UNIVERSITÀ DEGLI STUDI DI PARMA
FACOLTÀ DI INGEGNERIA
Corso di Laurea in Ingegneria Informatica

REALIZZAZIONE DI UN'ARCHITETTURA SOFTWARE RICONFIGURABILE PER UN ROBOT MOBILE DI SERVIZIO

Relatore:
Chiar.mo Prof. STEFANO CASELLI

Correlatori:
Dott. Ing. MONICA REGGIANI
Dott. Ing. FRANCESCO MONICA

Tesi di laurea di:
ALESSIO RUFFINI

28 Aprile 2004

Ai miei genitori e a padre Lorenzo.

Ringrazio il mio relatore, il Prof. Caselli, che mi ha seguito nell'impostazione e sviluppo di questo progetto denotando alte capacità professionali.

Ringrazio l'ing. Reggiani per i validi consigi ricevuti durante la fase di stesura della tesi.

Ringrazio l'ing. Monica per l'aiuto che mi ha dato durante il lavoro con il robot.

Ringrazio Jacopo, Michele, l'Eleonora e tutti gli studenti e i laureandi del laboratorio di robotica e della palazzina 1.

Saluto infine tutti gli amici con cui ho condiviso i momenti di svago e in particolare l'Arianna per il costante supporto morale.

Indice

Capitolo 1

Introduzione

La progettazione di sistemi software sempre più complessi ed articolati richiede di identificare, a più livelli, entità indipendenti ed autonome, hardware o software, che collaborano per lo svolgimento di un compito comune. Lo sviluppo dell'*Ingegneria del Software* deriva dalla necessità di mantenere sotto controllo la complessità intrinseca dei sistemi di grandi dimensioni e sottolinea in modo particolare l'esigenza di produrre software caratterizzato da un alto grado di riutilizzabilità e interoperabilità, per consentirne l'utilizzo in contesti molteplici ed in continua evoluzione.

Anche la *Robotica* mostra un crescente interesse verso l'adozione di strumenti che, favorendo lo sviluppo di applicazioni aperte, riconfigurabili e modulari, possono evolvere grazie all'apporto progressivo di hardware e software di differente provenienza. La crescita dei settori legati alla robotica di servizio, ed alla *domotica* in particolare, ha introdotto l'utilizzo della robotica *intelligente* anche al di fuori del campo accademico, focalizzando l'attenzione verso nuovi problemi divenuti recentemente di importanza cruciale. La necessità di produrre soluzioni a basso costo che siano nel contempo facili da utilizzare per l'utente finale ha spinto le aziende ad abbandonare progressivamente l'utilizzo di protocolli proprietari, largamente impiegati nell'hardware e nel software fino a pochi anni fa, in favore dell'uso di standard aperti che agevolano l'interconnessione delle diverse componenti del sistema e la loro sostituzione in presenza di nuove versioni in luogo delle precedenti ormai obsolete.

Questa tesi è stata svolta con l'intento di sviluppare un'architettura software

da utilizzare sul robot mobile *Nomad 200* disponibile presso il Laboratorio di Robotica del Dipartimento di Ingegneria dell'Informazione. Obiettivo della tesi è far sì che la nuova architettura sia caratterizzata da un alto grado di riconfigurabilità e modularità, in modo tale che i componenti software prodotti non siano specifici per una determinata architettura, ma possano essere facilmente esportati su un'altra architettura senza la necessità di essere completamente ripensati.

Il lavoro svolto in questa tesi si è reso necessario dopo l'aggiornamento hardware del *Nomad 200* effettuato lo scorso anno. Infatti *robotd*, il software di controllo del robot fornito dalla casa costruttrice, mostra i limiti della robotica del periodo in cui è stato costruito. In particolare, la ridotta capacità computazionale a bordo del robot aveva indirizzato i progettisti di *robotd* verso soluzioni che privilegiano l'accesso remoto all'uso locale. Al contrario, ora il robot, grazie all'aggiornamento hardware, dispone di un processore dell'ultima generazione che permette di svolgere algoritmi di notevole complessità senza far ricorso ad un calcolatore aggiuntivo. Un notevole limite di *robotd*, che può ridurre sensibilmente la reattività delle applicazioni, è anche la completa mancanza di meccanismi di comunicazione di tipo *push*: le letture dei sensori avvengono infatti tramite una richiesta effettuata dal client al server, introducendo così un tempo di latenza significativo, anche se difficilmente misurabile, tra l'istante in cui il dato viene letto dall'hardware e quello in cui diventa disponibile per il client. Questo intervallo può essere ridotto unicamente con l'aumento della frequenza con cui il dato viene richiesto; tale aumento introduce *overhead* sulle comunicazioni e comunque anche in questo caso l'intervallo non può essere completamente annullato. Inoltre *robotd*, che non utilizza nessuna funzionalità real-time del sistema operativo, non garantisce il funzionamento ottimale, con conseguenti prestazioni che decadono progressivamente quando l'unità di elaborazione è sovraccarica.

Per tutti questi motivi si è deciso di realizzare una nuova architettura con caratteristiche di riconfigurabilità, di modularità e di apertura a nuovi standard seguendo soluzioni alternative a quelle adottate dai progettisti di *robotd*. Dopo un attento studio delle varie architetture presenti nel panorama robotico è stato scelto come punto di partenza l'insieme dei *pattern di comunicazione* sviluppato nel corso del progetto OROCOS [1]. Questo progetto *open source* fornisce una base comune per lo sviluppo di applicazioni sia per robot mobili che per manipolatori, è modulare e dispone

di una chiara documentazione.

1.1 Robot mobili

Il campo della robotica e in particolare quello riguardante i robot mobili autonomi, costituisce un importante settore di ricerca applicativa dell'intelligenza artificiale. Gli studi principali affrontano problematiche quali il controllo del movimento del robot onde evitare ostacoli, la determinazione di una traiettoria ottimale per raggiungere un prefissato obbiettivo, la costruzione di mappe topologiche-simboliche dell'ambiente esplorato, fino a comprendere comportamenti autonomi per la gestione della ricarica degli accumulatori e di navigazione coordinata di più robot cooperanti.

1.1.1 Architettura generale di un robot mobile

Un generico robot mobile può essere schematizzato in quattro parti fondamentali: sensori, attuatori, controllore e memoria (figura 1.1).

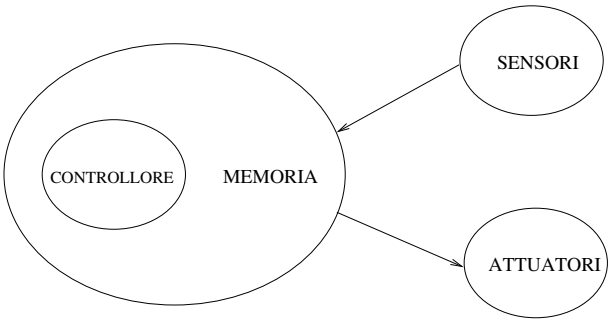

Figura 1.1: Architettura generale di un robot mobile.

I sensori estrapolano informazioni dall'ambiente circostante, mentre gli attuatori consentono il movimento fisico del robot. Gli attuatori più diffusi per i robot mobili sono i motori in corrente continua. Il controllore gestisce le risorse disponibili ed ha il compito di elaborare le percezioni sensoriali per generare istruzioni destinate agli attuatori. È questa la parte più delicata ed importante di qualunque architettura robotica perchè racchiude ogni tipo di controllo sul comportamento del

robot. Diversi metodi, basati su differenti approcci, vengono utilizzati dai ricercatori nella realizzazione del controllore. Grande interesse ha rivestito negli ultimi anni la trattazione approfondita della stessa architettura software del controllore. Particolarmente accreditata è la proposta di Brooks denominata *Subsumption Architecture* [2], che divide il sistema di controllo in diversi livelli gerarchici in base ai comportamenti esibiti dal robot mobile. Un'analisi delle architetture software di controllo è sviluppata nel capitolo 2.

La memoria viene impiegata per immagazzinare sia i programmi di controllo, sia le informazioni provenienti dai sensiori. Il software relativo al controllore viene normalmente scritto e sviluppato su una macchina ospite per poi trasferirlo al robot.

Differenti tipi di sensori possono trovare utile impiego su di un robot mobile. Distinguendoli in base alla loro diversa natura, essi possono essere classificati come sensori di distanza o di prossimità, sensori di velocità, sensori di contatto e sensori visivi. Due tipi di sensori largamente usati per la percezione *eterocettiva* sono il sensore ad onde ultrasoniche e quello a raggi infrarossi.

1.2 Il Nomad 200

Il robot utilizzato durante questa tesi è il *Nomad 200*, un robot mobile che la società statunitense *Nomadic Technologies Inc.* produceva durante la prima metà degli anni '90 assieme ad una vasta gamma di supporti per la robotica mobile. Recentemente la società è stata assorbita da una importante compagnia operante nel settore delle telecomunicazioni, ed ha perciò interrotto ricerca e produzione dei propri prodotti per la robotica.

La famiglia di robot mobili *Nomad* ha indubbiamente giocato un ruolo rilevante nella didattica e nella ricerca dell'ultimo decennio: lo dimostrano l'ampio utilizzo dei modelli *Scout*, *200* e *XR4000* nelle università europee e d'oltre oceano, come spesso si evince dalla letteratura specifica del settore robotico e dell'intelligenza artificiale, e la palese influenza che questi modelli hanno avuto sull'evoluzione dei supporti per la robotica mobile progettati dalle altre aziende negli anni seguenti.

Il Nomad 200 [3] è un robot di medie dimensioni che presenta una ricca dotazione di sensori ed attuatori particolarmente adatti ad un utilizzo in ambienti chiusi e

strutturati[1]. Il robot, di cui è presentata un'immagine in figura 1.2, è stato realizzato in modo tale da contenere un'unità autonoma di percezione, elaborazione e movimento in un ingombro complessivo relativamente ridotto, avente forma cilindrica di 50 cm di diametro e 80 cm di altezza (al piano superiore).

Figura 1.2: Il robot mobile Nomad 200.

1.2.1 Movimento

Il robot è sostenuto da tre ruote di identica dimensione, illustrate nello schema di figura 1.3, fissate in posizione simmetrica alla base della struttura. Il moto traslazionale del robot è prodotto dal movimento solidale delle ruote, a cui viene trasmessa

[1] In gergo si usa spesso il termine inglese *indoor environment*.

l'energia di un unico motore elettrico mediante un sistema di cinghie. La direzio-

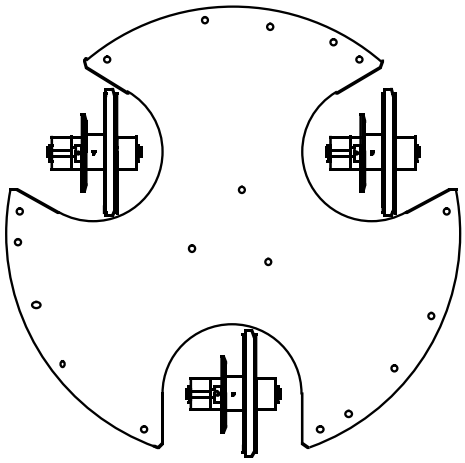

Figura 1.3: Le ruote (vista dal basso).

ne lungo cui si sposta il robot viene modificata da un secondo motore che ruota contemporaneamente l'asse di ogni ruota in modo sincrono. La parte superiore del robot (*torretta*) può ruotare in modo completamente indipendente rispetto alla base mediante un terzo motore posto nella parte inferiore.

Gli organi meccanici descritti, controllati da una scheda dedicata *Galil DMC-630* [4], consentono al robot di raggiungere velocità massime di spostamento e rotazione pari rispettivamente a 50 $\mathrm{cm/s}$ e 45 $^\circ/\mathrm{s}$.

Il sistema di locomozione permette comunque al Nomad di ruotare attorno al proprio asse geometrico senza che sia necessario spazio aggiuntivo oltre al normale perimetro del robot.

1.2.2 Sensorialità *propriocettiva*

Odometria

L'elettronica di controllo dei servomeccanismi fornisce un insieme di primitive fondamentali, necessarie al calcolo per odometria della posizione del robot: il sistema di elaborazione centrale contenuto all'interno del Nomad si avvale delle informazioni provenienti dagli organi di movimento per produrre una stima in tempo reale delle velocità istantanee di traslazione e rotazione del robot. Tali grandezze vengono

ulteriormente combinate ed integrate nel tempo per determinare il percorso compiuto sul piano durante il normale ciclo di funzionamento. La posizione corrente del robot, calcolata con una risoluzione temporale di 18.2 Hz, ne individua le coordinate (x, y) sul piano e l'orientazione della base rispetto all'asse delle ascisse con una precisione di circa 2.5 mm sugli assi traslazionali e 0.1° per quelli rotazionali.

Contemporaneamente alle misure odometriche il robot fornisce alcune informazioni riguardanti il proprio stato di funzionamento, quali la verifica di eventuali condizioni di stallo dei motori e lo stato di carica delle batterie.

Bussola

Sulla sommità del robot è posizionata una bussola elettronica *KVH C100* che rappresenta un sensore assoluto di orientazione rispetto al nord magnetico. Il principio di funzionamento dello strumento consiste nella misura elettronica dell'angolo formato tra un piccolo nucleo magnetico posto in fluttuazione libera all'interno del suo contenitore e la sua posizione a riposo. Un piccolo microprocessore dedicato interpreta le informazioni provenienti dall'ago magnetico e produce una stima dell'angolazione della torretta del Nomad con una precisione dell'ordine di 0.3°. La correttezza della misura è comunque fortemente influenzata dalla presenza di campi magnetici indesiderati nelle vicinanze del robot.

1.2.3 Sensorialità *eterocettiva*

Sensori di contatto

Lo sviluppo geometrico del Nomad 200 è realizzato in modo tale da proteggere quanto più possibile le componenti più delicate dalle eventuali collisioni che possono avvenire con gli ostacoli incontrati durante il movimento. Per questo motivo la base del robot presenta un diametro maggiore rispetto alla torretta ed è rivestita da due anelli sporgenti di gomma antiurto, visibili nel dettaglio riportato in figura 1.4. All'interno dello strato di gomma sono inseriti 20 micro interruttori (10 per ogni anello) che individuano la presenza di un oggetto che si trovi a contatto del robot; la disposizione di tali sensori, detti *sensori di pressione* o più semplicemente *bumper*, consente di rilevare la presenza di oggetti lungo tutto il perimetro del robot e

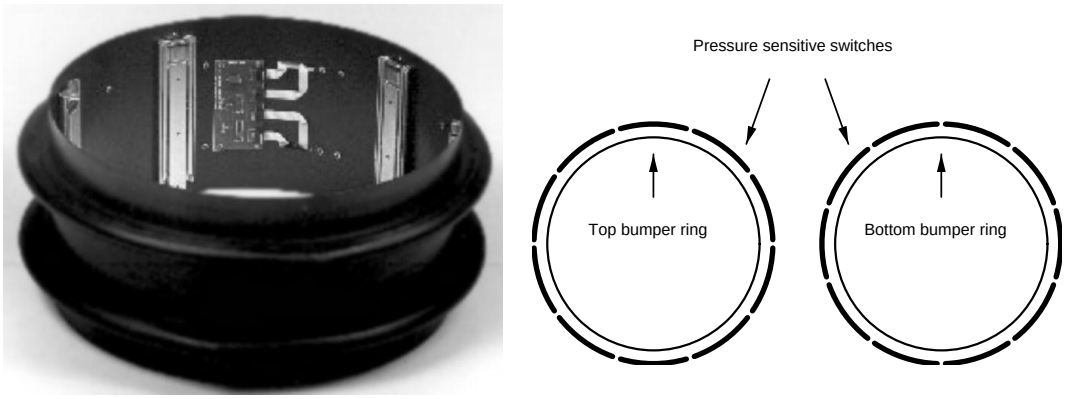

Figura 1.4: Sensori di contatto: *bumper*.

può essere efficacemente utilizzata per realizzare operazioni che ne coinvolgano lo spostamento (*object pushing*).

Sensori di prossimità

Nella sommità della torretta è alloggiato un sistema sensoriale in grado di rilevare la presenza di ostacoli nello spazio di movimento che si basa su un totale di 16 sensori sonar ultrasonici *Polaroid* controllati dalla scheda dedicata *Polaroid 6500*; il modulo *Sensus 200*, illustrato in figura 1.5, consente di misurare correttamente le distanze nell'intervallo $30 \div 600\ \mathrm{cm}$ con una precisione stimata pari all'1%. Il *beam*

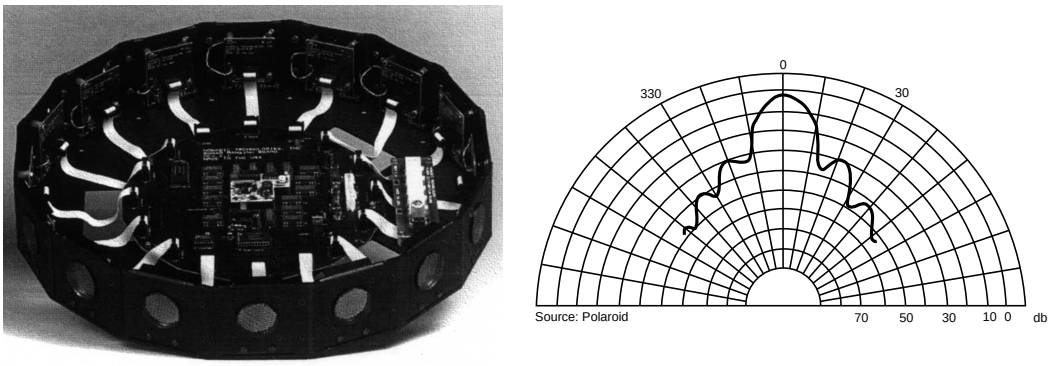

Figura 1.5: Sensori di prossimità: *sonar*.

pattern riportato in figura 1.5 (realizzato ad una frequenza di $49.4\ \mathrm{kHz}$) mostra come un singolo trasduttore sonar sia in grado di illuminare e rilevare efficacemente gli

oggetti che si trovano entro un cono massimo di circa 25°: i 16 sensori, che risultano sfasati tra loro di 22.5°, riescono quindi a coprire l'intero perimetro del robot.

I sonar utilizzati sfruttano la misura del *tempo di volo* di un treno di impulsi per identificare la distanza di un eventuale ostacolo. Dal momento che il medesimo trasduttore elettrostatico viene utilizzato sia per produrre il segnale che per riceverne l'eco di ritorno, diventa impossibile rilevare oggetti molto vicini perché gli echi prodotti si sommano ai fenomeni di risonanza propri del sensore. Il circuito stampato a cui sono connessi direttamente i 16 sensori realizza un *multiplexing* che consente di collegare alternativamente ogni sonar con un solo circuito di comando Polaroid posto al centro della scheda: da software è possibile abilitare selettivamente i singoli sensori e decidere l'ordine progressivo di sparo che viene ripetuto ciclicamente dall'hardware dedicato. Assieme al *firing pattern* è possibile specificare anche la durata del ciclo di ricezione che viene eseguito prima della commutazione del circuito di comando sul sonar successivo: valori bassi di tale intervallo consentono di eseguire un numero maggiore di misure nell'unità di tempo, ma l'utilizzo di tempi dilatati permette di rilevare echi provenienti da oggetti distanti, che altrimenti non possono tornare alla sorgente prima dell'invio dell'impulso successivo. L'intervallo di sparo può variare da un minimo di 4 ms ad un massimo di 1 s; il valore minimo consente di individuare ostacoli ad una distanza massima di circa 70 cm.

L'utilizzo di un solo circuito di comando per l'intero anello di sonar impedisce di attivare contemporaneamente più di un sensore alla volta: questa soluzione però, oltre a consentire un notevole risparmio di hardware, è di fatto implicita nell'uso di sensori di questo tipo, che, soffrendo fortemente di problemi di interferenza reciproca, sono generalmente inadatti ad un uso simultaneo.

Per misurare la distanza di oggetti particolarmente vicini al corpo del robot la Nomadic mette a disposizione il modulo *Sensus 300*, inserito nella parte inferiore della torretta, che dispone di 16 emettitori/ricevitori di radiazione infrarossa in grado di rilevare ostacoli fino ad una distanza massima di circa 60 cm (figura 1.6). Ogni sensore è formato da due diodi LED emettitori ed un fotodiodo rilevatore: la distanza viene misurata a partire dall'intensità della radiazione riflessa che viene opportunamente modulata dai diodi emettitori per ridurre l'effetto del rumore presente nell'ambiente.

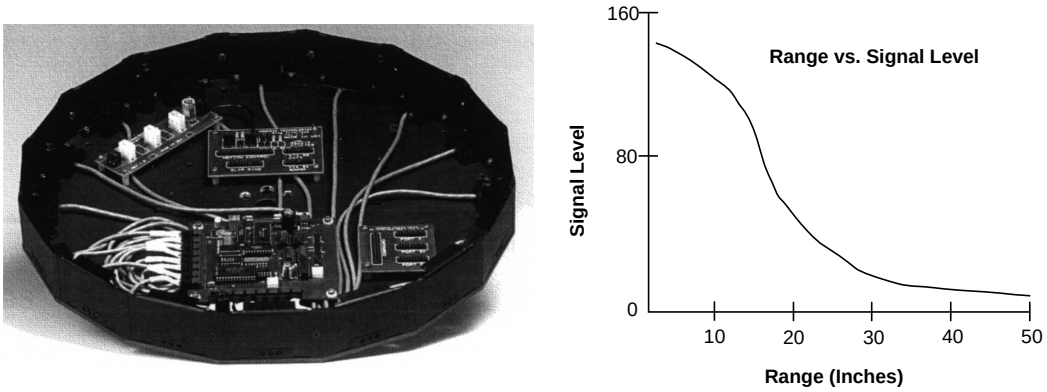

Figura 1.6: Sensori di prossimità: *infrarossi*.

L'intensità della radiazione misurata dal fotodiodo è funzione della distanza della superficie riflettente dell'ostacolo, come si evidenzia dal grafico di figura 1.6, ma dipende fortemente anche dal grado di illuminazione infrarossa dell'ambiente e dal grado di riflettività delle superfici. Per rendere significativo il valore acquisito, ogni ciclo di lettura prevede due misure consecutive effettuate alternativamente con il diodo di emissione acceso e spento; questo consente di produrre una misura differenziale che riduce l'effetto del rumore. Una fase di calibrazione dei sensori in funzione dell'ambiente di utilizzo resta comunque di particolare importanza.

La misura analogica dell'intensità di radiazione viene convertita in un segnale digitale mediante un VCO (*Voltage Controlled Oscillator*) che genera un'onda quadra di frequenza proporzionale al valore misurato. Un contatore ad alta frequenza provvede a misurare il periodo di questa oscillazione per determinare una stima digitale del valore desiderato. La misura complessiva richiede un tempo pari a circa $2.5\ \mathrm{ms}$.

1.3 Organizzazione della tesi

Partendo dalle motivazioni che hanno dato inizio a questo lavoro di tesi, sono descritte le fasi del progetto realizzato, che saranno illustrate nei prossimi capitoli.

Il *capitolo 2* espone l'analisi delle varie tipologie di architetture utilizzate nell'ambito della robotica focalizzando l'attenzione sull'architettura sviluppata nel corso del progetto *OROCOS*.

Il *capitolo 3* espone l'analisi e la realizzazione di un'architettura per la costruzione di applicazioni *behaviour-based* che fornisce un insieme di primitive di alto livello per la creazione dei comportamenti e la gestione delle comunicazioni tra essi. Il sistema è stato progettato in previsione di una sua possibile estensione verso architetture di tipo ibrido.

Nel *capitolo 4* è riportato il risultato ottenuto costruendo comportamenti di *Wall-Following* e di *CenterFollowing* per il Nomad con l'architettura progettata in precedenza. L'applicazione ha lo scopo di verificare il funzionamento di tale libreria in un contesto di utilizzo reale. Successivamente i due comportamenti sono utilizzati contemporaneamente attraverso un modulo di arbitraggio.

La tesi si conclude (*capitolo 5*) con una breve discussione dei risultati ottenuti e delle possibili evoluzioni del progetto.

Capitolo 2

Architetture Robotiche

2.1 Evoluzione delle architetture

La ricerca ha sviluppato architetture atte a guidare sistemi robotici, che debbono operare in un ambiente reale, per sua natura eterogeneo e complesso: cioè in un ambiente inacessibile (i sensori sono imperfetti e percepiscono solo gli stimoli nelle vicinanze del robot), non deterministico (un robot deve considerare l'incertezza nei valori su cui basa le sue decisioni) e dinamico (un robot deve sapere quando è meglio deliberare e quando deve reagire immediatamente).

La necessità di operare direttamente sul mondo reale impone limiti temporali che devono essere rispettati dai tempi di risposta del sistema; inoltre la percezione dell'ambiente di lavoro è soggetta ad errori di misura che devono essere attentamente considerati per la buona riuscita dei vari algoritmi realizzati. Per attuare compiti complessi, o analizzare ed elaborare informazioni percettive provenienti dai sensori più evoluti, sono necessari algoritmi dalla forte componente computazionale che, imponendo tempi lunghi di elaborazione, mal si adattano alla natura dinamica dell'ambiente di lavoro.

Per fronteggiare i problemi in questione e per far convivere più componenti dissimili al fine di realizzare lo scopo comune, sono state proposte numerose architetture che hanno seguito un'evoluzione continua nel corso degli ultimi decenni. Le proposte per i sistemi di governo possono essere raggruppate in tre categorie principali: *architetture deliberative*, *architetture reattive* e *architetture ibride*.

La *tecnica deliberativa* [5], proveniente nella maggior parte dei casi dall'intelligenza artificiale, rappresenta l'approccio classico che consente il funzionamento dei sistemi robotici; sono disponibili inoltre precise metodologie di ausilio al progettista nello sviluppo del sistema di controllo. La procedura di controllo si suddivide in tre fasi distinte, che saranno eseguite in modo sequenziale:

- *Sense*: percezione dell'ambiente, che viene rappresentato internamente mediante un modello logico-matematico;
- *Plan*: decisione dei compiti che il robot dovrà eseguire, in genere questa fase viene ricavata tramite algoritmi di pianificazione e inferenza logica;
- *Act*: intervento sul mondo reale.

Un'architettura deliberativa contiene, esplicitamente rappresentato, un modello simbolico del mondo. L'architettura di controllo assume decisioni attraverso un ragionamento logico, basato sulla conformità a determinate condizioni e/o su una manipolazione simbolica. Poter realizzare azioni basandosi sul solo ragionamento logico è un'idea molto interessante, ma con due problemi ancora lontani dall'essere totalmente risolti: quali informazioni del mondo reale rilevare affinchè la descrizione accurata consenta di prendere decisioni adeguate e come rappresentare le informazioni e porle in relazione tra loro in modo da costruire il modello simbolico su cui si basa il ragionamento logico.

La rappresentazione simbolica dell'ambiente, unita alle tecniche di ragionamento logico, è un potente strumento per realizzare compiti complessi e per consentire l'utilizzo di tecniche di apprendimento, vi sono però alcuni importanti limiti che hanno fatto vacillare il puro approccio deliberativo applicato alla robotica. La natura fortemente dinamica del mondo reale tende a far decadere velocemente la validità del modello, che deve essere continuamente ricostruito affinchè le fasi di pianificazione possano produrre risultati corretti: gli algoritmi di *sensing* e *planning* sono piuttosto onerosi, e impediscono al robot di effettuare rapidamente i compiti più semplici, riducendone in modo sensibile la reattività.

Verso la metà degli anni ’80, realizzando una piccola rivoluzione rispetto al passato, ha riscosso molto successo l’introduzione di *architetture reattive* per il governo adeguato di sistemi robotici [6] soggetti anche a vincoli temporali e interagenti con ambienti dinamici. Nelle tecniche deliberative l’azione di deliberazione è insensibile alle variazioni che avvengono in tempo reale e che inevitabilmente si presentano in un ambiente dinamico, rendendo la pianificazione predeterminata a lungo termine obsoleta, indipendentemente dalla complessità del modello che la supporta.

Le architetture reattive comprendono una speciale categoria che non possiede un modello simbolico dell’ambiente in cui opera e che non utilizza un ragionamento simbolico complesso; le decisioni vengono prese in risposta agli stimoli sensoriali. Per garantire una veloce risposta agli stimoli sensoriali si omette la rappresentazione complessa dell’ambiente, che per sua natura richiede grandi risorse di calcolo; vengono così cablate all’interno del codice una serie di regole di *stimolo-risposta*, che permettono al robot di rispondere immediatamente alle situazioni più significative che vengono individuate dall’apparato sensoriale.

In genere le architetture reattive sono formate da un insieme di moduli, intrinsecamente concorrenti, che operano a differenti livelli di priorità in modo tale che i moduli a livelli più bassi possano inibire quelli a livelli più alti. Ogni livello ha un obiettivo o comportamento: ai livelli più bassi si trovano i comportamenti primitivi, ai livelli superiori vi sono comportamenti più complessi che per il loro funzionamento sfruttano il lavoro prodotto dai componenti i livelli inferiori; viene così portato avanti uno scopo comune, garantendo potenzialmente un ampio margine di robustezza alle situazioni impreviste che si possono presentare.

Pur offrendo notevoli vantaggi, legati alla semplicità ed alla scarsa necessità di risorse computazionali, le architetture di tipo puramente reattivo sono una efficiente risposta unicamente per i sistemi che debbono perseguire scopi semplici; esse sono fortemente limitate nella complessità dei compiti che possono svolgere dall’assenza quasi totale di uno stato interno e dall’incapacità di pianificare azioni complesse e di apprendimento.

Uno sviluppo interessante di questa tecnica di progetto è rappresentato dall’architettura *subsumption*, presentata per la prima volta in [2]: il sistema è costituito da un certo numero di *comportamenti* (*behaviours*) che, eseguiti in parallelo, leggono i dati sensoriali e comandano gli attuatori in modo indipendente l’uno dall’altro. Lo

sviluppo dell'architettura avviene con una tecnica *bottom-up*, partizionando i comportamenti in una serie di livelli numerati che saranno sviluppati separatamente; i comportamenti di livello 0 svolgono i compiti più elementari, che possono essere realizzati e collaudati autonomamente, mentre i comportamenti dei livelli superiori eseguono via via operazioni più evolute, in grado di inibire o utilizzare dove serve (sussumere) le funzioni dei livelli inferiori.

I progetti sviluppati facendo uso di questo nuovo approccio hanno dimostrato come sia possibile fare eseguire al robot compiti evoluti, anche senza l'utilizzo di rappresentazioni complesse dell'ambiente, sfruttando la nascita di comportamenti che emergono dall'interazione tra i moduli contenuti nei singoli livelli [7].

Per superare i problemi ed unire i vantaggi degli approcci deliberativo e reattivo sono state utilizzate tecnologie miste all'interno delle quali coesistono comportamenti separati ma interconnessi che mostrano le caratteristiche dell'una o dell'altra tecnica realizzando un approccio ibrido al problema, sposando così sia gli approcci classici che quelli alternativi. I sistemi che vengono realizzati prendono il nome di *architetture ibride* [8, 9] acquisendo così le caratteristiche migliori degli approcci deliberativo e reattivo e conciliandone gli opposti estremi.

Le architetture ibride si compongono di due distinti livelli interagenti: il *livello inferiore* è composto da processi reattivi che realizzano i compiti più semplici ed è in grado di reagire ad un avvenimento senza necessitare di un ragionamento complesso, al contrario il *livello superiore*, mediante algoritmi complessi ed in genere più lenti, decide la strategia di lungo termine che il robot deve attuare per raggiungere lo scopo, sfruttando le tecniche di ricerca proposte dal filone principale dell'intelligenza artificiale simbolica.

L'attivazione dei comportamenti reattivi viene controllata dal livello superiore, che vede quello inferiore come un insieme di moduli elementari da utilizzare per svolgere attività più complesse. Molto spesso, per facilitare la coesistenza dei due livelli, viene introdotto un *livello intermedio* che ha il compito di ricevere comandi dal sottosistema deliberativo e gestire la sequenzializzazione dei componenti reattivi che devono essere attivati. In pratica, questo livello ha il compito di costruire dinamicamente un grafo di azioni elementari che vengono eseguite in sequenza o in concorrenza parziale tramite il livello inferiore [10].

L'esperienza suggerisce di creare un'architettura gerarchica ibrida, formata da moduli di controllo intelligenti con specifiche funzioni da svolgere. Questi moduli di controllo sono eterogenei: alcuni hanno la responsabilità di filtrare le informazioni sensoriali, stimare lo stato delle variabili e accertare eventi; altri devono produrre piani, allocare risorse libere e compiere attività di scheduling; altri ancora devono fissare le varie priorità, calcolare oneri e benefici, tener traccia delle risorse utilizzate e di quelle libere, correggere errori e agire sui malfunzionamenti.

Alcuni moduli hanno azioni che si protraggono per pochi millisecondi, altri azioni che si protraggono per secondi o minuti, altri azioni che terminano con la conclusione del compito stesso. Ogni modulo di controllo opera in modo concorrente ed indipendente, realizzando le proprie mansioni, in relazione allo stato dell'ambiente; perciò il sistema non è caratterizzato dall'inflessibilità dei sistemi centralizzati fortemente deliberativi. Un'altro aspetto fondamentale è costituito dal fatto che non occorre avere un modulo separato per ogni comportamento e nemmeno che tutti i comportamenti siano insieme attivi come avviene nelle architetture reattive, dato che esiste un sistema di controllo centrale che individua i moduli di controllo necessari a raggiungere lo scopo. Le architetture ibride sono un campo attivo di ricerca e di continuo sviluppo, in quanto comprendono sia la flessibilità della componente reattiva, determinando comportamenti intelligenti anche in ambienti dinamici e complessi, sia la solidità della componente deliberativa, garantendo caratteristiche di predicibilità.

Uno degli inconvenienti maggiori nell'unire la componente deliberativa e quella reattiva sta nella complessità di sintesi che ne deriva. Un sistema in tempo reale, basato sulla componente reattiva, deve rispettare i vincoli temporali imposti dall'ambiente; tutto ciò incontra difficoltà ad adattarsi ai sistemi deliberativi per due motivi: molte delle tecniche di intelligenza artificiale non permettono di ricavare tra i tempi di risposta quelli peggiori e spesso queste tecniche presentano una forte varianza nei tempi di risposta, rendendo un sistema volto a garantirne il funzionamento anche nel peggiore dei casi, poco utilizzato quando il sistema stesso svolge compiti complessi ma rientranti nella norma.

Per questi motivi i metodi di programmazione che fanno convivere le componenti deliberative e reattive costituiscono un tema di ricerca attivo e di forte interesse.

Riassumendo analizziamo nel dettaglio le caratteristiche, i pregi e i difetti delle tre tipologie di architetture trattate in precedenza ricordando che i sistemi deliberativi sono da preferire quando il mondo può essere modellato in modo accurato, l'incertezza è bassa ed esistono alcune garanzie sulla staticità dell'ambiente di lavoro durante l'esecuzione del compito. I sistemi reattivi hanno prestazioni robuste in domini complessi e dinamici, ma adottano ipotesi forti che non sono sempre vere quali: l'ambiente non ha consistenza temporale e stabilità, le informazioni sensoriali sono utili solo per il *task* corrente, localizzare un robot nel modello del mondo è difficile e la rappresentazione simbolica della conoscenza sul mondo è di scarso o nessun valore. Le architetture di tipo ibrido tentano di prendere le caratteristiche migliori delle due, conciliandone gli opposti estremismi.

architetture deliberative:

- *caratteristiche*: desumono informazioni di tipo simbolico secondo un approccio derivato dall'intelligenza artificiale classica, sono organizzate in livelli di astrazione separati l'uno dall'altro, le azioni sono decise ai livelli più alti, basate su modelli astratti rispetto all'ambito in cui operano, sono concentrate sugli aspetti globali del problema.

- *pregi*: usano modelli matematicamente trattabili, sono facilmente dotabili di obiettivi, sono ben conosciute e sono in grado di rivelare situazioni di stallo.

- *difetti*: non sono immuni da errori, necessitano di grandi risorse di calcolo, non si adattano bene ai mutamenti del mondo reale, sono rigide nei comportamenti e poco adattabili.

architetture reattive:

- *caratteristiche*: si basano sulla teoria dei controlli ed elaborano informazioni di tipo numerico, sono organizzate in livelli funzionali che interagiscono tra di loro, le azioni sono decise ad ogni livello in base ai dati sensoriali disponibili, non utilizzano rappresentazioni astratte rispetto all'ambito in cui operano e sono concentrate sugli aspetti locali della navigazione.

- *pregi*: reagiscono velocemente agli eventi, si adattano bene a situazioni nuove e impreviste, sono resistenti agli errori, esibiscono comportamenti opportunistici, possono essere sviluppate e testate incrementalmente e i sottosistemi sono riutilizzabili.

- *difetti*: usano modelli approssimati a causa della complessità della teoria, non sono facilmente dotabili di finalità di alto livello e possono restare intrappolate in condizioni di stallo.

architetture ibride:

- *caratteristiche*: uniscono i metodi dell'intelligenza artificiale simbolica e l'utilizzo dell'astrazione conservando i benefici dei sistemi reattivi quali: tempi di risposta veloci, robustezza e flessibilità. Le architetture di tipo ibrido tentano di prendere le caratteristiche migliori delle due, conciliandone gli opposti estremismi; sono caratterizzate da una struttura multilivello dove il livello più alto è di pianificazione, quello più basso è reattivo e quello intermedio si occupa di interfacciare gli altri due.

- *pregi*: offrono un alto supporto per la modularità ed una elevata robustezza, dovuta alla presenza di moduli per monitorare le prestazioni del comportamento reattivo. Da molti è ritenuta la migliore classe di architetture per compiti ed ambienti di tipo generale; elaborazione asincrona e modularità consentono di svincolare i tempi della pianificazione da quelli delle reazioni.

- *difetti*: il loro principale difetto è che si prestano poco ad una modellazione formale.

2.2 Stato attuale

L'aumento di complessità delle architetture software per la robotica è avvenuto contestualmente alla crescita dei sistemi cui corrispondono; ogni gruppo di ricerca tende a proporre soluzioni proprie che riflettono visioni personali e innovative, pur modellando i concetti di base discussi nel paragrafo precedente. La grande maggioranza dei sistemi proposti in letteratura si basano su strutture di tipo *behaviour-*

based che evolvono verso un modello ibrido (suddiviso in tre macro livelli) quando la difficoltà dei compiti da svolgere diventa elevata.

Per inquadrare con più facilità la struttura tipica di un moderno sistema robotico viene illustrata brevemente l'architettura progettata presso il *Laboratory for Analysis and Architecture of Systems* (LAAS/CNRS) [11] per la robotica mobile. Lo schema generale, illustrato in figura 2.1, evidenzia la rigida stratificazione che viene compiuta sia sulle componenti software che sul resto del sistema, in particolare sono presenti i tre livelli che caratterizzano le architetture ibride:

- *Functional Level*. Include un insieme di azioni elementari che il robot è in grado di compiere (*image processing*, *obstacle avoidance*, *motion control*, etc.), raggruppate in unità autonome e controllabili chiamate *Moduli*. Per permettere a questo livello un certo grado di indipendenza rispetto all'hardware e rendere i moduli adattabili ad altri robot, l'architettura inserisce uno strato software di interfaccia verso il sistema fisico: il *logical robot level*.

- *Execution control Level*. Controlla e coordina l'esecuzione delle funzioni distribuite nei moduli del livello inferiore in accordo con le sequenze di comandi che vengono inviate dal livello superiore. Questo livello è composto di un unico sistema, l'*Executive*, che funge da interfaccia tra il sistema reattivo, operante a frequenze elevate ($10 \div 100$ Hz), ed il sistema deliberativo, i cui processi mostrano tempi di elaborazione tipicamente superiori a 100 ms.

- *Decision Level*. Raggruppa gli algoritmi necessari alla pianificazione delle operazioni che il robot deve svolgere ed alla supervisione della loro esecuzione. In riferimento alla complessità dei compiti per i quali il sistema è realizzato questo livello può essere suddiviso in più strati che, operando a frequenze diverse, forniscono funzioni via via più evolute ed astratte: il livello superiore ha il compito di dialogare con l'operatore.

Per rendere più semplice lo sviluppo delle componenti dell'architettura, realizzata prevalentemente in linguaggio *C*, vengono utilizzati alcuni strumenti che provvedono a costruire il codice necessario partendo da una descrizione di alto livello realizzata mediante linguaggi appositamente definiti.

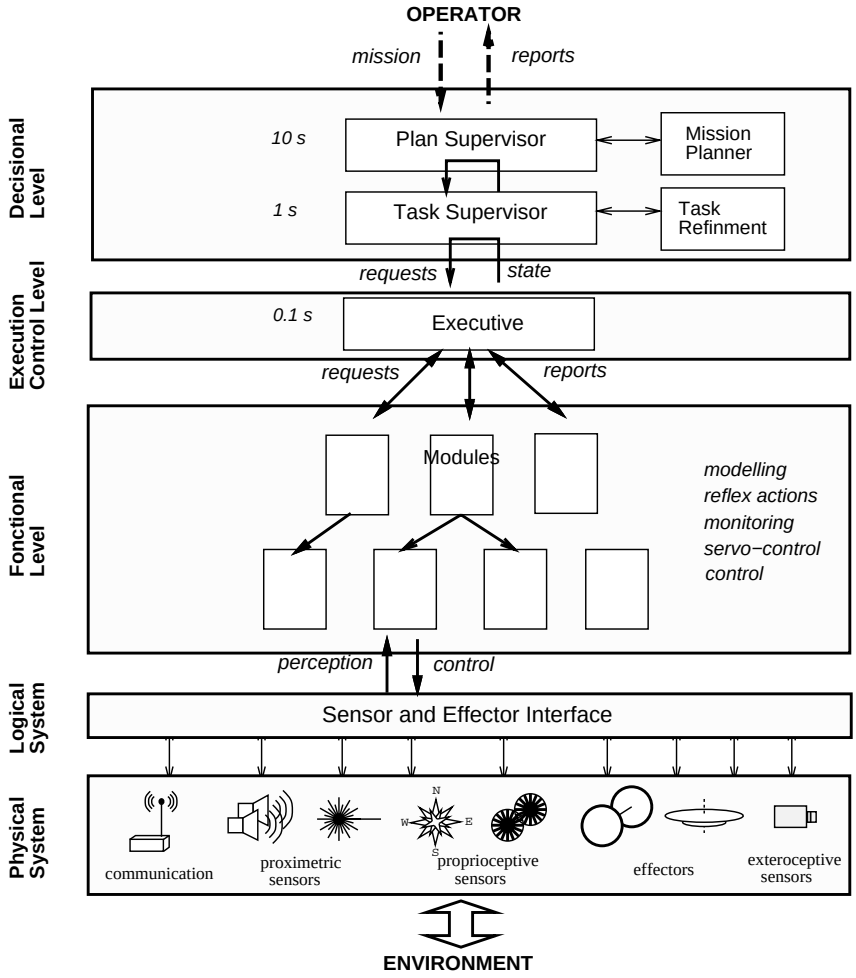

Figura 2.1: L'architettura di controllo sviluppata presso i laboratori del *LAAS/CNRS*.

Il codice associato ai singoli moduli viene infatti prodotto tramite un tool, denominato *Genom*, a partire da una descrizione formale dei servizi che esporta e dall'insieme degli algoritmi che li rendono realizzabili (chiamati *codel*). Per descrivere il comportamento dell'Executive è stato sviluppato un altro linguaggio formale (*Kheops*) ed un tool che provvede a tradurne le regole in codice *C* compilabile.

2.3 Orocos::SmartSoft

2.3.1 Motivazioni e Aspetti Generali

Un altro sistema attentamente studiato in questo lavoro di tesi e suo punto di partenza è il framework in corso di sviluppo nell'ambito del progetto *OROCOS* [1], nato con l'intento di produrre un'architettura *Open Source* che fornisca una base comune per lo sviluppo di applicazioni valide sia per robot mobili che per quelli manipolatori. Molte delle architetture esistenti vengono realizzate all'interno di un solo gruppo di ricerca; ne deriva che la struttura del sistema risulta influenzata dal contesto originale di utilizzo e che sia poco portabile verso altri sistemi. Al contrario, OROCOS ha come scopo principale la ricerca di soluzioni quanto più possibile generiche ed aperte, che sfruttano fin dove è possibile gli standard a disposizione. Si è ricorso quindi al linguaggio *C++* per sviluppare le varie parti del sistema, favorendo l'uso di librerie portabili come *POSIX* ed *ACE* per mantenere un alto grado di indipendenza dalla piattaforma di esecuzione. OROCOS ha scelto inoltre di realizzare in *CORBA* la maggior parte degli schemi di comunicazione, per permettere l'interoperabilità tra componenti eterogenei.

Il progetto, ancora nelle prime fasi di sviluppo, è suddiviso in sotto-sistemi che vengono portati avanti separatamente da alcuni gruppi di ricerca. Il nucleo del sistema, realizzato presso la *Katholieke Universiteit Leuven*, è rappresentato dal *Real-time motion kernel*: si tratta di un insieme di librerie che consentono l'inserimento dei moduli di controllo delle componenti hardware del robot all'interno di un ambiente che ne gestisce l'attivazione periodica in tempo reale. La libreria realizza in pratica uno strato di interfaccia verso il sistema operativo utilizzato (attualmente *RTLinux* e *RTAI*), fornendo gli strumenti di base per la programmazione concorrente in ambiente real-time, come primitive di sincronizzazione e task ad attivazione periodica. A questo livello la comunicazione tra le componenti, per garantirne un'elevata efficienza, avviene mediante meccanismi a memoria condivisa.

Il *Task execution sequencing* ha lo scopo di coordinare l'esecuzione delle componenti di livello *kernel* ed è realizzato dal laboratorio *LAAS/CNRS* [12], che può contare sull'esperienza proveniente da numerosi progetti, tra cui l'architettura de-

scritta in precedenza. In particolare il gruppo francese prevede di riutilizzare all'interno di OROCOS il tool di sviluppo real-time *Genom* [13].

Rilevanza particolare è stata riservata per il sottosistema che gestisce le comunicazioni tra i componenti dell'architettura. Presso l'istituto di ricerca *FAW* (Monaco) è stato condotto un minuzioso lavoro di analisi dei requisiti che un'applicazione robotica presenta in termini di primitive di comunicazione; n'è scaturito un documento che descrive l'insieme dei *pattern di comunicazione* che possono diventare utili in un sistema complesso [14]. Da questo progetto sono nate separatamente le due librerie *Orocos::SmartSoft* [15] e *Orocos@KTH* [16], sviluppate rispettivamente presso il già citato *FAW* e l'istituto svedese *KTH*, che forniscono due differenti soluzioni concrete, entrambe realizzate al di sopra di un *middleware* CORBA.

Il progetto OROCOS è tuttora incompleto, tuttavia lo sforzo progettuale effettuato fino ad ora ha messo in luce alcuni aspetti di notevole importanza, che sono stati attentamente considerati durante questo lavoro di tesi. In particolare l'apertura verso la filosofia di sviluppo *Open Source*, specialmente se viene favorita da un sistema che esalta la granularità dei componenti, consente un'evoluzione progressiva che può sfruttare a vari livelli l'apporto di persone che non sono direttamente legate al progetto.

L'enfasi sulla *granularità* del sistema e delle sue componenti è del resto ben visibile in molte architetture moderne: n'è un chiaro esempio *CLARAty* [17, 18] che, pur seguendo un approccio di tipo ibrido, propone di eliminare il livello intermedio, monolitico e difficile da modificare, per favorirne la scomposizione all'interno dei moduli che compongono gli altri livelli. È ugualmente interessante la soluzione adottata nella realizzazione di *OSCAR* [19] che promuove lo sviluppo di moduli distribuibili anche in forma binaria, per formare una sorta di database di comportamenti che può essere progressivamente accresciuto.

Caratteristiche fondamentali

Le principali caratteristiche di comunicazione del framework OROCOS sono:

- Pattern predefiniti di comunicazione per garantire l'interazione tra i vari com-

ponenti e per separare la realizzazione interna di un componente dal suo comportamento esterno.

- Pattern di comunicazione che forniscono una chiara descrizione dei metodi reperibile nell'interfaccia dei componenti.
- I pattern di comunicazione sono *thread safe* per alleggerire il costruttore del componente da complessi problemi di sincronizzazione.
- Le comunicazioni sono basate su *CORBA* utilizzando *IDL* per descrivere i dati utenti negli oggetti di comunicazione; questo permette indipendenza dalle piattaforme e dalla localizzazione.
- Si utilizzano oggetti communication, invece di metodi remoti di invocazione, riducendo così al minimo il traffico, inoltre è possibile utilizzare un'arbitraria struttura dati come *STL* senza essere costretti ad utilizzare sempre IDL.
- I pattern racchiusi nel framework consentono di utilizzare architetture multilivello; questi pattern addizionali permettono il funzionamento su più architetture.

Le principali caratteristiche del framework OROCOS sono:

- *Obiettivi*: l'obiettivo dei pattern di comunicazione è quello di garantire una puntuale e trasparente trasmissione dei dati tra i moduli, un modulo è tecnicamente realizzato come un processo e può essere attivo su più *thread*; i moduli possono essere distribuiti su più piattaforme tra loro differenti.

  La trasparenza permette di nascondere la struttura del network agli utenti e questo grazie ad un approccio *object-oriented* ove è possibile accedere ad altri moduli con chiamate a metodi, come se questi ultimi fossero contenuti nello stesso modulo. Per semplificare ulteriormente l'interfaccia di programmazione tutti i meccanismi di sincronizzazione e di aggancio riguardanti la struttura *multithread* sono già integrati all'interno dei pattern di comunicazione.

- *Reattività*: la ricezione dei messaggi è gestita in modo indipendente; ciò permette ad un modulo di ricevere messaggi anche se quest'ultimo sta svolgendo

contestualmente altre attività, eliminando così ritardi nella ricezione. Non vi è *deadlock*, quindi ogni modulo può ricevere contemporaneamente i propri messaggi.

- *Topologia*: la comunicazione è basata su messaggi in un ambiente *peer-to-peer*; ogni coppia di moduli, comunicanti tra loro, riceve una propria connessione *socket* che viene inizializzata una prima volta da un server centrale e successivamente chiusa quando un modulo viene disattivato. Quindi la comunicazione tra moduli è molto veloce non essendo gestita da un server centrale che spesso genera ritardi.

  La comunicazione utilizza molto spesso un'implementazione socket, che è ottimizzata per lo scambio di messaggi tra moduli sullo stesso *host*; questo evita inutili ritardi e non necessita di molta banda network, al contrario del server centrale che utilizza molta banda quando risiede su un differente *host* rispetto ai moduli che partecipano alla comunicazione.

- *Primitive*: tutte le comunicazioni tra moduli sono supportate da una serie di primitive; il set dei pattern deve essere piccolo e facilmente utilizzabile tenendo conto delle necessità degli utenti quali la semplicità d'uso e la chiarezza della struttura.

- *Threading*: Tutte le primitive di comunicazione sono *threadsafe*, thread multiple possono accedere allo stesso server contemporaneamente senza bisogno di ulteriori sincronizzazioni. Anche richieste annidate dalle stesse o da differenti thread non costituiscono un problema; ciò riduce notevolmente i disagi in strutture ricorsive bloccanti e semplifica la struttura interna del modulo. Anche i programmatori inesperti traggono immediati vantaggi, non essendo frastornati da complesse e difficili regole di sincronizzazione.

- *Timeout*: OROCOS si basa su una comunicazione sicura senza timeout; un alto livello di astrazione per i timeout si trova nell'interfaccia utente. Attualmente non sono disponibili timeout per i metodi bloccanti, quindi una *query* bloccante non ritorna finchè la risposta non è ricevuta; è permesso al meccanismo di configurazione di abbandonare una *query* bloccante quando un modulo deve essere disattivato. Se un processo cliente ha bisogno dei timeout

per una *query* dovrà utilizzare *query* non bloccanti, inoltre se una *query* deve essere soddisfatta entro un determinato tempo il server, invece di inviare la risposta, può rispondere con uno *status code* indicando che la risposta non può essere prodotta nei tempi previsti.

Principi base

I pattern di comunicazione sono posti all'estremità più alta del meccanismo di comunicazione, quindi i seguenti principi sono fondamentali:

- *Scambio di messaggi*: la comunicazione di basso livello è basata sullo scambio di messaggi e non su chiamate a procedure remote o a memoria condivisa; lo scambio di messaggi consente di disaccoppiare l'invocazione del metodo e il suo completamento dalla sua esecuzione.
- *Richieste/risposte asincrone*: le chiamate bloccanti nel lato client non corrispondono ad altrettante chiamate bloccanti nel lato server, ma consentono di disaccoppiare l'invocazione del metodo dal suo completamento; questo assicura il normale funzionamento del server. Fortunatamente i messaggi possono essere ricomposti senza cadere in tempi morti.
- *Peer to Peer network*: per ottenere ottimi risultati si deve evitare di ricorrere ad un server centrale dopo la fase di inizializzazione; l'utilizzo di un server centrale non è accettabile in applicazioni robotiche.
- *Gestione delle eccezioni*: non si utilizzano eccezioni nei pattern di comunicazione, sebbene il possibile utilizzo di esse, dipenda dalla piattaforma utilizzata e dal compilatore; Il motivo di questa scelta è la difficoltà nel gestire le eccezioni su sistemi piccoli o realtime.

2.3.2 Servizi offerti

L'implementazione attuale di OROCOS si basa su CORBA, anche se vengono utilizzate le socket per le comunicazioni. Queste ultime non sono visibili agli utenti finchè tutte le comunicazioni non siano gestite dalle primitive, che possono a loro volta essere realizzati con CORBA . Alcune specifiche parti del sistema operativo sono

basate su ACE, un framework riutilizzabile. Le chiamate bloccanti nel lato client sono sempre separate in un messaggio di richiesta ed uno di risposta, consentendo così al server di non rimanere bloccato dall'arrivo di messaggi multipli.

Pattern di comunicazione

La complessità di un applicativo software, utilizzato in un generico sistema basato su sensori e motori, può essere limitata solo con l'utilizzo di una serie di componenti discreti. Ciò impone di realizzare un'interfaccia che gestisca la comunicazione tra i componenti; l'interfaccia di ogni componente permette una chiara distinzione fra il suo comportamento esterno e la sua realizzazione interna. Il disaccoppiamento dei componenti è essenziale per limitare la complessità dei sistemi software multicomponente.

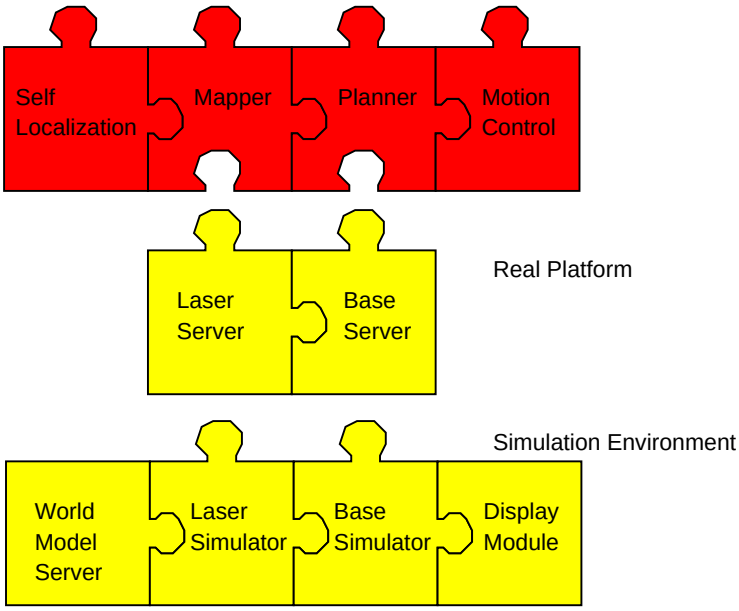

Figura 2.2: Le interfacce standarizzate dei componenti facilitano il rimontaggio dei componenti.

I pattern di comunicazione sono caratterizzati da *oggetti communication* che trasmettono i dati degli utilizzatori; i metodi per accedere al servizio sono forniti dagli stessi pattern di comunicazione e sono identici per ogni componente; que-

sto permette di capire in modo chiaro i servizi forniti dai vari componenti e come utilizzarli.

Le interfaccie standarizzate dei componenti facilitano la spiegazione dei servizi offerti e il rimontaggio dei componenti come mostrato in figura 2.2. Ogni servizio di un componente è basato sulla totale specificazione dei pattern che forniscono metodi di accesso testati e chiaramente strutturati.

I pattern di comunicazione aiutano il costruttore del componente e quello dell'applicazione nel creare ed utilizzare i componenti distribuiti in modo che la semantica dell'interfaccia sia predefinita dai pattern, senza tener conto da dove vengono utilizzati.

Struttura dei pattern di comunicazione

- *Oggetti communication*: Gli oggetti che caratterizzano i pattern di comunicazione sono chiamati *communication objects* e contengono sia la struttura dati da trasmettere sia i metodi di accesso.

- *Servizio*: Un pattern di comunicazione unito a un oggetto communication è chiamato *Service*. I metodi di accesso per un servizio sono completamente definiti dai pattern di comunicazione mentre gli oggetti communication definiscono totalmente il contenuto da trasmettere.

- *Server-Client/Producer-Consumer/Master-Slave*: Ogni pattern di comunicazione congiunge due terminali e quindi è formato da due parti; a seconda del pattern utilizzato prenderà il nome di *Server-Client/Producer-Consumer/Master-Slave* per distinguere le varie parti in gioco.

Pattern disponibili in Orocos::SmartSoft

Smartsoft è un framework software per realizzare sistemi complessi che prevedono anche sensori e motori, riducendone la complessità grazie ad una serie di template per i più comuni pattern di comunicazione. L'interfaccia di ogni componente permette una chiara distinzione fra il suo comportamento esterno e la sua realizzazione interna, elemento di notevole importanza per quei sistemi composti da numerosi componenti e sviluppati in contemporanea da diverse persone.

Il framework Smartsoft è distribuito in diverse realizzazioni che differiscono fra di loro per il sottostante meccanismo di comunicazione, la versione basata sul middleware CORBA è chiamata *Orocos::SmartSoft*. I seguenti pattern, evidenziati in [14], sono disponibili in Orocos::SmartSoft e sono riportati nella tabella 2.1:

Pattern	**Descrizione**
Send	Trasferisce dati dal client al server senza la ricezione di una conferma dal server. Rappresenta una comunicazione *monodirezionale* utile per inviare comandi e settare configurazioni.
Query	Il client effettua una richiesta al server ed attende la risposta. Rappresenta una comunicazione *bidirezionale* tra un solo client ed un solo server. Può essere utile quando un'informazione viene utilizzata ad una frequenza molto bassa rispetto alla velocità con cui viene prodotta: è più ragionevole che il client la richieda mediante una query piuttosto che venga prodotta una quantità eccessiva di aggiornamenti non necessari.
Autoupdate	Uno o più client si registrano presso un server richiedendo un'informazione che viene inviata non appena nuovi dati sono disponibili. Rappresenta una comunicazione di tipo *push*. È possibile ridurre il traffico, se il client richiede l'informazione ad una frequenza minore rispetto a quella con cui viene prodotta, mediante una richiesta che prevede l'invio dei dati soltanto ad ogni ennesimo aggiornamento (*autoupdate timed*).
Event	Il server individua un evento avvenuto sullo stato del sistema ed informa in modo asincrono il client che ne assicura la gestione. Gli eventi sono utilizzati principalmente per notificare modifiche nello stato che sono rilevanti per coordinare i task in esecuzione.
Configuration	Supporta una relazione di tipo *Master-Slave* tra i moduli e consente l'attivazione selettiva delle loro attività. Il modulo slave, quando un'attività viene disattivata, deve impedire che il proprio stato interno possa diventare inconsistente, deve proteggere l'esecuzione delle sezioni critiche e gestire la terminazione delle comunicazioni esistenti con gli altri moduli.

Tabella 2.1: Pattern di comunicazione identificati in *OROCOS*.

Altri pattern di utilizzo comune sono:

- *State*: Fornisce la possibilità della gestione dello stato interno di un com-

ponente, onde evitare che le attività vengano interrotte all'interno di sezioni critiche al fine di proteggere le risorse richieste. Permette la precedenza del componente esterno rispetto alle richieste interne quando è necessario un cambiamento di stato. Se un'attività è disattivata si può, senza rischi, modificare i parametri e riconfigurare dinamicamente la struttura dati fra i componenti. Il protocollo di stato può comunicare finchè la disattivazione di un componente non richieda la terminazione delle comunicazioni esistenti con gli altri moduli.

Il modulo slave può proteggere l'esecuzione delle sezioni critiche dall'interruzione o dal fallimento, evitando così, che una thread attiva, utilizzi dati inconsistenti o sia interrotta in un punto indesiderato. La richiesta del modulo master, per una specifica configurazione, è realizzata non appena tutti gli stati utilizzati vengono rilasciati.

- *Wiring*: Permette la configurazione dinamica della struttura dati fra componenti; il modulo client, parte del protocollo di comunicazione, può essere utilizzato come una porta per connettersi al server corretto. Wiring è utile per cambiare la struttura dati fra componenti in modo dinamico, per comporre diversi comportamenti di un insieme di funzionalità.

- *Altri*: Una serie di classi specializzate incapsulano la gestione delle thread, forniscono oggetti attivi e meccanismi di sincronizzazione.

La figura 2.3 mostra come l'interfaccia dei componenti sia formata da pattern di comunicazione standarizzati.

Quando ogni interfaccia esterna è basata sugli stessi pattern, tutti i metodi da lei forniti sono caratterizzati da una stessa predefinita e fissata semantica; questo consente di individuare facilmente l'interazione tra i componenti osservandone l'interfaccia e permette la semplice sostituzione di moduli con altri realizzati in modo diverso.

La struttura fondamentale dei pattern di comunicazione è mostrata alla figura 2.4. Questi pattern provvedono all'accesso ai servizi indipendentemente dalla loro localizzazione, su più piattaforme e in ambienti basati su thread.

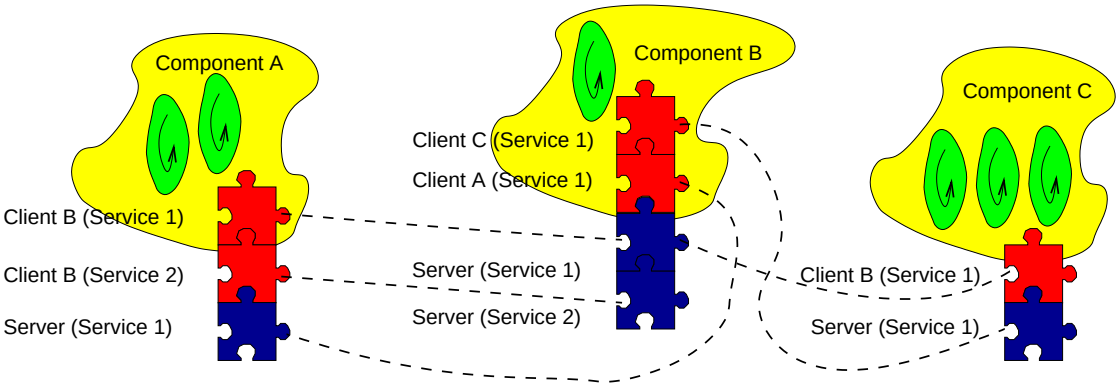

Figura 2.3: Interazione di componenti basata su predefiniti pattern di comunicazione.

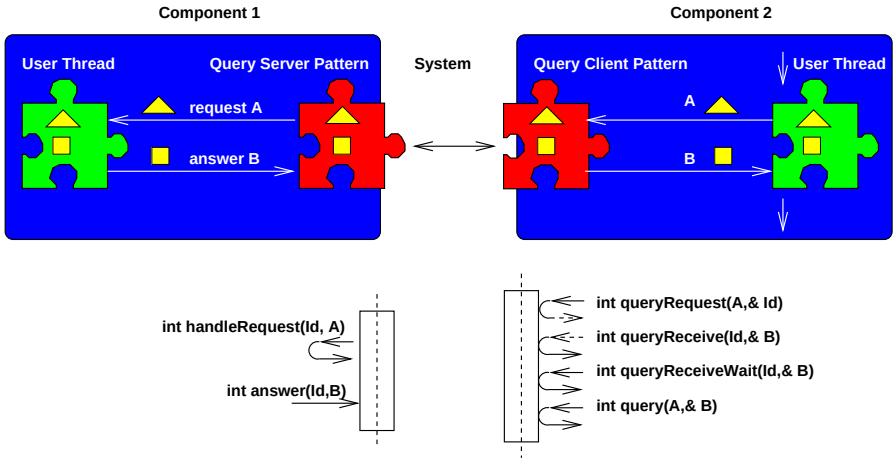

Figura 2.4: Esempio di comunicazione.

Ogni pattern di comunicazione è accompagnato da un oggetto communication; il *query-pattern*, ad esempio, necessita di due oggetti communication: uno contiene il messaggio di richiesta e l'altro la risposta. Finchè gli oggetti communication non sono trasmessi ai vari componenti non si genera *network traffic*.

Rappresentazione del client

Come evidenziato in figura 2.5, ogni modulo è caratterizzato da una classe client, formata dall'insieme dei servizi offerti dal modulo che riguardano il client. Se un client vuole accedere ad un modulo deve solo instanziare la classe client e può così accedere a tutti i servizi che questo modulo offre. In presenza di un unico ogget-

to client è estremamente semplice cambiare il server per ogni specifica necessità. Inoltre è possibile utilizzare due differenti server per lo stesso servizio all'interno del mudulo; questo permette di confrontare fra di loro approcci diversi al medesimo problema.

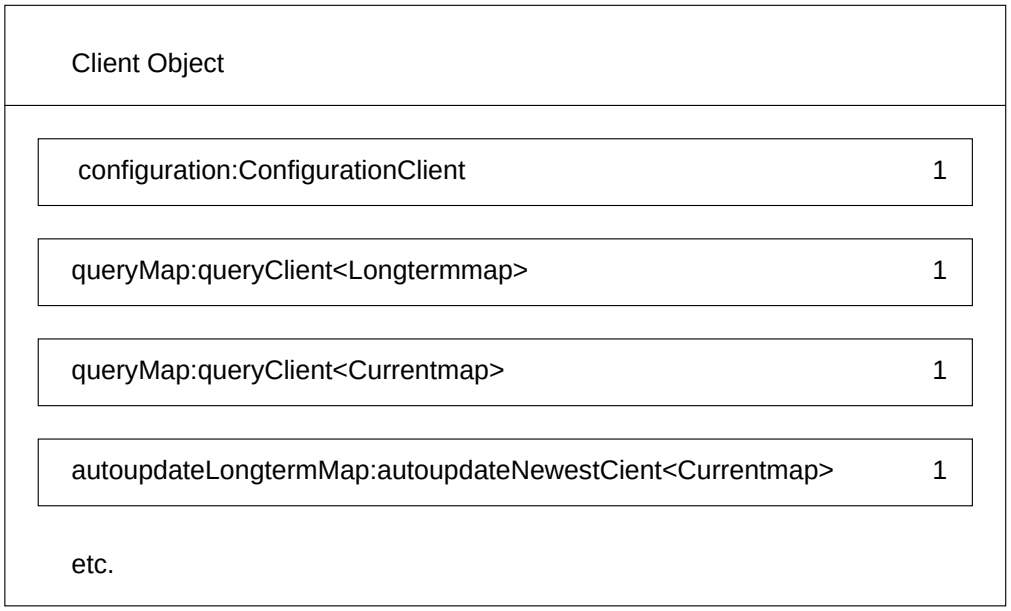

Figura 2.5: Tutti i servizi di un modulo sono raggruppati per formare l'oggetto client.

Servizio broker

Il servizio broker organizza il primo contatto tra moduli costituendo il server centrale per creare il piano di comunicazione tra i moduli, come mostrato in figura 2.6. Questo servizio è utilizzato solo all'inizio della comunicazione; una volta che i due moduli sono connessi essi scambiano i messaggi direttamente senza bisogno di un server centrale.

- Il server centrale rimane in ascolto sulla porta riservata 1387 e accetta la registrazione di nuovi moduli.
- Se il modulo A è attivo occupa una porta libera e informa il broker, via porta 1387, della sua disponibilità; così il broker può rintracciare il modulo A (indirizzo IP e porta).

- Il broker, una volta registrati tutti i messaggi, assegna un unico elemento identificativo per ogni tipo di messaggio, questo permette sia al server che al client di un pattern di comunicazione d'identificare i propri messaggi e di assegnarli all'apposito gestore.

- Ogni pattern di comunicazione ha il proprio set di messaggi predefiniti, che vengono utilizzati per gestire la comunicazione tra server e client; i messaggi generici sono personalizzati quando i modelli di comunicazione sono instanziati da uno specifico oggetto dati.

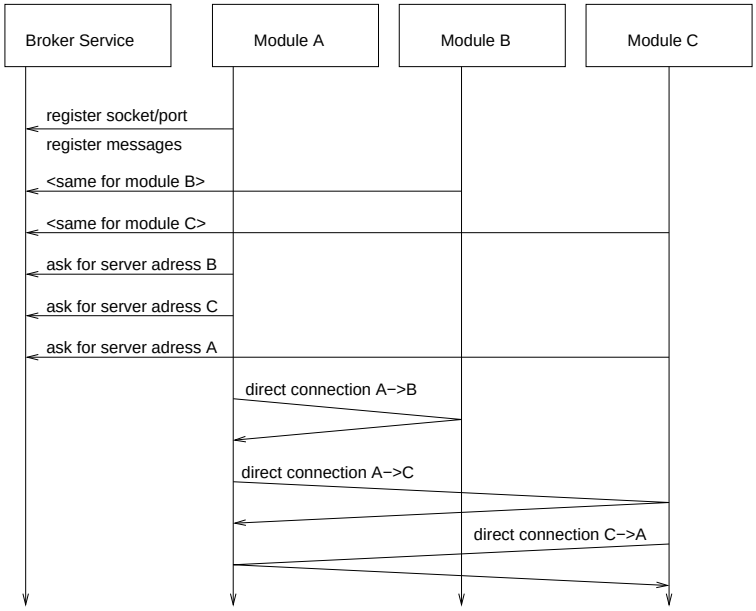

Figura 2.6: Funzionamento del Broker service.

2.3.3 Realizzazione

CORBA e i pattern di comunicazione

L'idea principe dalla quale discende il progetto CORBA è quella di garantire un accesso semplice e trasparente agli oggetti distribuiti. Tutti i metodi di un oggetto descritti all'interno del file IDL possono essere raggiunti da ogni posizione, tuttavia

tutti gli oggetti rimangono all'interno del processo ove sono stati creati. In particolare i metodi restituiscono *strutture* il cui formato è descritto tramite IDL, non possono restituire un oggetto, sebbene si possa avere un *reference* ad un oggetto situato in un altro processo.

CORBA non fornisce un meccanismo di accesso *threadsafe*; la sincronizzazione di accessi concorrenti ad uno stesso oggetto è occultata agli utenti e situata all'interno dei pattern di comunicazione anche quando CORBA è utilizzata a basso livello nella comunicazione.

OROCOS non trasferisce il *reference* di un oggetto, ma il suo contenuto e crea un nuovo oggetto del tipo richiesto dal client; accedere a questo nuovo oggetto non genera assolutamente traffico, tuttavia spetta al client cancellare l'oggetto una volta utilizzato. Un aspetto ancora più importante è l'estensione di un oggetto con l'ereditarietà; infatti gli oggetti restituiti possono essere estesi fornendo altri metodi di accesso. Ciò è attuato a livello client evitando di modificare il lato server quando un client necessita del proprio metodo d'accesso, traendo così notevoli vantaggi.

Tipico esempio è la modalità *autoupdate*, dove il client può accedere ai dati più recenti disponibili; si riduce così il traffico fra moduli creando un oggetto che contiene i dati nuovi prodotti dal server. I messaggi d'aggiornamento tra moduli sono prodotti unicamente quando sono disponibili nuovi dati.

Scopi principali della realizzazione basata su CORBA

Lo scopo principale dei pattern di comunicazione è di fornire un ampio set di metodi di accesso, basati su una interfaccia *object oriented*; una query è formata da una richiesta e da un oggetto come risposta; gli oggetti di risposta vengono sempre creati nel lato cliente e vi si accede localmente. Non si utilizza una realizzazione ingenua basata su CORBA, valga ad esempio la creazione di oggetti al cui interno troveremo nuovi dati che costituiranno la risposta del server, con limitazione contestuale del traffico. È molto importante separare tra di loro i messaggi di richiesta con quelli di risposta, utilizzando le funzionalità messe a disposizione dai metodi di CORBA.

Questi metodi sono evocati a livello dei pattern di comunicazione, garantendo gli stessi meccanismi di una realizzazione a scambio di messaggi basata su socket. Il grande vantaggio di un sistema basato su CORBA, è la possibilità di utilizza-

re IDL per descrivere le strutture dati da scambiare, rendendo tutto maggiormente compatibile.

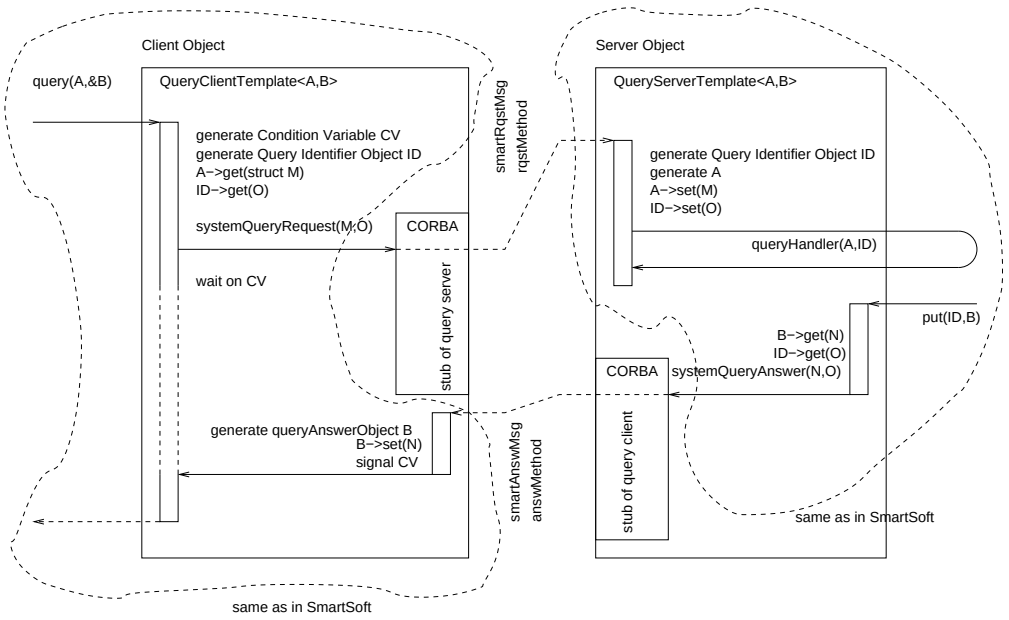

Figura 2.7: Pattern di OROCOS con CORBA.

Come mostrato in figura 2.7, l'IDL è necessario per descrivere la struttura dati che è oggetto di scambio tra il server ed il client del pattern di comunicazione. La parte d'interfaccia non contiene metodi individuali accessibili all'utente, ma solo metodi usati internamente al pattern stesso. I metodi d'interfaccia consentono di coordinare la comunicazione tra server e client e non hanno la possibilità di accedere ad un oggetto server da un modulo client.

I dati utente sono accessibili da metodi forniti dagli *oggetti dati* come gli oggetti *richiesta query o risposta query*. Quindi all'interno del framework OROCOS si devono definire gli oggetti communication con la descrizione del formato dati utilizzato per le comunicazioni tra server e client. La descrizione del formato dati non avviene tramite un linguaggio proprietario ma con IDL.

Nell'esempio in figura 2.7 l'utente realizza i due oggetti communication A e B che rappresentano gli oggetti *queryRequest* e *queryReceive*; A e B devono contenere i metodi *get e set* per accedere alle strutture dati M e N che vengono scambiate nella

comunicazione. Ovviamente è necessaria una descrizione IDL delle due strutture M e N .

Comparazione tra CORBA e socket

Vengono qui riassunti alcuni aspetti da tener in considerazione quando si decide di realizzare un meccanismo di comunicazione basato su socket o su CORBA:

- -*CORBA*: i pattern di comunicazione sono all'apice del meccanismo di comunicazione; alcuni strumenti di comunicazione di CORBA non sono inseriti nei pattern di Orocos per renderli più semplici ed avere una struttura più chiara.
- +*CORBA*: il grande vantaggio di CORBA è la possibilità di utilizzo delle librerie IDL.
- -*SOCKET*: crea un proprio meccanismo di comunicazione basato su socket e obbliga a realizzare routine per la gestione delle strutture dati; in OROCOS queste routine sono ormai superate e occorre aggiornarle, mentre con CORBA ciò non è necessario grazie alla presenza di IDL.
- +*SOCKET*: realizzare un meccanismo di comunicazione basato su socket è facilitato dalla presenza di una serie di strumenti di ACE già pronti quali il protocollo *Reactor/Connector* o oggetti per utilizzare messaggi bloccanti, tuttavia occorre perfezionare il meccanismo di scambio dei messaggi.
- -*CORBA*: la realizzazione basata su CORBA è un ottimo sistema software, nel contempo la sua struttura è complessa, tale da rendere difficile l'acquisizione di informazioni sulla sua struttura interna; manca inoltre un meccanisco di accesso *threadsafe*, si deve così introdurlo all'interno dei pattern di comunicazione.
- +*SOCKET*: un meccanismo basato su socket permette un accesso completo al meccanismo di comunicazione e al formato dei messaggi, ciò è particolarmente utile se si vuol tener traccia dei messaggi e fornire servizi di debug.

La ricerca di una fine scomposizione dei componenti del sistema robotico ha avuto l'effetto ulteriore di far crescere l'interesse del settore per le tecniche di design e programmazione orientate agli oggetti: l'approccio *object-oriented* si presta particolarmente a rappresentare entità autonome che collaborano in un ambiente complesso ed eterogeneo, quindi può essere particolarmente efficace per realizzare le componenti di un'applicazione robotica (è immediato il confronto con il concetto di *behaviour*) in cui devono convivere problemi legati alla concorrenza, alla distribuzione su di un sistema piuttosto vasto, all'esecuzione in tempo reale, ed in cui un approccio rigoroso è il più delle volte indispensabile [20, 21, 22, 23].

I linguaggi di programmazione orientati agli oggetti possiedono inoltre una maggiore potenza espressiva rispetto ai linguaggi procedurali: questo consente molto spesso di sfruttare direttamente i costrutti del linguaggio per costruire l'architettura del sistema, riducendo la necessità di linguaggi creati ad hoc per descrivere le componenti dell'applicazione. In questo senso è emblematico l'esempio di *Genom*: nonostante i singoli algoritmi di controllo (codel) vengano realizzati in linguaggio *C* per le specifiche formali del modulo è stato introdotto un ulteriore linguaggio descrittivo, delegando ad un tool automatico la produzione del codice *C* definitivo che contiene alcune parti di tipo generale, provenienti da un *module skeleton*, e il codice dei codel. Un linguaggio maggiormente espressivo come il *C++* può essere utilizzato *a due livelli* [24] per sviluppare l'architettura, permettendo di costruire un insieme di componenti di libreria molto più versatili ed estendibili, che possono essere utilizzati durante lo sviluppo dell'applicazione concreta.

La ricerca di un possibile riuso del software che la programmazione ad oggetti intende facilitare può essere realizzata mediante numerose tecnologie che sono state sviluppate nel corso degli anni, ma la robotica si è mostrata molto sensibile in particolare all'utilizzo di *Framework* come strumento per costruire sistemi aperti ed estendibili: OROCOS ne è un esempio, ed ulteriori riferimenti si possono trovare in [18] e [25].

Delle possibili definizioni di *framework* presenti in letteratura, vengono citate le seguenti (provenienti da [26]):

> *A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact.*

> *A framework is the skeleton of an application that can be customized by an application developer.*

Sfruttando le funzionalità tipiche dei linguaggi object-oriented, come l'ereditarietà e la programmazione generica, è possibile realizzare una libreria che contiene tutte le parti immutabili dell'architettura (*frozen spots*) ma che lascia la possibilità all'utente di realizzare codice specifico da inserire in alcuni punti, lasciati aperti dalla struttura (*hot spots*). È quindi possibile pensare al framework come ad uno scheletro sul quale si può realizzare con maggiore facilità un'applicazione specifica, sfruttando i pattern di design [27] che mette a disposizione. Per molti versi un framework è simile come potenza espressiva ad un *Application generator* [28], che costruisce un'applicazione concreta a partire da un linguaggio di descrizione di alto livello.

Capitolo 3

Realizzazione dell'architettura

3.1 Obiettivi

La libreria *object-oriented* sviluppata durante questo lavoro di tesi è stata creata con lo scopo di rendere disponibile una serie di moduli che, basandosi sul framework *OROCOS*, forniscano al programmatore gli strumenti necessari per realizzare un'applicazione di robotica mobile ad un livello di astrazione sufficientemente elevato. Nel contempo, la libreria consente di nascondere i dettagli legati allo scheduling dei task ed alla realizzazione delle comunicazioni tra di essi, fornendo tuttavia i mezzi per garantirne ove necessario il controllo. L'uso della nuova architettura per realizzare applicazioni specifiche per il *Nomad* (come viene descritto nel capitolo successivo) è una prima applicazione.

Focalizzando l'attenzione sulla robotica mobile sono state analizzare alcune architetture di pubblico dominio (descritte nel capitolo 2) con lo scopo di individuare le problematiche e le necessità di cui il progetto dovrà occuparsi. L'analisi dei *pattern di comunicazione* realizzati dai membri del progetto *OROCOS*, si è rivelata particolarmente utile per identificare le forme di comunicazione con le quali i moduli possono interagire fra di loro; una breve descrizione delle primitive proposte in [14] e poi utilizzate durante questo lavoro di tesi è riportata nella tabella 3.1.

I pattern di comunicazione permettono il disaccoppiamento dei componenti poi-

Pattern	**Descrizione**
Send	Trasferisce dati dal client al server senza la ricezione di una conferma dal server. Rappresenta una comunicazione *monodirezionale* utile per inviare comandi e settare configurazioni.
Autoupdate	Uno o più client si registrano presso un server richiedendo un'informazione che viene inviata non appena nuovi dati sono disponibili. Rappresenta una comunicazione di tipo *push*. È possibile ridurre il traffico, se il client richiede l'informazione ad una frequenza minore rispetto a quella con cui viene prodotta, mediante una richiesta che prevede l'invio dei dati soltanto ad ogni ennesimo aggiornamento (*autoupdate timed*).
Wiring	Permette la configurazione dinamica della struttura dati fra componenti; il modulo client, parte del protocollo di comunicazione, può essere utilizzato come una porta per connettersi al server corretto. Wiring è utile per cambiare la struttura dati fra componenti in modo dinamico, per comporre diversi comportamenti di un insieme di funzionalità.

Tabella 3.1: Pattern di comunicazione identificati in *OROCOS*.

chè entrambe le parti del meccanismo di comunicazione interagiscono in modo asincrono indipendentemente dalla modalità di accesso utilizzata dall'utente; gli oggetti communication garantiscono la diversità e la genericità anche con un insieme molto piccolo dei pattern di comunicazione.

In figura 3.1 è illustrato il metodo generale di funzionamento: tutti i componenti interagiscono unicamente tramite i servizi forniti dai pattern di comunicazione; questi non solo permettono il disaccoppiamento dei componenti ma anche l'accesso simultaneo ad un componente.

L'utilizzo dei pattern di comunicazione, come nucleo centrale di un'approccio a componenti, consente di ottenere le interfacce di quest'ultimi chiaramente strutturate e evita comportamenti insoliti di queste interfacce senza limitare la struttura interna dei componenti.

L'evoluzione progressiva dei sistemi robotici ha segnato l'affermazione di *ar-*

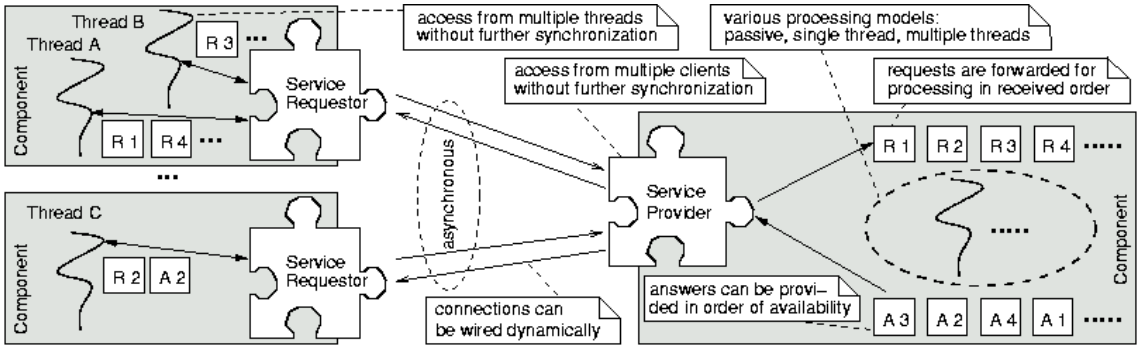

Figura 3.1: Interazione di componenti.

chitetture ibride, che consentono la cooperazione tra processi reattivi, dal ridotto carico computazionale ma dall'elevata frequenza di attivazione, con processi deliberativi, che richiedono tempi di elaborazione lunghi e difficilmente quantificabili. Il design dei singoli livelli di un architettura robotica pone in evidenza l'importanza del progetto a struttura modulare, ovvero formato da un insieme di moduli collegati tra di loro che ne formano l'ossatura; così strutturato il progetto favorisce la costruzione progressiva delle applicazioni ed il parziale riuso dei componenti.

Alla luce di queste considerazioni, dopo l'analisi dello stato dell'arte, si è convenuto di realizzare l'architettura in modo tale da consentire lo sviluppo di applicazioni per la robotica mobile secondo uno schema *behaviour-based*, limitando l'uso dell'architettura al solo livello reattivo e configurando la sua struttura in modo da favorirne l'integrazione con i livelli superiori. In particolare, l'utilizzo delle interfacce, per lo scambio di dati e comandi, dovrà garantirne la trasparenza rispetto al meccanismo che le implementa e assicurare inoltre la possibilità di estendere il supporto verso la comunicazione con dispositivi remoti e la teleoperazione.

Lo sviluppo del software nelle applicazioni robotiche che qui si vuole progettare può avvenire a diversi livelli di astrazione. In primo luogo è necessario costruire i componenti utili al funzionamento del framework OROCOS. Successivamente, i singoli componenti, che possono rappresentare behaviour di uso generale o possono essere legati ad un particolare hardware robotico, devono essere scelti e riuniti per ottenere l'applicazione desiderata.

3.2 Programmazione di un componente in *OROCOS*

3.2.1 Oggetti Communication

Gli oggetti communication caratterizzano i pattern di comunicazione e racchiudono al loro interno, i dati che devono essere introdotti nel canale di comunicazione; sono formati da due parti:

- la descrizione dei dati da trasmettere effettuata tramite IDL.
- l'oggetto communication vero e proprio che fornisce i metodi di accesso ai dati degli utenti.

I metodi d'accesso, come mostrato in figura 3.2, non sono inclusi nella descrizione IDL al fine di consentire l'utilizzo di tipi generali quali i contenitori STL nei metodi d'accesso ai dati.

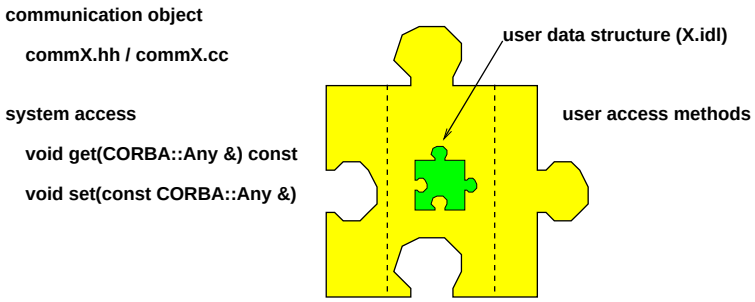

Figura 3.2: Struttura di un oggetto communication che caratterizza i pattern di comunicazione.

La figura 3.2 mostra la struttura di un oggetto communication. Quest'ultimo contiene i dati utente descritti tramite IDL. Solo la struttura dati, in modo non visibile agli utenti attraverso i pattern di comunicazione, è trasmessa e poi utilizzata per attivare un nuovo oggetto communication nella parte di ricezione del pattern di comunicazione.

I metodi: *void get(CORBA::Any)* e *void set(const CORBA::Any)* sono necessari per la comunicazione degli oggetti Communication, ma non sono direttamente utilizzati dagli utenti; altri metodi forniscono l'interfaccia utente che consente di accedere al contenuto degli oggetti communication.

3.2.2 Descrizione dettagliata di un Oggetto Communication

Vediamo ora un oggetto communication nella sua interezza. Esso, come si può dedurre dalla tabella 3.2, è formato da tre file; il loro nome ha una struttura ben precisa.

Nome file	**Esempio**	**Contenuto**
X.idl	timeOfDay.idl	Al suo interno vi è la descrizione IDL della struttura dati utilizzata dall'oggetto communication.
commX.hh	commTimeOfDay.hh	È il file header dell'oggetto communication e ne contiene la sua documentazione.
commX.cc	commTimeOfDay.cc	Contiene l'implementazione dell'oggetto communication.

Tabella 3.2: File che compongono l'oggetto communication.

Per realizzare un oggetto communication, si devono implementare i tre file indicati nella tabella 3.2; innanzitutto si deve descrivere la struttura dati utilizzando IDL.

Nell'esempio riportato nel listato 3.1 vi è una semplice struttura dati di un oggetto communication con lo scopo di trasmettere i valori del tempo.

```
//
// timeOfDay.idl
//
struct TimeOfDay
{
  short hour;
  short minute;
  short second;
};
```

Listato 3.1: descrizione del file timeOfDay.idl

Il listato numero 3.2 mostra il file header dell'oggetto communication, per trasmettere i valori del tempo contenuti nella struttura dati IDL appena descritta.

```
// commTimeOfDay.hh
class CommTimeOfDay
  {
  protected:
    TimeOfDay time;
    // questa e' la struttura dati descritta tramite IDL; si possono
    // aggiungere altre strutture dati ma solo se descritte con IDL.
  public:
    // costruttori , distruttori , costruttori di copia ...
    CommTimeOfDay();
    virtual ~CommTimeOfDay();

    // i seguenti metodi devono essere disponibili nell'oggetto
    // communication, sono sempre gli stessi dato che devono compiere
    // azioni di get e set sulla struttura dati IDL; questi metodi
    // sono utilizzati dai pattern di comunicazione
    // e non dagli utenti .
    void get(CORBA::Any &) const;
    void set(const CORBA::Any &);

    // i seguenti metodi costituiscono l'interfaccia utente , sono
    // forniti da chi realizza l'oggetto communication e forniscono
    // gli strumenti necessari per lavorare su quest'ultimo.
    void get(int &, int &, int &);
    void set(int , int , int );
    void print(ostream &os = cout) const;
    };
```

Listato 3.2: Descrizione del file commTimeOfDay.hh.

Infine, nel listato numero 3.3, osserviamo la realizzazione dei metodi che costituiscono l'ossatura dell'oggetto communication.

Come si può notare gli ultimi tre metodi sono quelli utilizzati dagli utenti e consentono a quest'ultimi di ricevere l'ora esatta in ore, minuti e secondi, di impostarla e di stamparla a video.

Quanto sopra descritto è un tipico oggetto communication che può essere uti-

lizzato all'interno dei pattern di comunicazione e scambiato tra moduli. Tutti gli oggetti communication permettono la derivazione di ulteriori classi; questo meccanismo consente di aggiungere facilmente nuovi metodi d'accesso agli oggetti già esistenti.

```
// commTimeOfDay.cc
//   intestazioni
CommTimeOfDay::CommTimeOfDay() {}
CommTimeOfDay::~CommTimeOfDay() {}
void CommTimeOfDay::get(CORBA::Any &a) const
{
  a «= time;
}
void CommTimeOfDay::set(const CORBA::Any &a)
{
  TimeOfDay *t;

  a »= t;

  time = *t;
}
void CommTimeOfDay::get(int &h, int &m, int &s)
{
  h = time.hour;
  m = time.minute
  s = time.second;
}
void CommTimeOfDay::set(int h, int m, int s)
{
  time.hour = h;
  time.minute = m;
  time.second = s;
}
void CommTimeOfDay::print(ostream&os) const
{
  // stampa a video l'ora esatta ....
}
```

Listato 3.3: Descrizione del file commTimeOfDay.cc.

3.3 Moduli realizzati

Il *Modulo* è considerato un componente attivo ed autonomo che opera all'interno dell'architettura. Il sistema in esecuzione è composto da un insieme di moduli che svolgono attività indipendenti e cooperano tra loro scambiando informazioni e comandi.

I moduli concreti svolgono compiti elementari che sarà possibile riunire per realizzare l'applicazione complessiva: un singolo modulo può rappresentare la versione concreta di un *behaviour*, oppure può rendere disponibile all'interno del sistema alcune interfacce verso l'hardware sensomotorio del robot (vedi figura 3.3).

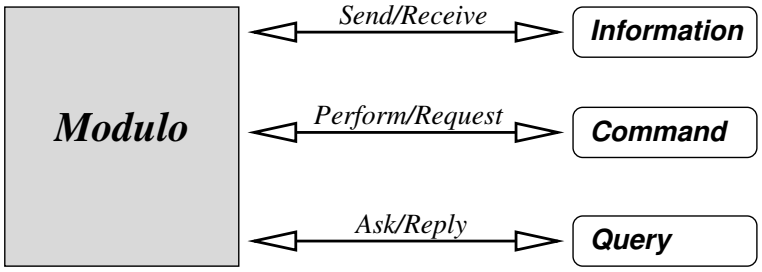

Figura 3.3: Modulo e suoi comportamenti.

3.3.1 Moduli per gli oggetti Communication

OROCOS non disponeva di moduli specifici per il *Nomad 200*, ma di moduli generali utilizzabili su più tipi di robot; obiettivo di questo lavoro di tesi è quello di estendere l'architettura fornita da OROCOS in modo specifico per il *Nomad 200*. Innanzitutto era necessario realizzare gli oggetti communication per ricevere le informazioni dai sensori, in particolare sono stati realizzati due oggetti communication: uno per ricevere i dati dai sonar e dagl'infrarossi, l'altro per ricevere le informazioni dai sensori di contatto (*bumper*). Come indicato nel paragrafo 3.2, il punto di partenza consiste nel descrivere la struttura dati utilizzata nella comunicazione tramite l'utilizzo di IDL; nel listato 3.4 è descritta la struttura dati necessaria per ricevere le letture sensoriali dei sonar e degl'infrarossi.

La struttura è composta da quattro variabili:

```
//
// smartDistance.idl
//
struct DistanceSensorData
{
  unsigned short sensor_id;
  TimeStamp time_stamp;
  boolean is_valid;
  unsigned short distance;
};
```

Listato 3.4: Descrizione del file smartDistance.idl.

1. *sensor_id*: memorizza il numero identificativo del sensore che ha effettuato la lettura, infatti sul Nomad 200 vi sono sedici sonar e sedici infrarossi.
2. *time_stamp*: memorizza l'istante in cui è avvenuta la lettura.
3. *is_valid*: indica se la lettura appena effettuata è valida oppure no.
4. *distance*: memorizza la distanza misurata.

Nel listato 3.5 è descritta la struttura dati necessaria per ricevere le informazioni riguardanti i bumper.

```
//
// smartContact.idl
//
struct ContactSensorData
{
  unsigned short sensor_id;
  TimeStamp time_stamp;
  long value;
};
```

Listato 3.5: Descrizione del file smartDistance.idl.

La struttura è composta da tre variabili:

1. *sensor_id*: memorizza il numero identificativo del sensore bumper che ha effettuato la lettura, infatti sul Nomad 200 vi sono venti micro interruttori che individuano la presenza di un oggetto che si trovi a contatto del robot.

2. *time_stamp*: memorizza l'istante in cui è avvenuta la lettura.

3. *distance*: memorizza la misura.

Se ne deduce che la struttura dati utilizzata per gli oggetti communication è estremamente semplice, ma allo stesso tempo efficace al fine di ricevere tutte le informazioni necessarie per il controllo del robot.

Per completare gli oggetti communication occorre realizzare i file commDistance.hh/coomDistance.cc e commContact.hh/commContact.cc; per semplicità illustreremo solo i file header, dato che gli altri non fanno altro che implementare i metodi relativi agli utenti o ad uso interno, che non sono sviluppati nei primi.

Nel listato 3.6 è ripotata la classe CommDistanceData necessaria per comunicare i dati relativi a sonar ed infrarossi.

```
1  // commDistance.hh
2  class CommDistanceData
3    {
4    protected:
5      SmartIDL::DistanceSystemData _data;
6
7    public:
8      // costruttori , distruttori , costruttori di copia e metodi ad uso interno
9      CommDistanceData();
10     CommDistanceData(const SmartIDL::DistanceSystemData &data);
11     virtual ~CommDistanceData();
12     void get(CORBA::Any &a) const;
13     void set(const CORBA::Any &a);
14     static inline std:: string identifier () { return "Smart::CommDistanceData";};
15     inline const SmartIDL::DistanceSystemData &get_idl() const { return _data; }
16
17     // metodi get utilizzabili dall' esterno
18     inline unsigned int get_num_measurements() const {
19     return _data. distance_data . length ();
20     }
21     inline unsigned int get_sensor_id(unsigned int i) const {
22     return _data. distance_data [i ]. sensor_id ;
```

```
23      }
24      inline CommTimeStamp get_measurement_time_stamp(unsigned int i) const {
25      return CommTimeStamp(_data.distance_data[i].time_stamp);}
26      inline bool is_measurement_valid(unsigned int i ) const {
27      return _data . distance_data [ i ]. is_valid ;}
28      inline double get_measured_distance(unsigned int i ,
29      const double unit = 0.001) const
30      { return _data . distance_data [ i ]. distance * 0.001 / unit ;}
31
32      // metodi set utilizzabili dall ' esterno
33      inline void set_num_measurements(unsigned int n) {
34      _data . distance_data . length (n);
35      for(unsigned int i =0; i<n; ++i ) {
36          set_sensor_id (i , i );
37          set_measurement_valid(i , false );
38      }
39      }
40      inline void set_sensor_id (unsigned int i , unsigned int id ) {
41      _data . distance_data [ i ]. sensor_id = id;
42      }
43      inline void set_measurement_time_stamp(unsigned int i ,
44      const CommTimeStamp &ts) {
45          _data . distance_data [ i ]. time_stamp = ts . get_idl ();
46      }
47      inline void set_measurement_valid(unsigned int i , bool is_valid ) {
48          _data . distance_data [ i ]. is_valid = is_valid ;
49      }
50     inline void set_measured_distance (unsigned int i , double dist ,
51     const double unit = 0.001) {
52        const unsigned int d = int (:: rint ( dist * unit * 1000));
53        if (d <= 65535) {
54          _data . distance_data [ i ]. distance = d;
55          set_measurement_valid(i , true );
56        }
57        else {
58          _data . distance_data [ i ]. distance = 65535;
59          set_measurement_valid(i , false );
60      }
61      }
62     void print ( std :: ostream &os = std :: cout ) const;
63   }
```

Listato 3.6: Descrizione della classe CommDistanceData.hh.

Analizzando il listato 3.6 si nota che il costruttore, alla riga 10, riceve come argomento un oggetto di tipo SmartIDL::DistanceSystemData; quest'ultimo non è altro che una sequenza di strutture DistanceSensorData, come definita nel file smartDistance.idl e indicata nel listato 3.4. Commentiamo ora nel dettaglio i metodi che formano l'interfaccia dell'oggetto communication:

- *get_num_measuraments*: restituisce il numero di misure memorizzate all'interno della struttura dati (riga diciotto).
- *get_sensor_id*: restituisce per ogni sensore il suo numero identificativo (riga ventuno).
- *get_measurement_time_stamp*: restituisce per ogni sensore l'istante in cui è stata effettuata la misura (riga ventiquattro).
- *is_measurement_valid*: indica se la misura appena effettuata è valida oppure no (riga ventisei).
- *get_measured_distance*: tra i metodi indicati è quello più importante e restituisce per ogni sensore la distanza misurata; in particolare si può notare come il secondo parametro passato permetta di avere la misurazione nell'unità di misura voluta, dato che in genere le misure restituite sono indicate in decimi di pollice (riga ventotto).
- *metodi set*: questi metodi set hanno gli stessi nome dei metodi get e fanno la cosa inversa, cioè settano il numero identificativo dei sensori, il tempo di misura, il numero di misure e così via.
- *print*: questo metodo stampa a video una serie d'informazioni ricavate dalla struttura dati.

Il file commContact.hh è molto simile a quello visto in figura 3.6 ed è basilare per ricevere le informazioni dai bumper; in particolare si differenzia dal commDistance.hh nel costruttore: *CommContactData(const SmartIDL::ContactSensorData &data)* riceve come argomento la struttura dati *SmartIDL::ContactSensorData*, come definita nel file smartContact.idl. Inoltre tutti i metodi get e set, visti in precedenza, sono disponibili anche all'interno di questo file; manca il metodo *is_measurement_valid*

e il suo corrispettivo set dato che la variabile *is_valid* non è presente nella struttura dati ContactSensorData. Infine non è presente nemmeno il metodo *get_measured_distance* dato che restituisce la distanza misurata, al suo posto vi è il metodo *get_measured_contact* che, insieme al corrispondente set, agisce sulla variabile *value* indicando se il particolare sensore è a contatto con qualche oggetto oppure no, come illustrato nel listato 3.7.

Come indicato nel paragrafo 3.3 è molto semplice costruire una struttura dati da trasmettere all'interno dei pattern di comunicazione; anche i metodi che formano l'interfaccia utente sono semplici e non fanno altro che accedere ed operare sulla struttura dati precedentemente definita.

```
//
// commContact.hh
//
// il valore di value indica se il bumper e'
// a contatto con qualche ostacolo
//
inline long get_measured_contact () const
{
  return _data.value;
}
```

Listato 3.7: Descrizione del metodo get_measured_contact.

È possibile aggiungere altri metodi per arricchire l'interfaccia dell'oggetto communication; nel paragrafi successivi analizzeremo nel dettaglio i moduli realizzati per controllare tutte le funzionalità del robot Nomad 200; in particolare vedremo i server che permettono di ricevere dal robot tutte le informazioni sensomotorie e di inviare una serie di comandi utili al suo funzionamento. Questi server ovviamente sfruttano gli oggetti communication descritti in questo paragrafo. Vedremo anche alcuni client realizzati al fine di testare le funzionalità dei moduli appena costruiti.

3.3.2 Modulo RobotConnection

Questo modulo serve per connettersi al robot fisico ed è costituito da un costruttore e da un distruttore. Come evidenziato nel listato 3.8, il costruttore si occupa di aprire la connessione col robot e di alcune inizializzazioni. In particolare viene fissato l'ordine e il tempo di *sparo* dei sonar e degli infrarossi grazie alle funzioni *conf_ir* e *conf_sn*. Nell'attuale situazione iniziale l'ordine di *sparo* è estremamente semplice: si parte dal sensore numero uno per arrivare al numero sedici per esaurire i sedici sensori che sono presenti sul robot. Per particolari comportamenti è comunque possibile modificare questo ordine per entrambi i tipi di sensori (sonar e infrarossi). Il distruttore viene utilizzato, data la sua natura, per disattivare il robot tramite l'utilizzo delle funzioni *st* e *ws* e per scollegarsi successivamente dal robot stesso.

3.3.3 Server per la gestione del robot

In questo paragrafo vengono analizzati nel dettaglio i server ed i moduli realizzati al fine di ottenere il controllo del robot nella sua globalità. Innanzitutto si deve ricordare che *robotd*, il software di controllo del Nomad 200, fornito dalla casa costruttrice del robot, ha la possibilità di impostare una maschera idonea a ricevere dai sensori e dai motori le sole informazioni che interessano. Una volta ottenute, dette informazioni verranno poi memorizzate all'interno di un vettore *state* al quale è possibile accedere per poi introdurle nei vari oggetti communication.

Il server, illustrato nel listato numero 3.9 serve per avere tutte le letture sensoriali dagli infrarossi. La classe *DistanceIrPushTimedHandler* viene derivata dalla quella fornita in OROCOS *PushTimedHandler*. Questo pattern consente di avere nuovi aggiornamenti ad ogni intervallo prefissato di tempo operando su un'oggetto CommDistanceData indicato nel listato 3.6. Il costruttore fissa a sedici il numero delle misurazioni da effettuare, ne consegue che si ottiene una misurazione per ogni sensore. Il metodo *handlePushTimer* legge i valori degli infrarossi sulla scheda del Nomad 200 atta a gestire i sensori e li memorizza nell'oggetto CommDistanceData; in particolare vengono memorizzati i tempi delle misurazioni, i relativi valori vengono convertiti in centimetri. Infine l'oggetto communication viene inviato ai client

```
//
// connect.hh
//

RobotConnection::RobotConnection(int id)
: _id(id)
{
   connect_robot(id);
   for(int i=0; i<16; i++) {
      ir_ord[i] = i;
      so_ord[i] = i;
   }
   conf_ir(dep, ir_ord);
   conf_sn(fire, so_ord);
}

RobotConnection::~RobotConnection()
{
   std::cerr « "RobotConnection::~RobotConnection():stop" « std::endl;
   st();  // comando di stop prima della disconnessione
   ws(TRUE, TRUE, TRUE, 5);
   // attesa di cinque secondi per lo stop di ogni asse
   disconnect_robot(_id);
}
```

Listato 3.8: Descrizione del modulo RobotConnection.

sottoscritti per la ricezione degli aggiornamenti.

Il modulo *smartNomad200SonarServer* per la gestione dei valori dei sonar è molto simile a quello appena trattato; anche qui viene impostato a sedici il numero delle misurazioni da effettuare, dato che i sonar sono in numero di sedici. Gli aggiornamenti giungono ad ogni intervallo prefissato di tempo e i valori letti dalla scheda, una volta convertiti in centimetri, vengono inseriti all'interno dell'oggetto *distdata_son* del tipo *CommDistanceData*. L'ultima operazione svolta da questo modulo è costituita dall'invio dell'oggetto communication, che contiene le informazioni utili, ai vari client sottoscritti.

```
//
// smartNomad200InfraredServer.hh
//

class DistanceIrPushTimedHandler:
public CHS::PushTimedHandler<Smart::CommDistanceData>
{

public:
  DistanceIrPushTimedHandler ();
  virtual ~DistanceIrPushTimedHandler();

  void handlePushTimer
  (CHS::PushTimedServer<Smart::CommDistanceData> &server) throw();

private:
  Smart::CommDistanceData distdata_ir;
  Smart::CommTimeStamp time_stamp;

};
```

Listato 3.9: Descrizione della classe DistanceIrPushTimedHandler.

Il server riportato nel listato 3.10 è preposto a ricevere le informazioni dai sensori di contatto. La classe *ContactPushTimedHandler* è derivata, come quelle viste in precedenza, da quella di OROCOS *PushTimedHandler* al fine di ottenere gli aggiornamenti sulle letture ad intervalli di tempo prefissati e lavora su un oggetto CommContactData. Il metodo *handlePushTimer* imposta la *maschera* di *robotd* in modo tale da ricevere le sole informazioni riguardanti i bumper. Quest'ultime, una volta ricavate dal vettore state di *Robotd*, vengono inserite nell'oggetto *contactdata* di tipo *CommContactData* e successivamente inviate ai client sottoscritti.

Il server illustrato nel listato 3.11 viene utilizzato per ottenere le informazioni odometriche; anche in questo caso, come nei moduli precedentemente trattati, la classe *BaseStatePushTimedHandler* è derivata da quella di OROCOS *PushTimedHandler* per ottenere gli aggiornamenti ad intervalli di tempo prefissati.

```
//
// smartNomad200ContactServer.hh
//

class ContactPushTimedHandler:
public CHS::PushTimedHandler<Smart::CommContactData>
{

public:
   virtual ~ContactPushTimedHandler();
   void handlePushTimer
   (CHS::PushTimedServer<Smart::CommContactData> &server) throw();

private:
   Smart::CommContactData contactdata;
   Smart::CommTimeStamp time_stamp;
};
```

Listato 3.10: Descrizione della classe ContactPushTimedHandler.

Il metodo *handlePushTimer* imposta la *maschera* di *robotd* in modo tale da ricevere le sole informazioni riguardanti l'odometria, in particolare:

- le coordinate x e y,
- la posizione delle ruote e della torretta,
- la velocità lineare e quella di rotazione sia delle ruote che della torretta.

Questi valori vengono poi inseriti negli oggetti *base_position* e *base_velocity* che conterranno rispettivamente la posizione del robot, la velocità lineare e quella di rotazione sia della torretta che delle ruote. A loro volta quest'ultimi verranno poi inglobati nell'oggetto *base_state* per il successivo invio ai client sottoscritti.

Il server riportato nel listato 3.12 viene utilizzato per impostare sia la velocità lineare del robot che quella di rotazione delle ruote e della torretta. La classe *NavigationVelocitySendHandler* è derivata da quella fornita da OROCOS di nome *SendServerHandler*; quest'ultimo pattern rappresenta una comunicazione *monodirezionale* utile per inviare comandi e settare configurazioni. La classe derivata opera

```
//
// smartNomad200PosServer.hh
//

class BaseStatePushTimedHandler:
public CHS::PushTimedHandler<Smart::CommBaseState>
{
public:
   virtual ~BaseStatePushTimedHandler();

   void handlePushTimer
   (CHS::PushTimedServer<Smart::CommBaseState> &server) throw();

private:
   Smart::CommTimeStamp time_stamp;
   Smart::CommBasePosition base_position;
   Smart::CommBaseVelocity base_velocity;
   Smart::CommBaseState base_state;
};
```

Listato 3.11: Descrizione della classe BaseStatePushTimedHandler.

su un'oggetto communication: *CommNavigationVelocity* definito da OROCOS, utile per immagazzinare i valori delle velocità che si vanno ad impostare. Il metodo *NavigationVelocitySendHandler*, una volta estrapolati i dati delle velocità da impostare dall'oggetto *cmd* di tipo *CommNavigationVelocity*, li elabora in modo che siano utilizzabili nelle unità di misura del Nomad 200; infine attraverso la funzione *mv*, fornita da *robotd*, imposta questi valori sul robot. In particolare, la velocità di rotazione della torretta sarà sempre uguale a quella di rotazione delle ruote al fine di ottenere un punto di riferimento durante la navigazione del robot.

L'ultimo modulo trattato in questo paragrafo è *smartNomad200Server*, illustrato nel listato 3.13 e di fondamentale importanza per il controllo del robot con la nuova architettura.

All'inizio del modulo vengono impostati i tempi di aggiornamento dei vari server visti nei listati precedenti, in particolare si riscontra che per l'aggiornamento dei sensori infrarossi occorrono quaranta millisecondi, sessantaquattro per i sonar,

```
//
// smartNomad200VelServer.hh
//

class NavigationVelocitySendHandler:
public CHS::SendServerHandler<Smart::CommNavigationVelocity>
{

public:

 NavigationVelocitySendHandler ();

 void handleSend(const Smart::CommNavigationVelocity &cmd) throw();

};
```

Listato 3.12: Descrizione della classe NavigationVelocitySendHandler.

trenta per i bumper e cento per l'odometria. Come visto in precedenza, tutti questi server derivano dal pattern fornito da OROCOS *PushTimedHandler* il quale fornisce gli aggiornamenti ad intervalli prefissati di tempo, quelli appena visti sono proprio gli intervalli di aggiornamento.

All'interno del *main* la prima operazione da effettuare è la dichiarazione del *component*, che in questo modulo chiameremo *smartNomad200Server*. Il component è tecnicamente realizzato come un processo; può contenere *thread* discrete e può interagire con altri component attraverso i predefiniti pattern di comunicazione. Infatti questo component collaborerà con quelli dei vari client per ricevere e inviare varie informazioni.

Subito dopo la dichiarazione del component ci si connette al robot, tramite *robotConnection* (trattato nel paragrafo 3.3.2). Successivamente vengono dichiarati i vari server: quello per gli infrarossi, quello per i sonar, quello per i bumper, quello per le velocità da impostare e quello per l'odometria; a ognuno di questi moduli viene associato il component *smartNomad200Server* in modo tale da poter ricevere da ognuno di essi gli aggiornamenti e da poter inviare comandi di velocità. Inoltre ad ogni modulo derivato dal pattern *PushTimedHandler* viene impostato il tempo di

```
// smartNomad200Server.cc

CHS::SmartComponent component("smartNomad200Server",argc,argv);
readParameters(argc,argv);
// Connect to robot
RobotConnection robotConnection(param_robotId);
// push current values to subscribed clients

DistanceIrPushTimedHandler distanceirPushTimedHandler;
CHS::ThreadQueuePushTimedHandler<Smart::CommDistanceData>
threadDistanceIrPushTimedHandler(distanceirPushTimedHandler);
CHS::PushTimedServer<Smart::CommDistanceData>
distanceIrPushTimedServer (&component, "distanceir",
threadDistanceIrPushTimedHandler,0.001* par_baseIrPushTimedInterval );
//idem per gli altri moduli push ...
// comandi di velocita' al robot
NavigationVelocitySendHandler navigationVelocitySendHandler;
CHS::ThreadQueueSendHandler<Smart::CommNavigationVelocity>
threadNavigationVelocitySendHandler( navigationVelocitySendHandler );
CHS::SendServer<Smart::CommNavigationVelocity>
navigationVelocitySendServer (&component,
"navigationvelocity", threadNavigationVelocitySendHandler );

// Start push services
distanceIrPushTimedServer.start();
//idem per gli altri push
component.run();
```

Listato 3.13: Parti primarie del server centrale.

aggiornamento, in precedenza definito all'inizio del file. Vengono poi successivamente attivati i servizi di *push*. Infine con *component.run()* inizia la comunicazione tramite i pattern di OROCOS.

3.3.4 Client per il testing

In questo paragrafo analizzeremo alcuni client realizzati allo scopo di testare i vari moduli costruiti per ampliare l'architettura sviluppata da OROCOS; potremo così comprendere come i vari client si sottoscrivono presso il server centrale per ricevere informazioni sensoriali o specificare particolari comandi di velocità da impostare sul Nomad.

Il client riportato nel listato 3.15 si collega al component *smartNomad200Server*, descritto nel listato 3.13, per ottenere gli aggiornamenti dal server degli infrarossi. Vediamo nel dettaglio come un client si può sottoscrivere presso il server centrale e scambiare con esso informazioni e comandi. Il costruttore della classe *InfraredClientThread* riceve come argomento un oggetto di tipo *PushTimedClient*, così come definito da OROCOS per gestire i client che si sottoscrivono al server al fine di ottenere aggiornamenti periodici. Il costruttore va poi ad impostare l'oggetto *_proxy* dichiarato come oggetto del tipo *PushTimedClient*.

```
// smartNomad200InfraredClient.cc

int InfraredClientThread :: svc(void)
{
  Smart :: CommDistanceData dis;
  CHS::StatusCode status ;
  CHS::StatusCodeConversion(_proxy.subscribe (10));
  while(true ) {
     status = _proxy.getUpdateWait(dis );
     // controlli ...
     std :: cout « dis « std :: endl ;
  }
  CHS::StatusCodeConversion(_proxy.unsubscribe ());
  return 0;
}
```

Listato 3.14: Descrizione del metodo svc.

All'interno del metodo *svc* illustrato nel listato numero 3.14, viene dichiarato un

oggetto del tipo *CommDistanceData*. Il metodo citato si sottoscrive poi al server per ricevere un aggiornamento ogni dieci e poichè il tempo di lettura degli infrarossi è di quaranta millisecondi ne consegue che il modulo riceverà le letture ogni quattrocento millisecondi. Successivamente i dati ricevuti vengono stampati a video; quando il client viene terminato cessa anche la sottoscrizione per gli aggiornamenti.

Il component di nome *smartNomad200InfraredClient*, dichiarato all'interno del main (listato 3.15), viene inviato come argomento assieme al component *smartNomad200Server* ad un'oggetto di tipo *PushTimedClient* che a sua volta verrà inviato al costruttore dell'oggetto *InfraredClientThread* precedentemente analizzato. Si pone in esecuzione il component *smartNomad200InfraredClient* in modo tale da potersi collegare con il server centrale e ricevere tutti gli aggiornamenti disponibili; infine il sistema di gestione delle thread si mette in attesa degli aggiornamenti provenienti dal server.

```
// smartNomad200InfraredClient.cc

CHS::SmartThreadManager *threadManager =
CHS::SmartThreadManager::instance();
component =
new CHS::SmartComponent("smartNomad200InfraredClient",argc,argv);
CHS::PushTimedClient<Smart::CommDistanceData>
infraredPushTimedClient(component,"smartNomad200Server","distanceir");
InfraredClientThread infraredClientThread(infraredPushTimedClient);
infraredClientThread.open();
component->run();
threadManager->wait();
```

Listato 3.15: Descrizione delle parti fondamentali del main.

Il secondo client realizzato si occupa di ricevere le letture dei sonar e di stamparle a video ed è molto simile a quello appena analizzato. Esso dispone di un metodo *svc*, identico a quello del client per la ricezione dei valori degli infrarossi, che si sottoscrive al server al fine di ricevere un aggiornamento su dieci; il main è praticamente identico a quello precedentemente indicato; ovviamente cambia il nome del

component che qui si chiamerà *smartNomad200SonarClient*.

Analizziamo in dettaglio il client che si occupa di impostare le velocità sul Nomad 200. Il client si collega al component *smartNomad200Server*, indicato al listato numero 3.13, per fissare le velocità da impostare sul robot. Il costruttore della classe

```
// smartNomad200VelClient.cc

int InfraredClientThread :: svc(void)
{
  Smart::CommNavigationVelocity vel;
  CHS::StatusCode status ;
  double v , omega;
  // richiesta inserimento dati
  while(true)
  {

    std :: cin » v » omega;
    vel . set_v(v , 0.01);
    vel . set_omega(omega * 0.00175);
    std :: cout « "sending" « vel « std :: endl ;
    status = _motor.send( vel );
    // controlli ...
  }
  return 0;
}
```

Listato 3.16: descrizione del metodo svc.

VelClientThread riceve come argomento un oggetto di tipo *SendClient*; quest'ultimo è un pattern definito da OROCOS per gestire i client che si sottoscrivono al server al fine di inviare informazioni o comandi senza la ricezione di conferme. Questi va poi ad impostare l'oggetto *_proxy* dichiarato come oggetto del tipo *SendClient*.

Il metodo *svc* illustrato nel listato numero 3.16 dichiara un oggetto del tipo *CommNavigationVelocity*, successivamente chiede all'utente tramite video di inserire la velocità lineare e quella di rotazione da impostare sul robot. Il metodo infine invia al server, attraverso il comando *send(vel)*, le velocità inserite che verranno impostate da quest'ultimo (server).

Il main utilizza lo stesso principio indicato nei client precedentemente citati. All'interno del main viene dichiarato un component di nome *smartNomad200VelClient* che verrà poi messo in esecuzione per collegarsi con il server che imposta le velocità; naturalmente qui vengono utilizzati oggetti del tipo *SendClient*.

In questo paragrafo sono stati analizzati una serie di client molto semplici, realizzati al fine di testare moduli per la ricezione dei valori sensoriali e per impostare le velocità del robot. Nel capitolo successivo verranno analizzati client che svolgono comportamenti più complessi, realizzati con lo scopo di valutare il funzionamento dell'architettura.

3.3.5 Limiti del software *robotd*

Come descritto nel paragrafo 3.3.3, i server atti a leggere le informazioni sensoriali o ad impostare valori di velocità sul robot sfruttano le funzioni messe a disposizione da *robotd*, software di controllo del Nomad 200.

Questo applicativo software fornito dalla Nomadic, società costruttrice del robot, è ormai sorpassato e evidenzia i limiti della robotica del periodo in cui è stato costruito. Prima dell'aggiornamento hardware, realizzato lo scorso anno, il robot disponeva di una ridotta capacità computazionale a bordo; questo fatto ha indotto i progettisti di *robotd* ad orientarsi verso soluzioni che privilegino l'accesso remoto all'uso locale. Il nuovo robot, al contrario, disponendo di un processore dell'ultima generazione, consente di svolgere algoritmi di notevole complessità senza la necessità di un calcolatore aggiuntivo.

Il software di controllo *robotd* in realtà viene fornito anche in forma di libreria; quest'ultima espone le stesse API del client remoto e può essere linkata al codice aggiuntivo per realizzare un programma che viene eseguito localmente sul robot. Tuttavia i tentativi effettuati per sfruttare questa ulteriore possibilità nel produrre programmi più efficienti e funzionali hanno prodotto risultati poco lusinghieri, principalmente a causa di un'interfaccia poco amichevole della libreria, che spesso differisce dalle specifiche e non implementa tutte le funzioni disponibili in modo remoto.

Un ulteriore limite di *robotd*, che può ridurre sensibilmente la reattività delle applicazioni, è la totale assenza di meccanismi di comunicazione di tipo *push*; le letture dei sensori avvengono infatti tramite una richiesta effettuata dal client al server, introducendo così un tempo di latenza abbastanza significativo e scarsamente misurabile tra l'istante in cui il dato viene letto dall'hardware e quello in cui diventa disponibile per il client. Questo intervallo può essere ridotto unicamente con l'aumento della frequenza con cui il dato viene richiesto; tale aumento introduce overhead sulle comunicazioni e anche seguendo questa strada l'intervallo non può essere completamente annullato.

Il software di controllo *robotd*, che non utilizza nessuna funzionalità real-time del sistema operativo, non garantisce il funzionamento ottimale, con conseguenti prestazioni che decadono progressivamente quando l'unità di elaborazione è sovraccarica. Infine, risulta difficile integrare nuovo hardware sensoriale nell'architettura preesistente, che tra l'altro non fornisce un'interfaccia omogenea neppure per l'hardware originale del robot (ad esempio la telecamera ed il pantilt non possono essere controllati se non localmente).

Per tutti questi motivi si è deciso di non utilizzare più *robotd*. Conseguentemente, partendo dalle funzioni fornite da quest'ultimo, sono state realizzate una serie di moduli che vanno ad interagire direttamente con l'hardware del Nomad 200 e ne permettono il controllo. Ciò, come vedremo nel capitolo relativo la sperimentazione, ha permesso la realizzazione di programmi molto più funzionali ed efficenti.

3.3.6 Eliminazione di *robotd*

Questo paragrafo analizza i due moduli principali realizzati al fine di eliminare robotd. Quest'ultimi dovranno interagire direttamente con le schede presenti sul robot che si occupano dei sensori e dei motori. Per ottenere quanto voluto è stata realizzata la parte di inizializzazione delle due schede, che prima veniva gestita dal software *robotd*, e l'insieme delle funzioni per il controllo dei sensori e dei motori.

Il primo modulo che andiamo ad analizzare nel dettaglio è *sensors.c*, che si

occupa della inizializzazione della scheda *Intellsys 100*; detta scheda, prodotta appositamente dalla Nomadic, controlla contemporaneamente i sensori di contatto, di prossimità e la bussola magnetica presenti sul Nomad 200. Il modulo fornisce anche tutte le funzioni per settare varie impostazioni relative ai sensori precedentemenți citati e per ricevere le letture da essi.

La parte di inizializzazione per l'utilizzo dei sensori avviene tramite l'uso della funzione *sens_init*, illustrata nel listato 3.17. Questa parte è quella più importante ed è peraltro molto semplice. La funzione riceve come argomento l'indirizzo di memoria della scheda dei sensori. Innanzitutto viene deciso l'ordine di *sparo* dei sensori, sia dei sonar che degli infrarossi; per entrambi i tipi di sensori si parte dal sensore numero uno per arrivare al sedici. Si prosegue con l'inizializzazione vera e propria della scheda attraverso la funzione *dpr_init*; questa funzione riceve come argomento l'indirizzo di memoria della scheda dei sensori e va ad impostare una serie di opzioni utili per il funzionamento della scheda stessa. Successivamente vengono impostate le opzioni per l'*inport*, l'*outport* e l'accensione dei led del display. Viene fatto partire il timer, poi si imposta sulla scheda il tempo e l'ordine di *sparo*, precedentemente fissati, sia dei sonar che degli infrarossi tenendo conto delle particolare impostazione dei sensori che si ricavano tramite *read_sonar_setup* e *read_ir_setup*.

Le funzioni più importanti e più utilizzate racchiuse all'interno di questo modulo sono:

- *set_infrared_filter*: questa funzione imposta il tempo di sparo dei sensori infrarossi.
- *set_IR_fireorder*: questa funzione imposta l'ordine di sparo dei sensori infrarossi.
- *set_sonar_fire_rate*: questa funzione imposta i tempo di sparo dei sensori sonar.
- *set_sonar_fireorder*: questa funzione imposta l'ordine di sparo dei sensori sonar.
- *sens_finalize*: questa funzione chiude la memoria della scheda dei sensori una volta terminata la connessione al robot.

```
// sensors.c

int sens_init (unsigned long dpr_address)
{
  int ir_fireorder [16], so_fireorder [16];
  int i, senserror;
  for(i=0; i<16; i++) {
        ir_fireorder [i] = i;
        so_fireorder [i] = i;
  }
  char version [100];
  // inizializzazione scheda
  odpr = (DPR_Block *) dpr_init(dpr_address);
  ioperm(0x378, 1, 1);    // settaggi per
  printer_pin_on (0xff); // l'inport, l'outport
  printer_pin_off (0x40); // e i led
  timer_open();
   set_sonar_fire_rate (1);
   set_sonar_fireorder ( so_fireorder );
  read_sonar_setup ();
    set_infrared_filter (0);
   set_IR_fireorder ( ir_fireorder );
  read_ir_setup ();
  inf_cal_read ();
  return 1;
}
```

Listato 3.17: Inizializzazione dei sensori.

- *get_short_infrared*: questa funzione ricava le letture degli infrarossi.
- *get_sonar*: questa funzione chiude estrapola le letture degli sonar.

Il secondo modulo da analizzare con cura è *motor.c*, che si occupa della inizializzazione della scheda *Galil DMC-630* [4], dedicata al controllo dei motori; all'interno del modulo troviamo una serie di funzioni utili al controllo dei movimenti del Nomad 200.

La parte di inizializzazione, per quanto rigurda i motori, avviene tramite l'utiliz-

zo della funzione *motor_init*, illustrata nel listato 3.18; questa funzione riceve come argomenti l'indirizzo della scheda dei motori e i valori di accelerazione e di velocità che il motore adotterà di default. Si passa successivamente all'inizializzazione della scheda attraverso la funzione *dmc_init*, che riceve come argomento l'indirizzo della scheda dei motori e va ad impostate una serie di valori che permettono lo scambio d'informazioni con la scheda stessa. Una volta terminata la fase d'inizializzazione della scheda si vanno ad impostare i valori di accelerazione e di velocità che il robot adotterà di default.

```
// motor.c

int motor_init ( unsigned long dmc_address, unsigned int x_acceleration ,
unsigned int y_acceleration , unsigned int z_acceleration ,
unsigned int x_velocity , unsigned int y_velocity , unsigned int z_velocity )
{
  int motor_status ;
  // Porta per la scheda
  if (ioperm(dmc_address , 2, 1) == -1) {
    printe ("Error .\n");
    exit (0);
  }
 /* Set up motor controller */
 if ((motor_status = dmc_init(dmc_address )) != 0) {
    printe ("Error .\n");
    if ( motor_status == 0x77)
      printe ("stop .\n");
  } else
    printe ("ok.\n");

  integrat_setup ();
  motor_setup( x_acceleration , y_acceleration , z_acceleration ,
    x_velocity , y_velocity , z_velocity );
  return 1;
}
```

Listato 3.18: Inizializzazione dei motori.

Le funzioni più importanti e più utilizzate racchiuse all'interno di questo modulo

sono:

- *motor_stop*: questa funzione ferma il robot.
- *motor_wait_stop*: questa funzione attende un tempo prefissato per la stop dei tre assi.
- *motor_finalize*: questa funzione chiude la memoria della scheda dei motori una volta terminata la connessione al robot.
- *motor_zero_axis*: questa funzione allinea le ruote con la torretta ed è molto utile.
- *motor_move*: questa funzione imposta i valori di velocità lineare e di rotazione del robot.

Ovviamente i server trattati nel paragrafo 3.3.3 e il modulo *RobotConnection* analizzato nel paragrafo 3.3.2 mutano leggermente in funzione dell'eliminazione di robotd. Infatti non vi sono più le funzionalità fornite da questo sistema software, ma quelle fornite dai moduli appena elencati.

Il modulo che ha subito maggiori modifiche è proprio *RobotConnection*. Di conseguenza è riportato il nuovo costruttore di questa classe nel listato 3.19.

Come si può dedurre dal listato 3.19, all'interno del costruttore vengono utilizzate unicamente le funzioni appartenenti ai moduli *motor.c* e *sensors.c*, presentate in questo paragrafo; ciò avviene ugualmente nel distruttore e in tutti i server che vanno ad operare direttamente sul robot.

Con la realizzazione di questi moduli è stata eliminata la presenza di robotd e tutte le problematiche derivanti dal suo utilizzo.

Il sistema risulta perfettamente funzionante e come si vedrà nel capitolo successivo, riguardante la sperimentazione, garantisce ottimi risultati.

```
// connect.cc

RobotConnection(int id, int model, const std::string &device, int conn)
  : _id(id)
{
    sens_init (0xd0000);
    motor_init (0x3e4   , 200, 300, 300, 100, 200, 200);
    for(int i=0; i<16; i++) {
          ir_ord[i] = i;
          so_ord[i] = i;
    }
      set_infrared_filter   ((unsigned char) dep);
     set_IR_fireorder (ir_ord );

     set_sonar_fire_rate  ((unsigned char) fire );
     set_sonar_fireorder (so_ord);

}
```

Listato 3.19: Costruttore di RobotConnection.

Capitolo 4

Sperimentazione

Nel capitolo 3 è stato illustrato nel dettaglio l'insieme dei moduli che fungono da interfaccia verso l'hardware del robot e gli strumenti che la nuova architettura mette a disposizione. Il passo successivo nella costruzione di un'architettura robotica consiste nella realizzazione di un insieme di behaviour che, sfruttando le funzionalità messe a disposizione dall'architettura stessa, ne controllano il comportamento.

Questo capitolo descrive il progetto di alcuni behaviour per il Nomad 200, che sono stati realizzati con lo scopo di collaudare il funzionamento della nuova architettura e porre le basi per la costruzione di un'architettura *behaviour-based* per il robot. Le parti più significative del codice sorgente sono disponibili nell'appendice A.

4.1 *Wall Following*

Per mostrare come l'architettura appena ultimata consenta di realizzare comportamenti di interesse concreto per il robot, è stata costruita un'applicazione elementare che permette al Nomad di muoversi in un ambiente chiuso mediante la sola navigazione sensoriale, accostandosi alla parete più vicina e mantenendone costante la distanza durante il moto.

4.1.1 Progetto

L'applicazione che si vuole realizzare consiste nel controllo continuo dell'allineamento del robot con la superficie verticale del muro, che per semplicità supporremo localizzato alla sua sinistra. Il sistema, schematizzato in figura 4.1, è retto dal seguente modello matematico:

$$\begin{cases} \dot{\alpha} = \omega \\ \dot{d} = v \sin \alpha \end{cases} \tag{4.1}$$

Dove v e ω sono rispettivamente la velocità lineare ad angolare del robot, α rappresenta l'angolo tra il muro e la direzione istantanea di movimento e d misura la distanza che si vuole controllare. La velocità lineare del robot viene mantenu-

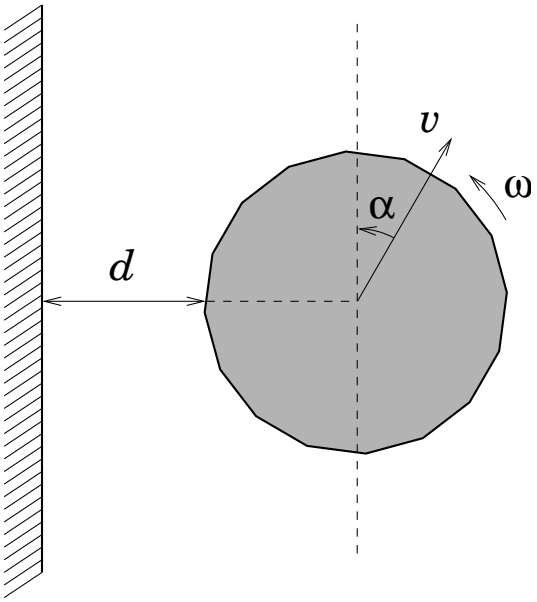

Figura 4.1: Modello geometrico del sistema.

ta costante durante il moto, per cui l'unico *ingresso manipolabile* del sistema è la velocità angolare ω.

Per regolare dinamicamente la distanza del robot dal muro viene utilizzata una tecnica di *controllo in retroazione* realizzata in due fasi. Il primo anello di retroazione (figura 4.2) consente di regolare l'angolo α impostando una velocità di rotazione ω proporzionale alla differenza tra il *set-point* istantaneo α^* e il valore misurato $\hat{\alpha}$.

Postulata la validitá del modello proposto, il *regolatore proporzionale* (avente guadagno pari a K_α) assicura la convergenza verso l'esatto valore del set-point grazie al comportamento integrale del sistema. Da semplici considerazioni geometriche si deduce che dovrà essere: $K_\alpha < 0$.

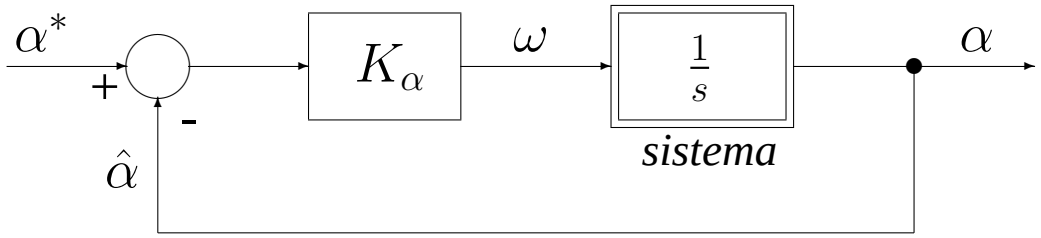

Figura 4.2: Regolatore dell'angolo α.

Il secondo anello di retroazione (figura 4.3) può ora sfruttare il set-point α^* per impostare una correzione dinamica che consenta di regolare la distanza d dal muro in base ad un valore d^* deciso a priori. Come si evince dallo schema 4.3, anche per questo regolatore è stato utilizzato un semplice controllo di tipo proporzionale (in questo caso caso avremo $K_d > 0$), infatti il sistema controllato presenta ancora un comportamento pressochè integrale per valori di α vicini allo zero, ove vale l'approssimazione: $\sin\alpha \simeq \alpha$.

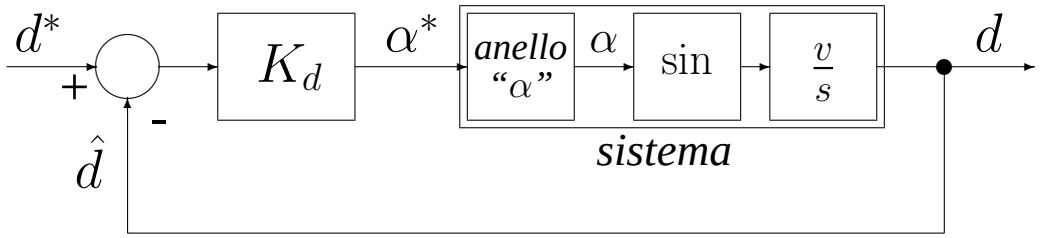

Figura 4.3: Regolatore della distanza d.

Per realizzare concretamente il controllo del Nomad serve infine un metodo che consenta di risalire alla misura istantanea di α e d a partire dai sensori del robot. Per misurare le distanze dagli ostacoli circostanti, il Nomad 200 fornisce due anelli separati costituiti da sedici sensori ciascuno: sonar e infrarossi, come illustrato nel capitolo 1. Questi anelli sensoriali presentano la medesima geometria, pur essendo caratterizzati da portate diverse; per questo motivo le considerazioni seguenti saran-

no valide per entrambe le categorie di sensori.

Siano $\{d_i\}_{i=0..8}$ le misure di distanza rilevate dai sensori posti lungo il lato sinistro del robot. Definiamo m come la posizione del sensore che ha misurato la distanza minore dal muro adiacente, escludendo per ora quello frontale e quello posteriore:

$$m : d_m = \min_{j=1..7}\{d_i\}; \tag{4.2}$$

I valori di α e d sono dati da:

$$\begin{cases} \alpha = \alpha_\Delta + \alpha_\delta \\ d = d_m \end{cases} \tag{4.3}$$

Dove i due contributi all'angolo α derivano dalla geometria dell'anello di sensori (figura 4.4):

- α_Δ dipende dall'indice m del sensore che ha rilevato la minore distanza:

$$\alpha_\Delta = \frac{\pi}{8}(m-4); \tag{4.4}$$

- α_δ viene ricavato dalle misure dei sensori $m-1$ ed $m+1$:

$$\sin\alpha_\delta = \frac{d_{m-1} - d_{m+1}}{l}. \tag{4.5}$$

4.1.2 Realizzazione

Per determinare la posizione reciproca tra il robot e la parete si è deciso di utilizzare i sensori infrarossi, in quanto quest'ultimi forniscono misure più precise quando il Nomad si trova molto vicino agli ostacoli.

Il modulo `smartNomad200Wallclient` si basa sulla classe *InfraredClientThread*, illustrata nel listato 4.1; quest'ultima riceve gli aggiornamenti sui valori degli infrarossi ogni quaranta millisecondi, tramite un'oggetto del tipo *PushTimedClient*. Successivamente la classe va ad impostare i valori di velocità sul ro-

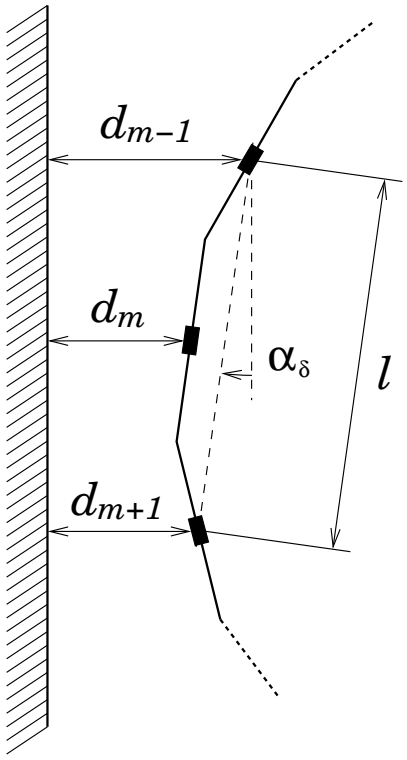

Figura 4.4: Stima di α_δ e d mediante i sensori del robot.

bot attraverso un oggetto di tipo *SendClient*. All'interno di questa classe vengono definiti:

- un oggetto *dis* di tipo *CommDistanceData*, che conterrà materialmente i valori ricavati dagli infrarossi;
- una variabile *desiredDistance* di tipo *double* con valore di 20 cm costituente la distanza desiderata dal muro;
- una variabile *VEL_LIN* con valore di 25 $\mathrm{cm/s}$, costituente la velocitá massima del robot durante questa applicazione;
- una serie di variabili utili al funzionamento del modulo.

Ad ogni iterazione, il codice del modulo provvede a stimare i valori correnti di α e d (sulla base dei dati sensoriali forniti dal modulo *smartNomad200InfraredServer* trattato nel capitolo 3) ed a calcolare il nuovo valore di ω mediante il controllo di

```
// smartNomad200Wallclient.cc

class InfraredClientThread : public CHS::SmartTask
{
public:
  InfraredClientThread (CHS::PushTimedClient<Smart::CommDistanceData>
  &infrared,
  CHS::SendClient<Smart::CommNavigationVelocity> &motor)
  : _infrared ( infrared ), _motor(motor) {}

  int svc ();

private:
  CHS::PushTimedClient<Smart::CommDistanceData> &_infrared;
  CHS::SendClient<Smart::CommNavigationVelocity> &_motor;

  // Distanza dal muro controllata
  Smart::CommDistanceData dis;

  const static double desiredDistance = 20.0; // centimetri
  const static double VEL_LIN = 25.0; // centimetri/s

  // Distanza geometrica tra due sensori in posizione i e i+2
  const static double SENSOR_DISTANCE = 17.0; // Distanza = 17 cm
  double wallDistance ; // centimetri
  double wallAngle; // radianti
  double i_values [16];


  void sensorRefresh ();
  static double P_control ( double K, double setPoint ,
  double measure, double maxVal, double minVal );

};
```

Listato 4.1: Classe InfraredClientThread.

tipo proporzionale. Il modulo provvede inoltre ad inviare al componente *smartNomad200VelServer* i nuovi valori di velocità. Per garantire l'allineamento delle ruote

con la torretta il moto rotatorio sarà sempre lo stesso per entrambi gli assi (torretta e ruote).

Impostando una velocità linerare di 25 $\mathrm{cm/s}$ ed un set-point di distanza pari a 20 cm, il test è proseguito con l'effettuazione di numerose prove che hanno consentito di individuare i valori dei parametri K_α e K_d. Detti valori consentono al robot di procedere senza oscillazioni pur seguendo correttamente l'andamento della parete. Durante quest'ultima fase è emersa l'impossibilità di controllare adeguatamente il robot quando lo spazio frontale diventa molto ridotto, ad esempio nel caso in cui il muro formi un angolo. In situazioni di questo tipo il behviour progettato è insufficiente per consentire al Nomad di muoversi senza investire la parete. Ció è dovuto al fatto che la velocità traslatoria impedisce al robot di seguire raggi di curvatura molto stretti. Per risolvere questo problema si è deciso di aggiungere all'interno del modulo WallFollowing un semplice meccanismo di regolazione della velocità di avanzamento; ad ogni iterazione del controllo, la velocità v viene impostata in modo proporzionale al valore di distanza rilevato dal sensore infrarosso n.1. Quando la misura supera i 35 cm la velocità non sale oltre i 25 $\mathrm{cm/s}$; inoltre viene effettuato un controllo sui primi tre sensori infrarossi. Nel caso in cui un ostacolo si trovi a distanza inferiore ai 20 cm la velocità viene impostata a 7 $\mathrm{cm/s}$ onde evitare collisioni con gli ostacoli e dare la possibilità al robot di allinearsi al muro in modo corretto.

Una serie di test condotti all'interno degli edifici del Dipartimento ha mostrato la correttezza del sistema realizzato. Il robot ha percorso tutto lo spazio a disposizione senza accusare la necessità di supervisione, mantenendosi a distanza costante dalla parete senza oscillazioni apprezzabili. Il comportamento si adegua velocemente alle variazioni graduali dell'angolo del muro ed è in grado di superare senza problemi le rientranze fino a 20 cm circa. Per discontinuità di entità maggiore il robot tende a curvare repentinamente e correggere in seguito la direzione con qualche oscillazione residua. Se lo spazio si riduce eccessivamente la velocità cala ed il comportamento riesce a riportare il Nomad nella corretta direzione di moto, evitando le collisioni con le pareti.

Per quanto concerne il *main* di questo modulo, illustrato nel listato numero 4.2, si può notare come esso sia molto simile a quello dei client analizzati nel capitolo 3. In particolare, in questo caso viene dichiarato un component di nome *WallClient*

```
// smartNomad200Wallclient.cc


CHS::SmartThreadManager
*threadManager = CHS::SmartThreadManager::instance();

component = new CHS::SmartComponent("WallClient",argc,argv);
CHS::PushTimedClient<Smart::CommDistanceData>
 infraredPushTimedClient(component,"smartNomad200Server","distanceir");
CHS::SendClient<Smart::CommNavigationVelocity>
 navigationVelocitySendClient (component,
"smartNomad200Server","navigationvelocity");

 InfraredClientThread    infraredClientThread
( infraredPushTimedClient ,   navigationVelocitySendClient );

 infraredClientThread .open();
component->run();
threadManager->wait();
```

Listato 4.2: Parti essenziali del main relativo al wall following.

che verrà inviato assieme al *component smartNomad200Server*, illustrato nel listato numero 3.13, ai due oggetti di tipo *PushTimedClient* e *SendClient*. Da ultimo il *component* viene messo in esecuzione per ricevere gli aggiornamenti dai sensori infrarossi ed andare ad impostare le velocità, da lui stesso calcolate, sul robot.

4.2 *Center Following*

La seconda applicazione che andiamo ad analizzare consente al Nomad di muoversi all'interno del corridoio del Dipartimento mediante la sola navigazione sensoriale, ponendosi al centro del corridoio stesso e mantenendosi equidistante dalle due pareti.

4.2.1 Progetto

L'applicazione che si vuole realizzare consiste nel controllo continuo del percorso del robot; questi deve seguire una traiettoria centrale e quindi equidistante dai muri del corridoio. Il sistema, schematizzato in figura 4.5, è retto dal seguente modello matematico (utilizzato anche nell'applicazione precedente):

$$\begin{cases} \dot{\alpha} = \omega \\ \dot{d} = v \sin \alpha \end{cases} \tag{4.6}$$

Dove v e ω sono rispettivamente la velocità lineare ad angolare del robot, α rappresenta l'angolo tra il muro e la direzione istantanea di movimento e d misura la distanza che si vuole controllare. La velocità lineare del robot viene mantenu-

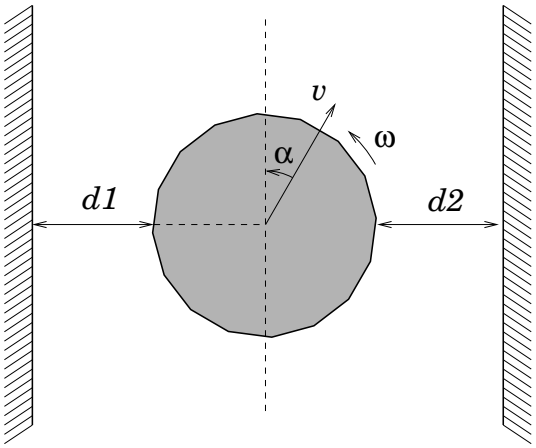

Figura 4.5: Modello geometrico del sistema.

ta costante durante il moto, per cui l'unico *ingresso manipolabile* del sistema è la velocità angolare ω.

Per regolare dinamicamente il percorso centrale del robot all'interno del corridoio, viene utilizzata una tecnica di *controllo in retroazione* realizzata in due fasi. Il primo anello di retroazione (figura 4.2) consente di regolare l'angolo α impostando una velocità di rotazione ω proporzionale alla differenza tra il *set-point* istantaneo α^* e il valore misurato $\hat{\alpha}$. In particolare la velocitá ω sarà positiva se il robot si trova più vicino al muro sulla destra che a quello alla sua sinistra, generando una svolta a sinistra e viceversa. Postulata la validitá del modello proposto, il *regolatore*

proporzionale (avente guadagno pari a K_α) assicura la convergenza verso l'esatto valore del set-point grazie al comportamento integrale del sistema. Da semplici considerazioni geometriche si deduce che dovrà essere: $K_\alpha < 0$.

Il secondo anello di retroazione (figura 4.3) può a questo punto sfruttare il set-point α^* per impostare una correzione dinamica che consenta al robot di portarsi in posizione centrale; in particolare il valore di d^* sarà zero in quanto vogliamo che le due distanze: d_1 e d_2 dai due muri siano identiche. Il termine che verrà sottratto a d^* rappresenta la differenza tra le distanze minime del robot dal lato sinistro e da quello destro (d_1 - d_2). È così possibile ricavare la distanza dal posizionamento centrale.

Siano $\{d_i\}_{i=1..7}$ e $\{d_i\}_{i=9..15}$ rispettivamente le misure di distanza rilevate dai sensori posti lungo il lato sinistro del robot e lungo il lato destro; definiamo m come la posizione del sensore che ha misurato la distanza minore sul lato sinistro e n come la posizione del sensore che ha misurato la distanza minore sul lato destro:

$$m : d_m = \min_{j=1..7}\{d_i\}; \tag{4.7}$$

$$n : d_n = \min_{j=9..15}\{d_i\}; \tag{4.8}$$

I valori di α e d sono dati da:

$$\begin{cases} \alpha = \alpha_\Delta + \alpha_\delta \\ d = d_1 - d_2 \end{cases}; \tag{4.9}$$

Dove i due contributi all'angolo α derivano dalla geometria dell'anello di sensori in figura 4.4, nel caso in cui il robot sia più vicino al muro di sinistra che a quello di destra; in caso contrario derivano dalla geometria dell'anello di sensori in figura 4.6:

- α_Δ dipende dall'indice m del sensore di sinistra che ha rilevato la minore distanza (nel caso in cui il muro di sinistra sia più vicino al robot di quello di destra):

$$\alpha_\Delta = \frac{\pi}{8}(m-4); \tag{4.10}$$

- α_δ viene ricavato dalle misure dei sensori $m-1$ ed $m+1$ (nel caso in cui il muro di sinistra sia più vicino al robot di quello di destra):

$$\sin \alpha_\delta = \frac{d_{m-1} - d_{m+1}}{l}. \tag{4.11}$$

- α_Δ dipende dall'indice n del sensore di destra che ha rilevato la minore distanza (nel caso in cui il muro di sinistra sia più lontano al robot di quello di destra):

$$\alpha_\Delta = \frac{\pi}{8}(n-12); \tag{4.12}$$

- α_δ viene ricavato dalle misure dei sensori $n-1$ ed $n+1$ (nel caso in cui il muro di sinistra sia più lontano al robot di quello di destra):

$$\sin \alpha_\delta = \frac{d_{n-1} - d_{n+1}}{l}. \tag{4.13}$$

4.2.2 Realizzazione

Per ricavare la posizione del robot, in navigazione all'interno del corridoio, sono stati utilizzati sia i sensori sonar, che garantiscono misure attendibili per distanze non troppo limitate, che quelli infrarossi che forniscono misure più precise quando il Nomad si trova molto vicino agli ostacoli.

Il modulo `smartNomad200Centclient` si basa sulla classe *InfraredClient-Thread*, illustrata nel listato 4.3; quest'ultima riceve gli aggiornamenti sui valori degli infrarossi e dei sonar grazie all'utilizzo di oggetti del tipo *PushTimedClient* e va ad impostare i valori di velocità sul robot attraverso un oggetto di tipo *SendClient*. All'interno di questa classe vengono definiti:

- un oggetto *dis* di tipo *CommDistanceData* che conterrà materialmente i valori ricavati dai sonar;

- un oggetto *disinfra* di tipo *CommDistanceData* che conterrà materialmente i valori ricavati dagli infrarossi;

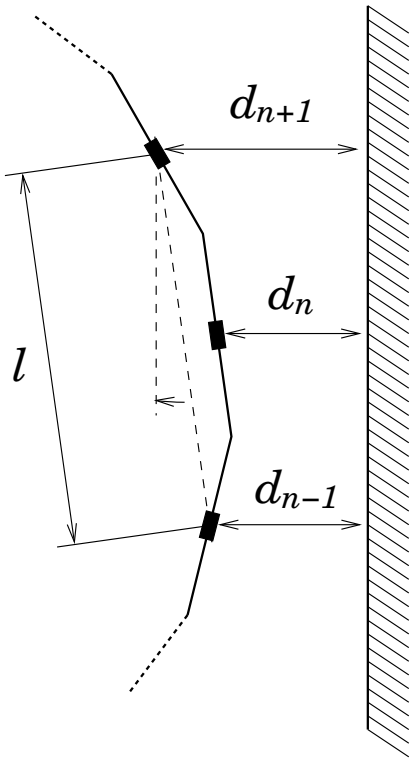

Figura 4.6: Stima di α_δ e d mediante i sensori del robot.

- una variabile *desiredDistance* di tipo *double* con valore di 0 cm differenza desiderata tra la distanza dal muro di destra e quello di sinistra;
- una variabile *VEL_LIN* contenente la velocità massima del robot durante questa applicazione, fissata a 25 $\mathrm{cm/s}$ normalmente;
- una array *v_values* che conterrà i sedici valori *filtrati* delle letture sensoriali; vengono letti sia i valori dei sonar che quelli degli infrarossi. Ogni valore di questo vettore rappresenta la misura più accurata fra la lettura dei sonar e quella degli infrarossi per ogni coppia di sensori corrispondenti.
- una serie di variabili utili al funzionamento del modulo.

Ad ogni iterazione il codice del modulo provvede a stimare i valori correnti di α e d (sulla base dei dati sensoriali forniti dai moduli *smartNomad200InfraredServer* e *smartNomad200SonarServer* trattati nel capitolo 3) ed a calcolare il nuovo valore

```
// smartNomad200Centclient.cc

class InfraredClientThread : public CHS::SmartTask
{
public:
  InfraredClientThread (CHS::PushTimedClient<Smart::CommDistanceData>
  &sonar, CHS::PushTimedClient<Smart::CommDistanceData> &infrared,
  CHS::SendClient<Smart::CommNavigationVelocity> &motor)
  :_sonar(sonar ), _infrared ( infrared ), _motor(motor) {}
  int svc ();

private:
  CHS::PushTimedClient<Smart::CommDistanceData> &_sonar;
  CHS::PushTimedClient<Smart::CommDistanceData> &_infrared;
  CHS::SendClient<Smart::CommNavigationVelocity> &_motor;
  // Distanza dal muro controllata
  Smart::CommDistanceData dis; //sonar
  Smart::CommDistanceData disinfra; // infrarossi
  const static double desiredDistance = 0.0; // centimetri
  const static double VEL_LIN = 15.0; // centimetri /s
  // Distanza geometrica tra due sensori in posizione i e i+2
  const static double SENSOR_DISTANCE = 17.0; // Distanza = 17 cm
  double wallDistances ; // distanza minima a sinistra
  double wallDistanced ; // distanza minima a destra
  double wallDistancefin ; // differenza delle due
  double wallAngle;
  double s_values [16]; // valori sonar
  double i_values [16]; // valori infrarossi
  double v_values [16]; // valori finali
  void sensorRefresh ();
  static double P_control ( double K, double setPoint ,
  double measure, double maxVal, double minVal );
};
```

Listato 4.3: Classe InfraredClientThread.

di ω mediante il controllo di tipo proporzionale. Da ultimo il modulo provvede ad inviare al componente *smartNomad200VelServer* i nuovi valori di velocità. Per garantire l'allineamento delle ruote con la torretta, il moto rotatorio sarà sempre lo

stesso per entrambi gli assi (torretta e ruote).

Come nel modulo precedente sono state calcolate sperimentalmente le costanti proporzionali da utilizzare all'interno degli anelli di retroazione.

Una serie di test condotti all'interno del corridoio del Dipartimento (Palazzina 1) ha mostrato la correttezza del sistema realizzato. Il robot ha percorso tutto lo spazio a disposizione senza la necessità di supervisione, mantenendosi al centro senza oscillazioni apprezzabili anche nelle situazioni più complicate.

4.3 *Gestione simultanea dei behaviour*

L'ultima parte della sperimentazione ha riguardato la creazione di un modulo per la gestione dei behaviour, denominato `master` (seguendo la terminologia OROCOS) che svolge in effetti la funzione di arbitro. Questo modulo gestisce la presenza contemporanea delle due applicazioni precedentemente analizzate in questo capitolo, in modo tale che si possano collegare al server centrale e realizzare i loro compiti quando determinate condizioni sono rispettate. In particolare le due applicazioni non possono essere attive e richiedere servizi contemporaneamente al server centrale `smartNomad200Server`, trattato nel capitolo 3; tuttavia esse continuano ad elaborare i dati ricevuti e a produrre i valori da impostare sul Nomad. Sarà poi il modulo `master` a decidere, in funzione dei dati sensoriali ricevuti, quale delle due applicazioni rendere operativa. Il Nomad ha così a disposizione due comportamenti: *wall following* e *center following*, inoltre il modulo `master` è realizzato in modo tale che il robot si possa muovere all'interno del corridoio della Dipartimento mantenendosi al centro del corridoio stesso finchè non incontra un ostacolo (ad esempio la fine del corridoio); a questo punto il modulo `master` disattiva il comportamento di *center following* e attiva quello di *wall following* permettendo al robot di aggirare l'ostacolo. Una volta superato l'ostacolo torna attivo il *center following*.

4.3.1 Modulo master

Il modulo `Master` si basa sulla classe *MasterClientThread* che riceve le letture sensoriali degli infrarossi grazie ad un oggetto di tipo *PushTimedClient*; i valori ri-

cevuti saranno poi elaborati all'interno del metodo *svc*, appartenente a questa classe e illustrato nel listato 4.4, al fine di individuare quale comportamento il robot debba adottare. All'interno della classe *MasterClientThread* vengono definiti:

- un oggetto *dis* di tipo *CommDistanceData* che conterrà materialmente i valori ricavati dagli infrarossi;
- due variabili di tipo *int* che memorizzeranno quante volte sono risultati attivi i due comportamenti;
- una variabile di tipo *enum* che indica se non è attivo nessun comportamento oppure quale comportamento è in esecuzione;
- una array *i_values* che conterrà i sedici valori degli infrarossi.

Il metodo *svc* (listato 4.4) utilizza il pattern *wiring* definito da OROCOS; infatti viene definito un oggetto *wiringmaster* che si occuperà di interfacciare i due comportamenti precedentemente trattati al server centrale. Vediamo come opera il pattern *wiring*: per prima cosa viene dichiarato un *wiringmaster* attraverso il comando CHS::WiringMaster wiringmaster(component) che riceve il nome del componente che svolgerà il ruolo di master; in questa applicazione il *wiringmaster* sarà proprio *smartNomad200Master*. In seguito il *wiringmaster* si occupa di interfacciare i vari componenti ad un prefissato modulo; in questa applicazione sarà `smartNomad200Server`, attraverso i comandi wiringmaster.disconnect(smartNomad200InfraredClient3,velPort) e wiringmaster.connect(smartNomad200InfraredClient3,velPort,smartNomad200Server,navigationvelocity). La prima istruzione consente di disconnettere il component *smartNomad200InfraredClient3* (comportamento di centerfollowing) dal server centrale. La seconda permette di collegare il component *smartNomad200InfraredClient3* al server centrale `smartNomad200Server` per usufruire del servizio di *navigationvelocity*. La gestione dei due behaviour da parte del modulo `Master` è illustrata in figura 4.7.

Il metodo *svc* (listato 4.4) si sottoscrive per ricevere gli aggiornamenti dagli infrarossi uno ogni dieci; l'iterazione del codice di questo metodo avverrà ogni quattrocento millisecondi visto che il tempo di aggiornamento dei sensori infrarossi è di quaranta millisecondi. In particolare ad ogni iterazione del codice viene controllato il valore dell'infrarosso frontale (numero zero), se il valore è inferiore a $50\ \mathrm{cm}$

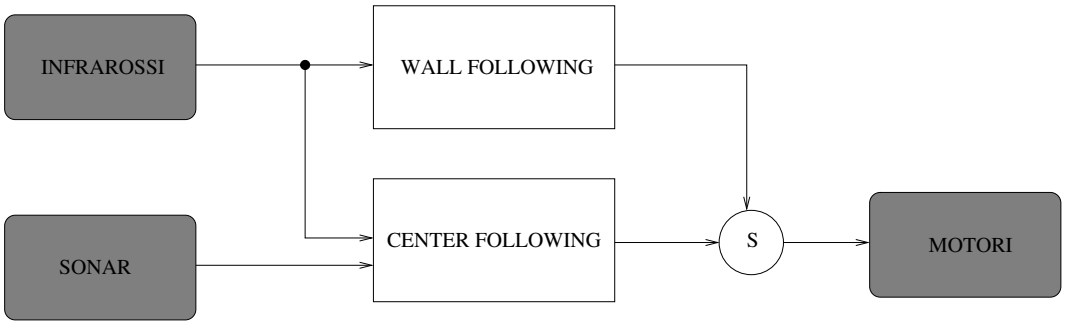

Figura 4.7: Gestione dei due behaviour con rappresentazione sumbsumtion.

ed è attivo il comportamento di *centerfollowing* quest'ultimo viene disattivato; in seguito si attiva il comportamento di *wallfollowing*. Se la distanza risulta superiore alla soglia precedentemente fissata, si procede ad attivare il comportamento di *centerfollowing* quando non vi è nessun comportamento attivo; nel caso in cui sia attivo il comportamento di *wallfollowing* viene immediatamente disattivato prima dell'attivazione di quello di *centerfollowing*.

```
// smartNomad200Master.cc

int MasterClientThread :: svc(void)
{
  CHS::WiringMaster wiringmaster(component);
  CHS::StatusCode status , statusw;
  CHS::StatusCodeConversion(_infrared. subscribe (10));
  while(true)
  {
     status = _infrared .getUpdateWait(dis );
     // controllo sulla ricezione degli infrarossi e memorizzazione
     // dei valori all' interno dell' array i_values ...
       if ( i_values [0] < 50 ) {
         if ( x == CENTCONN )
          statusw = wiringmaster. disconnect ("smartNomad200InfraredClient3",
          "velPort");
         if ( ( x == NONCONN) || (x == CENTCONN)) {
           x = WALLCONN;
           statusw = wiringmaster. connect("smartNomad200InfraredClient1",
           "velPort","smartNomad200Server","navigationvelocity");
         }
         a++;
       }
       else {
         if ( x == NONCONN ) {
          x = CENTCONN;
          statusw = wiringmaster. connect("smartNomad200InfraredClient3",
          "velPort","smartNomad200Server","navigationvelocity");
         }
         else {
           if ( x == WALLCONN) {
             statusw = wiringmaster. disconnect ("smartNomad200InfraredClient1",
             "velPort");
             x = CENTCONN;
             statusw = wiringmaster. connect("smartNomad200InfraredClient3",
             "velPort","smartNomad200Server","navigationvelocity");
           }
         }
         b++;
       }
  }
  CHS::StatusCodeConversion(_infrared. unsubscribe ());
  return 0;
}
```


Listato 4.4: Metodo svc della classe MasterClientThread.

4.4 *Dati statistici*

In questo paragrafo sono riportati alcuni dati statistici ricavati dal funzionamento del *Nomad 200*. In particolare il robot ha navigato all'interno del corridoio del Dipartimento per un'ora esatta ad una velocità di 25 $\mathrm{cm/s}$ utilizzando sia il comportamento di *CenterFollowing* che quello di *WallFollowing*, in presenza di ostacoli fissi e dinamici. Il *Nomad 200* ha navigato per il $91,3\%$ del tempo con un comportamento di *CenterFollowing* e per il restante $8,7\%$ con quello di *WallFollowing*. Durante questo periodo di tempo sono stati riscontrati due urti che non hanno compromesso il funzionamento del robot.

Durante l'utilizzo del comportamento di *WallFollowing* è stata mantenuta una distanza media dal muro di 18,3 cm a fronte di una distanza desiderata di 20 cm con una varianza σ pari a 17,11. Il comportamento di *CenterFollowing* ha evidenziato uno scostamento medio dal centro del corridoio di 9,3 cm con una varianza σ pari a 33,3.

Vediamo nel dettaglio il funzionamento del robot durante un periodo di navigazione.

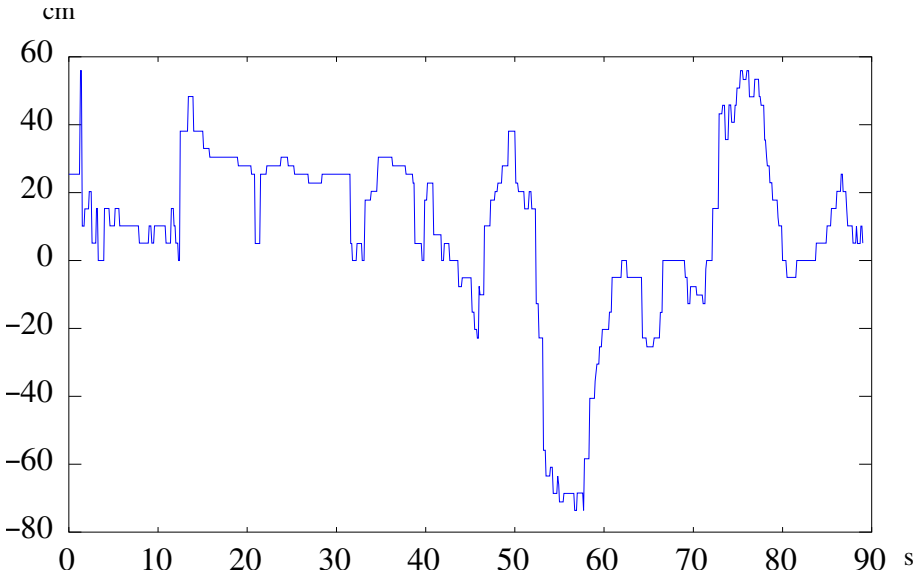

Figura 4.8: Navigazione del robot al centro del corridoio.

In figura 4.8 è riportato il percorso seguito dal robot durante un primo tratto di *CenterFollowing*. Come si può notare il robot mostra un ottimo funzionamento

mantenendosi al centro del corridoio (i picchi nel grafico sono dovuti alla presenza di ostacoli o a rientranze nei muri del corridoio).

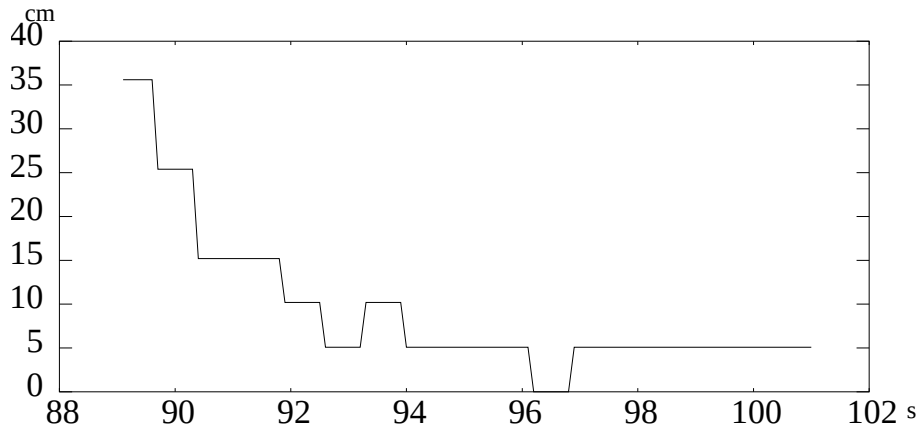

Figura 4.9: Comportamento di WallFollowing.

Nel tratto successivo (figura 4.9) il robot attiva il comportamento di *WallFollowing* avendo raggiunto la fine del corridoio. Qui è evidenziato come questo comportamento venga attivato in situazioni di emergenza per evitare ostacoli; naturalmente la distanza mantenuta dal muro non può essere esattamente quella voluta proprio per la situazione estremamente complicata in cui si trova il robot. Subito dopo aver

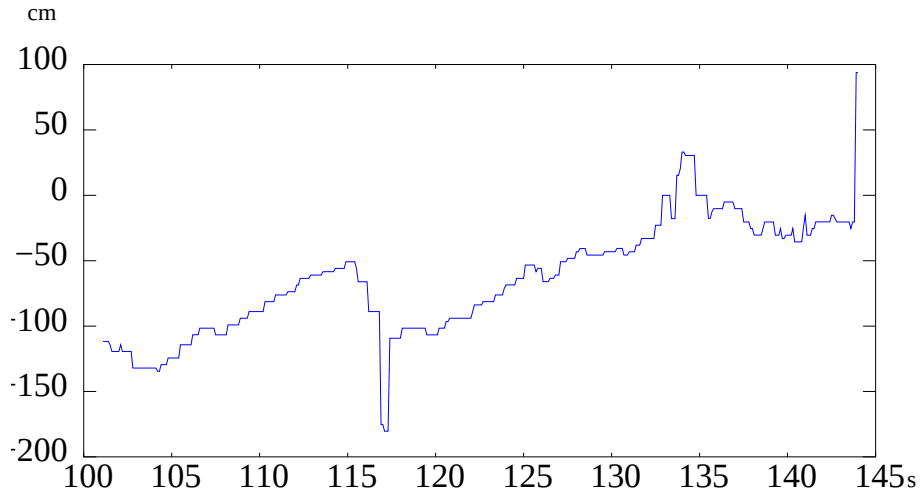

Figura 4.10: Navigazione del robot al centro del corridoio.

superato l'ostacolo viene riattivato il comportamento di *CenterFollowing* (figura

4.10); il robot impiega qualche secondo per allinearsi al centro del corridoio, ma poi riprende la sua navigazione ottimale.

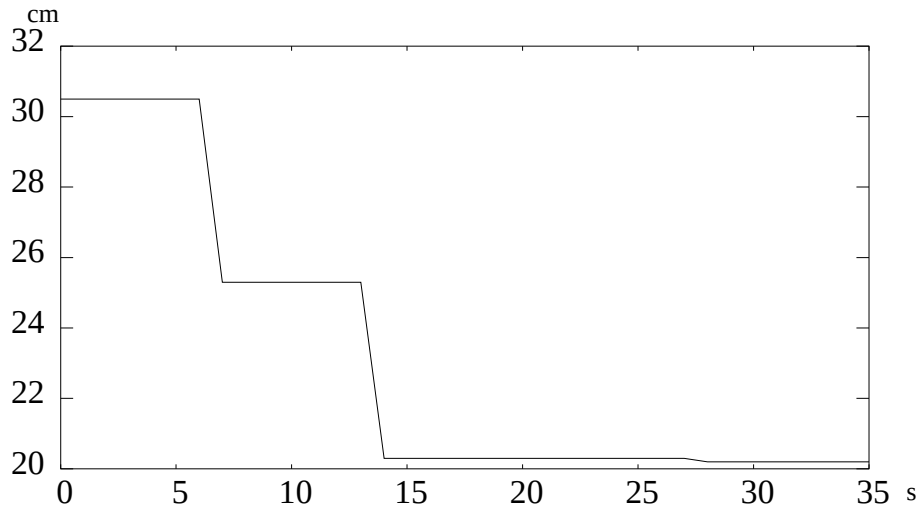

Figura 4.11: WallFollowing ottimale.

In figura 4.11 è riportato il funzionamento del robot con il comportamento di *WallFollowing* in situazione non critica; si nota come il robot si avvicini al muro e mantenga da esso la distanza prefissata.

Capitolo 5

Conclusioni

In questa tesi è stato descritta la realizzazione di un'architettura robotica per il governo di un robot mobile, che consenta la realizzazione di applicazioni concrete in modo semplice e immune da errori. Essa garantisce la possibilità di sostituire agevolmente i componenti e di estendere il supporto per assicurare nuove funzionalità, grazie anche all'apertura verso gli standard del settore informatico.

La prima fase del lavoro di tesi è consistita nello studio del progetto *open source* sviluppato da OROCOS, descritto nel capitolo 2, al fine di disporre di una serie di pattern di comunicazione per lo scambio d'informazioni tra i moduli che caratterizzano la nuova architettura realizzata.

Il lavoro realizzato in questa tesi ha portato alla costituzione di un insieme di moduli indipendenti, attivando così un'architettura robotica di tipo reattivo che dispone di un set completo di *primitive* per la comunicazione; questa architettura può gestire più *behaviour* operanti contemporaneamente sur robot. Inoltre non è più necessario utilizzare *robotd*, il software di controllo originario del *Nomad 200*, grazie alla realizzazione di un'ulteriore serie di moduli che interagiscono direttamente con le schede presenti sul robot al fine di ricevere informazioni riguardanti i sensori e i motori e di controllare il robot.

L'ultima fase del lavoro di tesi ha consentito la creazione di moduli compatibili con la nuova architettura del Nomad 200; questi moduli permettono di analizzare e di verificare concretamente il funzionamento del progetto sviluppato. In particolare i *behaviour* realizzati hanno permesso lo sviluppo di alcune applicazioni di naviga-

zione, evidenziando buone prestazioni pur mantenendo una sufficiente semplicità, astrazione e rapidità di sviluppo nella realizzazione del codice.

I probabili sviluppi e relative utilizzazioni che in futuro possono derivare da questo lavoro di tesi sono i seguenti:

- Lo studio e l'utilizzo di ulteriori protocolli di comunicazione da utilizzare in particolari situazioni complesse dove è necessaria la presenza di specifiche forme di comunicazione; valga come esempio il *query* pattern previsto nello standard OROCOS.
- La costruzione di un insieme di moduli che permettano di riprodurre alcune funzionalità assicurate dal precedente software di controllo e non replicate perchè ritenute d'importanza secondaria.

Appendice A

Codice dei behaviour per il Nomad 200

In questa appendice viene illustrato il codice sorgente dei moduli che sono stati realizzati per verificare il comportamento del *Nomad 200*. I listati riportano soltanto le parti più significative del codice, che è stato parzialmente ridotto per aumentarne la chiarezza. Per la descrizione degli algoritmi riguardanti la sperimentazione si veda il capitolo 4.

A.1 *Behaviour* per il robot

Listato A.1: Comportamento "*Center Following*" per il robot.

```
#include <cmath>
#include <iostream>
#include <sys/time.h>
#include <fstream>
#include "smartSoft.hh"

#include "commDistance.hh"
#include "commNavigationVelocity.hh"
```

```
CHS::SmartComponent *component;


class InfraredClientThread : public CHS::SmartTask
{
public:
  InfraredClientThread (CHS::PushTimedClient<Smart::CommDistanceData>
  &sonar, CHS::PushTimedClient<Smart::CommDistanceData> &infrared,
  CHS::SendClient<Smart::CommNavigationVelocity> &motor)
  :_sonar(sonar ), _infrared ( infrared ), _motor(motor) {}

  int svc ();
  ofstream out;
private:
  CHS::PushTimedClient<Smart::CommDistanceData> &_sonar;
  CHS::PushTimedClient<Smart::CommDistanceData> &_infrared;
  CHS::SendClient<Smart::CommNavigationVelocity> &_motor;
  // Distanza dal muro controllata
  Smart::CommDistanceData dis;
  Smart::CommDistanceData disinfra;

  const static double desiredDistance = 0.0; // centimetri
  const static double VEL_LIN = 15.0; // centimetri /s
  // Distanza geometrica tra due sensori in posizione i e i+2
  const static double SENSOR_DISTANCE = 17.0; // Distanza = 17 cm
  double wallDistances ;
  double wallDistanced ; // centimetri
  double wallDistancefin ; // centimetri
  double wallAngle;

  double s_values [16];
  double i_values [16];
  double v_values [16];
  void sensorRefresh ();
  static double P_control ( double K, double setPoint ,
  double measure, double maxVal, double minVal );
};

int InfraredClientThread :: svc(void)
```

```
{
  Smart::CommNavigationVelocity vel;
  CHS::StatusCode status;
  struct timeval *tempi;
  CHS::StatusCodeConversion(_sonar.subscribe(1));
  CHS::StatusCodeConversion(_infrared.subscribe(1));
  while(true)
  {
     status = _sonar.getUpdateWait(dis);
     status = _infrared.getUpdateWait(disinfra);
     if ( status != CHS::SMART_OK)
     {
       CHS::StatusCodeConversion(status);
     }
     else {

       sensorRefresh();
       double desiredAngle = P_control(0.06, desiredDistance,
        wallDistancefin , 1.05, 0.05);
       double rotatSpeed = P_control(-150.0, desiredAngle,
       wallAngle , 450.0, 0.0);
       double traslSpeed = v_values[1] * 100.0 / 35.0;
       if ( traslSpeed > VEL_LIN )
         traslSpeed = VEL_LIN;
       if ( traslSpeed < 15.0 )
         traslSpeed = 0.0;

       if ((s_values[0] < 80.0)||(s_values[1] < 80.0)||
       (s_values[15] < 80.0)) {
            traslSpeed = 7.0;
       }

       vel.set_v(traslSpeed, 0.01);
       vel.set_omega(rotatSpeed * 0.00175);
       status = _motor.send(vel);
     }
  }
  CHS::StatusCodeConversion(_sonar.unsubscribe());

  return 0;
}
```

```
void InfraredClientThread :: sensorRefresh ()
{

  for ( std :: size_t i = 0; i < 16; ++ i ) {
    // Le letture sono in pollici ( circa )
    s_values [ i ] = dis . get_measured_distance ( i , 0.01);
  }

  for ( std :: size_t i = 0; i < 16; ++ i ) {
    // Le letture sono in pollici ( circa )
    i_values [ i ] = disinfra . get_measured_distance ( i , 0.01);
  }

  for ( std :: size_t i = 0; i < 16; ++ i ) {

    if ( i_values [ i ] < 70.0 )
      v_values [ i ] = i_values [ i ];
    else
      v_values [ i ] = s_values [ i ];
  }

  // Calcola il valore minimo
  std :: size_t minPoss = 4;  // valore di default == robot allineato
  for ( std :: size_t i = 1; i < 5; ++ i ) {
    if ( v_values [ i ] < v_values [ minPoss ] )
      minPoss = i ;
  }

  wallDistances = v_values [ minPoss ];
  std :: size_t minPosd = 12;  // valore di default == robot allineato
  for ( std :: size_t i = 12; i < 16; ++ i ) {

    if ( v_values [ i ] < v_values [ minPosd ] )
      minPosd = i ;
  }
  wallDistanced = v_values [ minPosd ];
  wallDistancefin = wallDistances - wallDistanced ;
  double delta ;

  if ( wallDistancefin < 0 ) {
```

```
      delta = v_values[minPoss-1] - v_values[minPoss+1];
      double sinalpha = delta / std :: sqrt ( SENSOR_DISTANCE *
      SENSOR_DISTANCE + delta * delta );
      wallAngle = std :: asin ( sinalpha ) + 22.5 / 180.0 * M_PI *
      ( static_cast <int>(minPoss ) - 4 );
    }
    else {
      delta = v_values[minPosd-1] - v_values[minPosd+1];
      double sinalpha = delta / std :: sqrt ( SENSOR_DISTANCE *
      SENSOR_DISTANCE + delta * delta );
      wallAngle = std :: asin ( sinalpha ) + 22.5 / 180.0 * M_PI *
      ( static_cast <int>(minPosd ) - 12 );
    }

}

  double InfraredClientThread :: P_control ( double K, double setPoint ,
  double measure, double maxVal, double minVal )
  {
  double val = K * ( setPoint - measure);

   if ( val > maxVal )
     return maxVal;

   if ( val < -maxVal )
     return -maxVal;

   if ( std :: abs( val ) < minVal )
     return 0.0;

  return val;
  }


int main (int argc , char *argv [])
{
  try
  {
    CHS::SmartThreadManager *threadManager =
    CHS::SmartThreadManager::instance();
    component = new CHS::SmartComponent(
```

```
    "smartNomad200InfraredClient3",argc,argv);
    CHS::PushTimedClient<Smart::CommDistanceData>
    sonarPushTimedClient(component,"smartNomad200Server","distanceson");
    CHS::PushTimedClient<Smart::CommDistanceData>
    infraredPushTimedClient(component,"smartNomad200Server","distanceir");
    CHS::SendClient<Smart::CommNavigationVelocity>
      navigationVelocitySendClient (component,
    "smartNomad200Server","navigationvelocity");
    InfraredClientThread infraredClientThread(sonarPushTimedClient,
    infraredPushTimedClient, navigationVelocitySendClient);
    infraredClientThread.open();
    component->run();
    threadManager->wait();
  }
  catch (const CORBA::Exception &)
  {
    std::cerr << "CORBA exception" << std::endl;
    return 1;
  }
  catch (...)
  {
    std::cerr << "exception" << std::endl;
    return 1;
  }

  return 0;
}
```

Listato A.2: Comportamento "*Wall Following*" per il robot.

```
#include <cmath>
#include <iostream>
#include <sys/time.h>
#include <fstream>
#include "smartSoft.hh"

#include "commDistance.hh"
#include "commNavigationVelocity.hh"

CHS::SmartComponent *component;

class InfraredClientThread : public CHS::SmartTask
{
public:
  InfraredClientThread (CHS::PushTimedClient<Smart::CommDistanceData>
  &infrared, CHS::SendClient<Smart::CommNavigationVelocity> &motor)
  : _infrared ( infrared ), _motor(motor) {}
  int svc ();

private:
  CHS::PushTimedClient<Smart::CommDistanceData> &_infrared;
  CHS::SendClient<Smart::CommNavigationVelocity> &_motor;

  // Distanza dal muro controllata
  Smart::CommDistanceData dis;

  const static double desiredDistance = 20.0; // centimetri
  const static double VEL_LIN = 25.0; // centimetri/s

  // Distanza geometrica tra due sensori in posizione i e i+2
  const static double SENSOR_DISTANCE = 17.0; // Distanza = 17 cm
  double wallDistance; // centimetri
  double wallAngle; // radianti
  double i_values [16];
  void sensorRefresh ();
  static double P_control ( double K, double setPoint,
  double measure, double maxVal, double minVal );

};
```

```
int InfraredClientThread::svc(void)
{

  Smart::CommNavigationVelocity vel;
  CHS::StatusCode status;
  struct timeval *tempi;
  CHS::StatusCodeConversion(_infrared.subscribe(1));
  while(true) {
    status = _infrared.getUpdateWait(dis);
    if (status != CHS::SMART_OK)
    {
      CHS::StatusCodeConversion(status);
    }
    else {

      sensorRefresh();
      double desiredAngle = P_control(0.06, desiredDistance,
      wallDistance, 1.05, 0.05);
      double rotatSpeed = P_control(-300.0, desiredAngle,
      wallAngle, 450.0, 0.0);
      double traslSpeed = i_values[1] * 100.0 / 35.0;
      if (traslSpeed > VEL_LIN)
        traslSpeed = VEL_LIN;
      if (traslSpeed < 15.0)
        traslSpeed = 0.0;
      for (std::size_t i = 0; i < 2; ++i)
        if (i_values[i] < 20.0) {
          traslSpeed = 7.0;
        }

      vel.set_v(traslSpeed, 0.01);
      vel.set_omega(rotatSpeed * 0.00175);
      status = _motor.send(vel);
    }
  }
  CHS::StatusCodeConversion(_infrared.unsubscribe());
  return 0;
}

void InfraredClientThread::sensorRefresh()
```

```
{
   // Questo metodo calcola wallDistance e wallAngle
   // Aggiornamento del valore degli infrared memorizzato localmente
  for ( std :: size_t i = 0; i < 9; ++ i ) {
     // Le letture sono in pollici ( circa )
     i_values [ i ] = dis . get_measured_distance ( i , 0.01);
  }

   // Calcola il valore minimo
  std :: size_t minPos = 4;   // valore di default == robot allineato
  for ( std :: size_t i = 1; i < 8; ++ i )
    if ( i_values [ i ] < i_values [ minPos ] )
      minPos = i ;

  wallDistance = i_values [ minPos];
  double delta = i_values [ minPos-1] - i_values[minPos+1];
  double sinalpha = delta / std :: sqrt ( SENSOR_DISTANCE *
  SENSOR_DISTANCE + delta * delta );
  wallAngle = std :: asin ( sinalpha ) + 22.5 / 180.0 * M_PI *
  ( static_cast <int>(minPos ) - 4 );
}

double InfraredClientThread :: P_control ( double K, double setPoint ,
double measure, double maxVal, double minVal )
{
  double val = K * ( setPoint - measure);

   if ( val > maxVal )
     return maxVal;

   if ( val < -maxVal )
    return -maxVal;

   if ( std :: abs( val ) < minVal )
     return 0.0;

  return val ;
}

int main ( int argc , char *argv [])
{
```

```
  try
  {
    CHS::SmartThreadManager *threadManager =
    CHS::SmartThreadManager::instance();
    component = new CHS::SmartComponent(
    "smartNomad200InfraredClient1",argc,argv);
    CHS::PushTimedClient<Smart::CommDistanceData>
     infraredPushTimedClient(component,"smartNomad200Server","distanceir");

    CHS::SendClient<Smart::CommNavigationVelocity>
     navigationVelocitySendClient
    (component,"smartNomad200Server","navigationvelocity");
     InfraredClientThread  infraredClientThread (infraredPushTimedClient,
     navigationVelocitySendClient );
     infraredClientThread .open();
    component->run();
    threadManager->wait();
  }
  catch (const CORBA::Exception &)
  {
    std :: cerr « "CORBA exception" « std :: endl;
    return 1;
  }
  catch (...)
  {
    std :: cerr « "exception" « std :: endl;
    return 1;
  }

  return 0;
}
```

Bibliografia

[1] Open RObot COntrol Software, OROCOS. `http://www.orocos.org`.

[2] R.A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[3] Nomadic Technologies Inc. *The Nomad 200 User Guide*, December 1993.

[4] Galil Motion Control Inc. Multi Axis Motion Controllers. `http://www.galilmc.com`.

[5] N. Nilsson and R.E. Fikes. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5(2), 1971.

[6] R. Arkin. *Behavior-based Robotics*. 1998.

[7] R.A. Brooks. A Robot that Walks; Emergent Behavior from a Carefully Evolved Network. In *IEEE International Conference on Robotics and Automation '89*, pages 292–296, May 1989.

[8] E. Gat. Integrating Planning and Reaction in a Heterogeneous Asynchronous Architecture for Controlling Mobile Robots. In *Tenth National Conference on Artificial Intelligence (AAAI)*, 1992.

[9] D. Lyons, A. Hendriks, and S. Mehta. Achieving robustness by casting planning as adaptation of a reactive system. In *IEEE conference on Robotics and Automation*, 1991.

[10] M. Schoppers. A Software Architecture for Hard Real-Time Execution of Automatically Synthetized Plans or Control Laws. In *Conf. on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS '94)*, March 1994.

[11] R. Alami et al. An Architecture for Autonomy. *The International Journal of Robotics Research*, 17(4):315–337, April 1998.

[12] OROCOS at LAAS. `http://www.laas.fr/~mallet/orocos`.

[13] S. Fleury, M. Herrb, and R. Chatila. GenoM: a Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture. Technical report, Laboratory for Analysis and Architecture of Systems (LAAS).

[14] C. Schlegel. Communications Patterns for OROCOS. Hints, Remarks, Specifications. Technical report, Research Institute for Applied Knowledge Processing (FAW), February 2002.

[15] OROCOS at FAW. *`http://www1.faw.uni-ulm.de/orocos`*.

[16] OROCOS at KTH. *`http://cogvis.nada.kth.se/orocos`*.

[17] R. Volpe et al. The CLARAty Architecture for Robotic Autonomy. Technical report, Jet Propulsion Laboratory (JPL), 2001.

[18] I.A.D. Nesnas. Toward Developing Reusable Software Components for Robotic Applications. Technical report, Jet Propulsion Laboratory (JPL), 2001.

[19] S.A. Blum. Towards a Component-based System Architecture for Autonomous Mobile Robots. Technical report, Institute for Real-Time Computer Systems, Technische Universität München.

[20] L.B. Becker and C.E. Pereira. SIMOO-RT: An Object-Oriented Framework for the Development of Real-Time Industrial Automation Systems. *IEEE Tansactions on Robotics and Automation*, 18(4):421–430, 4 August 2002.

[21] N.R.S. Raghavan and T. Waghmare. DPAC: An Object-Oriented Distributed and Parallel Computing Framework for Manufacturing Applications. *IEEE Tansactions on Robotics and Automation*, 18(4):431–443, 4 August 2002.

[22] D. Brugali and M.E. Fayad. Distributed Computing in Robotics and Automation. *IEEE Tansactions on Robotics and Automation*, 18(4):409–420, 4 August 2002.

[23] G. Butler, A. Gantchev, and P. Grogono. Object-oriented design of the subsumption architecture. *Software Practice and Experience*, 31:911–923, 2001.

[24] C. Pescio. C++, Java, C#: qualche considerazione. *C++ Informer*, (12), 12 October 2000. `http://www.eptacom.net`.

[25] M. Salichs et al. Pattern-Oriented Implementation for Automatic and Deliberative Skills of a Mobile Robot. In *1st Int'l Workshop on Advances in Service Robotics (ASER 03)*, 2003.

[26] R.E. Johnson. Frameworks = (Components + Patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[27] E. Gamma, R. Helm, R. Jhonson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[28] J.C. Cleaveland. Building application generators. *IEEE Software*, (4):25–33, July 1988.

[29] R. Arkin and T. Balck. AuRA: Principles and Practice in Review. *Journal of Experimental and Theoretical Artificial Intelligence*, 9:175–189, 1997.

[30] Directed Perception Inc. Pan-Tilt Units. `http://www.dperception.com`.

[31] B. Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26:251–321, 1985.

[32] J.J. Borrelly et al. The ORCCAD Architecture. *The International Journal of Robotics Research*, 17(4):338–359, April 1998.

[33] B. Meyer. *Object-Oriented Software Construction*. 2nd edition, 1997.

[34] Object Management Group. Real-Time CORBA Specification, August 2002.

[35] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*, volume 2. 2000.