Integrazione e test del simulatore SimSpark in ROS Limiti e possibilità di utilizzo di ROS in ambiente industriale

Laureando: Alessandro Ordan

Relatore: Enrico Pagello

Corso di laurea in Ingegneria Informatica

Data di Laurea: 12 Dicembre 2011

Anno Accademico: 2011/12

Indice

Introduzione

La robotica è una scienza fortemente interdisciplinare e coinvolge aspetti di meccanica, automazione, elettronica, intelligenza artificiale, matematica, fisica, ecc, ma l'elemento fondamentale e che ne costituisce il "collante" è certamente l'informatica.

Il software gioca infatti un ruolo di estrema importanza in robotica, ruolo molto cresciuto negli ultimi anni e destinato a ulteriori sviluppi. Ne è un chiaro esempio l'evoluzione delle piattaforme hardware, che ha portato alla realizzazione di robot estremamente complessi e dotati di un elevato numero di gradi di libertà. In questi dispositivi l'aspetto informatico è fondamentale per la realizzazione di sistemi di controllo efficaci ed efficienti!

Analogamente anche i sistemi robotici si sono notevolmente sviluppati: l'operazione in ambienti non strutturati, così come l'interazione con l'uomo e con altri robot, porta alla necessità di potenziare gli aspetti di percezione. Sistemi robotici moderni sono caratterizzati da un gran numero di sensori, dispositivi di acquisizione, visione 2D e 3D, ecc. L'elaborazione efficiente dei dati, la distribuzione del calcolo in presenza di dispositivi embedded, l'interfacciamento con software e driver di terze parti, sono tutte problematiche di software engineering da affrontare per realizzare qualsivoglia applicazione.

Parlando infine di robotica umanoide tutti gli aspetti presentati risultano ulteriormente amplificati: dispositivi ancora più complessi, mobilità in ambienti tipicamente umani e dinamici, necessità di più alti livelli di percezione.

Appurata quindi quale sia l'importanza dell'informatica in robotica è ovvio ritenere altrettanto importante la risoluzione delle problematiche di sviluppo software. Queste sono caratterizzate non solo dai classici aspetti di software engineering, ma risultano arricchite dall'interazione finale con dispositivi hardware:

- riutilizzo e condivisione del codice: possibilità di riutilizzare quanto sviluppato in contesti differenti da quelli originari. Problematiche come quelle di *planning* sono comuni a molte applicazioni robotiche pertanto le soluzioni sviluppate dovrebbero poter essere riutilizzabili

- portabilità del codice: possibilità di utilizzare il codice in piattaforme o framework differenti dal quelli originari
- reperibilità e utilizzabilità di strumenti, librerie e quant'altro: estendere alla robotica la possibilità di disporre di componenti *off the shelf*, senza dover reinventare o ricreare ad ogni nuova applicazione soluzioni custom
- interoperabilità: tra software di terze parti e in presenza di molteplici driver
- astrazione: il software in robotica è caratterizzato da molteplici livelli di astrazione, si va dal controllo hardware di basso livello, a problematiche di coordinazione multirobot. A ciò si lega l'esigenza di utilizzare linguaggi di programmazione e interfacce differenti
- disponibilità di piattaforme hardware: lo sviluppo software è in generale indipendente da qualsiasi risorsa che non sia un computer, quando il software è applicato alla robotica questo non è più vero. La realizzazione di un programma di controllo può essere condotta autonomamente fino al momento in cui la disponibilità del robot non diviene necessaria: per testare il codice prodotto, per applicarlo allo scopo previsto. La disponibilità e il costo dei robot diviene quindi un fattore critico limitante alla ricerca informatica e allo sviluppo software in robotica
- difficoltà di testing e debugging su robot reali. Anche disponendo dei robot, testing e debugging su questi dispositivi risulta particolarmente problematico: elevato tempo di setup, limiti delle interfacce di controllo, limiti degli strumenti di debugging su dispositivi embedded

Soluzioni a queste problematiche si possono trovare nelle comuni tecnologie di software engineering. Al fine del riutilizzo, della condivisione e della portabilità del codice sono nati diversi strumenti che consentono di astrarre la programmazione dal sistema operativo ospite, dalle diverse tipologie di dispositivi e dai sistemi di comunicazione, veri e propri sistemi di sviluppo per la robotica (*Robotic Software System* o RSS) [1]. In questo contesto la competizione tra software proprietari e opensource ha visto la vittoria di questi ultimi, portando alla nascita di ferventi comunità di sviluppatori. Si assiste in particolare alla condivisione di un numero sempre crescente di moduli e librerie a supporto delle più comuni attività inerenti alla robotica: interfacciamento hardware (driver), motion planning, navigation, mapping, localization, fino alla simulazione.

ROS[S1], Player[S2], YARP[S5], OpenRTM-aist[S3], Orocos[S4], Urbi[S23], Microsoft Robotics Studio[S6], sono solo un breve elenco dei numerosi strumenti di sviluppo nati a questo scopo. In questo lavoro si è utilizzato e analizzato ROS, introdotto nella sezione 2, un RSS component-based nato da pochi anni ma che sta avendo una rapida diffusione in ambito accademico e, recentemente, anche industriale.

Per quanto riguarda le problematiche derivanti dal legame tra hardware e software in robotica, sono stati sviluppati molteplici sistemi di simulazione fisica e grafica, in ambiente sia 2D che 3D. La simulazione in robotica può coinvolgere una grande quantità di aspetti differenti: robot con ruote, robot umanoidi, robot volanti, aspetti di controllo, movimento, navigazione, interazione in sistemi multirobot, ecc. Non è possibile gestire con un unico software tanta variabilità di contesto: il dettaglio e l'accuratezza richiesta in una simulazione di grasp planning di un braccio robotico non sarà mai compatibile con la simulazione di uno sciame di centinaia di piccoli robot. Esistono pertanto decine di simulatori differenti, ciascuno con caratteristiche peculiari che li rendono idonei in particolari contesti. Nell'ambito della robotica umanoide in sistemi multirobot l'ambiente di simulazione SimSpark[S8] ha avuto particolare sviluppo, nascendo nel 2004 come piattaforma ufficiale di simulazione per la 3D Soccer Simulation League di RoboCup. Il simulatore, introdotto nel capitolo 1, è un sistema di simulazione fisica 3D, multi-agente e general purpose, che offre un elevato livello di astrazione e interfacce di controllo semplici.

La flessibilità e la praticità della simulazione porta con se anche alcuni svantaggi: dipendenza del codice dall'ambiente di simulazione, dal suo SDK o dai suoi sistemi di interfacciamento. Questi problemi di portabilità fan si che moduli sviluppati per la simulazione non possano essere in genere utilizzati direttamente sul robot reale. L'integrazione di simulatori all'interno di RSS consente di alleviare questi aspetti, offrendo la possibilità di rilasciare i servizi di simulazione con alto livello di astrazione e attraverso interfacce standard, migliorando la portabilità del codice. Parlando di ROS esistono già alcuni esempi di integrazione di questo tipo: Stage [S9] per la simulazione 2D, Gazebo [S10] e Webots per la simulazione 3D, tuttavia nessun progetto a noi noto è stato rivolto all'integrazione del simulatore SimSpark.

Il lavoro sviluppato in questa tesi è stato proprio l'integrazione della simulazione SimSpark in ROS, con la speranza di aver trattato un tema di interesse per tutta la comunità scientifica legata alla robotica umanoide e in particolare in ambito RoboCup. A dimostrazione della riuscita del sistema si è in oltre sviluppata la simulazione del robot umanoide Robovie-X, disponibile allo IAS-Lab[S11] del dipartimento di ingegneria informatica di Padova. Il framework prodotto sarà quindi utilizzato come

strumento didattico nel corso di Robotica Autonoma. In questo contesto l'integrazione di SimSpark in ROS risulta doppiamente utile: da un lato consente l'utilizzo del simulatore della RoboCup, particolarmente indicato ad un'applicazione didattica per la sua semplicità di utilizzo, dall'altro espone lo studente all'ambiente ROS.

Dall'utilizzo di ROS si sono potuti osservare aspetti interessanti e potenzialità per il suo impiego anche in ambito industriale, in particolare per il gran numero di strumenti messi a disposizione. Da queste osservazioni è stata effettuata una valutazione di ROS come strumento di sviluppo software per robotica industriale, individuandone pregi, difetti, limiti e possibilità di utilizzo.

Procedendo ad una breve descrizione della struttura del testo, i primi tre capitoli sono introduttivi ai sistemi software ROS e SimSpark, e potranno pertanto essere saltati qualora si disponesse di conoscenze sufficienti nell'abito. Nel capitolo 1 verrà presentato un confronto tra i principali sistemi si simulazione per robotica. Saranno trattate le motivazioni che hanno portato alla scelta di SimSpark, per passare ad una breve descrizione della nascita, dell'evoluzione e delle caratteristiche attuali del simulatore. Nel capitolo 2, analogamente, si presenterà il sistema di sviluppo ROS, il confronto con sistemi equivalenti e le motivazioni che hanno portato alla scelta di ROS come RSS per questo lavoro, per proseguire con la descrizione dettagliata delle sue caratteristiche. La specifica dei passi necessari all'installazione e alla configurazione degli ambienti ROS e SimSpark verrà riportata nel capitolo 3, prerequisiti necessari all'esecuzione di package sviluppati.

Nel capitolo 4 sarà presentato il lavoro svolto per l'integrazione del simulatore SimSpark in ROS, mentre nel capitolo 5 sarà mostrato un esempio di applicazione delle funzionalità del package nella simulazione del robot Robovie-X. Nel capitolo 6 sarà riportata la descrizione del robot umanoide Robovie-X e le caratteristiche del modello 3D realizzato. Infine nel capitolo 7 si svilupperanno gli aspetti legati all'applicazione di ROS in ambito industriale con particolare riferimento alle potenzialità e ai limiti attuali del sistema a tale impiego. Le conclusioni sul lavoro svolto, i risultati ottenuti e i possibili lavori futuri sono riportate nel capitolo 8.

Capitolo 1

Il simulatore SimSpark

SimSpark è un simulatore fisico multiagente con connotazione general-purpose, in ambiente tridimensionale, basato sul framework applicativo Spark [2]. Il confronto di SimSpark con i principali software di simulazione comparabili attualmente disponibili è riportato in tabella 1.1. Come si può osservare quasi tutti i simulatori offrono l'utilizzo di sensori e dispongono di avanzati motori fisici e grafici. Molti offrono interfacce API nei più comuni linguaggi mentre SimSpark e UsarSim si differenziano per l'utilizzo di un'interfaccia di rete. Sono quasi tutti general purpose, tranne OpenSim, sviluppato per la modellazione e lo studio di sistemi muscolo-scheletrici, molto potente per i suoi strumenti di risoluzione della cinematica e dinamica diretta e inversa. Infine fatta eccezione per Webots e Microsoft Robotics Developer Studio, entrambi closed source, sono tutti open source. MSRDS è rilasciato con licenza gratuita mentre Webots a pagamento. Webots presenta le caratteristiche più complete, ma la sua essenza closed source aggiunta ad un costo elevato lo rende poco desiderabile in ambito accademico e di ricerca. Analogamente molto interessante risulta MSRDS, in particolare per l'introduzione del concetto di WebService in robotica. Tuttavia anche in questo caso la chiusura del codice ha contribuito non poco alla scarsa diffusione del sistema.

Attualmente Gazebo risulta il sistema di simulazione maggiormente integrato con ROS, ne è stata infatti sviluppata una versione standalone indipendente dal framework Player, a cui era originariamente legato. Gazebo risulta tuttavia un simulatore piuttosto grezzo, e talvolta instabile. La scelta di sviluppare un wrapper ROS per SimSpark nasce dal suo utilizzo in ambiente RoboCup, il che lo rende necessariamente potente nel supporto alla simulazione multirobot, nonchè oggetto di probabile interesse per tutta la comunità scientifica della Soccer Simulation League.

Caratteristiche	SimSpark	Gazebo	Webots	UsarSim	Simtk-OpenSim	OpenRave	MSRDS
OS	Linux Windows Mac	Linux Windows Mac	Linux Windows Mac	Linux Windows	Linux Windows	Linux Windows	Windows
Linguaggi	TCP Sockets	C, C++ Python Ruby	C, C++ Python Java	TCP Sockets	C++ Matlab	C++ Python Matlab	C# MSVPL
Physics engine	ODE	ODE Bullet	ODE	Unreal Engine	Proprio	ODE Bullet	NVidia PhysX
Graphics engine	OpenGL OGRE	OGRE	OpenGL OGRE	Unreal Engine	Proprio	OpenGL	XNA Engine
Sensori	Odometria giroscopio accelerometro visione suono contatto pressione	Odometria giroscopio accelerometro visione GPS range finder ecc	Odometria giroscopio accelerometro visione GPS range finder ecc	Odometria giroscopio accelerometro visione RFID ecc	Odometria altri da modellare	Odometria laser scanner visione ecc	Odometria sonar visione ecc
Licenza	Open source	Open source	Closed source a pagamento	Open source	Open source	Open source	Closed source gratuito
Framework	Nessuno	Player ROS	ROS Player URBI	Moast Player	OpenSim Matlab- Simulink	Player ROS Matlab	MSRDS
Caratteristiche	General purpose RoboCup soccer simulation MRS	General purpose MRS	General purpose	General purpose RoboCup rescue simulation MRS	Specializzato nella simulazione di sistemi muscolo-scheletrici	Specializzato nella manipolazione	General purpose Service Oriented MRS

Tabella 1.1: Confronto tra i principali simulatori 3D

1.1 Breve storia

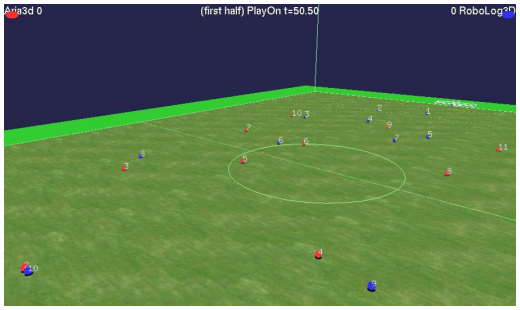
Il progetto SimSpark nasce nel 2003 quando, al Symposium RoboCup [3], fu proposto di passare dalla simulazione 2D fino ad allora utilizzata ad una simulazione 3D più realistica. La scelta divenne ufficiale, sempre nel 2003, durante la discussione della road map per la Soccer Simulation League, quando fu deciso di aggiungere alle competizioni la simulazione 3D. La prima versione di SimSpark fu utilizzata in RoboCup nel 2004; in essa i giocatori erano modellati come semplici sfere in ambiente fisico 3D (vedasi figura 1.1a). Questa struttura primitiva venne successivamente sviluppata, prima con una versione di robot umanoide grezzo realizzato sulla base del robot Fujitsu HOAP-2, poi con il robot umanoide SoccerBot e infine con la versione attuale del robot Nao. Il simulatore è nato con una struttura modulare che ne consentisse una connotazione general purpose, tuttavia ne è stata sviluppata solo la versione per la simulazione del gioco del calcio.

1.2 Architettura di SimSpark

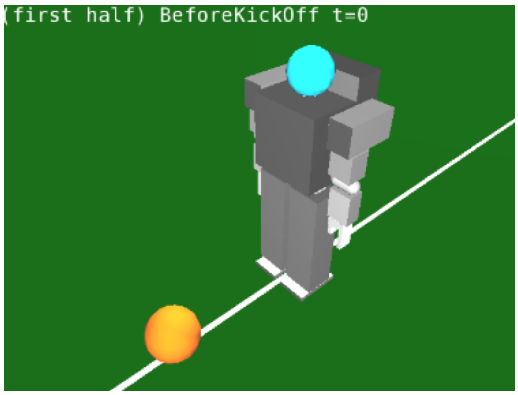
SimSpark è costruito sul framework Spark che fornisce modularità al sistema e funzionalità di astrazione: accesso al disco, logging, caricamento di librerie condivise. In particolare Spark fornisce un sistema di caricamento dinamico di oggetti C++ in una sorta di filesystem di istanze e l'interfacciamento di queste da script Ruby. Spark è composto da un insieme di librerie trua cui le principali sono:

- `Zeitgeist`: fornisce un meccanismo per operare con Class Object C++ e organizza le istanze attive in una gerarchia strutturata come un filesystem. Com'è noto infatti C++ non dispone nativamente di alcuna gestione per le classi oggetto, ne la possibilità di instanziare classi note solo runtime, per questo Zeitgeist fornisce un sofisticato class factory system. Alcuni oggetti, detti server, forniscono servizi all'intero sistema, tra questi il FileServer, LogServer e lo ScriptServer.

- `Oxygen`: libreria che fornisce servizi di simulazione e meccanismi utili alla gestione dell'ambiente 3D, come trasformazioni geometriche, costruzione di primitive geometriche elementari, primitive di collisione e corpi dotati di proprietà fisiche (massa, inerzia, attrito, ecc) e fornisce interfacce per la simulazione fisica di corpi rigidi. Oxygen in oltre gestisce e aggiorna gli agenti connessi alla simulazione e il monitor che la visualizza. Infine fornisce interfacce per la spe-



(a) Prima versione di SimSpark - robot modellati come sfere



(b) Seconda versione di SimSpark - robot umanoide basato su HOAP-2 di Fujitsu

(c) Terza versione di SimSpark - robot umanoide SoccerBot

(d) Attuale versione di SimSpark - robot umanoide Nao

Figura 1.1: Versioni di SimSpark

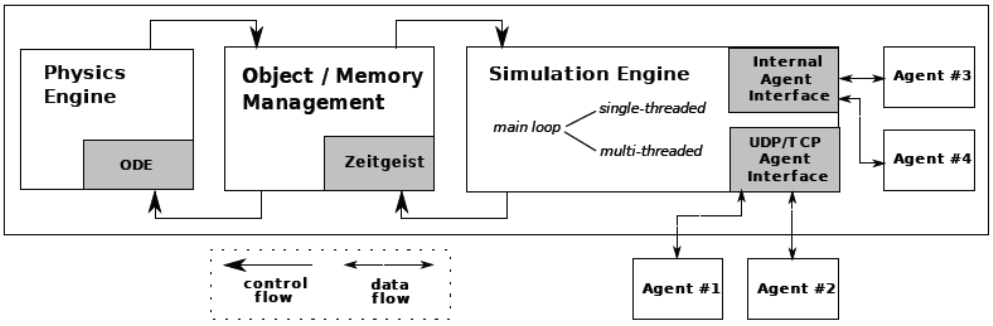


Figura 1.2: Architettura software di SimSpark

cifica di un run loop (vedere 1.3.1) personalizzato con l'aggiunta o l'esclusione di plugin rispetto l'esecuzione del loop standard. L'ambiente della simulazione viene descritto nel cosiddetto *scene graph*, un insieme di classi anch'esse caricate nella gerarchia organizzata da Zeitgeist. La specifica dello scene graph avviene in appositi script ruby (vedere a proposito il paragrafo 1.5.1).

- `Kerosin`: libreria che si occupa della visualizzazione della scena. Le diverse funzionalità sono lasciate a specifici plugin allo scopo di rendere flessibile e personalizzabile la configurazione.

Vi sono poi altre librerie a supporto delle operazioni geometriche, moltiplicazione di matrici, rotazioni, ecc.

Il motore fisico del simulatore è esterno al nucleo del framework ed è attualmente realizzato con ODE (Open Dynamic Engine). L'interfaccia con Zetigeist è stata tuttavia sufficientemente astratta da rendere possibile l'utilizzo di altri motori fisici come Bullet o PhysX.

1.3 Elementi fondamentali

Gli elementi fondamentali del sistema SimSpark sono:

- **Il server:** è il core del simulatore, eseguendo il Simulation Update Loop gestisce l'avanzamento della simulazione fisica, la grafica e l'interazione con gli agenti

- **Il monitor:** interfaccia di visualizzazione dell'ambiente virtuale, può essere interna o esterna al server. Nel caso di sia esterno, il monitor riceve le informazioni per la renderizzazione in streaming

- **La rete:** è uno dei punti di forza del sistema SimSpark. Le interazioni con il server possono essere effettuate attraverso chiamate API o attraverso comunicazioni di rete (TCP/IP). Nel framework presentato si utilizza l'interfaccia di rete.

- **Gli agenti:** software che controllano i robot determinandone il behaviour. Possono essere interni al sistema di simulazione, utilizzando chiamate API, o esterni utilizzando l'interfaccia di rete, che presenta il vantaggio di disaccoppiare il controllo dalla simulazione, offrendo in oltre un livello di astrazione dal linguaggio di programmazione. Per quando riguarda il controllo degli agenti, SimSpark fornisce due tipologie di elementi per l'interazione:

  – **I perceptor:** sono output del simulatore, sensori e dispositivi attraverso i quali effettuare letture dello stato del robot simulato e dell'ambiente di simulazione stesso. Tra questi si trovano sensori d'odometria, accelerometri, giroscopi, sensori di forza e di contatto. Per la simulazione del calcio sono disponibili anche un sensore di vista e d'udito, nonché "sensori" dello stato del gioco: tempo attuale di gioco dall'inizio del match, risultato attuale della gara, stato del gioco (fuorigioco, calcio d'angolo, rigore, ecc).

  – **Gli effector:** sono gli input della simulazione, meccanismi di attuazione. Tra questi si trovano: controllo dei giunti (sia hinge che universal) e, per la simulazione del calcio, anche un sistema semplificato di riproduzione vocale, nonché un controllo degli agenti in modalità trainer per la disposizione iniziale dei robot sul campo da gioco.

  La modularità del framework Spark e il suo funzionamento a plugin lasciano grande liberà alla creazione di nuovi effector e perceptor.

1.3.1 Il server

L'applicazione *rcssserver3d* è il server di SimSpark e si occupa di gestire la simulazione e il suo avanzamento, la renderizzazione della scena nel caso di monitor interno o la comunicazione con il monitor esterno, e l'interazione con gli agenti connessi. Lo stato della simulazione è aggiornato durante l'esecuzione loop di simulazione (SUL, Simulation Update Loop). Il SUL è personalizzabile con l'utilizzo di plugin definiti *SimControlNode*. E' possibile registrate SimControlNode presso il server di simulazione associandoli agli eventi del SUL. Essi saranno così richiamati in

risposta ai corrispondenti aventi. I tipi di eventi che possono essere generati durante la simulazione, come da come da figura 1.4, sono:

- **init**: lanciato una sola volta all'avvio del server di simulazione
- **start cycle ⇒ sense agent ⇒ act agent ⇒ physical update ⇒ end cycle**: eventi ripetuti nel loop di simulazione. Start cycle viene lanciato all'inizio di ogni ciclo, sense agent per la lettura delle informazioni provenienti dai perceptor e comunicazione di queste agli agenti collegati. Act agent è l'evento associato al controllo degli effector secondo quanto predisposto dagli agenti nel ciclo di simulazione precedente. Nel physical update, gestito da ODE, avviene il controllo e l'avanzamento della simulazione fisica dell'ambiente e degli agenti: posizione, velocità, accelerazione, controllo collisioni, ecc. Infine end cycle per la gestione della terminazione del ciclo si simulazione.
- **done**: lanciato una sola volta alla chiusura del server di simulazione
- **time measurement**: eventi per l'aggiornamento del tempo di simulazione, utilizzabili ad esempio per la sincronizzazione con il tempo reale.

Durante l'esecuzione del SUL, eseguito con frequeza di 50Hz (periodo 20ms), il server non gestisce la latenza delle comunicazioni, pertanto non è garantita la riproducibilità degli eventi. La ripetizione della medesima simulazione nelle stesse condizioni può portare pertanto a risultati differenti. Infine si riporta che l'evento act agent è relativo alle azioni predisposte dall'agente nel ciclo di esecuzione precedente, vi è infatti un periodo di ritardo tra il comando di un'azione e la sua esecuzione secondo lo schema in figura 1.3 Esiste un'ulteriore implementazione del ciclo di simulazione, una

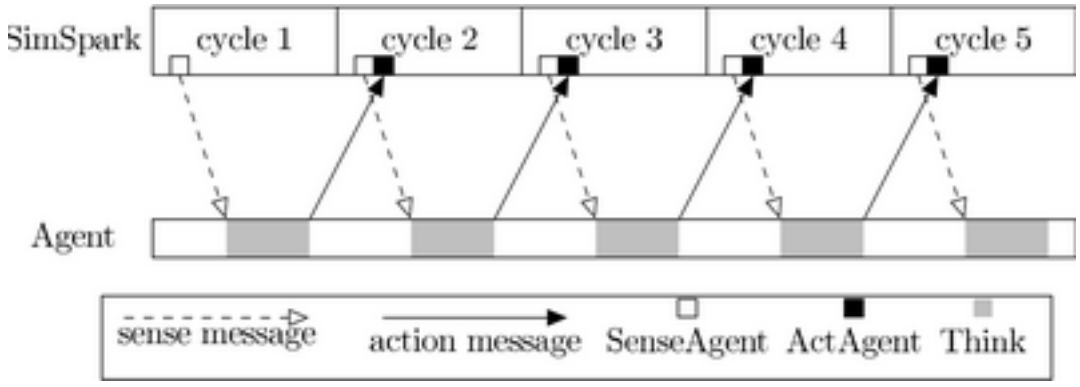


Figura 1.3: Ritardo di esecuzione delle azioni del ciclo di simulazione

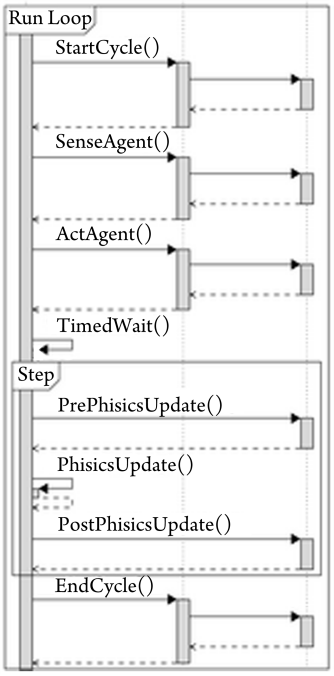


Figura 1.4: Single-threaded simulation Update Loop di SimSpark

versione multi-threaded sperimentale per macchine multiprocessore. La specifica del tipo di loop utilizzare può essere effettuata, come sempre, attraverso appositi script, in questo caso specificando `simulationServer.setMultiThreads(false)` o `simulationServer.setMultiThreads(true)` nello script di configurazione *spark.rb*. In modalità multi-threaded ogni SimControlNode deve gestire concorrentemente parti differenti della simulazione cosicchè da consentire l'esecuzione un'esecuzione in parallelo priva di conflitti. In oltre lo stato della simulazione è mantenuto in un albero chiamato *active scene*. Il motore fisico e i SimControlNode interagiscono attraverso l'active scene in maniera concorrente, dato che l'aggiornamento della simulazione fisica avviene sulla base dello stato iniziale del ciclo, mentre i SimControlNode aggiornano lo stato attuale. Pertanto il physical update può essere eseguito parallelamente ai SimControlNode. Questa versione del loop è schematizzata in figura 1.5

1.4 Interazione con il simulatore

Si è scelto di interfacciarsi con il simulatore attraverso la sua interfaccia di rete. Questa scelta consente in primo luogo indipendenza dal linguaggio di programmazione scelto per lo sviluppo degli agenti, ma anche una separazione fisica, consentendo l'installazione di server e l'esecuzione degli agent su macchine differenti; condizione

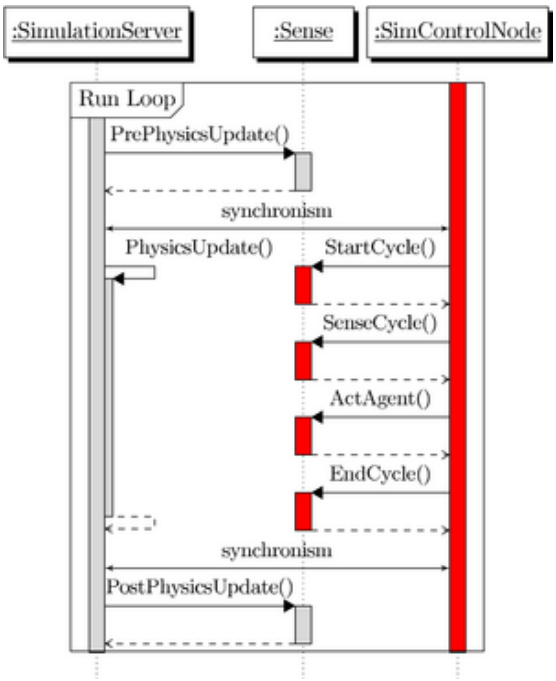


Figura 1.5: Multi-threaded simulation Update Loop di SimSpark

fondamentale per l'ambiente RoboCup. L'interfaccia di rete di SimSpark è basata su socket TCP; la porta di ascolto del server è di default la 3100. Ogni robot simulato è associato ad un canale di comunicazione TCP e l'interazione con il simulatore, effettuata attraverso lo scambio di messaggi, perdura quanto la connessione. I messaggi hanno tutti un formato standard e utilizzano *S-Expression* (o *sexp*), sotto forma di stringhe, come struttura dati di base. Una S-Expression di SimSpark può essere definita attraverso la seguente grammatica:

$$\begin{aligned} character &\rightarrow A|B|...|Z|1|...|9 \\ atom &\rightarrow character^{+} \\ list &\rightarrow (s\_expression^{*}) \\ s\_expression &\rightarrow atom|list \end{aligned}$$

L'uso delle S-Expression come formato di dati consente, rispetto altre tipologie di struttura, un parsing facilitato mantenendo un adeguata leggibilità dei messaggi, facilitando così anche il debugging. Un ulteriore vantaggio del sistema è l'estensibilità: aggiungere nuovi elementi alla simulazione, perceptor, effector e corrispondenti messaggi, può essere fatto in maniera semplice e senza side-effects, poiché i nuovi messaggi saranno ignorati dai moduli preesistenti e che ancora non li supportano. Analogamente grazie alla struttura ricorsiva delle S-Expression è possibile estendere messaggi preesistenti. I messaggi sono codificati in formato ASCII con un byte per

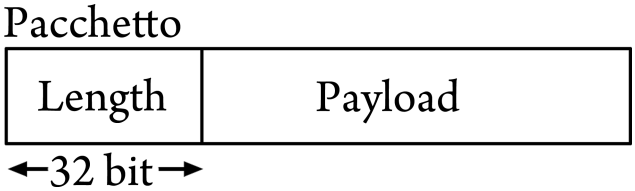


Figura 1.6: Struttura dei pacchetti

carattere. Come evidenziato in figura 1.6 ogni pacchetto è composto di un header di 32 bit, contenente l'unico campo length che specificante la lunghezza in byte del payload, e il payload. Quest'ultimo contiene la sequenza di byte della stringa che compone l' S-Expression o la sequenza di S-Expression che costituisce un burst di messaggi. Il campo length deve essere specificato secondo la notazione BigEndian. I messaggi ricevibili e inviabili dipendono dalle caratteristiche del robot utilizzato e dalla modalità di simulazione. I messaggi si classificano in due tipologie principali:

- Effector Message: portano comandi verso gli attuatori del robot o verso il server (Agent → Server)
- Perceptor Message: portano informazioni provenienti dai perceptor del robot e dell'ambiente verso l'agente (Server → Agent)

Ognuno dei due tipi, a sua volta, si divide in messaggi *general purpose* e messaggi *soccer-specific*. Tra gli effector message principali ci sono quelli per l'inizializzazione del robot:

- ***CreateEffector message:*** è il primo messaggio che deve essere inviato a seguito dell'apertura del canale di comunicazione con il server. E' un messaggio general-purpose e consente la specifica del modello di robot da caricare, attraverso l'indicazione dello specifico *scene descriptor file*, introdotto nel paragrafo 1.5.1
- ***InitEffector message:*** è un messaggio specifico per la simulazione del calcio. In questo contesto è il secondo messaggio che deve essere inviato e specifica il nome del nuovo robot e la squadra di appartenenza.

Superata questa fase di inizializzazione, il server inizia a comunicare i valori forniti dai diversi perceptor di cui il robot dispone. Allo stesso tempo il processo può inviare al server messaggi di controllo degli attuatori. Si riportano i principali effector e perceptor assieme ai corrispondenti messaggi. I nomi indicati negli esempi sono puramente indicativi e relativi alla descrizione RSG utilizzata (si veda a proposito

il paragrafo 1.5.1 e 1.5.2), contenente, oltre alla descrizione geometrica, fisica e di collisione del robot, anche le caratteristiche dei giunti, degli attuatori e dei sensori. I messaggi vengono inviati dal server all'agente ad ogni ciclo (non tutti i messaggi sono presenti in ogni ciclo) raggruppati in un unico pacchetto.

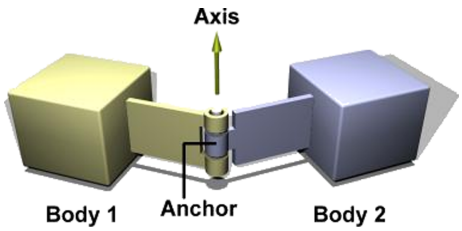
1.4.1 General purpose effector

I general purpose effector sono disponibili in ogni tipologia di simulazione (calcistica o generica):

- **CreateEffector:** è un effector del server, consente la specifica del modello di robot da caricare, indicato come percorso del file rsg contenente la descrizione del robot. Il percorso è relativo alla directory di installazione del server. Per maggiori informazioni sulle directory utilizzate dal simulatore si faccia riferimento al paragrafo 1.5.2.

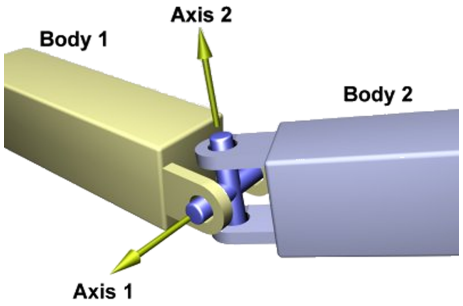
FORMATO	(SCENE <filename>)
ESEMPIO	(scene rsg/agent/robovie_x/robovie_x.rsg)
DESCRIZIONE	Carica nella scena attuale il robot umanoide Robovie-X associandolo all'agente connesso

- **Hinge Joint:** è l'effector di un giunto a singolo grado di libertà (cerniera), i limiti del giunto sono specificati all'interno dello scene descriptor file (si veda a proposito il paragrafo 1.5.1). Il messaggio di azionamento di un giunto Hinge specifica la velocità in gradi per ciclo a cui attivare l'attuatore del giunto. Tale velocità è mantenuta fino al prossimo messaggio di azionamento. Se viene raggiunto il fine corsa il giunto continuerà a urtare il fine corsa, con oscillazioni attorno alla posizione finale, fino ad un nuovo comando di velocità.



FORMATO	(<name> <ax>)
ESEMPIO	(lle1 1)
DESCRIZIONE	Imposta una velocità di 1 grado per ciclo (50 gradi al secondo) per l'effector `lle1`. Secondo la convenzione comunemente utilizzata in SimSpark ciò corrisponde all'effector del primo giunto della gamba sinistra del robot, secondo l'espressione regolare $(h\|((l\|r)\cdot(a\|l)))e(0..9)^+$ dove, le lettere per esteso corrispondono a: (head|((left|right) · (arm|leg)))effector(0..9)$^+$. Si veda a proposito il paragrafo 1.5.2.

- **Universal Joint:** è un giunto con due gradi di libertà. Il messaggio di azionamento specifica le velocità in gradi per ciclo a cui attivare i due attuatori del giunto contemporaneamente.



FORMATO	(<name> <ax1> <ax2>)
ESEMPIO	(lae1 -2.3 1.2)
DESCRIZIONE	Imposta una velocità di -2.3 gradi per ciclo e 1.2 gradi per ciclo ai due assi di rotazione del giunto. Secondo la sopra citata convenzione ciò corrisponde all'effector del primo giunto del braccio sinistro, qualora il robot sia dotato di un giunto universal in tale posizione. Il robot Robovie-X non possiede alcun giunto universal.

1.4.2 Soccer effector

Sono effector specifici per la simulazione del calcio, comunque utilizzabili in una simulazione generica indicando l'uso dei plugin associati:

- **InitEffector:** è un effector del server, consente la specifica del nome del robot caricato e della squadra di appartenenza

FORMATO	(init (unum <playernumber>)(teamname <your-teamname>))
ESEMPIO	(init (unum 10)(teamname RoboTeam))
DESCRIZIONE	Assegna al robot il numero 10 e lo inserisce tra i membri della squadra *RoboTeam*

- **Beam:** è un effector del server, consente la disposizione del robot sul campo da gioco prima dell'inizio della partita. Il messaggio specifica le coordinate XY relative al centro del campo in cui posizionare il robot e l'angolo iniziale in gradi del robot rispetto l'asse X. Per le convenzioni sulla disposizione del e le dimensioni del campo da gioco si faccia riferimento al sito [S19] contenente le regole aggiornate.

FORMATO	(beam <x> <y> <rot>))
ESEMPIO	(beam 10.0 -10.0 0.0)

- **Say:** è una sorta di altoparlante con cui il robot comunica verbalmente con gli altri robot inviando messaggi in broadcast sotto forma di stringhe prive di spazi. E' possibile configurare l'effector tramite script impostando la distanza massima di propagazione del "suono".

FORMATO	(say <message>)
ESEMPIO	(say Salve_A_Tutti)

1.4.3 General purpose perceptor

Perceptor installabili nel robot e che forniscono output dalla simulazione con una frequenza che dipende dal tipo di perceptor.

- **Gyro Rate:** è un sensore giroscopio che viene fissato ad un componente del robot. Fornisce indicazioni circa la variazione di orientamento del link del robot a cui è vincolato. La variazione è espressa nelle tre coordinate spaziali del sistema di riferimento del componente stesso. Il messaggio, identificato con il nome `GYR`, specifica quindi il nome del giroscopio e tre valori descriventi la variazione di velocità angolare nell'ultimo ciclo (in gradi al secondo) nei tre assi corrispondenti. Viene inviato un messaggio per ogni giroscopio per ogni ciclo di aggiornamento. L'effector GYR non implementa alcun modello di errore.

FORMATO	(GYR (n <name>) (rt <x> <y> <z>))
ESEMPIO	(GYR (n torso_gyr) (rt 0.01 0.5 0.46))
DESCRIZIONE	Il giroscopio denominato *torso_gyr* ha rilevato una variazione di orientamento pari a 0.01, 0.5 e 0.46 gradi/s negli assi x, y e z rispettivamente, nel sistema di riferimento del link torso

- **Hinge Joint:** è il sensore di odometria, assimilabile ad un encoder, del corrispondente effector. Il messaggio è identificato dal nome HJ e contiene come parametri il nome del giunto e la posizione in gradi rilevata rispetto la sua origine. Il messaggio viene inviato per ogni giunto per ogni ciclo di aggiornamento e non implementa alcun modello di errore.

FORMATO	(HJ (n <name>) (ax <ax>))
ESEMPIO	(HJ (n laj3) (ax -1.02))
DESCRIZIONE	La posizione attuale del giunto hinge denominato `laj3` rispetto la sua origine è di -1.02 gradi. Secondo la convenzione utilizzata da SimSpark, $(h\|(l\|r)\cdot(a\|l))j(0..9)^{+}$, ciò corrisponde al terzo giunto del braccio sinistro.

- **Universal Joint:** è il sensore di odometria del corrispondente effector. Il messaggio è identificato dal nome UJ e contiene come parametri il nome del giunto e le due posizioni rilevate in corrispondenza dei due gradi di libertà del giunto. Il messaggio viene inviato per ogni giunto per ogni ciclo di aggiornamento.

FORMATO	(UJ (n <name>) (ax1 <ax1>) (ax2 <ax2>))
ESEMPIO	(UJ (n laj1 2) (ax1 -1.32) (ax2 2.00))
DESCRIZIONE	La posizione attuale del giunto universal denominato `laj1` rispetto la sua origine nei due assi è di -1.32 e 2.00 gradi rispettivamente. Esso corrisponde a un ipotetico primo giunto del braccio sinistro.

- **Touch:** E' un sensore di contatto che commuta qualora il componente del robot a cui è vincolato entri in contatto con qualche elemento della simulazione. Il messaggio è identificato dal nome TCH e specifica come parametri il nome del sensore e lo stato, 0 se assenza di contatto, 1 se presenza di contatto. Il messaggio è inviato ad ogni ciclo di aggiornamento della simulazione.

FORMATO	(TCH n <name> val <bit>)
ESEMPIO	(TCH n bumper1 val 1)
DESCRIZIONE	Il sensore di contatto denominato `bumper1` ha rilevato contatto.

- **Force Resistance:** sensore che rileva la forza che agisce sul link del robot ove è collocato. Il messaggio, identificato con il nome FRP, specifica il nome n del sensore, i valori del vettore del punto di origine della forza c, espresso in coordinate relative al sistema di riferimento del corpo a cui il sensore è solidale, e il vettore della forza risultante f. Sensori di questo tipo sono spesso collocati sotto i piedi del robot. Il messaggio viene inviato ad ogni ciclo di aggiornamento ma sono in presenza di contatto effettivo (forza non nulla).

FORMATO	(FRP (n <name>) (c <px> <py> <pz>) (f <fx> <fy> <fz>))
ESEMPIO	(FRP (n lf) (c -0.14 0.08 -0.05) (f 1.12 -0.26 13.07))
DESCRIZIONE	Il sensore di forza denominato `lf` ha rilevato un insieme di forze applicata al link cui è vincolato la cui risultante è centrata nel punto $(X, Y, Z) = (-0.14, 0.08, -0.05)$, rispetto il sistema di riferimento del link, e di valore dato dal vettore $(F_X, F_y, F_z) = (1.12, -0.26, 13.07)$ Newton per componente.

- **Accelerometer:** è un accelerometro e fornisce un valore di accelerazione del link a cui è vincolato, comprensivo dell'accelerazione di gravità. Il messaggio è identificato dal nome ACC e specifica il nome del sensore e il vettore descrivente l'accelerazione misurata ed è inviato ad ogni ciclo di aggiornamento.

FORMATO	(ACC (n <name>) (a <x> <y> <z>))
ESEMPIO	(ACC (n torso_acc) (a 0.00 0.00 9.81))
DESCRIZIONE	Il sensore di accelerazione denominato `torso_acc` ha rilevato un'accelerazione data dal vettore $(a_x, a_y, a_z) = (0.00, 0.00, 9.81)$.

1.4.4 Soccer perceptor

Si riportano infine i perceptor specifici al gioco del calcio:

- **Vision:** Sono disponibili due tipi di sensori di visione, uno omnidirezionale e uno a campo ristretto (120°). Il sensore fornisce indicazione degli elementi della simulazione visualizzati, che possono essere:
  - altri giocatori: di cui vengono indicati i nomi, la squadra di appartenenza e le parti del corpo che sono visualizzate, con le relative coordinate.
  - linee del campo: di cui viene fornita una coppia di punti che identificano la posizione della linea
  - marker del campo: posizionati ai bordi del campo e in corrispondenza delle porte. Viene fornito l'identificativo e la posizione del marker.
  - la palla: ne viene fornita la posizione.

  Ciascuna posizione è specificata in coordinate sferiche rispetto al sistema di riferimento della telecamera, secondo quanto indicato in figura 1.7. Il messaggio è identificato dal nome `See` e può specificare: una o più sottostrutture relative agli oggetti visualizzati, palla o marker, identificati dal proprio nome e associati alla posizione; può contenere una o più sottostrutture relative alla visione di giocatori (struttura identificata dal nome `P`) che a sua volta specifica il nome e la squadra del giocatore e le parti di corpo visualizzate; infine una o più sottostrutture relative alle linee del campo visualizzate, identificate dal nome `L` e associate ciascuna ad una coppia di punti. Il sensore di visione implementa due tipologie di errore:
  - errore di calibrazione della posizione della videocamera: uniformemente distribuito tra -5 e +5 mm nei tra assi x,y e z.
  - rumore dinamico E_d, E_φ e E_ϑ con distribuzione normale a media nulla e scarto quadratico medio rispettivamente 0.0965, 0.1225 e 0.1480. L'errore E_d è in oltre pesato di un fattore distanza/100.

  Il messaggio è inviato ogni tre cicli di aggiornamento. In tabella 1.2 è riportato il formato del messaggio ed un esempio.

- **Game State:** è uno pseudo-sensore del server che fornisce informazioni circa lo stato del gioco, come il tempo trascorso dall'inizio di ogni tempo e lo stato attuale della partita. Il messaggio è identificato dal nome GS e presenta un parametro per il tempo t e uno per lo stato pm.

FORMATO	(GS (t <time>) (pm <playmode>))
ESEMPIO	GS (t 0.00) (pm BeforeKickOff)

FORMATO	(See (<name> (pol <distance> <angle1> <angle2>))$^+$ (P (team <teamname>)(id <playerID>) (<bodypart> (pol <distance> <angle1> <angle2>))$^+$)$^+$ (L (pol <distance> <angle1> <angle2>) (pol <distance> <angle1> <angle2>))$^+$)
ESEMPIO	(See (G2R (pol 17.55 -3.33 4.31)) (G1R (pol 17.52 3.27 4.07)) (F1R (pol 18.52 18.94 1.54)) (F2R (pol 18.52 -18.91 1.52)) (B (pol 8.51 -0.21 -0.17)) (P (team teamRed) (id 1) (head (pol 16.98 -0.21 3.19)) (rlowerarm (pol 16.83 -0.06 2.80)) (llowerarm (pol 16.86 -0.36 3.10)) (rfoot (pol 17.00 0.29 1.68)) (lfoot (pol 16.95 -0.51 1.32))) (P (team teamBlue) (id 3) (rlowerarm (pol 0.18 -33.55 -20.16)) (llowerarm (pol 0.18 34.29 -19.80)))) (L (pol 12.11 -40.77 -2.40) (pol 12.95 -37.76 -2.41))
DESCRIZIONE	Dalla sua posizione il robot vede i due pali della porta di destra (`G1R` e `G2R`), le bandierine degli angoli della metà campo destra (`F1R` e `F2R`), la palla, i giocatori numero 1 e 3 appartenenti rispettivamente alla squadra *teamRed* e *teamBlue*, di cui vede specifiche parti del corpo, e una linea del campo da gioco

Tabella 1.2: Formato del messaggio del vision perceptor

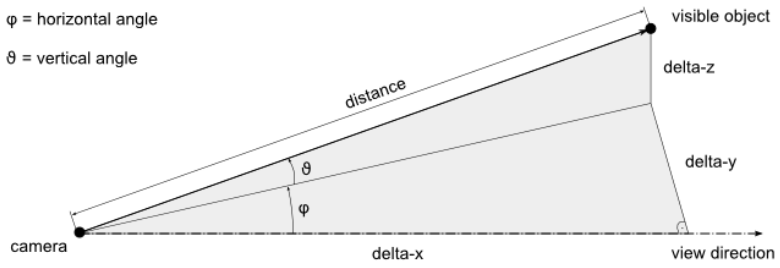


Figura 1.7: Convenzioni dei valori del perceptor `See`

- **Agent State:** è uno pseudo-sensore del robot che fornisce informazioni sul livello di batteria e temperatura interna del robot.

FORMATO	(AgentState (temp <degree>) (battery <percentile>))
ESEMPIO	(AgentState (temp 48) (battery 75))

- **Hear:** è un sensore d'udito, il duale del corrispondente effector Say. E' in grado di ricevere i messaggi pronunciati dagli altri robot fornendo anche un'indicazione circa l'angolo di provenienza del messaggio, rispetto il riferimento della testa. Fornisce infine anche un'indicazione dell'istante di tempo in cui il messaggio è stato ricevuto. Sono previsti limiti per la lunghezza dei messaggi, la distanza a cui il messaggio può essere udito e il numero di messaggi udibili contemporaneamente.

FORMATO	(hear <time> (self|<direction>) <message>)
ESEMPIO	(hear 12.3 self Salve_A_Tutti) (hear 12.3 -12.7 Salve_A_Tutti)
DESCRIZIONE	Il primo esempio riporta il messaggio ricevuto da un robot a seguito del proprio invio della stringa "Salve_A_Tutti". Il secondo l'eventuale messaggio ricevuto da un altro robot posto nelle vicinanze del primo.

1.5 Specifica dell'ambiente di simulazione

SimSpark si prefigura come simulatore general-purpose e come tale risulta essere altamente configurabile. Esistono almeno quattro livelli di personalizzazione di SimSpark.

- Il primo è certamente la realizzazione di un server di simulazione specializzato. E' possibile creare server personalizzati a partire dal framework Spark, tuttavia

al momento esiste solo la versione realizzata per il gioco del calcio, chiamata *rcssserver3d*. Dall'analisi del codice sorgente di rcssserver3d è interessante osservare come sia estremamente semplice e compatto, specificato in meno di 200 righe, e come gran parte delle configurazioni sia demandata allo script Ruby *rcssserve3d.rb*. Infatti al fine di eliminare il problema della ricompilazione a seguito di modifiche, anche strutturali, apportate al sistema, SimSpark adotta un meccanismo di specifica della configurazione all'interno di script. Ruby è un linguaggio interpretato e non richiede pertanto compilazione. La potenzialità di utilizzo dei questi script deriva dall'utilizzo di Spark, che consente di creare istanze di oggetti, aggiungere istanze alla gerarchia e richiamare metodi C++ a partire da script. Un analogo meccanismo di script viene utilizzato per la specifica dei modelli di robot utilizzabili nelle simulazioni.

- Il secondo livello di personalizzazione è quindi quello degli script. Di default *rcssserver3d* interpreta all'avvio lo script *rcssserver3d.rb*, ma uno script differente può essere specificato richiamando l'eseguibile con l'opzione *–script-path*. Lo script ne richiama poi altri, dedicati alla specifica di variabili come: i pulsanti della tastiera utilizzati per il movimento della scena (specificati nel sottoscipt *bindings.rb*); i materiali utilizzati nella simulazione (specificati nel sottoscipt *rcs-materials.rb*); eventuali variabili utilizzate poi negli script RSG per la definizione di scena e robot (dimensioni dello scenario, fattore di scala degli elementi visualizzati, ecc).

- Il terzo livello è quello della scena. La scena costituisce l'ambiente in cui si svolge la simulazione. La scena predefinita in SimSpark è quella di un campo da calcio, con due porte, le bandierine agli angoli e una palla a centro campo.

- Infine l'ultimo grado di personalizzazione è dato dalla scelta del robot da visualizzare, specificato come visto nel paragrafo 1.4.1 con il create effector. Come la scena, anche il modello del robot è specificato all'interno di script RSG.

Oggetti, variabili e quant'altro specificato all'interno degli script viene caricato in determinate posizioni della gerarchia gestita da Zeitgeist. Ad esempio i materiali vengono collocati in *kerosin/MaterialSolid*, i comandi da tastiera caricati come membri d'oggetto di */sys/server/input* (l'istanza server dedicata al parsing dell'input da tastiera), ecc. Analogamente anche i modelli dei robot e lo scenario sono caricati all'interno della gerarchia di istanze e possiedono una propria struttura gerarchica, è comune infatti specificare la scena e in particolare i robot sotto forma di albero

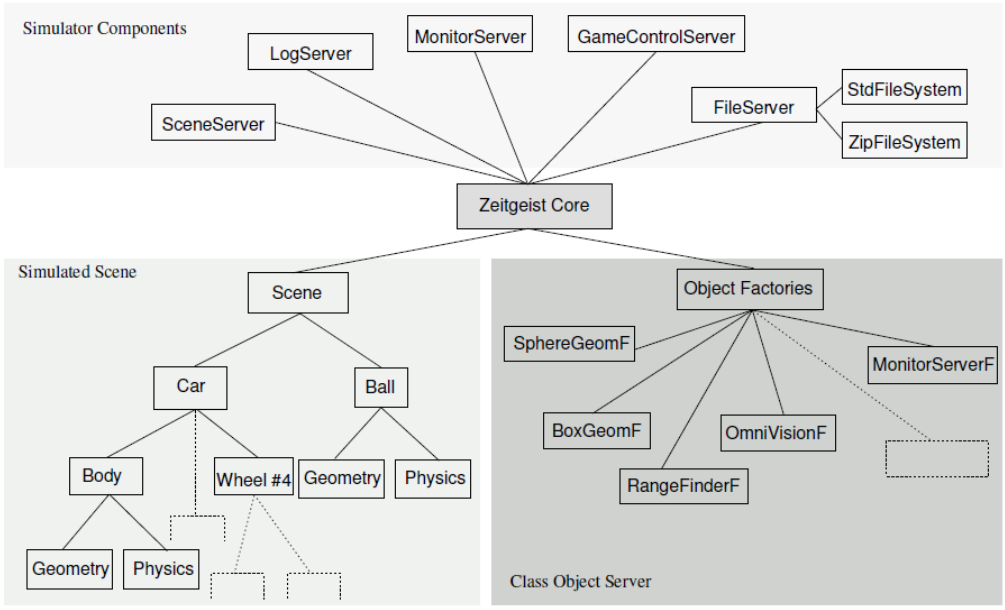


Figura 1.8: Esempio di gerarchia delle istanze Spark

o grafo. Il grafi che specificano scena e modello sono quindi implementati con gli strumenti di Spark e caricati nella gerarchia delle istanze. In figura 1.8 è mostrato un esempio di struttura gerarchica delle istanze gestite da Spark. Il linguaggio Ruby, come linguaggio di scripting ad oggetti, è molto potente per la specifica di configurazioni, ma non adeguato per la realizzazione semplice di una struttura a grafo come quella di descrizione di un robot. Gli sviluppatori di SimSpark hanno pertanto ideato un linguaggio specifico per la definizione della scena e dei robot, il Ruby Scene Graph o RSG.

1.5.1 Ruby Scene Graph - RSG

Il linguaggio è basato su S-Expression (si veda per una definizione il paragrafo 1.4), effettuando una mappatura diretta tra la struttura gerarchica della sexp con la gerarchia di oggetti del grafo di scena. Il linguaggio è stato reso sufficientemente potente da supportare porzioni procedurali specificate in linguaggio Ruby (da cui Ruby Scene Graph). Vi sono alcune tipologie principali di espressioni nel linguaggio RSG, il tipo dell'espressione è specificato dal suo primo elemento terminale. Espressioni che non sono di una delle tipologie principali sono interpretate come chiamate a metodo, infine elementi terminali che iniziano con il simbolo $ sono interpretati come parametri template e sostituiti con il loro valore effettivo. I commenti iniziano con il simbolo del punto e virgola. Le principali espressioni sono:

- *header*: è la prima espressione presente in ogni file RSG, specifica il tipo di linguaggio utilizzato e la sua versione. Il tipo header è indicato con il nome `RubySceneGraph` o `RSG`, al momento l'unico header interpretato è `(RSG 0 1 )`. Segue il corpo del file contenuto in un'unica grande sexp. Il corpo può contenere un'espressione opzionale che che caratterizza il file come template, specificandone i parametri, e un insieme di espressioni di tipo `node`.

- `node`: i nodi del grafo sono dichiarati con l'espressione `(node <ClassName>)` o abbreviato con `nd` dove `<ClassName>` specifica il nome della classe registrata nel class factory system di Zeitgeist. L'interpretazione di un'espressione node comporta la creazione di una nuova istanza di oggetto del tipo specificato, e la sua aggiunta nella gerarchia di Zeitgeist. I nodi sono inseriti nella gerarchia come figli del nodo la cui espressione include la propria. Nodi non appartenenti ad alcuna sopra-espressione sono nodi top level e verranno inseriti come figli del nodo che importa il file RSG a cui appartengono. In questo modo l'espressione e la sua struttura gerarchica rispecchia direttamente la struttura gerarchica che sarà creata in Zeitgeist.

FORMATO	`(node <ClassName>`
	`  ;Qui posso richiamare metodi e impostare attributi`
	`  ;della classe <ClassName>`
	`)`

ESEMPIO	Vedere esempio per `template`

- `importScene - template`: il linguaggio RSG consente il riutilizzo di porzioni di scene graph sotto forma di macro. E' possibile richiamare una macro attraverso l'espressione `(importScene <filename> <parameter>*)` che, in fase di esecuzione, chiama ricorsivamente la procedura di importazione sul file specificato, sostituendone opportunamente i parametri e impostando la scena come figlia del nodo padre attuale. Un file template si caratterizza per avere l'espressione `(template <parameter>*)`, o `templ` in forma abbreviata, specificata all'inizio del corpo del file. Segue un esempio di specifica e richiamo di un file template:

FORMATO	`(template <parameter>*)`

ESEMPIO	File Box.rsg:
	`(RubySceneGraph 0 1)`
	`(`
	`  (template $lenX $lenY $lenZ $material)`
	`  (node Box`
	`  (setExtents $lenX $lenY $lenZ)`
	`  (setMaterial $material)`
	`  )`
	`)`

DESCRIZIONE	File template per la costruzione di un parallelepipedo di dimensioni e materiale personalizzato

FORMATO	`(importScene <filename> <parameter>*)`
ESEMPIO	`(importScene Box.rsg 8 20 5 $gold)`

- espressione `define` o `def`: definisce una variabile che avrà visibilità nel file corrente e in tutti i file importati dal file corrente

FORMATO	`(def $<variableName> <value>)`
ESEMPIO	`(def $radius 25.5)` `(def $PI 3.14159)`

- espressione `eval`: richiama l'interprete Ruby per il calcolo di un'espressione matematica. Il calcolo può coinvolgere costanti o variabili, il risultato dell'operazione viene sostituito all'espressione.

FORMATO	`(eval <expression>)`
ESEMPIO	`(eval 5 * 3.1)` `(def $circum (eval 2 * $PI * $radius))`
DESCRIZIONE	E' possibile utilizzare `eval` all'interno di altre espressioni; il valore calcolato sarà sostituito.

Per quanto riguarda le chiamate a metodo, esse sono espresse come una sexp che specifica come primo elemento terminale il nome della funzione o metodo, seguito da

una lista opzionale di parametri: `(<method name> <parameter>*)`. La chiamata a metodo è risolta nel contesto dell'espressione `node` che la racchiude e determina la chiamata all'interfaccia Ruby della classe C++ corrispondente. Non tutte le classi delle librerie Spark dispongono di un'iterfaccia Ruby. Mancando una documentazione ufficiale a proposito, le descrizioni qui presentate si sono ricavate da un'opera di reverse engineering dei file RSG presenti in SimSpark. I metodi utilizzati sono stati ricercati nella documentazione API autogenerata con Doxygen a seguito della compilazione di SimSpark.

- `Space`: è una classe della libreria `oxygen` che incapsula un oggetto space di ODE. Uno Space ODE è una geometria non posizionabile che può contenere altre geometrie e Space e viene utilizzata per la gestione efficiente delle collisioni. E' da utilizzare come nodo radice per un modello di robot se si vuole usufruire del controllo collisioni.

- `Transform`: classe utilizzata per effettuare trasformazioni geometriche locali, come traslazioni e rotazioni, rispetto il nodo padre. In RSG viene utilizzata per il posizionamento delle geometrie e dei giunti che compongono il modello.

- `StaticMesh`: classe della libreria `kerosin` che incapsula una geometria a mesh e si occupa della sua renderizzazione. Utilizzato in RSG per la specifica degli elementi grafici del modello tramite mesh. Impostando questo nodo come figlio di un noto `Transform` è possibile impostare rotazioni e traslazioni alla mesh.

- `Box, Cylinder, Sphere`: classi della libreria `kerosin` per la costruzione e la renderizzazione di parallelepipedi, cilindri e sfere. Utili per la creazione di un robot o elementi di un robot attraverso primitive anzichè attraverso mesh.

- `BoxCollider, CylinderCollider, SphereCollider`: classe della libreria `oxygen` che incapsula un oggetto Box, Cylinder e Sphere ODE, geometrie dotate di proprietà di collisione. In SimSpark il modello di collisione del robot non può essere specificato tramite mesh, ma deve essere composto attraverso primitive geometriche.

- `RigidBody`: classe che incapsula le proprietà di un corpo rigido.

- `TimePerceptor`: classe che implementa il perceptor del tempo di simulazione

- `AgentState`: classe che raggruppa perceptor dello stato dell'agente (livello della batteria, temperatura interna, messaggi sentiti, ecc).

- `GameStatePerceptor`: usato nella simulazione calcistica per ricevere informazioni sullo stato del gioco.
- `HearPerceptor`: classi che implementa i corrispondenti perceptor nella simulazione calcistica.
- `GyroRatePerceptor, Accelerometer, ForceResistancePerceptor, TouchPerceptor`: classi che implementano i corrispondenti perceptor.
- `SayEffector, BeamEffector`: classi che implementano i corrispondenti effector nella simulazione calcistica.
- `VisionPerceptor, RestrictedVisionPerceptor`: classi che implementano rispettivamente un sensore di visione omnidirezionale o grandangolare con rumore parametrizzabile. Utilizzabile nella simulazione calcistica.
- `ObjectState`: identifica un elemento visualizzabile con il sensore di visione.
- `HingeJoint, UniversalJoint, BallJoint, FixedJoint, Hinge2Joint, SliderJoint`: classi che implementano le corrispondenti tipologie di giunto ODE.
- `HingePerceptor, Hinge2Perceptor, UniversalJointPerceptor`: classi che implementano i corrispondenti perceptor, mancano al momento perceptor per giunti ball e slider.
- `HingeEffector, Hinge2Effector, UniversalJointEffector`: classi che implementano i corrispondenti effector, mancano al momento effector per giunti ball e slider.

Infine è presente un semplice costrutto `switch` che consente la valutazione selettiva di sottoespressioni:

```
(switch $variable
    (<value1>
        ;Espressione da valutare se $variable = <value1>
    )
    (<value2>
        ;Espressione da valutare se $variable = <value2>
    )
    ...
)
```

1.5.2 Modello RSG del robot umanoide Robovie-X

Date le premesse del paragrafo precedente è possibile introdurre la descrizione RSG del modello del Robovie-X realizzata, utilizzabile da ora nella simulazione SimSpark. La descrizione è stato realizzata a partire da quella URDF descritta nel paragrafo 2.8 con i modelli mesh descritti nel capitolo 6. RSG, a differenza di URDF, non consente la specifica di geometrie collisione a mesh, pertanto nel caso si voglia usufruire delle funzionalità di renderizzazione delle mesh per visualizzare un modello più realistico sarà comunque necessario specificare primitive elementari di collisione utilizzando gli elementi messi a disposizione da RSG: parallelepipedi, cilindri e sfere.

Per quanto riguarda le convenzioni geometriche adottate in RSG si riportano le informazioni desunte dal reverse engineering della descrizione del Nao. Mancando un inquadramento globale delle effettive convenzioni adottate a alla loro motivazione, alcune informazioni riportate potrebbero essere non del tutto corrette:

- Il posizionamento delle origini del link è relativo al sistema di riferimento globale dell'intero robot. Pertanto, nota la posizione relativa di un elemento rispetto il suo predecessore nella catena cinematica e la posizione di questo rispetto l'origine, è necessario sommare le due posizioni per ottenere la posizione assoluta del link.

- La posizione dei punti di ancoraggio dei giunti sono relative all'origine non trasformata del link (il sistema di riferimento della mesh).

- Le rotazioni sono euleriane ZYX.

- Nonostante i numerosi tentativi e l'analisi del codice non è stato possibile chiarire quale sia la convenzione adottata per la specifica dell'asse di rotazione dei giunti. Data la semplicità dei giunti del modello, aventi assi di rotazione quasi sempre paralleli ad un asse coordinato, è stato più semplice procedere per tentativi.

La descrizione si sviluppa su più file, quello principale è *robovie_x.rsg* contenente il nodo radice e la specifica del torso del robot. Il file importa le descrizioni delle altre parti del corpo, *head.rsg*, *hand.rsg*, *leg.rsg*, realizzate come template. Le variabili degne di nota dichiarate nel file principale sono:

- `$gravity`: booleana, consente di attivare o disattivare l'azione della forza di gravità sul robot. Utile in fase di debugging del movimento dei giunti.

- `$scale_factor`: reale, consente di impostare un fattore di scala globale del modello. Utile ad esempio per caricare robot con dimensioni differenti nella stessa simulazione o per adattare il modello del robot alla dimensione dell'ambiente.

- `$showCollisions`: booleana, se `true` vengono visualizzate le geometrie utilizzate per la gestione delle collisioni.

Ogni file specifica nella parte iniziale alcune variabili per il dimensionamento del modello, il posizionamento delle geometrie e dei giunti, la massa delle parti:

- `$<name>_radius`: specifica il raggio per componenti sferici. Utilizzato nella descrizione per specificare il raggio della sfera di collisione per testa, spalle e caviglie.

- `$<name>_(DX|DY|DX)`: specificano rispettivamente la dimensione x, y e z per elementi di collisione a parallelepipedo. Le dimensioni sono espresse rispetto il riferimento globale prima di applicare la trasformazione del link.

- `$<name>_mass`: specifica la massa dell'elemento. L'inerzia dell'elemento è calcolata distribuendo la massa uniformemente sulla primitiva geometrica utilizzata per la specifica del link come corpo rigido.

- `$<childLinkName>JOF<parentLinkName>_(X|Y|Z)`: imposta la posizione x, y, z del giunto (JOF = Joint Origin From) posto tra i link indicati, rispetto il sistema di riferimento non trasformato del link padre.

- `$<childLinkName>LOF<parentLinkName>_(X|Y|Z)`: imposta la posizione x, y, z dell'origine (che deve coincidere con il centro di massa) del link (LOF = Link Origin From) figlio rispetto il sistema di riferimento non trasformato del link padre.

- `$<linkName>Name`: specifica il nome da assegnare al link

- `$Joint#PerName`, `$Joint#EffName`: specifica il nome da assegnare al perceptor e all'effector del giunto numero #. Per convenzione in SimSpark i giunti sono nominati secondo una numerazione progressiva lungo la catena cinematica (si veda a proposito la descrizione dell'effector Hinge Joint nel paragrafo 1.4.1).

- `$<jointName>_(min|max)`: limiti inferiori e superiori dell'escursione del giunto.

Per quanto riguarda il modello a mesh SimSpark supporta file in formato OBJ Wavefront. Le specifiche dei materiali utilizzati nel modello devono essere salvate su corrispondenti file MTL Wavefront (Material Template Library). Tutti i più comuni software di modellazione 3D consentono di esportare i modelli in questo formato. Nella directory di rssserver3d sono predisposte due cartelle per il caricamento dei modelli: in ∼`/models`, vanno posti i file OBJ; in ∼`/materials` i corrispondenti file MTL. RSG posiziona il centro delle mesh nell'origine del link a cui le mesh appartengono. Nel medesimo punto si trovano anche le origini delle primitive fisiche e di collisione, pertanto l'origine del link coincide necessariamente con il suo centro di massa. Non è possibile separare questi punti ed è al più possibile spostare l'origine stessa. Pertanto per una corretta visualizzazione del modello, rappresentazione delle proprietà fisiche e di collisione è necessario che le mesh siano centrate nel loro punto mediano o nel centro del proprio bounding box. Sarebbe poi auspicabile che l'ingombro della mesh fosse assimilabile ad una delle geometrie primitive disponibili (parallelepipedo, cilindo o sfera) per avere una distribuzione della massa, inerzia del corpo e gestione delle collisioni coerente con il modello 3D sottostante. Qualora l'articolazione del modello non consenta una condizione del genere può essere opportuno scomporre la parte in porzioni più semplici, connesse tra loro tramite giunti fixed.

1.5.3 Descrizione RSG dello scenario di simulazione

Come precedentemente indicato anche lo scenario 3D di simulazione viene descritto mediante script RSG. In particolare la descrizione viene caricata dallo script Ruby *robivie-x-sim.rb*, a sua volta caricato da *rcssserver3d.rb*. Al momento non è stato creato un ambiente personalizzato per la simulazione del Robovie-X e viene utilizzato lo scenario originale del simulatore, il campo da calcio. Nella descrizione dell'ambiente è comunque possibile impostare le luci, attraverso un nodo `Light`, un piano di appoggio della simulazione, solitamente specificato a partire da file mesh, con primitiva di collisione `PlaneCollider`.

Capitolo 2

ROS

ROS (Robot Operating System) è un *Robotics Software System* (RSS)[12] per lo sviluppo di software per robotica che, operando come meta sistema operativo, fornisce funzionalità di framework e communication middleware. ROS fornisce servizi di astrazione dall'hardware, controllo di device, message passing tra processi, gestione del codice sorgente, gestione delle dipendenze. Offre strumenti per l'integrazione, la compilazione e l'autogenerazione del codice, mentre usufruisce del sistema operativo host per funzionalità di basso livello, come lo scheduling dei processi, l'accesso al disco e la comunicazione di rete. ROS è solo uno di tanti strumenti esistenti a supporto dello sviluppo software in robotica, è relativamente recente ma sta avendo un successo e una diffusione crescente in ambito accademico.

Effettuare un confronto valido tra RSS non è facile ne sempre possibile: diversi sistemi differiscono per framework, communication middleware e utility messe a disposizione. Diversi lavori sono stati effettuati in tal senso [5],[9],[6]. Il confronto che si intende fare in questo paragrafo è molto più semplice e semplicistico, ma rispecchia il fatto che la scelta tra sistemi RSS si riduce spesso al confronto delle loro caratteristiche macroscopiche, in particolare alla quantità e alla qualità di utility fornite e alla sua learning curve. In tabella 2.1 e 2.2 è riepilogato il confronto di ROS con altri diffusi sistemi RSS. A prima vista i diversi framework condividono caratteristiche comuni: architetture distribuite modulari orientate alla comunicazione tra i processi, supporto a più linguaggi di programmazione, supporto molteplici a dispositivi e robot. Per quanto riguarda in particolare le funzionalità di communication middleware si osserva una divisione in tre gruppi:

- comunicazione di basso livello: come Player, basata su messaggistica TCP e UDP

- comunicazioni di alto livello: come OpenRTM, alcune forme di YARP, implementate su CORBA, o MSRDS che addirittura introduce meccanismi tipici dei WebService dotando i nodi di interfacce web vere e proprie
- comunicazione di livello intermedio: utilizzata da ROS, URBI e YARP in altre forme. In particolare ROS fornisce funzionalità di name resolution semplici basate su XMLRPC e trasporto su TCP e UDP adeguatamente astratto.

I meccanismi più semplici vantano naturalmente maggior velocità a scapito di una minor capacità di controllo. Da un'analisi generale delle caratteristiche MSRDS è forse il sistema che presenta funzionalità più innovative: innanzi tutto si basa su un'architettura service-oriented, a differenza degli altri framework, di tipo component-based; fornisce un ambiente di sviluppo integrato e di alto livello, consentendo l'utilizzo di un linguaggio di programmazione visuale (VPL). Un forte punto di debolezza del sistema Microsoft è ancora una volta la sua essenza closed source, seppur in distribuzione gratuita, che in un contesto tipicamente accademico come quello della robotica umanoide è una forte limitazione allo sviluppo apportato dalla comunità. Un altro forte svantaggio è il supporto a soli sistemi operativi Microsoft. ROS da parte sua ha come punti di forza la coordinazione operata da WillowGarage, una compagnia che investe un terzo del suo personale in attività di R&D[S13], l'applicazione del sistema al controllo di robot sofisticati come il PR2, il supporto a framework e librerie esterne per le funzionalità non direttamente fornite da ROS. Un ulteriore punto di forza è la raccolta e l'organizzazione del codice e della documentazione sviluppata attorno a ROS, mantenuta su repository centralizzati, che rende più chiara e uniforme la ricerca di librerie di terze parti, Ultimo, ma non meno importante, aspetto che rende ROS un RSS ad elevata attrazione è la community nata e sviluppatasi attorno, che in pochi anni ha prodotto una grande quantità di strumenti a supporto delle principali funzionalità inerenti la robotica autonoma: driver per sensori e dispositivi di acquisizione, supporto a molteplici modelli di robot, molte tipologie di strumenti e algoritmi a supporto delle problematiche classiche della robotica mobile autonoma: motion planning, navigazione, mapping, ma anche trasformazione di coordinate, image processing ecc. Alla data dello sviluppo di questo testo nel sito web di ROS erano presenti oltre 3000 package di terze parti raggruppati in 400 stack e oltre 200 repository di codice. Numeri che non hanno paragone con alcun altro strumento RSS esistente. Queste sono le motivazioni che hanno spinto alla scelta di ROS come strumento di convergenza per lo sviluppo di applicazioni allo IAS-Lab.

Caratteristiche	ROS	OpenRTM-aist	Player	Yarp
Architettura	Client-Server a supporto di rete peer to peer	Basato su CORBA e modularizzato gestisce solo connessioni punto a punto	Modulare con comunicazione TCP e UDP point to point	Peer to peer con molteplici tecnologie
OS	Ubuntu più supporto sperimentale a Windows e MacOS	Linux, Windows, MacOS	Linux, Windows, MacOS	Linux, Windows più supporto sperimentale a MacOS
Linguaggi	C++, Python, Lisp più supporto sperimentale a Java e Lua	C++, Python, Java	Interfacce TCP e UDP, client library in C, C++, Python, Ruby e supporto sperimentale Java	C, C++ nativo, supporto a molti altri linguaggi tramite SWIG
Simulatori	Stage di base, supporto a Gazebo, Webots	Nessuno di base, supporto a Gazebo, OpenHRP3	Stage, Gazebo	No
Opensource	Si	Si	Si	Si
Real-time	No, supporto a realtime topic, supporto a Orocos RTT	No	No	No
Robot/Driver	Innumerevoli robot e driver di dispositivi	Laser range finder, altro?, non disponibili dal sito	Molteplici robot e driver supportati	Molteplici driver
Moduli	Tantissimi, soprattutto algoritmi	Molteplici, compreso un editor grafico per la configurazione dei moduli e della rete	Pochi strumenti di supporto	Interfacce a dispositivi
Utility	Strumenti per creazione package, repository codice, molteplici tool command line	Strumenti per creazione package, tool grafici e command line per la configurazione di moduli e ambiente	Nessuno	Nessuno

Tabella 2.1: Confronto tra i principali sistemi di middleware

Caratteristiche	Orocos	MS Robotics Developer Studio	URBI
Architettura	Librerie separate, supporta architetture più complesse tramite ROS, CORBA, YARP	Service based	Client-Server
OS	Linux, Windows, MacOS	Windows	Linux, Windows, MacOS
Linguaggi	C++	C#, MSVLP	UrbiScript, C++
Simulatori	No	Proprio	Webots, UBUurbi OBU
Opensource	Si	No	Si
Real-time	Si, attraverso la libreria RTT	?, Gestione della concorrenza tramite CCR	Si
Robot/Driver	Molteplici	Alcuni robot supportati e driver MS, Kinekt, ecc	Diversi robot
Moduli	KDL, RTT, Toolchain, BFL Simulink	Molteplici moduli integrati	Poche decine di moduli
Utility	Strumenti di configurazione simili a quelli usati da ROS	Programmazione visuale con VPL	Linguaggio script per programmazione parallela event-driven

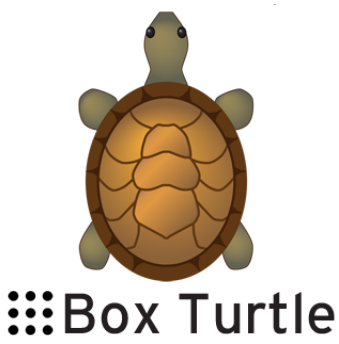
Tabella 2.2: Confronto tra i principali sistemi di middleware

2.1 Breve storia

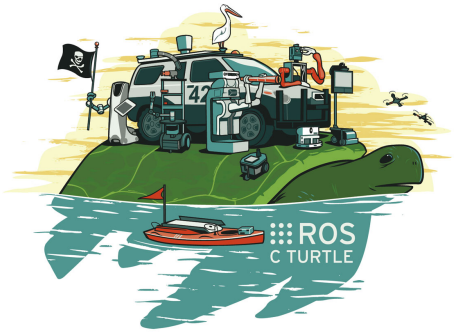
Lo sviluppo di ROS inizia nel 2007, con sotto il nome di *switchyard*, da parte Morgan Quigley e del suo team allo Stanford Artificial Intelligence Laboratory, nell'ambito del progetto *STanford Artificial Inntelligence Robot project* (STAIR [S12]). Dal 2008 lo sviluppo è continuato principalmente attraverso il gruppo Willow Garage [S13] come software per il controllo del robot PR2. Da allora sono state rilasciate nuove versioni di ROS (figura 2.1) con una frequenza di circa due colte l'anno. Ogni nuova release è caratterizzata da un crescente numero di strumenti di sviluppo e moduli sviluppati dalla attivissima community. Il rilascio della prossima release è previsto per Marzo 2012 e introdurrà consistenti novità, anche architetturali, al sistema..

2.2 ROS come component-based framework per robotica

Con Component-Based Software Engineering (CBSE) si intende un approccio alla creazione di sistemi software che mira a riutilizzare il più possibile software

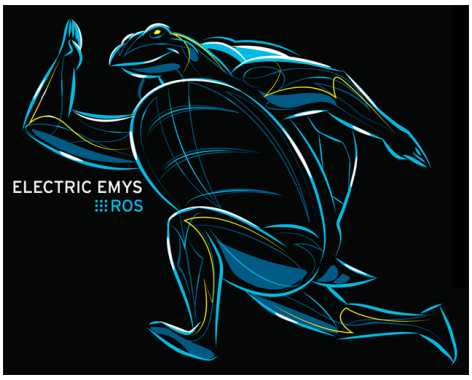


(a) Release 1 - 2 Marzo 2010



(b) Release 2 - 2 Agosto 2010

(c) Release 3 - 2 Marzo 2011



(d) Release 4 - 30 Agosto 2011

Figura 2.1: Versioni release di ROS

già sviluppato, focalizzandosi sul collegamento e la riconfigurazione di componenti preesistenti [7]. L'approccio CBSE spinge verso uno sviluppo del software modulare, facilitando la gestione delle dipendenze, riducendo i costi di mantenimento e incrementando la flessibilità e la robustezza del sistema. Le caratteristiche che un framework component-based per robotica dovrebbe possedere, secondo [7], sono:

1. Dovrebbe essere sviluppato sin dall'inizio con il preciso obiettivo di implementare un approccio component-based

2. Dovrebbe essere open-source, caratteristica fondamentale per la collaborazione tra comunità accademiche, istituti e centri di ricerca

3. Dovrebbe mantenere un repository online di componenti e un efficace meccanismo di documentazione

4. Dovrebbe non essere strettamente legato a particolari meccanismi di comunicazione, come CORBA, ACE, TCP/IP, ecc

5. Non dovrebbe imporre un'architettura predefinita. L'architettura del sistema dovrebbe invece emergere dall'insieme di componenti utilizzati e dal modo in cui sono collegati e composti

6. Dovrebbe favorire l'estensibilità, facilitando la creazione di nuovi componenti

7. Dovrebbe garantire scalabilità del sistema, evitando la dipendenza delle componenti da alcuna entità centrale di controllo.

In questo capitolo si mostrerà come ROS implementi gran parte di queste caratteristiche, dimostrandosi un valido strumento a supporto dello sviluppo software per robotica.

ROS è stato strutturato e sviluppato sin dall'inizio per implementare una architettura component-based, supportando modularità attraverso nodi, librerie e plugin. Il progetto è open-source e fornisce repository online per la condivisione dei componenti ROS. L'organizzazione dei componenti è strutturata in package, raggruppati in stack, raccolti infine in repository (si veda a proposito il paragrafo 2.5). Oltre ai repository, ROS fornisce un efficiente meccanismo di documentazione basato su wiki e supporto tramite ROS Answer (una forma di forum Question&Answer) e Ticket. Gli strumenti della wiki consentono la creazione di documentazioni e tutorial ben strutturati, potendo includere immagini, video e porzioni di codice. Al momento offre due differenti protocolli di trasporto, uno basato su TCP e uno basato su UDP (trattati nel paragrafo 2.4.3), ma non è vincolato ad essi, implementando meccanismi di hanshaking di più alto livello tra i componenti per la definizione del protocollo da utilizzare. In futuro probabilmente saranno aggiunti nuovi livelli di trasporto in relazione a specifiche esigenze. Come descritto nel rossimo paragrafo, ROS non impone alcuna architettura al sistema, lasciandola emergere dal collegamento effettuato tra i nodi ROS. L'estensibilità di ROS è quantomeno dimostrata dal lavoro introdotto nel capitolo 4, che ha portato in breve tempo all'integrazione in ROS di nuove funzionalità di simulazione. Infine i principali aspetti in cui ROS risulta inadeguato, trattati ampliamente nel capitolo 7, sono relativi alla dipendenza del framework dal communication middleware, la presenza di un Master centralizzato per la gestione della rete e il mancato supporto alla computazione e alla comunicazione real-time.

2.3 Architettura ROS - il ROS Computation Graph

Le informazioni per la stesura di questa sezione sono state tratte, ove non diversamente indicato, dal sito web del progetto ROS [S1] e dal paper [4]. Se non diversamente indicato si fa riferimento al client library per il linguaggio C++ (paragrafo 2.6).

L'architettura ROS è basata sul *ROS Computational Graph*, una rete peer to peer di processi intercomunicanti loosely coupled. Gli elementi principali del ROS Computational Graph sono:

1. Il Master
2. I Nodi
3. Il Parameter Server
4. I Bag

Si analizzano ora in dettaglio le funzionalità e i ruoli dei diversi componenti:

- **Il Master**: è il processo fondamentale per l'esecuzione di ROS. Il suo compito è la gestione del naming e delle risorse. Per far ciò fornisce ai nodi servizi di name registration, deregistration e lookup, consentendo loro di individuarsi e contattarsi nella rete, mantiene traccia dei service forniti dai nodi, quindi dei publisher e subscriber per ogni topic della rete. Infine nella versione attuale gestisce internamente il Parameter Server. Il coinvolgimento del master nella comunicazione tra nodi è limitato alla sola funzione di lookup, come una sorta di server DNS, una volta che i nodi si sono localizzati procedono nella loro interazione in maniera peer to peer. Ovviamente i nodi per poter contattare il master devono conoscerne l'URI (host:port) dove resta in ascolto il server XMLRPC del master, questo è specificato nella variabile d'ambiente `ROS_MASTER_URI` del sistema operativo host su cui ROS è installato. Il master espone i propri servizi attraverso API basate sulla specifica XMLRPC[S14], si veda a proposito il paragrafo 2.4.3.3. Il Master viene avviato, assieme al parameter server e al sistema di logging, dall'eseguibile *roscore*, la prima (e unica) applicazione da avviare per rendere operativo il ROS computation graph.

- **I nodi**: i processi in ROS sono nodi interconnessi in un grafo che comunicano attraverso topic, service e parameter server. Un processo diviene un nodo ROS

dopo essersi registrato presso il Master; la registrazione consiste nella comunicazione via XMLRPC del proprio nome e dell'URI (host:port) ove resta in ascolto il proprio server XMLRPC per le comunicazioni in ingresso dal Master e i callback. I nodi costituiscono i *componenti* di ROS come component-based distributed framework, infatti, come si mostrerà in questo capitolo, soddisfano la definizione fornita in [7]: sono unità di sviluppo e costruzione del sistema; implementano interfacce ben definite; forniscono specifiche funzionalità; sono riconfigurabili senza necessità di modifica del codice sorgente . In linea di principio il sistema di controllo di un robot è costituito da molteplici nodi che interagiscono scambiandosi flussi di informazioni o offrendo servizi gli uni agli altri attraverso specifiche interfacce. Ad esempio è comune scomporre le diverse funzionalità di controllo in altrettanti processi: uno dedicato al controllo degli attuatori, uno dedicato alla localizzazione, uno al path planning, ecc. Incapsulare i processi in nodi ROS porta innumerevoli vantaggi: semplifica le modalità di interfacciamento consentendo l'interazione tra moduli scritti in linguaggi differenti, introduce meccanismi di fault tolerance limitando eventuali crash ai singoli nodi e consentendo tramite l'utilizzo degli strumenti ROS il rilancio automatico di nodi crashati.

Un nodo ROS implementa di tre tipologie di API fornitegli dal client library:

- *Slave API*: fornite dal server XMLRPC, creato e gestito all'interno del client library. Consentono al nodo di effettuare comunicazioni con il Master, ricevere callback, negoziare i protocolli di trasporto con gli altri nodi.
- *API dei protocolli di trasporto*: per la comunicazione tra nodi tramite protocolli TCPROS e UDPROS.
- *Command line API*: che forniscono al nodo le funzionalità necessarie ad effettuare il remapping degli argomenti. Questa funzionalità di ROS consente di cambiare qualsiasi nome ROS utilizzato da un nodo (nome del nodo, dei suoi topic, dei parametri che utilizza) attraverso parametri da linea di comando; ciò consente ad esempio di lanciare più istanze del medesimo nodo sotto diverse configurazioni.

Queste API sono comunque mascherate da funzionalità di più alto livello del client library, tali da rendere l'interazione con il Master totalmente trasparente. Si vedano a riguardo gli esempi del paragrafo 2.6.

- **Il parameter server**: il parameter server è un dizionario condiviso e accessibile dai nodi. Nella versione ROS attuale alla data di stesura di questo testo il parameter server è integrato e gestito dal master. Il parameter server può essere utilizzato per memorizzare e recuperare parametri del sistema runtime, rendendoli disponibili a tutto il computation graph, tuttavia non è stato studiato come un sistema ad alte performance. I parametri vengono memorizzati in un namespace gerarchico, la risoluzione dei nomi e il recupero dei valori dei parametri è, anche i questo caso, realizzato tramite funzionalità di alto livello dei client library. I tipi di dato supportati, riportati in tabella 2.3, sono più semplici di quelli per msg e srv.

32-bit integers
booleans
strings
doubles
iso8601 dates
lists
base64-encoded binary data

Tabella 2.3: Tipi di dato supportati dal parameter server

- **I bag**: sono il meccanismo utilizzato da ROS per effettuare il logging delle comunicazioni topic. Attraverso un normale processo di sottoscrizione è possibile associare un bag a uno o più topic. I messaggi scambiati saranno memorizzati automaticamente su file *.bag* in forma serializzata, in questo modo è possibile ad esempio memorizzare e analizzare successivamente i dati prodotti dai sensori del sistema. Gli strumenti principali forniti da ROS per la gestione dei bag sono: *rosbag*, che fornisce un interfaccia unificata da linea di comando per la registrazione, il replay la compressione; *rxbag*, che invece fornisce strumenti grafici per la visualizzazione dei dati memorizzati.

2.4 Comunicazione in ROS

Come evidenziato in [8] la comunicazione loosely coupled riveste un ruolo fondamentale in sistemi component-based distribuiti, poiché consente ai componenti del sistema di cooperare senza generare forme di interdipendenza che limitano la

riutilizzabilità e la riconfigurabilità del sistema. Il grado di disaccoppiamento tra componenti può essere specificato sotto tre punti di vista:

1. *space decoupling*: i componenti del sistema devono poter interagire senza conoscere nulla l'uno dell'altro se non l'interfaccia implementata, non devono sussistere riferimenti statici tra componenti che interagiscono. Ciò permette di ottenere maggior flessibilità del sistema, consentendo la sostituzione run-time di componenti, attuando principi di riutilizzabilità.

2. *time decoupling*: sussiste quando i componenti non devono partecipare obbligatoriamente alle interazioni nel medesimo istante, garantendo maggior robustezza e fault tolerance al sistema,

3. *synchronization decoupling*: consente interazioni asincrone tra componenti.

In ROS la comunicazione tra nodi non implica mai la conoscenza reciproca dei nodi, ma solo del nome del service offerto o del topic utilizzato. La risoluzione dell'URI per attuare la comunicazione effettiva viene offerta dal Master attraverso servizi di name resolution, in maniera totalmente mascherata. In questo modo ROS implementa un elevato grado di space decoupling tra i nodi. Analogamente nei prossimi paragrafi sarà mostrato come i topic forniscano funzionalità di comunicazione asincrona introducendo synchronization e time decoupling in ROS.

Parlare delle funzionalità di comunicazione tra nodi in ROS implica trattare due aspetti: le interfacce di comunicazione di alto livello e la comunicazione di basso livello che ne consentono l'implementazione. Per quanto riguarda la comunicazione di alto livello ROS fornisce tre meccanismi di interazione tra i nodi:

- Comunicazione sincrona RPC-style, effettuata attraverso i *service*

- Streaming asincrono di dati attraverso *topic*

- Memoria condivisa attraverso il salvataggio di dati su *parameter server*

Del parameter server si è già parlato nel precedente paragrafo, esso può essere utilizzato come memoria condivisa per la comunicazione concorrente tra nodi, tuttavia si ricorda che non è implementato come un sistema ad elevata efficienza. Vanno invece approfonditi i concetti di service e topic.

2.4.1 Service

I service costituiscono l'implementazione in ROS di funzionalità RPC e introducono nel computation graph ruoli di nodi service provider e nodi service client.
Un nodo provider può fornire uno o più service identificati con un nome-stringa che deve essere unico nella rete. In particolare il service è vincolato solo al proprio nome e non al nodo provider che lo fornisce, purchè ve ne sia uno solo nella rete con quel nome. I service, come i topic e i parametri salvati nel parameter server, utilizzano un namespace gerarchico.
Un nodo client che voglia usufruire del servizio invierà un messaggio di richiesta, specificando il nome del service richiesto e i parametri di ingresso specifici, attendendo la risposta. Il nodo non saprà chi espleterà la sua richiesta e nemmeno se tale servizio è effettivamente fornito, esistono tuttavia funzioni per verificare la presenza di un service provider nella rete. Oltre al nome ogni service è identificato dalla coppia di messaggi request e response che ne trasportano rispettivamente parametri i ingresso e d'uscita. A partire dalla specifica dei messaggi request e response viene automaticamente generato un digest utilizzato per verificare la corrispondenza tra i messaggi inviati dal client e quelli effettivamente supportati dal provider, lanciando un'eccezione qualora non vi sia corrispondenza.

Per poter realizzare in nodo service provider è innanzi tutto necessario definire la tipologia di messaggi in ingresso e uscita che saranno supportati. E' possibile utilizzare messaggi preesistenti in qualche package ROS, indicandone la dipendenza nel file manifest del package e includendo le i corrispondenti file header, oppure è possibile creare messaggi personalizzati. I messaggi personalizzati da utilizzare per service devono essere specificati all'interno di file *.srv* posti nella cartella *srv* della directory del package. Decommentando nel file *CMakeLists.txt* la funzione `rosbuild_gensrv()` il tool di compilazione *rosmake* si occuperà della conversione dei file in corrispondenti sorgenti C++ o Python a seconda del client library utilizzato. Il nome del file *.srv* sarà anche il nome della classe autogenerata che incapsula i messaggi request e response. Tale classe conterrà la definizione e l'implementazione della struttura dati per i messaggi, dei metodi per l'accesso, per la serializzazione e quant'altro. La sintassi dei file è molto semplice e del tipo in tabella 2.4. I tipi di dato supportati, con le relative implementazioni in C++ e la serializzazione effettuata, sono riportati in tabella 2.5. Oltre al tipo di messaggi scambiati un nodo service provider dovrà specificare e implementare la funzione che si occuperà dell'esecuzione del servizio. In C++ questo viene effettuato impostando un puntatore alla funzione o al metodo servente, che sarà quindi richiamata in callback a seguito di una richiesta

<tipo dato> <nome istanza> ... <tipo dato> <nome istanza> – – –	Struttura request
<tipo dato> <nome istanza> ... <tipo dato> <nome istanza>	Struttura response

Tabella 2.4: Struttura di un file srv

in arrivo. Un esempio di fornitura di service è riportato in appendice B.

2.4.2 Topic

I topic forniscono comunicazione streaming unidirezionale (dai publisher verso i subscriber) agendo come named bus. Attraverso i topic i nodi possono scambiarsi messaggi assumendo il ruolo di publisher - subscriber anonimi. Un utilizzo tipico dei topic è la creazione di pipeline per l'elaborazione progressiva dei dati.

Nodi che generano dati li possono pubblicare come publisher in topic di interesse, nodi che sono interessati invece a ricevere determinate comunicazioni possono sottoscrivere un topic come subcriber. In generale per ogni topic vi possono essere molteplici publisher e subscriber, tanto che un nodo può essere sia publisher che subscriber del medesimo topic. I messaggi scambiati in un topic devono essere del medesimo tipo, pertanto come nel caso dei service, i topic sono identificati dal proprio nome, sempre con namespace gerarchico, e dal tipo di messaggio supportato.

Un nodo che voglia assumere il ruolo di publisher di un topic dovrà specificarne il nome e il tipo di messaggi che intenderà pubblicare oltre alla dimensione del buffer per i messaggi in uscita non ancora inoltrati ai subscriber. ROS si occuperà di registrare il nodo presso il master aggiungendolo alla lista dei publisher di quel topic. Successivamente il nodo potrà pubblicare i propri messaggi sul topic in maniera asincrona. Un nodo che invece intenda assumere il ruolo di subscriber dovrà indicare il nome del topic di interesse, il tipo di messaggio che si aspetta di ricevere e la funzione che si occuperà della gestione dei messaggi ricevuti, richiamata poi in callback a seguito della ricezione di un messaggio. E' compito del master aggiornare i subscriber fornendo la lista dei publisher del topic sottoscritto e il relativo URI. Successivamente il subscriber contatterà direttamente ciascun publisher per stabilire

ROS type	C++ type	Serialization
bool	uint8_t	unsigned 8-bit int
int8	int8_t	signed 8-bit int
uint8	uint8_t	unsigned 8-bit int
int16	int16_t	signed 16-bit int
uint16	uint16_t	unsigned 16-bit int
int32	int32_t	signed 32-bit int
uint32	uint32_t	unsigned 32-bit int
int64	int64_t	signed 64-bit int
uint64	uint64_t	unsigned 64-bit int
float32	float	32-bit IEEE float
float64	double	64-bit IEEE float
string	std::string	ascii string
time	ros::Time	secs/nsecs signed 32-bit ints

Tabella 2.5: Tipi di dato supportati da service e topic nei rispettivi file

un canale di comunicazione. I dettagli del meccanismo di funzionamento di basso livello si rimandano al prossimo paragrafo.

Nella progettazione di un topic è possibile utilizzare messaggi predefiniti in altri package specificandone la dipendenza sul file manifest e includendone il corrispondente header, oppure è possibile creare messaggi personalizzati. I messaggi personalizzati da utilizzare nei topic devono essere specificati in file *.msg* posti nella cartella *msg* della directory del package. Decommentando nel file *CMakeLists.txt* la funzione `rosbuild_genmsg()` il tool di compilazione *rosmake* si occuperà della conversione dei file in corrispondenti sorgenti C++ o Python a seconda del client library utilizzato. Il nome del file *.msg* sarà anche il nome della classe autogenerata che incapsula il messaggio. La sintassi dei file msg è del tutto simile a quella dei file srv, a meno del fatto che non vi è distinzione in request - response, essendo la comunicazione unidirezionale. I tipi di dato supportati sono gli stessi dei messaggi per service, riportati in tabella 2.5. E' possibile utilizzare altri msg come tipo di dato, creando così strutture arbitrariamente complesse.

2.4.3 Comunicazione di basso livello

Dietro la semplice interfaccia della comunicazione tramite service, topic e parameter server si cela un meccanismo comunicativo di basso livello più complesso,

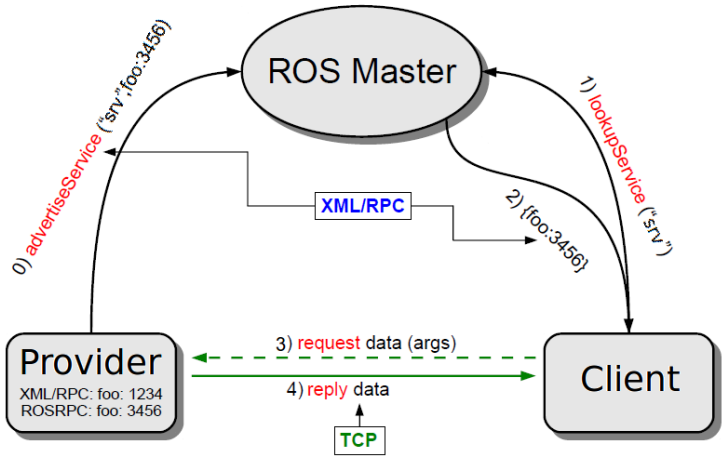


Figura 2.2: Schema di funzionamento di un service

basato sui protocolli XMLRPC, TCPROS e UDPROS.

- **Implementazione di service**: lo schema di funzionamento di un service è riportato in figura 2.2, si articola in alcuni passi e comporta l'uso dei protocolli XMLRPC e TCP:

  0. Il nodo provider richiede a ROS, attraverso l'apposita funzionalità del client library, di fornire il generico servizio *srv*. La richiesta determina la chiamata a procedura remota verso il Master tramite protocollo XMLRPC, con la specifica dell'URI TCP di ascolto del nodo provider e il nome del service fornito.
  1. Un nodo client che voglia usufruire del service si rivolge al Master con una richiesta di lookup del service *srv*, ricevendo come risposta l'URI del nodo provider.
  2. Dopo aver stabilito una connessione TCPROS con il service provider e lo scambio di un Connection Header, il client invia il messaggio reqest in forma serializzata
  3. Nel nodo provider sarà richiamata la funzione di callback a gestione del service e una volta predisposto il messaggio di risposta, questo sarà comunicato al nodo client in forma serializzata tramite lo stesso canale TCP precedentemente predisposto.

  Secondo lo schema presentato il Master viene contattato ad ogni nuova richiesta del service, anche se proveniente da un nodo client che ha effettuato pre-

cedenti richieste. Ciò viene effettuato per garantire fault-tolerance in quanto l'URI del nodo provider potrebbe cambiare. Ciò potrebbe accadere a seguito di un crash del provider stesso o a causa dell'ingresso in ROS di un nuovo nodo provider, che determina la terminazione immediata del vecchio nodo. In presenza di frequenti chiamate ad un service, o in generale quando l'overhead determinato dal processo di lookup è eccessivo, è possibile mantenere una connessione persistente verso il service provider che consenta di inviare richieste multiple sulla stessa connessione. Questa forma è rischiosa poichè non fornisce nessuna garanzia di fault-tolerance.

- **Implementazione di topic**: il meccanismo che sta dietro al funzionamento della comunicazione tramite topic, riportato in figura 2.3, è più complesso.

  0. Il nodo publisher comunica a ROS la sua intenzione di pubblicare messaggi nel topic *bar*. Le funzionalità del client library intraprendono una comunicazione verso il Master via XMLRPC attraverso la quale il nodo si registra come publisher per il topic specifico, indicando l'URI del proprio server XMLRPC.
  1. Il nodo subscriber comunica a ROS la sua intenzione di sottoscrivere il topic *bar*. Le API del client library inviano tale richiesta verso il Master.
  2. Il Master, avendo già almeno un publisher per il topic richiesto, risponde al nodo comunicandogli l'URI del publisher. In caso di successive registrazioni di nuovi publisher, il Master ricontatterà tutti i subscriber comunicandogli l'URI.
  3. Il nodo subscriber contatta il publisher via XMLRPC per richiedere una connessione al topic e specifica la lista dei protocolli di trasporto supportati (ROSTCP o ROSUDP).
  4. Il publisher sceglie il protocollo da utilizzare e lo comunica al subscriber, assieme all'URI da raggiungere per l'eventuale connessione (TCPROS).
  5. Il nodo subscriber effettua, nel caso più comune di utilizzo del protocollo ROSTCP, una connessione verso l'URI del publisher comunicata.
  6. I prossimi messaggi pubblicati da publisher raggiungeranno il subscriber attraverso il canale aperto.

  Si osservi che una volta stabilita la connessione non è più richiesto il coinvolgimento del Master che in nessun momento è coinvolto nel flusso di dati scambiati nel topic.

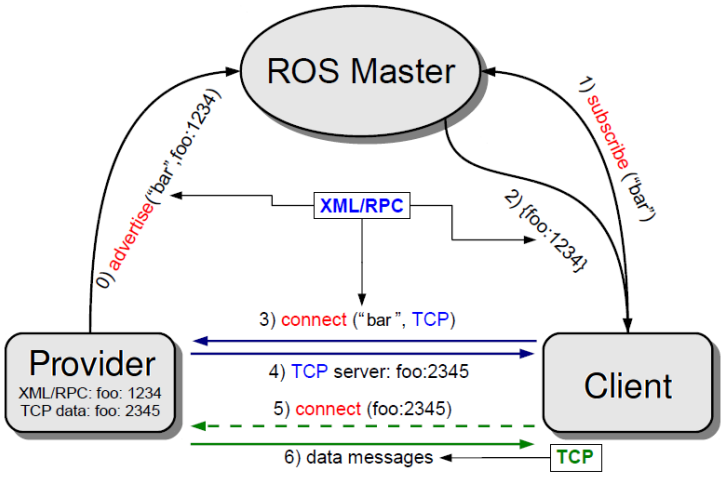


Figura 2.3: Schema di funzionamento di un topic

- Parameter Server: le richieste rivolte al parameter server sono implementate direttamente attraverso chiamate XMLRPC verso il Master.

Di seguito si descrivono i protocolli di basso livello utilizzati nella comunicazione tra nodi e Master.

2.4.3.1 TCPROS

E' il protocollo di gran lunga più utilizzato in ROS. Viene utilizzato per trasmissione di messaggi request e response tra client e service provider ed è uno dei protocolli utilizzabili per le comunicazioni di messaggi topic, assieme a UDPROS. Come si evince dal nome TCPROS utilizza comuni socket TCP/IP come protocollo di trasporto, specificando un header di più alto livello contenente alcuni campi specifici per ROS:

- callerid: nome del nodo che sta inviando i dati
- topic: nome del topic in fase di sottoscrizione
- service: nome del service che si sta chiamando
- md5sum: digest md5 del file msg o srv che definisce il formato dei messaggi scambiati (calcolato sul file originale dopo la rimozione dei commenti, la rimozione degli spazi, riordinamento della struttura del file)
- type: nome del tipo di messaggio o service (vedi file .srv e .msg)

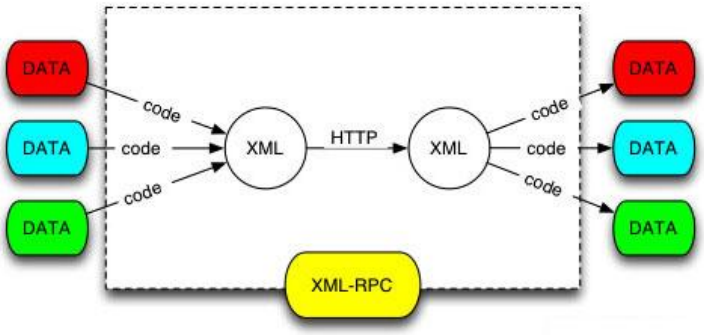


Figura 2.4: Schema di funzionamento di XMLRPC

- persistent: specifica che la connessione per la chiamata a servizio dovrà essere persistente
- tcp_nodelay
- latching

2.4.3.2 UDPROS

UDPROS è basato sul protocollo UDP. L'uso di UDPROS è conveniente quando è da privilegiarsi una comunicazione a bassa latenza, seppur non del tutto affidabile, come lo streaming audio. Un'altra condizione in cui UDPROS può essere conveniente è il presenza di un livello fisico poco affidabile, come connessioni wireless, che rende inefficace TCP. Infine l'utilizzo di UDPROS può essere conveniente in presenza di subscriber multipli al medesimo topic che siano raggruppati nella medesima sottorete, nel qual una comunicazione condivisa via broadcast UDP può risultare più efficiente. Anche UDPROS aggiunge al datagramma UDP un ulteriore header che specifica dettagli della comunicazione ROS.

2.4.3.3 XMLRPC

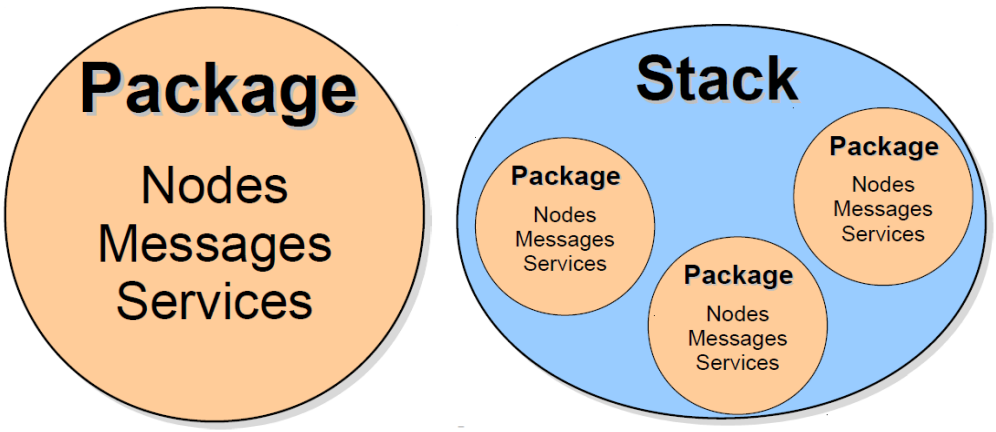
XMLRPC è una specifica, a cui corrispondono molteplici implementazioni, di un protocollo per remote procedure calling che utilizza HTTP protocollo di trasporto e XML come codifica (figura 2.4). I vantaggi apportati da XMLRPC sono la relativa leggerezza rispetto altri protocolli come SOAP, e l'essenza stateless. I client library di ROS implementano proprie versioni del protocollo, ad esempio roscpp utilizza il package `xmlrpcpp`, tuttavia come visto le funzionalità dei client library consentono

di mascherare totalmente l'interazione con il master, nascondendo tutti i meccanismi di comunicazione a basso livello.

2.5 ROS Filesystem

Tutti i componenti di ROS sono organizzati in package e stack:

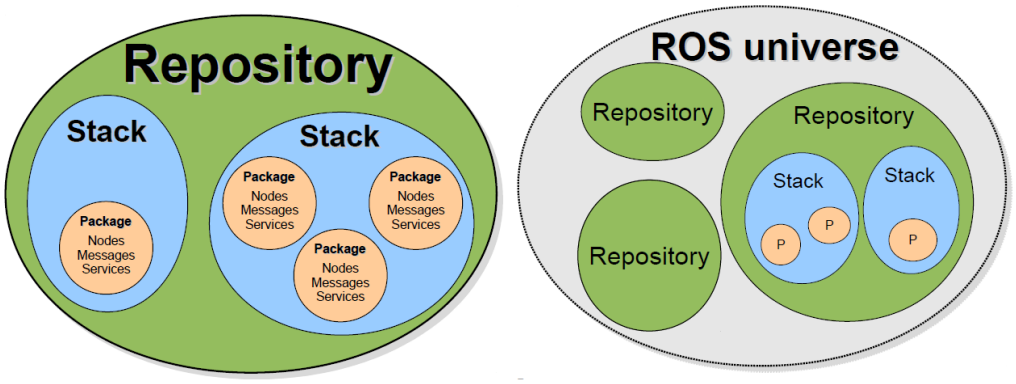
- **Package:** costituiscono il livello più basso dell'organizzazione del software in ROS, possono contenere eseguibili (nodi ROS), librerie ROS, file di configurazione e quant'altro.
- **Stack:** sono collezioni di package che forniscono funzionalità aggregate di alto livello, come lo stack di navigazione, image processing, ecc. Lo stack è anche il mezzo attraverso il quale avviene la ridistribuzione del software ROS.



(a) Package ROS (b) Stack ROS

La descrizione di package e stack avviene attraverso corrispondenti file manifest, in cui è possibile specificate metadati, dipendenze, termini di licenza e flag di compilazione. ROS prevede per stack e package un namespace flat, pertanto, indipendentemente dalla directory di installazione, i nomi di package e stack devono essere unici nel sistema. Gli stack sono infine raccolti all'interno di repository per consentirne la distribuzione. L'insieme dei repository costituisce l'universo ROS.

Tutto in ROS rispetta la struttura stack-package, dalle funzionalità di compilazione e creazione package, appartenenti allo stack *ros*, le API per C++, raggruppate nello stack *roscpp*, gli strumenti di analisi del node graph e delle comunicazioni, appartenenti allo stack *rx*, e persino i moduli utilizzati dal sito internet `www.ros.org`,



(a) Repository ROS (b) Universo ROS

appartenenti allo stack *rosorg*. Infine a questi, ma in realtà a molti altri, si aggiungono il gran numero, sempre in crescita, di stack e package condivisi dalla comunità, caricati nei repository di ROS. Alla data della stesura di questo testo, sul sito `www.ros.org` si potevano trovare più di 400 stack, per un totale di più di 3000 package condivisi. Stack e package non devono essere posti necessariamente nelle directory di sistema di ROS. E' infatti possibile estendere il filesystem specificando directory di lavoro aggiuntive nella variabile d'ambiente ROS_PACKAGE_PATH.

ROS fornisce innumerevoli strumenti da linea di comando per la gestione del filesystem, raggruppati nello stack *ros*:

- package *rosbash*: raccolta di comandi shell utilizzabili con bash.
  - *roscd*: consente la navigazione delle directory degli stack e package di ROS, specificandone solo il nome anzichè il percorso completo.
  - *roscp*: consente di copiare rapidamente file tra package
  - *rosed*: consente di editare rapidamente file a partire dal package a cui appartengono. E' possibile personalizzare l'editor da utilizzare.
  - *rospd*: comando equivalente a pushd per roscd.
  - *rosd*: mostra la lista di percorsi memorizzati con rospd.
  - *rosls*: mostra il contenuto della directory di un package o stack.
  - *rosrun*: consente di lanciare un eseguibile ROS specificandone solo il nome e il package di appartenenza anziché il percorso completo.
- package *roscreate*: raccolta di strumenti per la creazione di package e stack.

  - *roscreate-pkg*: consente la creazione di un nuovo package. Il tool, che consente la specifica delle dipendenze come parametri, crea automaticamente il file manifest per il progetto, le directory in cui porre i sorgenti, i file srv e i file msg, e quanto necessario per la compilazione con rosmake.
  - *roscreate-stack*: consente la creazione del manifest per lo stack nonchè il calcolo di tutte le dipendenze dei package che vi appartengono.

- package *rosdep*: contiene il solo strumento *rosdep* utilizzato per la gestione delle dipendenze in ROS. Il tool infatti consente di verificare le dipendenze associate ad un package, di scaricarle dal repository, di installarle e compilarle automaticamente.

- package *rosmake*: contiene il solo strumento *rosmake* utilizzato per la compilazione dei package con risoluzione delle dipendenze.

- package *rospack*: contiene il solo strumento *rospack* che consente molteplici comandi, dalla ricerca di stack e package nel sistema, alla ricerca di dipendenze, al calcolo dei flag di compilazione, del percorso dei file header e delle preprocessor definition associate ad un package.

- package *roslaunch*: appartenente allo stack `ros_comm`, contiene un unico strumento, `roslaunch`. La funzione di `roslaunch` è quella di semplificare caricamento di ambienti costituiti da molti nodi, consentendo la distribuzione del calcolo su macchine differenti, estendendo le semplici funzionalità del comando *rosrun*. Fornisce inoltre funzionalità aggiuntive come il caricamento automatico del Master e il controllo di esecuzione dei nodi: può ad esempio riavviare automaticamente nodi o terminarli tutti in caso di crash. Il funzionamento di `roslaunch` si basata sull'interpretazione di file di configurazione XML, aventi estensione *.roslaunch*, che specificano: i nodi da lanciare, i parametri da impostare, azioni da compiere in caso di crash, e molto altro. `roslaunch` risulta particolarmente utile per effettuare il remapping dei nomi poiché consente di specificare come modificare nomi e namespace dei nodi caricati. E' possibile parametrizzare le opzioni da linea di comando richiamandole come variabili all'interno del file launch. Per maggiori dettagli si rimanda alla sezione 4.2 dove viene descritto un esempio di launch file.

2.6 Client library roscpp - Principali API per l'utilizzo di ROS in C++

Un ROS client library è un insieme di strumenti e porzioni di codice utili all'implementazione dei concetti di ROS in un dato linguaggio di programmazione. Un client library fornisce pertanto meccanismi per la creazione di nodi ROS, implementazione di service, accesso al parameter server e utilizzo dei topic. Attraverso i client library ROS può supportare molteplici linguaggi di programmazione. Al momento i linguaggi pienamente supportati sono C++ e Python, mentre sono in fase di sviluppo client library per Octave, Lisp, Java e Lua. La filosofia dei client library non è comunque quella di duplicare i package nei diversi linguaggi di programmazione, ma semmai quella di rendere disponibili linguaggi con caratteristiche differenti, per l'uso in contesti differenti: *roscpp*, il client library per C++, è stato predisposto per lo sviluppo di software che richiede alte prestazioni; *rospy* invece mette a disposizione la rapidità di sviluppo e le capacità di introspezione di un linguaggio di scripting object-oriented come python, utilizzabile quindi per l'implementazione rapida di algoritmi in fase di testing, ideale per sviluppare funzionalità di configurazione e inizializzazione (ad esempio il ros Master e gran parte dei tool ROS sono sviluppati con *rospy*); ancora *roslisp* utile per la sua interattività. Il package per l'integrazione del controllo di SimSpark da ROS è stato sviluppato con *roscpp* essendo richieste alte performance per la conversione di formato dei messaggi scambiati tra simulatore e computation graph.

Nelle sezioni a seguire verrà effettuata una panoramica delle funzioni fornite da *roscpp* per l'interazione con ROS.

2.6.1 Inizializzazione e terminazione di un nodo

Come presentato nel paragrafo 2.3, un processo, per poter diventare un nodo ROS, deve innanzi tutto effettuare il remapping degli argomenti da linea di comando, quindi registrarsi presso il ROS Master. La gestione del remapping viene effettuata automaticamente attraverso la funzione:

```
void ros::init( int &argc,
                char **argv,
                const std::string &name,
                uint32_t options = 0
              )
```

che prende come input la lista degli argomenti da linea di comando passata al main, il nome da assegnare al nodo, che potrà essere modificato attraverso il remapping, e alcune opzioni.

La registrazione presso il Master e l'avvio dei server interni per l'interazione con ROS avvengono con la prima creazione di un'istanza della classe `ros::NodeHandle`. E' possibile creare più istanze di `NodeHandle`, tutte faranno riferimento alla medesima configurazione del nodo (la registrazione presso il Master viene infatti effettuata solo alla prima instanziazione, successive instanziazioni incrementano solo il contatore delle istanze attive). A partire da una qualsiasi istanza di `ros::NodeHandle` è possibile accedere alle funzionalità di ROS quali la sottoscrizione di topic, la pubblicazione di messaggi, l'azione sul parameter server, ecc. Al deallocamento dell'ultima istanza attiva di `NodeHandle` il nodo viene automaticamente deregistrato dal Master. E' comunque possibile forzare la deregistrazione attraverso la chiamata alla procedura:

```
void ros::shutdown()
```

2.6.2 Pubblicazione e sottoscrizione di un topic

Roscpp implementa in C++ la ricezione di messaggi da topic attraverso una forma di callback. All'atto della sottoscrizione del topic è richiesta la specifica di una funzione, o metodo di istanza, che si occuperà di "rispondere" alle chiamate del sistema. Quando si verifica la pubblicazione di un nuovo messaggio su un topic di interesse la notifica viene accodata nella coda dei callback del nodo, `ros::CallbackQueue`, incapsulata nel `ros::NodeHandle`. E' possibile anche specificare l'uso di più code, utile ad esempio per non rallentare la coda nel caso di callback particolarmente lenti, o nel caso di utilizzo di thread specifici.

C++ non implementa nativamente funzionalità di callback: il processo dovrà pertanto segnalare quando intende gestire i callback in coda effettuando la chiamata alla funzione `void ros::spinOnce()`. A seguito della chiamata saranno richiamati i callback per tutte le richieste pendenti in coda. Al termine riprenderà la normale esecuzione. `ros::spinOnce()` fornisce pertanto un meccanismo di esecuzione unatantum delle richieste in coda. Per dedicare tutto il tempo del thread alla gestione dei callback è disponibile la funzione `void ros::spin()` che predispone il processo in uno stato di servizio continuativo della coda, dal quale potrà uscire sono a seguito della terminazione del nodo. Poichè ROS supporta il multithreading è soluzione comune avviare un thread di gestione dei callback il quale effettua nella procedura

`run` un'unica chiamata bloccante a `ros::spin()`. E' possibile anche utilizzare più thread di servizio, associati alla stessa coda o a code differenti.

Per pubblicare in un topic è necessario partire dalla definizione dei messaggi che saranno in esso condivisi. E' possibile utilizzare i messaggi definiti in un altro package o definirne di personalizzati, secondo quanto specificato nel paragrafo 2.4.2. In entrambi i casi per poter utilizzare l'oggetto C++ che incapsula i messaggi occorre includere i file header autogenerati da *rosmake* a partire dai file msg.

La pubblicazione in un topic viene effettuata attraverso i metodi della classe `ros::NodeHandle`, in particolare

```
template< class M >
ros::Publisher ros::NodeHandle::advertise(const std::string &topic,
                                          uint32_t queue_size,
                                          bool latch = false
                                         )
```

effettua la registrazione del nodo presso il Master come publisher del topic specificato. Il metodo consente la specifica della dimensione di un buffer per i messaggi in uscita, dove questi verranno memorizzati qualora il nodo pubblichi con una velocità maggiore di quella con cui *roscpp* riesce a distribuire i messaggi. A coda piena i messaggi più vecchi inizieranno ad essere scartati. `advertise` ritorna un'istanza della classe `ros::Publisher` attraverso la quale è possibile effettuare le pubblicazioni tramite il metodo `publish`:

```
template< typename M >
void ros::Publisher::publish(const M &message)
```

Alla distruzione della variabile il nodo viene automaticamente deregistrato dal Master come publisher del topic. Il seguente codice di esempio mostra l'implementazione minimale della pubblicazione nel topic *topic_test* nel nodo *publisher_node_test*:

```
...
ros::init(argc,argv,"publisher_node_test");
ros::NodeHandle nd;
ros::Publisher pub=nd.advertise<pkg::Msg>("topic_test",100);
...
pkg::Msg msg;
//Compilazione del messaggio da pubblicare
...
pub.publish(msg);
...
```

La sottoscrizione ad un topic può essere fatta il almeno due modi. Se si vuole ricevere un solo messaggio o controllare gli istanti di disponibilità alla ricezione di messaggi, con eventuale timeout, è possibile utilizzare la funzione

```
template< class M >
boost::shared_ptr<M const> ros::topic::waitForMessage(const std::string
    &topic)
```

che ritorna un puntatore al messaggio ricevuto. Tra gli overloading della funzione ve ne sono alcuni che permettono la specifica di un intervallo di timeout oltre il quale si ha il ritorno forzato dalla funzione. Per effettuare invece una sottoscrizione permanente è possibile ricorrere ai metodi della classe `ros::NodeHandle`, in particolare

```
template< class M >
ros::Subscriber ros::NodeHandle::subscribe(
        const std::string &topic,
        uint32_t queue_size,
        void(*)(M) fp,
        const TransportHints &transport_hints = TransportHints())
```

effettua la sottoscrizione del topic specificato, indicando la dimensione della coda per i messaggi in ingresso, il puntatore alla funzione per la ricezione in callback dei messaggi e un parametro opzionale per la specifica del tipo di protocollo di trasporto da utilizzare. La funzione di callback deve avere prototipo del tipo

```
void callbackFunction(const pkg::Msg::ConstPtr& message);
```

Alla funzione verrà passato il puntatore condiviso al messaggio ricevuto. Il metodo ritorna un'istanza della classe `ros::Subscriber` alla distruzione della quale il nodo sarà automaticamente deregistrato dal Master come subscriber del topic. Il codice in listato 2.1 mostra l'implementazione minimale della sottoscrizione del topic *topic_test* da parte del nodo *subscriber_node_test*, ove la ricezione dei messaggi viene gestita dalla funzione `testCallback`:

2.6.3 Pubblicazione e richiesta di un service

La fornitura di un service viene implementata con la tecnica del callback, attraverso un meccanismo del tutto simile a quello utilizzato per la sottoscrizione di un topic. Per realizzare un service è necessario partire dalla definizione dei messaggi request e response specificandoli in un file srv, come descritto nel paragrafo 2.4.1, o utilizzando i tipi di messaggi definiti da un altro package. In entrambi i casi per po-

```
bool testCallback(pkg::Msg &msg){
  //Lettura del messaggio ed elaborazione
  ...
}

  ...
  ros::init(argc,argv,"subscriber_node_test");
  ros::NodeHandle nd;
  ros::Subscriber sub=nd.subscribe("topic_test",testCallback);
  ros::Spin();
  ...
```

Listato 2.1: Sottoscrizione C++ di un topic

ter utilizzare l'oggetto C++ che incapsula i messaggi occorre includere i file header autogenerati da *rosmake* a partire dai file srv.

La pubblicazione di un service viene effettuata attraverso i metodi della classe `ros::NodeHandle`, in particolare

```
template< class MReq , class MRes >
ros::ServiceServer ros::NodeHandle::advertiseService(
        const std::string &service,
        bool(*)(MReq &, MRes &) srv_func)
```

effettua la pubblicazione di un service con il nome specificato, indicando che sarà gestito dalla funzione passata come puntatore. La funzione per il callback deve avere un prototipo del tipo

```
bool callback(pkg::MsgReq& request, pkg::MsgRes& response)
```

Il parametro `request` porterà il messaggio request, il parametro `response` conterrà il messaggio response e dovrà essere compilato dalla funzione. Esistono diversi overloading del metodo che consentono anche la specifica di metodi di istanza come callback. In tutti i casi viene ritornata un'istanza di `ros::ServiceServer` che costituirà un handle al service. Richiamando appositi metodi dell'oggetto o semplicemente alla distruzione dell'istanza il service sarà automaticamente deregistrato presso il Master. Per poter abilitare la gestione effettiva del service sarà necessario richiamare le funzioni di spin. Il listato 2.2 mostra l'implementazione minimale della fornitura del service *service_test* nel nodo *provider_node_test*, gestita dalla funzione `testCallback`.

E' possibile effettuare la chiamata ad un service in almeno due modi: direttamente, con la funzione

```
bool testCallback(pkg::ReqMsg &req, pkg::ResMsg &res){
  //Lettura del messaggio request req
  ...
  //Esecuzione del service
  ...
  //Compilazione del messaggio response res
  ...
}

  ...
  ros::init(argc,argv,"provider_node_test");
  ros::NodeHandle nd;
  ros::ServiceServer srv=nd.advertiseService("service_test",
     testCallback);
  ros::Spin();
  ...
```

Listato 2.2: Fornitura di un service

```
template< class MReq , class MRes >
bool ros::service::call(const std::string &service_name,
                        MReq &req,
                        MRes &res
                        )
```

specificando il nome del service, il messaggio request e il messaggio response, e tramite il metodo

```
template< class MReq , class MRes >
ros::ServiceClient ros::NodeHandle::serviceClient(
        const std::string &service_name,
        bool persistent = false,
        const M_string &header_values = M_string())
```

della classe `ros::NodeHandle` che ritorna un'istanza della classe `ros::ServiceClient`. Il metodo consente alche di specificare se instaurare una connessione persistente con il service provider e parametri opzionali da utilizzare durante l'handshake tra nodo client e provider. `ros::ServiceClient` fornisce funzionalità quali la verifica dell'esistenza e attesa di registrazione di un nodo provider per un dato service e ovviamente un metodo `bool call(MReq & req, MRes & res)` per la chiamata a service. Il metodo `ros::ServiceClient::call` e la funzione `ros::service::call` ritornano entrambi un valore booleano che varrà TRUE se la chiamata è stata completata con successo, FALSE in caso contrario. Il codice nel listato seguente

mostra l'implementazione minimale della richiesta del service *service_test* nel nodo *client_node_test*.

```
...
ros::init(argc,argv,"client_node_test");
ros::NodeHandle nd;
ros::ServiceClient srv=nd.serviceClient<pkg::ReqMsg,pkg::ResMsg>("
    service_test");
pkg::ReqMsg req;
pkg::ReqMsg res;
...
//Settaggio dei campi del messaggio request req
...
srv.call(req,res);
//Lettura del messaggio response res
...

//Oppure nella versione piu' compatta
ros::service::call<pkg::ReqMsg,pkg::ResMsg>("service_test",req,res);
```

2.7 Funzionalità di alto livello in ROS

Come già trattato all'inizio del capitolo, ROS cerca di lasciare la maggior libertà possibile nella definizione dell'architettura da implementare nel sistema, attraverso la specifica dei nodi da utilizzare e dalla loro interconnessione. Al crescere della dimensione del sistema da realizzare è però utile disporre di meccanismi che facilitino e strutturino funzionalità di più alto livello. Di seguito si riportano alcune di queste funzionalità offerte da ROS:

- *Trasformazioni geometriche.* Il package `tf` fornisce un framework distribuito basato su ROS per il calcolo di posizioni in sistemi di riferimento multipli che variano nel tempo. `tf` mantiene in particolare le relazioni tra sistemi di riferimento nel tempo, consentendo di trasformare punti, vettori, ecc, tra due qualsiasi sistemi di riferimento in ogni istante temporale. Può quindi essere utilizzato anche per il calcolo della cinematica diretta del robot rispetto un qualsiasi sistema di riferimento in un qualsiasi istante temporale.

- *Visualizzazione.* Lo stack `visualization` fornisce strumenti per la visualizzazione dello stato di robot, dei dati provenienti da sensori e in generale di dati all'interno di ROS. Il package più importante è `rviz`, che fornisce il vero

strumento di visualizzazione 3D. `rviz` può essere utilizzato per visualizzare point cloud, dati da telecamere, mappe, ma anche informazioni sullo stato del robot, come la posizione dei suoi giunti. Può visualizzare la posizione del robot a partire dalla descrizione URDF e dall'attuale trasformazione dei link fornitagli da `tf`. L'interfaccia di Rviz può essere personalizzata con il caricamento di specifici plugin, funzionalità che comunque è in fase di rinnovamento e che sarà ridefinita nella prossima release (ROS Fuerte).

- *Action, task.* Il package `actionlib` fornisce interfacce standardizzate per l'interazione con task preentable in ROS. Le funzionalità offerte sono particolarmente utili nei contesti in cui l'esecuzione di un service può impiegare molto tempo e il client voglia poter cancellare la richiesta effettuata durante la sua esecuzione, o ricevere informazioni sul progresso del service.

- *Message Ontology.* Lo stack `common_msgs` fornisce un ontologia di messaggi base per robotic system, definendo molteplici classi di messaggi come: `actionlib_msgs` messaggi per rappresentare azioni, `diagnostic_msgs` messaggi per l'invio di informazioni di diagnostica, `geometry_msgs` per rappresentare le più comuni primitive geometriche, `nav_msgs` per funzionalità di navigazione, `sensor_msgs` messaggi per rappresentare dati provenienti da sensori, ecc. `common_msgs` consente e promuove quindi l'utilizzo di tipologie di messaggi standard al fine di facilitare l'interoperabilità dei componenti ROS.

- *Plugin* Il package `pluginlib` fornisce strumenti per la scrittura e il caricamento dinamico di plugin (classi C++) utilizzando l'infrastruttura di package build ROS. Il package consente la realizzazione di classi plugin, la descrizione del plugin e la sua registrazione attraverso i normali meccanismi di sviluppo software ROS (file manifest, ecc), quindi fornisce funzionalità di caricamento dinamico dei plugin offerti.

- *Filtri.* Lo stack `filters` è una libreria di funzionalità che consente la realizzazione di meccanismi di processing dei dati sotto forma di sequenze di filtri digitali.

- *Modellazione dei robot* ROS adotta un linguaggio per la descrizione dei robot chiamato URDF, Unified Robot Description Format, fornendo nello stack `robot_model` strumenti per il parsing, la conversione e l'utilizzo di questi file. URDF è un linguaggio basato su XML per la descrizione di un robot. Non tutti i robot possono essere descritti con URDF poichè la capacità espressiva del

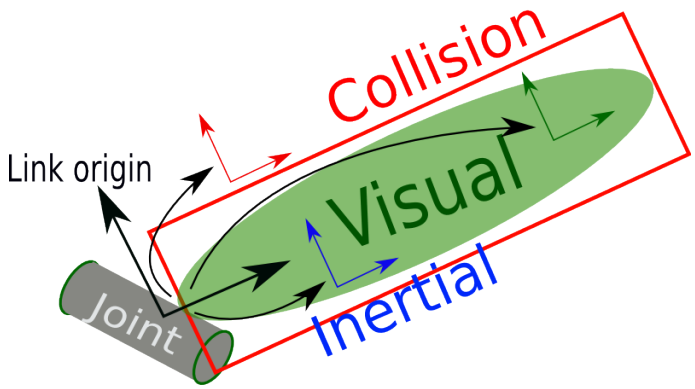


Figura 2.5: Relazioni tra i sistemi di riferimento dell'elemento link

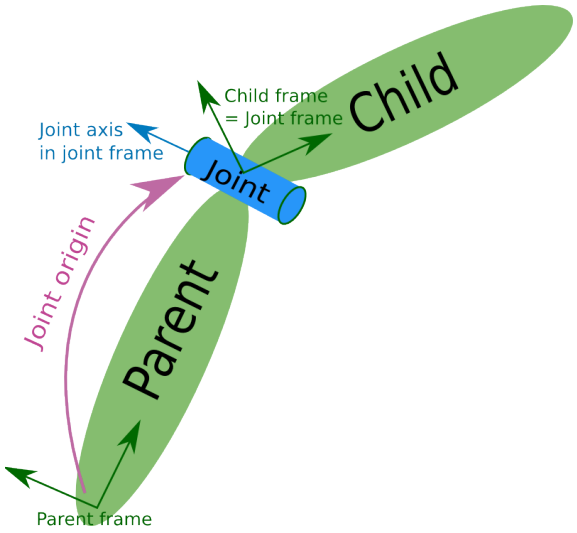


Figura 2.6: Relazioni tra i sistemi di riferimento dell'elemento joint

linguaggio consente la rappresentazione di sole strutture ad albero costituite da link rigidi connessi da giunti. Non è pertanto possibile descrivere elementi flessibili ne robot paralleli.

URDF supporta la descrizione:

- della cinematica e della dinamica del robot
- della rappresentazione visuale del robot
- del collision model del robot

riconducendola ad una lista di elementi link, le parti rigide del robot, e joint, i giunti che connettono i link. URDF non dispone quindi di molte tipologie di tag:

`<robot>` radice del formato URDF, specifica come unico attributo **name**, il nome del robot.

`<link>` elemento per la descrizione di un corpo rigido. Possiede come unico attributo **name**, il nome del link, ed elementi per la specifica delle proprietà visuali, di inerzia e di collisione del link. Visualizzazione, inerzia e collisione sono implementati come elementi dotati di un proprio sistema di riferimento, che può coincidere con quello del link o essere diverso, come da figura 2.5. Pertanto è consentita la specifica di un offset e di una rotazione del sistema di riferimento dell'elemento rispetto il sistema di riferimento del link:

`<visual>` definisce le proprietà visuali del link, quali la forma, il materiale, ecc.

`<origin>` (opzionale: valori nulli se non specificato) rappresenta la trasformazione del sistema di riferimento dell'elemento di visualizzazione rispetto quello del link. Gli attributi che supporta sono:

xyz offset x,y e z dell'origine

rpy angoli in radianti di roll, pitch e yaw

`<geometry>` (obbligatorio) descrive la forma dell'elemento visuale che può essere

`<box>` parallelepipedo con l'origine nel suo centro e dimensioni specificate nell'attributo **size**

`<cylinder>` cilindro con l'origine nel suo centro e dimensioni specificate dagli attributi **radius** e **length**

`<sphere>` sfera con l'origine nel suo centro e dimensione specificata dall'attributo **radius**

`<mesh>` elemento mesh contenuto nel file specificato dall'attributo **filename**, eventualmente scalato con l'attributo **scale** attorno al suo bounding box. Sono supportati i principali formati di modello 3D: 3ds, obj, stl, ecc.

`<material>` (opzionale) descrive il materiale da utilizzare per la visualizzazione. E' possibile richiamare un materiale definito al livello superiore indicandone il nome nell'attributo **name**, oppure è possibile definirlo attraverso gli elementi

`<color>` l'attributo **rgba** consente la specifica del colore del materiale come vettore rgba con valori nel range [0,1]

`<texture>` texture del materiale specificata dal file **filename**

`<collision>` (opzionale) descrive le proprietà di collisione del link. E' possibile impostare proprietà di collisione uguali a quelle dell'elemento visual, tuttavia per ragioni di efficienza a volte si usa utilizzare un modello di collisione più semplice, basato su primitive geometriche.

`<origin>` (opzionale: valori nulli se non specificato) rappresenta la trasformazione del sistema di riferimento dell'elemento di collisione rispetto quello del link. Gli attributi che supporta sono gli stessi del corrispondente elemento visual.

`<geometry>` (obbligatorio) descrive il modello geometrico da utilizzare per il controllo collisioni. La struttura dell'elemento è la stessa del corrispondente elemento visual.

`<inertial>` (opzionale) descrive le proprietà di inerzia del link.

`<origin>` (opzionale: valori nulli se non specificato) rappresenta la trasformazione del sistema di riferimento dell'elemento di inerzia rispetto quello del link. Gli attributi che supporta sono gli stessi del corrispondente elemento visual.

`<mass>` massa del link specificata dall'attributo **value**

`<inertia>` la matrice di inerzia rispetto il sistema di riferimento dell'elemento di inerzia. Essendo simmetrica vengono specificati solo sei elementi attraverso gli attributi **ixx**, **ixy**, **ixz**, **iyy**, **iyz** e **izz**

`<joint>` descrive la cinematica, la dinamica e i limiti del giunto. Un giunto connette il parent link con il child link, come da figura 2.6, aggiungendo un elemento all'albero cinematico. Gli attributi di un giunto sono il nome **name** e il tipo **type**, entrambi obbligatori. I tipi di giunto supportati sono

- **revolute**: giunto a cerniera (hinge) che ruota attorno a suoi assi in un range limitato
- **continuous**: giunto che può ruotare senza limiti attorno ai suoi assi

- **prismatic**: giunto traslazionale chè può traslare lungo gli assi in un range limitato
- **fixed**: giunto bloccato in tutti i gradi di liberà
- **floating**: giunto libero in tutti i suoi gradi di libertà
- **planar**: giunto che vincola il movimento in un piano.

Gli elementi di joint sono:

`<origin>` (opzionale: valori nulli se non specificato) rappresenta la trasformazione dal parent link al child link. Il giunto è posizionato nell'origine del child link. Gli elementi sono gli stessi precedentemente presentati.

`<parent>` (obbligatorio) l'attributo **name** rappresenta il nome del parent link

`<child>` (obbligatorio) l'attributo **name** rappresenta il nome del child link

`<axis>` (opzionale: (1,0,0) se non specificato) è l'asse del giunto rispetto il sistema di riferimento del giunto. Asse di rotazione per giunti rotazionali, di traslazione per giunti prismatic, la normale del piano di scivolamento per giunti planar. L'attributo **xyz** rappresenta le componenti del vettore normalizzato.

`<dynamics>` (opzionale) specifica le proprietà fisiche del giunto, utile per la simulazione. Gli attributi sono

damping fattore di smorzamento del giunto, in $\left[\frac{N \cdot s}{m}\right]$ per giunti traslazionali, $\left[\frac{N \cdot m \cdot s}{rad}\right]$ per giunti rotazionali

friction fattore di attrito statico del giunto, in $[N]$ per giunti traslazionali, $[N \cdot m]$ per giunti rotazionali

`<limit>` (obbligatorio solo per giunti revolute e prismatic) possiede i seguenti attributi

upper (opzionale, 0 di default) limite superiore del giunto in metri o radianti a seconda del tipo di giunto

lower (opzionale, 0 di default) limite inferiore del giunto

effort (obbligatorio) massimo sforzo applicabile al giunto

velocity (obbligatorio) massima velocità del giunto

Infine sono state create alcune estensioni a URDF utilizzate da applicazioni specifiche, come l'elemento `<gazebo>`, utilizzato dal simulatore gazebo. Il

formato di descrizione URDF viene utilizzato da molti package ROS tra cui `robot_state_publisher` per ottenere alberi cinematici per `TF` a partire dallo stato dei giunti, per la simulazione con `simulator_gazebo`, per la visualizzazione dello stato del robot su `rviz`, per la creazione degli alberi cinematici utilizzati dalla libreria Orocos KDL, ecc.

Il formato URDF è semplice ma prolisso. In particolare determinate porzioni della descrizione sono ripetute più volte, uguali o con minime variazioni: basti pensare alle parti ripetute di un robot o simmetriche come gli arti. Per questo è stato sviluppato il linguaggio XACRO (Xml mACRO) per URDF, che consente:

Inclusione di un altro file XACRO

Dichiarazione di proprietà associate ad un nome, sia di tipo numerico che letterale. La dichiarazione viene effettuata attraverso il costrutto `property`, mentre l'espansione indicando il nome della property all'interno di `${}`:

```
<!-- Dichiarazione !-->
<xacro:property name="<nome della property>" value="<valore
    associato>" />
<!-- Esempio di dichiarazione!-->
<xacro:property name="pi" value="3.141592654" />
<xacro:property name="mesh_format" value="3ds" />

<!-- Espansione !-->
"${<nome della property>}"
<!-- Esempio di espansione !-->
<mesh filename="${mesh_path}${mesh_file_name}.${mesh_format
    }"/>
```

Dichiarazione di property block attraverso il medesimo costrutto `property`. Consente la definizione di una porzione di codice XML da espandere dove necessario attraverso il costrutto `insert_block`:

```
<!-- Dichiarazione !-->
<xacro:property name="<nome del property block>">
  <!-- Blocco XML del property block !-->
</xacro:property>
<!-- Esempio di dichiarazione!-->
<xacro:property name="inerzia">
  <inertia ixx="1" ixy="0.0" ixz="0.0" iyy="1" iyz="0.0"
      izz="1"/>
```

```
</xacro:property>

<!-- Espansione !-->
<xacro:insert_block name="<nome del property block>" />
<!-- Esempio di espansione !-->
<xacro:insert_block name="inerzia_1" />
```

Espressioni matematiche all'interno di `${}`, è possibile anche utilizzare `property` come operandi:

```
<!-- Esempio !-->
<xacro:property name="pi" value="3.1415926535897931" />
<circle circumference="${2.5 * pi}" />
```

Richiamo di comandi *rospack* all'interno di `$()` es:

```
<include filename="$(find robovie_x_model)/xacro/torso.
    xacro" />
```

utilizzato per ricercare il path del package `robovie_x_model` e creare il percorso assoluto al file torso.xacro.

Dichiarazione di macro attraverso il tag `macro`, che possiede gli attributi **name**, il nome della macro, e **params**, la lista dei parametri. L'utilizzo dei parametri si effettua allo stesso modo delle property, all'interno di `${}`. Una macro è composta da costrutti URDF ed eventualmente da altre macro. Una macro viene richiamata attraverso il costrutto `<xacro:nome_macro parametro1="...">`

```
<!-- Dichiarazione !-->
<xacro:macro name="<nome macro>" params="<nome parametro1
    > <nome parametro2> ...">
   <!-- Corpo della macro, utilizzo dei parametri !-->
</xacro:macro>

<!-- Esempio, macro per il calcolo dell'elemento inertia
    di un parallelepipedo !-->
<xacro:macro name="box_inertia" params="dx dy dz mass">
    <inertia
      ixx="${(1/12)*mass*(dy*dy +dz*dz)}"
      ixy="0.0"
      ixz="0.0"
      iyy="${(1/12)*mass*(dx*dx +dz*dz)}"
      iyz="0.0"
      izz="${(1/12)*mass*(dx*dx +dy*dy)}"
```

```
    />
</xacro:macro>

<!-- Espansione per un cubo di spigolo 5 e massa 1 !-->
<xacro:box_inertia dx="5" dy="5" dz="5" mass="1"/>
```

Una volta realizzata la descrizione xacro del modello è necessario convertirla nel formato URDF. Per farlo è sufficiente ricorrere allo strumento *xacro.py* del package *xacro*, con il comando:

```
$rosrun xacro xacro.py model.xacro > model.urdf
```

2.8 Modello XACRO-URDF del robot umanoide Robovie-X

Per lo sviluppo del modello URDF del robot Robovie-X si è fatto ampio uso dei costrutti del linguaggio XACRO, cogliendo l'occasione per sviluppare macro riutilizzabili in futuro per lo sviluppo di altri modelli. La descrizione è organizzata in più file. Il file principale è *robovie_x.xacro* dal quale viene effettuato il caricamento degli altri file e l'espansione delle macro:

- **Proprietà del modello e definizioni macro globali**

  property.xacro vi si trova la dichiarazione di costanti (come PI) e variabili globali del modello: il fattore di scala, il formato e il percorso relativo al package dei file mesh, il nome dei giunti e dei link, le dimensioni delle primitive utilizzate per gli elementi collision dei link (secondo la medesima convenzione utilizzata nella descrizione RSG 1.5.1), la massa dei link, i limiti dei giunti, gli offset dei sistemi di riferimento degli elementi visual, inertial e collision rispetto il riferimento del link (LOF=Link Offset From...), gli offset dei giunti rispetto il riferimento del link (JOF = Joint Offset From...).

  macro.xacro vi si trovano le dichiarazioni delle macro utilizzate per la creazione di elementi link e joint. Le macro sviluppate sono:

  1. Creazione della matrice di inerzia per sfera e parallelepipedo: macro

     ```
     <xacro:macro name="sphere_inertia" params="radius
         mass">
     ```

e

```
<xacro:macro name="box_inertia" params="dx dy dz
    mass">
```

che accettano come paramerti le dimensioni della geometria e la massa, effettuando il calcolo di un elemento `<inertia>`.

2. Creazione di un elemento link a mesh con primitiva di collisione sferica: macro

```
<xacro:macro name="make_sphere_link" params="
    link_name mesh_file_name link_origin_xyz
    link_origin_rpy link_radius link_mass">
```

che prende come parametri il nome del link da creare, il nome del file (senza percorso ed estensione, definite globalmente) del modello a mesh, l'offset e la rotazione dei sistemi di riferimento degli elementi visual, inertial e collision del link, quindi il raggio della primitiva sferica e la massa del link. L'espansione della macro crea un blocco link completo delle componenti visual, inertial e collision.

3. Creazione di un elemento link a mesh con primitiva di collisione a parallelepipedo: macro

```
<xacro:macro name="make_box_link" params="link_name
    mesh_file_name link_origin_xyz link_origin_rpy
    link_dx link_dy link_dz link_mass">
```

che prende i medesimi parametri della macro precedente a meno delle dimensioni della geometria per l'elemento collision, che questa volta è un parallelepipedo. L'espansione della macro crea un blocco link completo delle componenti visual, inertial e collision.

4. Creazione di un elemento joint: macro

```
<xacro:macro name="make_joint"params=
            "joint_name joint_type joint_axis_xyz
             joint_limit_effort joint_limit_zero
                 joint_limit_lower
                 joint_limit_upper
                 joint_limit_velocity
             joint_origin_xyz
             joint_parent joint_child"
>
```

che prende come parametri, nell'ordine, il nome e il tipo del giunto da creare, il versore che definisce l'asse di rotazione, i limiti del giunto

secondo quanto precedentemente specificato, la posizione del giunto rispetto l'origine del link padre, il nome del link padre e figlio tra i quali è posto il link.

- **Definizioni macro del torso**

  torso.xacro utilizza le macro generiche per definire il torso del robot

- **Definizioni macro della testa**

  head.xacro utilizza le macro generiche per definire la testa del robot

- **Definizioni macro del braccio** definiscono macro per le corrispondenti parti del braccio, quindi specificano parametri per l'indicazione del braccio da realizzare, se quello destro o quello sinistro.

  shoulder.xacro

  arm.xacro

  hand.xacro

- **Definizioni macro della gamba** definiscono macro per le corrispondenti parti della gamba, quindi specificano parametri per l'indicazione della gamba da realizzare, se quello destra o quella sinistra.

  hip.xacro

  leg.xacro

  knee.xacro

  ankle.xacro

  foot.xacro

Una volta realizzate le macro per la descrizione delle singole parti del robot in file separati, nel file *robovie-x.xacro* vengono inclusi tali file e vengono espanse le macro, specificando componenti per la parte destra e sinistra del corpo.

Utilizzando lo strumento `urdf_to_graphiz` del package `urdf_parser` è possibile ottenere una rappresentazione pdf dell'albero cinematico del robot. L'albero ottenuto per il modello del Robovie-X è riportato in figura 2.7. I link del modello sono rappresentati tramite rettangoli, mentri i joint tramite cerchi.

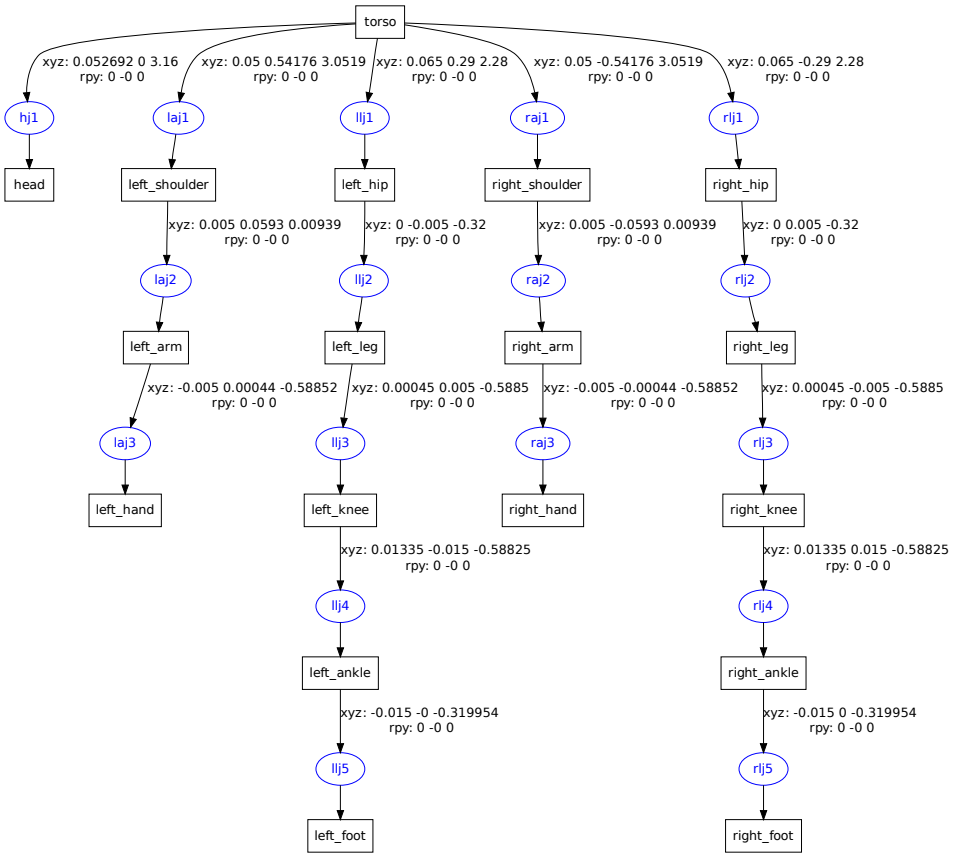


Figura 2.7: Albero cinematico del modello Robovie-X in URDF

Capitolo 3

Installazione di ROS e SimSpark, configurazione dell'ambiente

Terminate le premesse dei capitoli precedenti, prima di iniziare la presentazione del lavoro svolto, è necessario illustrare brevemente i passi di installazione e le configurazioni necessarie all'utilizzo dell'ambiente. Il lavoro è stato sviluppato sotto la piattaforma Linux Ubuntu 10.04.

3.1 Installazione di ROS - Creazione del package

Lo sviluppo del progetto è iniziato utilizzando la versione Diamondback di ROS, poi dopo il rilascio della nuova versione ad agosto 2011, si è passati alla versione Electric Emys. Il cambio di versione non ha comportato problemi di compatibilità con il lavoro già sviluppato, tuttavia è da riportare che ROS è un framework ancora in fase di forte sviluppo e soggetto a frequenti modifiche, soprattutto per quanto riguarda l'organizzazione e la distribuzione dei package. La prossima release, il cui rilascio è previsto per marzo 2011, introdurrà notevoli cambiamenti anche alla struttura interna del sistema. L'installazione di ROS in Ubuntu è molto semplice se effettuata attraverso i repository. Sul sito del progetto ROS [S15] è possibile trovare indicazioni sui passi da effettuare per l'installazione. Non è stata effettuata nessuna particolare configurazione rispetto la procedura standard.

Una volta terminata l'installazione, è conveniente definire uno spazio di lavoro personale, directory che può essere inclusa al filesystem di ROS aggiungendone il percorso alla variabile d'ambiente `ROS_PACKAGE_PATH`. Al fine di rendere permanenti le modifiche alle variabili d'ambiente è conveniente aggiungere i comandi al file *.bashrc*:

```
#definisce le variabili d'abiente utilizzate da ROS
source /opt/ros/electric/setup.bash
#aggiuge una directory personalizzata al filesystem di ROS
export ROS_PACKAGE_PATH=~/your_workspace:$ROS_PACKAGE_PATH
```

Entrati nello spazio di lavoro è possibile effettuare la creazione del package attraverso lo strumento roscreate-pkg, presentato nel paragrafo 2.5. Il comando, oltre che il nome del package da creare, accetta come parametri la lista delle dipendenze che sono previste per il package: *roscpp* è la dipendenza standard per lo sviluppo del codice in C++; *std_srvs* e *std_msgs* sono i package per i parametri principali per i messaggi di service e topic. A seguito della chiamata:

```
$ roscreate-pkg simulator_simspark roscpp std_srvs std_msgs
```

ROS crea una sottocartella nella directory di lavoro, assegnandogli il nome del package, e al suo interno crea in automatico il file manifest, il file di configurazione CMakeLists.txt per la compilazione con *rosmake* e un certo numero di directory preimpostate ove porre i sorgenti, i file msg e srv. Da qui si procede sviluppando i sorgenti all'interno della cartella src, configurando di conseguenza il file CMakeLists.txt, eseguendo il build attraverso il comando ROS *rosmake*. *rosmake* effettua anche la compilazione dei file msg e srv generando i corrispondenti sorgenti C++. Qualora si disponga della directory di un package non distribuita tramite repository, è sufficiente porla all'interno del proprio spazio di lavoro, ROS riconoscerà automaticamente l'aggiunta del nuovo package al suo primo utilizzo.

3.2 Installazione di SimSpark

Anche in questo caso l'utilizzo del repository semplifica l'installazione in Ubuntu. La guida per l'installazione di SimSpark può essere trovata nel sito web del progetto [S16]. Non sono richieste particolari configurazioni per l'utilizzo del simulatore. I comandi per l'avvio del server e del monitor sono rispettivamente *rcssserver3d* e *rcssmonitor3d*. Al momento SimSpark non consente la specifica di modelli di robot posti in directory diverse da quelle di sistema preposte, in oltre per personalizzare l'ambiente di simulazione può essere necessario modificare gli script Ruby di rcssserver3d. Per questo motivo può essere più conveniente effettuare la compilazione manuale dei sorgenti e l'installazione in una directory della propria home. In questo secondo caso bisogna installare come prerequisiti: CMake versione 2.6 o successiva, Ruby 1.8 o successiva, Boost C++ Library, Open Dynamics Engine, FreeType,

Developer Image Library (DevIL), OpenGL, SDL. Per eseguire la compilazione è sufficiente:

1. Entrare nella directory del progetto da compilare
2. Creare la cartella Build `$mkdir build` ed entrarvi
3. Eseguire `$cmake ..` per configurare il package, verificare le dipendenze e generare i Makefile
4. Se si desidera modificare le impostazioni del build, ad esempio la directory di installazione, eseguire `$ccmake .`
5. Eseguire `$make` per effettuare il build
6. Eseguire `$make install` effettuare l'installazione
7. Eseguire `$make uninstall` effettuare la disinstallazione
8. Con i comandi `$make doc` e `$make pdf` è possibile generare la documentazione Doxygen e la guida PDF aggiornata.

Per poter utilizzare il modello del Robovie-X è necessario copiare alcuni file nelle directory di sistema:

- copiare i file rsg della descrizione RSG del robot nella directory `/rsg/agent/robovie_x/`
- copiare i file obj del modello 3d mesh del robot nella directory `/models`
- copiare i file mtl dei materiali del modello 3d nella directory `/materials`

Infine l'unica modifica necessaria da apportare agli script Ruby è relativa alla posizione Z in cui vengono inizialmente caricati i robot nella simulazione, associata alla variabile AgentRadius nel file naosoccersim.rb. In futuro probabilmente si distribuirà una versione del server SimSpark direttamente all'interno di uno stack `simulator_simspark`, per semplificare la procedura di compilazione e installazione.

Capitolo 4

Il package simulator_simspark

`simulator_simspark` è il package sviluppato in ambiente ROS per l'integrazione delle funzionalità di simulazione di SimSpark. Il nome si è scelto in aderenza al nome di altri wrapper presenti in ROS come simulator_gazebo, che integra la simulazione di Gazebo. Lo spirito del progetto è stato sin dall'inizio quello di sfruttare al massimo le funzionalità di SimSpark rendendole conformi alle interfacce standard di ROS. Le caratteristiche che si sono volute implementare quindi sono:

1. accessibilità da ROS: lo scopo del lavoro
2. supporto multiagente: per sfruttare le potenzialità del simulatore
3. controllo degli effector attraverso interfacce ROS: per uniformarsi al linguaggio ROS
4. ricezione dello stato dei perceptor attraverso interfacce ROS: per uniformarsi al linguaggio ROS e consentire l'uso dei suoi strumenti
5. indipendenza del wrapper dal modello di robot simulato: per consentirne l'utilizzo con robot differenti
6. possibilità di aggiungere nuovi effector e perceptor
7. possibilità di utilizzare i package ROS a supporto del controllo del robot

Sulla base di queste linee guida è stato sviluppato `simpark_virtual_driver`, un nodo che agisce come un wrapper per la simulazione SimSpark ponendosi nel mezzo tra gli ambienti ROS e Spark. Infine sono stati creati e alcuni file launch per la gestione semplificata del caricamento dell'ambiente.

4.1 Il nodo simspark_virtual_driver

Il nodo `simspark_virtual_driver` agisce come wrapper tra le interfacce ROS e la comunicazione TCP/IP verso SimSpark. Per quanto riguarda l'interfaccia ROS è stato necessario create nuovi tipi di strutture msg e srv poichè quelle esistenti non erano complete o eccessivamente dettagliate. Si è quindi stabilito un collegamento uno a uno tra effector e perceptor SimSpark con service e topic ROS, mappando ogni effector in un service e ogni perceptor in un topic. Per quanto riguarda invece l'interfacciamento con il simulatore si è scelto di effettuarlo attraverso connessione TCP/IP. Inizialmente la ragione di questa scelta è stata la possibilità di utilizzare un simulatore in esecuzione su una macchina differente da quella da cui veniva effettuato il controllo, soluzione adottata oltretutto nell'utilizzo di SimSpark nella SSL di RoboCup. Indubbiamente l'utilizzo dell'interfaccia di rete introduce passi di conversione dei messaggi aggiuntivi rispetto all'utilizzo diretto delle API di SimSpark. Una soluzione differente poteva essere quella di installare ROS anche sulla macchina dove il simulatore è in esecuzione, effettuare l'interfacciamento a SimSpark tramite API e lasciare la comunicazione via rete agli automatismi di ROS. Ricordando però l'esigenza di mantenere un unico ROS Master attivo nella rete, questa soluzione sembrava inadeguata alle esigenze di una simulazione multi agente come quella usata nella SSL: decine di robot con decine di nodi ROS di controllo ciascuno creano computation graph piuttosto grande! L'interfaccia di rete consente invece la connessione al simulatore di molteplici ROS computation graph indipendenti, ciascuno con il proprio Master. Il wrapper creato consente pertanto quello che si potrebbe definire supporto ROS multigraph.

Per quanto riguarda invece le funzionalità offerte dal nodo `simspark_virtual_driver`, se ne riconoscono tre principali:

1. in fase di caricamento, nel caso in cui la simulazione venga eseguita localmente, si occupa di verificare che il server SimSpark *rcssserver3d* e il monitor *rcssmonitor3d* siano in esecuzione, e in caso contrario ne effettua l'avvio;

2. fornisce al computation graph un service per il caricamento di un nuovo robot nella simulazione, ritornando un handle per effettuare l'interazione, e un service per la cancellazione di un robot dalla simulazione;

3. fornisce, a seguito del caricamento di un robot nel simulatore, service per il controllo degli effector e topic su cui verranno pubblicate le informazioni provenienti dai perceptor, effettuando la conversione di formato.

Queste funzionalità, descritte approfonditamente nelle prossime sezioni, sono implementante attraverso la classe VirtualDriver che costituisce la vera implementazione di `simspark_virtual_driver`, lasciando alla funzione `main()` il mero compito di crearne un'istanza e richiamarne i metodi. Si parlerà di classi e istanze di classe riferendosi in entrambi i casi al nome della classe stessa; dal contesto sarà chiaro a cosa ci si riferisce.

4.1.1 Verifica e lancio del simulatore

L'implementazione di questa funzionalità è l'unico aspetto del codice che lo rende non portabile su sistemi Windows e Mac e potrà essere in futuro modificato o rimosso. La funzione di lancio automatico del simulatore è stata introdotta per velocizzare il debugging del sistema ed è implementata attraverso `fork` e `execve` del processo di `simspark_virtual_driver`. La verifica di esecuzione di server e monitor viene effettuata attraverso una chiamata a sistema (posix `system()` call) specificando l'esecuzione del comando `ps -All|grep rcssserver3d` e `ps -All|grep rcssmonitor3d`. Per effettuare l'avvio in locale del server di SimSpark e del monitor è sufficiente richiamare il metodo `int VirtualDriver::startSimSpark()` che ritorna 1 in caso di caricamento corretto. Esiste anche il metodo `int VirtualDriver::stopSimSpark()` che ne effettua la terminazione attraverso chiamata a sistema del comando `killall -9 rcssserver3d` e `killall -9 rcssmonitor3d`. L'URI per la comunicazione con il server viene caricato dal parameter server (si veda a proposito il paragrafo 4.2) e qualora non presente viene impostato come locale sulla porta di default: `127.0.0.1:3100`. L'avvio automatico del server viene comunque effettuato solamente qualora vi sia il parametro privato "auto_load" impostato a true, vedere a proposito 4.2.

4.1.2 Lancio e terminazione di agenti simulati

Un altro metodo della classe `VirtualDriver` richiamato in fase di avvio del nodo `simspark_virtual_driver` è `void VirtualDriver::publishServices()`. Il metodo si occupa di pubblicare in ROS i due service necessari a creare e cancellare un robot nella simulazione:

`add_new_agent:` service che si occupa del caricamento di un nuovo robot all'interno della simulazione. Come si può vedere dal contenuto del file srv, riportato in tabella 4.1, i parametri del request message sono `robot_name`, `robot_model`

e `namespace_to_use` anche se al momento solo gli ultimi due sono effettivamente utilizzati. `robot_model` specifica il modello di robot da utilizzare nella simulazione. E' possibile utilizzare il modello originale di SimSpark, l'Adelbaran Nao, e il modello del Robovie-X realizzato in questo lavoro. Ai fini di un corretto caricamento del modello è necessario che la descrizione del robot sia posta all'interno della directory di sistema di SimSpark, rispettando la seguente convenzione:

```
<SimSpark directory>/rsg/agent/<robot_model>/<robot_model>.rsg
```

La funzione di callback a servizio di `add_new_agent`

```
bool VirtualDriver::addNewAgent(simulator_simspark::AddNewAgent
    ::Request &req, simulator_simspark::AddNewAgent::Response &
    res)
```

si occupa avviare una nuova istanza di VirtualAgent, i cui dettagli si rimandano a alla sezione 4.1.3, passandogli i parametri del request message e l'URI del server SimSpark a cui connettersi. Il puntatore all'istanza viene quindi associato ad un handle numerico (banalmente il numero di istanze create dall'avvio del nodo) e memorizzato in una mappa. L'handle ritornato con il messaggio response potrà essere utilizzato in un secondo momento per effettuare la cancellazione del robot attraverso il service `kill_agent`.

string robot_model
string robot_name
string namespace_to_use
– – –
bool result
uint32 simulation_handle

Tabella 4.1: contenuto del file AddNewAgent.srv

`kill_agent:` service che effettua la terminazione di un robot simulato. I parametri del request e response message sono riportati in tabella 4.2, in particolare il parametro `simulation_handle` viene utilizzato dalla funzione di callback

```
bool VirtualDriver::killAgent(simulator_simspark::KillAgent::
    Request &req, simulator_simspark::KillAgent::Response &res)
```

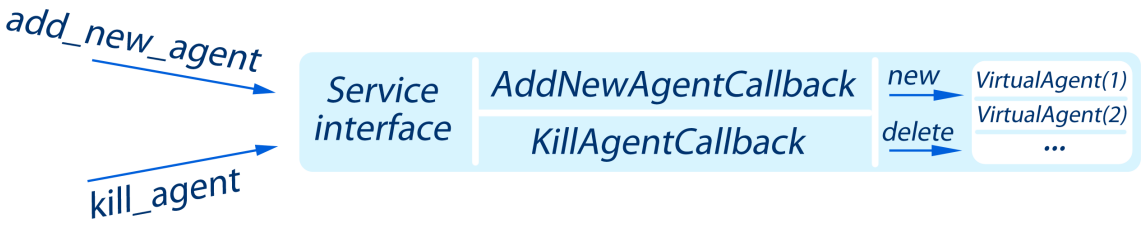


Figura 4.1: Nodo `simspark_virtual_driver`

per recuperare dalla mappa delle istanze attive di VirtualAgent quella relativa al robot interessato. Se individuata ne viene richiamato il metodo `void VirtualAgent::stop()`, quindi viene cancellata dalla mappa e distrutta.

uint32 simulation_handle
– – –
bool result

Tabella 4.2: contenuto del file KillAgent.srv

In figura 4.1 è riportata una rappresentazione del nodo `simspark_virtual_driver`.

4.1.3 Controllo dei robot simulati

Come visto la classe `VirtualDriver` si occupa solo del caricamento del nodo e dei servizi di avvio e terminazione delle sessioni di simulazione. In effetti tutto il lavoro di interazione con il simulatore e conversione del formato dei messaggi è svolto da istanze della classe `VirtualAgent`. Il motivo di questa divisione di ruoli è legato alle caratteristiche desiderate, descritte all'inizio del capitolo. In particolare per fornire il supporto multiagente si è scelto di incapsulare tutti gli aspetti di interazione con il singolo robot simulato all'interno di una classe, derivando quindi il supporto a più robot attraverso la creazione di istanze multiple.

La classe `VirtualAgent` fornisce l'interfacciamento tra ROS e SimSpark per il controllo di un robot simulato e come tale deve:

1. gestire la connessione TCP con SimSpark, avviandola e mantenendola attiva;

2. inizializzare il robot nella simulazione inviando un messaggio `CreateEffector` (si veda a proposito la sezione 1.4.1);

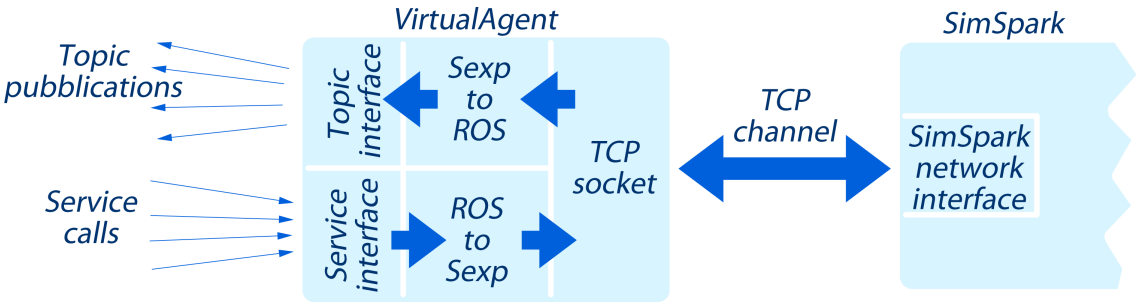


Figura 4.2: Classe `VirtualAgent`, interfacce ROS e SimSpark

3. esporre in ROS i servizi di controllo degli effector del robot e pubblicare i topic per la comunicazione delle notifiche di stato provenienti dai perceptor;

4. restare in ascolto sul canale TCP dei messaggi provenienti dal simulatore, inoltrare le informazioni dai perceptor convertendo i messaggi dal formato SimSpark a quello ROS;

5. effettuare lo `spin` dei service pubblicati servendoli e convertendo i messaggi in transito dal formato ROS a quello SimSpark.

In figura 4.2 si riporta lo schema funzionale della classe `VirtualAgent` con le interfacce verso ROS e SimSpark.

4.1.3.1 Gestione della connessione TCP con SimSpark

Si ricorda dalla sezione 1.4 che l'interfaccia di rete di SimSpark prevede la comunicazione di messaggi attraverso socket TCP. Ogni robot resta associato al proprio socket e la simulazione perdura fino alla chiusura del socket stesso. Per gestire la comunicazione TCP in maniera semplice e indipendente dal sistema operativo si è ricorso alle funzionalità delle librerie C++ Boost [S17], in particolare della libreria *asio*. Asio è una libreria che fornisce tra le altre cose supporto all'I/O asincrono su rete. Delle molte classi messe a disposizione si è fatto uso della più semplice, `boost::asio::tcp::iostream` che consente di agire su un socket come con le stesse modalità di uno stream C++. Non si è fatto uso delle funzionalità di I/O asincrono della libreria poichè ritenuto un livello di ottimizzazione per ora non necessario. Usufruendo del supporto multithreading delle librerie, le funzioni di ricezione dei messaggi da SimSpark e quella di invio dei messaggi verso il simulatore sono state implementate in due thread che agiscono sul medesimo socket. Singolarmente queste funzionalità non necessitano di I/O asincrono: nella comunicazione SimSpark⇒ROS

la ricezione, il parsing e l'inoltro in ROS dei messaggi è una sequenza di operazioni eseguite serialmente e ciclicamente, in maniera sufficientemente veloce da rientrare in ricezione prima dell'arrivo di nuovi messaggi dal simulatore; nella comunicazione ROS⇒SimSpark è invece utile l'attesa del completamento della comunicazione in modo da certificare il completamento dell'operazione nel ritorno dalla chiamata a service.

Per quanto riguarda i dettagli implementativi, la creazione e l'apertura del socket TCP viene effettuata nel costruttore dalla classe `VirtualAgent` richiamando il costruttore della classe `tcp::iostream stream` e passando come parametri l'indirizzo IP e la porta del server:

```
//Membro della classe VirtualAgent
tcp::iostream stream;

//Inizializzato nel costruttore
VirtualAgent::VirtualAgent(
        string robot_model,
        string client_namespace,
        string server_address,
        string server_port):stream(server_address, server_port),...
```

La classe `VirtualAgent` implementa quindi due metodi per l'invio e la ricezione di messaggi nel socket:

```
1 bool VirtualAgent::send(std::string message)
2 {
3   //[verifica bontà del socket...]
4   unsigned int len = htonl(message.length());
5   stream.write((const char*)(&len), sizeof(unsigned int));
6   stream << message;
7   return true;
8   //[..]
9 }
```

gestisce l'invio di messaggi verso SimSpark. Si ricorda che la messaggistica da e verso il server di simulazione prevede che i primi 32 bit del messaggio siano la dimensione del corpo del messaggio che segue espressa secondo l'ordine dei bit TCP, solitamente Big-Endian (si veda a proposito la sezione 1.4). Per questo alla riga 4 viene richiamata la funzione `htonl` per la conversione di notazione, quindi viene inviata la dimensione del messaggio e il messaggio stesso come sequenza di caratteri ASCII.

Per la ricezione di messaggi da SimSpark viene invece fornita la funzione

```
std::string VirtualAgent::receive(void)
{
    std::string message("");
    //[verifica bont\'a del socket...]
    char buff[sizeof(unsigned int)];
    stream.read(buff, sizeof(unsigned int));
    unsigned int len = ntohl(*((unsigned int *)&buff));
    char *buff=(char *)malloc(len*sizeof(char));
    stream.read(buff, len);
    message = std::string(buff, len);
    free(buff);
    //[...]
    return message;
}
```

che esegue le operazioni opposte a quelle della `send`: per prima cosa riceve al dimensione del corpo del messaggio e la converte di formato, quindi predispone un buffer per il messaggio e lo legge, salvandolo infine all'interno di un oggetto `string`.

4.1.3.2 Inizializzazione del robot nella simulazione

L'inizializzazione del robot viene effettuata nel costruttore della classe. Il modello del robot è parametro del costruttore, viene quindi creato un `CreateEffector` message e inviato verso il server:

```
...
std::stringstream strs;
strs.str("");
strs<<"(scene rsg/agent/"<<robot_model<<"/"<<robot_model<<".rsg)";
send(strs.str());
receive();
...
```

Dall'implementazione si capisce l'importanza di organizzare correttamente le descrizioni dei modelli all'interno delle directory di SimSpark. La chiamata a `receive` alla riga 6 serve a confermare l'avvenuto caricamento del modello. Si osserva in questa chiamata si celi l'esigenza di I/O asincrono, infatti qualora il modello rsg non sia trovato o sia errato, nessuna comunicazione ritornerà dal server (che probabilmente sarà anche crashato) e l'esecuzione resterà bloccata sulla chiamata a receive. Sarebbe utile disporre di un timeout oltre il quale dichiarare la connessione fallita agendo di conseguenza. Questa funzionalità non è stata ancora implementata.

4.1.3.3 Esposizione servizi in ROS

Come esposto nella sezione 2.4 ROS fornisce essenzialmente due tipologie di comunicazione: sincrona RPC-style attraverso service e streaming asincrono di dati attraverso topic. Le tipologie di comunicazione da gestire sono due: comunicazione da ROS a SimSpark per il controllo degli effector, implementata attraverso service; comunicazione da SimSpark a ROS per le notifiche dai perceptor, implementata attraverso topic. La scelta di utilizzare topic per le notifiche dai perceptor è indubbiamente la più sensata e consente la fruizione contemporanea da più nodi delle informazioni pubblicate. Il controllo degli effector invece poteva essere effettuato sia tramite topic che tramite service. La scelta di utilizzare service è stata dovuta perlopiù alla sincronicità della chiamata, il cui ritorno certifica il completamento dell'operazione. Un'interfaccia a topic per il controllo degli effector sarà eventualmente aggiunta in futuro poichè potrebbe essere utile al controllo di più robot o al controllo del robot reale e simulato contemporaneamente.

Durante l'esecuzione del costruttore della classe `VirtualAgent` vengono pubblicati tanti service quanti effector si possono controllare e tanti topic quanti i perceptor che possono inviare notifiche. ROS richiede di assegnare nomi univoci ai service e topic, si è scelto di associare il nome alla tipologia di perceptor/effector servito: `<nome effector>_effectors_service` per i service, `<nome perceptor>_perceptors_topic` per i topic. Al fine di mantenere univocità dei nomi, vista la possibilità di avere molteplici agenti attivi contemporaneamente, si è sfruttata la struttura gerarchica dei nomi di ROS raggruppando service e topic all'interno di un namespace. Il namespace da utilizzare è specificato dall'agente in fase di chiamata al service `add_new_agent` nel parametro `namespace_to_use` del request message.

I service sono pubblicati direttamente nel costruttore della classe `VirtualAgent`, inizializzando altrettanti oggetti `ros::ServiceServer`, membri della classe. La gestione dei topic è invece più complessa e trattata più avanti assieme alle problematiche di parsing dei messaggi. Per ora basti sapere che vi è una classe per ogni tipologia di messaggio perceptor la quale incapsula quanto necessario alla pubblicazione del topic corrispondente, queste classi implementano tutte l'interfaccia `RosMessage`. Istanze di queste classi sono raccolte all'interno della classe `MessageMap`. Ogni `VirtualAgent` ha come membro un'istanza di `MessageMap` quindi, nell'ordine, il costruttore di `VirtualAgent` richiama quello di `MessageMap` che richiama a sua volta costruttori di tutti i `RosMessage` supportati, i quali, ognuno per la propria tipologia di perceptor, pubblicano i topic associati.

In tabella 4.3 si riporta la lista dei service e topic pubblicabili. Al momento ven-

gono pubblicati tutti i service e i topic disponibili senza considerare gli effector e i perceptor effettivamente installati nel robot, di conseguenza su un topic associato ad un perceptor non presente non sarà mai pubblicato alcun messaggio. Analogamente il controllo di effector non utilizzati non avrà alcun effetto. Una soluzione potrebbe essere l'analisi della descrizione URDF del robot, se questa includesse una rappresentazione di effector e perceptor, oppure l'analisi della descrizione RSG. Comunque è solo un problema di forma in quanto non comporta errori.

4.1.3.4 Ascolto del canale TCP - conversione e inoltro dei messaggi in ROS

L'ascolto del socket per comunicazioni da SimSpark, il parsing dei messaggi e la pubblicazione dei topic viene effettuata nel medesimo contesto. A occuparsene è un apposito thread avviato durante l'esecuzione del costruttore della classe `VirtualAgent`. Anche il multithreading è stato gestito attraverso funzionalità della libreria boost, la boost::thread:

```
...
simspark_comm_thread = boost::shared_ptr<boost::thread>(new boost::
    thread( boost::bind( &VirtualAgent::socketListeningThread, this)));
...
```

La funzione inizializza il puntatore `simspark_comm_thread`, membro della classe `VirtualAgent`, all'istanza del thread creato. La funzione `boost::bind`, che consente di generalizzare l'interfacciamento a funzioni, metodi di istanza, object function occupandosi del bind dei parametri, viene utilizzata per impostare il metodo d'istanza `VirtualAgent::socketListeningThread`, riportato nel listato 4.1, come funzione *run* del thread. Come si osserva dal codice la funzione è strutturalmente semplice ed esegue ciclicamente ricezione, parsing e pubblicazione dei messaggi. La semplicità strutturale nasconde la complessità intrinseca del processo di parsing. Come spiegato nella sezione 1.4, tutte le comunicazioni dai perceptor giungono attraverso il medesimo canale, quello TCP di ascolto, raggruppate in burst di messaggi.Ogni 20ms circa i server SimSpark invia un burst contenente la concatenazione di molti messaggi. Il parsing dei messaggi deve quindi prevedere:

1. Separazione di ogni singolo messaggio dal burst
2. Identificazione del tipo di messaggio, il tipo di perceptor di provenienza
3. Estrazione dei parametri presenti, con supporto alla variabilità della struttura S-Expression (il numero e il tipo dei parametri presenti può variare nel tempo)

Service per effector	
Nome del service	**Effector associato**
∼/create_effectors_service	Da implementare - fornirà funzionalità di caricamento personalizzato dell'ambiente
∼/hingejoint_effectors_service	Controllo di giunti hinge
∼/universaljoint_effectors_service	Controllo di giunti universal
∼/beam_effectors_service	Posizionamento del robot in modalità trainer (solo simulazione calcio)
∼/say_effectors_service	Controllo vocale (solo simulazione calcio)
Topic per perceptor	
Nome del topic	**Perceptor associato**
∼/gyrorate_perceptors_topic	Notifiche dai gyrorate perceptor
∼/hingejoint_perceptors_topic	Notifiche dai perceptor (odometria) dei giunti di tipo hinge
∼/universaljoint_perceptors_topic	Notifiche dagli universal joint perceptor
∼/touch_perceptors_topic	Notifiche provenienti dai sensori di contatto
∼/forceresistance_perceptors_topic	Per le notifiche provenienti dai sensori di forza
∼/accelerometer_perceptors_topic	Informazioni provenienti dagli accelerometri
∼/vision_perceptors_topic	Informazioni provenienti dal sensore di visione (solo gioco del calcio)
∼/gamestate_perceptors_topic	Informazioni sullo stato del gioco (solo gioco del calcio)
∼/agentstate_perceptors_topic	Informazioni circa lo stato interno dei robot (solo gioco del calcio)
∼/hear_perceptors_topic	Informazioni provenienti dai sensori d'udito

Tabella 4.3: Nomi dei service e topic pubblicati dal virtualAgent, dove a ∼ si sostituisce il namespace scelto dall'agente

```
1  void VirtualAgent::socketListeningThread()
2  {
3      std::string message;
4      while(should_run_)
5      {
6          message = receive();
7          boost::shared_ptr<parser::PredicateList> predList = sexpparser.
               Parse(message);
8          message_map.clean();
9          for(parser::PredicateList::TList::const_iterator iter =
               predList->begin();iter != predList->end();iter++)
10             message_map.findMessageAndParse(*iter);
11         message_map.publishMessages();
12     }
13 }
```

Listato 4.1: Funzione run del thread di ascolto del socket

4. Compilazione del messaggio specifico per il topic associato al perceptor sorgente

5. Pubblicazione del messaggio nel topic

I passi vanno eseguiti per ogni messaggio del burst. Un'ulteriore funzionalità desiderabile è il raggruppamento dei messaggi per tipologia: messaggi come quelli provenienti dal vision perceptor sono incapsulati all'interno della medesima espressione `see` e vengono trattati assieme; differente è il caso per i messaggi dai percetor di odometria, che arrivano separati nel burst e devono essere ricompattati. Per questo motivo la pubblicazione dei messaggi ROS sui topic viene ritardata alla terminazione del parsing del burst.

Il riconoscimento, la separazione e la navigazione della struttura ricorsiva delle S-Expression non è concettualmente complicata, ma nemmeno banale. In tal senso viene in aiuto la libreria di parsing di S-Expression utilizzata da SimSpark e racchiusa in Oxygen. Dalla scelta iniziale di linkare esternamente il parser si è passati alla decisione di importare il codice del parser all'interno del progetto, questo al fine di limitare le dipendenze con moduli esterni a ROS. Si osservi in particolare che l'utilizzo di `simulator_simspark` non implica necessariamente la presenza delle librerie Spark nel medesimo sistema, in quanto il server, sfruttando la connessione TCP, può essere eseguito su una macchina differente. Le classi del parser sono state raggruppate all'interno del namespace `parser`. La classe principale che si occupa del parsing è

`parser::SexpParser`. Essa dispone in particolare del metodo `Parse` che, acquisita la stringa di burst, ritorna il riferimento ad un oggetto `parser::PredicateList`. `PredicateList` incapsula il concetto di lista di predicati, ove per predicato di intende una S-Expression. In quanto *lista* di predicati essa fornisce iteratori per scorrere i singoli messaggi (predicati) del burst. Ogni messaggio è ritornato attraverso un riferimento al predicato che lo incapsula. Il predicato infine, istanza della classe `parser::Predicate`, fornisce metodi per:

- accesso al nome del predicato (primo parametro dell'S-Expression) corrispondente al nome del messaggio, secondo quanto riportato nella sezione 1.4
- l'accesso diretto ai valori di parametri, qualora questi siano valori terminali, secondo la grammatica 1.4
- accesso diretta a valori di parametri, qualora questi siano terminali contenuti in sotto S-Expression di primo livello, ovvero di singolo livello di nesting
- accesso ad un iteratore di una lista di sotto-predicati, qualora la S-Expression presenti struttura ricorsiva

Al fine di semplificare le operazioni di parsing descritte, nonchè di facilitare l'estensione futura dei messaggi perceptor supportati, si è realizzato un particolare framework di decodifica dei messaggi in cui ogni tipologia di messaggio è associata ad una classe distinta di decodifica/pubblicazione in ROS.
Queste classi hanno in un'interfaccia comune: tutte devono poter effettuare il parsing di una S-Expression, ciascuna per il proprio tipo di perceptor message; tutte devono compilare i campi del proprio tipo di messaggio ROS e pubblicarlo alla fine del parsing sul topic corrispondente. Si sono volute esplicitare queste caratteristiche comuni realizzando l'interfaccia `RosMessage`:

```
class RosMessage
{
protected:
    bool m_bModified;
    ros::Publisher topic;
public:
    virtual bool parse(const parser::Predicate & predicate)=0;
    virtual bool publish(void)=0;
    virtual void clean(void)=0;
    RosMessage(){m_bModified=false;}
};
```

che presenta interfacce per i metodi di parsing di un predicato, `parse`, di pubblicazione del messaggio, `publish`, di cancellazione del messaggio, `clean`. L'interfaccia fornisce anche alcuni membri comuni: `m_bModified` per indicare se c'è qualcosa da pubblicare, e `topic`, inizializzato opportunamente dal costruttore della sottoclasse, conterrà l'istanza per la gestione del topic. Implementare la gestione di un perceptor message significa quindi:

- creare una nuova classe che estenda l'interfaccia `RosMessage`
- aggiungervi come membro un'istanza del messaggio ROS per il corrispondente topic
- specificare nel costruttore la pubblicazione del topic specifico
- implementare i metodi dell'interfaccia `RosMessage`: `parse` dovrà effettuare il parsing del predicato passato come parametro aggiungendo quanto estratto nel messaggio ROS; `publish` dovrà pubblicare se necessario, attraverso il membro `topic`, il messaggio ROS; infine `clean` effettuerà la pulitura della del messaggio per iniziare un nuovo ciclo di parsing.

In appendice A è riportato un esempio di implementazione del meccanismo di parsing e inoltro di un messaggio perceptor HingeJoint verso la rete ROS.

Come citato precedentemente, le molteplici istanze delle classi derivate da `RosMessage`, una per ogni specifica tipologia di perceptor message, sono raccolte all'interno della classe `MessageMap`. Come suggerisce il nome la `MessageMap` incapsula una mappa associativa, che collega le istanze dei `RosMessage` a stringhe contenenti il nome del corrispondente perceptor message. Ciò consente, ricavato il nome del perceptor message attraverso l'apposita funzione del parser, un agile e rapido recupero dell'istanza del `RosMessage` in grado di effettuare la conversione del messaggio SimSpark in messaggio ROS. Come detto, poichè le funzionalità ad alto livello delle classi di conversione sono le medesime, è sufficiente richiamare i metodi dell'interfaccia `RosMessage` ritornata dalla `MessageMap`, la risoluzione del metodo sarà effettuata dai meccanismi di polimorfismo di C++.

E' giunto quindi il momento di approfondire la struttura della classe `MessageMap`. Tra i membri della classe si trovano: innanzi tutto una mappa associativa, che associa specifiche istanze di `RosMessage` alla stringa che identifica il tipo di perceptor message supportato; istanze di classi derivate da `RosMessage`, una per ogni tipologia di perceptor message da supportare, sono i puntatori a queste istanze ad essere

inseriti nella mappa dal costruttore della classe. I metodi pubblici della classe sono invece:

```
void clean();
boost::shared_ptr<RosMessage> find(const std::string &key);
bool findMessageAndParse(const parser::Predicate & predicate);
bool publishMessages();
```

ora descritti in dettaglio.

```
void MessageMap::clean(){
  std::map<std::string,boost::shared_ptr<RosMessage>>::const_iterator
      it;
  for(it=message_map.begin();it!=message_map.end();it++)
       it->second->clean();
}
```

richiama il corrispondente metodo delle classi di parsing contenute nella mappa e viene richiamato ad ogni ciclo di ricezione di messaggi da SimSpark, prima di iniziare il parsing, dal metodo `socketListeningThread` di `VirtualAgent`.

```
boost::shared_ptr<RosMessage> MessageMap::find(const std::string &key){
  std::map<std::string,boost::shared_ptr<RosMessage>>::const_iterator
      it;
  it=message_map.find(key);
  if(it!=message_map.end())
      return (it->second);
  return boost::shared_ptr<RosMessage>();
}
```

effettua la ricerca sulla mappa di un'istanza di `RosMessage` in grado di effettuare il parsing del tipo di perceptor message passato, ritornandone il puntatore se trovato, NULL altrimenti.

```
 bool MessageMap::findMessageAndParse(const parser::Predicate &
     predicate)
{
   boost::shared_ptr<RosMessage> message=find(predicate.name);
    if(message.get()!=NULL)
        return message->parse(predicate);
    return false;
}
```

viene richiamato dal thread, prende come input il predicato che incapsula un perceptor message e utilizza il metodo `find` per trovare nella mappa un'istanza di `RosMessage` in grado di effettuarne il parsing, quindi ne richiama il metodo `parse` precedentemente descritto.

```
bool MessageMap::publishMessages()
{
    bool return_value=true;
    std::map<std::string,boost::shared_ptr<RosMessage>>::const_iterator
        it;
    for(it=message_map.begin();it!=message_map.end();it++)
        return_value&=(it->second->publish());
    return return_value;
}
```

richiama il metodo `parse` su tutte le istanze di `RosMessage` contenute nella mappa. Il metodo viene chiamato da `socketListeningThread` al termine del parsing dell'intero burst, realizzando così l'obiettivo di raggruppamento dei messaggi.

4.1.3.5 Spin per la fornitura dei service - creazione dei messaggi per SimSpark

Come presentato nella sezione 2.6.3, la fornitura dei service in ROS richiede l'esecuzione di una funzione di *spin*. L'esigenza di gestire istanze di `VirtualAgent` multiple all'interno del nodo `simspark_virtual_driver` in un contesto multithreaded implica l'esigenza di utilizzare altrettante code per la gestione dei callback, come descritto nel paragrafo 2.6.2. Si è quindi aggiunto il membro `ros::CallbackQueue agent_queue_;` alla classe `VirtualAgent`, impostandolo nel costruttore della classe come coda per i callback essa relativi, attraverso il comando:

```
agent_node_handle_.setCallbackQueue(&agent_queue_);
```

dove `agent_node_handle_` è un'istanza di `ros:NodeHandle` anch'essa membro della classe. In questo modo i service successivamente registrati e gli eventuali topic sottoscritti faranno riferimento alla coda dei callback `agent_queue_` anzichè alla coda interna di `VirtualDriver`. L'utilizzo esplicito di una coda di callback modifica la tecnica di spin, che deve essere implementata manualmente. Si è scelto di creare un thread specifico per la fornitura dei service. Il thread viene avviato durante l'esecuzione del costruttore della classe `VirtualAgent` e come il thread per l'ascolto del canale TCP fa uso delle librerie boost:

```
...
ros_spin_thread = boost::shared_ptr<boost::thread>(new boost::thread(
   boost::bind( &VirtualAgent::spinThread, this)));
...
```

Il metodo `spinThread()` non fa altro che richiamare ciclicamente la gestione dei callback pendenti nella coda `agent_queue_`, continuando finchè lo stato del nodo è buono e finchè il flag `should_run_`, utilizzato per la terminazione dei thread, è `true`. Il metodo `callAvailable` di `ros::CallbackQueue` serve tutti i callback in coda, uscendo anticipatamente dall'attesa dopo 100ms di inattività.

```
void VirtualAgent::spinThread()
{
    while(ros::ok() && should_run_)
    {
      queue.callAvailable(ros::WallDuration(0.1));
    }
}
```

Come descritto nella sezione 4.1.3.3 `simspark_virtual_driver` crea un service per ogni tipologia di effector supportata. Essendovi un metodo di callback differente per ogni service, la conversione dei messaggi da ROS a SimSpark è implementata differentemente in ogni callback a seconda della tipologia di effector. La gestione di questo tipo di comunicazione risulta molto più semplice infatti, noto il formato della S-Expression utilizzato dal'effector message, è sufficiente comporne la stringa con i parametri forniti dal request message del service e inviare il messaggio a SimSpark attraverso il metodo `send` di `VirtualAgent`. In appendice B è riportato un esempio di implementazione di service per il controllo dell'effector SimSpark HingeJoint dalla rete ROS.

4.2 Il simulator_simspark launch file

E' stato sviluppato un file launch, chiamato *simspark_virtual_driver.launch*, per semplificare il caricamento del nodo `simspark_virtual_driver`.

```
<launch>
  <arg name="address" default="127.0.0.1"/>
  <arg name="port" default="3100" />
  <arg name="auto_load" default="true" />

  <node name="simspark_virtual_driver" pkg="simulator_simspark" type="
      simspark_virtual_driver" output="screen">
    <param name="rcssserver_address" value="$(arg address)" />
    <param name="rcssserver_port" value="$(arg port)"/>
    <param name="auto_load" value="$(arg auto_load)"/>
  </node>
```

```
</launch>
```

Il file, come descritto in sezione 2.5 utilizza il formato XML. E' possibile osservare l'utilizzo di quattro tag principali:

`<launch>` è l'elemento root di ogni file launch

`<arg>` consente di parametrizzare il file launch attraverso la specifica di valori che possono essere costanti o passati come argomento da linea di comando. Gli attributi dell'elemento sono:

`name` nome associato all'argomento, se non vengono specificati altri attributi l'argomento è obbligatorio

`default` mutuamente esclusivo con `value`, imposta un valore di default per l'argomento nel caso questo non venga impostato

`value` mutuamente esclusivo con `default`, imposta un valore costante che non può essere modificato o reimpostato da linea di comando

Nella fattispecie vengono creati tre argomenti: due per la specifica dell'URI del server SimSpark, inizializzati ai valori di default per la simulazione in locale, e uno per impostare l'autoavvio del server SimSpark (in locale). Per richiamare il valore dell'argomento si utilizza la sintassi "`$(arg <nome argomento>)`"

`<node>` specifica un nodo da far eseguire da *roslaunch*. Qualora siano specificati più nodi non è data garanzia sull'ordine di caricamento. Gli attributi più importanti sono:

`pkg` specifica il package di appartenenza del nodo da caricare

`type` nome dell'eseguibile da caricare. Deve appartenere al package specificato

`name` nome assegnato al nodo caricato. Non va specificato il namespace, assegnabile invece con l'attributo `ns`

`args` argomenti opzionali da passare al nodo

`respawn` "true|false", specifica se ricaricare automaticamente il nodo in caso di crash

`required` "true|false", specifica se interrompere l'esecuzione dell'intero roslaunch in caso di crash del nodo

`ns` consente di modificare il namespace del nodo

`output` "log|screen", consente di impostare dove indirizzare lo stdout e lo stderr del nodo, se sul file log di ROS o su schermo. Il file log ROS si trova in $ROS_HOME/log

`<node>` consente in oltre di impostare alcuni sottoelementi tra cui `<param>`. Nel file sopra riportato il tag node viene utilizzato per caricare il nodo del wrapper SimSpark impostando l'output a schermo e creando alcuni parametri privati

`<param>` consente di definire un parametro nel Parameter Server. Oltre a semplici valori è possibile specificate file di testo o file binari. Può essere utilizzato come elemento di `<launch>`, nel qual caso specifica un parametro globale, oppure utilizzato all'interno del tag `<node>`, specificando così un parametro privato del nodo, accessibile anteponendo $\sim$ al nome del parametro. Gli attributi principali dell'elemento sono:

`name` nome assegnato al parametro

`value` valore assegnato al parametro, da omettere nel caso di specifica di file binari o testuali

`type` tipo del parametro, a scelta tra "str|int|double|bool". Se non specificato `roslaunch` effettua un parsing automatico

`textfile` specifica il nome del file che sarà associato al parametro. Il file sarà caricato come stringa

`textfile` specifica il nome del file che sarà associato al parametro. Il file sarà caricato come stringa

In *simspark_virtual_driver.launch* viene utilizzato per passare l'URI del server SimSpark al nodo `simspark_virtual_driver` come parametro privato.

Capitolo 5

Applicazione - l'ambiente didattico robot_control

Come primo utilizzo del lavoro di integrazione i SimSpark in ROS si è realizzato il package `robot_contro`, un framework didattico che sarà utilizzato nelle esperienza di laboratorio del corso di "Robotica Autonoma" del dipartimento di ingegneria informatica dell'università di Padova. Il corso tratta, tra le altre cose, problematiche di controllo dei movimenti di robot umanoidi e durante le sessioni di laboratorio sarà assegnato agli studenti il compito di effettuare la movimentazione del robot Robovie-X nella simulazione SimSpark. Una simile esperienza è stata effettuata nel corso tenutosi nell'anno accademico 2010-2011.

Durante l'esperienza dello scorso anno è stato fornito agli studenti un programma scheletro che già implementava e nascondeva i dettagli della connessione e dell'interfacciamento TCP con il simulatore. Il programma forniva le sole funzionalità di controllo dei giunti e la ricezione dei dati odometrici, utilizzati per mantenere aggiornato lo stato del robot. Lo stato del robot era rappresentato da due vettori, uno per la velocità e uno per la posizione dei giunti. Il vettore di velocità poteva essere impostato attraverso un'apposita funzione, quindi le velocità venivano inviate al simulatore attraverso l'hinge-joint effector message. Il vettore della posizione dei giunti veniva aggiornato ad ogni ciclo di messaggistica hinge-joint perceptor message proveniente da SimSpark.

Lo studente poteva interagire con la simulazione a partire da una funzione `think`, richiamata dal thread a seguito dell'aggiornamento di stato proveniente da SimSpark. Dalla funzione era possibile scegliere nuovi valori di velocità da assegnare ai giunti in funzione della posizione raggiunta. Scopo dell'esperienza era la realizzazione di un controllo di più alto livello che consentisse movimenti complessi, in particolare

l'obiettivo finale era ottenere una camminata stabile. L'approccio suggerito era un controllo basato su frame motori.

Per il corso dell'anno accademico 2011-2012 si è voluto ricreare un ambiente di sviluppo simile, utilizzando questa volta l'interfaccia ROS offerta da `simulator_simspark`. L'utilizzo di `simulator_simspark` porta diversi vantaggi, Per quanto riguarda la simulazione

- possibilità di controllare tutti i sensori messi a disposizione da SimSpark. Nel framework precedente era possibile utilizzare solo i sensori di odometria
- possibilità di utilizzare tutti gli effector del simulatore, in particolare quelli per riposizionare il robot sul campo

Per quanto riguarda invece l'introduzione di ROS, i vantaggi derivati sono:

- contatto semplificato con l'ambiente di sviluppo ROS, alle sue tecnologie e alle sue interfacce
- possibilità di utilizzare le interfacce ROS per implementare canali di comunicazione tra i robot di alto livello, aprendo prospettive di simulazione multirobot
- possibilità di utilizzare i numerosi package ROS per andare oltre il semplice controllo del movimento, potendo approfondire aspetti di interesse

Il lavoro poi si inserisce nel progetto di passare gradualmente a ROS come ambiente di sviluppo di tutte le attività del corso.

5.1 Il package `robot_control`

L'ambiente didattico per il controllo del Robovie-X simulato è stato sviluppato come package ROS e chiamato `robot_control`. Il nome non fa direttamente riferimento alla simulazione poiché l'intenzione è stata quella di creare un ambiente strutturato, costituito da interfacce generiche riutilizzabili, per il controllo di molteplici simulatori, molteplici robot, e possibilmente anche robot reali.

5.1.1 Struttura delle classi

In figura 5.1 è riportato lo schema della gerarchia di classi realizzata. Con sfondo bianco sono riportate le classi implementate mentre in quelle in grigio sono classi che potranno essere aggiunte in futuro: il controllo del robot Robovie-X reale, il

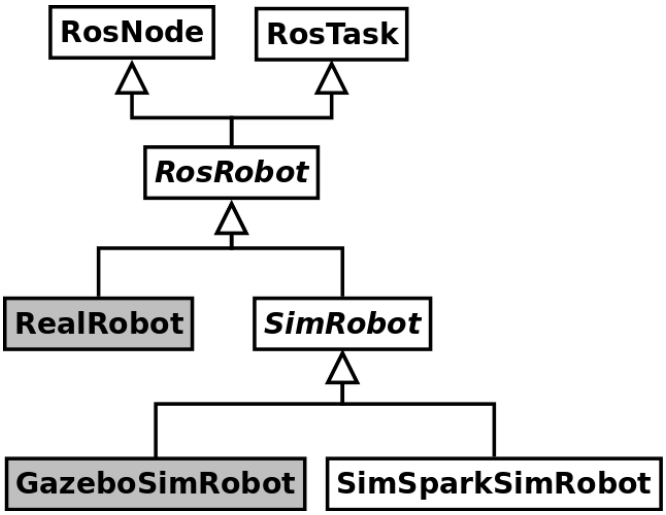


Figura 5.1: Gerarchia delle classi in `robot_control`

controllo della simulazione Gazebo.
Le funzionalità introdotte dalle classi sono:

`RosTask` costituisce la classe base per un task ROS e fornisce funzionalità per l'esecuzione di un thread di spin, utile nel caso il nodo voglia sottoscrivere topic o offrire service

`RosNode` costituisce la classe base per un nodo ROS, offrendo metodi per l'inizializzazione del nodo e la sua terminazione

`RosRobot` costituisce l'interfaccia di un robot e ne incapsula lo stato. Definisce gli attributi che caratterizzano lo stato e le proprietà di un robot come: il nome, il modello, la descrizione URDF, lo stato corrente dei giunti (posizione e velocità), lo stato futuro o target dei giunti (posizione da raggiungere / velocità da raggiungere). Le funzionalità che implementa sono:

- aggiornamento dello stato corrente del robot a partire da un vettore di nuove posizioni e/o velocità dei giunti
- aggiornamento dello stato target a partire da un vettore di nuove posizioni e/o velocità dei giunti
- pubblicazione dello stato (messaggi `sensor_msgs/JointState`) sul topic `/joint_states_out`. E' possibile effettuare il remapping di questo topic per pubblicare lo stato su `tf`

ascolto sul topic `/joint_states_in` di messaggi `sensor_msgs/JointState` per fissare un nuovo stato target. E' possibile effettuare il remapping di questo topic per associarlo al controllo `joint_state_publisher`

controllo in velocità dei giunti

controllo in posizione dei giunti

La classe definisce anche i seguenti metodi astratti, da implementare in classi derivate:

connessione al driver (reale o virtuale)

disconnessione dal driver (reale o virtuale)

comando al driver della velocità dei giunti

`SimRobot` costituisce l'interfaccia generica per la simulazione di un robot, al momento vuota

`SimSparkSimRobot` rappresenta l'implementazione per la simulazione in SimSpark, quindi utilizza le interfacce specifiche del wrapper `simspark_virtual_driver` per il controllo della simulazione. Fornisce callback per ogni tipologia di perceptor supportata e implementa i metodi astratti di `RosRobot` per la connessione e la disconnessione al driver `simspark_virtual_driver` e per il comando della velocità dei giunti. Infine implementa il controllo degli altri effector di SimSpark

Come esempio di utilizzo del framework viene fornito un semplice programma di debugging che, estendendo a sua volta `SimSparkSimRobot`, fornisce un controllo da tastiera dei giunti, consente di visualizzare a video i messaggi scambiati con il `simspark_virtual_driver` e testarne le diverse funzionalità.

5.1.2 Controllo della simulazione

Nella classe `SimSparkSimRobot` è possibile trovare callback per ogni tipologia di perceptor supportata. Un primo livello di controllo avviene con l'implementazione di questi metodi, lasciati da completare allo studente. In questo modo è possibile compiere azioni in base ai valori forniti da accelerometri, giroscopi e quant'altro. Non esiste alcun metodo "think" poichè non è presente un unico canale di feedback dalla simulazione. Eventualmente sarà interesse dello studente implementare una propria funzione "think" richiamata al termine del callback di aggiornamento dello stato dei

giunti. Il framework fornito è comunque in grado di utilizzare autonomamente le informazioni di stato dei giunti per aggiornare lo stato interno del robot, mantenuto da `RosRobot`. Per quanto riguarda il controllo del robot sono state implementate alcune funzionalità che consentono un controllo semplice:

- controllo in velocità: attraverso il metodo

  ```
  setJointVelocity(std::vector<std::string> joint_name, std::vector
      <float> velocity)
  ```

  è possibile attivare specifici giunti del robot ad una certa velocità. Se non arrestato prima, il giunto viene automaticamente fermato al raggiungimento del fine corsa

- controllo in posizione: attraverso il metodo

  ```
  setJointAngle(std::vector<std::string> joint_name, std::vector<
      float> angle, float time)
  ```

  è possibile attivare i giunti ad una velocità data dalla differenza tra la posizione attuale e la posizione desiderata divisa per il tempo d'azione impostato

- controllo in posizione tramite interfaccia grafica: si veda a proposito la sezione 5.1.3.

Ulteriori e più raffinati controlli possono essere implementati dagli studenti utilizzando le informazioni sullo stato del robot messe a disposizione dalle classi. Ad esempio il controllo in posizione non fa alcuna valutazione sulla fattibilità del movimento nel tempo impostato, in particolare, rispetto alla frequenza di aggiornamento della simulazione, la posizione di destinazione può essere superata prima che sia dato il comando di arresto. Questa condizione, a seconda dell'implementazione del controllo di posizione, può generare un posizionamento corretto o una condizione di instabilità.

5.1.3 Strumenti di controllo avanzati

Per un controllo ancora più semplice dei giunti del robot è possibile utilizzare il package `joint_state_publisher` che realizza un'interfaccia grafica a partire dalla descrizione URDF del robot. Lo strumento effettua un'analisi della descrizione URDF, che deve essere caricata nel parametro `robot_description`, individuando tutti i giunti non fixed e i loro range di variazione. Quindi viene costruita un'interfaccia

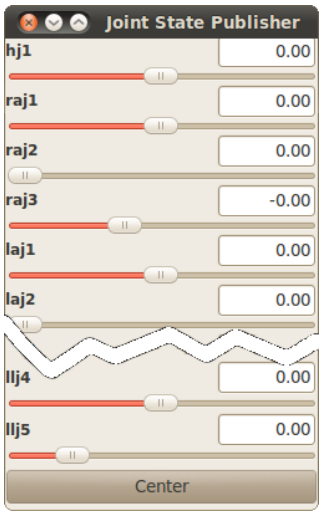


Figura 5.2: Interfaccia di controllo giunti tramite slider

grafica con uno slider di controllo per ciascun giunto, simile a quella riportata in figura 5.2. Periodicamente, con una frequenza impostabile, l'applicazione pubblica messaggi `sensor_msgs/JointState` sul topic `/joint_states`. Effettuando il remapping del nome del topic su `/joint_states_in` si effettua il collegamento tra il nodo e `robot_control`, come spiegato nella sezione 5.1.4. I giunti vengono attivati ad una velocità di default, impostabile dall'utente.

La classe `RosRobot` effettua anche la pubblicazione dello stato del robot su `tf` avvalendosi delle funzionalità di `robot_state_publisher`. `robot_state_publisher` è un package che facilita l'uso di `tf`: prende in input dal topic `/joint_states` l'angolo dei giunti attraverso messaggi `sensor_msgs/JointState` e, utilizzando la descrizione URDF del robot, calcola e pubblica le posizioni 3D dei suoi link (origini e rotazioni su messaggi `tf/tfMessage`) sul topic `/tf`. Ciò consente di utilizzare il package `tf` e tutti gli strumenti ROS basati su di esso, come rviz.

5.1.4 Il launch file

Si è realizzato un launch file per il programma di esempio fornito agli studenti. Il file può essere utile per comprensione dei meccanismi di name remapping e consente la personalizzazione dell'ambiente di lavoro: consente l'avvio selettivo del controllo dei giunti tramite GUI e l'avvio di rviz per la visualizzazione dello stato del robot. Il contenuto del file è riportato nel listato 5.1. Nella prima parte del file sono definiti gli argomenti specificabili da linea di comando, tutti opzionali e con un valore di default:

`urdf_model` percorso completo del file contenente la descrizione URDF del robot da simulare (il modello di default è quello del Robovie-X, contenuto nel package `robovie_x_model`)

`gui` valore booleano per utilizzare o meno l'input grafico della posizione dei giunti (utilizzato di default)

`rviz` valore booleano per avviare o meno rviz e visualizzare lo stato del robot (non utilizzato di default)

`robot_name` nome da assegnare al robot ("DEIbot" di default)

`robot_model` nome del modello del robot coerente con il nome assegnato alla descrizione RSG in SimSpark ("robovie_x" di default)

`node_name` nome da assegnare al nodo di controllo, utilizzato come namespace per i service e i topic forniti da `simspark_virtual_driver`

Segue poi la specifica di alcuni parametri nel Parameter Server:

`robot_description` stringa con il contenuto della descrizione URDF del robot da simulare. Il file caricato è quello specificato nell'argomento `urdf_model`. Il parametro viene utilizzato da rviz, per la visualizzazione, da `robot_state_publisher` per il calcolo della cinematica diretta, da `joint_state_publisher` per ricavare i giunti non fixed del modello e i loro limiti e da `node_sample` per ricavarne i limiti dei giunti

`use_gui` parametro utilizzato dal nodo `joint_state_publisher` per impostare la visualizzazione o meno della finestra di dialogo con slider per il controllo dei giunti.

Infine il file specifica i nodi da caricare:

`joint_state_publisher`, solo se l'argomento `gui` è impostato a true

`rviz` e `robot_state_publisher`, solo se l'argomento `rviz` è impostato a true

`node_sample`

Si osservi che parametri e nodi sono opportunamente rimappati al fine di consentire il caricamento di più nodi di controllo. Il namespace assegnato è il nome del nodo di controllo. Va aggiunto che a causa di un bug presente nella versione di ROS in

uso, la specifica di namespace per rviz e `robot_state_publisher` non effettua il remapping del topic `/tf`, pertanto non sarà possibile simulare più robot visualizzando lo stato con rviz per più di uno. Alle righe 13 e 17 si può osservare l'uso del remapping per reindirizzare i topic, nella fattispecie `joint_state_publisher` e `robot_state_publisher` utilizzano entrambi lo stesso topic `joint_states`, il primo come publisher il secondo come subscriber. Non rimappare i topic comporterebbe per `robot_state_publisher` la ricezione dei messaggi di `joint_state_publisher`, con il remapping invece viene creato un topic per le comunicazioni tra `joint_state_publisher` e `node_sample`, chiamato `joint_states_in`, e uno per le comunicazioni tra `node_sample` e `robot_state_publisher` chiamato `joint_states_out` risolvendo ogni ambiguità. In figura 5.3 è riportato l'aspetto dell'ambiente di debugging: si può osservare l'interfaccia di controllo dei giunti tramite slider, e le opzioni disponibili da linea di comando. In figura 5.4 è riportato invece la rappresentazione del Computation Graph ottenuta con *rxgraph* per un robot caricato, infine in figura 5.5 la stessa rappresentazione per tre robot caricati. E' possibile osservare come il numero di connessioni topic (in questo caso tutte sottoscritte dai nodi di controllo) aumenti considerevolmente e come apparentemente il nodo `simspark_virtual_driver` centralizzi l'interfacciamento. La semplice rappresentazione di *rxgraph* nasconde la complessità interna del nodo.

5.2 La libreria KDL

La libreria KDL (Kinematics and Dynamics Library) è originariamente parte del framework Orocos, interfacciata in ROS attraverso package `orocos_kdl`, per C++, e `python_orocos_kdl`, per python. La libreria può essere utilizzata: per effettuare trasformazioni si vettori, punti e sistemi di riferimento; per la risoluzione di cinematica e dinamica diretta e inversa di catene cinematiche e alberi cinematici. Per maggiori dettagli si rimanda alla documentazione riportata nel home page della libreria [S20].

Per ottenere la rappresentazione KDL dal modello URDF è possibile utilizzare le funzionalità del package `kdl_parser`:

```
#include <kdl_parser/kdl_parser.hpp>

...
KDL::Tree my_tree;
ros::NodeHandle node;
std::string robot_desc_string;
```

```
<launch>
  <arg name="urdf_model" default="$(find robovie_x_model)/model.urdf"/>
  <arg name="gui" default="True" />
  <arg name="rviz" default="False" />
  <arg name="robot_name" default="DEIbot" />
  <arg name="robot_model" default="robovie_x" />
  <arg name="node_name" default="node_sample" />

  <param name="$(arg node_name)/robot_description" textfile="$(arg
      urdf_model)" />
  <param name="$(arg node_name)/use_gui" value="$(arg gui)"/>

  <node if="$(arg gui)" name="joint_state_publisher" pkg="
      joint_state_publisher" type="joint_state_publisher" ns="$(arg
      node_name)">
    <remap from="joint_states" to="joint_states_in"/>
  </node>

  <node if="$(arg rviz)" name="robot_state_publisher" pkg="
      robot_state_publisher" type="state_publisher" ns="$(arg node_name
      )">
    <remap from="joint_states" to="joint_states_out"/>
  </node>

  <node if="$(arg rviz)" name="rviz" pkg="rviz" type="rviz" args="-d $(
      find robovie_x_model)/urdf.vcg" ns="$(arg node_name)"/>

  <node name="$(arg node_name)" pkg="robot_control" type="node_sample"
      output="screen" required="true" args="" />
</launch>
```

Listato 5.1: launch file per l'applicazione di debugging

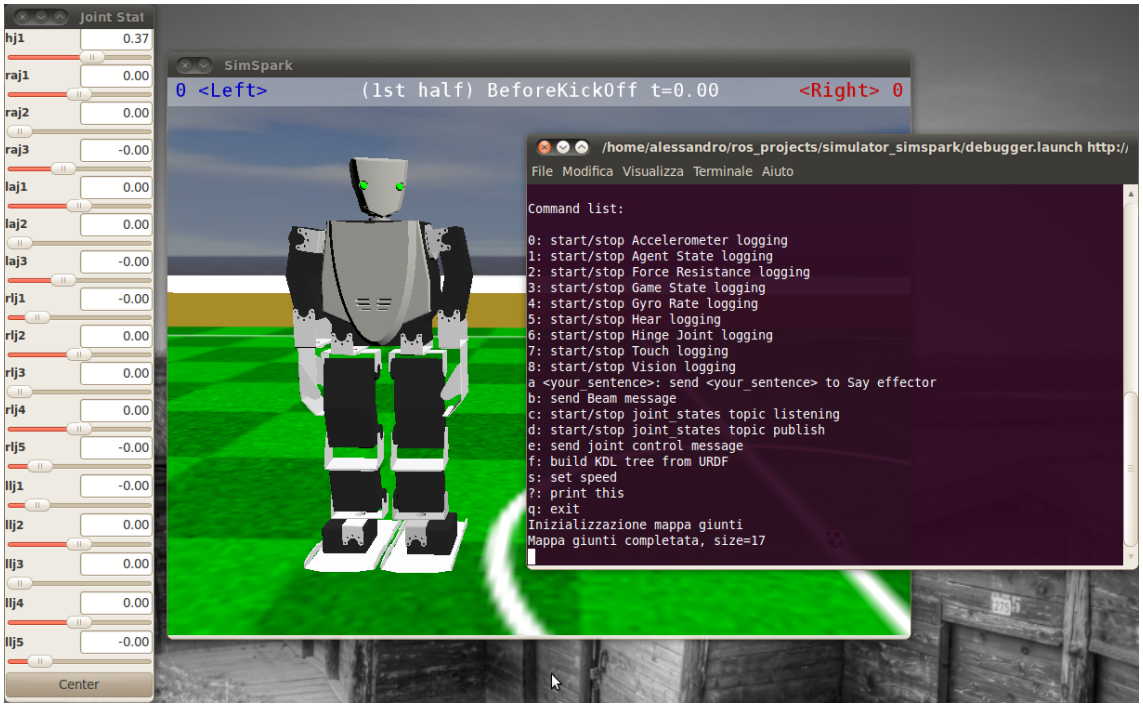


Figura 5.3: Ambiente di lavoro di debugging

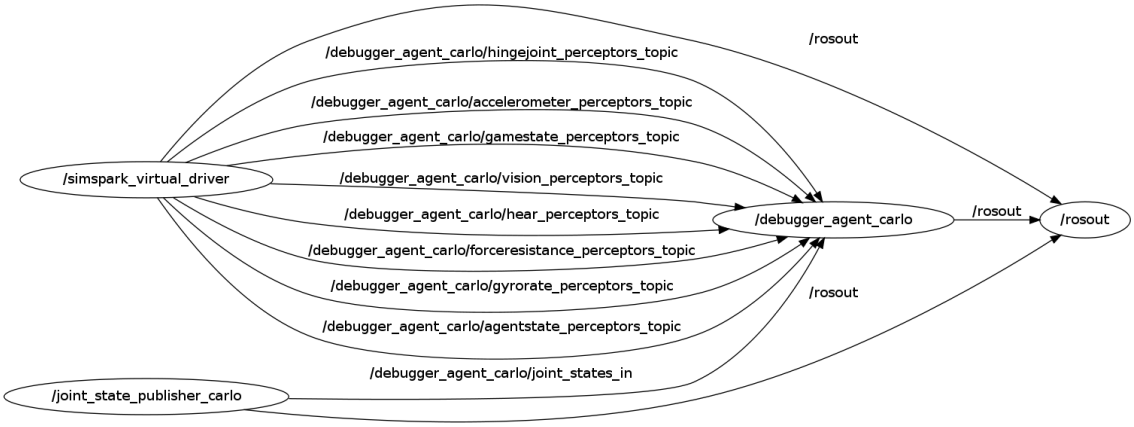


Figura 5.4: Rappresentazione del Computation Graph con un solo robot simulato

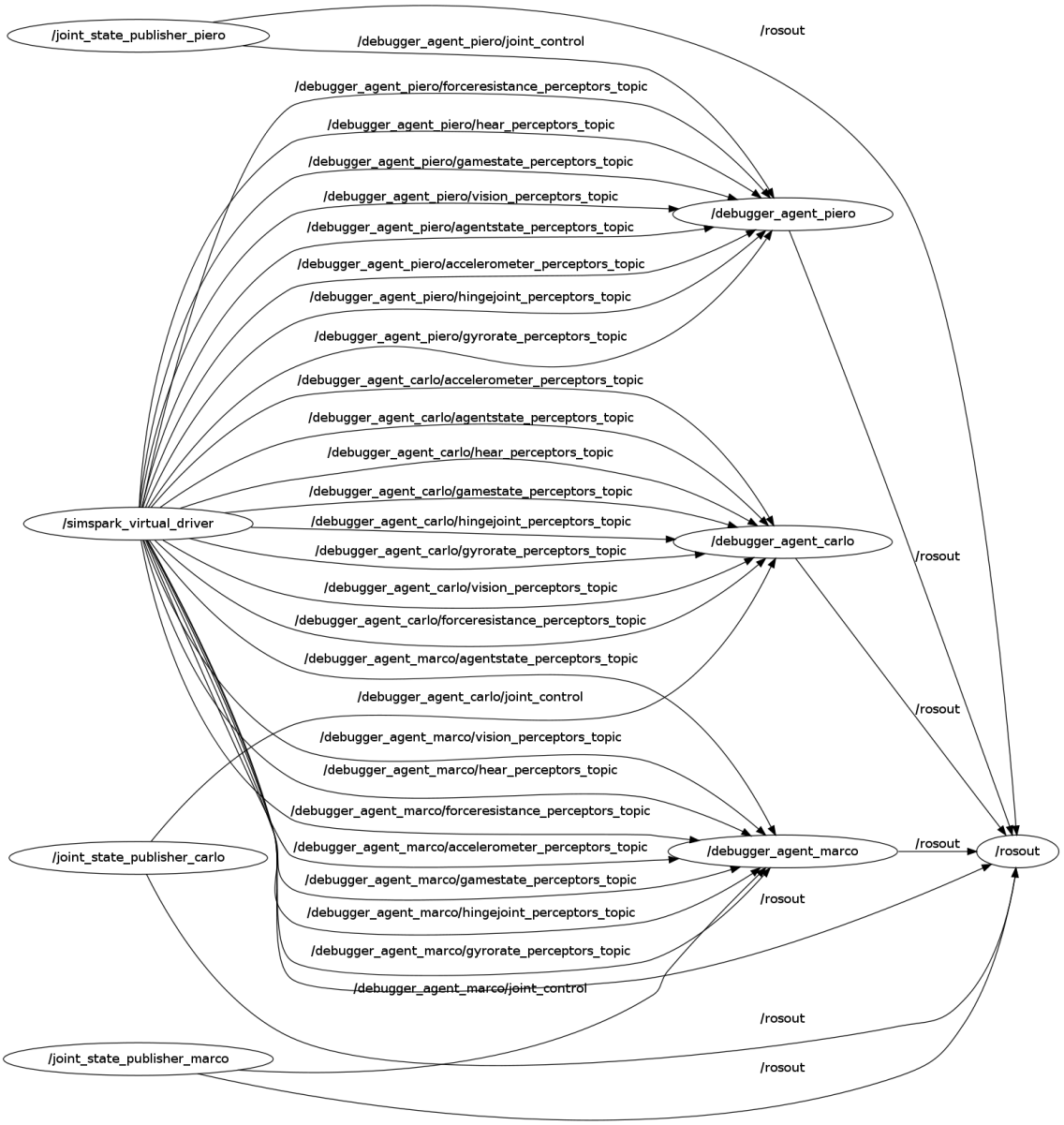


Figura 5.5: Rappresentazione del Computation Graph con un tre robot simulati

```
node.param("robot_description", robot_desc_string, string());
kdl_parser::treeFromString(robot_desc_string, my_tree);
...
```

La funzione `treeFromString` consente di ottenere la rappresentazione `KDL::Tree` a partire dalla descrizione URDF o COLLADA del robot, in questo caso passata come stringa.

Capitolo 6

Il robot umanoide Robovie-X

In questo capitolo si riportano i dettagli del modello 3D del robot Robovie-X. Il robot è prodotto dalla azienda giapponese Vstone ed è disponibile nelle versioni lite, standard e pro, che differiscono per numero di gradi di libertà. La versione qui considerata è quella standard, che presenta un totale di 17 DOF: uno per la testa, sei per le braccia, dieci per le gambe. Il modello 3D è stato realizzato dallo studente di dottorato Fuben He, in visita al dipartimento di ingegneria informatica dell'università di Padova. Il modello originale è stato sviluppato con il software di modellazione Catia e successivamente esportato in file OBJ Wavefront, separandone le parti corrispondenti ai link in file distinti. Il modello è stato successivamente modificato con il software di modellazione Blender, per adattarlo ai formati e alle convenzioni geometriche utilizzate dalla descrizione URDF ROS e SRG SimSpark:

- gli elementi (i link del robot), originariamente centrati sul centro dei giunti, sono stati ricentrati nel centro del proprio bounding box. Ciò ha consentito la specifica più semplice delle primitive di collisione e di inerzia nelle descrizioni URDF e RSG. In particolare, come riportato nel paragrafo 1.5.1, RSG non consente la specifica di un'origine per le primitive di collisione differente da quella del link; è pertanto fondamentale che le mesh siano centrate nel proprio box di ingombro

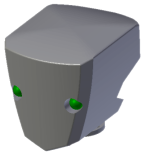
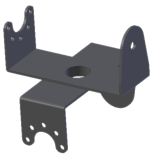
- il modello è stato scalato di un fattore 1/100 per renderlo compatibile alle unità di misura di Blender e alle dimensioni dell'ambiente di simulazione

- il modello è stato colorato con Blender rispettando la colorazione del robot reale. Il modello originale infatti non esportava i materiali applicati al robot, che risultava quindi completamente grigio. Le mesh sono quindi state salvate

in formati file che mantenessero l'informazione dei materiali applicati: 3DS per ROS, OBJ + MTL (Material Template Library) Wavefront per SimSpark.

Per realizzare le descrizioni URDF e RSG del robot Robovie-X è stato poi necessario misurare la posizione relativa di ogni link rispetto il link padre, ad esempio: la posizione della testa, delle spalle e dell'anca rispetto il busto, la posizione del braccio rispetto la spalla, e così via. Mancando un modello completamente assemblato è stato necessario ricomporre il robot a partire dai link separati, misurandone di volta in volta la posizione relativa. In tabella 6.2 sono riportate le posizioni dei link dei giunti rispetto il link padre nel modello scalato. Nel caso di parti simmetriche, come braccia e gambe, le posizioni sono relative alle parti destre del corpo. Nelle colonne *Dim X*, *Dim Y* e *Dim Z* si riportano le dimensioni del bounding box di ciascun link, utilizzate per dimensionare le primitive di collisione del modello. *LO X*, *LO Y* e *LO Z* rappresentano l'offset dell'origine della mesh del link rispetto l'origine del link stesso (Link Offset), che per convenzione viene fatto coincidere con la posizione del giunto che collega il link con il link padre nella catena cinematica. L'elemento torso è la radice dell'albero cinematico, quindi la sua mesh è centrata sull'origine del link. *JO X*, *JO Y* e*JO Z* indicano invece l'offset dell'origine del giunto che collega il link al link padre nella catena cinematica, rispetto l'origine del link stesso. Infine *Rot X*, *Rot Y* e *Rot Z* riportano la rotazione da applicare a ciascuna mesh per far combaciare i link.

Per la modellazione dell'inerzia del robot è stata fatta una valutazione della massa delle sue parti, a partire dalla massa delle sue componenti. In tabella 6.1 sono riportate le masse approssimate dei link del modello. La massa totale del robot risulta leggermente inferiore rispetto le specifiche, pari a circa 1300g per il robot con batteria, differenza dovuta probabilmente al mancato conteggio delle viti. Infine nelle tabelle 6.1 e 6.2 si riportano alcune immagini del modello mesh assemblato e il modello a primitive utilizzato per il controllo delle collisioni.

Link	massa [g]	Mesh
Torso	426,5	
Head	60	
Shoulder	7	
Arm	88,5	
Hand	21	
Hip	13	

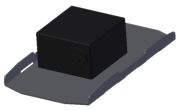
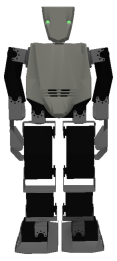
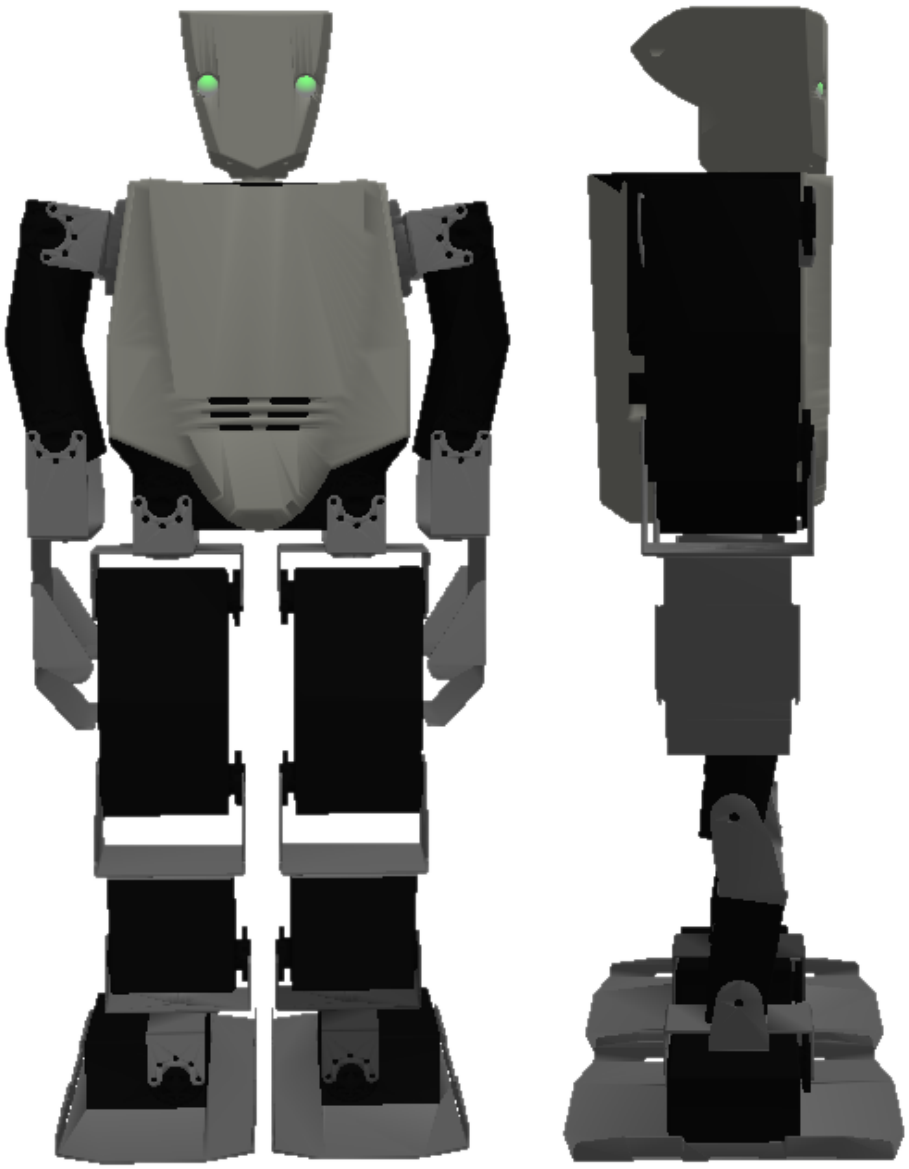
Link	massa [g]	Mesh
Leg	88,5	
Knee	56	
Ankle	13	
Foot	77	
Totale	1214,5	

Tabella 6.1: Massa assegnata ai link del modello

Link	Dim X	Dim Y	Dim Z	LO relativo a	LO X	LO Y	LO Z	JO X	JO Y	JO Z	Rot X [°]	Rot Y [°]	Rot Z [°]
Torso	0,7	1,02	0,92	-	-	-	-	-	-	-	90	0	0
Head	0,595	0,505	0,439	Torso	0,052692	0	0,7525	0,052692	0	0,46	90	0	0
Right Shoulder	0,47	0,26	0,291	Torso	0,05	-0,54176	0,351895	0,05	-0,54176	0,351895	-9	0	0
Right Arm	0,253	0,797	0,46	Shoulder	0,005	-0,08209	-0,28515	0,005	-0,0593	0,00939	90	0	-90
Right Hand	0,47	0,26	0,291	Torso	0,05	-0,54176	0,351895	0,05	-0,54176	0,351895	-9	0	0
Right Hip	0,47	0,46	0,47	Torso	0,065	-0,29	-0,58005	0,065	-0,29	-0,42	90	0	90
Right Leg	0,253	0,797	0,46	Hip	0,0228	0,005	-0,45445	0	0,005	-0,15995	90	0	0
Right Knee	0,247	0,76	0,485	Leg	0,0131	0,0025	-0,6037	-0,02235	-0,005	-0,294	90	0	0
Right Ankle	0,47	0,46	0,47	Knee	-0,0221	0,0075	-0,43855	-0,0221	0,0075	-0,27855	90	0	0
Right Foot	1,05	0,24	0,6	Ankle	-0,015	-0,05	-0,169954	-0,015	0	-0,159954	90	0	0

Tabella 6.2: Distanze relative tra link e giunti del modello Robovie-X



(a) Vista frontale (b) Vista laterale

Figura 6.1: Immagini del modello

(a) Mesh per la visualizzazione

(b) Primitive di collisione

Figura 6.2: Immagini del modello

Capitolo 7

Aspetti di applicabilità industriale

ROS è un sistema nato per la robotica sperimentale, confinato inizialmente a centri di ricerca e università è tuttora in fase di forte sviluppo. Per trattare problematiche di applicabilità di ROS alla robotica industriale bisogna innanzi tutto partire dalle sue caratteristiche come RSS per robotica sperimentale, evidenziandone pregi e difetti rispetto alle comuni esigenze di software engineering in questo ambito. Chiarite le criticità generali di ROS sarà possibile affrontare gli aspetti specifici della robotica industriale.

Questo capitolo è organizzato pertanto in quattro parti:

1. analisi delle esigenze di software engineering nella robotica sperimentale e confronto con le caratteristiche offerte da ROS

2. analisi delle esigenze specifiche della robotica industriale e confronto con le caratteristiche offerte da ROS

3. raccolta delle criticità individuate e conclusioni

4. approfondimento sull'utilizzo di ROS nei robot industriali Motoman

7.1 Esigenze di software engineering nella robotica sperimentale

Le esigenze di software engineering nella robotica sperimentale sono perlopiù legate a problematiche di flessibilità, portabilità, interoperabilità, e scalabilità del sistema. Una risposta a tali esigenze viene fornita dai Robotics Software System

(component-based), che offrono funzionalità a supporto della *computazione*, *comunicazione*, *coordinazione* e *configurazione* nel sistema.

Per analizzare le caratteristiche di ROS come RSS si è scelto di procedere con due approcci: un'analisi strutturale e una funzionale. Nell'analisi strutturale verrà utilizzato l'approccio proposto da [12] che consiste nel valutare un RSS in termini del grado di separazione e indipendenza delle sue componenti di framework e communication middleware da quelle puramente algoritmico-implementative e supporto driver. L'analisi funzionale valuterà l'adeguatezza di ROS alle esigenze di software engineering nella robotica sperimentale. Sulla base di ciascuna analisi saranno evidenziati punti di forza e criticità di ROS.

7.1.1 Analisi strutturale di ROS

Per analizzare le funzionalità di ROS come Robotics Software System si è scelto do utilizzare l'approccio proposto da [12], che scompone un RSS in tre componenti software:

1. Componente algoritmica e driver di supporto (DA)

2. Funzionalità di comunicazione o communication middleware (CM)

3. Framework di sviluppo software o robotic software framework (RSF)

Per RSF si intende il modo in cui il RSS definisce e implementa la modularità e gli strumenti che mette a disposizione per la gestione del sistema. Ogni RSS possiede un proprio RSF ed è proprio la componente software legata al framework che limita la portabilità del codice tra framework e RSS differenti.

Un communication middleware è presente in RSS distribuiti e può essere sviluppato in maniera personalizzata o può essere realizzato a partire da una tecnologia esistente, come CORBA o ACE. I CM custom sono spesso fortemente interlacciati con il framework, limitandone così la riutilizzabilità, a vantaggio solitamente di una maggiore semplicità.

Infine vi è la componente di codice, principalmente di due tipologie: implementazione di algoritmi e driver per l'acquisizione da sensori, il controllo di attuatori, l'interfacciamento con robot.

Secondo [12] un buon RSS dovrebbe mantenere il più possibile distinte le componenti DA, CM e RSF per favorire interoperabilità, maggior riutilizzabilità e portabilità del codice tra sistemi RSS differenti. La realtà dei fatti mostra invece come tutti

i più diffusi RSS, ROS compreso, tendano a fondere questi aspetti. Le conseguenze che ne derivano sono:

- limitata portabilità dei moduli al di fuori del RSS
- limitata interoperabilità tra sistemi differenti, legata all'utilizzo di diversi CM
- limitata possibilità di confronto tra RSS, legata alla difficoltà di testare i medesimi algoritmi su sistemi differenti

A causa di questo limite, i criteri che guidano la scelta tra diversi RSS si riduce spesso al confronto tra il numero e la tipologia di funzionalità, driver e algoritmi, forniti.

Effettuando l'analisi della struttura di ROS come RSS costituito dalle parti DA, RSF e CM ci si accorge di come alcune componenti del sistema mescolino questi concetti, in particolare DA+CM e RSF+CM, ma siano comunque implementate forme di indipendenza della componente algoritmica:

RSF - ROS come framework. Per quanto riguarda la gestione e l'implementazione della modularità ROS supporta tre tipologie di componenti del sistema:

- **nodi**: presentati nel paragrafo 2.3, costituiscono moduli eseguibili connessi alla rete ROS. I nodi sono tuttavia anche gli elementi base per la comunicazione in ROS, forniscono service e interagiscono con i topic, infine sono utilizzati per l'implementazione di unità funzionali, implementando driver, algoritmi, interfacce per i simulatori, ecc. A tutti gli effetti quindi un nodo ROS racchiude caratteristiche RSF, CM e DA.
- **librerie**: ROS consente la creazione di librerie attraverso gli stessi meccanismi disponibili per i nodi. Le librerie sono sviluppate all'interno di package, in questo modo ROS ne fornisce la gestione delle dipendenze e la distribuzione attraverso i repository. L'unica differenza tra lo sviluppo di una libreria e lo sviluppo di un nodo ROS sta in una riga del file CMakeLists.txt del package, in cui si specifica la creazione dell'uno o dell'altro. L'utilizzo della libreria avviene altrettanto semplicemente: è sufficiente specificarne la dipendenza nel file manifest del package, quindi includerne gli header e specificare il linking in CMakeLists. La gestione delle librerie in ROS rappresenta un aspetto di puro RSF.
- **plugin**: citati nel paragrafo 2.7, costituiscono funzionalità di alto livello in ROS per la distribuzione runtime di istanze di oggetto. Anche i plugin

sono realizzati coinvolgendo solo aspetti di RSF: l'implementazione fa uso delle funzionalità di class factory delle librerie C++ POCO [S21], integrata con le specifiche ROS tramite file manifest; come nodi e librerie, anche i plugin vanno sviluppati all'interno di package, usufruendo quindi della gestione delle dipendenze e ridistribuzione ROS. Non sono coinvolte funzionalità di comunicazione poichè la risoluzione delle dipendenze è svolta localmente.

Le altre caratteristiche di ROS come framework sono:

- visualizzazione 3D dello stato dei robot, delle mappe, dei dati da sensori, forniti dallo strumento *rviz*, presentato nel paragrafo 2.7
- logging e visualizzazione di dati sensoriali registrati, implementati attraverso i ROS bag, presentati nel paragrafo 2.3
- plotting runtime di dati scalari letti da topic. La funzionalità è implementata attraverso lo strumento *rxplot*
- visualizzazione del ros computation graph, nodi caricati e topic pubblicati, attraverso lo strumento *rxgraph*
- esecuzione del sistema: avvio del Master, il lancio di nodi singoli, la creazione ed il lancio di configurazioni complesse di nodi con *roslauch*, ecc.
- interazione, introspezione e debugging del sistema: *rostopic* per la visualizzazione di informazioni sui topic, ricezione e pubblicazione di messaggi; *rosservice* per la visualizzazione di informazioni sui service e funzionalità a servizio manuale; *rosnode* per ricevere informazioni sui nodi attivi, verificarne la connettività, terminarli; *rosparam* per visualizzare i parametri del sistema, salvarli su file, caricarli da file, aggiungerne nuovi o rimuoverne; *rosmsg* è uno strumento per l'introspezione del formato dei messaggi scambiati; *roswtf* effettua l'analisi dello stato del sistema cercando e riportando eventuali errori o anomalie
- gestione del filesystem ROS: creazione package e stack, compilazione, navigazione, strumenti introdotti nel paragrafo 2.5.

Riportando le funzionalità presentate nel grafico in figura 7.1, che ne mostra le componenti DA, RSF e CM, è possibile osservare gran come parte delle caratteristiche framework di ROS mescolino aspetti propri del CM e DA. Fortunatamente viene fornito il supporto a librerie e plugin in una forma indipendente

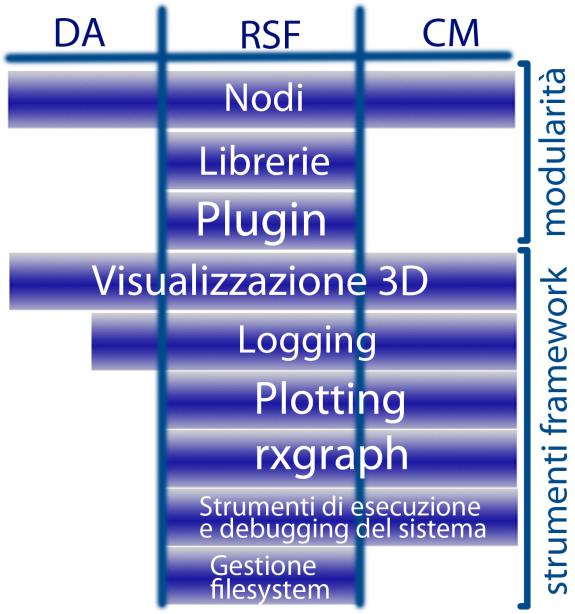


Figura 7.1: Distribuzione delle funzionalità del framework nelle componenti DA, RSF e CM

dal CM, consentendo quindi l'utilizzo di software di terze parti. Esempi di applicazione immediata di questa possibilità sono l'integrazione della libreria di visione *openCV*, sviluppata anch'essa da Willow Garage, e le librerie Orocos. La fusione di aspetti di RSF con DA e CM non risulta quindi limitante.

CM - ROS come communication middleware. ROS implementa un communication middleware custom, presentato nei paragrafi 2.3 e 2.4, basato su un mix di protocolli personalizzati (XMLRPC, TCPROS, UDPROS) sulla base dei quali sono fornite tipologie di comunicazione RPC e streaming asincrono. L'utilizzo di un CM custom si contrappone alla scelta adottata da altri RSS di fare uso di strumenti preesistenti, sviluppati nell'ambito dei sistemi distribuiti, come CORBA e ICE. Le conseguenze di questa scelta determinano vantaggi e svantaggi:

PRO

- è possibile ridurre l'implementazione alle sole funzionalità di interesse, consentendo la realizzazione di un sistema più snello e leggero, adatto all'esecuzione in contesti embedded, nonché una notevole semplificazione nell'utilizzo
- non esistendo uno standard universale per i CM non vi è *una* scelta corretta nel supporto a CM come CORBA o ICE e in ogni caso ciò

non comporterebbe necessariamente l'interoperabilità con altri RSS

Alla luce di questi fatti emerge che in ogni caso risulta necessaria la creazione di specifici adattamenti per l'interazione con altri sistemi, condizione gestibile più facilmente in presenza di sistemi semplici e di rapida implementazione. Ciò ha consentito ad esempio l'interoperabilità di ROS con OpenRTM-aist [13], implementata effettuando una traduzione di basso livello tra i protocolli di trasporto dei due RSS, con OpenRAVE e URBI.

CONTRO
- mancato interfacciamento diretto con altri RSS
- necessità di reinventare tecniche di comunicazione già esistenti e rodate
- necessità di reinventare tecniche di gestione della comunicazione già esistenti e rodate: QoS, supporto real-time, coordinazione e gestione concorrenza

Il CM in ROS risulta strettamente legato al framework: service e topic sono associati al formato dei dati scambiati la cui specifica e traduzione in codice sorgente avviene per mezzo delle funzionalità del framework. Senza il framework la gestione del middleware sarebbe estremamente complessa. I nodi costituiscono poi l'elemento principale per l'interazione tra i componenti ROS attraverso il CM e come visto racchiudono aspetti propri del framework e incapsulano elementi di DA.
Questa dipendenza rende ROS incompatibile al supporto diretto di altri CM, consentendo al più la realizzazione di wrapper che, ponendosi nel mezzo dei due sistemi, effettuino la conversione dei protocolli.

DA - Le librerie, i driver e gli algoritmi in ROS . ROS fornisce un grande repository di API, distinguendosi rispetto altri RSS per l'offerta di implementazioni dei più comuni algoritmi utilizzati in robotica. Le principali funzionalità offerte sono classificabili nelle categorie riportate in tabella 7.1. Le colonne rappresentano il tipo di modularità implementata:

ROS : come nodi interfacciati tramite service o topic, fruibili dal Computation Graph. Funzionalità di questo tipo mescolano concetti di DA, RSF e CM

Libreria (C++ o Python) : come libreria communication independent. L'implementazione effettuata determina il grado di "purezza" del codice da

TIPO	FUNZIONALITA'	ROS	C++	Python
Driver	Interfacce	✓	✓	
	Implementazioni	✓	✓	
Supporto	Basic Datatype	✓	✓	✓
	Manipulating message stream	✓	✓	✓
	Actions	✓	✓	✓
	Executive/Task Manager (Coordination)	✓		✓
	Robot Model		✓	
Algoritmi	Filtering data		✓	
	3D processing	✓	✓	
	Image processing		✓	✓
	Transforms/Coordinates		✓	✓
	Navigation	✓	✓*	✓*
	2D-3D simulation	✓	✓	
	Realtime Controller	✓	✓	
	Motion planning	✓	✓*	✓*
Third-party	Orocos KDL		✓	✓
	OpenRAVE manipulation planning	✓	✓	
	OpenSlam	✓		

Tabella 7.1: Funzionalità DA offerte da ROS

aspetti prettamente RSF. I componenti riportati con * sono implementanti attraverso *ROS actionlib* (paragrafo 2.7), pertanto mescolano concetti di CM e framework.

Molte componenti DA presentano sia interfacce ROS che API da libreria, consentendo quindi la più ampia libertà implementativa, indipendenza dal framework e dal middleware.

Alla luce della caratterizzazione presentata ROS risulta un RSS adeguatamente strutturato, consentendo forme di indipendenza del codice dal framework e da aspetti prettamente comunicativi. La criticità maggiore emersa dall'analisi è la stretta interdipendenza tra framework e communication middleware da cui derivano difficoltà nell'estensione del supporto ad altri CM in ROS. Una possibile soluzione potrebbe essere l'introduzione di un nuovo livello di astrazione tra le interfacce di comunicazione ROS e l'implementazione utilizzata, esplicitando la struttura di communication middleware esistente ma slacciandola da spetti prettamente pertinenti al RSF.

7.1.2 Analisi funzionale di ROS

La seconda analisi che si intende effettuare è relativa all'adeguatezza delle funzionalità offerte da ROS. Sono stati realizzati molti lavori [7][8][9][14][15][16] volti a definire quali siano le caratteristiche che un RSS completo dovrebbe offrire. Considerando l'unione delle caratteristiche salienti si ottiene quanto riportato di seguito. Le funzionalità sono state raggruppate per contesto: G = caratteristiche generali del sistema RSS, F = caratteristiche del framework, M = caratteristiche del communication middleware, H = caratteristiche di alto livello.

Per quanto riguarda le funzionalità generali che un RSS dovrebbe offrire, si trovano:

G1 **OpenSource.** L'essenza opensource è fondamentale per la collaborazione attiva della comunità robotica, nonchè per la collaborazione tra istituzioni accademiche. Velocizza il processo di sviluppo, di testing e di debugging del RSS. E' però importante che si eserciti una forma di controllo e coordinazione nell'organizzazione del software per evitare la proliferazione di componenti di scarsa qualità, non testati, mal documentati o incompleti.

G2 **Qualità.** E' necessario esercitare una forma di controllo della qualità dei componenti del sistema, in particolare per quanto specificato nel punto precedente.

G3 **Repository online.** E' importante che il codice sia raccolto opportunamente in repository online per consentire rapido accesso alle versioni più recenti del software. Analogamente è importante la presenza di una documentazione adeguata dei componenti, possibilmente uniforme per tutto il sistema.

ROS soddisfa parzialmente queste esigenze: è opensource ed offre repository online per la raccolta e ridistribuzione del codice. Promuove lo sviluppo della documentazione attraverso una forma di wiki centralizzata, fornendo strumenti per la realizzazione di tutorial, inclusione di codice sorgente, di immagini e video. Per la discussione e la risoluzione dei problemi sono disponibili diverse forme di supporto: mailing-list, ROS Answer (una forma di forum Question&Answer), richiesta di assistenza tramite Ticket. Ciò che invece manca totalmente è una forma di valutazione della qualità del software. Non è raro ad esempio trovare package completamente privi di documentazione, senza riferimento al grado di completamento, test, funzionalità. Mancano indicazioni uniformi per la specifica del grado di stabilità.

Sarebbe interessante dotare la wiki di strumenti per la specifica uniforme delle caratteristiche di qualità dei package, offrendo magari agli utenti la possibilità di rilasciare giudizi. Andrebbe anche strutturato meglio l'elenco dei package e stack disponibili, suddividendoli per aree funzionali.

Le caratteristiche richieste al framework sono invece:

F1 **Indipendenza da middleware.** Le ragioni sono state esposte nel paragrafo precedente.

F2 **Linguaggi di programmazione, compilatori.** Non esiste un linguaggio di programmazione ideale ed ognuno vanta caratteristiche che lo rende ideale in determinate condizioni. Per questo un RSF dovrebbe supportare lo sviluppo in diversi linguaggi di programmazione mantenendo uniformità nelle interfacce. Sarebbe opportuno lo sviluppo di nuovi linguaggi di programmazione per avere maggior controllo sul parallelismo e gestione degli eventi, nonchè disporre di forme di programmazione visuale che semplifichino la creazione, la programmazione e l'interconnessione dei componenti.

F3 **Portabilità.** Legata al grado di indipendenza del codice dal framework e dal middleware che è possibile ottenere, al fine di utilizzare i moduli in altri RSS.

F4 **Architecture agnostic.** L'RSS non deve vincola lo sviluppo del sistema ad una particolare tipologia architetturale, viceversa l'architettura deve emergere

dalla composizione dei moduli, secondo le esigenze del sistema o la scelta dello sviluppatore.

F5 **Estensibilità.** L'RSS deve fornire strumenti per la creazione di nuovi componenti del sistema.

F6 **Riutilizzabilità.** Il sistema deve fornire un grado di astrazione adeguato che consenta lo sviluppo di software riutilizzabile in contesti differenti da quelli originari: diverso sistema operativo, diversa connettività, diversi componenti di interazione.

F7 **Supportare computazione real-time.** La robotica è caratterizzata da evidenti esigenze real-time, soprattutto per quanto riguarda le funzionalità di controllo a basso livello. Il framework pertanto deve supportare lo sviluppo di moduli real-time, gestendo forme di priorità tra i task coinvolti.

Come analizzato nel paragrafo precedente, ROS non fornisce indipendenza dal communication middleware. Il supporto a diversi linguaggi di programmazione è invece fornito dall'utilizzo dei client library. Recentemente è stato sviluppato il supporto al framework URBI consentendo anche l'utilizzo del linguaggio UrbiScript in ROS, un linguaggio orchestration script con funzionalità di alto livello, come gestione del parallelismo ed event-based programming, integrate nella semantica del linguaggio. Come mostrato precedentemente ROS fornisce meccanismi per garantire portabilità del codice attraverso l'uso di librerie. L'architettura del sistema non è in alcun modo vincolata da ROS, e dipende da come si effettua l'interconnessione dei nodi e l'organizzazione dei namespace. Per quanto riguarda l'estensibilità, il framework ROS è ricco di strumenti da linea di comando per la creazione e la ridistribuzione di nuovi package e stack, mentre la riutilizzabilità è coadiuvata dalle tecniche di parametrizzazione e gestione del naming offerte.

Un aspetto totalmente non considerato dall'ambiente ROS (nativo) è quello del supporto real-time. ROS offre tuttavia interoperabilità con il framework Orocos Toolchain e Orocos RTT (Real-Time Toolkit) per il controllo e la comunicazione real-time di basso livello (non ROS based). Sono disponibili poi alcune funzionalità nello stack `pr2_mechanism` per il controllo del robot PR2 in un loop hard-realtime, secondo la documentazione portabili ad altri robot.

Per quanto riguarda gli aspetti di comunicazione, le esigenze sono:

M1 **Scalabilità.** Il communication middleware non deve dipendere da alcuna entità centralizzata

M2 **Interoperabilità.** Il sistema deve essere interfacciabile con altri middleware per garantire interoperabilità con altri RSS.

M3 **Comunicazione real-time.** Analogamente alla computazione real-time, determinate applicazioni richiedono comunicazione real-time e supporto a reti real-time.

M4 **Web Service.** Il supporto ai web service garantirebbe forme di integrazione avanzata di componenti robotiche connesse alla rete, fornendo comunicazione flessibile e adattabilità a configurazioni che variano dinamicamente.

Come visto nel paragrafo precedente, probabilmente il punto più debole di ROS è il suo communication middleware. La sua semplicità favorisce interoperabilità con altri sistemi consentendo la creazione di componenti di interfacciamento tra i sistemi. L'utilizzo di un middleware custom tuttavia esclude ogni possibilità di interoperabilità diretta. La rete ROS si basa sull'esecuzione di un processo Master con funzioni di name resolution, resource lookup, callback: anche quando la rete è distribuita su più macchine tale processo deve essere mantenuto centralizzato e il suo URI impostato manualmente su tutte le macchine. Questa condizione può costituire un limite alla crescita del Computation Graph e all'efficienza del sistema, soprattutto in un contesto multirobot, anche se non sono riportati test in tal senso. E' importante osservare come questa esigenza sia stata recepita da Willow Garage: il supporto multimaster è in fase di sviluppo è una prima versione sarà forse disponibile nella prossima release (ROS Fuerte, prevista per Marzo 2012). Analogamente al mancato supporto real-time nella computazione non vi è supporto real-time nella comunicazione, se non forme di pubblicazione/sottoscrizione di topic che non interrompono il real-time behavior del processo. Se per la computazione real-time si può sfruttare l'interoperabilità con altri framework, per introdurre comunicazione real-time in ROS è necessario lavorare allo sviluppo del communication middleware. Una differente strategia potrebbe essere quella di svincolare il framework dal CM, introducendo il supporto ad un middleware real-time.
Per quanto riguarda infine la struttura service-oriented c'è da dire che quasi nessun RSS implementa la modularità tramite Web Service, fatta eccezione ad esempio per Microsoft Robotics Developer Studio, prediligendo in genere una più semplice architettura component-based.

Si evidenziano infine alcune caratteristiche di alto livello che un RSS dovrebbe fornire per facilitare lo sviluppo software nei moderni sistemi robotici:

H1 **Adattabilità, Riconfigurabilità dinamica.** Un RSS dovrebbe fornire strumenti per la parametrizzazione e la configurazione offline e online del sistema, caratteristica fondamentale per supportare sistemi dinamici come quelli multirobot. Ciò implica la possibilità di agire su tre livelli di configurazione:

(a) *di sistema*: capacità di configurare o riconfigurare applicazioni senza necessità di interromperne l'esecuzione del sistema. Deve quindi essere possibile fermare e riavviare componenti in maniera dinamica senza interferire con l'attività del robot.

(b) *di applicazione*: capacità di riconfigurare applicazioni senza accedere al codice o ricompilare per modificarne run-time le caratteristiche o sostituire componenti non funzionanti.

(c) *di componente*: capacità di configurare le proprietà di singoli componenti del sistema e selezionare dinamicamente diverse possibili implementazioni (es. diversi algoritmi di motion planning)

H2 **Strumenti di coordinazione.** La coordinazione si riferisce alla gestione e all'ordinamento distribuito delle interazioni tra componenti del sistema anche su più livelli. La struttura da implementare deve poter essere quella di un sistema-di-sistemi.

H3 **Individuazione e configurazione automatica delle risorse.** I sistemi robotici sono caratterizzati da elevata dinamicità: dispositivi possono essere connessi e disconnessi in maniera dinamica causando una variazione del numero e del tipo di risorse disponibili. L'RSS deve pertanto fornire meccanismi di autoadattamento e auto-ottimizzazione del sistema. Nel contesto di sistemi distribuiti vi è la presenza di macchine con differenti capacità e potenza computazionale, l'RSS dovrebbe non solo fornire strumenti per distribuire l'esecuzione sulle diverse macchine, ma essere in grado di spostare la computazione al variare del carico del sistema.

H4 **Supporto alla collaborazione.** Le problematiche di collaborazione e cooperazione tra robot e uomo-robot sono una tematica di forte interesse in robotica. L'RSS dovrebbe fornire funzionalità ad alto livello di astrazione per sviluppare tali meccanismi di collaborazione.

ROS dispone di alcune funzionalità di alto livello, presentante nel paragrafo 2.7. Offre un alto livello di configurabilità:

- a livello di singolo nodo, utilizzando parametri globali o privati, è possibile implementare la modifica dinamica del behavior del processo. Attraverso l'utilizzo di plugin è possibile caricare dinamicamente differenti implementazioni di algoritmi all'interno dello stesso nodo.

- a livello di *applicazione*, intesa come sottosistema di nodi interagenti, sono forniti meccanismi di configurazione offline come *roslaunch* per la specifica dei componenti e dei parametri di ciascun componente.

- a livello di sistema ROS offre tutti i meccanismi per il supporto alla dinamicità, aggiunta, rimozione, sostituzione di componenti senza compromettere l'esecuzione del sistema. Tali funzionalità non sono automatiche, ma dipendono dalla corretta implementazione dei meccanismi comunicativi.

Per quanto riguarda la coordinazione ROS non implementa forme di controllo rigido del sistema, in virtù del legame loosely coupled tra i componenti. Attraverso *actionlib* è possibile avere un maggior coordinamento esercitando forme di controllo sulle azioni intraprese. Per il coordinamento in contesti ben definiti e strutturati è possibile sfruttare la libreria SMACH (State MACHine), realizzata attraverso actionlib, che consente un coordinamento rigido tramite macchina a stati finiti. Altre forme di coordinamento, in contesti non strutturati o di basso livello, devono essere implementanti in altro modo.

In ROS non esiste una rappresentazione esplicita di *risorsa*. L'individuazione automatica può essere implementata nel senso di collegamento dinamico di componenti, la dinamicità può essere implementata attraverso la comunicazione loosely coupled offerta e l'auto configurabilità attraverso la modifica dinamica dei parametri del sistema. E' possibile distribuire la computazione tra le macchine del sistema, ma non sono implementate forme di distribuzione automatica o ridistribuzione del carico.

Infine non sono offerti strumenti a supporto della collaborazione, anche se Willow Garage sta sviluppando ricerche in tal senso con il progetto *Building Manager*.

7.2 Esigenze di software engineering nella robotica industriale

L'evoluzione della robotica industriale sta lentamente seguendo i progressi della robotica sperimentale e la prossima generazione di applicazioni robotiche nell'in-

dustria introdurrà macchine capaci di modificare il proprio behaviour in relazione all'ambiente nel quale si troveranno ad operare, alla cooperazione con gli altri robot e alla collaborazione con l'uomo. I tradizionali concetti di cella manifatturiera caratterizzati da un sistema statico e preprogrammato stanno lasciando il posto a sistemi dinamici più complessi. Analogamente le problematiche di controllo di impianto, di monitoraggio della qualità, di parametrizzazione, cresceranno di livello incrementando le esigenze di accesso ai dati di produzione e alle variabili di controllo del sistema. Infine il tutto dovrà integrarsi con i processi di coordinamento interaziendale per rafforzare l'approccio CIM (Computer Integrated Manifacturing).

Nel *Sectorial Report on Industrial Robot Automation* (EURON 2005 [11]) viene definita la visione a medio-lungo termine (10-15 anni) dello sviluppo della robotica industriale in ambito manifatturiero: il ruolo dei robot sarà di assistenza al lavoratore nei lavori manuali con piena e simbiotica integrazione nel sistema manifatturiero. Sulla base di questo obiettivo l'articolo identifica le sfide tecnologiche da affrontare e gli ostacoli da superare. In particolare vengono riconosciute le seguenti challege:

- percezione: servirà una grande quantità di sensori e la capacità di elaborarne l'informazione, allo scopo di fornire al robot massima conoscenza dell'ambiente nel quale sta operando. Ciò coinvolge aspetti di comunicazione e computazione
- real-time physical action: real-time motion planning e approccio goal-driven per adattarsi a condizioni mutevoli
- affidabilità: è necessario gestire la complessità del sistema per garantire sicurezza e fault tolerance al sistema in un'ottica di funzionamento 24/7
- riduzione del tempo di sviluppo: la flessibilità del sistema consente la possibilità di compire rapide scelte aziendali. Lo sviluppo software deve essere opportunamente rapido, consentendo l'adozione o il riutilizzo di funzionalità prerealizzate, senza dover "reinventare" la soluzione
- architetturali: implementazione di architetture che consentano al robot di apprendere nuove competenze e sviluppare capacità di gestione delle situazioni anomale
- aspetti comunicativi: interazione e dialogo uomo-robot e robot-robot
- apprendimento: l'apprendimento continuo di nuove conoscenze e capacità da parte del robot. Il robot deve esprimere capacità cognitive, capire l'ambiente e interagirvi

- decision making: alta reattività e interazione derivata dall'inserimento del robot in un ambiente variabile e con presenza umana

Elevata connettività, cooperazione tra robot, cooperazione uomo-robot, capacità cognitive, comportamento reattivo e proattivo sono funzionalità richiedono elevata capacità di:

- comunicazione: di basso e di alto livello, veloce, affidabile e real-time
- computazione: l'elaborazione real-time dei dati sensoriali richiede una notevole potenza di calcolo, impensabile da implementare totalmente all'interno dei controllori embedded. La prospettiva del calcolo distribuito è indispensabile
- configurazione: un sistema dinamico richiede capacità di adattamento dei dispositivi in un'ottica plug&play
- coordinamento: un sistema dinamico e autonomo richiede coordinamento tra i dispositivi per poter implementare l'adattamento, non più processi preprogrammati

Queste caratteristiche si riconoscono negli obiettivi di un RSS component-based 7.1, che possono quindi costituire un valido appoggio allo sviluppo software.

La robotica industriale aggiunge tuttavia ulteriori problematiche [10] [17] alle esigenze trattate nel capitolo precedente. Effettuando un'analisi strutturata secondo le componenti di framework, comunicazione e algoritmica si ottiene:

Esigenze di framework

- Semplificazione del processo di sviluppo software: programmazione di alto livello, programmazione visuale, progettazione model driven mirate alla generazione automatica del software.
- Capacità di integrazione con framework software preesistenti. Alto livello di astrazione per la parametrizzazione del sistema, supporto alla configurazione offline, online e ibrida.
- Calcolo distribuito trasparente e adattabile: l'esecuzione di algoritmi complessi determina la necessità di distribuire il calcolo su server di servizio multiclient ad alte prestazioni. Deve essere supportata l'eventualità di condizioni anomale o di guasto dei server attraverso forme di riconfigurazione dinamica e automatica del sistema, trasferendo la computazione su altre macchine ed eventualmente

rallentando il processo produttivo in funzione delle risorse disponibili. Tutte le applicazioni accessibili devono apparire come locali in ogni parte del sistema.

- Riconfigurabilità: software di controllo, moduli di interfacciamento con robot e sensori devono poter essere connessi e disconnessi dal sistema in esecuzione continuativa, con un ottica plug&play.

- Supporto a piattaforme embedded: i sistemi industriali sono caratterizzato dalla presenza di molteplici piattaforme embedded (MPU, DSP, PLC) . Interoperabilità e portabilità implicano l'esigenza di un'architettura leggera, in grado di essere eseguita su piattaforme multiple e supportare molteplici interfacce.

- Esigenze Real-Time e di QoS di livello superiore rispetto al contesto della robotica sperimentale. Gestione di differenti livelli di priorità delle operazioni, per garantire adeguati livelli di sicurezza ed affidabilità del sistema, resource reservation, ecc.

- Autenticazione, priorità e coordinamento: accesso alle risorse del sistema sulla base di policy di sicurezza, gestione della priorità e coordinazione delle azioni da compiere.

Le caratteristiche RSF di ROS, già trattate nei paragrafi precedenti, risultano ancora inadeguate per un utilizzo diretto in ambito industriale. In particolare è critica la mancanza di supporto real-time, QoS e gestione degli aspetti di sicurezza e diritti di accesso. ROS è un meta-OS basato su kernel UNIX di cui utilizza le funzionalità di scheduling, networking e filesystem. Il supporto a piattaforme embedded, legato alle caratteristiche fornite dal sistema, è offerto dagli strumenti dello stack *eros*, attualmente in fase di sviluppo. ROS soddisfa invece ampliamente le esigenze di riconfigurabilità, dinamicità e funzionamento plug&play.

Per quanto riguarda gli strumenti di programmazione, ROS manca di un proprio IDE anche se supporta la creazione semi-automatica di progetti Eclipse. Mancano totalmente funzionalitù di programmazione di alto livello e programmazione visuale. In tal senso risulta molto interessante l'integrazione di ROS con il framework Urbi [S23]. E' stato in oltre realizzato una sorta di client library per il supporto del linguaggio UrbiScript, che consente la programmazione tramite gli strumenti Gostai Studio [S24], ambiente di programmazione di alto livello che integra funzionalità di programmazione visuale e a blocchi. Gostai Studio non è purtroppo opensource ne disponibile gratuitamente. Manca infine qualsivoglia lavoro indirizzato alla programmazione model driven.

Esigenze di communication middleware

- Interoperabilità con middleware specifici: in tutti i contesti in cui i dispositivi coinvolti non hanno le caratteristiche computazioni per implementare il middleware di sistema, ma utilizzano specifici sistemi di comunicazione adeguati alle esigenze del sistema, ad esempio le specifiche AUTOSAR [S22] [18] per il contesto automotive.
- Real-Time e QoS: le esigenze di comunicazione real-time diventano ancor più stringenti nel contesto industriale.
- Supporto e interfacciamento con reti industriali come DeviceNet, ControlNet, Profibus, e canali di comunicazione di più basso livello come Modbus.
- Tecniche di data distribution efficienti e scalabili: per consentire un adeguato controllo di impianto, monitoraggio della produzione, manutenzione, data logging, quality control
- Comunicazione streaming ad elevata capacità: ad esempio l'utilizzo pervasivo di sensori di visione 2D e 3D comporta l'esigenza di distribuire nel sistema grandi quantità di dati come streaming continuo. Il communication middleware deve fornire supporto a queste forme comunicative.

Mentre ROS fornisce adeguato supporto alla comunicazione streaming, implementabile attraverso topic, e data distribution attraverso parameter server, risulta inadeguato negli aspetti di comunicazione real-time e non fornisce nessun controllo sulla priorità delle comunicazioni. Al momento ROS non supporta l'interfacciamento con protocolli di comunicazione industriale ma non è difficile pensare alla realizzazione di tale supporto sotto forma di nodi wrapper, o conversione dei protocolli. Le stesse considerazioni valgono per quanto riguarda forme di comunicazione di basso livello, per le quali però ROS fornisce interfacce standardizzate nello stack *rosserial* (attualmente implementa solo la comunicazione seriale di Arduino).

Esigenze legate ad algoritmi e driver Le problematiche algoritmiche sono le stesse della robotica sperimentale, ma si arricchiscono di connotazioni di efficienza, affidabilità e fault tolerance. In particolare è importante disporre di ampia scelta tra algoritmi con differenti complessità computazionali, selezionabili secondo la esigenze di sistema e le risorse disponibili. Deve essere fornita la più ampia interfacciabilità con dispositivi industriali, solitamente differenti da quelli utilizzati nella robotica sperimentale.

Conclusioni

L'analisi effettuata mostra come ROS stia diventando un valido strumento a supporto della robotica sperimentale, con diverse lacune per l'applicazione industriale. Le criticità del communication middleware lo rendono ben lontano dal soddisfare le esigenze di sicurezza e affidabilità di un sistema industriale. Un altro importante scoglio da affrontare è risultato essere quello del supporto real-time: è necessario in tal senso specificare le modalità nelle quali ROS dipende dal sistema sistema operativo e quali configurazioni possa invece offrire a livello di framework e communication middleware.

Risultano invece interessanti le prospettive di integrazione di ROS come *strumento* di sistema, aggregatore di sottosistemi senza la pretesa di rivoluzionare le attuali architetture industriali. Una conferma di tal proposito viene dal recente (rispetto la data di realizzazione di questo lavoro) annuncio di utilizzo di ROS per il controllo di un braccio robotico industriale, presentato nel prossimo paragrafo.

7.3 Motoman Robotics Division adotta ROS per il controllo del SIA20

A fine agosto 2011 nel sito `http://www.businesswire.com` è uscita la notizia di una collaborazione stretta tra Yaskawa America's Motoman Robotics Division e il Southwest Research Institute (SwRI), uno dei più grandi centri di ricerca indipendenti degli USA, per introdurre ROS nella linea di robot industriali Motoman (maggiori dettagli nell'articolo "ROS Everywhere!" - `http://www.everything-robotic.com/2011/09/ros-everywhere-says-willow-garages.html`). E' il primo caso di utilizzo di ROS nell'ambito della robotica industriale seppur già da tempo SwRI utilizzi ROS per applicazioni di visione industriale e robotica autonoma mobile (vedere a proposito il sito `www.ros.swri.org`).

L'obiettivo del progetto secondo W. Clay Flannigan, manager alla sezione di Robotics and Automation Engineering dello SwRI, è la realizzazione di un interfaccia general purpose tra le funzionalità di elaborazione e percezione ROS con le esigenze architetturali della robotica industriale. L'approccio che si sta seguendo è quello di integrare ROS negli attuali sistemi di controllo industriali, mantenendone le caratteristiche di affidabilità e sicurezza ma mettendo a disposizione le funzionalità algoritmiche (a partire da quelle di path planning e graps planning) e message

passing ROS. Il lavoro verrà rilasciato (nei primi mesi del 2012) in forma opensource e ridistribuito tramite stack ROS *ROS Industrial stack* allo scopo di stimolare la comunità e altri produttori ad espandere il lavoro, introducendo il supporto ad altri robot, sensori e controller industriali. Se questa operazione andasse a buon fine porterà un grande vantaggio su entrambi i fronti:

- Motoman potrebbe aggiungere molte funzionalità ai propri robot manipolatori proponendo nuove soluzioni nel processo manifatturiero, come il robot a due mani Motoman (figura 7.2), facilitando il trasferimento tecnologico dal mondo della ricerca al proprio business e attirando nuovi clienti.

- per ROS costituirebbe, dal punto di vista commerciale, una grande prova di funzionalità e credibilità, dal punto di vista dello sviluppo l'afflusso di esperienze e know-how di industrial robotics, con conseguente ampliamento della comunità.

La prima funzionalità Motoman implementata tramite ROS sarà un'applicazione pick & place in ambiente non strutturato utilizzando le funzionalità di percezione 2D e 3D e quelle path, grasp planning.

Se i progetto avrà successo costituirà un primo passo importante per il trasferimento del know-how tra robotica sperimentale e industriale, stimolando un ulteriore sviluppo di ROS in tal senso.

(a) Motoman SIA20

(b) Motoman Dual-Arm robot

(c) Motoman con visione

Figura 7.2: Braccio robotico supportato da ROS e possibili evoluzioni

Capitolo 8

Conclusioni

Il lavoro di integrazione delle funzionalità di simulazione SimSpark in ROS ha dimostrato ancora una volta le potenzialità e la versatilità del middleware di Willow Garage. Il supporto fornito nel sito web e i numerosi tutorial sviluppati dalla comunity consentono una curva di apprendimento rapida e l'ottenimento di risultati immediati. L'integrazione del simulatore SimSpark sviluppato ne consentirà un utilizzo più semplice, mettendo a disposizione dello sviluppatore i numerosi strumenti ROS. Prossimamente il package di wrapping sarà caricato nel repository ROS e condiviso con la comunità, nella speranza di riscuotere successo in ambito RoboCup e attirare anche questa community verso l'utilizzo di ROS.

L'ambiente di programmazione ROS-based realizzato fornisce allo studente supporto per lo sviluppo delle sue applicazioni, consentendogli di analizzare molteplici problematiche. Senza eccedere in superficialità o eccessivo dettaglio è possibile sviluppare molteplici tematiche di simulazione, che vanno dal semplice controllo dei movimenti hand-coded, allo studio di problematiche di motion planning o di comunicazione in ambito multi robot.

La documentazione raccolta circa lo sviluppo di modelli 3D e relative descrizioni URDF e SRG offrirà un importante supporto a eventuali lavori di modellazione futura, consentendo di velocizzarne la realizzazione attraverso le macro sviluppate, e fornendo indicazioni sull'approccio corretto da seguire.

Infine il lavoro ha messo in luce le potenzialità e i limiti di ROS come strumento di sviluppo per robotica, in particolare per l'ambito industriale, evidenziandone possibili criticità. Queste osservazioni potranno essere spunto di lavori futuri volti a colmare i gap individuati ed offrire uno strumento sempre più completo ed efficace.

8.1 Lavoro futuro

Il lavoro svolto lascia aperte molte strade di sviluppo futuro: in primis la possibilità di completare il framework di controllo con l'aggiunta del driver per il controllo robot Robovie-X reale. Dal punto di vista della simulazione c'è forse il maggior lavoro da poter svolgere: collaborare con la comunità SimSpark per creare una versione del simulatore *davvero* general purpose, con caratteristiche di configurazione runtime migliori, uno strumento di creazione dell'ambiente di simulazione visuale, una forma di indipendenza dal modello del robot più efficace. Un altro aspetto da sviluppare potrebbe essere la realizzazione di uno strumento di creazione automatica della descrizione RSG a partire dal modello URDF, per evitare il tedioso compito di mantenere due descrizioni distinte, oppure, ancor meglio, l'aggiunta a SimSpark del linguaggio di descrizione URDF. In questo caso sarebbe però necessario creare estensioni al linguaggio per la specifica di perceptor ed effector, in quanto la descrizione URDF è da intendersi a solo scopo strutturale. Ancora, è possibile estendere SimSpark con la creazione di nuovi perceptor, secondo le esigenze specifiche.

Infine un lavoro certamente ad compiere sarà la creazione di uno stack a raccolta degli strumenti sviluppati, l'inclusione dei sorgenti del simulatore e la semplificazione del processo di compilazione, adattandolo agli strumenti ROS. Quindi il caricamento e il condivisione nei repository ROS e la creazione della documentazione per la wiki, passi fondamentali per attirare l'attenzione della comunità sul lavoro svolto.

Sitografia

[S1] ROS: http://www.ros.org

[S2] Player: http://playerstage.sf.net

[S3] OpenRTM-aist: http://www.openrtm.org

[S4] Orocos: http://www.orocos.org

[S5] YARP: http://eris.liralab.it/yarp

[S6] Microsoft Robotics Studio: http://msdn.microsoft.com/en-us/robotics/default.aspx

[S7] ODE: http://www.ode.org

[S8] SimSpark: http://simspark.sourceforge.net
Wiki: http://simspark.sourceforge.net/wiki/index.php

[S9] Integrazione Stage in ROS: http://www.ros.org/wiki/stage

[S10] Integrazione Gazebo in ROS: http://www.ros.org/wiki/simulator_gazebo

[S11] Home page dello IAS-Lab dell'università di Padova http://robotics.dei.unipd.it/

[S12] Progetto STAIR: http://stair.stanford.edu

[S13] Gruppo WillowGarage: http://www.willowgarage.com
http://www.willowgarage.com/pages/about-us

[S14] XML-RPC Home Page http://www.xmlrpc.com

[S15] Guida all'installazione ROS: http://www.ros.org/wiki/diamondback/Installation/Ubuntu

[S16] Guida all'installazione SimSpark: http://simspark.sourceforge.net/wiki/index.php/Installation_on_Linux

[S17] Librerie C++ Boost: http://www.boost.org

[S18] Vstone, azienda produttrice di Robovie-X: http://www.vstone.co.jp/english/products/robovie_x

[S19] Regole per il gioco del calcio in SimSpark http://simspark.sourceforge.net/wiki/index.php/Soccer_Simulation

[S20] Home page della libreria Orocos KDL http://www.orocos.org/kdl

[S21] Librerie C++ POCO http://pocoproject.org

[S22] AUTomotive Open System ARchitecture http://www.autosar.org

[S23] Urbi framework project http://www.urbiforge.org

[S24] Urbi-Gostai project http://www.gostai.com/products/studio

Bibliografia

[1] J. Kramer and M. Scheutz, *Development environments for autonomous mobile robots: A survey*, Autonomous Robots, vol. 22, no. 2, pagine 101-132, 2007.

[2] O. Obst and M. Rollmann, *SPARK - a Generic Simulator for Physical Multiagent Simulations*, Computer Systems Science and Engineering, September 2005.

[3] Martin Riedmiller et al., *RoboCup: Yesterday, Today, and Tomorrow Workshop of the Executive Committee in Blaubeuren, October 2003*, in Lecture Notes in Computer Science, Robot Soccer World Cup VII, Springer Berlin / Heidelberg, 2004.

[4] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully B. Foote, Jeremy Leibs, Rob Wheeler, Andrew Y. Ng., *ROS: an open-source Robot Operating System*, in ICRA Workshop on Open Source Software, 2009.

[5] A. Shakhimardanov, E. Prassler *Comparative Evaluation of Robotic Software Integration Systems: A Case Study*, IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2007, pp. 3031-3037

[6] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, *A Review of Middleware for Networked Robots* International Journal of Computer Science & Network Security, Vol. 9 No. 5, pp. 139-148, May 2009.

[7] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, A. Oreback *Towards component-based robotics*, IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2005, pp. 163-168

[8] Davide Brugali, Azamat Shakhimardanov *Component-Based Robotic Engineering (Part II)*, IEEE Robotics & Automation Magazine, Vol. 17, No. 1. (March 2010), pp. 100-112.

[9] Nader Mohamed, Jameela Al-Jaroodi, and Imad Jawhar *Middleware for Robotics: A Survey*, In Proc. of The IEEE Intl. Conf. on Robotics, Automation, and Mechatronics (RAM 2008), pp. 736-742, Sep. 2008.

[10] M. Amoretti, M. Reggiani, S. Caselli *Designing Distributed, Component-based Systems for Industrial Robotic Applications* Industrial Robotics - Programming, Simulation and Applications, edited by Low Kin Huat, ARS, Austria, 2007.

[11] *Sectoral Report on Industrial Robot Automation* White paper - Insustrial Robot Automation - FP6-001917, DR.14.1 - 2005.

[12] Alexei Makarenko, Alex Brooks, Tobias Kaupp *On the Benefits of Making Robotic Software Frameworks Thin* In IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'07) Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware (November 2007)

[13] Geoffrey Biggs, Noriaki Ando, and Tetsuo Kotoku *Native Robot Software Framework Inter-operation* SIMPAR 2010 proceedings (November 2010), Springer volume 6472, pp. 180-191.

[14] N. Mohamed, J. Al-Jaroodi, I. Jawhar *A Review of Middleware for Networked Robots* in International Journal of Computer Science & Network Security, Vol. 9 No. 5, pp. 139-148, May 2009.

[15] Gregory Broten and David Mackay *Barriers to Adopting Robotics Solutions* IEEE ICRA 2007 Workshop on Software Development and Integration in Robotics (SDIR-II)

[16] A. Rotenstein, A. Rothenstein, M. Robinson, and J. Tsotsos *Robot middleware must support task-directed perception* in ICRA 2nd International Workshop on Software Development and Integration into Robotics, April 14 2007, rome, Italy.

[17] Saehwa Kim, Seongsoo Hong *Reference Middleware Architecture for Real-Time and Embedded Systems: A Case for Networked Service Robots* Handbook of Real-Time and Embedded Systems , Chapman & Hall/CRC Computer & Information Science Series, 2007

[18] Dietmar Schreiner *Component Based Communication Middleware for AUTOSAR* PhD thesis at Vienna University of Technology - Institute of Computer Languages, Compilers and Languages Group

Appendice A

Esempio di fornitura di un perceptor topic

In questa sezione si presenta un esempio di fornitura di un perceptor topic con lo scopo di mostrare il processo di parsing di un S-Expression e pubblicazione del corrispondente messaggio ROS. L'esempio vuole anche costituire un riferimento per un eventuale aggiunta futura di nuovi perceptor. Come spiegato nel paragrafo 4.1.3.4, il parsing delle S-Expression e l'inoltro in ROS del messaggio è coadiuvato dalla classe `MessageMap`, che associa ad ogni predicato di perceptor interpretato, l'istanza della classe `RosMessage` in grado di effettuarne il parsing e che si occuperà della pubblicazione nel topic.

L'esempio che si riporta è quello per la conversione e la pubblicazione di messaggi provenienti dal HingeJoint perceptor. Volendo implementare il supporto a questo perceptor, ma in generale per qualsiasi perceptor, è necessario effettuare due operazioni:

1. Definire la struttura del messaggio msg da pubblicare nel topic, specificandolo in un apposito file msg o utilizzando un messaggio esistente.

2. Implementare una nuova classe `HingeJointMessage` che estenda l'interfaccia `RosMessage` implementandone i metodi. La classe dovrà in oltre incapsulare il tipo di messaggio per il topic ROS associato.

3. Aggiungere alla mappa `message_map`, membro classe `MessageMap`, il puntatore ad un'istanza della classe `HingeJointMessage`, associandolo alla stringa che identifica il perceptor supportato.

```
class RosMessage
{
protected:
    bool m_bModified;
    ros::Publisher topic;
public:
    virtual bool parse(const parser::Predicate & predicate)=0;
    virtual bool publish(void)=0;
    virtual void clean(void)=0;
    RosMessage(){m_bModified=false;}
};
```

Listato A.1: Interfaccia RosMessage

Per quanto riguarda la specifica del messaggio da pubblicare sul topic, si è scelto di definirne uno specifico per i messaggi perceptor da giunti hinge. E' stato pertanto creato il file *HigeJointPerceptor.msg* all'interno della directory *msg*, con il contenuto:

```
string[] name
float32[] value
```

Il messaggio pertanto specificherà due array, uno per i nomi dei giunti e uno per i valori associati, creando coppie ("nomeGiunto",valore). La compilazione *rosmake* del package determinerà l'autogenerazione della classe `simulator_simspark::HingeJointPerceptor` che incapsula il messaggio.

L'interfaccia `RosMessage` è riportata nel listato A.1. I metodi da implementare sono: `parse`, che deve effettuare il parsing del predicato passato componendo il messaggio ROS, `publish`, che deve effettuare la pubblicazione del messaggio ROS nel topic, `clean` che effettua l'azzeramento del messaggio per iniziare un nuovo ciclo di parsing e infine il costruttore della classe, che deve pubblicare il topic. L'header della classe è riportato nel listato A.2, come si può vedere l'unica aggiunta rispetto l'interfaccia, a parte la dichiarazione del costruttore, è il membro privato `msg` del tipo del messaggio ROS del topic associato al perceptor. L'implementazione costruttore effettua la pubblicazione del topic su cui verranno inoltrate le notifiche dai perceptor dei giunti hinge:

```
HingeJointMessage::HingeJointMessage(ros::NodeHandle &n,const std::
    string &name_space)
{
    std::stringstream strs;
    strs<<name_space<<"/hingejoint_perceptors_topic";
```

```
class HingeJointMessage:RosMessage
{
private:
    simulator_simspark::HingeJointPerceptor  msg;
public:
    HingeJointMessage(ros::NodeHandle &n,
        const std::string &name_space);
    ~HingeJointMessage(){}
    bool parse(const parser::Predicate & predicate);
    bool publish(void);
    void clean(void);
};
```

Listato A.2: Header della classe HingeJointMessage

```
    topic = n.advertise<simulator_simspark::HingeJointPerceptor>(strs.
        str(),1000);
}
```

si può osservare la specifica del nome del topic sul quale verranno pubblicati i messaggi, quindi la creazione dell'istanza della classe `ros::Publisher` per la pubblicazione. Viene impostato un buffer di 1000 messaggi per i messaggi in uscita.

L'implementazione del metodo parse effettua l'estrazione dei parametri dal messaggio da SimSpark, S-Expression incapsulata in un oggetto `parser::Predicate`. Ad ogni invocazione viene aggiornato il messaggio ROS con l'aggiunta delle nuove notifiche:

```
bool HingeJointMessage::parse(const parser::Predicate & predicate)
{
    std::string name;
    float value;
    parser::Predicate::Iterator iter(predicate);

    if(!predicate.GetValue(iter, ``n'', name))
        return false;
    if(!predicate.GetValue(iter, ''ax'', value))
        return false;

    msg.name.push_back(name);
    msg.value.push_back(value);
    m_bModified=true;
    return true;
}
```

ottenuto un iteratore del predicato, viene utilizzato il metodo `GetValue` di `Predicate` per effettuare la ricerca e la lettura di un parametro identificato per nome, nella fattispecie “n“ e ”ax“, i due parametri del messaggio HingeJoint perceptor (si veda il paragrafo 1.4.3 per una descrizione dettagliata del formato dei messaggi). Una volta ottenuti il nome del giunto e il valore associato, le informazioni vengono agganciate al messaggio ROS e viene settato il flag `m_bModified`, utilizzato dal metodo `publish` per verificare la presenza di qualcosa da pubblicare.

L'implementazione del metodo `publish` effettua la pubblicazione del messaggio ROS nel topic corrispondente. Il metodo è invocato dalla `MessageMap` al termine della fase di estrazione e decodifica dei messaggi dal burst

```
bool HingeJointMessage::publish(void)
{
    if(m_bModified)
    {
        topic.publish(msg);
        return true;
    }
    return false;
}
```

Infine l'implementazione del metodo clean resetta il messaggio ROS. E' richiamato dalla `MessageMap` prima dell'inizio di un nuovo ciclo di estrazione e decodifica dei messaggi dal burst:

```
void HingeJointMessage::clean(void)
{
    m_bModified=false;
    simulator_simspark::HingeJointPerceptor  cleared;
    msg=cleared;
}
```

per effettuare il reset del messaggio ROS viene creata una nuova istanza ”pulita“ e copiata nel messaggio, diversamente è possibile procedere al reset specifico di ogni membro della classe.

Infine resta da aggiungere un'istanza della classe alla `MessageMap`. Ciò viene effettuato nel costruttore della `MessageMap`:

```
MessageMap::MessageMap(ros::NodeHandle &n,const std::string &name_space
    )
{
    message_map["GYR"]=boost::shared_ptr<RosMessage>((RosMessage *)new
        GyroRateMessage(n,name_space));
```

```
    message_map["hear"]=boost::shared_ptr<RosMessage>((RosMessage *)new
        HearMessage(n,name_space));
    ...
    message_map["See"]=boost::shared_ptr<RosMessage>((RosMessage *)new
        VisionMessage(n,name_space));

    message_map["HJ"]=boost::shared_ptr<RosMessage>((RosMessage *)new
        HingeJointMessage(n,name_space));
}
```

l'istanza della classe `HingeJointMessage` viene associata alla stringa "HJ" che identifica il messaggio del perceptor HingeJoint.

Appendice B

Esempio di fornitura di un effector service

In questa sezione si presenta un esempio di implementazione di una funzione di callback per la fornitura di un service all'interno di `VirtualAgent`, descrivendo i passi da compiere per aggiungere il controllo di nuove tipologie di effector, utilizzando come riferimento la gestione del Hinge joint effector service.

Le operazioni da svolgere per aggiungere la gestione di un nuovo tipo di effector sono:

1. Definire la struttura dei messaggi request e response del service, specificandoli in un apposito file srv o utilizzando un messaggio esistente.

2. Implementare una nuova funzione di callback in `VirtualAgent`, in questo caso `hingeJointSCallback`, che si occupi della gestione delle chiamate al service.

3. Aggiungere un handle per il service come mebro della classe `VirtualAgent`, quindi registrale il service, in questo caso "`hingejoint_effectors_service`", associandolo al callback creato.

Per quanto riguarda la specifica del formato dei messaggi request e response si è scelto di definirne uno di nuovo. Si è quindi creato il file *HingeJointEffector.srv* all'interno della directory *srv* del package indicando:

```
string[] name
float32[] speed
---
bool result
```

Il messaggio request è quindi costituito da due vettori i cui elementi corrispondenti formano le coppie (“<nome giunto>”,<velocità giunto>), mentre il messaggio response riporta lo stato di completamento del controllo dell'effector. La compilazione del package con *rosmake* autogenera le classi C++

`simulator_simspark::HingeJointEffector`,
`simulator_simspark::HingeJointEffectorRequest` e
`simulator_simspark::HingeJointEffectorResponse`.

Dopo la generazione delle classi è possibile procedere alla scrittura della funzione di callback, riportato nel listato B.1. Il prototipo della funzione di callback deve specificare come parametri i messaggi request e response appena generati, si è scelto di specificare il callback come metodo della classe `VirtualAgent`.

Nel particolare, nelle righe 3-8 viene fatto un controllo di coerenza sulla dimensione dei vettori: poichè gli elementi corrispondenti rappresentano coppie i due vettori devono avere la stessa dimensione. Si procede quindi scorrendo le coppie (nome del giunto,velocità del giunto) e componendo per ciascuna il messaggio S-Expression da inviare a SimSpark, inviato utilizzando il metodo `send` di `VirtualAgent` che effettua l'invio di una stringa sul socket. Il formato del messaggio per l'effector HingeJoint è riportato nel paragrafo 1.4.1. Infine bisogna aggiungere il membro `ros::ServiceServer hingeJointService;` alla classe `VirtualAgent`, che costituirà l'handle per il service, membro da inizializzare nel costruttore della classe:

```
1  VirtualAgent::VirtualAgent(...)
2  {
3      ...
4      std::stringstream strs;
5      strs<<client_namespace_<<"/beam_effectors_service";
6      beamEffectorService = n.advertiseService(strs.str(),&VirtualAgent::
           beamEffectorSCallback, this);
7      strs.str("");
8      ...
9      strs<<client_namespace_<<"/universaljoint_effectors_service";
10     universalJointService = n.advertiseService(strs.str(),&VirtualAgent
           ::universalJointSCallback, this);

12     strs<<client_namespace_<<"/hingejoint_effectors_service";
13     hingeJointService = n.advertiseService(strs.str(),&VirtualAgent::
           hingeJointSCallback, this);
14     ...
15 }
```

```
1  bool VirtualAgent::hingeJointSCallback(simulator_simspark::
       HingeJointEffectorRequest &req, simulator_simspark::
       HingeJointEffectorResponse &res)
2  {
3      unsigned int n=req.name.size();
4      if(n!=req.speed.size())
5      {
6          res.result=false;
7          return true;
8      }
9      for(unsigned int i=0;i<n;i++)
10     {
11         std::stringstream message_str;
12         message_str << "(" << req.name[i] << " " << req.speed[i] << ")"
               ;
13         if(!send(message_str.str()))
14         {
15             res.result=false;
16             return true;
17         }
18     }
19     res.result=true;
20     return true;
21 }
```

Listato B.1: Metodo callback per la gestione del service di controllo dei giunti Hinge

Alle righe 12-13 viene effettuata la registrazione del service, specificando il namespace scelto dal client ROS che ha richiesto la creazione dell'agente simulato, effettuando l'inizializzazione della membro `hingeJointService`. Il service viene quindi messo in servizio alla successiva creazione del thread di spin.