiesta del primo NXT, chiede l'applicazione di cui fa parte e fa partire un nuovo thread che gestisca il rispettivo scenario. Quando arriva il secondo NXT, il server accetta la richiesta e, dopo aver chiesto l'applicazione di cui fa parte, lo associa al thread già in esecuzione per questo scenario.

Il thread su PC che gestisce questo case study è implementato nella classe LTMpc. La fase che gestisce l'avvio dell'applicazione è implementata nel modo seguente:

```
30   public void execute() throws Exception {
31     log("LTM manager: getting first role...");
32     // get first role
33     roles[0] = BT.receiveStringValue(dis[0], logger);
34     log("First client connected: " + roles[0]);
35
36     // send wait
37     BT.sendInt(dos[0], 2, logger);
38     log("wait sent.");
39
40     // update TC
41     sendNewClientTuple(roles[0]);
42
43     log("Waiting for second client.");
44     while (now_clients < 2 && !inhibit) {
45       Thread.sleep(100);
46     }
47
48     // get second role
49     roles[1] = BT.receiveStringValue(dis[1], logger);
50     log("Second client connected: " + roles[1]);
51     sendNewClientTuple(roles[1]);
52
53     sem.wait();
54     if (!inhibit) {
55       for (int i = 0; i < roles.length; i++) {
56         if (roles[i].equals("leader"))
57           leaderId = i;
58         else if (roles[i].equals("follower"))
59           followerId = i;
60         else
61           log("Error @ roles");
62       }
63       // if valid send start to both clients
64       if (leaderId != -1 && followerId != -1) {
65         BT.sendInt(dos[0], 1, logger);
66         BT.sendInt(dos[1], 1, logger);
67       } else {
68         inhibit = true;
69         log("Cannot proceed. No leader, or no follower
           available. Stopping.");
```

```
70        }
71      }
72      //...
```

La controparte sui robot è implementata nella classe LTMbrick:

```
63        LCD.clear();
64        selection = mainMenu.select();
65
66        if (selection == 0){ // Launch program
67
68          //choose role
69          role = roleMenu.select();
70
71          //connect via BT to PC
72          LCD2.cleanWrite(0, "Connecting BT...");
73          RemoteDevice btrd = Bluetooth.getKnownDevice(
              masterName);
74          if (btrd == null) Signals.signalError(1, "No masterTC
              ");
75
76          NXTConnection btc = Bluetooth.waitForConnection();//
              Bluetooth.connect(btrd);
77          if (btc == null) Signals.signalError(1, "Conn. fail")
              ;
78
79          dis = btc.openDataInputStream();
80          dos = btc.openDataOutputStream();
81
82          //communicate program to PC: LTM
83          LCD2.cleanWrite(0, "Connection OK.");
84          LCD2.cleanWrite(1, "Sending program");
85          BT.sendStringValues(dos, new String[]{this.pName}, 2)
              ;
86
87          //communicate role to PC: leader/follower
88          LCD2.cleanWrite(1, "Program sent.");
89          LCD2.cleanWrite(2, "Sending role...");
90          BT.sendStringValues(dos, new String[]{rolesItems[role
              ]}, 3);
91
92          //wait until PC says to start
93          LCD2.cleanWrite(2, "Role sent.");
94          LCD2.cleanWrite(3, "Waiting start...");
95          int go = dis.readInt();
96          if (go==1){           // ok1 signal
97            LCD2.cleanWrite(3, "Start received.");
98          } else if (go==2){    // can't start yet
99            LCD2.cleanWrite(3, "Wait for 2nd NXT");
100           go = dis.readInt();
101           if (go==1){         // ok2 signal
102             LCD2.cleanWrite(3, "Start recieved.");
103           } else {
104             Signals.signalProtocolError(4, "Unknown go2");
105           }
```

```
106         } else {
107           Signals.signalProtocolError(4, "Unknown go1");
108         }
109         LCD2.cleanWrite(0, "executing "+pName);
```

Dopo questa fase di sincronizzazione della connessione il primo robot cerca l'oggetto nell'arena, lo raggiunge e lo prende. Poi invia un segnale al thread su PC che inserisce una tupla nel centro di tuple. Scatta una reazione che informa il secondo robot dello stato dell'applicazione. Questo inizia ad emettere un segnale sonoro che il primo robot deve rilevare. Quando il primo robot individua il segnale sonoro, si orienta verso di esso e raggiunge la zona franca A, dove deposita l'oggetto. A questo punto il primo robot informa il thread su PC del completamento del suo compito e indietreggia un pochino. Il thread su PC inserisce una tupla nel tuple center che fa scattare una reazione. In seguito ad essa il secondo robot viene informato che può procedere a prendere l'oggetto e trasportarlo nella zona franca B.

Il codice sviluppato per la fase principale di questo case study è simile a quello dei case studies precedenti sia per quanto riguarda la parte su PC, che quella per NXT.

Per ultimo si chiudono le connessioni e si liberano le risorse.

5.5.2 Risultati

Questo case study ha un limite tecnologico relativo all'accuratezza dei sensori ultrasonico e sonoro. Questi hanno un impatto determinante nella rilevazione dell'oggetto da raccogliere e nella comunicazione della posizione via aria.

Capitolo 6

Conclusioni

Il lavoro svolto è stato molto educativo. I risultati raggiunti sono positivi, in quanto si sono superati alcuni limiti incontrati nei lavori precedenti.

6.1 La fase di sviluppo

Il processo di sviluppo con LeJOS può essere valutato per quanto riguarda il tempo necessario e la difficoltà da affrontare assumendo come parametro di confronto lo sviluppo con iCommand. La distribuzione della logica del progetto è cambiata: ora è in parte su PC, in parte sui robot. Questo cambiamento implica una fase di analisi in cui si deve decidere cosa e quanto delegare ai robot e al PC. In questo lavoro, al PC si è assegnato il compito della coordinazione (ReSpecT) e la parte relativa alla gestione della connessione. Ai robot è stato assegnato il compito di eseguire le azioni fisiche, avendo essi stessi la logica d'esecuzione dell'applicazione di cui fanno parte. Con iCommand solo il PC aveva questa conoscenza del flusso delle azioni. Lo scoglio da affrontare è relativo alla scrittura del firmware per i robot NXT. Infatti, iCommand era compatibile con il firmware standard NXT, mentre la nuova soluzione implica lo sviluppo di un firmware specifico per ogni scenario analizzato in parte. Inoltre, distribuendo la logica applicativa tra entità diverse (PC e robot), sorge la necessità di adottare un protocollo di comunicazione ad-hoc. In questo lavoro il protocollo utilizzato è a stati finiti e fa uso di invio di stringhe e interi.

6.2 Analisi delle prestazioni, limiti e miglioramenti

Il framework sviluppato si è dimostrato adatto a gestire tutti i case studies dei lavori precedenti. In generale, la struttura proposta (in termine di classi) può essere facilmente riutilizzata per inserire nuovi case studies, eventualmente più complessi.

Il più evidente vantaggio raggiunto riguarda la comunicazione della variazione dei valori rilevati dai sensori del NXT. Con iCommand tali valori erano richiesti ad intervalli regolari dal coordinatore, ovviamente un polling dispendioso. LeJOS NXJ permette che sia il NXT stesso ad inviare una lettura dei sensori qualora si verifichi una variazione dei valori rilevati.

I limiti rilevati riguardano l'accuratezza dei sensori e l'affidabilità delle connessioni Bluetooth.

Come avvertenza si indica che il codice sorgente scritto ha fatto uso di varie semplificazioni per motivi di studio, che però non ne pregiudicano la validità in vista di un utilizzo diverso.

Bibliografia

[1] *La Bibbia*. 1991. traduzione Nuova Diodati.

[2] Matteo Amaducci. Esperienze in robotica con lego midstorm e respect. tesi di laurea trienale, ALMA MATER STUDIORUM UNIVERSITA' DI BOLOGNA, 2009.

[3] Isaac Asimov. Runaround (circolo vizioso). *Astounding Science Fiction*, 1942.

[4] Isaac Asimov. *Robots and Empire (I robot e l'Impero)*. 1985.

[5] Matteo Casadei and Andrea Omicini. *From tuple spaces to tuple centres*. 2001.

[6] Matteo Casadei and Andrea Omicini. *ReSpecT Guide*, 2008.

[7] Matteo Casadei and Andrea Omicini. *Situated tuple centres in ReSpecT*. 2009.

[8] LeJOS Java for Lego Mindstorms website. *LeJOS homepage*. http://lejos.sourceforge.net/.

[9] LeJOS Java for Lego Mindstorms website. *The leJOS NXJ Tutorial*. http://lejos.sourceforge.net/nxt/nxj/tutorial/index.htm.

[10] Matteo Mosca. Coordinazione sociale di robot mindstorm in respect. tesi di laurea trienale, ALMA MATER STUDIORUM UNIVERSITA' DI BOLOGNA, 2011.

[11] Karel Čapek. *Rossum's Universal Robots (I robot universali di Rossum)*. 1920.