Università degli Studi di Pisa
Facoltà di Ingegneria

Centro Interdipartimentale di Ricerca
"Enrico Piaggio"

TESI DI LAUREA IN INGEGNERIA INFORMATICA

Progettazione e sviluppo di una infrastruttura a componenti per il controllo di sistemi di agenti mobili

RELATORI: **Ch.mo Prof. Ing. A. Bicchi** _____
Ch.mo Prof. Ing. G. Dini _____
Ing. R. Schiavi _____

CANDIDATO: **De Paolis Giovanni** _____

Anno Accademico 2006/2007

Ai miei genitori.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Gli AUV (Autonomous Unmanned Vehicle) sono robot che hanno la capacità di muoversi autonomamente nel proprio ambiente senza essere vincolati a rimanere in un unico punto, al contrario di quanto accade per altri tipi di robot, quali ad esempio bracci robotici, che solitamente sono vincolati ad una superficie fissa.
In questo capitolo viene presentata una breve storia della robotica mobile e viene introdotto il quadro applicativo in cui questa tesi si colloca.

1.1 Storia della robotica mobile

I primi esempi di robot mobili si svilupparono in ambito bellico, durante la seconda guerra mondiale e sono il frutto delle scoperte fatte nell'ambito di nuovi campi di ricerca quali cibernetica e informatica. Essi erano principalmente la prima forma di bombe intelligenti, bombe aeree che detonavano solo ad una certa distanza dall'obiettivo e possedevano un sistema di guida e di controllo radar. Esempi di queste bombe erano il *V1* ed il *V2* visibili nelle figure 1.1(a) e 1.1(b). I primi robot mobili terrestri si hanno solo verso la fine degli anni quaranta, fra il 1948 ed il 1949, quando W. Greta Walter costruisce *Elmer* ed *Elise* due robot autonomi che per la loro forma ricordano delle tartarughe (Fig. 1.2). Il loro nome ufficiale era *Machina Spe-*

(a) V1

(b) V2

Figura 1.1: I missili V1 e V2.

culatrix poiché con il loro comportamento sembravano esplorare l'ambiente circostante. Elmer ed Elise erano equipaggiati con un sensore di luce. Trovata una fonte di luce si muovevano verso questa evitando o spastando gli ostacoli sul proprio cammino. Questi robot dimostrarono che si può ottenere un comportamento complesso da un progetto molto semplice dato che avevano l'equivalente di due cellule nervose. Per l'evoluzione di *Elmer* ed

Figura 1.2: Machina Speculatrxi (Elmer).

Elise si dovrà aspettare fino agli anni sessanta quando, fra il 1961 ed il 1963, la Johan Hopkins University sviluppa *Beast*, un robot in grado di muoversi utilizzando un sonar, con la capacità di ritornare al punto di ricarica nel momento in cui la carica delle batterie scende sotto la soglia critica. Nel 1969 nasce il primo robot commerciale, *Mowbot*, un robot tagliaerba. In figura 1.3 è mostrato il manifesto pubblicitario di questo robot. Nei primi anni settanta si ha un grosso passi avanti nell'ambito della robotica mobile. Lo Stanford Research Institute realizza *Shakey*, un robot così chiamato a causa del suo moto a scatti. *Shakey* (figura 1.4) era provvisto di una telecamera, un range finder, un sensore d'urto e una collegamento radio. *Shakey* fu il primo robot in grado programmare le proprie azioni, ovvero dato un comando generale il

Figura 1.3: Pubblicità Mowbot, robot tagliaerba.

robot elaborava i passi necessari per portarlo a termine. Verso la fine degli

Figura 1.4: Shakey.

anni settanta aumenta l'interesse del pubblico verso i robot anche grazie a film come Star Wars dove due personaggi cardine sono robot: R2D2 e C3PO. Grazie anche a questo interesse si intensifica lo studio verso robot domestici, con scopi di intrattenimento o educativi. Nascono così verso la fine degli anni settanta, primi anni ottanta *RB5X*, ancora prodotto, e *HERO* (figure 1.5(a) e 1.5(b) rispettivamente). Questi robot venivano controllati da un computer montato a bordo e provvisti di detettori di movimento. Gli anni successivi vedono la robotica aumentare di importanza e assurgere al ruolo di disciplina nelle università. Gli sforzi della ricerca sono orientati verso la realizzazione di robot sempre più affidabili, piccoli e dai costi contenuti. I risultati non si fanno attendere molto e nel 1991 nasce Khepera (Fig. 1.6(a)) il primo robot miniaturizzato, con un diametro di 55mm e un'altezza di 30mm. Tra il 1996 e il 1997 la Nasa invia Mars Pathfinder su marte con Sojourner (Fig. 1.6(b),

(a) RB5X.

(b) Hero.

Figura 1.5: Robot educativi.

un robot che esplora la superficie del pianeta comandato dalla terra. Un sofisticato sistema di collision avoidance permette a Sojourner di muoversi in un ambiente difficile. Nel 2002 viene commercializzato RoboSapien (1.6(c)), un robot biomorfo venduto come un giocattolo.

Al giorno d'oggi quindi gli sviluppi tecnologici della robotica mobile han-

(a) Khepera.

(b) Sojourner.

(c) Robosapien.

Figura 1.6: Robot contemporanei.

no consentito la penetrazione degli agenti mobili nella vita di tutti i giorni degli individui. Esempi sono i dispositivi per la pulizia dei pavimenti di case e piscine come quelli sviluppati da *iRobot* e mostrati in figura 1.7(a) e figura 1.7(b); o quelli utilizzati all'interno di ambienti abitativi come quelli sviluppati da *Mobile Robots inc.*

Questi robot eseguono compiti come il trasporto di oggetti fra gli uffici di un azienda o il controllo degli ambienti di un edificio; il robot *HVAC* ad esempio, permette di monitorare all'interno di uno stabile la qualità dell'aria,

(a) iRobot Scooba. Robot per la pulizia dei pavimenti.

(b) iRobot verro600. Robot per la pulizie delle piscine.

Figura 1.7: Esempi di cleaning robot.

la presenza di fumo e radiazioni oltre che monitorare le rete wifi e trovare gli accessi non autorizzati. Le figure 1.8(a), 1.8(b), 1.8(c) presentano alcuni dei robot della *Mobile Robots inc* all'opera.

(a) Mobile Robot. Robot per il trasporto di oggetti.

(b) Mobile Robot. Robot per il trasporto di vivande.

(c) Mobile Robot. HVAC, robot per la sorveglianza di ambienti.

Figura 1.8: Esempi building robot.

Entrando nell'ambito di applicazioni più particolari troviamo l'utilizzo dei robot mobili in circostanze ostili o a rischio per l'uomo. Un primo esempio sono i robot utilizzati in ambienti militari. L'*Eod*, visibile nelle figure 1.9(a) e 1.9(b), prodotto da *iRobot*, è un robot destinato ai reparti speciali con scopi di assistenza nel disinnesco di ordigni, sorveglianza e recupero ostaggi.

Il robot può rapidamente penetrare in aree inaccessibile e pericolose quali

(a) iRobot eod. Robot per applicazioni militari.

(b) iRobot eod con operatore.

Figura 1.9: Esempi di tactical robot.

macerie, fogne, tunnel e altri ambienti dove veicoli più larghi avrebbero difficoltà di accesso.
Un'altro esempio di robot utilizzati per eseguire compiti particolari sono quelli sviluppati da *inoctun* per l'ispezione di condutture. Questi robot (Fig. 1.10(a)-Fig. 1.10(d), forniti di telecamera, sono resistenti all'acqua e sono in grado di risalire all'interno della conduttura evitando gli ostacoli e superando i giunti di derivazione. I robot più piccoli possono entrare in condutture di 100mm di diametro.

(a) inoctun. Versatrax 100.

(b) inoctun. Versatrax 150.

(c) inoctun. Versatrax 150 Crawler.

(d) inoctun. Versatrax con operatore.

Figura 1.10: Esempi di robot per l'ispezione di condutture.

Per concludere questa panoramica sui robot mobili terrestri un ultimo esempio che vogliamo citare sono i robot guide museali. Anche se sono ancora dei prototipi sono cominciati ad apparire in diversi musei nel mondo. Questi robot accolgono il visitatore, si muovono all'interno del museo e illustrano le opere in esso contenute. Sono sistemi molto complessi, dotati di varie tipologie di sensori per potersi muovere nell'ambiente e per interagire con gli utenti. Le figure 1.11(a)- 1.11(c) ne mostrano alcune esempi. In questa

(a) Sage

(b) Rostro

(c) Minerva

Figura 1.11: Esempi di robot guide museali.

tipologia di robot spicca *Asimo*, un robot umanoide che, creato nel 2000 dalla Honda, è ora impegnato come guida per i visitatori in un padiglione del Miraikan, National Museum of Emerging Science and Innovation di Tokyo (Fig 1.12).

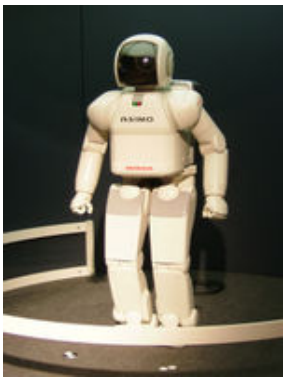


Figura 1.12: Honda Asimo.

1.2 Sistemi multi-agente

La crescente disponibilità di robot ha comportato la possibilità di avere più agenti in una stessa area di lavoro ciascuno dei quali compie il proprio task o collabora nell'assolvimento di un task comune. Vi è stata la necessità di sviluppare delle strategie di controllo che garantissero l'assenza di conflitti. L'insieme dei robot può essere visto come un sistema multi-agente (*multi-agente system* MAS). In informatica, un sistema multi-agente è un sistema composto da diversi agenti, capaci di mutua interazione. L'interazione può essere nella forma di scambio di messaggi o di cambiamento del comune ambiente di lavoro. Gli agenti possono essere entità autonome, quali agenti software o robot. Un MAS può anche essere composto da agenti umani. La società in generale può essere considerata un esempio di MAS. Caratteristica comune fra gli agenti è la non completezza delle informazioni in loro possesso o la limitata capacità di risolvere un problema. I sistemi multi-agente hanno conosciuto negli ultimi anni un notevole aumento di interesse e sono stati proposti per diverse applicazioni quali la gestione del traffico aereo, l'esplorazione del pianeta e la sorveglianza. I MAS introducono nuovi problemi da superare quali la gestione dell'informazione distribuita, la coordinazione fra gli agenti, la scelta dei protocolli di comunicazione, il progetto e la verifica di leggi di controllo decentralizzato e la sicurezza. Tornando nell'ambito dei robot mobili uno dei problemi fondamentali è quello di trovare delle leggi di controllo che garantiscano l'assenza di collisioni fra gli agenti. Esistono due approcci al problema: uno *centralizzato* e uno *decentralizzato*. In un approccio centralizzato un unico *decision maker* deve conoscere la configurazione corrente e quella desiderata di tutti gli agenti per poter determinare i controlli per ogni agente che garantiscano l'assenza di collisioni. Anche se esistono algoritmi centralizzati corretti e completi, essi richiedono un grande quantità di risorse computazionali. Inoltre questi approcci sono responsabili degli errori del decision maker che si riflettono un comportamento anormale degli agenti. L'approccio decentralizzato è più veloce nella reazione alle situazioni inaspettate ma la verifica della sicurezza è un problema e l'effetto domino dei

possibili conflitti può portare, in alcune condizioni, alla mancata convergenza alle soluzioni. Questo ultimo approccio, nonostante le difficoltà che presenta, appare quello più promettente per poter avere dei sistemi multi-agente autonomi anche in ambienti non prettamente concepiti per i robot.

1.3 Obiettivi della tesi

In questo lavoro di tesi è stata progettata e realizzata una infrastruttura che permette di controllare sistemi di robot mobili in ambienti con presenza di ostacoli sfruttando un approccio decentralizzato. I robot devono adempiere il compito globale di raggiungere dei goal prefissati, mentre il compito locale consiste nel mantenere un comportamento autonomo e indipendente nell'evitare eventuali ostacoli che possano impedire il raggiungimento dell'obiettivo, siano essi altri robot o ostacoli di natura generica.
Per aumentare la flessibilità dell'infrastruttura, svincolandola dalle particolari tecnologie usate, i robot sono stati sviluppati secondo una filosofia a componenti. Con componenti intendiamo delle entità che hanno degli scopi ben specifici all'interno di un sistema e che sono, in qualche modo, logicamente individuabili e separabili dal sistema stesso. Facendo ricorso ad una similitudine possiamo pensare ai componenti come ai mattoni di un edificio. In questo modo la nostra piattaforma può raggiungere un alto grado di riconfigurabilità ed eterogeneità. Infatti i robot possono essere adattati alle diverse situazioni e ne può essere alterato il comportamento aggiungendo o eliminando dei componenti, i quali, purché rispettino il protocollo per interfacciarsi al sistema, possono essere realizzati con tecnologie diverse rispetto a quelle già in uso.
Il lavoro di tesi si è particolarmente focalizzato su:

- Lo studio della politica di controllo degli agenti in un ambiente con presenza di ostacoli;
- Lo studio del middleware e delle tecnologie a componenti per dispositivi

embedded e l'implementazione dei componenti software del prototipo dell'agente mobile RaIN2;

- L'implementazione del server di visione come sistema di localizzazione degli agenti.

1.4 Struttura della tesi

La tesi presenta la seguente struttura:

Capitolo 2 Realizzazione di una politica per la gestione del traffico dei robot che garantisca, oltre al raggiungimento del goal, l'assenza di collisioni sia fra robot che verso ostacoli fissi.

Capitolo 3 Introduzione al Middleware e alle tecnologie a componenti, motivazioni che portano al loro utilizzo, principali vantaggi e svantaggi, panoramica sulle diverse tecnologie, scelte progettuali.

Capitolo 4 Descrizione della piattaforma robotica, componenti hardware utilizzati ed implementazioni del software.

Capitolo 5 Realizzazione di un server di visione per la localizzazione dei robot.

Capitolo 6 Test e conclusioni.

Appendice Codice realizzato.

Capitolo 2

Politica per la gestione del traffico dei robot

In questo capitolo vengono analizzate delle soluzioni per la realizzazione di una politica per la gestione di un insieme di robot in una ambiente con ostacoli statici. Punto di partenza del lavoro è la Generalized Roundabout Policy (GRP) sviluppata da Pallottino et al. in [1] [2]. Questa politica garantisce la sicurezza nella gestione degli agenti mobili in una ambiente privo di tali ostacoli. Utilizzando la GRP abbiamo cercato soluzioni che garantissero la sicurezza del moto degli agenti mobili anche in presenza di ostacoli fissi. Data l'importanza che ricopre la GRP nel nostro lavoro riportiamo nel seguito una breve descrizione della politica e del suo funzionamento, quindi le estensioni che sono state studiate.

2.1 Generalized Roundabout Policy

La Generalized Roudabout Policy (GRP) è una politica di controllo cooperativa e decentralizzata per veicoli non-olonomi con raggio di curvatura limitato che permette a questi, data una configurazione iniziale, di raggiungere in un tempo finito la configurazione finale evitando le collisioni. Questa politica garantisce, sotto certe condizioni iniziali, la sicurezza dalle collisioni per un

numero arbitrariamentre grande di veicoli. Inoltre attraverso una condizione (necessaria e sufficiente) sulla configurazione finale assicura il raggiungimento di quest'ultima, eliminando quindi eventuali stalli.
Vediamo ora la politica più in particolare. Consideriamo che i veicoli si muovano in un ambiente illimitato e privo di ostacoli, a velocità costante e con raggio di curvatura limitato. I veicoli sono a conoscienza solo della posizione e dell'orientamento attuali di alcuni veicoli entro un certo raggio di comunicazione. In particolare non vengono scambiate informazioni riguardo ai goal che i singoli agenti devono raggiungere o al particolare task che stanno svolgendo. Tutti gli agenti prendono delle decisioni basate su un comune insieme di regole decise a priori e assumono che tutti gli altri veicoli usino le stesse regole.

Consideriamo n veicoli. La configurazione dell'i-esimo agente è data da

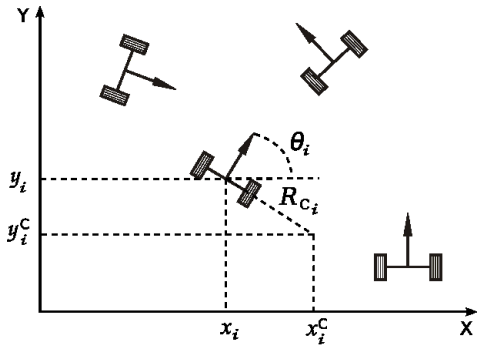


Figura 2.1: Rappresentazione sul piano cartesiano della configurazione dell'i-esimo agente.

$g_i \in SE(2)$, il gruppo di trasformazioni rigide sul piano. Come mostrato in figura 2.1 in coordinate la configurazione dell'i-esimo agente è data dalla tripla $g_i = (x_i, y_i, \theta_i)$ dove x_i e y_i specificano le coordinate del punto di riferimento del veicolo, mentre l'angolo di rotta θ_i è formato dall'asse longitudinale del corpo del veicolo con l'asse $x = 0$. R_{C_i} rappresenta il raggio di curvatura del veicolo. Ogni agente entra nell'ambiente di lavoro con una determinata configurazione iniziale $g_i(0) = g_{0,i} \in SE(2)$ e gli viene assegnata una configurazione finale $g_{i,f} \in SE(2)$. I veicoli si muovono su una traiettoria

continua $g_i : \mathbb{R} \to SE(2)$ in accordo con il modello

$$\begin{cases} \dot{x}_i(t) &= v_i \cos(\theta_i(t)) \\ \dot{y}_i(t) &= v_i \sin(\theta_i(t)) \\ \dot{\theta}(t) &= \omega_i(t) \end{cases} \tag{2.1}$$

dove $\omega_i : \mathbb{R} \to \left[-\frac{1}{R_{C_i}}, \frac{1}{R_{C_i}}\right]$ è il segnale di controllo di curvatura. Senza perdita di generalità poniamo che v_i sia costante e uguale ad 1 per ogni agente. Inoltre si considerano tutti gli agenti con il medesimo raggio di curvatura $R_{C_i} = R_C$ e senza perdita di generalità poniamo $R_C = 1$. Definiamo una mappa $d : SE(2) \times SE(2) \to \mathbb{R}^+$ come distanza fra due agenti; in coordinate

$$d(g_1, g_2) = \| (x_1, y_1) - (x_2, y_2) \|_2 \ .$$

Si dice che è avvenuta una *collisione* tra due veicoli se questi sono più vicini di una specificata distanza euclidea d_s, ovvero $d(g_i(t_c), g_j(t_c)) < d_s$. Quindi se associamo ad ogni agente mobile un *disco di sicurezza* di raggio $R_s = d_s/2$ centrato sulla posizione del veicolo una collisione avviene quando due dischi si sovrappongono.
La GRP si basa sul concetto di disco riservato legato ad ogni agente e visibile in figura 2.2. Data la configurazione g per un veicolo, il disco riservato ha rag-

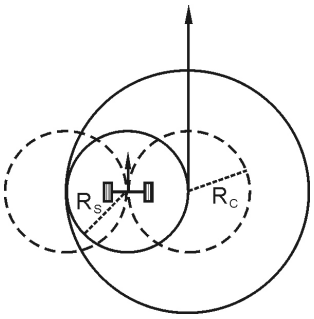


Figura 2.2: La regione riservata di un veicolo non olonomo.

gio $R = 1 + R_s$ e centro in $(x_i^C, y_i^C) = (x_i + \sin(\theta_i), y_i - \cos(\theta))$. Inoltre eredita lo stesso angolo di rotta θ_i del veicolo. La configurazione $g_i^C = (x_i^C, y_i^C, \theta_i^C)$ del

disco riservato ha dinamica $\dot{g}_i^C = ((1+\omega(t))\cos(\theta_i^C), (1+\omega(t))\sin(\theta_i^C), \omega(t))$. Si può notare come si possa interrompere il moto del disco riservato ponendo $\omega = -1$, ovvero facendo ruotare il veicolo con il massimo raggio di curvatura. Condizione sufficiente per evitare le collisioni è che gli interni dei dischi riservati non si sovrappongano mai dato che contengono il disco di sicurezza del veicolo. Quindi se il disco riservato del veicolo i-esimo è in contatto con i dischi degli agenti con indici in $J_i \subset \{1, \ldots, n\}$ il moto degli agenti è vincolato come segue

$$\dot{x}_i^C(x_i^C - x_j^C) + \dot{y}_i^C(y_i^C - y_j^C) \geq 0, \forall j \in J_i. \tag{2.2}$$

Quindi la velocità del veicolo i-esimo è costretta a rimanere in un cono convesso Θ, detto *cono ammissibile*, determinato dall'intersezione di un certo numero di semi-piani vicini. Θ è difinito come la mappa che associa alle configurazioni di un agente g e dei suoi vicini l'insieme delle possibili direzioni in cui la regione riservata dell'agente in esame può traslare senza violare i vincoli dati dalla 2.2, ovvero $\Theta : SE(2) \times 2^{SE(2)} \to 2^{\mathbb{S}^1}$[1].
Per un insieme connesso non vuoto $B \subset \mathbb{S}^1$, $B \neq \emptyset$, possiamo definire $\max(B)$ e $\min(B)$ come gli elementi sulla frontiera di B, rispettivamente nella direzione positiva e negativa rispetto alla bisettrice di B. Ora, indicando con $\overline{g} \subset \{g_1, \ldots, g_n\} \subset SE(2)$ l'insieme di cardinalità n che raccoglie le informazioni disponibili relative agli altri agenti possedute dal i-esimo veicolo, possiamo definire la mappa $\Theta^-(g, \overline{g}) = \Theta(g, \overline{g}) \setminus min(\Theta(g, \overline{g}))$. La figura 2.3 mostra un esempio di cono ammissibile con Θ^- che, in altre parole, è l'insieme aperto ottenuto rimuovendo il contorno di Θ in senso orario. Se Θ è un sottoinsieme proprio di $\mathbb{S}^1$, allora $\max(\Theta)$, $\min(\Theta)$ e Θ^- son ben definiti. Se $\Theta = \emptyset$, o $\Theta = \mathbb{S}^1$ allora si pone $\Theta^- = \Theta$.

Quando il controllo ω vale -1 il disco riservato è fermo e il veicolo si trova in uno stato detto `hold`. Quindi, in presenza di ostacoli, se l'angolo di rotta non appartiene al cono ammissibile ovvero $\theta \notin \Theta^-$ l'agente mobile può bloccare il moto del disco riservato entrando nello stato di `hold`.

[1] $\mathbb{S}^1 = \{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\}$

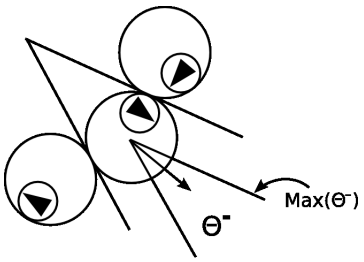


Figura 2.3: Insieme delle direzioni permesse in cui il centro dell'i-esima regione riservata può muoversi.

In presenza di un solo ostacolo fermo che si pone tra la posizione attuale del veicolo ed il suo goal, ponendo il controllo $\omega = (1 + R_s)^{-1}$ quando i due dischi si toccano e non appena la direzione del veicolo entra nel cono ammissibile, si ottiene l'aggiramento del disco riservato dell'ostacolo. Quando il veicolo si trova in questa situazione si dice che è in uno stato di `roll`.
Se l'ostacolo è mobile potrebbe accadere che la tangenza dei dischi riservati si perda durante l'esecuzione della manovra di aggiramento. Per questo viene aggiunto un nuovo stato, detto `roll2`, nel quale il controllo ω vale 1 in modo da poter ristabilire il contatto fra i dischi. In questo stato è possibile entrarvi solo se lo stato precedente era `roll`.
Se, invece, lo spazio è privo di ostacoli, l'agente mobile può tranquillamente raggiungere il proprio goal a partire dalla sua configurazione iniziale commutando lo stato da quello di `hold` a quello di `straight` dove il controllo vale $\omega = 0$. La commutazione avviene al momento opportuno, ovvero quando l'orientamento del veicolo è uguale all'orientamento del vettore Δ_f che congiunge la posizione attuale del centro del disco riservato con quella del centro del disco nella configurazione finale. Una volta che il centro del disco riservato ha raggiunto la posizione finale il veicolo ottiene l'orientamento finale rientrando nello stato di `hold`.
La politica della GRP è basata sui quattro modi di operazione accennati sopra in cui è assegnato un valore costante per il controllo ω. Il comportamento ad anello chiuso del veicolo può essere quindi modellizzata come un sistema ibrido i cui stati sono $Q = \{\texttt{roll}, \texttt{roll2}, \texttt{hold}, \texttt{straight}\}$.

In figura 2.4 è possibile vedere l'automa ibrido che descrive la GRP con gli stati descritti e le guardie che permettono il passaggio da uno stato all'altro. L'angolo ϕ è l'angolo che esprime l'orientamento del vettore Δ_f, mentre la

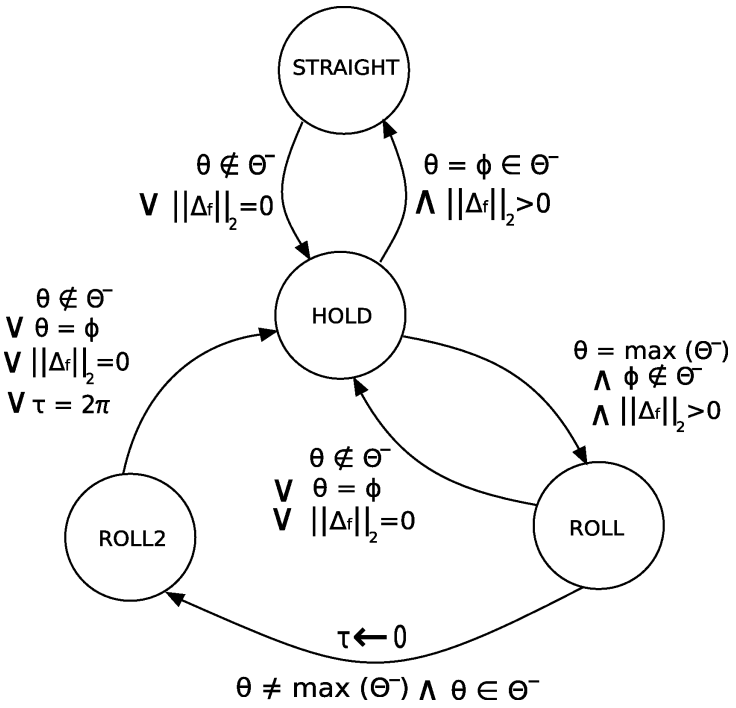


Figura 2.4: Automa ibrido che descrive la GRP.

quantità τ serve per aggiungere una condizione di time-out e far sì che l'agente esca dallo stato `roll2` se ha compiuto una rotazione di 2π rad.
Per quanto riguarda le condizioni di sicurezza ed eliminazione degli stalli consideriamo le due proprietà per una politica π:

- P_1: Un insieme di configurazioni $G = \{g_i, i = 1, , n\}$ è non sicuro per la politica π se esiste un insieme di configurazioni di arrivo $G_f = \{g_{f,i}, i = 1, \ldots, n\}$ tale che l'applicazione di π porta ad una collisione;
- P_2: Un insieme di configurazioni di arrivo $G_f = \{g_{f,i}, i = 1, \ldots, n\}$ è detto bloccante per la politica π se esiste un insieme di configurazioni $G = \{g_i, i = 1, \ldots, n\}$ da cui l'applicazione della politica π conduce ad un dead- o live-lock.

A questo punto possiamo dire che:

- *Proposizione 1*: La proprietà $P_1(G)$ è verificata per la GRP se e solo se i dischi riservati non si sovrappongono;
- *Proposizione 2*: La proprietà $P_2(G)$ è verificata per la GRP se G_f viola la condizione di non densità la quale enuncia che un qualsiasi cerchio di raggio $\rho(m)$ con $1 < m \leq n$ può contenere al massimo $m - 1$ centri dei dischi riservati.

Per le dimostrazioni e gli approfondimenti si rimanda a [1] [2].

2.2 Soluzioni per il Collision-Obstacle Avoidance

La Generalized Roudabout Policy garantisce il Collision Avoidance, ovvero che in un ambiente con più robot in movimento e senza ostacoli non avvengano collisioni. Il nuovo obiettivo è quello di estendere la GRP e portare i robot in un ambiente con ostacoli garantendo anche l'Obstacle Avoidance ovvero l'assenza di collisioni con gli ostacoli. Per fare questo sono state studiate alcune soluzioni che, come la GRP, rispettano il vincoli di essere distribuite e real-time, fornendo delle soluzioni che non siano ottime in termini di tempo impiegato per compiere il task e/o di distanza percorsa.

Aggiunta di un nuovo stato alla GRP

Una soluzione possibile è quella di aggiungere un nuovo stato all'automa che descrive gli stati della GRP, che consenta di fatto l'aggiramento degli ostacoli. Per fare questo possiamo ipotizzare, utilizzando un approccio simile a quello utilizzato da V.J. Lumelsky e K.R. Harinarayan in [3], che:

1. Il robot sia in grado di distinguere fra un robot ed un ostacolo fisso e fra due robot;

2. Il robot abbia la capacità di muoversi lungo il bordo degli ostacoli fissi;

3. La distanza di rilevamento degli ostacoli da parte dei sensori sia sul bordo del disco riservato;

4. Il centro del disco riservato nelle configurazioni iniziali e finali sia ad una distanza maggiore o uguale al raggio del disco riservato dal bordo degli ostacoli.

Il nuovo stato da aggiungere all'automa di figura 2.4 viene definito `roll on obstacle`. In questo modo è possibile aggirare un ostacolo statico che impedisce il raggiungimento del goal da parte del veicolo. Quando il disco riservato del robot diventa tangente al bordo dell'ostacolo il veicolo è costretto ad entrare prima in `hold` poi a muoversi lungo il perimetro finché lo spazio verso il goal non diventa privo di ostacoli. In figura Fig. 2.5 è mostrato l'automa ibrido con il nuovo stato.

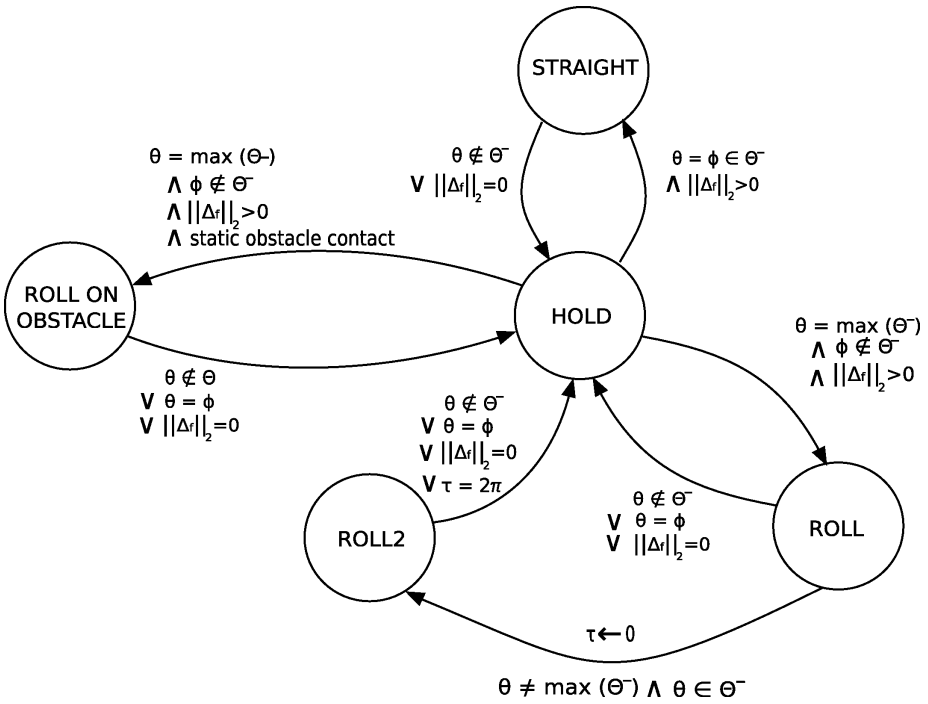


Figura 2.5: Automa ibrido che descrive la GRP con il nuovo stato.

Aggiunta di nuove guardie alla GRP

Analizzando meglio la politica precedente si è visto che in realtà il nuovo stato `roll on obstacle` può essere riprodotto con l'aggiunta di nuove guardie, utilizzando gli stessi stati della GRP originale. Per fare questo, però, dobbiamo formulare delle nuove ipotesi:

1. Il robot è in grado di distinguere fra un robot ed un ostacolo fisso e fra due robot;
2. Il robot conosce le coordinate del punto di contatto fra il disco riservato e l'ostacolo fisso;
3. La distanza di rilevamento degli ostacoli da parte dei sensori sia sul bordo del disco riservato;
4. Il centro del disco riservato nelle configurazioni inizilai e finali sia ad una distanza maggiore o uguale al raggio del disco riservato dal bordo degli ostacoli.
5. Gli ostacoli possono essere approssimati con un poligono, ovvero il loro bordo deve essere esprimibile con una serie di segmenti.

Il nuovo automa ibrido ottenuto è mostrato in figura Fig. 2.6. Con questa politica riproduciamo lo stato `roll on obstacle` utilizzando gli stati già codificati della GRP. In pratica una volta che il robot è venuto in contatto con l'ostacolo cerca di seguirne il perimetro muovendosi lungo la tangente all'ostacolo nel punto di contatto. Se questo movimento gli fa perdere il contatto si ritenta il ripristino attraverso il passaggio allo stato `roll2`. Questo fino a quando non viene rilevato che lo spazio verso il goal è privo di ostacoli. Per fare questo l'agente calcola il cono ammissibile proiettando un agente virtuale dall'altra parte dell'ostacolo in modo che il centro del disco riservato dell'agente virtuale sia a distanza $2\,(R_C + R_s)$ dal centro del proprio disco riservato e che il vettore che congiunge i due centri sia ortogonale con il vettore tangente il profilo dell'ostacolo nel punto di contatto (Fig. 2.7).

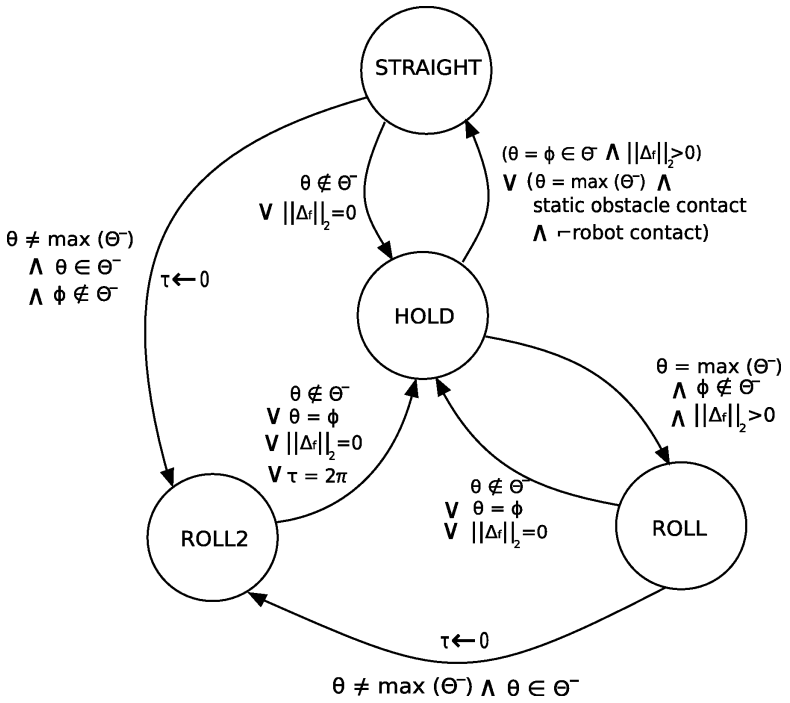


Figura 2.6: Automa ibrido che descrive la GRP con le nuove guardie.

L'agente è in grado di fare queste operazioni poiché abbiamo ipotizzato che conosca le coordinate del punto di contatto fra l'ostacolo ed il proprio disco riservato e quindi sappia in quale direzione proiettare l'agente virtuale. In questo modo, per come è stato pensato il calcolo del cono ammissibile nella GRP, il suo massimo corrisponderà alla tangente al punto di contatto fra il veicolo e l'ostacolo. Quindi se l'orientazione dell'agente lo consente si entrerà nello stato di `straight` essendo verificate le condizioni della guardia a destra dell'or nel passaggio da `hold` a `straight` in figura 2.6. Va notata la necessità nella guardia della condizione sull'assenza di contatto con altri agenti. Infatti, in questo modo, se due agenti verificano le condizioni di vicinanza mentre entrambi sono in contatto con l'ostacolo uno dei due può momentaneamente abbandonare il profilo dell'ostacolo per effettuare il `roll` ed evitare così possibili condizioni di stallo. Raggiunta la fine dell'ostacolo si ha una apertura del cono ammisibile Θ. A questo punto se $\phi \in \Theta^{-}$, ovvero lo spazio fra l'agente ed il goal è privo di ostacoli, potendo raggiungere il goal l'agente

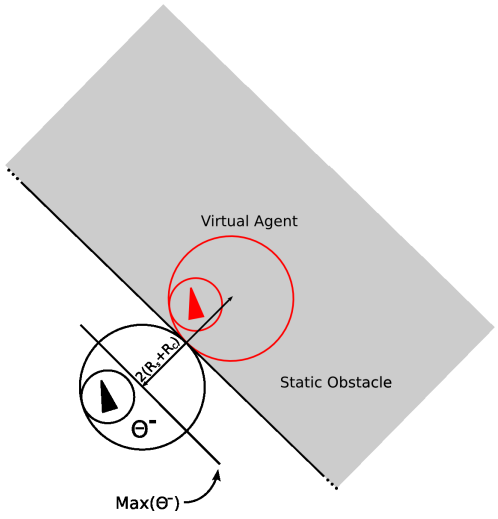
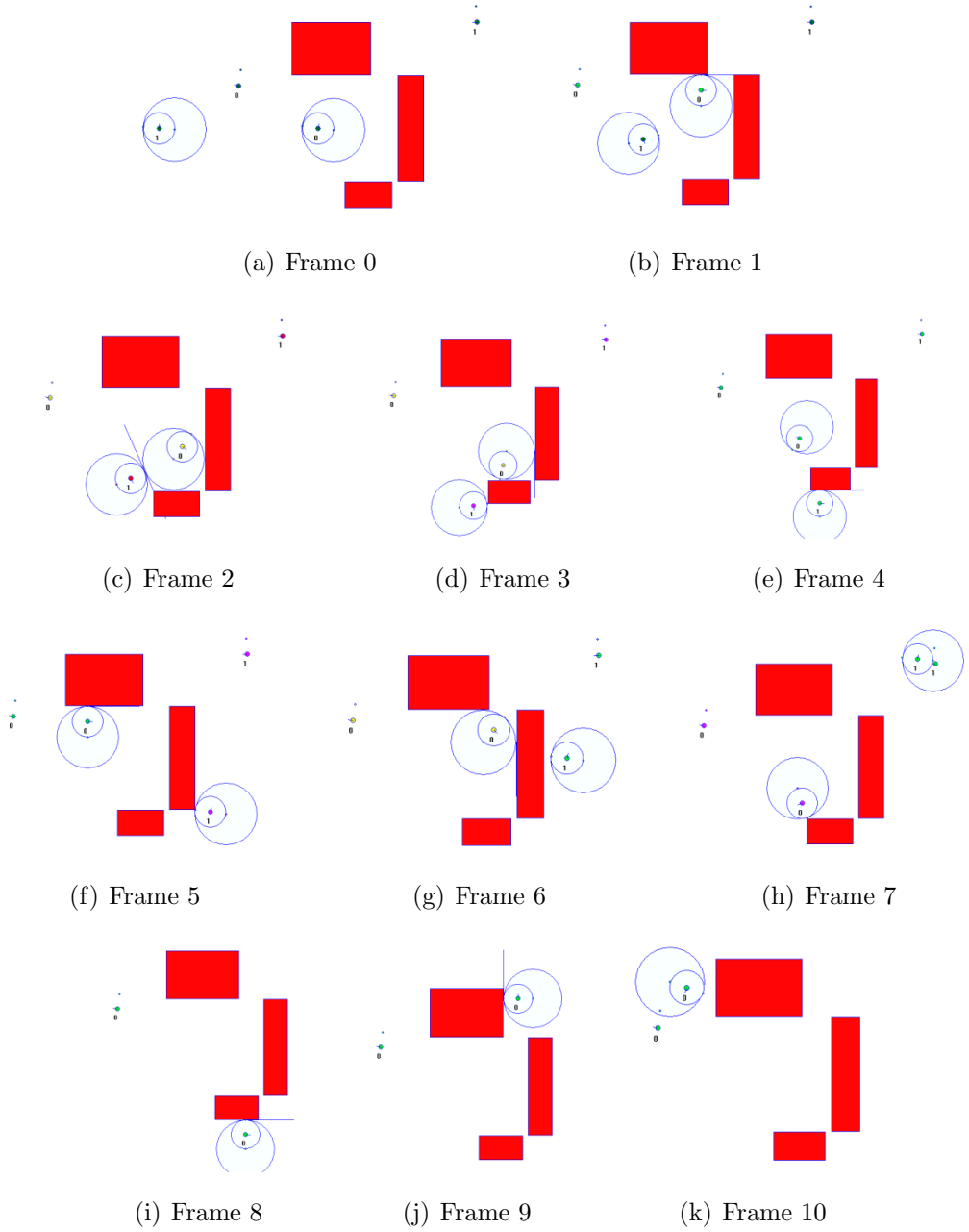


Figura 2.7: Contatto fra una agente ed un'ostacolo statico.

si distacca dall'ostacolo e procede verso il goal attraverso l'uso combinato degli stati `straigth` e `roll`. Se la condizione su ϕ non è verificata l'agente tenterà di ripristinare il contatto dell'ostacolo mediante lo stato `roll2`. Va notato che la condizione di time out τ nel passaggio da `straight` a `roll` non sarebbe strettamente necessaria. Infatti il robot non rischia come nel caso di contatto con i veicoli di effettuare più volte il giro su se stesso. La condizione è stata comunque aggiunta per correttezza rispetto al resto dell'algoritmo dato che la verifica di τ è posta fra le guardie di uscita dallo stato `roll2`. Con questa politica di controllo il veicolo tenta di aggirare l'ostacolo sempre in senso antiorario. Quello che si ottiene è il raggiungimento del goal anche se in maniera non ottima dato che non viene utilizzato il percorso di minor lunghezza. Di seguito (Fig. 2.8) mostriamo, come esempio di applicazione della politica descritta, una simulazione con due agenti e tre ostacoli. I rettangoli rossi rappresentano gli ostacoli statici mentre i cerchi grandi sono i dischi riservati dei due veicoli coinvolti nella simulazione. I cerchi più piccoli rappresentano i dischi di sicurezza e i pallini colorati corrispondo agli agenti. Ogni agente ha un proprio numero distintivo. Per meglio capire in quale stato si trova ogni agente è associato ad ogni stato un colore:giallo e verde



(a) Frame 0 (b) Frame 1

(c) Frame 2 (d) Frame 3 (e) Frame 4

(f) Frame 5 (g) Frame 6 (h) Frame 7

(i) Frame 8 (j) Frame 9 (k) Frame 10

Figura 2.8: Immagini tratte dalla sequenza di una simulazione.

scuro per l'`hold` con o senza un ostacolo rispettivamente, verde chiaro per lo `straight`, rosso per il `roll` e porpora per il `roll2`. In figura 2.8(a) notiamo i veicoli nello stato iniziale. Entrambi si trovano nello stato di `hold`, colore verde scuro, perché la loro orientazione non corrisponde a quella necessaria per raggiungere il goal. Nel tentativo di raggiungere ogniuno il proprio goal i due agenti vengono in contatto; questo porta il veicolo 0 in `straight` lungo l'ostacolo mentre il veicolo 1 è in `hold` nel tentativo di ripristinare l'orientamento che lo porterebbe verso la meta(Fig. 2.8(b)). A questo punto i veicolo vengono di nuovo in contatto. Il veicolo 0 non possibilità di muoversi ed entra in `hold` mentre il veicolo 1 effettua il `roll` (Fig. 2.8(c)). Nelle figure 2.8(d) e 2.8(e) notiamo come l'agente 1 comincia a scivolare lungo l'ostacolo per raggiungere la propria meta e come l'agente 0 cerca di raggiungere il goal utilizzando la via più breve. In figura Fig. 2.8(f) si vede come questo non sia possibile a causa dell'ostacolo e mentre l'agente 1 raggiunge il proprio goal l'agente 0 è costretto a girare intorno a tutti gli ostacoli per raggiungere il proprio(Fig. 2.8(g)-Fig. 2.8(k)).

Integrazione della GRP con l'Artificial Potential Field policy(APF)

Una soluzione alternativa è quella che fonde la Generalized Roundabout Policy con l'Artificial Potential Field policy(APF). Questa soluzione è di più facile realizzazione e può essere applicata nel caso in cui non sia possibile soddisfare le ipotesi necessarie nelle politiche precedenti[2]. Sia la GRP che l'APF sono di tipo decentralizzato e vengono attuate dall'agente mobile stesso. La prima viene utilizzata per evitare le collisioni fra gli agenti mentre la seconda per evitare quelle contro degli ostacoli fissi. La necessità di unire le due politiche, piuttosto che utilizzare una sola politica alternativa, è dovuta al fatto che la sola APF nella sua realizzazione non può garantire l'assenza di urti in presenza di ostacoli mobili.
L'Artificial Potential Field (APF) policy [4] fornisce una efficace pianificazio-

[2]In particolare risulta spesso complesso se non impossibilie discernere con certezza un ostacolo da un agente.

ne del moto per scopi pratici. L'idea su cui si basa questo tipo di pianificatore consiste nel definire un campo di potenziale artificiale nello spazio di lavoro del robot, in modo che esso venga respinto dagli ostacoli e attratto verso il goal. La conoscenza che il robot ha dell'ambiente è locale ed è legata alla capacità di sensing dei sensori. Questo approccio viene realizzato mediante la ricerca del minimo della funzione potenziale, osservando l'andamento decrescente del gradiente. Gli ostacoli sono circondati da campi di potenziale repulsivo, mentre il goal è circondato da una campo di potenziale attrattivo. Di solito il campo di potenziale del goal ha una forma a ciotola, in modo da guidare il robot verso il suo centro se non ci sono ostruzioni. Nel caso vi siano degli impedimenti nell'ambiente dei potenziali a forma di collina sono aggiunti al campo attrattivo, nei punti in cui sono situati gli ostacoli. Il robot subisce una forza uguale al gradiente negativo del potenziale. Questa forza guida il robot finché questo non raggiunge il minimo dell'energia. Il potenziale attrattivo è dato dalla seguente funzione

$$U_{att}(\mathbf{x}) = \begin{cases} k_a \left|\mathbf{x} - \mathbf{x}_d\right|^2 & \text{se } |\mathbf{x} - \mathbf{x}_d| \leq d_a \\ k_a \left(2d_a \left|\mathbf{x} - \mathbf{x}_d\right| - d_a^2\right) & \text{se } |\mathbf{x} - \mathbf{x}_d| > d_a \end{cases} \tag{2.3}$$

dove $\mathbf{x}$ rappresenta il vettore posizione del robot, $\mathbf{x}_d$ il vettore posizione del goal, d_a il raggio del range quadratico e k_a il guadagno proporzionale della funzione. La forza attrattiva F_{att} è il gradiente negativo di questa funzione quindi

$$\mathbf{F}_{att}(\mathbf{x}) = -\nabla U_{att} = \begin{cases} -2k_a \left(\mathbf{x} - \mathbf{x}_d\right) & \text{se } |\mathbf{x} - \mathbf{x}_d| \leq d_a \\ -2d_a k_a \frac{\mathbf{x} - \mathbf{x}_d}{|\mathbf{x} - \mathbf{x}_d|} & \text{se } |\mathbf{x} - \mathbf{x}_d| > d_a \end{cases}. \tag{2.4}$$

Per il potenziale repulsivo abbiamo

$$U_{rep}(\mathbf{x}) = \begin{cases} \frac{1}{2} k_r \left(\frac{1}{\rho} - \frac{1}{\rho_0}\right)^2 & \text{se } \rho \leq \rho_0 \\ 0 & \text{se } \rho > \rho_0 \end{cases}, \tag{2.5}$$

dove ρ_0 rappresenta la distanza limite di influenza del potenziale repulsivo e ρ la distanza dall'ostacolo. La scelta di ρ_0 dipende dalla velocità massima

del robot e dal periodo di campionamento del controllo. La forza repulsiva è data da

$$\mathbf{F}_{rep}(\mathbf{x}) = -\nabla U_{rep} = \begin{cases} k_r \left(\frac{1}{\rho} - \frac{1}{\rho_0}\right) \frac{1}{\rho^2} \frac{\partial \rho}{\partial \mathbf{x}} & \text{se } \rho \leq \rho_0 \\ 0 & \text{se } \rho > \rho_0 \end{cases}, \tag{2.6}$$

dove $\partial\rho/\partial x$ può essere rappresentato come

$$\frac{\partial \rho}{\partial \mathbf{x}} = \left(\frac{\partial \rho}{\partial x} \frac{\partial \rho}{\partial y}\right)^T = \frac{\mathbf{x} - \mathbf{x}_0}{\rho} \tag{2.7}$$

con $\mathbf{x}_0$ che rappresenta il vettore posizione dell'ostacolo più vicino in un sistema di coordinate cartesiano.
Il potenziale totale si ottiene attraverso la somma del potenziale attrattivo con i potenziali repulsivi, da cui si ricava la forza totale $\mathbf{F}(\mathbf{x}) = \mathbf{F}_{att} + \mathbf{F}_{rep}$.
A questo punto è possibile capire perché questa politica non riesce da sola a garantire l'assenza di collisioni fra due agenti in movimento. Infatti la velocità di un agente può essere scritta come $\mathbf{v} = R\overline{\omega} + \mathbf{v}_l$ dove $\mathbf{v}_l$ è la velocità lineare, ω la velocità angolare e R il raggio di curvatura del veicolo. Poiché

$$\begin{cases} \omega \propto \left(\frac{1}{\rho} - \frac{1}{\rho_0}\right) & \text{se } \rho \leq \rho_0 \\ \omega = 0 & \text{se } \rho > \rho_0 \end{cases}$$

se la velocità con cui due agenti si stanno venendo incontro è più alta della velocità di variazione di ρ, che è legata al periodo di campionamento del controllo, la componente legata alla velocità angolare non è sufficiente per evitare la collisione.
La politica del potenziale artificiale anche se di facile attuazione presenta il problema dei minimi locali. Infatti, in funzione di come sono disposti gli ostacoli nell'ambiente, può accadere che il robot rimanga intrappolato in un minimo locale senza più riuscire a raggiungere il goal. Il problema è ben conosciuto ed in letteratura esistono diverse soluzioni per superarlo. Si vedano ad esempio [5], [6], [7].
Il Collision-Obstacle avoidance è realizzato mediante la convivenza delle due

politiche che si occupano ciascuno di un aspetto specifico nella pianificazione del moto del robot. La GRP si occupa di evitare altri robot, sia fermi sia in movimento, mentre l'APF evita ostacoli statici generici. In figura 2.9 è mostrato il diagramma di flusso dell'algoritmo risultante. All'arrivo di

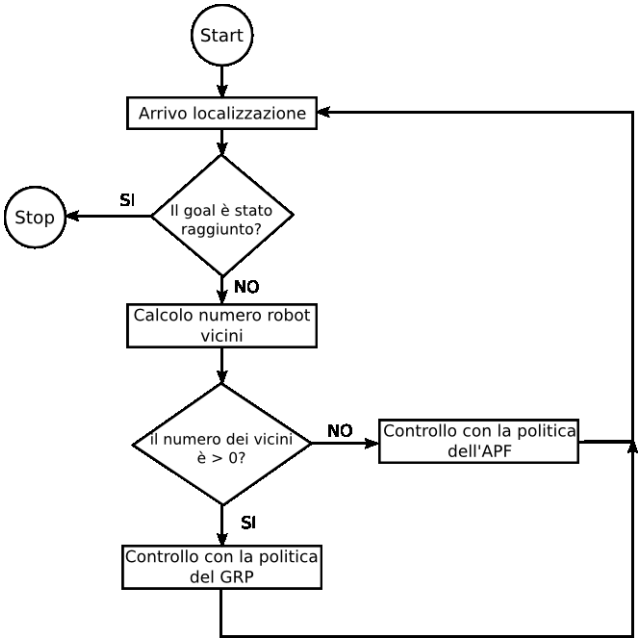


Figura 2.9: Diagramma di flusso dell'algoritmo di integrazione fra GRP e APF.

una nuova localizzazione, sia del robot corrente che di altri robot presenti nell'ambiente, se non è stato raggiunto il goal, si controlla che non ci siano condizioni di tangenza fra il disco riservato dell'agente ed altri dischi riservati. Se non ve ne sono il controllo del veicolo viene eseguito applicando l'APF altrimenti la GRP. Quest'ultima, nel pianificare il moto del veicolo, terrà conto delle informazioni provenienti dai sensori per poter evitare eventuali ostacoli fissi presenti nelle vicinanze del robot.
Un inconveniente di questa soluzione è la difficoltà, a causa della lontananza e diversità dei due approcci, di rendere la politica risultante più omogenea e cooperativa.

2.3 Confronto e scelta

Nella scelta di quale politica utilizzare in questo lavoro si è optato per quella che fonde GRP e APF. L'estensione della GRP offre un modo omogeneo di trattare gli ostacoli, siano essi agenti o barriere di altra natura, ma le ipotesi che il robot sappia distinguere fra un ostacolo ed un altro robot e che sappia individuare esattamente il punto di contatto del disco riservato con l'ostacolo sono difficilmente implementabili su una piataforma hardware low cost quale quella presentata in questo lavoro. L'integrazione fra GRP e APF vince per la sua semplicità di realizzazione anche se non vi è alcun tipo di collaborazione fra la due politiche che la compongono e, di fatto, il veicolo viene gestito con la politica del potenziale artificiale. Infatti l'APF causa l'allontanamento degli agenti prima che i dischi riservato vengano in contatto. Solo se questo non avviene interviene la GRP per assicurare l'assenza delle collisioni.

Capitolo 3

Middleware e Componenti

In questo capitolo viene affrontato l'argomento del Middleware; quale è il suo scopo, quali sono attualmente le tecnologie esistenti e quali sono i requisiti necessari per il funzionamento su sistemi embedded.

3.1 Problema e Motivazioni

La necessità di unire applicazioni diverse in un unico quadro lavorativo al fine di sfruttarne al massimo le potenzialità, senza conoscerne obbligatoriamente tutti i dettagli, ha portato allo sviluppo di uno strato software aggiuntivo, detto Middleware, che consente di ottenere un elevato livello di servizio per gli utenti, ed un elevato livello di astrazione per i programmatori. Inoltre, consente di facilitare la manutenzione, la stesura e l'integrazione di nuove applicazioni. Il Middleware si pone fra il sistema operativo e l'applicazione. Esistono quattro tipi principali di Middleware: transazionale, orientato ai messaggi, procedurale e a componenti. In questa sede ci occuperemo solo del middleware a componenti.
Un componente è un elemento di sistema che offre un servizio predefinito ed è in grado di comunicare con gli altri componenti. É una unità con sviluppo e versioning indipendente. Viene incapsulato e non è legato a nessun contesto. I componenti possono interagire gli uni con gli altri senza conoscere

nulla della loro rispettiva struttura interna e dell'ambiente in cui vengono eseguiti (per esempio, il sistema operativo e i protocolli di rete). L'unica conoscenza necessaria è relativa alla struttura dell'interfaccia verso l'esterno. Tale proprietà consente di fatto la sostituzione di un componente basato su una tecnologia con un altro che usi una tecnologia diversa, purché questi implementino la stessa interfaccia, senza che ci siano ripercussioni sul resto del sistema. Il Middleware media le interazioni di un componente con il sistema fornendo una interfaccia di programmazione trasparente.
Con questa tecnica di programmazione si ottiene:

La riutilizzabilità del codice: Un nuovo software può essere generato da componenti già esistenti e testati.

La portabilità del codice: grazie alla mediazione del Middleware i componenti posso essere utilizzati su macchine differenti senza la necessità di modifiche negli altri componenti.

La velocizzazione nella realizzazione delle applicazioni. Più componenti possono essere sviluppati contemporaneamente in parallelo.

La semplificazione nello sviluppo delle applicazioni: il Middleware è in grado di gestire una serie di errori in modo che né il programmatore né l'applicazione ne debbano tener conto.

Utilizzare il Middleware significa anche che gli sviluppatori hanno più software da dover acquistare, installare, manutenere ed imparare; il codice generato dai sistemi middleware potrebbe non essere efficiente quanto il codice scritto da un programmatore; i costi, in termini di uso delle risorse messe a disposizione dalla macchina, che deve gestire anche l'interazione fra i diversi componenti, sono maggiori. Infatti l'esecuzione del middleware introduce un *overhaed* che comporta un maggiore sfruttamento della CPU e un maggiore consumo di memoria, quindi un degrado delle prestazioni della macchina. Nei sistemi embedded tempo-reale, provvisti di processori con bassa capacità di calcolo e di poca memoria, il Middleware attualmente in commercio risulta

inadeguato perché non riesce a garantire i vincoli temporali, nelle esecuzioni dei programmi, e di spazio, nel consumo di memoria.

3.2 Panoramica

3.2.1 Middleware a componenti

Importanti esempi di middleware a componenti correntemente in uso sono Java Remote Method Invocation (Java RMI), Microsoft Component Object Model (COM) e Common Object Request Broker Architecture (CORBA). Purtroppo queste ambienti non sono finalizzati ai sistemi embedded o ai sistemi di controllo distribuito. I vincoli sulle risorse disponibili, tipici di questi sistemi, implicano severi requisiti addizionali sul Middleware. Per fare questo sono state sviluppate delle estensioni al Middleware generico. Un esempio è il real-time CORBA [8], che è caratterizzato da politiche di scheduling prioritarie per i threads ed esporta alcuni parametri di controllo nei protocolli di comunicazione. Il problema nell'utilizzo del real-time CORBA è che la sua realizzazione è specifica per sistemi embedded statici collegati fra loro via cavo e quindi non è possibile gestire sistemi embedded eterogenei che comunicano mediante reti *wireless*.
Diversi domini di applicazione hanno accentuato l'importanza di sviluppare infrastrutture software specifiche per il proprio dominio. Per esempio, l'industria automobilistica ha costituito la partnership di sviluppo AUTOSAR, per avere sui veicoli del software che sia modulare, scalabile, trasferibile e riusabile. L'impegno di AUTOSAR è quello di fornire un'architettura *open system* per il sistema automobile basato su interfacce standard per i differenti livelli del sistema stesso.
Middleware specifico per il controllo e per i requisiti real-time è stato di recente studiato anche in ambito accademico:

Etherware è un middleware per il controllo distribuito che punta sulla capacità di mantenere i canali di comunicazione durante il riavvio

e l'aggiornamento dei componenti, e sul recupero dalle situazioni di guasto[9];

Controlware è un middleware che utilizza un controllo in retroazione per garantire le performance nei sistemi software[9].

Anche queste tecnologie presentano degli aspetti che non le rendono idonee all'utilizzo con i sistemi embedded.
Etherware è implementato in Java e i componenti comunicano mediante documenti XML. Questa scelta, se da un lato ne aumenta la portabilità dall'altro ne limita l'uso a sistemi dotati di elevate disponibilità in termini di risorse.
Il Controlware presenta delle limitazioni sulla capacità di catturare il comportamento essenziale del sistema controllato. La modellizzazione effettuata non tiene conto dell'informazione sull'intervallo dei parametri per cui il modello può essere considerato una buona approssimazione del sistema.

3.2.2 Componenti

L'ingegneria del software basata sui componenti è stata utilizzata con successo in diversi progetti principalmente per applicazioni da ufficio e di eBusiness. Per le applicazioni da ufficio la tecnologia COM è la più largamente usata. I componenti COM spesso non hanno una dimensione contenuta dato che ogni componente comprende una sostanziale quantità di funzionalità dell'applicazione. Altre classi di modelli a componenti largamente diffuse sono quelle che hanno la loro base sui modelli ad oggetti distribuiti. Queste includono il CORBA Component Model (CCM), gli Enterprise Java Beans (EJB) e .NET. Il modello .NET può essere visto come una evoluzione distribuita di COM che utilizza il Common Language Runtime (CLR). Il CLR è una macchina virtuale che può essere paragonata a quella del Java. Questa è l'implementazione Microsoft del Common Language Infrastructure (CLI) standard, che definisce un ambiente di esecuzione per il codice del programma.
Nelle tecnologie a componenti per sistemi embedded, proprietà non funzio-

nali quali sicurezza, reattività, gestione della memoria e affidabilità sono di particolare interesse. Confrontati con i modelli a componenti descritti sopra, questi sono molto più limitati nelle funzionalità. Spesso i modelli a componenti sono progettati per applicazioni di natura algoritmica. Individualmente i componenti sono tipicamente più piccoli di quelli per applicazioni non embedded e maggior risalto è dato alla loro aggregazione. Non ci sono al momento dei buoni esempi di tecnologie a componenti commerciali per i sistemi embedded. Tuttavia, è un campo dove la ricerca sta investendo considerevoli sforzi [10].
Una soluzione che ha permesso di implementare un modello a componenti su un sistema embedded quale quello utilizzato per i robot si è ottenuta attraverso il progetto di ricerca europeo RUNES (Reconfigurable Ubiquitous Networked Embedded Systems, vedi figura 3.1), in cui questo lavoro si innesta, che ha cercato di colmare questa lacuna.
Il progetto RUNES si è occupato della fusione fra reti e sistemi embedded con il fine di applicare quest'ultimi in ambienti molto più complessi di quelli in cui oggi sono utilizzati. Per poter gestire questa complessità, RUNES

Figura 3.1: Logo RUNES.

ha proposto sistemi middleware scalabili e mezzi per lo sviluppo di applicazioni che permettano agli utilizzatori, ai programmatori e ai progettisti la flessibilità di interagire con un ambiente più dettagliato solo dove necessario. L'architettura proposta in questo ambito si è prefissata di:

- Costruire sistemi middleware che siano adattivi ed auto-organizzanti;

- Assicurare che il middleware sia robusto e prevedibile abbastanza da rendere invisibili le elaborazioni;
- Fornire gli strumenti sia per una stima automatica dell'usabilità che per il debug.

Questo per avere un middleware che fornisca il giusto livello di astrazione, necessario per il controllo delle applicazioni, con una architettura leggera e scalabile. Inoltre deve essere in grado di fornire supporto ad una estesa varietà di applicazioni di controllo, che vanno dalle reti di sensori ai sistemi di controllo distribuito. Il middleware di RUNES fa da collante fra sensori, attuatori, gateway, dispositivi di routing, sistemi operativi, stack di rete e applicazioni. Definisce degli standard per implementare interfacce software e funzionalità che permettano lo sviluppo di un codice ben definito e riusabile. L'elemento fondamentale del middleware di RUNES è il *componente software*. Da un punto di vista astratto un *componente* è un modulo software autonomo con ben definite funzionalità che può interagire con altri componenti solo attraverso *interfacce* e *ricettacoli*.
É stato dimostrato che questa tecnologia lavora bene con il sistema operativo Contiki, che è stato sviluppato per dispositivi con poca memoria e bassa capacità di calcolo[11]. L'implementazione del modello a componenti in Contiki è conosciuto come Component RunTime Kernel (CRTK).

3.3 Contiki OS

Contiki è un sistema operativo per sistemi embedded multi-tasking, ad alta portabilità, connesso in rete. Diversamente da molti altri sistemi operativi embedded, che per aggiornare il codice richiedono che sia ricostruita e scaricata sul dispositivo l'immagine binaria completa dell'intero sistema (l'applicazione, il sistema operativo, le librerie), Contiki fornisce la possibilità di caricare e scaricare dinamicamente a run-time moduli e programmi. Poiché, in molti casi, un'applicazione è molto più piccola dell'intero sistema si ha un

minore consumo di energia quando viene trasmessa attraverso la rete ed un minore tempo di trasferimento.
A causa dell'eterogeneità dei sistemi embedded, per poterne aumentare la portabilità, Contiki è stato realizzato in modo che l'unica astrazione fornita sia il multiplexing della CPU e il supporto per il caricamento dei programmi e dei servizi. Altre astrazioni vengono fornite come librerie in modo da avere una gestione dinamica dei servizi.
Contiki è basato su un kernel *event-drive* dove il preemptive multi-threading è implementato come una libreria dell'applicazione che può essere collegata ai programmi se esplicitamente richiesto, per sfruttare i vantaggi sia dei sistemi *event-drive* che dei *preemptible threads*.
In ambienti con memoria limitata, il modello a multithread spesso consuma larga parte delle risorse di memoria. Ogni thread deve avere il proprio stack e poiché è difficile sapere in anticipo quanto spazio nello stack un thread userà, questo tipicamente deve essere sovradimensionato. Inoltre la memoria per ogni stack deve essere allocata quando il thread viene creato e non può essere condivisa fra thread concorrenti, ma può essere usata solo da quello per cui è stata allocata. Infine il modello a thread concorrenti deve prevedere dei meccanismi di locking delle risorse condivise. Quindi per avere concorrenza senza l'utilizzo di meccanismi di locking e di uno stack per processo si è utilizzato un sistema *event-drive*. In questo modo i processi sono implementati come gestori di eventi che vengono eseguiti fino a completamento. Poiché un gestore di evento non può essere bloccato, tutti i processi possono usare lo stesso stack, condividendo effettivamente le scarse risorse di memoria. I meccanismi di locking non sono più necessari perché due gestori di eventi non possono mai girare in concorrenza l'uno con l'altro. Un modello di programmazione guidato dagli stati può essere difficile da gestire per i programmatori oltre al fatto che non tutti i programmi possono essere facilmente espressi come macchine a stati. Un esempio sono le elaborazioni necessarie per la crittografia che richiedono generalmente diversi secondi per essere completate da una CPU su piattaforma con poche risorse. In un sistema operativo puramen-

te *event-drive* elaborazioni lunghe comportano una monopolizzazione della CPU che rende il sistema incapace a rispondere ad eventi esterni. Nei sistemi operativi basati su *preemtive multi-threading* questo tipo di inconveniente non sussiste in quanto le elaborazioni lunghe possono essere prelazionate.
Un sistema in cui sta girando Contiki è costituito dal kernel, dalle librerie, dal caricatore di programma e da un insieme di processi. Un processo può essere o una applicazione o un servizio. Un servizio implementa funzionalità utilizzate da più di una applicazione. Tutti i processi possono essere dinamicamente rimpiazzati a tempo di esecuzione. La comunicazione fra processi passa sempre attraverso il kernel che non fornisce uno strato di astrazione hardware, ma lascia che i driver del dispositivo e le applicazioni comunichino direttamente con l'hardware. Un processo è definito da una funzione *event-handler* e da una *poll-handler*. Lo stato del processo viene tenuto nella memoria privata del processo e il kernel mantiene solo un puntatore a questo stato. Tutti i processi condividono lo stesso spazio di indirizzamento e la comunicazione avviene attraverso l'invio di eventi. Un sistema con Contiki

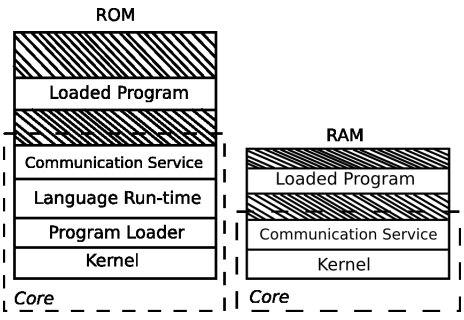


Figura 3.2: Partizionamento in nucleo e programmi caricati.

e diviso in due parti: il nucleo e i programmi caricati (Fig.3.2). Il partizionamento viene fatto a tempo di compilazione ed è specifico del contesto in cui Contiki è usato. Tipicamente il nucleo è costituito dal kernel di Contiki, il caricatore dei programmi, le librerie di supporto e uno stack di comunicazione con i drivers per l'hardware di comunicazione. Il nucleo è compilato in una singola immagine binaria e caricato sul dispositivo prima dell'entrata in funzione. Generalmente il nucleo non è più modificato dopo che è diventato

operativo, ma si può notare che è possibile usare uno speciale boot loader per sovrascriverlo. I programmi sono caricati nel sistema dal caricatore di programma. Questo può ottenere il codice binario o attraverso lo stack di comunicazione o attraverso i dispositivi di archiviazione quali la EEPROM. Infatti per essere caricati i programmi devono essere prima salvati nella EEPROM. Maggiori dettagli su Contiki si possono trovare in [11].
Contiki gira su una varietà di sistemi che vanno dai micro controllori quali MSP430 e AVR fino ai vecchi home-computers. Lo spazio di memoria occupato dal codice è dell'ordine dei kilobytes e l'uso della memoria può essere configurato per essere abbassato fino alla decina di bytes. Contiki supporta il protocollo TCP/IP attraverso lo stack uIP TCP/IP embedded.

uIP Lo stack open source uIP TCP/IP fornisce la connettività ai microcontrollori ad 8 bit pur mantenendo la interoperabilità e la conformità agli standard RFC[1].
uIP è una implementazione dello stack del protocollo TCP/IP per micro controllori a 8 o a 16 bit. Questo fornisce i protocolli necessari per la comunicazione attraverso internet con poca occupazione di memoria da parte del codice e poca richiesta di RAM. La dimensione del codice è di pochi kilobytes e l'occupazione della RAM dell'ordine delle centinaia di bytes.
uIP è software open source scritto in C con le seguenti caratteristiche:

- Codice sorgente ben documentato e commentato;
- Dimensione molto piccola del codice;
- Bassa occupazione della RAM, configurabile a tempo di compilazione;
- Supporto dei protocolli ARP, SLIP, IP, UDP, ICMP(ping) e TCP;

[1]Gli RFC (Requests For Comments) sono una collezione di documenti che descrivono varie pratiche, già in uso o suggerite, rilevanti per Internet. Molti RFC riguardano i *protocolli*, disposizioni e convenzioni tecniche che specificano come deve avvenire lo scambio di dati fra sistemi.

- Inclusione di un insieme di applicazioni d'esempio;
- Un certo numero di connessioni TCP concorrenti attive, il cui massimo può essere stabilito a tempo di compilazione;
- Un certo numero di connessioni TCP passivamente in ascolto, il cui massimo può essere stabilito a tempo di compilazione;
- RFC conforme alle implementazioni del protocollo TCP e IP, con l'inclusione del controllo di flusso, il riassemblamento dei frammenti e la ritrasmissione della stima del time-out.

3.3.1 Component RunTime Kernel (CRTK)

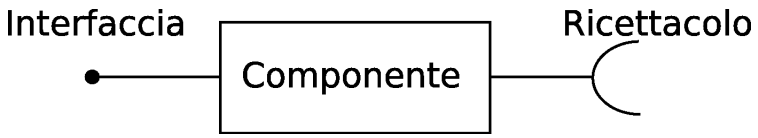


Figura 3.3: Schematizzazione di un componente.

Un componente è un'entità che implementa alcune specifiche funzionalità. Può offrire dei servizi attraverso un'**interfaccia**, che non è altro che una lista di prototipi di funzione, e ne può ricevere mediante un **ricettacolo** (Fig 3.3).
Il componente deve essere dichiarato e definito, possiede costruttori e distruttori ed implementa delle funzioni che possono essere utilizzate al suo interno ed altre che vengono fornite ad altri componenti mediante le interfacce. Solitamente al suo interno vengono avviati dei processi, che sono entità di Contiki e possono gestire timers, aspettare eventi e inviare messaggi ad altri processi. Di seguito è mostrata la sintassi per dichiarare un componente:

```
// component_name.h

#include "crtk.h"
```

```
DECLARE_COMPONENT( component_name ,
{
    void (* function_A)(< arguments list >);
    void (* function_B)(< arguments list >);
    void (* function_C)(< arguments list >);
    ...
} ) ;
```

e la sua definizione:

```
// component_name.c

#include "component_name.h"
...

static void construct(void) {
printf ("Component istantiated ...\n");
// SOME ACTIONS
}

static void destroy(void) {
printf ("Component is destroyed ...\n");
}
static int a;
static unsigned char * b;

static void function_A (< arguments list>){
printf ("function_A implementation \n") ;
}

static void function_B (< arguments list>){
printf ("function_B implementation \n");
```

```
}
IMPLEMENT_COMPONENT(component_name,{function_A,
                    function_B});}
```

É importante sottolineare che la definizione del costruttore e del distruttore è necessaria, mentre quella delle funzioni che faranno parte dell'interfaccia del componente è facoltativa. Attraverso queste funzioni è possibile avviare processi e implementare le funzionalità richieste.
Per quanto riguarda le regole di visibilità va detto che le variabili definite fuori delle funzioni hanno visibilità globale e livello di componente, mentre le variabili definite dentro le funzioni hanno visibilità locale.
Un'interfaccia è una lista di funzioni che saranno implementate da una componente. La stessa interfaccia può essere implementata da diversi componenti in modi differenti dato che contiene solo le dichiarazioni delle funzioni. La sintassi per implementare un'interfaccia è la seguente:

```
// interface_name.h

#include "crtk.h"

DECLARE_INTERFACE( interface_name,}
{
    void (* function_A)(< argument list >);
    void (* function_B)(< argument list >);
    ...
})
```

Un ricettacolo è un link fra il componente e l'interfaccia utilizzata dal componente. Permette di specificare quale componete si sta utilizzando per implementare l'interfaccia. Cambiando il componente che implementa le funzione si realizza il rebinding dinamico fra i componenti. Infatti quando un componente A fornisce una funzionalità mediante la sua interfaccia ad un

componente B che la usa attraverso il ricettacolo si dice che avviene il binding fra i due componenti. É riportata di seguito la sintassi per la definizione del ricettacolo ed il binding.

```
// component_A.c

static Receptacle receptacle_name;

static void construct(void) {
    ...
    INVOKE(''crtk'',crtk,connect(''component_B'',
           &receptacle_name));
    ...
}
```

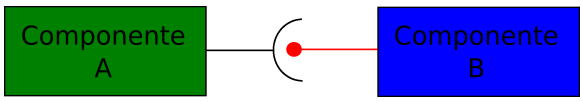


Figura 3.4: Binding fra il componente A ed il componente B attraverso la connessione del ricettacolo di A con l'interfaccia di B.

In figura 3.4 è mostrato graficamente il binding fra un componente A ed un componente B. I servizi messi a disposizione dal componenti B mediante la sua interfaccia sono ora a disposizione del componente A che li può utilizzare attraverso il ricettacolo, come si può vedere in figura 3.5

```
// component_A.c
...
INVOKE(receptacle_name,interface_name,
       function_A(<argument list>));
...
```

Attraverso il rebind dinamico e possibile far sì che il servizio fornito da un componente possa essere fornito da un'altro effettuando la commutazione a runtime (Fig. 3.6).

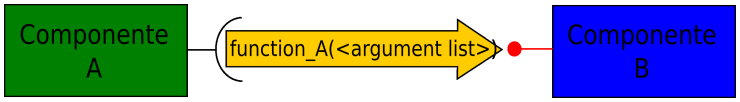


Figura 3.5: Richiesta di un servizio del componente A al componente B mediante chiamata alla funzione di interfaccia di B.

```
// component_A.c

...

if (<condition>) {
    INVOKE("crtk",crtk,connect("component_B",\&rec_A));
}
else {
    INVOKE("crtk",crtk,connect("component_C",\&rec_A));
}
...
```

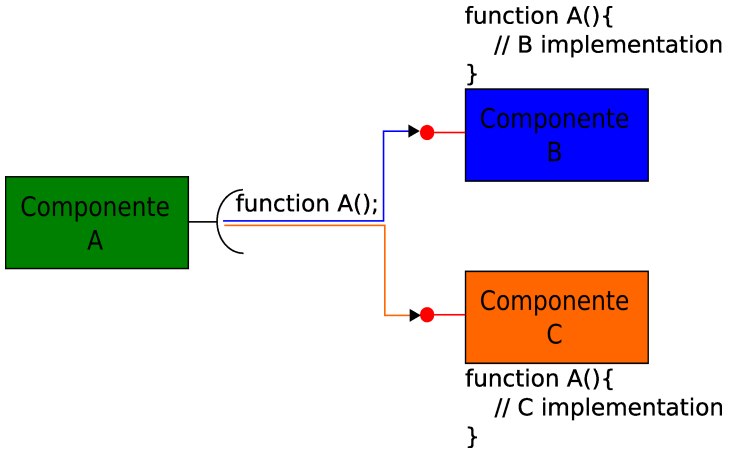


Figura 3.6: Esempio Rebinding dinamico; il componente A effettua una chiamata di funzione, attraverso il ricettacolo, che viene eseguita dal componente B o C in dipendenza del fatto che siano verificate o meno certe condizioni.

Capitolo 4

Descrizione dell'agente mobile

La piattaforma sviluppata in questo lavoro ha come base quella descritta in [12]. In figura 4.1 è mostrata l'architettura software per la gestione decentralizzata del traffico di robot mobili. Per facilitare la comprensione del resto del capitolo riportiamo di seguito una breve descrizione dei componenti di base e delle loro funzionalità.

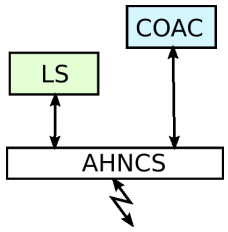


Figura 4.1: Moduli Software

Il Localization System (LS), di cui parleremo nel capitolo 5, fornisce le posizioni di tutti gli agenti nell'ambiente di lavoro. Il Collision-Obstacle Avoidance Component (COAC) è responsabile del coordinamento dei robot in una area con presenza di ostacoli fissi di forma generica. La localizzazione avviene mediante un visual server che invia i propri dati all'agente attraverso l' Ad-Hoc Networking Comunication Service (AHNCS). Queste informazioni vengono raccolte dal robot che le invia al componente COAC. Quest'ultimo all'arrivo di una nuova localizzazione valuta la politica di evitamento delle

collisioni. Uno schema che mostra la realizzazione di quanto detto è illustrato in figura 4.2. In questo caso le informazioni sulla localizzazione vengono raccolte da una scheda TMote Sky, di cui si parlerà più approfonditamente in seguito, che le invia al componente COAC. La TMote riceve la risposta riguardo alle azioni da intraprendere dal COAC ed invia i riferimenti da seguire al controllore dinamico del veicolo. In questo capitolo verrà presentata

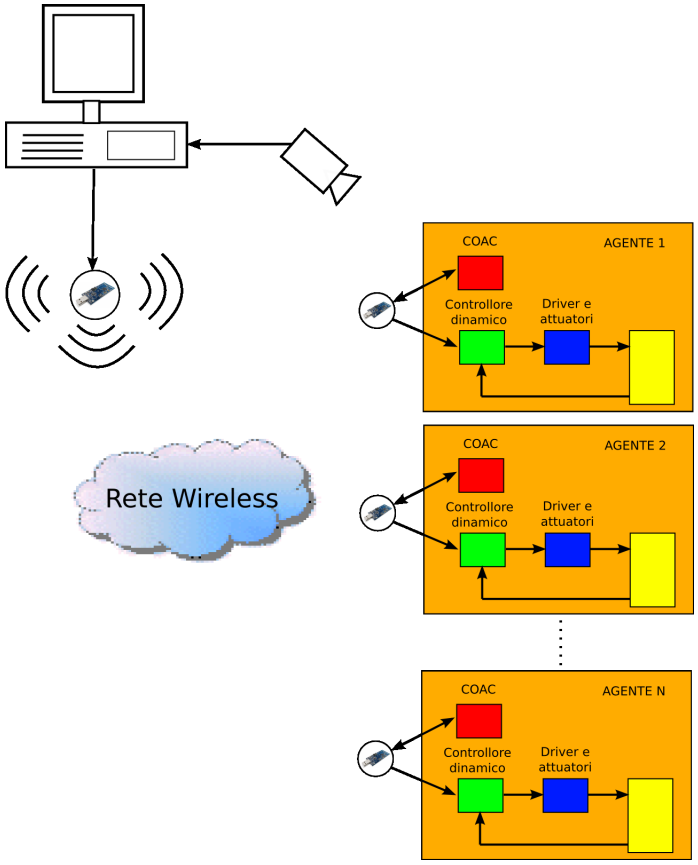


Figura 4.2: Schema Infrastruttura.

la piattaforma robotica sviluppata sotto il profilo hardware e software.

4.1 Prototipo dell'agente: RAiN 2

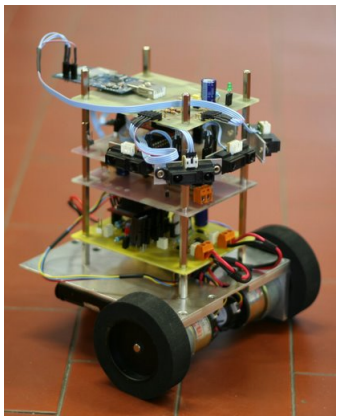


Figura 4.3: RAiN 2

Il robot è costituito da uno chassis di $14cm \times 13cm \times 8cm$ che supporta i motori, le batterie e l'elettronica. A causa dell'utilizzo di tecnologia a basso costo, per la localizzazione, l'agente utilizza un sistema di visione esterno fornito da un server. Cinque sensori ad infrarossi, montanti ai lati del robot fungono da rilevatori di ostacoli. Il cuore dell'elettronica di controllo è rappresentato da una scheda Tmote Sky che fornisce l'elemento di calcolo principale, oltre a permettere le comunicazioni wireless con il protocollo 802.15.4, comunemente noto come Zigbee. La Tmote-Sky è stata utilizzata per la sua compatibilità con questo protocollo e per i bassi consumi. Sono inoltre presenti nell'agente alcuni controllori Cypress PSoC che, come esplicitamente descritto nel resto della trattazione, implementano servizi aggiuntivi.

4.1.1 TMote-Sky

L'insieme degli agenti mobili che condividono lo stesso spazio operativo e che scambiano informazioni con l'esterno possono essere visti come i nodi di una rete di sensori wireless con topologia variabile. La gestione di una rete per un controllo multiagente non è banale dato che i nodi hanno risorse energetiche limitate e qualità dei canali di comunicazione bassa. Inoltre, l'energia dispo-

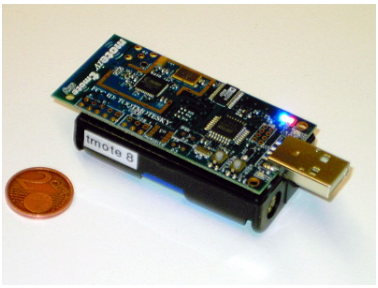


Figura 4.4: Tmote sky

nibile è utilizzata non solo per lo scambio dei dati, ma anche per il calcolo del controllo cinematico e dinamico oltre che per l'attuazione dei motori. La tecnologia converge quindi verso nodi di elaborazione estremamente miniaturizzati, che integrino la totalità delle funzionalità *on-chip*. Questo porta a considerare capacità di elaborazione e memorie di dimensioni molto ridotte. A causa di questi motivi le reti di sensori pongono problematiche nuove per quanto riguarda la progettazione del software. Questo avviene sia a livello di software di sistema, sia per quanto riguarda la progettazione dell'intero sistema distribuito. Per l'implementazione di una piattaforma hardware-software per il test di algoritmi di controllo multiagente sono stati scelti degli *smart sensors*, ovvero dei sensori wireless di nome Mote. In particolare si è fatto uso dei sensori TMote Sky la cui immagine è riportata in figura 4.4, sviluppata dalla MoteIv Corporation, società consolidata nello sviluppo di piattaforme wireless. Le TMote Sky sono sensori di piccola dimensione e presentano caratteristiche utilissime per realizzare uno schema di controllo decentralizzato in quanto mettono a disposizione:

- Dimensioni di H×W×P = 1.26×2.56×0.24 in.;
- Velocità di trasmissione dati di 250 kbps @ 2,4GHz sfruttando il protocollo IEEE 802.15.4 e un tranceiver Wireless;
- Un facile scambio di informazioni con altri dispositivi che supportano gli stessi standard;

- Un microprocessore della Texas Instruments con velocità di clock di 8MHz con una memoria RAM di 10K e una memoria Flash di 40K;
- Un'antenna integrata sulla scheda, con un raggio di azione di 50m in ambienti chiusi e di 125m in ambienti all'aperto;
- L'utilizzo dello standard IEEE 802.15.4 per ottenere un bassissimo consumo di corrente;
- Un rapido risveglio dallo stato di basso consumo $Sleep(< 6\mu s)$
- Una facile programmabilità attraverso una comune porta USB del PC;
- Un supporto per i più diffusi sistemi operativi sviluppati per la tecnologia Mote.

Oltre alle caratteristiche tecniche utili al progetto, tali dispositivi permettono di abbattere i costi dato che gli schematici e tutto l'occorrente per la loro costruzione fisica sono di dominio pubblico e facilmente reperibili sulla rete. Possono quindi essere realizzati autonomamente in laboratorio.

Protocollo Zigbee

ZigBee è il nome di una specifica per un insieme di protocolli di comunicazione ad alto livello che utilizzano piccole radio digitali a bassa potenza e basato sullo standard IEEE 802.15.4 per wireless personal area networks (WPANs). Il protocollo 802.15.4 consente ridotti consumi, in quanto i dispositivi possono essere disattivati temporaneamente, ha un basso costo e consente di ottenere un'alta densità di nodi per rete. Questo protocollo è pensato per reti a bassa velocità con scambio di informazione costituito da dispositivi alimentati tramite batterie che non possono essere sostituite frequentemente, come nelle reti di sensori. Queste caratteristiche lo rendono idoneo per applicazioni distribuite di misura e controllo. Nonostante l'esistenza di numerosi protocolli wireless, *Zigbee* è quello che più si adatta all'applicazione progettata. Infatti tecnologie largamente diffuse come Bluetooth non sono

state studiate per applicazioni di controllo decentralizzato dove l'energia e le risorse sono limitatissime. Nonostante i dispositivi hardware che supportano il protocollo 802.15.4 siano molto simili ai dispositivi che supportano il Bluetooth, la potenza richiesta nelle trasmissioni è molto diversa proprio a causa del diverso protocollo di trasmissione adottato. Si consideri che un dispositivo che fa uso della tecnologia *Zigbee*, in *sleep mode* assorbe solamente circa 1μA di corrente, a differenza dei dispositivi Bluetooth, che assorbono circa 15mA. Inoltre mentre il Bluetooth supporta una tipologia di trasmissione uno a uno, Zigbee supporta anche uno a molti, molti a molti e molti a uno. Le velocità di trasmissione raggiunte dalle due tcnologie sono entrambe molto elevate e sono rispettivamente 250Kbps per Zigbee e 723Kbps per Bluetooth. La tecnologia 802.15.4 presenta una velocità inferiore rispetto ad altre tecnologie, ma tuttavia più che sufficiente per gli scopi di questo lavoro. I dispositivi ZigBee devono rispettare le norme dello standard IEEE 802.15.4-2003 Low-Rate Wireless Personal Area Network (WPAN). Esso specifica il protocollo di livello fisico (PHY), e la parte del livello data link del Medium Access Control (MAC), ovvero i primi due livelli del modello ISO/OSI. Questo standard opera alle frequenze radio assegnate per scopi industriali, scientifici e medici (o anche banda ISM) 2.4 GHz, 915 MHz e 868 MHz. Nella banda 2.4 GHz, utilizzata in questo lavoro ci sono 16 canali ZigBee, da 3 MHz ciascuno. Il protocollo 802.15.4 adotta un sistema di modulazione del tipo *Direct Sequence Spread Spectrum* (DSSS), tecnologia di trasmissione a frequenza diretta a banda larga, cioè ogni bit viene trasmesso come una sequenza ridondante di bit detta *chip*. Il segnale risultante viene quindi propagato su una banda maggiore di quella necessaria ma con livelli di potenza molto bassi. Infatti si chiami C la capacità del canale espressa in bit per secondo; essa è la massima velocità di trasmissione per una teorica *Bit Error Rate (BER)*, e rappresenta quindi la performance del canale desiderata. Detta B l'ampiezza di banda richiesta espressa in Hz, essa rappresenta il prezzo da pagare per la trasmissione in quanto la frequenza è una risorsa limitata, e indicando con S/N il rapporto segnale-rumore la giustificazione teorica di

quanto detto risiede nel fatto che per il teorema di *Shannon* vale la seguente relazione:

$$C = B\,Log_2\left(1 + \frac{S}{N}\right). \tag{4.1}$$

Il rumore può essere provocato da ostacoli, campi magnetici, e onde estranee alla trasmissione che interferiscono. Dall'equazione 4.1 si può notare che si può mantenere una data prestazione C del canale o addirittura incrementarla aggiungendo banda al segnale, anche quando la potenza del segnale è inferiore alla potenza del rumore. Infatti, se nella 4.1 portiamo il logaritmo in base e e applichiamo lo sviluppo in serie di *McLaurin* si ottiene:

$$\frac{C}{B} = 1.443\left(\frac{S}{N} - \frac{1}{2}\left(\frac{S}{N}\right)^2 + \frac{1}{3}\left(\frac{S}{N}\right)^3\right). \tag{4.2}$$

In genere nelle applicazioni dove viene usato il DSSS il rapporto S/N è basso. Considerando quindi S/N$\ll$1, l'espressione di Shannon diventa semplicemente

$$\frac{C}{B} = 1.433\frac{S}{N}. \tag{4.3}$$

Quindi per poter trasmettere informazione libera da errore per un dato rapporto segnale-rumore nel canale, è necessario incrementare la banda di trasmissione.

La modalità di base di accesso al canale specificato da IEEE 802.15.4-2003 è il Carrier Sense Multiple Access / Collision Avoidance (CSMA/CA). Questo significa che i nodi, controllano se il canale è libero, quando devono trasmettere. Se nessuno sta trasmettendo, è possibile inviare il pacchetto, altrimenti è necessario attendere il termine della trasmissione in corso. A questo punto un nodo attende un numero random di *Slot*. Uno *Slot* è un intervallo di tempo, stabilito a priori, di una lunghezza tale da consentire il passaggio dalla fase di ascolto a quella di trasmissione. Il nodo che avrà atteso il minor tempo inizia a trasmettere. Questa soluzione garantisce che in media ogni nodo abbia a propria disposizione la stessa quantità di banda trasmissiva. Ovviamente questo meccanismo non previene in maniera assoluta le

collisioni. Poiché i nodi non sono in grado di ricevere mentre stanno trasmettendo, non è possibile rendersi conto se sta avvenendo un'effettiva collisione. Sono stati aggiunti così altri meccanismi atti a gestire più efficacemente le trasmissioni e che non fanno uso del CSMA. Nelle reti con i beacon abilitati dei nodi speciali chiamati *Zigbee Routers* trasmettono periodicamente dei *beacon* per confermare la loro presenza agli altri nodi della rete. Nelle reti *beacon-oriented* vengono untilizzati dei Guaranteed Time Slots (GTS). I nodi diventano attivi solo dopo la trasmissione del beacon. Tra un beacon e l'altro i nodi entrano in modalità *sleep* in modo da abbassare il duty-cycle e incrementare la durata delle batterie. Gli intervalli fra un beacon e l'altro vanno da 15.36 ms a 15.36*214 = 251.65824 s a 250 kbit/s, da 24ms a 24*214 = 393.216 s a 40 kbit/s e da 48 ms a 48*214 = 786.432 s a 20 kbit/s. Si ricorda che i beacon sono dei segnali che indicano la locazione di un dispositivo o la sua prontezza ad eseguire un compito. Nelle reti wireless un segnale beacon include la mappa del traffico indicando la disponibilità di pacchetti bufferizzati destinati a specifici nodi della rete.

4.1.2 Stub Component e microcontrollori

Nel realizzare robot a basso costo, un problema a cui si deve far fronte è l'implementazione di funzionalità complesse su dispositivi con capacità ridotte. I vincoli sulla capacità di calcolo e di memorizzazione diventano fattori fondamentali nella realizzazione del codice secondo un modello a componenti. Per superare questi problemi i componenti CRTK sono stati progettati come degli stubs, oggetti locali che inviano un messaggio ad un oggetto remoto per richiedere l'operazione associata. I componenti diventano quindi solo delle interfacce attraverso cui chiamare la funzione vera è propria che viene eseguita su microcontrollori esterni. In ogni microcontrollore vengono implementate una o più funzionalità specifiche. Con questo approccio si ottiene un miglioramento delle prestazioni rispetto alle performance ottenute dalla sola Tmote. Alleggerendo il carico computazionale, si possono rendere paralleli più servizi, ma soprattutto è possibile realizzare componenti la cui

complessità è svincolata dalle risorse messe a disposizione dalla sola Tmote. Una architettura di questo tipo oltre a permettere l'esecuzione di applicazioni complesse su dispositivi con limitate capacità, rende possibile una parallelizzazione hardware dei servizi svolti dall'applicazione. Questo è fondamentale nei sistemi dove molte delle informazioni elaborate provengono dai sensori. Questi producono dei segnali elettrici che hanno bisogno di un certo tempo di stabilizzazione. Durante questo tempo il sistema è costretto ad una attesa attiva che non può essere evitata. In una architettura centralizzata, dove la parallelizzazione è solo software, l'attesa di queste informazioni paralizzerebbe il sistema. Con la distribuzione delle funzionalità dell'applicazione ai vari microcontrollori queste attese non bloccano il sistema perché solo una parte di esso ne è interessata, quella coivolta nella lettura dei dati provenienti dall'esterno, mentre il resto è libero di portare avanti le proprie elaborazioni. A questi vantaggi va aggiunta la flessibilità raggiunta dal sistema e la riduzione dei costi, dato che i microcontrollori hanno un costo più basso rispetto ad un calcolatore in grado di compiere le stesse elaborazioni.
Il limite di una architettura di questo tipo è dato dal collo di bottiglia delle comunicazioni fra Tmote e microcontrollori che avvengono mediante il bus I2C.

Bus I2C

"I2C" sta per "IIC", ovvero "Inter Integrated Circuit" e in italiano si legge i-quadro-ci. Come dice il nome, si tratta di un meccanismo pensato per far comunicare diversi circuiti integrati. L'adozione di un canale seriale condiviso come I2C permette di limitare notevolmente il numero di segnali elettrici che bisogna utilizzare rispetto all'approccio in voga precedentemente, secondo il quale ogni integrato aveva un bus indirizzi, un bus dati e un chip select dedicato. Per semplificare l'interfacciamento hardware, questi bus seriali interni al sistema sono sincroni, cioè dotati di un segnale di clock pilotato dall'integrato master. Nel caso di I2C, il bus è composto da due segnali, chiamati SDA e SCL, per serial data e serial clock(Fig. 4.5). I segnali sono

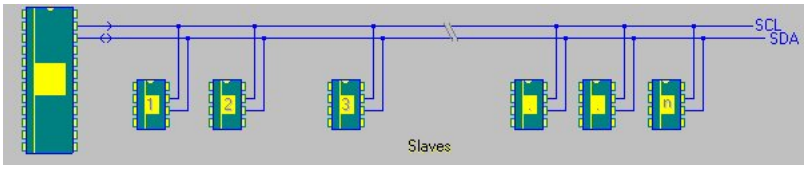


Figura 4.5: Schematizzazione bus I2C

normalmente alti e vengono abbassati da uno dei componenti sul bus. Viene implementato un protocollo CSMA/CA (Carrier-Sense Multiple-Access with Collision Avoidance) che permette, tramite un meccanismo di priorità, trasmissioni contemporanee sul bus, anche se nella maggior parte dei casi si usa un solo master per cui non sono possibili collisioni. Anche nella nostra applicazione, dato che la Tmote-Sky non può essere slave, abbiamo un solo master. Ciò ha il vantaggio il fatto che non si rischiano conflitti sull'acquisizione del bus ma ha come svantaggio il fatto che il master può essere costretto a delle attese attive per leggere le elaborazioni degli slave.
Ogni periferica sul bus risponde a due indirizzi consecutivi lunghi 8 bit, un indirizzo per la scrittura e uno per la lettura; si parla quindi di indirizzi a 7 bit cui viene aggiunto un bit di lettura/scrittura. Dopo ogni byte ricevuto lo slave deve trasmettere un bit di conferma (acknowledgement o ack); in mancanza di ack il master sospende la trasmissione. Per leggere il master invia due pacchetti: tramite il primo si seleziona il subaddress della periferica, col secondo pacchetto si leggono uno o più byte di informazione, pilotando il segnale SCL e leggendo SDA che viene ora pilotato dallo slave; dopo la lettura di ogni byte il master invierà un bit di acknowledgement. Secondo il protocollo il segnale SDA viene modificato solo quando il clock è basso, con l'eccezione dei segnali di start e di stop (che possono così essere identificati da tutte le stazioni sul bus anche senza seguire le sequenze di ack). In figura 4.6, per esempio, è rappresentata una scrittura verso l'integrato con indirizzo 0x20, senza mostrare i dettagli della trasmissione dei byte successivi al primo. La velocità di trasmissione di I2C, inizialmente limitata a 100Kib/s è stata poi elevata a 400Kib/s e successivamente a 3.4Mib/s; si

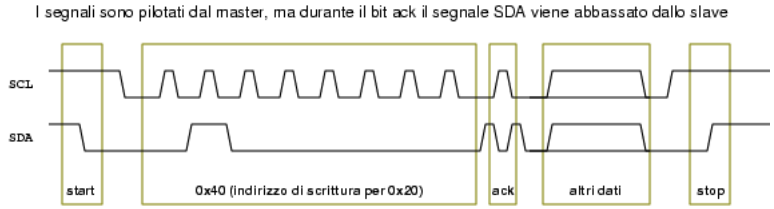


Figura 4.6: Esempio di temporizzazione di una scrittura su I2C

tratta comunque solo dei valori massimi consentiti, mentre non ci sono limiti minimi imposti ciò implica che master e slave possano di fatto pilotare i segnali molto più lentamente.

Microcontrollori Cypress PSoC

Poiché la maggior parte delle funzionalità del robot non vengono eseguite sulla TMote Sky ma su microcontrollori la scelta di quest'ultimi è fondamentale. Per questo lavoro sono stati utilizzati i microcontrollori PSoC della Cypress. La sigla PSoC è l'acronimo per Programmable System on Chip. Questo dispositivo consiste di diversi sottosistemi su un singolo chip. I sistemi stessi e le connessioni fra loro possono essere configurate per ottenere un tipo di sistema particolare. Ciò comporta un notevole risparmio di costi in quanto un sistema che avrebbe previsto l'uso di più componenti può essere realizzato mediante un singolo PSoC. La famiglia dei PSoC è costituita da Array di segnali misti con dispositivi per il controllo montati *on chip*. Questi dispositivi sostituiscono i tradizionali componenti che costituiscono un microcontrollore, con il vantaggio di trovarsi tutti su un singolo chip dal basso costo e (ri-)programmabile. I PSoC includono dei blocchi configurabili di logica analogica e digitale, così come interconnessioni programmabili. Questa architettura permette all'utente di creare delle configurazioni su misura specifiche per l'applicazione che sta sviluppando. Inoltre, come mostrato in figura 4.7 sui PSoC troviamo una CPU, una memoria Flash programmabile e una memoria SRAM per i dati. L'architettura di un PSoC è composta da quattro aree principali: il Core, il sistema digitale, il sistema analogico e le

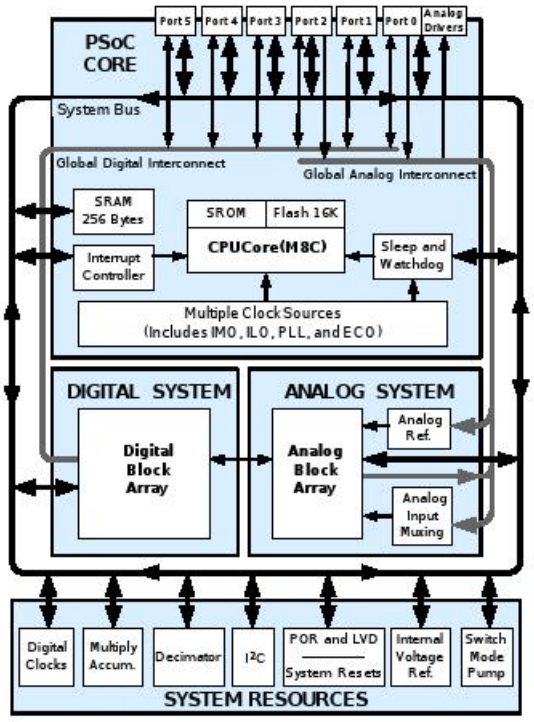


Figura 4.7: Sottosistemi PSoC

risorse di sistema.

Il **Core** è il motore del microcontrollore ed è costituito dalla CPU, dalla memoria, dai generatori di clock e da una periferiche di I/O generica chiamata General Purpose I/O (GPIO). La CPU è costituita da un processore che raggiunge la velocità di 24MHz ed utilizza un controller per le interruzioni con 17 vettori, per semplificare la programmazione degli eventi real-time. L'esecuzione dei programmi è cronometrata e protetta usando i timer di Sleep e Watch Dog. Dipendentemente dalla serie di PSoC la memoria comprende 16K o 32K di Flash per il salvataggio dei programmi, 256 bytes o 2K di SRAM per i dati e fino a 2K di EEPROM simulata utilizzando la Flash. Nei PSoC troviamo diversi generatori di clock:

Internal Main Oscillator (IMO) : un oscillatore interno principale a 24 MHz che può anche essere portato a 48MHz per essere usato con sistemi digitali;

Internal Low speed Oscillator (ILO) : un oscillatore a bassa potenza a 32kHz per implementare i timer di Sleep e Watch Dog;

External Crystal Oscillator (ECO) : un oscillatore a 32.768 kHz che è può essere utilizzato come Real Time Clock o per generare, con un PLL, sistemi accurati a 24 MHz.

Le periferiche di I/O generiche forniscono le connessioni alla CPU con risorse di tipo analogico o digitale. Va notato che il modo di pilotaggio di ogni pin può essere selezionato fra otto opzioni avendo così una grande flessibilità nell'interfacciamento con l'esterno.
Il **sistema digitale** è composto da 8 blocchi digitali che costituiscono ciascuno una risorsa a 8 bit che può essere usata da sola o combinata insieme per formare periferiche a 8, 16, 24, 32 bit. Le periferiche digitali disponibili sui PSoC sono: PWM, PWM con Dead band, Contatori, Timers, UART, SPI slave e master, I2C slave e multi-master, Controllore/Generatore di ridondanza ciclica, IrDA, e generatori di sequenze pseudo ramdom. I blocchi digitali possono essere connessi con ogni GPIO attraverso una serie di bus che possono portare ogni segnale su ogni pin. Questo si riflette in una maggiore facilità nella progettazione dell'hardware in cui il PSoC andrà inserito.
Il **sistema analogico** è composto da 12 blocchi configurabili ogniuno con un circuito con il proprio amplificatore operazionale (opamp) per la generazione di segnali analogici complessi. Le periferiche analogiche sono molto flessibili e possono essere adattate per aderire a specifiche applicazioni. Tra le funzioni analogiche disponibili si trovano: convertitori analogico digitali, filtri, amplificatori, DAC, DTMF dialer, modulatori, rivelatori di picco, ecc.
Le **risorse di sistema**, alcune delle quali sono state citate precedentemente, forniscono delle utilità che rendono il sistema più completo. Esempi sono i moltiplicatori, i decimatori e i rivelatori di basso voltaggio.

4.1.3 Software implementato sul RAiN2

In questo sottoparagrafo viene discussa l'implementazione del software, il cui listato è presente in appendice, eseguito sull'agente. I componenti software che implementano il codice per il funzionamento della piattaforma robotica sono due: il Collision-Obstacle Avoidance Component (COAC) e il Motor Component (MC). Il primo realizza la politica risultante dall'integrazione fra GRP e APF e quindi l'algoritmo mostrato in figura 2.5, il secondo gestisce la dinamica del vicolo mediante il controllo dei motori.
All'accensione del robot viene inizializzato un processo il quale ciclicamente attende l'arrivo di un messaggio proveniente dal server di localizzazione. Il messaggio, che descrive lo stato del veicolo e del centro del disco riservato, può essere di due tipi: di inizializzazione o di stato. La distinzione avviene per mezzo di un byte che ne specifica la natura. Infatti le coordinate del goal da raggiungere vengono specificate attraverso il server di localizzazione. In figura 4.8 sono mostrate le informazioni sullo stato che vengono inviate all'agente. Il vettore di stato è costituito da l'identificativo del robot (id),

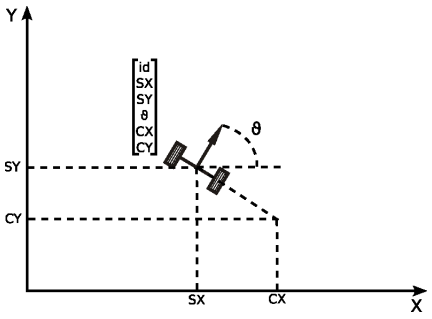


Figura 4.8: Descrizione del vettore di stato di un agente.

dalle coordinate e dall'orientazione del robot (SX, SY, θ) e dalle coordinate del centro del disco riservato (CX, CY). I messaggi inviati dal server di localizzazione sono inviati in broadcast a tutti gli agenti presenti nell'area di lavoro. L'informazione riguardo l'identificativo di ciascun robot serve per capire a chi è riferita una certa localizzazione e verrà utilizzata, come si vedrà in seguito, per l'attuazione della politica di collision-obstacle avoidance. Inoltre

questa informazione viene utilizzata dagli agenti per scartare i messaggi di inizializzazione che non riguardano quel determinato robot.
Il protocollo di comunicazione fra il server e gli agenti prevede che il messaggio sia composto da 12 byte. La sua struttura è mostrata in figura 4.9 dove ID è l'identificativo del robot, MT specifica se le informazioni all'interno del messaggio sono di inizializzazione o di stato, SX e SY sono le coordinate del robot, OR è l'orientazione θ dell'agente, CX e CY le coordinate del centro del disco riservato. Tutti i dati riguardanti lo stato sono impacchettati su

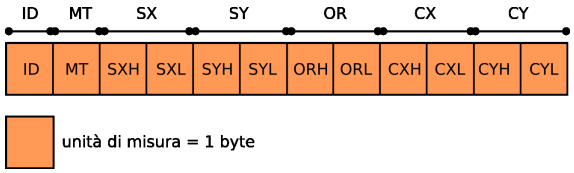


Figura 4.9: Descrizione composizione messaggio di localizzazione.

due byte, in modo che il range di numeri rappresentabili sia tale da coprire tutte le coordinate dell'area di lavoro.
Una volta ricevuto il messaggio e discriminata la sua tipologia vengono chiamate le funzioni del componente COAC, fornite mendiante la sua interfaccia che è così realizzata:

```
 DECLARE_INTERFACE(icoac,
{
  void (* COAC_initParam) (unsigned int8 IdBot, int8 numNeigh);
  void (* COAC_start) (int8 *commad);
  void (* COAC_setGoalInt) (unsigned int8 status[],
int8 threshold, short len);
  void (* COAC_robPositionInt) (unsigned int8 idBot,
unsigned int8 status[], short len);
  void (* COAC_robotIdentification)(unsigned int8 *botID);
  void (* COAC_robotRecognized)(unsigned int8 *botREC);
});
```

Le funzioni dichiarate nell'interfaccia svolgono le seguenti attività:

- La `COAC_initParam(unsigned int8 IdBot, int8 numNeigh)` prende come ingresso il numero identificativo del robot e il numero dei vicini. I parametri servono per assegnare un *nome* al robot e per inizializzare le strutture dati necessarie per salvare le informazioni riguardo alle localizzazioni degli altri robot nell'ambiente. Infatti le localizzazioni provenienti dal visualServer vengono inviate in broadcast e sono disponibili a tutti i veicoli. Le informazioni relative agli altri veicoli vengono utilizzate per l'applicazione della politica di Collision avoidance.

- La `COAC_start(int8 *commad)` è la funzione principale ed quella che attiva l'esecuzione della GRP e della APF. Essa ritorna lo stato in cui si deve trovare il veicolo e che verrà utilizzato dalla `commandToMotor`.

- La `COAC_setGoalInt(unsigned int8 status[], int8 threshold, short len)` è una funzione di inizializzazione che serve per settare il goal che il veicolo dovrà raggiungere. Il vettore `status` contiene lo stato del goal espresso in ascisse, ordinate e orientamento. Il parametro `len` specifica la lunghezza del vettore `status`. Il parametro `threshold` indica la soglia sotto la quale si considera il goal raggiunto.

- La `COAC_robPositionInt(unsigned int8 idBot, unsigned int8 status[], short len)` aggiorna la posizione del veicolo nell'ambiente operativo. Deve essere chiamata prima della `COAC_start()`. Anche in questo caso troviamo il vettore di byte `status[]` che contiene la posizione attuale del robot e il parametro `len` che specifica la sua lunghezza. Inoltre troviamo l'identificativo del robot `idBot`. Poiché i messaggi sulle localizzazioni vengono mandati in broadcast dal server di visione l'identificativo del robot serve per chiarire a quale dei veicoli nell'ambiente è riferita la localizzazione. Ogni robot memorizza queste informazioni e le utilizza per vedere se sono verificate le condizioni di vicinanza.

- `COAC_robotIdentification(unsigned int8 *botID)` e `COAC_robotRecognized(unsigned int8 *botREC)` Sono due funzioni di utilità

che ritornano i valori di alcuni parametri interni al componente. La prima ritorna l'identificativo del robot mentre la seconda il byte su cui vengono salvate le informazioni relative ai vicini del robot.

La funzione `COAC_setGoalInt` viene chiamata quando è stato ricevuto un messaggio di inizializzazione indirizzato al robot corrente. Viceversa le funzioni `COAC_robPositionInt`, `COAC_robotRecognized` e `COAC_start` vengono lanciate a fronte di una ricezione di un messaggio di stato. Ottenuto il comando da far eseguire ai motori dal COAC viene invocata la `commandToMotor`, la funzione fornita dall'interfaccia del componente Motor Component (MC). L'interfaccia di questo componente ha un'unica funzione:

```
DECLARE_INTERFACE(imotor,
{
  void (* commandToMotor) (int8 command,float u[]);
});.
```

Questa funzione manda il comando delle velocità ai motori attraverso il parametro `command` e ritorna le velocità delle ruote nel vettore `u[]`. Poiché i possibili stati del veicolo sono quattro per ogniuno di essi è stata associata una opportuna velocità delle ruote, costante in quello stato.
Inviato il comando ai motori il processo cicla e ritorna ad aspettare degli eventi di tipo TCP/IP.
Come già detto nel paragrafo 4.1.2 i componenti utilizzati, il COAC ed il MC, sono in realtà degli stubs. Le funzioni appena discusse svolgono le azione preliminari all'invio dei dati ai dispositivi che effettivamente realizzano il corpo del componente.
Il COAC è realizzato utilizzando due PSoC differenti; in uno vi è implementato la GRP mentre nell'altro l'APF. Il componente MC usa un altro PSoC ancora in cui si trova l'effettivo controllore dei motori. La comunicazione tra la TMote Sky dove si trovano i componenti COAC e MC e i PSoC avviene mediante il bus I2C. Di seguito mostreremo quali siano le effettive operazioni svolte dalle funzioni sopra descritte, quali protocolli di comunicazione vengano utilizzati e quali sono le operazioni implementate sui PSoC.

Implementazione degli stub component su TMote Sky

La funzione `COAC_setGoalInt` salva le coordinate del goal, ricevute attraverso un messaggio di inizializzazione del server, in una opportuna struttura dati quindi prepara e invia un messaggio al PSoC che implementa la GRP, che chiameremo PSoC_GRP. La struttura del messaggio è simile a quella del messaggio arrivato mediante TCP/IP con l'aggiunta di due campi, ciascuno di un byte, uno in testa e uno in coda che specificano l'inizio e la fine della trama trasmessa: i dati contenuti fra questi due *headers* corrispondono al payload del messaggio. Per il byte di start è stato scelto il valore esadecimale 0x7D e come byte di stop è stato scelto il valore 0x7E.
La funzione `COAC_robPositionInt` salva le informazioni contenute nel messaggio di localizzazione proveniente dal server in due strutture dati differenti. Le coordinate del centro del disco riservato vengono salvate in `robot_information`, un array di *unsigned short int* dove si trovano le coordinate di tutti i centri dei dischi riservati di tutti i veicoli riconosciuti. L'inserimento nell'array è posizionale e legato all'identificativo del robot. Le coordinate del robot sono salvate in `status_robot` un'array di byte senza segno, ciascuna su due byte.
La funzione `COAC_start` innanzitutto controlla che non sia stato raggiunto il goal quindi, se questa condizione non è verificata prepara ed invia un messaggio per il PSoC che implementa l'APF, che chiameremo PSoC_APF, prelevando le informazioni dalle strutture dati `status_robot` e `goal_robot` dove la funzione `COAC_robPositionInt` e la funzione `COAC_setGoalInt` le avevano salvate. Per il protocollo progettato per la comunicazione con questo dispositivo il messaggio deve avere la struttura di figura 4.10. SX, SY e OR rappresentano lo stato del'agente, mentre GX e GY sono le coordinate del goal. Effettuato l'invio vengono calcolate le condizioni di vicinanza con gli altri agenti controllando la distanza del centro del disco riservato da quello degli altri vicini. I robot vicini vengono segnalati settando ad uno il bit relativo all'indice del robot nella variabile `neighbours`. Vengono quindi inviate le informazioni sullo stato corrente al PSoC_GRP. Il messaggio composto ha

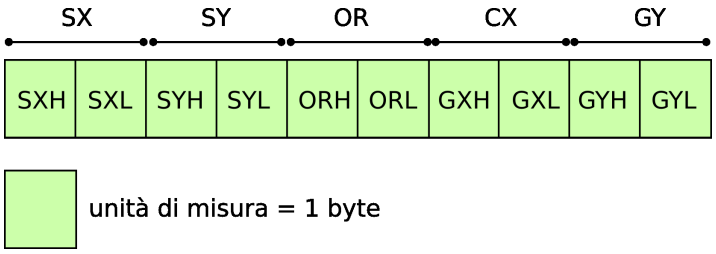


Figura 4.10: Descrizione composizione messaggio per il PSoC_APF.

una lunghezza variabile legata al numero di agenti che sono in contatto con il robot corrente. In figura 4.11 ne è mostrato un esempio nel caso di N vicini. Come nel caso di invio della localizzazione del goal, anche in questo caso sono

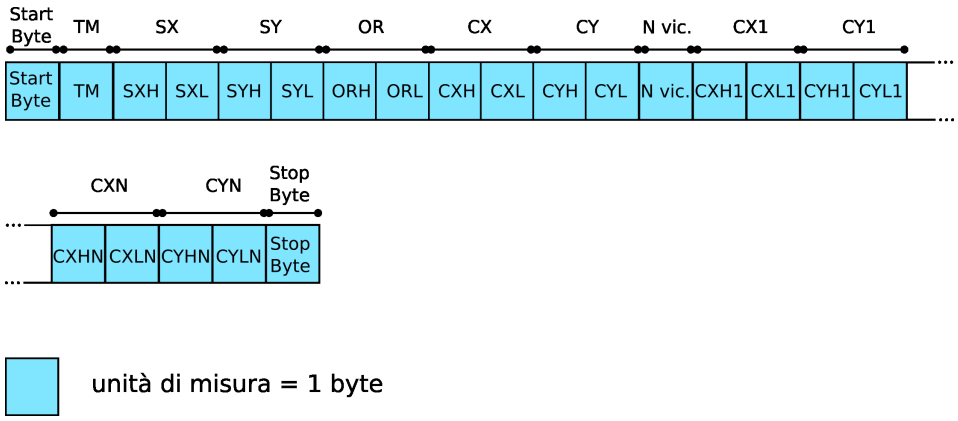


Figura 4.11: Descrizione composizione messaggio per il PSoC_GPR.

presenti i due byte di intestazione in testa (Start Byte) e in coda (Stop Byte) che racchiudono la trama spedita, quindi un byte, Type Message (TM), dove è indicata la tipologia del messaggio e le informazioni riguardo lo stato del veicolo e del centro del disco riservato (SX, SY, OR, CX, CY). Oltre a ciò è presente un campo che indica il numero di agenti i cui dischi riservati sono "in contatto" con il disco riservato dell'agente corrente. Se questo numero è zero allora il messaggio si chiude subito dopo con il byte di stop, altrimenti se è N vengono aggiunte le informazioni riguardo le coordinate dei centri dei dischi riservati degli altri veicoli (CX1, CY1, CX2, CY2, ..., CXN, CYN) prelevandole dall'array `robot_information` in base ai bit posti ad uno nel

byte `neighbours`.
Effettuate le operazioni di invio ed aspettato il tempo necessario per avere la certezza che le elaborazioni sui PSoC siano terminate vengono lette le riposte. Il PSoC_GRP ritorna un unico byte che rappresenta la codifica dello stato in cui dovrà trovarsi l'agente, mentre il PSoC_APF ritorna cinque byte strutturati come in figura 4.12. I primi cinque byte rappresentano la codifica della

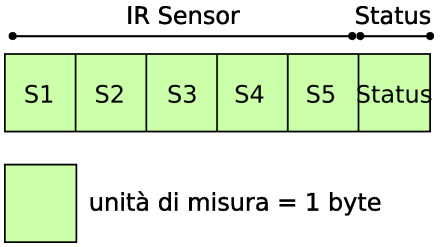


Figura 4.12: Descrizione composizione messaggio proveniente dal PSoC_APF.

distanza degli ostacoli dal veicolo prelevata dai cinque sensori ad infrarossi di cui è provvisto l'agente. L'ultimo byte è la codifica dello stato in cui si deve trovare il veicolo. Si deve precisare che l'applicazione della politica dell'APF non prevede l'utilizzo di una macchina a stati come per la GRP. In realtà il prodotto dell'algoritmo è un vettore la cui orientazione indica la direzione che l'agente deve seguire mentre il modulo è proporzionale alla velocità che il veicolo dovrà tenere. Queste informazioni però non sono utilizzabili direttamente perché non compatibili né con i risultati della GRP né con il Motor Component che si aspetta uno stato nel quale la velocità è costante. Quindi, come si vedrà in seguito, si è fatto in modo che il PSoC_APF restituisca uno dei quattro stati previsti anche dalla GRP. Le informazioni provenienti dai sensori vengono utilizzate durante l'attuazione della GRP per evitare le collisioni con ostacoli statici.
Ritornato il nuovo stato per l'agente la `COAC_start` termina e parte la chiamata alla `commandToMotor`. Questa funzione in base allo stato ricevuto compone e invia un messaggio al PSoC, sui cui è implementato il controllore dei motori, con i riferimenti di velocità che devono essere inseguiti. In figura 4.13 è

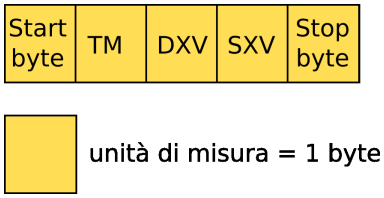


Figura 4.13: Descrizione composizione messaggio per il controllore dei motori.

mostrata la struttura del messaggio destinato al controllore dei motori. Sono presenti i due byte di inizio e fine messaggio (Start-Stop byte), un byte che specifica il tipo di messaggio(TM) e due byte con i riferimenti per le velocità del motore destro (DXV) e sinistro (SXV).

Implementazione dei componenti sui PSoC

Sul **PSoC per il controllo motori** sono presenti le seguenti funzionalità:

- Ricezione dati dal Motor Component
- Implementazione del controllore PID
- Conteggio degli impulsi provenienti dagli encoders

Per realizzare queste funzionalità sul PSoC sono stati programmati un modulo per la comunicazione su I2C, un modulo timer per l'integrazione numerica per il controllore PID e due moduli PWM per comandare i motori. Una volta attivato il main attende ciclicamente l'arrivo di un messaggio via I2C. All'arrivo del primo messaggio con almeno uno dei riferimenti di velocità diverso da zero viene attivato il timer. Il modulo timer genera periodicamente un interrupt. Il valore del periodo può essere impostato via software. Per l'integrazione numerica del controllore PID il periodo è stato scelto pari a 100ms. Ad ogni interrupt, viene eseguita la procedura di integrazione numerica mediante la chiamata alla funzione `void Integra()`. Per evitare problemi di overflow legati al tipo delle variabili utilizzate sono stati previsti meccanismi di saturazione software dei valori (soprattutto per la componente integrale) al fine di garantire che i valori calcolati non causino l'overflow delle variabili

stesse. L'uscita calcolata del PID viene convertita in un valore di tensione restituito ai motori tramite le chiamate alle API dei moduli PWM. La conversione avviene regolando il valore del duty-cicle del PWM dal momento che la tensione ai capi del motore è pari al valor medio del duty-cicle stesso. Con 8 bit di risoluzione il PWM utilizzato è quantizzato in un valore percentuale da 0% a 100%: con il duty-cicle a 0% la tensione ai capi del motore corrisponde a 0V, mentre con il duty-cicle al 100% la tensione è pari alla tensione di alimentazione. I valori compresi sono quantizzati uniformemente sui 254 intervalli rimanenti. Attraverso un coefficiente di proporzionalità si scala l'uscita del controllore PID in Volts su un valore ben preciso del duty-cicle in percentuale e si quantizzata sui 256 livelli. Il coefficiente di proporzionalità e calcolato secondo la seguente relazione:

$$PWM = u \times \frac{\frac{100}{7.2}}{256} \tag{4.4}$$

dove u rappresenta l'uscita del controllore in Volts, 7.2 è il valore massimo della tensione di alimentazione e i valori 100 e 256 servono rispettivamente per scalare la tensione di controllo su un valore percentuale e per quantizzare il valore ottenuto su 256 intervalli. Questo risultato è poi utilizzato come valore a cui settare ad ogni passo di integrazione il valore di uscita del modulo PWM. Il compito di lettura degli encoder è un'azione critica. Questo perché un ritardo nella lettura degli encoders provoca un allungamento dell'intervallo di campionamento che, supponendo che l'asse del motore giri a velocità costante, porta ad una lettura di spostamenti angolari non costanti; il controllo allora cercherebbe di regolare la velocità angolare per mantenerla costante, deviandola invece dall'attuale situazione di equilibrio. A questo scopo è stato scelto un approccio *event-based* per la gestione degli encoders: questo consente di mantenere il processore mediamente meno impegnato, in quanto nel caso che l'asse del motore sia fermo, il microcontrollore non è impegnato a leggere gli encoder. Per catturare la transizione di stato degli encoders sono stati utilizzati due blocchi digitali con funzione di inverter logici. Poiché la posizione dei blocchi sul PSoC ne determina la priorità questi sono stai posti in modo che durante l'esecuzione della routine di interruzione

del timer un'interruzione scatenata dall'encoder non rimanga pendente fino al termine dell'esecuzione della routine di interruzione del timer stesso. All'arrivo di due riferimenti di velocità entrambi nulli viene disattivato il timer con il conseguente arresto del veicolo. Questo perché la politica di controllo del veicolo prevede che questo non si fermi mai durante l'esecuzione del task. Quindi quando i riferimenti sono entrambi nulli significa che il task è stato portato a termine.
Sul **PSoC che implementa la GRP** sono svolti ciclicamente tre compiti principali:

- Ricezione dei dati provenienti dal Collision-Obsacle Avoidance Component;
- Memorizzazione goal/Esecuzione della GRP;
- Comunicazione del nuovo stato al Collision-Obsacle Avoidance Component.

Il PSoC, quindi prevede un solo blocco digitale per la realizzazione delle comunicazioni su I2C. Per quanto riguarda il primo punto si attende una scrittura sul bus I2C. I due tipi di messaggi provenienti dalla TMote Sky producono azioni differenti. Se il messaggio è di inizializzazione le informazioni contenute vengono salvate in una opportuna struttura dati, altrimenti, nel caso di messaggio di stato, viene attivata l'esecuzione della GRP mediante la chiamata alla funzione `DecisionMaking()`. Con le informazioni contenute in un messaggio di stato e quelle relative al goal precedentemente memorizzate vengono svolti i conti per il calcolo del cono Θ e dell'angolo ϕ(vedi par. 2.1) e viene deciso il nuovo stato del veicolo ritornato mediante l'istanziamento di una variabile globale. Questa varibile viene poi copiata nel buffer di uscita del blocco digitale I2C pronta per essere letta.
Il modulo che implementa l'APF prevede le seguenti funzionalità:

- Ricezione dei dati provenienti dal Collision-Obsacle Avoidance Component;

- Lettura dei sensori ad infrarossi;
- Esecuzione dell'APF;
- Comunicazione del nuovo stato al Collision-Obsacle Avoidance Component.

I moduli digitali programmati sul PSoC sono: un modulo per la comunicazione I2C, un modulo per la conversione analogico digitale, un amplificatore di segnale e un modulo PWM come generatore di interruzioni. Il PWM genera sul fronte di salita dell'onda quadra un interrupt. Ad ogni interrupt, viene eseguita la procedura di lettura dei sensori e il calcolo della distanza mediante la funzione `PWM_interrupt`. Il valore del periodo può essere impostato via software ed è stato scelto pari a 67ms. Vengono valutati quegli ostacoli la cui distanza è inferiore a 50cm. Le distanze valutate vengono rappresentate su 256 livelli. Distanze maggiori rilevate vengono rappresentate con il livello massimo. Sono stati utilizzati dei meccanismi di saturazione software per i dati rilevati dai sensori in modo da evitare problemi di overflow legati al tipo di variabili usate. Anche in questo caso, come nel modulo per il controllo dei motori, è stato scelto un approccio *event-based* per la gestione della concorrenza fra il processo di lettura dei sensori e quello di applicazione della APF. Questo per garantire la priorità al processo di lettura in modo da avere consistenza fra i dati letti e la condizione reale de robot, non essendo presente sui microcontrollori PSoC un sistema operativo e quindi nessun meccanismo software per l'assegnamento e la gestione dei livelli di priorità dei task. Il processo che implementa l'APF attende l'arrivo di una localizzazione sul bus I2C quindi applica la politica utilizzando le seguenti funzioni:

- `CalcoloForzaAttrattiva()`: utilizza le informazioni sulla posizione attuale e sulla posizione del goal ottenute dal COAC per calcolare modulo e orientazione del vettore che esprime la forza attrattiva verso il goal;
- `CalcoloForzaRepulsiva()`: utilizza le informazioni provenienti dagli

infrarossi per calcolare modulo e orientazione della forza repulsiva dagli ostacoli;

- `CalcoloForzaTotale()`: calcola modulo e orientamento della forza risultante quindi riporta questa informazione in uno stato fra quello previsti dalla GRP e lo ritorna mediante l'istanziamento di una variabile globale. Il valore preso da quest'ultima verrà reso disponibile al COAC attraverso il buffer di uscita come precedentemente descritto e il processo ritorna in attesa di una nuova localizzazione sul bus.

Capitolo 5

Visual Server

In questo capitolo viene illustrato il meccanismo di localizzazione mediante visione, il server di visione sviluppato e la tecnologia di riconoscimento di immagini che si è utilizzata per realizzarlo.

5.1 Retroazione visiva

Lo schema di controllo relativo ad ogni agente può essere schematizzato come in figura 5.1. Come si può notare dalle figura la localizzazione mediante

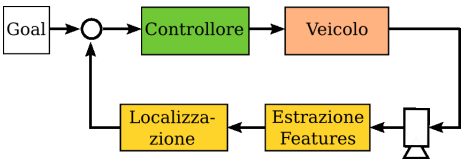


Figura 5.1: Schema di controllo in retroazione di ogni agente.

visione è un elemento essenziale nell'anello di controllo. Se il server di visione non riesce a identificare il marker del robot l'anello si apre. Il server invia le informazioni relative ad una localizzazione in maniera asincrona ad ogni riconoscimento del marker, quindi la sequenza di campioni generata, relativamente allo stato del veicolo, è casuale. Un algoritmo di controllo che usa un campionamento casuale converge con probabilità che tende ad uno

all'aumentare del numero di campioni. É necessario quindi che il numero di localizzazioni da parte del server di visione sia quanto più possibile elevato. Più precisamente affinché una sequenza di campioni sia più prossima possibile allo spazio degli stati $\mathcal{C}$ che caratterizza un percorso dalla posizione iniziale a quella finale, esclusi gli estremi, occorre che sia *densa*. Poiché $\mathcal{C}$ è uno spazio topologico [1] dalla topologia si ha che se U e V sono due spazi topologici, l'insieme U è detto essere *denso* in V si $cl(U) = V$[2]. Una sequenza di campioni casuale risulta essere *probabilmente densa*. Infatti supponiamo che lo spazio degli stati $\mathcal{C} = [0, 1]$. Sia $I \subset [0, 1]$ un intervallo di lunghezza e. Se k campioni sono scelti indipendentemente a caso la probabilità che nessuno di questi cada dentro I è $(1 - e)^k$. Come k tende ad infinito la probabilità tende a zero. Quindi una sequenza infinita di campioni scelti a caso è *densa con probabilità uno*, senza garanzie che lo sia. Nella pratica se il numero di campioni è sufficientemente elevato, il fatto che la sequenza sia solo densa con un certa probabilità non produce ripercussioni sul comportamento del robot. Se la sequenza generata non possiede abbastanza campioni allora quello che si osserva è la difficoltà del robot di riuscire a raggiungere il goal.

[1] Un insieme X è detto *spazio topologico* se esiste una collezione di sottoinsiemi di X chiamati insiemi aperti per cui valgono i seguenti assiomi

1. L'unione di un numero contabile di insiemi aperti è un insieme aperto.
2. L'intersezione di un numero finito di insiemi aperti è un insieme aperto.
3. Sia X che $\emptyset$ sono insiemi aperti.

[2] Dato un insieme U, $int(U)$ sia l'insieme dei punti interni ad U e ∂U l'insieme dei punti sulla frontiera di U. Allora $cl(U) = int(U) \cup \partial U$

5.2 Localizzazione mediante visione

Per la determinazione dello stato degli agenti appartenenti allo scenario, è stata utilizzata una telecamera modello Logitech WebCam Orbit. La telecamera necessita di un processo di calibrazione per impostare parametri necessari per rendere robusto ed efficiente il funzionamento del sistema finale.

La determinazione dello stato degli agenti attraverso una telecamera richiede di ricavare le coordinate dei punti nello spazio, partendo dai punti appartenenti all'immagine generata. La telecamera genera una trasformazione che associa i punti sulla scena ai punti sull'immagine. Di seguito mostreremo le relazioni che caratterizzano questa trasformazione. Poiché nel passaggio dallo spazio tridimensionale a quello bidimensionale dell'immagine si ha la perdita dell'informazione legata alla profondità si è considerata nota l'altezza (indicata in seguito con Z_c) a cui è stata posta la WebCam.

Consideriamo il modello geometrico interno di una telecamera, mostrato in figura 5.2. Gli oggetti posti davanti al *piano focale F*, il piano tangente alla

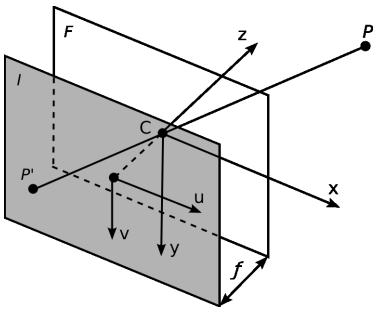


Figura 5.2: Modello geometrico interno telecamera.

superficie del foro della telecamera (*pinhole*), emettono dei raggi luminosi per emissione riflessa e una parte di loro entra attraverso il foro, andando a colpire il *piano immagine I* interno alla camera. La distanza tra i piani F ed I è chiamata *distanza focale*, la retta normale al piano focale passante per il foro *asse ottico* e l'intersezione tra l'asse ottico ed il piano immagine visibili *punto principale*. Sia f la distanza focale e si fissi un sistema di ri-

ferimento cartesiano tridimensionale (X, Y, Z) con origine nel centro ottico C. Si scelga un punto P della scena di coordinate (x, y, z). Sia P' (x, y, z) il corrispondente punto sul piano I. L'immagine è invertita rispetto alla scena originale. Attraverso considerazioni geometriche e sfruttando la similitudine tra i triangoli segue che:

$$\frac{x}{z} = \frac{-x}{f}$$

$$\frac{y}{z} = \frac{-y}{f}$$

da cui si ricava che:

$$x = \frac{-f}{z}x$$

$$y = \frac{-f}{z}y$$

$$z = -f$$

Le equazioni precedenti descrivono la *proiezione prospettica*. La telecamera è composta dalla parte ottica (le lenti) e dal CCD che costituisce il piano immagine formato da una matrice di n×m elementi sensibili alla luce. Questi elementi restituiscono un potenziale elettrico proporzionale all'intensità della radiazione luminosa incidente. Il potenziale elettrico viene digitalizzato e si ottiene così l'immagine finale, composta di N×M pixel. Non è detto che ad ogni pixel corrisponda un solo elemento fotosensibile, spesso le due matrici sono legate da una relazione lineare, ma per semplicità si assume che vi sia una relazione uno ad uno.
Consideriamo un sistema di riferimento per lo scenario la cui origine non sia coincidente con il sistema cartesiano della telecamera, centrato in C, e altre due sistemi di riferimento legati all'immagine. I quattro sistemi di riferimento considerati sono:

1. Il sistema di riferimento $3d$ detto *sistema mondo*;

2. Il sistema di riferimento $3d$ standard della telecamera centrato in C;

3. Il sistema di riferimento $2d$ per l'immagine;

4. Il sistema di riferimento nel piano immagine digitale.

Dato un qualunque punto P della scena, esso potrà essere individuato in ogniuno di questi diversi sistemi. In particolare si indicheranno con $X = (X, Y, Z)$ le coordinate del punto P nel *sistema mondo*, con $X_C = (X_C, Y_C, Z_C)$ coordinate del punto P nel sistema della camera, con $x = (x, y)$ le coordinate del punto P nel sistema dell'immagine e con $w = (u, v)$ le coordinate del punto P nel sistema dell'immagine digitale. L'ultimo sistema tiene in considerazione la discretizzazione fatta durante la conversione dell'immagine da analogica a digitale. Il sistema mondo e il sistema telecamera sono legati tra loro da una *trasformazione isometrica*[3] composta da una matrice R e un vettore di traslazione t come descritto nell'equazione seguente

$$\begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}. \tag{5.1}$$

La 5.1 può essere riscritta in forma compatta

$$\overline{X}_C = P_R \overline{X}, \tag{5.2}$$

dove le lettere con la barra superiore indicano che ci stiamo riferendo a coordinate omogenee. Ricordando le equazioni che descrivono la proiezione prospettica, si ricava

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{-f}{Z_C} & 0 & 0 & 0 \\ 0 & \frac{-f}{Z_C} & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix}. \tag{5.3}$$

[3]Si definisce trasformazione isometrica una mappa biettiva fra due spazi metrici che preserva le distanze ovvero $d(f(x), f(y)) = d(x, y)$, dove f è la mappa e $d(a, b)$ è la funzione distanza.

Inserendo un fattore di scala $s = Z_C$ si ottiene

$$\begin{bmatrix} sx \\ sy \\ s \end{bmatrix} = \begin{bmatrix} -f & 0 & 0 & 0 \\ 0 & -f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} \tag{5.4}$$

che indicata in modo più compatto diventa

$$\overline{x} = P_P \overline{X}_C. \tag{5.5}$$

A questo punto occorre tenere conto che si sta lavorando su immagini digitali, quindi discrete e formate da un numero finito di pixel. Le relazioni che esistono tra le coordinate dell'immagine (x, y) e le coordinate in pixel (u, v) sono le seguenti

$$\begin{cases} u = u_0 + k_u x, \\ v = v_0 + k_v y, \end{cases} \tag{5.6}$$

dove il punto (u_0, v_0) sono le coordinate del punto principale mentre k_u e k_v indicano la dimensione efficace del pixel lungo le due dimensioni u e w. Riportando queste relazioni in forma matriciale si ottiene

$$\begin{bmatrix} su \\ sv \\ s \end{bmatrix} = \begin{bmatrix} k_u & 0 & 0 & u_0 \\ 0 & k_v & 0 & v_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} sx \\ sy \\ 1 \end{bmatrix} \tag{5.7}$$

$$\overline{w} = P_c \overline{x}. \tag{5.8}$$

Sostituendo nella 5.8 il valore di $\overline{x}$ dato dalle 5.5 si ottiene

$$\overline{w} = P_c P_p \overline{X}_C. \tag{5.9}$$

A questo punto sono a disposizione tutte le grandezze necessarie per passare dal sistema mondo alle coordinate in pixel. Sostituendo nella 5.9 la 5.2 si ricava la relazione

$$\overline{w} = P_c P_p P_R \overline{X} \tag{5.10}$$

Raggruppando le matrici in modo che

$$P_c P_p P_R = P_{cp} P_R = \begin{bmatrix} -fk_u & 0 & u_0 & 0 \\ 0 & -fk_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

si ottiene:

$$P_{cp} = [K|0]\,,$$

con

$$K = \begin{bmatrix} -fk_u & 0 & u_0 \\ 0 & -fk_v & v_0 \\ 0 & 0 & 1 \end{bmatrix}$$

e analogamente si ricava che

$$P_R = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$$

con

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad t = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}.$$

Quindi la relazione 5.10 può essere espressa in questo modo

$$\overline{w} = K\,[I|0] \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \overline{X} = K\,[R|t]\,\overline{X} = P\overline{X}.$$

La matrice P che ha dimensioni 3×4 prende il nome di *matrice di proiezione prospettica* (MPP) e rappresenta il modello geometrico della telecamera, mentre la matrice $[R|t]$ rappresenta sia la rotazione che la traslazione del sistema mondo rispetto al sistema fissato al pinhole, e i suoi valori numerici rappresentano i parametri estrinseci della telecamera. Nella matrice K si può porre $a_u = -fK_u$ e $a_v = -fK_v$. Tali grandezze rappresentano la lunghezza focale espressa in pixel e insieme alle coordinate (u_0, v_0) del punto centrale vanno a far parte dell'insieme dei parametri intrinseci della telecamera. Questi due insiemi di parametri identificano completamente la telecamera. Per parametri intrinseci della telecamera si è fatto riferimento a [13] in quanto la WebCam utilizzata è della stessa tipologia di quella utilizzata in quel lavoro. Per i parametri estrinseci abbiamo posto

$$[R|t] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1700 \end{bmatrix}$$

Le relazioni appena descritte sono riassunte in figura 5.3.

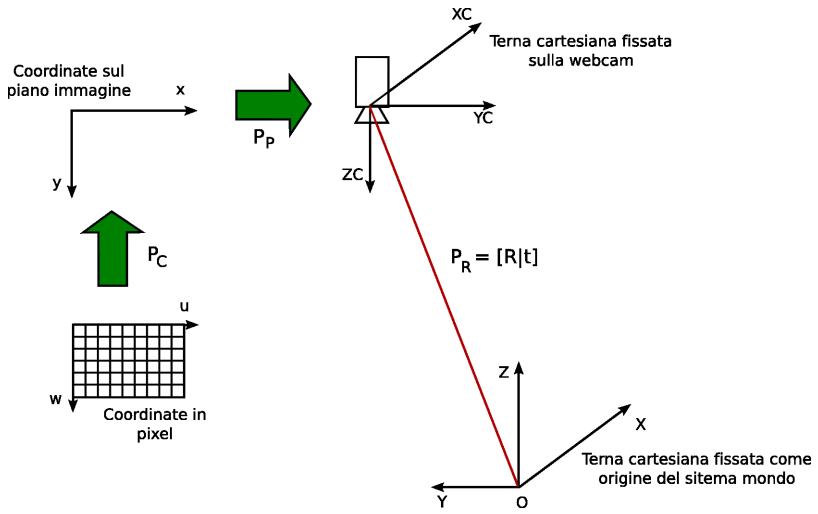


Figura 5.3: Trasformazioni fra sistemi di riferimento per la localizzazione mediante telecamera.

5.3 Visual Server

L'applicazione è stata realizzata usando la versione 4.0 della libreria Qt di Trolltech per Windows. Qt è una libreria multipiattaforma per lo sviluppo di programmi con interfaccia grafica tramite l'uso di widget. Qt usa una versione estesa del linguaggio C++, ma esistono interfacce per Python, C e Perl. Gira sulle piattaforme principali ed integra funzioni per il networking, l'accesso ai database SQL, parsing di documenti XML ed API multipiattaforma per l'uso dei file. Il server è composto da due thread: uno dedicato al riconoscimento dei veicoli e l'altro alla spedizione dei messaggi di localizzazione agli agenti. Questo per fare in modo che le operazioni specifiche di ciascun thread non rallentino il sistema complessivo se realizzate su un unico processo. Infatti sia il riconoscimento che lo stabilimento di una connessione richiedono dei tempi che se sommati possono creare di ritardi notevoli nell'invio della localizzazione. Questo si traduce sia in un aumento del periodo durante il quale il robot si trova in anello aperto sia nella mancata sincronia tra localizzazione e posizione reale del'agente. I due thread sono stati programmati secondo il modello produttore-consumatore (vedi figura 5.4).

Essi condividono una struttura dati in cui vengono salvate le informazioni

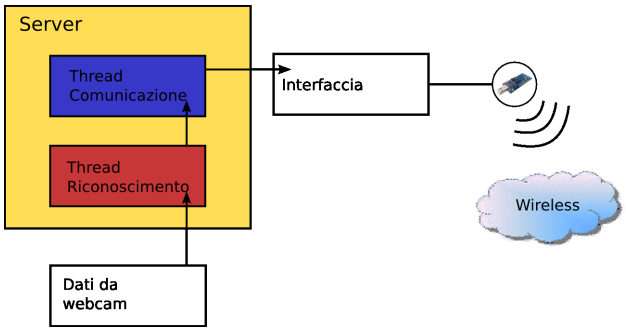


Figura 5.4: Architettura Visual Server.

riguardo il vettore di stato del veicolo (vedi Fig. 5.5), oltre a due campi che specificano se i dati sono validi e se riguardano lo stato attuale o il goal. Ogni veicolo ha un proprio spazio riservato all'interno della struttura dati. Tutte le volte che si ha un riconoscimento da parte del *server* questo entra nella sezione critica, se disponibile, e salva i dati appena descritti, anche sovrascrivendoli se il thread consumatore, che chiameremo *sender*, non ha avuto il tempo di inviarli. Questo per far si che al momento dell'invio le informazioni siano le più aggiornate e coerenti possibile. Se il *server* trova che il *sender* è bloccato in attesa di nuovi riconoscimenti lo sveglia. Una volta svegliato il *sender* prepara un messaggio per la rete con le nuove informazioni, lascia la sezione critica e si appresta a creare la connessione con i robot e a spedire il messaggio. Quindi entra di nuovo nella sezione critica per vedere se ci sono altri dati da spedire. Se si, procede come sopra altrimenti si blocca, in attesa di essere sbloccato dal *server*. La struttura dati con le informazioni riguardo le localizzazioni è considerato dal sender, ai fini dell'invio, come un array ciclico. Quando il sender entra nella sezione critica per effettuare l'invio dei dati controlla quali sono le informazioni disponibili a partire dall'elemento successivo a quello che è stato spedito per ultimo. Questo per evitare che si inneschi un meccanismo di priorità all'interno della struttura dati. Infatti se si procedesse ad inviare le informazioni scansionando l'array sempre con lo stesso ordine potrebbe accadere che riconoscimenti frequenti di un robot

inibiscano l'invio delle localizzazioni ai robot le cui informazioni sono posizionate più in basso nell'ordine della struttura dati.

Azioni ausiliare svolte dal server, ma per questo non meno importanti,

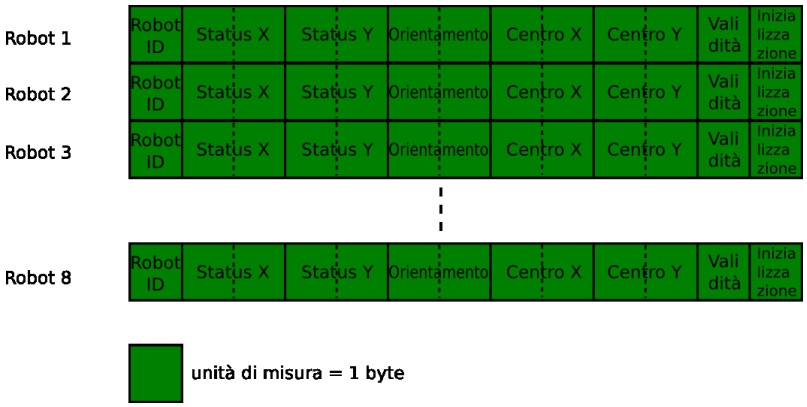


Figura 5.5: Struttura dati dove vengono salvate le informazioni riguardo le localizzazioni da parte del server.

sono l'acquisizione ed il rilascio della WebCam e la gestione del database dei robot. Queste azioni sono realizzate sfruttando le funzionalità messe a disposizione dalle Qt. Il database utilizizzato è di tipo *SQLite*. *SQLite* è una libreria scritta in C che implementa un motore SQL in meno di 250K incorporabile all'interno di applicazioni, funzionante senza nessuna installazione e/o configurazione. Per ogni robot le informazioni salvate nel database sono l'identificativo, l'indirizzo IP e le immagini del marker. Il processo di riconoscimento avviene mediante il confronto delle immagini acquisite dalla telecamera con quelle presenti nella basi di dati. In figura 5.6 possiamo vedere come si presenta l'interfaccia utente quando si fa partire l'applicazione. Quando l'applicazione parte si trova in attesa delle operazioni da svolgere che sono comandate mediante la pulsantiera posta sulla sinistra. Attraverso il bottone `init` il Visual Server esegue una serie di passi fondamentali per le altre operazioni quali l'ottenimento e l'inizializzazione della WebCam e del database. Con `Robot database` si accede alla GUI che permette la gestione del database dei robot. Per ogni robot viene salvato l'indice identificativo, l'indirizzo IP e la collezione di immagini campione del marker associato al

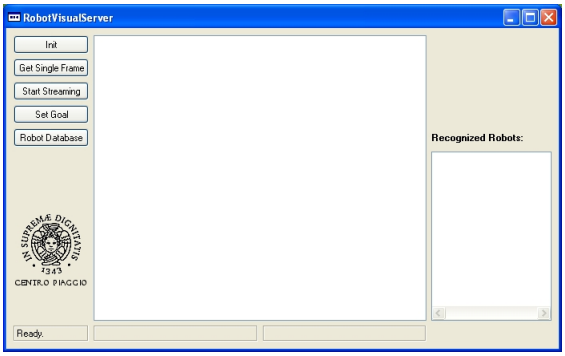
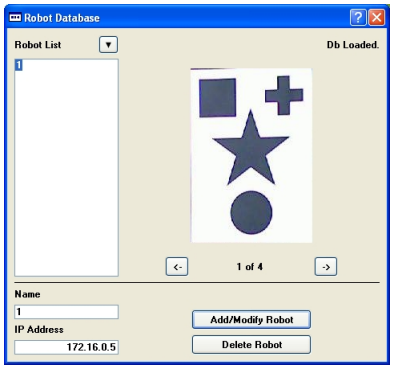
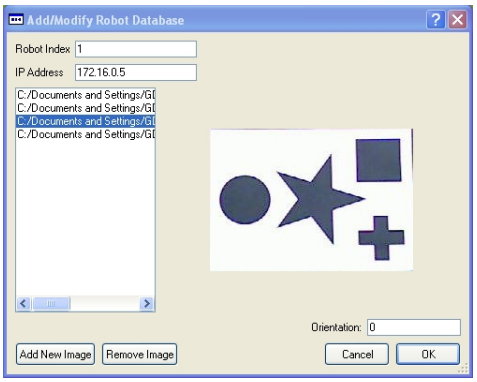


Figura 5.6: Interfaccia utente Visual Server.

robot ogniuna con un orientazione diversa. Nella GUI di figura 5.7(a) vengono mostrati i robot presenti nel database e le informazioni a loro associate. Mediante il tasto `Delete Robot` si elimina il robot selezionato dalla base di dati, con il tasto `Add/Modify Robot` si accede alla GUI che permette l'inserimento di tutti i dati relativi ad un nuovo robot(Fig. 5.7(b)). Il pulsante



(a) GUI per la visualizzazione del database



(b) GUI per la modifica del database

Figura 5.7: Gestione del database dei robot.

`Get Single Frame` permette di avere una istantanea di preview del campo di visione della telecamera che verrà visualizzata nel riquadro centrale. Con il pulsante `Set Goal` si forniscono i goal ai vari agenti mediante l'accesso alla GUI di figura 5.8. I goal dei veicoli vengono settati o mediante la foto degli agenti posti nella configurazione finale che dovranno raggiungere o

mediate l'inserimento delle coordinate della posizione finale espresse in pixel o cm. Infine, con il pulsante `Start Streaming` si da il via al processo di

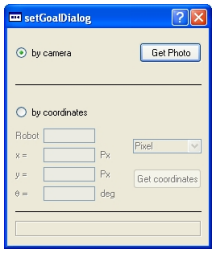


Figura 5.8: GUI per la regolazione dei goal.

riconoscimento e localizzazione del server. Nel riquadro di destra della finestra principale (Fig. 5.6) viene mostrata la lista dei robot riconosciuti con la relativa orientazione. Un esempio di riconoscimento è mostrato in figura 5.9. Una volta avviato lo streaming video il Visual Server compie ciclicamente

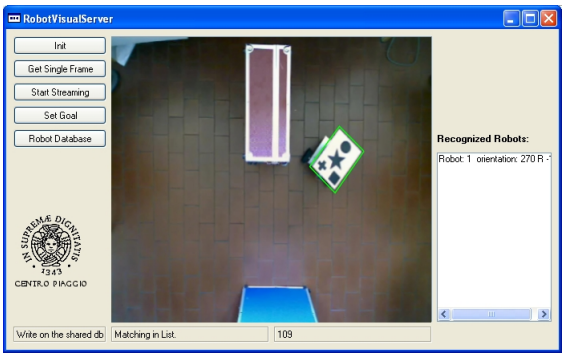


Figura 5.9: GUI per la regolazione dei goal.

due azioni, localizzazione dei robot e invio dei dati.

5.4 SIFT

SIFT è un algoritmo per la computer vision che estrae caratteristiche distintive da un'immagine, utilizzato per compiti come la corrispondenza fra differenti viste di un oggetto o di un scena(ad esempio stereo-visione) o il

riconoscimento di oggetti. L'identificazione mediante SIFT è robusta alle rotazioni, trasformazioni di scala e cambiamenti globali di illuminazione. L'immagine originale viene progressivamente sfocata attraverso un filtro Gaussiano con σ che varia da 1 a 2. Il risultato è una serie di immagini con grado di sfocatura diverso (uno scale-space prodotto dal filtraggio in cascata). Ogniuna di queste immagini viene poi sottratta rispetto alle vicine più strette (in base al σ) per produrre una nuova serie di immagini. I passi principali nell'estrazione delle caratteristiche da un'immagine sono:

1. Rilevamento degli estremi nello scale-space - un tipo specifico di blob detection dove ogni pixel nell'immagine è confrontato con gli otto più vicini e i nove pixel confrontati con quelle delle altre immagini della serie.

2. Localizzazione del punto chiave - i punti chiave vengono scelti fra gli estremi nello scale-space

3. Assegnamento dell'orientamento - per ogni punto chiave, in una finestra 16×16, vengono calcolati gli istogrammi del gradiente delle direzioni.

4. Descrittore del punto chiave - rappresentazione in un vettore a 128 dimensioni.

Grazie alle sue proprietà di robustezza nei confronti delle variazioni fra immagine campione e immagine corrente SIFT è molto usato in ambiti di rilevamento e descrizione. Esistono diversi tentativi di miglioramento delle SIFT che hanno avuto più o meno successo. Ad esempio Krystian Mikolajczyk ha presentato una variante a SIFT chiamata GLOH (Gradient Location and Orientation Histogram). É un descrittore simile a SIFT che considera più regioni spaziali per gli istogrammi. L'alta dimensione del descrittore è ridotta a 64 mediante l'analisi dei componenti principali (principal components analysis PCA). Un differente schema di matching chiamato SURF (Speeded Up Robust Features) è stato presentato da Herbert Bay. La versione standard di SURF è più veloce di SIFT e, sostiene l'autore, più robusta rispetto

alle trasformazioni dell'immagine. Nonostante tutto SIFT continua ad essere l'algoritmo più diffusamente usato.
Per ottenere le migliori prestazioni, abbiamo adottato una versione commerciale di SIFT chiamata ERSP,di Evolution Robotics, una libreria che garantisce performance di circa 10 frames per secondo su una macchina Pentium. Le tecnologie chiave di ERSP sono:

ViPR™ - É lo standard nel software per il Visual Pattern Recognition e fornisce un certo riconoscimento di pattern, dettagli e marker in ambienti realistici.

vSLAM® - É un sistema per la mappatura e localizzazione visuale che rende il robot capace di utilizzare solo una camera insieme agli encoder delle ruote per stimare la propria posizione nell'ambiente con circa 10cm di accuratezza.

ERSA™ - L'Evolution Robotics Software Architecture è un sistema operativo specifico per robot che fornisce utilità per la gestione del'hardware del robot e dei componenti software.

La tecnologia utilizzata sulla nostra piattaforma è esclusivamente la ViPR. ViPR analizza una immagine per ricavarne le feature che vengono immagazzinate come modello di addestramento in un database. Per il riconoscimento, questa tecnologia confronta le feature prese dall'immagine catturata dalla telecamera e le confronta con quelle presenti nel database per trovare la migliore corrispondenza basandosi su un meccanismo di voto. Verrà così trovata l'esatta corrispondenza utilizzando poche feature. Questo da la possibilità a ViPR di superare occlusioni significative.

Capitolo 6

Test e conclusioni

6.1 Test della piattaforma

Sono stati fatti diversi esperimenti per testare la piattaforma hardware e software realizzata. Si è posta la WebCam ad un altezza di 2 m dal suolo con una risoluzione di 640 × 480 pixel. Questa risoluzione risulta essere un buon compromesso fra qualità dell'immagine e velocità di esecuzione delle SIFT per il riconoscimento degli oggetti. Risoluzioni minori producono immagini la cui qualità è tale da non poter distinguere i simboli sui marker. Risoluzioni maggiori rendono il processo di riconoscimento così lento da rendere inconsistenti la localizzazione del server con la posizione attuale del veicolo. La velocità di avanzamento dell'agente è stata posta pari a 0.3 m/s, il raggio R_C uguale a 15 cm. Le costanti k_r per la valutazione della forza repulsiva (eq. 2.6) hanno valori differenti sui diversi sensori ad infrarossi. Questo per cercare di salvaguardare, anche con la politica del potenziale artificiale, il disco riservato dell'agente. Poiché, per la GRP, il centro del disco riservato si trova alla destra del veicolo, se ci si pone alle sue spalle, i sensori hanno k_r crescenti da sinistra verso destra. La figura 6.1 riporta uno dei test eseguiti. La "X" verde indica l'obiettivo che deve essere raggiunto dal veicolo che deve riuscire a raggiungerlo passando attraverso un varco formato da due ostacoli. Le immagini di figura 6.1 sono state riprese con la stessa angolazione e dalla

stessa distanza di quelle riprese dalla WebCam del visual server. In maniera asincrona, ad ogni riconoscimento del marker, il server invia la localizzazione all'agente che tenta di raggiungere l'obiettivo seguendo una linea retta. Ogni avvicinamento agli ostacoli provoca un cambio di traiettoria. Come si può notare dalla figura il robot riesce a trovare il varco tra i due ostacoli e a raggiungere il goal. Quello che si nota nell'esecuzione dei test è l'andamento a

Figura 6.1: Test del RaIN2 in ambiente con presenza di ostacoli. L'ordine della sequenza temporale delle immagini è da sinistra verso destra e dall'alto verso il basso.

scatti del robot dovuto alla discretizzazione dei risultati dell'APF per essere adattati alla GRP. Infatti è stato fatto in modo che l'APF ne riproduca gli stati. Nell'esecuzione della politica del potenziale artificiale i passaggi fra uno stato e l'altro sono molto rapidi a causa della rapidità con cui varia il vettore della forza risultante. Quello che ne deriva è il comportamento a scatti dell'agente.
Altro fatto emerso dalla simulazioni è difficoltà delle SIFT a generare un elevato numero di riconoscimenti del robot in ambienti con variazioni relative di luminosità. Se l'area inquadrata dalla telecamera presenta dei passaggi di

luminosità (luce-ombra) la capacità di riconoscimento ne risulta deteriorata. Questo può produrre dei problemi al robot in quanto cala il numero di campioni sul quale applicare l'algoritmo di controllo (si veda paragrafo 5.4). Il test presentato in figura 6.2 mostra gli effetti dei mancati riconoscimenti. Nel frame in basso a sinistra si può notare come l'agente stia passando sul goal ma non si arresti a causa della mancata localizzazione. Quindi è costretto ad una rotazione supplementare per recuperare l'obiettivo. Presentiamo infine

Figura 6.2: Test del RaIN2 in ambiente con presenza di ostacoli. L'ordine della sequenza temporale delle immagini è da sinistra verso destra e dall'alto verso il basso.

un ultimo test (Fig. 6.3), dove, grazie un sufficiente numero di localizzazioni, si assiste ad un comportamento dell'agente molto più "lineare".

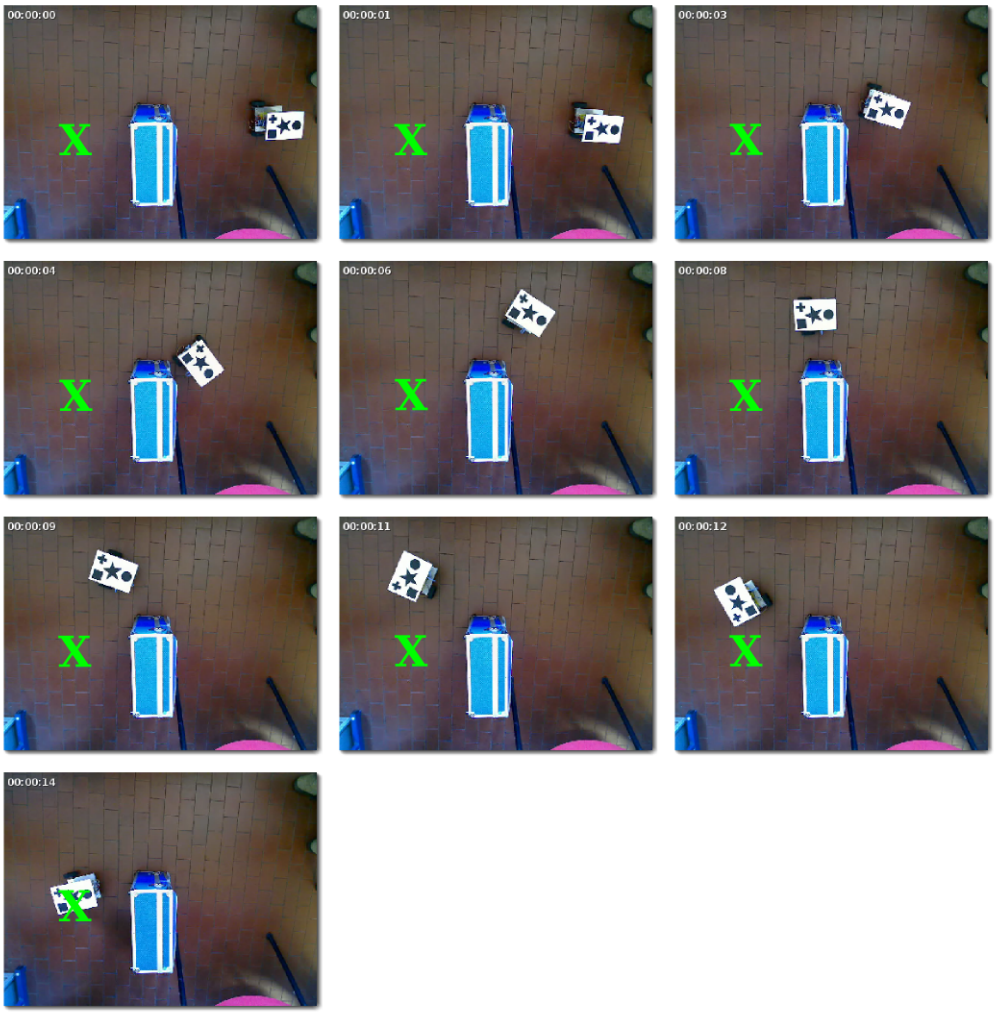


Figura 6.3: Test del RaIN2 in ambiente con presenza di ostacoli. L'ordine della sequenza temporale delle immagini è da sinistra verso destra e dall'alto verso il basso.

L'utilizzo di più WebCam situate non molto in alto rispetto al piano dove si muovo gli agenti potrebbe essere una soluzione per cercare di aumentare i riconoscimenti da parte del server.Infatti in questo modo si ridurebbero le variazioni relative di luminosità nelle aree inquadrate e si aumenterebbe il numero di feature che possono essere catturate dalle WebCam per effettuare il riconoscimento.

6.2 Conclusioni

In questo lavoro è stata studiata una infrastruttura a componenti per il controllo di sistemi di agenti mobili. Grazie al ridotto utilizzo di memoria e CPU il sistema operativo Contiki ed il CRTK sono stati utilizzati per importare il modello a componenti sui sistemi embedded. I componenti sono stati realizzati come degli stubs in modo dal alleggerire il carico computazionale sulla TMote Sky. I componenti realizzati sono stati implementati sui microcontrollori rendendo modulare anche l'architettura hardware dei robot.
Come politica di controllo dei robot è stata utilizzata l'integrazione fra la Generalized Roundabout Policy e l'Artificial Potential Field. La politica, decentralizzata e real-time, è di facile realizzazione e risulta idonea per robot provvisti di sensori non sofisticati.
Un server di visione basato su SIFT fornisce la localizzazione degli agenti nell'area di lavoro.

6.3 Sviluppi futuri

Gli sviluppi di questo lavoro di tesi posso essere indirizzati nell'estendere la politica di controllo a robot di natura e forma diversa che si muovo in un ambiente con ostacoli di forma generica. Ciò per fare in modo che i robot siano in grado di compiere il proprio task in un ambiente che sia il più generico possibile. I robot possono essere muniti, grazie all'architettura modulare sviluppata, di nuovi moduli hardware forniti di sensori, per permettere la localizzazione dell'agente senza l'utilizzo del server di visione, e di microcontrollori, per gestire la comunicazione con la TMote Sky e svolgere le funzionalità relative alla localizzazione. Questo porterebbe ad una decentralizzazione maggiore della piattaforma rendendo ancora più autonomi gli agenti. Una soluzione alternativa potrebbe essere l'utilizzo dei telefoni cellulari di ultima generazione al posto delle TMote Sky. Con la telecamera che molti di essi incorporano si potrebbero implementare tecniche di *visual servoing* per il controllo del robot. Utilizzando sempre delle tecnologie a

componenti si potrebbero sviluppare dei software per il controllo del robot in modo da aggiungere alle diverse funzioni di un telefono cellulare anche quella di controllore dell'agente.

Appendice A

Codice

In questo appendice viene presentato il codice sviluppato per la realizzazione dell'infrastruttura a componenti per il controllo di sistemi di agenti mobili.

A.1 Codice dei componenti CRTK

A.1.1 Collision-Obstacle Avoidance Component

```
/*************************************************************
* File icoac.h
* Dichirazione delle funzioni che implementano l'interfaccia
* del Collision-Obstacle Avoidance Component
*/

#ifndef __ICOAC_H__
#define __ICOAC_H__
#include "crtk.h"

DECLARE_INTERFACE(icoac,
{
  void (* COAC_initParam) (unsigned int8 IdBot, char numNeigh);
  void (* COAC_start) (int8 *commad);
```

```
  void (* COAC_setGoalInt) (unsigned int8 status[],
        char threshold, short len);
  void (* COAC_robPositionInt) (unsigned int8 idBot,
unsigned int8 status[], short len);
  void (* COAC_robotIdentification)(unsigned int8 *botID);
  void (* COAC_robotRecognized)(unsigned int8 *botREC);
});
#endif /* ICOAC_H */

/*********************************************************
* File coac.h
* Dichirazione delle funzioni del
*  Collision-Obstacle Avoidance Component
*/

#ifndef __COAC_H__
#define __COAC_H__
#include "crtk.h"

DECLARE_COMPONENT(coac,
{
  void (* COAC_initParam) (unsigned int8 IdBot, char numNeigh);
  void (* COAC_resetParam)();
  void (* i2cSend) (unsigned int8 *buffer, int8 address,
        int8 len, char startPos);
  void (* i2cReceive) (unsigned int8 *buffer, int8 address,
        int8 len);
  void (* COAC_start) (int8 *commad);
  void (* add_to_buff)(int8* buff, int8* bytes, int size,
        int startPos);
  void (* COAC_setGoalInt) (unsigned int8 status[],
        int8 threshold, short len);
```

```
  void (* COAC_robPositionInt) (unsigned int8 idBot,
        unsigned int8 status[], short len);
  void (* COAC_robotIdentification)(unsigned int8 *botID);
  void (* COAC_robotRecognized)(unsigned int8 *botREC);
});
#endif /* COAC_H */
```

Si può notare come nel componente siano presenti delle funzioni interne utilizzate solo dal componente stesso e non visibili dall'esterno.

```
/*************************************************************
* File coac.c
* Implementazione del corpo del
* Collision-Obstacle Avoidance Component
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "contiki.h"
#include "contiki-net.h"

#include "dev/leds.h"
#include "dev/i2c.h"

#include "cac.h"
#include "dev/i2ccom.h"

// In order to see debug messages put x in the follow line,
//(i.d. #define DBG(x)    x)
#define DBG(x)
```

```
///---------LOCALS
//table where to save the coordinates (x,y) of the centers of
//recognized robot (max 8)
static unsigned short *robot_information;
//variable that contains the ID of this robot
static unsigned int8 robot_ID;
//flag's variable that it indicates which robots have been
//recognized
static unsigned int8 recognized;
//flag's variable that it indicates which robots are in the
//neighbourhoods
static unsigned int8 neighbours;
//array where to save the message with informations about this
//robot (SX; SY; THETA)
static unsigned int8 status_robot[6];
//array where to save the message with informations about this
//robot (SX; SY; THETA) in pixel
static unsigned int8 status_robot_px[6];
//array where to save the message with informations about goal
//of this robot (GX; GY; GTHETA)
static unsigned int8 goal_robot[6];
//variable where save the goal's threshold
static unsigned int8 goalThreshold;
static unsigned long nop;

static void construct()
{
  robot_information = NULL;
  robot_ID = 0x00;
  recognized = 0x00;
  neighbours = 0x00;
```

```
  goalThreshold = 0x00;
  nop = 0;
  int i;
  for (i = 0; i < 6 ; i++)
  {
    status_robot[i] = 0x00;
    status_robot_px[i] = 0x00;
    goal_robot[i] = 0x00;
  }
  printf("Collision Avoidance Component Instantiated..\n");
}

//function in order to initalize the variables for CAC
static void CAC_initParam(unsigned int8 IdBot, int8 numRobot)
{
  if((numRobot > 0 && numRobot < 9) && (IdBot > 0 &&
        IdBot <= numRobot))
  {
    robot_ID = IdBot;
    recognized = 0x00;
    neighbours = 0x00;
    robot_information = (unsigned short*)
        malloc (sizeof(unsigned short)* 2*numRobot);
    //numRobot is multiplied for 2 because
    //it has to save x and y components
  }
}

//function in order to reset the variables for CAC
static void CAC_resetParam()
{
```

```
  robot_ID = 0;
  recognized = 0x00;
  neighbours = 0x00;
  free (robot_information);
}

static void add_to_buff(int8* buff, int8* bytes, int size,
        int startPos)
{
    int i;
    i = 0;
    while(i < size)
    {
        buff[startPos+i]=bytes[i+1];
        i += 2;
    }
    i = 1;
    while(i < size)
    {
      buff[startPos+i]=bytes[i-1];
      i += 2;
    }
}

//functions in order to send and receive on i2c
static void i2cSend (unsigned int8 *buffer, int8 address,
        int8 len, int8 startPos)
{
  I2C_ENABLE();
  if(i2c_start() == -1)
  {
```

```
    DBG(printf("i2c_start() failed\n");)
  }
  else
  {
    int i;
    i2c_write(address);
    for (i = 0; i < len; i++)
      i2c_write(buffer[i+startPos]);
      i2c_stop();
    DBG(for (i = 0; i < len; i++)
      printf (" 0x%02x",buffer[i+startPos]);)
    DBG(printf("\n sent data\n");)
  }
  I2C_DISABLE();
}

static void i2cReceive (unsigned int8 *buffer, int8 address,
        int8 len)
{
  I2C_ENABLE();
  if(i2c_start() == -1)
  {
    DBG(printf("i2c_start() failed\n");)
  }
  else
  {
    int i;
    i2c_write(address);
    for(i = 0; i < len-1; i++)
      buffer[i] = i2c_read(1);
    buffer[len-1] = i2c_read(0);
```

```
    i2c_stop();
  }
  I2C_DISABLE();
}

//function in order to save the status of current robot
//and the center of the other robots
static void CAC_robPositionInt (unsigned int8 idBot,
unsigned int8 status[],short len)
{
  if(idBot > 0 && idBot < 9)
  {
    int8 flag = 0x00;
    *(robot_information+((idBot-1)*2)) =
        ((unsigned short)status[8])*256 +
                (unsigned short)status[9];
    *(robot_information+((idBot-1)*2+1)) =
        ((unsigned short)status[10])*256 +
        (unsigned short)status[11];

    flag = 0x01;
    flag <<= (idBot-1);
    recognized |= flag;
    if(idBot == robot_ID)
    {
      int i;
      int j;
      unsigned short tmp[2];
      for(i = 0;i<len-6;i++)
        status_robot_px[i] = status[i+2];
      i = 0;
```

```
      j = 0;
      while (i < 4)
      {
        tmp[j] = ((((unsigned short)status_robot_px[i])*256 +
(unsigned short)status_robot_px[i+1])* CAMPO_VISIVO_MAX)/
RESOLUTION_X;
        i += 2;
j++;
      }
      add_to_buff(status_robot, (int8*)tmp, 4, 0);
      status_robot[4] = status_robot_px[4];
      status_robot[5] = status_robot_px[5];
    }
  }
}

//function in order to save the goal(x,y,theta) of current
//robot and the threshold for the goal
static void CAC_setGoalInt (unsigned int8 status[],
        int8 threshold, short len)
{
  int j = 0;
  int i = 0;
  unsigned short tmp[3];
  int8 mess[len+2];
  while(i < 4)
  {
    tmp[j] = ((((unsigned short)status[i+1])*256 +
        (unsigned short)status[i+2])*CAMPO_VISIVO_MAX)/
                RESOLUTION_X;
    j++;
```

```
    i += 2;
  }
  tmp[2] = (((unsigned short)status[5])*256 +
        (unsigned short)status[6]);
  add_to_buff(goal_robot, (int8*)tmp, 6, 0);
  goalThreshold = threshold;
  mess[0] = I2C_START_BYTE;
  for (i = 0; i < len; i++)
    mess[i+1] = status[i];
  mess[len+1] = I2C_STOP_BYTE;
  i2cSend(mess, CAC_COMP_ADDR_WRITE, len+2, 0);
  //because len is the lenght of the message and 2 are the
  //start and the stop byte.(len + 2) =
  //StartByte,TypeMess, Sx, Sy, Theta, Cx, Cy, StopByte
}

//function in order to implement the CAC
static void CAC_start (int8 *command)
{
  unsigned int8 *messForComponent;
  int i;
  int j;
  int8 flag;
  int x_difference;
  int y_difference;
  int threshSquare;
  int8 count_neigh;
  unsigned int8 outBuffer[6];
  int8 commandToMotor;
  unsigned int8 commandFromCAC[1];
  short tmp = 0;
```

```
//calculating the distance from the goal
x_difference =
  (((int)status_robot[0])*256 + (int)status_robot[1]) -
      (((int)goal_robot[0])*256 + (int)goal_robot[1]);
y_difference =
  (((int)status_robot[2])*256 + (int)status_robot[3]) -
      (((int)goal_robot[2])*256 + (int)goal_robot[3]);
threshSquare = (int)goalThreshold * (int)goalThreshold;
if(((x_difference * x_difference) +
      (y_difference * y_difference)) <
               threshSquare)
  commandToMotor = GOAL_REACHED;
else
{
  ///-------begin composition and sending message to OAC
  messForComponent = (unsigned int8*)
       malloc(sizeof(unsigned int8)*10);
  //10 because the buffer is composed by: xHI, xLO, yHI, yLO,
  //thetaHI, thetaLO, xGoalHI, xGoalLO, yGoalHI, yGoalLO
  for(i = 0; i < 6; i++)
    messForComponent[i] = status_robot[i];
  for(i = 6; i < 10; i++)
    messForComponent[i] = goal_robot[i-6];
  DBG(printf("messForComponent = ");
     for(i = 0; i < 10;i++)
        printf("0x%02x ",*(messForComponent+i));
     printf("\n");)

  i2cSend(messForComponent, OAC_COMP_ADDR_WRITE, 10, 0);
  free(messForComponent);
```

```
///-------end composition and sending message to OAC

///-------begin neighbours count
i = 0;
flag = 0x01;
//it calculates if the status of neighbourhood are changed
while(flag != 0)
{
  if((recognized & flag) != 0  && i != robot_ID-1)
  {
    x_difference = (int)
         (*(robot_information+((robot_ID-1)*2))) -
              (int)(*(robot_information+(i*2)));
    y_difference = (int)
         (*(robot_information+((robot_ID-1)*2)+1)) -
               (int)(*(robot_information+(i*2)+1));
    threshSquare = (int)(RS*RS);
    if(((x_difference*x_difference)+
              (y_difference*y_difference)) < threshSquare)
    {
      neighbours |= flag;
      DBG(printf("Robot %d: add neighbour(robot %d)\n",
              robot_ID,i);)
    }
    else
    {
      neighbours &= ~flag;
      DBG(printf("Robot %d: remove neigbour(robot %d)\n",
              robot_ID,i);)
    }
  }
```

```
    flag <<= 0x01;
    i++;
  }
  count_neigh = 0x00;
  flag = 0x01;
  while(flag != 0)
  {
    if(neighbours & flag)
      count_neigh++;
      flag <<= 1;
  }
 ///-------end neighbours count

///-------begin composition and sending message to CAC
  messForComponent = (unsigned int8*)
              malloc(sizeof(int8)*(4*count_neigh+14));
  // the buffer is mode of 14 fixed elements which are
  //STARTBYTE, Type Mess, xHI, xLO, yHI, yLO, thetaHI,
  //thetaLO, CxHI, CxLO, CyHI, CyLO, numVicini, STOPBYTE,
  //plus 4 bytes (CxHI, CxLO, CyHI, CyLO), for each neighbour
  messForComponent[0] = I2C_START_BYTE;
  messForComponent[1] = I2C_MSG_TYPE_STATUS;
  for(i = 0; i < 6; i++)
    messForComponent[i+2] = status_robot_px[i];
  add_to_buff(messForComponent,
       (int8*)(robot_information+((robot_ID-1)*2)), 4, 8);
  messForComponent[12] = count_neigh;
  i = 0;
  j = 0;
  flag = 0x01;
  while(flag != 0)
```

```
    {
      if((neighbours & flag) != 0)
      {
        add_to_buff(messForComponent,
                (int8*)(robot_information+(i*2)), 4, (4*j)+13);
        j++;
      }
      flag <<= 0x01;
      i++;
    }
    messForComponent[(4*count_neigh+13)] = I2C_STOP_BYTE;
    i2cSend (messForComponent, CAC_COMP_ADDR_WRITE,
         (4*count_neigh+14), 0);
    free(messForComponent);
  ///-------end composition and sending message to CAC
  for(nop = 0;nop < 10000;nop++);

  ///-------begin receiving component
   i2cReceive (outBuffer, OAC_COMP_ADDR_READ, 6);
   DBG(for(i = 0; i< 9;i++)
    printf("outBuffer[%d] = %d\n",i,outBuffer[i]);)
   for(nop = 0;nop < 10000;nop++);
   i2cReceive (commandFromCAC, CAC_COMP_ADDR_READ, 1);

///-------end receiving component

   ///-------begin navigation without neighbours
   if(count_neigh == 0 || commandFromCAC[0] == STRAIGHT)
   {
     commandToMotor = outBuffer[5];
   }
```

```
///-------end navigation without neighbours

///-------begin navigation with neighbours
else
{
  switch (commandFromCAC[0])
  {
    case HOLD:
      commandToMotor = HOLD;
      if(outBuffer[2] < THRESHOLD_S3 ||
              outBuffer[3] < THRESHOLD_S4 ||
                      outBuffer[4] < THRESHOLD_S5)
        commandToMotor = HALT;
    break;

    case ROLL:
      commandToMotor = ROLL;
      if(outBuffer[0] < THRESHOLD_S1 ||
             outBuffer[1] < THRESHOLD_S2 ||
                      outBuffer[2] < THRESHOLD_S3)
        commandToMotor = HALT;
    break;

    case ROLL2:
    commandToMotor = ROLL2;
    if(outBuffer[0] < THRESHOLD_S1 ||
            outBuffer[1] < THRESHOLD_S2 ||
                     outBuffer[2] < THRESHOLD_S3)
      commandToMotor = HALT;
    break;
```

```
        default:
          commandToMotor = HALT;
        break;
      }
    }
  }
  *command = commandToMotor;
}

unsigned int8 CAC_robotIdentification(unsigned int8 *botID)
{
  *botID = robot_ID;
}

unsigned int8 CAC_robotRecognized(unsigned int8 *botREC)
{
  *botREC = recognized;
}

static void destroy() {
  CAC_resetParam();
  printf("Collision Avoidance Component Going Away...\n");
}

/*
 * Component definition
 */
IMPLEMENT_COMPONENT(cac,{CAC_initParam,CAC_start,
    CAC_setGoalInt, CAC_robPositionInt,
          CAC_robotIdentification,CAC_robotRecognized});
```

A.1.2 Motor Component

```
/*************************************************************
* File imotor.h
* Dichirazione delle funzioni che implementano l'interfaccia
* del Motor Component
*/
#ifndef __IMOTOR_H__
#define __IMOTOR_H__
#include "crtk.h"

DECLARE_INTERFACE(imotor,
{
  void (* commandToMotor) (int8 command,float u[]);
});
#endif /* IMOTOR_H */

/*************************************************************
* File motor.h
* Dichirazione delle funzioni del
* Motor Component
*/
#ifndef __MOTOR_H__
#define __MOTOR_H__
#include "crtk.h"

DECLARE_COMPONENT(motor,
{
  void (* commandToMotor) (int8 command,float u[]);
});
```

```
#endif /* MOTOR_H */

/*************************************************************
* File motor.c
* Implementazione del corpo del
* Motor Component
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "contiki.h"
#include "contiki-net.h"

#include "dev/leds.h"
#include "dev/i2c.h"

#include "motor.h"
#include "dev/i2ccom.h"

// In order to see debug messages put x in the follow line,
//(i.d. #define DBG(x)    x)
#define DBG(x)


//function in order to send command to motor
static void commandToMotor (int8 command,float u[])
{
  I2C_ENABLE();
  if (i2c_start() == -1)
  {
```

```
  DBG(printf("i2c_start() failed in sending command
       to motor\n");)
}
else
{
  i2c_write(MOTOR_COMP_ADDR_WRITE);
  i2c_write(I2C_START_BYTE);
  i2c_write(I2C_MSG_TYPE_REMOTE);
  switch (command)
  {
    case STRAIGHT:
      leds_toggle(LEDS_RED | LEDS_BLUE);
      DBG(printf("Motion Set Straight\n");)
      i2c_write(V_STRAIGHT);// DX
      i2c_write(V_STRAIGHT);// SX
      leds_toggle(LEDS_RED | LEDS_BLUE);
      u[0] = V_STRAIGHT;
      u[1] = V_STRAIGHT;
    break;

    case HOLD:
      DBG(printf("Motion Set Hold\n");)
      i2c_write(V_HOLD);//DX;
      i2c_write(STOP_WHEEL);//SX;
      u[0] = V_HOLD;
      u[1] = STOP_WHEEL;
    break;

    case ROLL:
      leds_toggle(LEDS_RED);
      DBG(printf("Motion Set Roll\n");)
```

```
  i2c_write(V_STRAIGHT/3);//DX;
  i2c_write(2*V_STRAIGHT/3);//SX;
  leds_toggle(LEDS_RED);
  u[0] = V_STRAIGHT/3;
  u[1] = 2*V_STRAIGHT/3;
break;

case ROLL2:
  leds_toggle(LEDS_BLUE);
  DBG(printf("Motion Set Roll2\n");)
  i2c_write(STOP_WHEEL);//DX;
  i2c_write(V_HOLD);//SX;
  leds_toggle(LEDS_BLUE);
  u[0] = V_STRAIGHT;
  u[1] = V_HOLD;
break;

case GOAL_REACHED:
  DBG(printf("Motion Set Goal_reached\n");)
  i2c_write(STOP_WHEEL);//DX;
  i2c_write(STOP_WHEEL);//SX;
  u[0] = STOP_WHEEL;
  u[1] = STOP_WHEEL;
break;

default:
  DBG(printf("No data on i2c\n");)
  i2c_write(STOP_WHEEL);//DX;
  i2c_write(STOP_WHEEL);//SX;
  u[0] = STOP_WHEEL;
  u[1] = STOP_WHEEL;
```

```
      break;
    }
    i2c_write(I2C_STOP_BYTE);
    i2c_stop();
    I2C_DISABLE();
  }
}

IMPLEMENT_COMPONENT(motor,{commandToMotor});
```

A.1.3 Main Process

Di seguito riportiamo il codice del processo principale che attende i le localizzazioni provenienti dal visual server e richiede i servizi ai componenti.

```
/*************************************************************
* File pisa.c
* Dichiarazione del processo PUDP
*/
#ifndef PISA_H
#define PISA_H

PROCESS_NAME(pudp_process);

#endif /* PISA_H */

/*************************************************************
* File pisa.c
* Implementazione del corpo del processo PUDP
*/

#include <stdio.h>
#include <math.h>
```

```
#include "contiki.h"
#include "contiki-net.h"

#include "dev/leds.h"

#include "dev/i2c.h"
#include "dev/i2ccom.h"

#include "pisa.h"
#include "icac.h"
#include "imotor.h"

#define DBG(x)

PROCESS(pudp_process, "pudp process");

PROCESS_THREAD(pudp_process, ev, data)
{
   static struct uip_udp_conn *c;
   //variable in order to count every how many messages
   //to reset the array recognized
   static unsigned short countMes = 0x00;

  static Receptacle rec;
  static Receptacle rec1;

  PROCESS_EXITHANDLER(goto exit);
  PROCESS_BEGIN();
```

```
unsigned command = 0x00;
unsigned char recognizedBots;
unsigned char flag;
unsigned char robotInMess;
unsigned char bot;

float test[2];
short i;

INVOKE("crtk",crtk,connect("cac",&rec));
INVOKE("crtk",crtk,connect("motor",&rec1));

INVOKE(rec,icac,CAC_initParam(0x01,8));
printf("pudp_process starting\n");

{
  uip_ipaddr_t any;
  uip_ipaddr(&any, 0,0,0,0);
  c = udp_new(&any, HTONS(0), NULL);
  uip_udp_bind(c, HTONS(1010));
}
while(1)
{
  while(1)
  {
    PROCESS_YIELD(); //it waits for new message


    if(ev == tcpip_event && uip_newdata())
    {
      u8_t *src = ((struct uip_udpip_hdr *)
```

```
            &uip_buf[UIP_LLH_LEN])->srcipaddr.u8;
       printf("%d.%d.%d.%d: %d \n",
      src[0], src[1], src[2], src[3], uip_len);
       unsigned char *data = uip_appdata;
       //it saves the ID of robot,that is in the message,
       //in robotInMess
       robotInMess = data[0];
       printf("Messaggio dal robot %d \n",robotInMess);
       countMes = (countMes+1)%500;
       if(countMes == 0x00)
         recognizedBots = 0x00;
       for(i = 0; i< uip_len;i++)
         printf(" 0x%02x",data[i]);
       printf("\n");

       ///--------SENDING FINAL STATUS FOR THE CURRENT ROBOT
       INVOKE(rec,icac,CAC_robotIdentification(&bot));
       //it controls if the message is of initialization for
       //this robot
       if(robotInMess == bot &&
                data[1] == I2C_MSG_TYPE_FIN*AL_STATUS)
       {
INVOKE(rec,icac,CAC_setGoalInt ((data+1), 10, uip_len-1));
     }
     ///-----END SENDING FINAL STATUS FOR THE CURRENT ROBOT

     ///--------------------------HANDLING STATUS MESSAGES
     //it controls that the message isn't of initialization
     else if(data[1] != I2C_MSG_TYPE_FINAL_STATUS)
     {
        INVOKE(rec,icac,CAC_robPositionInt
```

```
                (robotInMess, data,uip_len));
          INVOKE(rec,icac,CAC_robotRecognized
                (&recognizedBots));
          flag = 0x01;
          flag <<= (bot-1);
          //it controls if this robot has been recognized
          if((recognizedBots & flag) != 0)
          {
            INVOKE(rec,icac,CAC_start(&command));
            printf("before call commandToMotor\n");
            INVOKE(rec1,imotor,commandToMotor(command,test));
  }
        }
          ///-------------------END NAVIGATION WITH NEIGHBOURS
      }
        ///-----------------------END HANDLING STATUS MESSAGES
    }
  }

  exit:
   /* Contiki does automatic garbage collection of uIP state
    * and we need not worry about that. */

  printf("pudp_process exiting\n");
  PROCESS_END();
}
```

A.2 Codice microcontrollori

A.2.1 GRP component

```
//-------------------------------------------------------------
```

```
// C main line
//-------------------------------------------------------------

// part specific constants and macros
#include <m8c.h>
// PSoC API definitions for all User Modules
#include "PSoCAPI.h"
#include "algoritmo.h"
#include "globals.h"
#include <math.h>
#include "i2ccom.h"

// Global Variables
BYTE abBuffer[I2C_BUFFER_LENGTH];
BYTE outBuffer[I2C_OUTBUFFER_LENGTH];
BYTE status;  ///!!! BYTE!!!

void main(){

/* Start the slave and wait for the master */
    I2CHW_Start();
    I2CHW_EnableSlave();
    I2CHW_DisableMstr();

    INT_MSK0 = 32;

/* Start the slave and wait for the master */
    I2CHW_Start();
    I2CHW_EnableSlave();
    I2CHW_DisableMstr();
```

```
/* Enable the global and local interrupts */
    M8C_EnableGInt;
    I2CHW_EnableInt();

/* Setup the Read and Write Buffer - set to the same buffer */
    I2CHW_InitRamRead(outBuffer,I2C_OUTBUFFER_LENGTH);
    I2CHW_InitWrite(abBuffer,I2C_BUFFER_LENGTH);
    I2CHW_ClrWrStatus();
    I2CHW_ClrRdStatus();

    // Init I2C out Buffer
    outBuffer[0]=0x00;

    // Hybrid automaton state initialization
    Control = HOLD;

    /* Read commands */
    while(1){
      while(1){
        // Wait for completion of writing on I2C
        status = I2CHW_bReadI2CStatus();

        if( status & I2CHW_WR_COMPLETE){

          // Enter if received packet...

          /* Data received - clear the Write status */
          I2CHW_ClrWrStatus();
          /* Reset the pointer for the next read data */
          I2CHW_InitWrite(abBuffer,I2C_BUFFER_LENGTH);
```

```
// Reject a paket if the first byte is not
//I2C_START_BYTE
if (abBuffer[0]!=I2C_START_BYTE)
  break;

switch (abBuffer[1]){

  case I2C_MSG_TYPE_STATUS:
    // Refresh vehicle status
    // If the vehicle has reached the goal
    //already, it dosn't evaluate the
    //algorithm until a new goal is set.
    if (Control == GOAL_REACHED)
    {
      outBuffer[0]=GOAL_REACHED;
    }
    else
    {
      // Copying data
      Status.x = abBuffer[2];
      Status.x = (Status.x*256)+(short)abBuffer[3];
      Status.y = abBuffer[4];
      Status.y = (Status.y*256)+(short)abBuffer[5];
      Status.theta = abBuffer[6];
      //From degree to rad
      Status.theta = (Status.theta*256+
          (float)abBuffer[7])*0.01744444;
      C1x = abBuffer[8];
      C1x = (C1x*256)+(short)abBuffer[9];
      C1y = abBuffer[10];
      C1y = (C1y*256)+(short)abBuffer[11];
```

```
                C2x = C3x = C2y = C3y =0;
                if(abBuffer[12] >= 1)
                {
                  C2x = abBuffer[13];
                  C2x = (C2x*256)+(short)abBuffer[14];
                  C2y = abBuffer[15];
                  C2y = (C2y*256)+(short)abBuffer[16];
                }

                if(abBuffer[12] == 2)
                {
                  C3x = abBuffer[17];
                  C3x = (C3x*256)+(short)abBuffer[18];
                  C3y = abBuffer[19];
                  C3y = (C3y*256)+(short)abBuffer[20];
                }
                //Collision Avoidance Algorithm
                DecisionMaking();
              }
//Control is the (global) return value of the DecisionMaking()
//function that implements CA algorithm, ready to be sent over
//I2C
              outBuffer[0] = Control;
            break;

            case I2C_MSG_TYPE_FINAL_STATUS:
      // Hybrid automaton state reset
              Control = HOLD;
              outBuffer[0] = Control;
              // Aggiorno lo stato finale
              FinalStatus.x = abBuffer[2];
```

```
        FinalStatus.x = (FinalStatus.x*256) +
            (short)abBuffer[3];
FinalStatus.y = abBuffer[4];
        FinalStatus.y = (FinalStatus.y*256) +
            (short)abBuffer[5];
        FinalStatus.theta = abBuffer[6];
        FinalStatus.theta = ((FinalStatus.theta*256) +
            (float)abBuffer[7])*0.01744444444444;
FinalCenterx = abBuffer[8];
        FinalCenterx = (FinalCenterx*256) +
            (short)abBuffer[9];
        FinalCentery = abBuffer[10];
        FinalCentery = (FinalCentery*256) +
            (short)abBuffer[11];
        Guardie &= ~DistNull_MASK;
      break;

    default:
      // Se arriva un comando sconosciuto
      outBuffer[0] = HOLD;
    break;
  }
}
// READING CYCLE
// wait until data is echoed
// want to know if RD_NOERR is SET AND RD_INCOMPLETE is
//  CLR
if( status & I2CHW_RD_NOERR )
{
  /* Data echoed - clear the read status */
  I2CHW_ClrRdStatus();
```

```
        /* Reset the pointer for the next data to echo */
        I2CHW_InitRamRead(outBuffer,I2C_OUTBUFFER_LENGTH);

        // Copying current status in the first byte of buffer
        outBuffer[0]=Control;
      }
    }

  }// End of external while

}

//-------------------------------------------------------------
// Implementazione della GRP
//-------------------------------------------------------------
#include <m8c.h>
#include "math.h"
#include "i2ccom.h"
#include "algoritmo.h"

extern BYTE             outBuffer[I2C_OUTBUFFER_LENGTH];
extern BYTE             status;

unsigned char           NumTouchers;
agentstatus             Status, FinalStatus;
unsigned char           Control;
vettd                   MaxTheta, MinTheta;
vettd                   Phi, Theta;
//Guards -Internal parameter used as switching values
unsigned char           Guardie;
short                   FinalCenterx;
short                   FinalCentery;
```

```
short                   C1x;
short                   C1y;
short                   C2x;
short                   C2y;
short                   C3x;
short                   C3y;
BYTE                    ToucherID;
float                   pMin, pMax;
float                   Ris;
float                   DistToGoal;

// Computes the safe cone where the robot can move
void ComputeConoA2(){
  int segno;
  vettd v0;
  vettd v1;
  vettd vRis;

  v0.x = C1x - C2x;
  v0.y = C1y - C2y;
  v1.x = C1x - C3x;
  v1.y = C1y - C3y;

  vRis.x = v0.x + v1.x;
  vRis.y = v0.y + v1.y;

  segno = PVett(v0, vRis);

  if(segno > 0){
    MinTheta = VettOrtoCW(v1.x, v1.y);
    MaxTheta = VettOrtoCCW(v0.x, v0.y);
```

```
  }
  else{
    MinTheta = VettOrtoCW(v0.x, v0.y);
    MaxTheta = VettOrtoCCW(v1.x, v1.y);
  }
  return;
}

void ComputeCono(){
  int a;
  int b;

  if (NumTouchers == 0){
    MaxTheta.x = 1;
    MaxTheta.y = 1;
    return;
  }

  if (NumTouchers == 1){

    if( ToucherID == SECONDO ){
      a = C1x - C2x;
      b = C1y - C2y;
    }

    if( ToucherID == TERZO ){
      a = C1x - C3x;
      b = C1y - C3y;
    }

    MaxTheta = VettOrtoCCW(a, b);
```

```
    MinTheta.x = -MaxTheta.x;
    MinTheta.y = -MaxTheta.y;
    return;
  }

  if (NumTouchers == 2){
    ComputeConoA2();
    return;
  }
}

float PVett(vettd v1, vettd v2){
  return ((v1.x*v2.y)-(v1.y*v2.x));
}

vettd VettOrtoCW(int vx, int vy){
  vettd vRis = VettOrtoCCW(vx, vy);
  vRis.x = -vRis.x;
  vRis.y = -vRis.y;
  return vRis;
}

vettd VettOrtoCCW(int vx, int vy){
  vettd vRis;
  if( (vy == 0) &&  (vx > 0) ){
    vRis.x = 0;
    vRis.y = 1;
    return vRis;
  }
  if( (vy == 0) &&  (vx < 0) ){
    vRis.x = 0;
```

```
    vRis.y = -1;
    return vRis;
  }
  vRis.x = -vy;
  vRis.y = vx;
  return vRis;
}

float PScal(vettd v1, vettd v2){
  return ((v1.x*v2.x)+(v1.y*v2.y));
}

float Distance(short a1x, short a1y, short a2x, short a2y){
  // Se non ricevo un veicolo imposto come massima la sua
  //distanza
  if ( (a2x==0) && (a2y==0) ) return 65535;
  return sqrt(pow((a1x - a2x),2)+pow((a1y - a2y),2));
}

void ThetaEqPhi(){
  vettd vPhi;
  vettd vTheta;
  float pVett;
  float pScal;
  vPhi = NormalizeVett(Phi);
  vTheta = NormalizeVett(Theta);
  pVett = PVett(vPhi, vTheta);
  pScal = PScal(vPhi, vTheta);
  if((pVett < SOGLIA_THETA_EQ_PHI) && (pVett > -0.25) &&
      (pScal > 0.25))
    Guardie |= ThetaEqPhi_MASK;
```

```
  else
    Guardie &= ThetaEqPhi_NOT_MASK;
}

vettd NormalizeVett(vettd v){
  vettd vRis;
  float Norma = sqrt(v.x*v.x + v.y*v.y);
  vRis.x = v.x/Norma;
  vRis.y = v.y/Norma;
  return vRis;
}

void ThetaInCono(){
  vettd vMax = NormalizeVett(MaxTheta);
  vettd vMin = NormalizeVett(MinTheta);

  if( (MaxTheta.x == 1) && (MaxTheta.y == 1) ){
    Guardie |= ThetaInTh_MASK;
    return;
  }

  if( (MaxTheta.x == 0) && (MaxTheta.y == 0) ){
    Guardie &= ThetaInTh_NOT_MASK;
    return;
  }

  if( NumTouchers == 1 ) {
    vMin.x = -vMax.x;
    vMin.y = -vMax.y;
  }
```

```
  pMax = PVett(Theta, vMax);
  pMin = PVett(Theta, vMin);

  if((fabs(pMax) < SOGLIA_ZERO) && (fabs(pMin) < SOGLIA_ZERO)){
    Guardie |= ThetaInTh_MASK;
    return;
  }

  if( (pMax > 0) && (pMin > 0) ){
    Guardie &= ThetaInTh_NOT_MASK;
    return;
  }

  if( (pMax < 0) && (pMin < 0) ){
    Guardie &= ThetaInTh_NOT_MASK;
    return;
  }

  if( (pMax >= 0) && (pMin < 0) ){
    Guardie |= ThetaInTh_MASK;
    return;
  }
  Guardie &= ThetaInTh_NOT_MASK;
}

void ThetaEqMax(){
  vettd Prov;
  if( (MaxTheta.x == 1) && (MaxTheta.y == 1) ){
    Guardie &= ThetaEqMax_NOT_MASK;
    return;
  }
```

```
  if( (MaxTheta.x == 0) && (MaxTheta.y == 0) ){
    Guardie &= ThetaEqMax_NOT_MASK;
    return;
  }

  Prov = NormalizeVett(MaxTheta);
  Ris = fabs(PVett(Theta, Prov));

  if((Ris < SOGLIA_THETA_MAX) && (PScal(Theta, MaxTheta) > 0))
    Guardie |= ThetaEqMax_MASK;
  else
    Guardie &= ThetaEqMax_NOT_MASK;
}

void PhiInCono(){
  float pMin, pMax;
  vettd PhiVett = Phi;

  if( (MaxTheta.x == 1) && (MaxTheta.y == 1) ){
    Guardie |= PhiInCone_MASK;
    return;
  }

  if( (MaxTheta.x == 0) && (MaxTheta.y == 0) ){
    Guardie &= PhiInCone_NOT_MASK;
    return;
  }
  pMax = PVett(PhiVett, MaxTheta);
  pMin = PVett(PhiVett, MinTheta);

  if( Control == ROLL )
```

```
    pMin = PVett(PhiVett, PhiVett);

  if( (pMax > 0) && (pMin < 0) ){
    Guardie |= PhiInCone_MASK;
  }
  else{
    Guardie &= PhiInCone_NOT_MASK;
  }
}

// Collision Avoidance Algorithm
void DecisionMaking(){

  Guardie &= ~DistNull_MASK;

  ComputeVTheta();
  ComputeVPhi();

  DistToGoal = Distance(Status.x, Status.y, FinalStatus.x,
      FinalStatus.y);

  if( (DistToGoal < SOGLIA_DIST_NULL) ) {
    Control = GOAL_REACHED;
    outBuffer[0] = Control;
  }

  ToucherID = 0;
  NumTouchers = 0;

  DistToGoal = Distance(C1x, C1y, C2x, C2y);
```

```
if (DistToGoal < 2*(RS+RC)*SCALING){
  NumTouchers += 1;
  ToucherID |= SECONDO;
}

DistToGoal = Distance(C1x, C1y, C3x, C3y);

if (DistToGoal < 2*(RS+RC)*SCALING){
  NumTouchers += 1;
  ToucherID |= TERZO;
}
ComputeCono();
ThetaEqPhi();
ThetaInCono();
ThetaEqMax();
PhiInCono();

switch(Control){

  case STRAIGHT:{
    if(((Guardie & ThetaInTh_MASK) == 0) ||
        ((Guardie & DistNull_MASK) == DistNull_MASK))
    {
      Control = HOLD;
    }
    else{
      Control = STRAIGHT;
    }
    if(((Guardie & ThetaEqPhi_MASK) == 0) &&
        ((Guardie & ThetaInTh_MASK) == ThetaInTh_MASK))
    {
```

```
    Control = HOLD;
  }
  break;
}

case HOLD:{
  if(((Guardie & ThetaInTh_MASK) == ThetaInTh_MASK) &&
      ((Guardie & ThetaEqPhi_MASK) == ThetaEqPhi_MASK) &&
          ((Guardie & DistNull_MASK) == 0))
  {
    Control = STRAIGHT;
  }
  else{
    if(((Guardie & ThetaEqMax_MASK) == ThetaEqMax_MASK) &&
        ((Guardie & ThetaEqPhi_MASK) == 0) &&
            ((Guardie & DistNull_MASK) == 0) )
    {
      Control = ROLL;
    }
    else{
      Control = HOLD;
    }
  }
  break;
}
case ROLL:{
  if(((Guardie & ThetaInTh_MASK) == 0) ||
      ((Guardie & ThetaEqPhi_MASK) == ThetaEqPhi_MASK) ||
          ((Guardie & DistNull_MASK) == DistNull_MASK) )
  {
    Control = STRAIGHT;
```

```
      }
      else{
        if(((Guardie & ThetaInTh_MASK) == ThetaInTh_MASK) &&
            ((Guardie & ThetaEqMax_MASK) == 0))
         {
           Control = ROLL2;
         }
         else{
           Control = ROLL;
         }
       }
       break;
    }

    case ROLL2:{
      if(((Guardie & ThetaEqPhi_MASK) == ThetaEqPhi_MASK) ||
          (Guardie & DistNull_MASK == DistNull_MASK) ||
              ((Guardie & ThetaInTh_MASK) == 0))
      {
        Control = HOLD;
      }
      else{
        Control = ROLL2;
      }
      break;
     }
  }
}

void ComputeVTheta(){
  vettd v;
```

```
  v.x = cos(Status.theta);
  v.y = sin(Status.theta);
  Theta = v;
}

void ComputeVPhi(){
  Phi.x = FinalCenterx - C1x;
  Phi.y = FinalCentery - C1y;
}
```

A.2.2 APFcomponent

```
//-------------------------------------------------------------------
// C main line
//-------------------------------------------------------------------

// part specific constants and macros
#include <m8c.h>
// PSoC API definitions for all User Modules
#include "PSoCAPI.h"
#include "math.h"
#include "PWM8.h"
#include "AMUX8.h"
#include "infrarossi.h"
#include "gradient.h"

#pragma fastcall16 PWM_interrupt
#pragma fastcall16 ADC_interrupt

// I2C Communication Buffers and status
BYTE    outBuffer[OUTBUFFER_SIZE];
BYTE    inBuffer[INBUFFER_SIZE];
```

```
BYTE    status;

struct FORCE {
int x;
int y;
} ForzaAttrattiva, ForzaRepulsiva;


// MAIN USED VARIABLES

int acquisi=0;
float dist_exp[5]={0,0,0,0,0};
char mux=0;

#define a  7426.0
#define h -8.692
#define c 48.43
#define d -0.6997

// riferimento ADC
#define Vref 2.5

//////////////////////////////////////////////////////
void main ()
{
   // outBuffer Initialization ;
   outBuffer[0] = 255;
   outBuffer[1] = 255;
   outBuffer[2] = 255;
   outBuffer[3] = 255;
   outBuffer[4] = 255;
```

```
outBuffer[5] = 255;
//X v
inBuffer[0] = 0;
inBuffer[1] = 0;
//Y v
inBuffer[2] = 0;
inBuffer[3] = 0;
//Theta v
inBuffer[4] = 0;
inBuffer[5] = 0;
//Xgoal
inBuffer[6] = 200;
inBuffer[7] = 0;
//Ygoal
inBuffer[8] = 0;
inBuffer[9] = 0;

// Inizializzazione blocchi
// MUX
AMUX8_Start();
// Start the slave and wait for the master
I2CHW_Start();
I2CHW_EnableSlave();
I2CHW_DisableMstr();
// PWM
PWM8_EnableInt();
PWM8_Start();
// Amplificatore
PGA_Start(3);
// ADC
DELSIG11_Start(3);
```

```
DELSIG11_StartAD();
DELSIG11_SetPower(3);
// Abilitazione interruzioni generali
M8C_EnableGInt;
// Abilitazione I2C interrupt
I2CHW_EnableInt();
PRT2DR = 0x00;
// Comincio ad acquisire dal sensore S1
// S1
AMUX8_InputSelect(0x04); // ADC Port_0_4
PRT2DR |= 0x04;          // VCC Port_2_2
// Setup the Read and Write Buffer - set to the same buffer
I2CHW_InitRamRead(outBuffer,OUTBUFFER_SIZE);
I2CHW_InitWrite(inBuffer,INBUFFER_SIZE);

while(1)
{
  /* Echo forever */
  status = I2CHW_bReadI2CStatus();
  /* Wait to read data from the master */
  if( status & I2CHW_WR_COMPLETE )
  {
    /* Data received - clear the Write status */
    I2CHW_ClrWrStatus();
    /* Reset the pointer for the next read data */
    I2CHW_InitWrite(inBuffer,INBUFFER_SIZE);
    CalcoloForzaAttrattiva();
    CalcoloForzaRepulsiva();
    CalcoloForzaTotale();
  }
  //wait until data is echoed
```

```
     //want to know if RD_NOERR is SET AND RD_INCOMPLETE is CLR
     if( status & I2CHW_RD_COMPLETE )
     {
       /* Data echoed - clear the read status */
       I2CHW_ClrRdStatus();
       /* Reset the pointer for the next data to echo */
       I2CHW_InitRamRead(outBuffer,OUTBUFFER_SIZE);
     }
   } // end while
} //end main

// Viene richiamata ogni interrupt del PWM
void PWM_interrupt ()
{
  float tmp;
  // Effettuo una lettura del sensore e calcolo la distanza
  // ogni 67ms circa (periodo del pwm)
  tmp = ( ( ( (float)acquisi)/1024 )*Vref ) +Vref;
  dist_exp[mux]=a*exp(h*tmp) + c*exp(d*tmp);
  // Effettuo lo switch dell'ingresso dell'ADC e
  // dell'alimentazone dei sensori
  switch (mux) {
    // S2
    case 0:
      tmp =  Ir_Bytescale(dist_exp[1]);
      if (tmp>255) outBuffer[1] = 255;
      else outBuffer[1] = tmp;
      PRT2DR = 0x00;     // Spengo tutte le porte
      AMUX8_InputSelect(0x02); // ADC Port_0_2
      PRT2DR |= 0x10;          // VCC Port_2_4
      mux++;
```

```
break;
//S3
case 1:
  tmp =  Ir_Bytescale(dist_exp[2]);
  if (tmp>255) outBuffer[2] = 255;
  else outBuffer[2] = tmp;
  PRT2DR = 0x00;  // Spengo tutte le porte
  AMUX8_InputSelect(0x00); // ADC  Port_0_0
  PRT2DR |= 0x40;          // VCC  Port_2_6
  mux++;
break;
//S4
case 2:
  tmp =  Ir_Bytescale(dist_exp[3]);
  if (tmp>255) outBuffer[3] = 255;
  else outBuffer[3] = tmp;
  PRT2DR = 0x00;  // Spengo tutte le porte
  AMUX8_InputSelect(0x06); // ADC Port_0_6
  PRT2DR |= 0x01;          // VCC Port_2_0
  mux++;
break;
//S5
case 3:
  tmp =  Ir_Bytescale(dist_exp[4]);
  if (tmp>255) outBuffer[4] = 255;
  else outBuffer[4] = tmp;
  PRT2DR = 0x00;  // Spengo tutte le porte
  AMUX8_InputSelect(0x05); // ADC Port_0_5
  PRT1DR |= 0x40;          // VCC Port_1_6
  mux++;
break;
```

```
    //S1
    case 4:
      tmp =  Ir_Bytescale(dist_exp[0]);
      if (tmp>255) outBuffer[0] = 255;
      else outBuffer[0] = tmp;
      PRT2DR = 0x00;  // Spengo tutte le porte
      PRT1DR &= ~ 0x40;  // Spengo la porta Port_1_6
      AMUX8_InputSelect(0x04); // ADC Port_0_4
      PRT2DR |= 0x04;          // VCC Port_2_2
      mux=0;
    break;
  }// end case
}// end PWM_interrupt

//-------------------------------------------------------------
// Implementazione dell'APF
//-------------------------------------------------------------

#include <m8c.h>
#include "infrarossi.h"
#include "math.h"
#include "gradient.h"

// External Variables
extern BYTE    outBuffer[OUTBUFFER_SIZE];
extern BYTE    inBuffer[INBUFFER_SIZE];
extern BYTE    status;

extern struct FORCE {
int x;
int y;
} ForzaAttrattiva, ForzaRepulsiva;
```

```
float angleG;

/*************************************************************/
/* Norma2 */
/*************************************************************/
double Norma2(double x, double y)
{
return (sqrt(pow(x,2)+pow(y,2)));
}


/*************************************************************/
/* Evaluation Actractive Force */
/*************************************************************/
void CalcoloForzaAttrattiva()
{
  double norma;
  int tmpG;
  ForzaAttrattiva.x = ((int)inBuffer[XGOAL_HI])*256 +
      inBuffer[XGOAL_LOW];
  tmpG = (((int)inBuffer[XVEHICLE_HI])*256 +
      inBuffer[XVEHICLE_LOW]);
  ForzaAttrattiva.x = ForzaAttrattiva.x - tmpG;


  ForzaAttrattiva.y = ((int)inBuffer[YGOAL_HI])*256 +
      inBuffer[YGOAL_LOW];
  tmpG = (((int)inBuffer[YVEHICLE_HI])*256 +
      inBuffer[YVEHICLE_LOW]);
  ForzaAttrattiva.y = ForzaAttrattiva.y - tmpG;
```

```
  norma = Norma2(ForzaAttrattiva.x,ForzaAttrattiva.y);

  if (norma < MIN_ATTRACTION)
  {
    ForzaAttrattiva.x = (int) (MIN_ATTRACTION *
        ForzaAttrattiva.x / norma);
    ForzaAttrattiva.y = (int) (MIN_ATTRACTION *
        ForzaAttrattiva.y / norma);
  }

}

/*************************************************************/
/* CalcoloForzaRepulsiva */
/*************************************************************/
void CalcoloForzaRepulsiva()
{
  double direction, intensity;
  double angle;
  short tmpG;
  // angle = Vehicle angle absolute coordinates
  tmpG = ((short)(inBuffer[THETAVEHICLE_HI])*256 +
      inBuffer[THETAVEHICLE_LOW]);
  angle = ((double)tmpG)*PI/180.0;

  // Contributo S1
  direction = MINUS_HALF_PI + angle;
  intensity = K1/outBuffer[S1];
  ForzaRepulsiva.x = intensity*cos(direction);
  ForzaRepulsiva.y = intensity*sin(direction);
```

```
  // Contributo S2
  direction = MINUS_3_4_PI + angle;
  intensity = K2/outBuffer[S2];
  ForzaRepulsiva.x += intensity*cos(direction);
  ForzaRepulsiva.y += intensity*sin(direction);
  // Contributo S3
  direction = MINUS_PI + angle;
  intensity = K3/outBuffer[S3];
  ForzaRepulsiva.x += intensity*cos(direction);
  ForzaRepulsiva.y += intensity*sin(direction);
  // Contributo S4
  direction = PLUS_3_4_PI + angle;
  intensity = K4/outBuffer[S4];
  ForzaRepulsiva.x += intensity*cos(direction);
  ForzaRepulsiva.y += intensity*sin(direction);
  // Contributo S5
  direction = HALF_PI + angle;
  intensity = K5/outBuffer[S5];
  ForzaRepulsiva.x += intensity*cos(direction);
  ForzaRepulsiva.y += intensity*sin(direction);
}

/*************************************************************/
/* CalcoloForzaTotale */
/*************************************************************/
void CalcoloForzaTotale()
{
  short angleDeg;
  double angle, theta, doubleX, doubleY, pVett, pScal;
  char commandToMotor;
```

```
  ForzaRepulsiva.x += ForzaAttrattiva.x;
  ForzaRepulsiva.y += ForzaAttrattiva.y;

  doubleX = (double)ForzaRepulsiva.x;
  doubleY = (double)ForzaRepulsiva.y;
  angle = atan2(doubleY,doubleX);
  angleG = angle;
  angleG = angle;

  //Command wheel policy
  angleDeg = ((short)inBuffer[4])*256 + (short)inBuffer[5];
  theta = (double)angleDeg;
  theta = (theta*PI)/180.0;
  angleG = theta;
  angleG = theta;
  pVett = (cos(angle)*sin(theta)) - (sin(angle)*cos(theta));
  pScal = (cos(angle)*cos(theta)) + (sin(angle)*sin(theta));
  if (pVett >= -0.5 && pVett <= 0.5 && pScal > 0)
    commandToMotor = STRAIGHT;
  else if ((pVett < -0.5 && pScal > 0) ||
      (pVett < 0 && pScal <= 0))
    commandToMotor = ROLL2;
  else
    commandToMotor = HOLD;

  outBuffer[COMMAND] = commandToMotor;
}
```

A.2.3 Controllo motori

```
//-----------------------------------------------------------
// C main line
```

```
//-------------------------------------------------------------------

// part specific constants and macros
#include <m8c.h>
// PSoC API definitions for all User Modules
#include "PSoCAPI.h"
#include "globals.h"
#include "psocgpioint.h"
// Api di controllo I2C
#include <i2CHWCommon.h>
// Define per comunicazione
#include "i2ccom.h"


/* I2C I/O Buffers */
BYTE    abBuffer[32];   // Input buffer
BYTE outBuffer[3];   // Out buffer
BYTE    status;         // Com status

/*************************************************************/
/* Funzioni di utilità per calcolo controllo                 */
/*************************************************************/


void Integra(){
  ErroreDX = (RifVelDX - MisuraEncoderDX);
  ErroreSX = (RifVelSX - MisuraEncoderSX);

  ProporzionaleErroreDX = Kp*(ErroreDX);
  ProporzionaleErroreSX = Kp*(ErroreSX);
```

```
IntegraleErroreDX += Ki*(ErroreDX + ErroreDXOld)/2;
IntegraleErroreSX += Ki*(ErroreSX + ErroreSXOld)/2;

DerivataErroreDX = Kd*(ErroreDX - ErroreDXOld);
DerivataErroreSX = Kd*(ErroreSX - ErroreSXOld);

ErroreDXOld = ErroreDX;
ErroreSXOld = ErroreSX;

MisuraEncoderDXOld = MisuraEncoderDX;
MisuraEncoderSXOld = MisuraEncoderSX;

uDX = ProporzionaleErroreDX + DerivataErroreDX +
    IntegraleErroreDX;
uSX = ProporzionaleErroreSX + DerivataErroreSX +
    IntegraleErroreSX;

// MOTORE DESTRO
// Saturazione sul controllo
if (uDX > 255)
  uDX = 255;
// Setto rotazione positiva
// DX avanti
PRT1DR |= M1A_MASK;
PRT1DR &= ~M1B_MASK;

// Se il riferimento di velocità è nullo disattivo
// l'enable motor L298
if (uDX == 0){
  // DX zero
  PRT1DR &= ~M1A_MASK;
```

```
  PRT1DR &= ~M1B_MASK;
}

if (uDX < 0){
  // Se il riferimento è <0 setto rotaizone negativa
  uDX = -uDX;
  PRT1DR &= ~M1A_MASK;
  PRT1DR |= M1B_MASK;
}

// MOTORE SINISTRO
// Saturazione del controllo
if (uSX > 255)
  uSX = 255;

// Setto rotazione positiva
// SX avanti
PRT1DR |= M2A_MASK;
PRT1DR &= ~M2B_MASK;

// Se il riferimento di velocità è nullo disattivo
// l'enable motor L298
if (uSX == 0){
  // SX zero
  PRT1DR &= ~M2A_MASK;
  PRT1DR &= ~M2B_MASK;
}

if (uSX < 0){
// Se il riferimento è <0 setto rotaizone negativa
uSX = -uSX;
```

```
  PRT1DR &= ~M2A_MASK;
  PRT1DR |= M2B_MASK;
  }

  tmpDX = (BYTE)uDX;
  tmpSX = (BYTE)uSX;

  PWM8_DX_WritePulseWidth(tmpDX);
  PWM8_SX_WritePulseWidth(tmpSX);
}

/*************************************************************/
/* Main                                                      */
/*************************************************************/
void main()
{
  // Inizializzazione dei blocchi digitali e
  // delle rispettive interruzioni
  // Inverter per encoder1
  DigInv_1_Start();
  DigInv_1_EnableInt();
  // Inverter per encoder2
  DigInv_2_Start();
  DigInv_2_EnableInt();

  Timer_Integrazione_EnableInt();
  //PWM per controllo motore 1 (DX)
  PWM8_DX_Start();
  PWM8_DX_WritePulseWidth(0);
  //PWM per controllo motore 2 (SX)
  PWM8_SX_Start();
```

```
PWM8_SX_WritePulseWidth(0);

/* Start the slave and wait for the master */
I2CHW_Start();
I2CHW_EnableSlave();
I2CHW_DisableMstr();

/* Enable the global and local interrupts */
M8C_EnableGInt;
I2CHW_EnableInt();

/* Setup the Read and Write Buffer-set to the same buffer */
I2CHW_InitRamRead(outBuffer,3);
I2CHW_InitWrite(abBuffer,32);
I2CHW_ClrWrStatus();
I2CHW_ClrRdStatus();

/* Inizializzazione delle varibili */
// Misura encoder
MisuraEncoder1 = 0;
MisuraEncoder2 = 0;
// Controllo
MisuraEncoderDX = 2;
MisuraEncoderSX = 2;
MisuraEncoderDXOld = 2;
MisuraEncoderSXOld = 2;
tmpDX = 0;
tmpSX = 0;
uDX = 0;
uSX = 0;
IntegraleErroreDX = 0;
```

```
IntegraleErroreSX = 0;
engine_on = 0x00;

// Init I2C out Buffer
outBuffer[0]=0x05;
outBuffer[1]=0x06;
outBuffer[2]=0x07;

// Setto a zero l'uscita verso i motori
PRT1DR &= ~M1B_MASK;
PRT1DR &= ~M1A_MASK;
PRT1DR &= ~M2B_MASK;
PRT1DR &= ~M2A_MASK;

/* Lettura dei comandi */
while(1){
  while(1){
    // Aspetto il completamento di una scrittura su I2C
    status = I2CHW_bReadI2CStatus();
    if( status & I2CHW_WR_COMPLETE){
      // Se entro ho ricevuto un pacchetto....
      /* Data received - clear the Write status */
      I2CHW_ClrWrStatus();
      /* Reset the pointer for the next read data */
      I2CHW_InitWrite(abBuffer,32);
      switch (abBuffer[1]){
        case I2C_MSG_TYPE_STOP:
          Timer_Integrazione_Stop();
          // Resetto le variabili di controllo
          // Misura encoder
          MisuraEncoder1 = 0;
```

```
  MisuraEncoder2 = 0;
  // Controllo
  MisuraEncoderDX = 2;
  MisuraEncoderSX = 2;
  MisuraEncoderDXOld = 2;
  MisuraEncoderSXOld = 2;
  tmpDX = 0;
  tmpSX = 0;
  uDX = 0;
  uSX = 0;
  IntegraleErroreDX = 0;
  IntegraleErroreSX = 0;

  // Arresto i motori
  PRT1DR &= ~M1B_MASK;
  PRT1DR &= ~M1A_MASK;
  PRT1DR &= ~M2B_MASK;
  PRT1DR &= ~M2A_MASK;

  // Reset PWMs
  PWM8_DX_WritePulseWidth(0);
  PWM8_SX_WritePulseWidth(0);
break;

case I2C_MSG_TYPE_GO:
  // Autorizzato a muovermi
  // Resetto le variabili di controllo
  MisuraEncoderDX=2;
  MisuraEncoderSX=2;
  MisuraEncoderDXOld=2;
  MisuraEncoderSXOld=2;
```

```
  ErroreDX=0;
  ErroreSX=0;
  ErroreDXOld=0;
  ErroreSXOld=0;
  ProporzionaleErroreDX=0;
  ProporzionaleErroreSX=0;
  DerivataErroreDX=0;
  DerivataErroreSX=0;
  IntegraleErroreDX=0;
  IntegraleErroreSX=0;
  tmpDX=0;
  tmpSX=0;
  uDX=0;
  uSX=0;
  // Avvio il timer per il controllo
  Timer_Integrazione_Start();
break;

case I2C_MSG_TYPE_REMOTE: // Controllo remoto
  // Assegno il valore ricevuto normalizzato
  // 805 tacche a colpo = 0.3 m/s
  RifVelDX = (int) (abBuffer[2]*10);
  RifVelSX = (int) (abBuffer[3]*10);
  if((RifVelDX != 0 || RifVelSX != 0) &&
      engine_on == 0)
  {
    MisuraEncoderDX=2;
    MisuraEncoderSX=2;
    MisuraEncoderDXOld=2;
    MisuraEncoderSXOld=2;
    ErroreDX=0;
```

```
  ErroreSX=0;
  ErroreDXOld=0;
  ErroreSXOld=0;
  ProporzionaleErroreDX=0;
  ProporzionaleErroreSX=0;
  DerivataErroreDX=0;
  DerivataErroreSX=0;
  IntegraleErroreDX=0;
  IntegraleErroreSX=0;
  tmpDX=0;
  tmpSX=0;
  uDX=0;
  uSX=0;
  engine_on = 0x01;
  // Avvio il timer per il controllo
  Timer_Integrazione_Start();
}
else if((RifVelDX == 0 && RifVelSX == 0) &&
    engine_on != 0)
{
  Timer_Integrazione_Stop();
  // Resetto le variabili di controllo
  // Misura encoder
  MisuraEncoder1 = 0;
  MisuraEncoder2 = 0;
  // Controllo
  MisuraEncoderDX = 2;
  MisuraEncoderSX = 2;
  MisuraEncoderDXOld = 2;
  MisuraEncoderSXOld = 2;
  tmpDX = 0;
```

```
            tmpSX = 0;
            uDX = 0;
            uSX = 0;
            IntegraleErroreDX = 0;
            IntegraleErroreSX = 0;
            engine_on = 0x00;
            // Arresto i motori
            PRT1DR &= ~M1B_MASK;
            PRT1DR &= ~M1A_MASK;
            PRT1DR &= ~M2B_MASK;
            PRT1DR &= ~M2A_MASK;
            // Reset PWMs
            PWM8_DX_WritePulseWidth(0);
            PWM8_SX_WritePulseWidth(0);
          }
        break;

        default:
          // Se arriva un comando sconosciuto
          // NON Arresto il veicolo -> Setto a zero il
          // controllo motore
          PRT1DR |= M1B_MASK;
          PRT1DR |= M1A_MASK;
          PRT2DR |= M2B_MASK;
          PRT2DR |= M2A_MASK;
          Timer_Integrazione_Stop();
        break;
      }
    }
  }
}
```

```
status++;
}
```

Bibliografia

[1] L. Pallottino, V. G. Scordio, E. Frazzoli, and A. Bicchi. Probabilistic verification of decentralized policy for conflict resolution in multi-agent systems. In *Internetional conference on Robotics and Automation ICRA 2006*, pages 2448–2453. IEEE, 2006.

[2] L. Pallottino, Vincenzo G. Scordio, E. Frazzoli, and A. Bicchi. Decentralized cooperative policy for conflict resolution in multi-vehicle system. *Trans. on Robotics*, 2007.

[3] V. J. Lumelsky and K. R. Harinarayan. Decentralized motion planning for multiple mobile robots: The cocktail party model. *Autonomous Robots 4*, 1997.

[4] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robot. *International journal of Robotic Research*, 1:90–98, 1986.

[5] M. G. Park and M. C. Lee. Artificial potential field based path planning for mobile robots using a virtual obstacle concept. In *International Conference on Advanced Intelligent Mechatronics AIM 2003*. IEEE, 2003.

[6] B. Zhang, W. Chen, and M. Fei. An optimized method for path planning mased on artificial potential field. In *International Conference on Intelligent systems Design and Application ISDA 2006*. IEEE, 2006.

[7] C. Qixin, H. Yanwen, and Z. Jingliang. An evolutionary artificial potential field algorithm for dynamic path plannin of mobile robot. In *International Conference on Intelligent Robots andSystems*. IEEE, 2006.

[8] D5.1-survey of middlewarefor networked embedded systems. http://-www.ist-runes.org/public_deliverables.html, RUNES Deliverable.

[9] D6.1-requirements report on control-aware embedded networks. http://-www.ist-runes.org/public_deliverables.html, RUNES Deliverable.

[10] A. Danesi. *Component Based Design for Networked Robotics*. PhD thesis, Università degli studi di Pisa, Facoltà di Ingegneria.

[11] A. Dunkels, B. Grönvall, and T. Voigt. Contiki a lightweight and flexible operating system for tiny networked sensor. In *Proceeding of IEEE Workshop on Embedded Networked Sensor*. IEEE, 2004.

[12] A. Danesi, A. Fagiolini, I. Savino, L. Pallottino, R. Schiavi, G. Dini, and A. Bicchi. A scalable platform for safe and secure decentralizated traffic management of multiagent mobile systems. In *ACM Workshop on Real-Word Wireless Sensor Networks*. IEEE, 2006.

[13] A. Convalle. Progettazione e controllo di sistema multiagente tramite rete di sensori e attuatori. Master's thesis, Facoltà di ingegneria Università di Pisa, 2005.

[14] S. M. Lavalle. *Planning Algorithms*, chapter 4, pages 128–168. Cambidge University Press, 2006.

[15] S. M. Lavalle. *Planning Algorithms*, chapter 5, pages 185–244. Cambidge University Press, 2006.

[16] P. Ögren and N. E. Leonard. A convergent dynamic window approach to obstacle avoidance. *Trans. on Robotics*, 2005.

[17] D. Bruijen, J. van Helvoort, and R. van de Molengraft. Realtime motion path generation using subtargets in a rapidly changing environment. *Robotics and Autonomous Systems*, 2007.

[18] L. Lee, R. Bhatt, and V. Krovi. Comparison of alternate methods for distribuited motion plannin of robot collectives within a potential field framework. In *International Conference on Robotics and Automation ICRA 2005*. IEEE, 2005.

[19] Q. Zhu, Y. Yan, and Z. Xing. Robot path planning based on artificial potential field approach with simulated annealing. In *International Conference on Intelligent systems Design and Application ISDA 2006*. IEEE, 2006.

[20] P. Ögren and N. E. Leonard. Obstacle avoidance in formation. In *International Conference on Robotics and Automation ICRA 2003*. IEEE, 2003.

[21] R. O. Saber and R. M. Murray. Flocking with obstacle avoidance: Cooperative with limited communication in mobile networks. In *Conference on Decision and Control*. IEEE, 2003.

[22] L. De Caria. Progettazione e controllo di sistema multiagente tramite rete di sensori e attuatori. Master's thesis, Facoltà di ingegneria Università di Pisa, 2005.

[23] J. Latombe C. M. Clark, S. M. Rock. Motion planning for multiple mobile robot systems using dynamic networks. In *IEEE Int. Conference on Robotics and Automation*. IEEE, 2003.