UNIVERSITÀ DI BRESCIA
FACOLTÀ DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione

Laboratorio di Robotica Avanzata
Advanced Robotics Laboratory

Corso di Robotica Mobile
(Prof. Riccardo Cassinis)

Adattamento della libreria Aria al protocollo SCIP 2.0 per il laser Hokuyo UHG-08LX

Elaborato di esame di:

Annarosa Agarossi, Silvio Finardi, Massimo Pachera, Mattia Rigo

Consegnato il: **19 luglio 2010**

Sommario

Il lavoro svolto consiste nello studio del protocollo SCIP2.0 per il laser Hokuyo UHG-08LX, l'adattamento della libreria Aria per supportare tale protocollo, la realizzazione di due programmi dimostrativi: uno per la lettura dei dati dal laser e uno per sfruttare tali misurazioni.

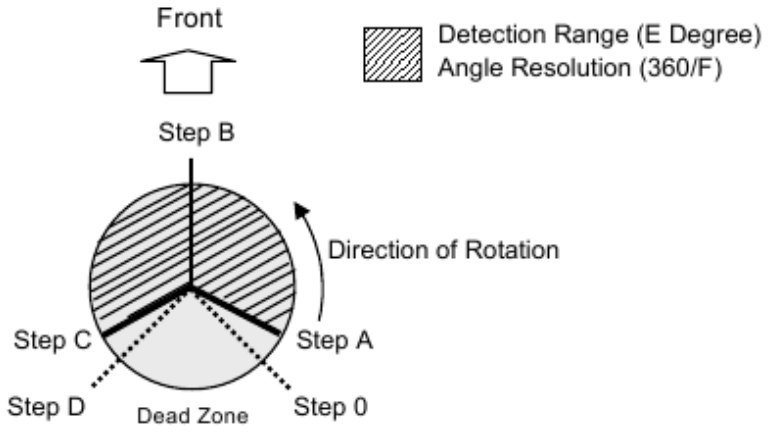
1. Introduzione

Il tema centrale del lavoro descritto nella presente relazione riguarda la modifica della libreria Aria al fine di garantire la compatibilità del laser range scanner Hokuyo UHG-08LX. Il problema di fondo consiste nel fatto che tale dispositivo sfrutta come protocollo di comunicazione la versione 2.0 di SCIP, mentre la libreria Aria, necessaria per il funzionamento del robot MORGUL, implementa solo la versione 1.1. Tale problema comporta l'impossibilità di utilizzare il dispositivo UHG-08LX con MORGUL. Di qui la necessità di adattare la libreria Aria al protocollo utilizzato dal nuovo dispositivo.

2. Il problema affrontato

2.1. Confronto tra i dispositivi URG-04LX e UHG-08LX

I Laser Range Scanner Hokuyo URG-04LX e UHG-08LX hanno caratteristiche piuttosto simili eccetto il protocollo di comunicazione e alcuni parametri di misura. Per quanto riguarda il protocollo di comunicazione, il modello URG è compatibile con la versione 1.1 del protocollo SCIP, mentre il modello UHG è compatibile con la versione 2.0. Faremo un confronto più dettagliato delle differenze tra i due protocolli nel paragrafo successivo.



Per quanto concerne i parametri di misura, entrambi i dispositivi ruotano in senso antiorario su un angolo di circa 270°. Il *Detection Range E* è l'angolo massimo in cui il sensore può rilevare delle misure e la risoluzione angolare è *360°/F*, ove *F* è detto *Slit Division* ed è pari a 1024 per entrambi i dispositivi. Ogni punto di misurazione è detto s*tep.*

Ciò che distingue i due modelli è l'estensione del *Detection Range*:

		URG-04LX	UHG-08LX
Step 0	Primo punto di misura	0	0
Step A	Primo *step* di misura nel *Detection Range*	44	0
Step B	*Step* frontale	384	384
Step C	Ultimo punto del *Detection Range*	725	768
Step D	Ultimo punto di misura	768	768

Come è possibile notare dalla tabella, nel modello URG-04LX gli *step 0* e *A* non coincidono, così come anche gli *step* finali *C* e *D*. Il *Detection Range* dell'URG-04LX è quindi 239.77°, mentre quello dell'UHG-08LX si estende fino a coprire l'intero intervallo di misurazione di 270.35°.

2.2. Confronto tra i protocolli SCIP1.1 e SCIP2.0

In questo paragrafo riportiamo la comparazione tra i messaggi dei protocolli SCIP1.1 e SCIP2.0, risultante dallo studio della documentazione del protocollo di comunicazione per i laser Hokuyo. Rispetto alla documentazione originale, al fine di rendere più chiara la struttura dei comandi, sono stati colorati i vari campi con colori diversi, inoltre è stato cambiato l'allineamento di alcuni messaggi della versione 1.1 per renderli più facilmente confrontabili con la versione 2.0.

2.2.1. Differenze generali

Innanzitutto vediamo la struttura generica dei messaggi e le differenze di massima tra le due versioni, distinguendo tra i comandi inviati dall'host al sensore e le risposte inviate dal sensore all'host.

HOST → SENSOR

SCIP1.1		
Command (1 byte)	Parameter	LF or CR

SCIP2.0			
Command symbol (2 byte)	Parameter	String Characters (Max 16 letters)	LF or CR or both

La prima differenza fondamentale tra i due protocolli è la lunghezza dei comandi. In SCIP1.1 infatti un comando è identificato con un singolo carattere (1 byte in codifica ASCII), mentre in SCIP2.0 sono necessari due caratteri.

Oltre alla lunghezza del simbolo del comando si può notare che SCIP2.0 introduce un campo opzionale di lunghezza variabile String Characters. Esso ha la funzione di identificativo nel caso sia necessario inviare lo stesso comando più di una volta e occorra discriminare le risposte ricevute. Se utilizzato, va tenuto presente che la stringa deve necessariamente iniziare con un punto e virgola (';') per separarla dai campi Parameter.

SENSOR → HOST

SCIP1.1							
Command	Parameter	LF	Status (1 byte)	LF	Data	LF	LF

SCIP2.0										
Command Symbol	Parameter	String Characters	LF	Status (2 byte)	Sum (1 byte)	LF	Data	Sum	LF	LF

Nelle risposte provenienti dal sensore la versione 2.0 del protocollo introduce alcuni campi Sum di lunghezza 1 byte che possono servire per verificare l'integrità dei pacchetti.

Il campo Status occupa 1 byte in SCIP1.1, mentre ne occupa 2 in nella versione 2.0. La documentazione della prima versione stabilisce come codici di errori tutti gli stati diversi da '0' (30_{16}), mentre la versione 2.0 considerati codici di errore tutti i codici diversi da '00' e '99'.

2.2.2. Comando di Informazioni di Versione

Questo comando serve a ricevere dal laser informazioni di versione quali il numero di serie, il produttore, etc.

HOST → SENSOR

SCIP1.1	
V	LF or CR

SCIP2.0			
V	V	String Characters	LF

SENSOR → HOST

SCIP1.1				
V	LF	Status	LF	
Vendor Information			LF	
Product Information			LF	
Firmware Version			LF	
Protocol Version			LF	
Sensor Serial Number			LF	LF

SCIP2.0								
V	V	String Characters	LF	0	0	P	LF	
Vendor Information					;	Sum	LF	
Product Information					;	Sum	LF	
Firmware Version					;	Sum	LF	
Protocol Version					;	Sum	LF	
Sensor Serial Number					;	Sum	LF	LF

2.2.3. Comandi di abilitazione e disabilitazione del laser

Per accendere o spegnere il laser e abilitarlo ad effettuare misurazioni, nel protocollo SCIP1.1 è presente il solo comando 'L' con l'uso di un apposito codice di controllo ('0'=OFF; '1'=ON). In SCIP2.0 sono presenti invece due comandi distinti: 'BM' e 'QT'. Va inoltre notato che nella versione 1.1 del protocollo il laser viene acceso e abilitato alla misurazione automaticamente, mentre nella versione 2.0 il laser è inizialmente spento e disabilitato di default. È possibile stabilire se il laser è abilitato o meno osservando il LED verde sul laser: se lampeggia è OFF, se emette luce continua è ON.

HOST → SENSOR

SCIP1.1		
L	Control Code (1 byte)	LF

SCIP2.0 - Accensione			
B	M	String Characters	LF

SCIP2.0 - Spegnimento			
Q	T	String Characters	LF

SENSOR → HOST

SCIP1.1						
L	Control Code	LF	Status	Data	LF	LF

SCIP2.0 - Accensione							
B	M	String Characters	LF	Status	Sum	LF	LF

Il comando 'BM' ritorna 3 possibili status:

- 00 – Command received without any Error.
- 01 – Unable to control due to laser malfunction.
- 02 – Laser is already on.

SCIP2.0 - Spegnimento								
Q	T	String Characters	LF	0	0	P	LF	LF

2.2.4. Comando per impostare la velocità di comunicazione

Questo comando serve a cambiare la velocità di comunicazione del sensore (misurata in Kbps).

HOST → SENSOR

SCIP1.1			
S	Baud Rate (6 byte)	Reserved Area (7 byte)	LF

I 7 byte del campo “Reserved Area” non hanno alcun effetto sul sensore.

SCIP2.0				
S	S	Bit Rate (6 byte)	String Characters	LF

I valori possibili del parametro Baud/Bit Rate (espressi in Kbps) possono essere

- 19.2 (solo SCIP2.0)
- 38.4 (solo SCIP2.0 e non supportato da tutti i modelli)
- 57.6
- 115.2
- 250
- 500
- 750

I valori devono essere rappresentati con una stringa di 6 caratteri, ad esempio:

- 57.6 = ‘057600’
- 115.2 = ‘115200’

SENSOR → HOST

SCIP1.1						
S	Baud Rate	Reserved Area	LF	Status	LF	LF

SCIP2.0								
S	S	Bit Rate	String Characters	LF	Status	Sum	LF	LF

La versione 2.0 specifica cinque possibili valori del campo “Status”

- 00 – Command received without any Error.
- 01 – Bit Rate has non-numeric value.
- 02 – Invalid Bit Rate.
- 03 – Sensor is already running at the defined bit rate.
- 04 – Not compatible with the sensor model.

2.2.5. Comandi di acquisizione dati

Per acquisire i dati dal sensore nel protocollo SCIP1.1si usa il comando ‘G’. Quando il sensore riceve tale comando invia all’host l’ultima lettura che ha effettuato e memorizzato.

Nella versione 2.0 del protocollo esistono invece due tipi di comando per ottenere le letture dal laser: il comando ‘GD’ (o ‘GS’) agisce in modo simile al vecchio comando ‘G’, mentre il comando ‘MD’ (o ‘MS’) effettua la misurazione dopo aver ricevuto il comando, e permette anche di specificare quante scansioni effettuare e l’intervallo tra ogni scansione.

Usare ‘D’ oppure ‘S’ come secondo carattere del comando ha un effetto sulla codifica dei dati ricevuti: ‘MD’ e ‘GD’ usano una codifica a 3 caratteri, mentre ‘MS’ e ‘GS’ usano una codifica a due caratteri. SCIP1.1 usa solo la codifica a 2 caratteri.

Per semplicità riportiamo la comparazione solo per il comando ‘GSGD’ in quanto all’incirca equivalente al comando della versione precedente.

HOST → SENSOR

SCIP1.1				
G	Starting Point (3 byte)	End Point (3 byte)	Cluster Count (2 byte)	LF

SCIP2.0						
G	D or S	Starting Step (4 byte)	End Step (4 byte)	Cluster Count (2 byte)	String Characters	LF

SENSOR → HOST

1. Se la quantità di dati è minore o uguale a 64 byte

SCIP1.1				
G	Starting Point	End Point	Cluster Count	LF
Status		LF		
Data		LF	LF	

SCIP2.0																			
G	D or S	Starting Step			End Step			Cluster Count		String Characters									LF
0	0	P	LF	Timestamp (4 byte)				Sum	LF										
Data								Sum	LF	LF									

2. Se la quantità dei dati è esattamente un multiplo di 64 byte

SCIP1.1																			
G	Starting Point			End Point			Cluster Count			LF									
Status			LF																
Data Block 1 (64 byte)																		LF	
….....																		LF	
Data Block N (64 byte)																		LF	LF

SCIP2.0																			
G	D or S	Starting Step			End Step			Cluster Count		String Characters									LF
0	0	P	LF	Timestamp				Sum	LF										
Data Block 1 (64 byte)																	Sum	LF	
….....																	Sum	LF	
Data Block N (64 byte)																	Sum	LF	LF

3. Se la quantità dei dati è maggiore di 64 byte e termina con *n* byte di resto

SCIP1.1																			
G	Starting Point			End Point			Cluster Count			LF									
Status			LF																
Data Block 1 (64 byte)																		LF	
….....																		LF	
Data Block N-1 (64 byte)																		LF	
Data Block N (*n* byte)																		LF	LF

SCIP2.0							
G	D or S	Starting Step	End Step	Cluster Count	String Characters	LF	
0	0	P	LF	Timestamp	Sum	LF	
Data Block 1 (64 byte)						Sum	LF
….....						Sum	LF
Data Block N-1 (64 byte)						Sum	LF
Data Block N (*n* byte)						Sum	LF LF

SCIP2.0 specifica inoltre i messaggi di ritorno in caso di errore:

SCIP2.0						
G	D or S	Starting Step	End Step	Cluster Count	String Characters	LF
Status		Sum	LF	LF		

Il campo status può assumere i seguenti valori:

- 01 – Starting Step has non numeric value.
- 02 – End Step has non numeric value.
- 03 – Cluster Count has non numeric value.
- 04 – End Step is out of range.
- 05 – End Step is smaller than Starting Step.
- 10 – Laser is off.

50~98 – Hardware trouble (such as laser, motor malfunction, etc...)

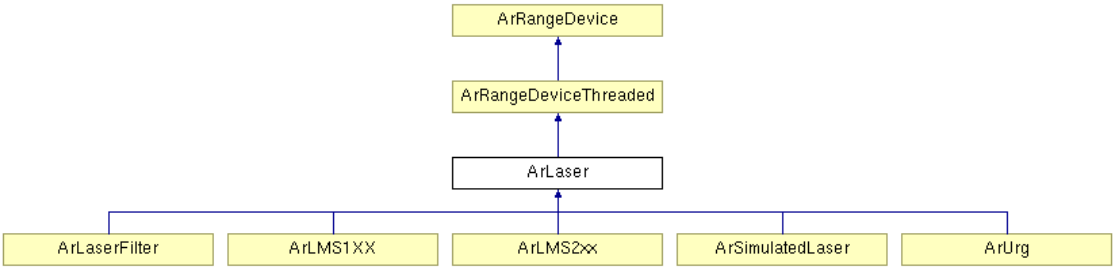
3. La soluzione adottata

Dopo aver esaminato attentamente le specifiche relative ai protocolli di comunicazione SCIP 1.1 e SCIP 2.0 (non compatibili tra loro) e averne compreso le differenze principali abbiamo deciso di modificare la classe ArUrg adattandola alle nuove regole di comunicazione. Tale classe (il cui codice si trova in /usr/local/Aria/src/ArUrg.cpp) era già stata parzialmente modificata dal prof. Cassinis il quale aveva adattato il comando "v" (che serve per richiedere le informazioni di versione al sensore) trasformandolo nel corrispondente comando "vv" e leggendo le informazioni di risposta mandate dal sensore. Il nostro compito è stato quello di adattare anche tutti gli altri comandi al protocollo 2.0.

3.1. Modifica della classe ArUrg

Compito della classe ArUrg è specializzare la classe ArLaser implementando gli opportuni metodi per mandare i comandi al sensore laser Hokuyo URG e interpretare le risposte da esso restituite. Come già detto la classe ArUrg fornita con Aria è in grado parlare con il sensore laser utilizzando il protocollo SCIP 1.1. In questo paragrafo verranno illustrate le modifiche apportate alla classe al fine di utilizzare il protocollo SCIP 2.0 per il colloquio con il sensore UHG-08LX.

Innanzi tutto è opportuno tener presente la gerarchia delle classi da cui ArUrg discende:



Fra gli altri, i metodi della classe ArUrg che vale la pena di menzionare sono i seguenti:

- **readLine**: legge una stringa di caratteri proveniente dal sensore; è un metodo protetto che viene utilizzato all'interno della classe per la lettura delle risposte provenienti dal sensore.
- **writeString**: scrive una stringa di caratteri che viene mandata al sensore; è un metodo protetto che viene utilizzato all'interno della classe per mandare i vari comandi necessari al sensore.
- **sensorInterp**: metodo protetto che si preoccupa di dare una giusta interpretazione dei dati che il sensore laser restituisce a fronte di un certo comando ricevuto sulla base delle specifiche del protocollo.
- **runThread**: ereditando dalla classe ArRangeDeviceThreaded, la classe ArUrg implementa il metodo virtuale e protetto runThread grazie al quale tutta la procedura di spedizione dei comandi di acquisizione delle distanze verso il sensore laser e l'acquisizione e interpretazione dei dati provenienti dal sensore avviene in un thread diverso del programma chiamante.
- **setParams e setParamsByStep**: metodi protetti che servono a impostare i parametri necessari per l'invio del comando di acquisizione delle misurazioni effettuate in un determinato arco di circonferenza.

Di seguito vengono illustrate le modifiche che sono state effettuate ai vari metodi della classe.

3.1.1. Costruttore

Il costruttore della classe ArUrg non è stato modificato: esso infatti non fa altro che inizializzare gli attributi ereditati dalla classe ArLaser; tali attributi riguardano la lista delle velocità di comunicazione disponibili per il dispositivo, la regione polare di acquisizione, il nome del dispositivo e altri parametri non importanti ai nostri scopi.

Inizialmente il lavoro era orientato a mantenere la compatibilità fra le due versioni dei protocolli SCIP quindi è stata introdotta una variabile SCIPVersion inizializzandola a zero (al momento della costruzione dell'oggetto di tipo ArUrg, ancora non sappiamo quale versione del protocollo usa il sensore).

```
SCIPVersion=0; //SCIP Version not known yet
```

3.1.2. setParams

Un metodo importante della classe ArUrg è **setParams**. Esso infatti accetta, fra i parametri di ingresso la regione polare da scandire (compresa tra startingDegrees e endingDegrees) e si preoccupa di valorizzare correttamente gli attributi startingStepRaw, endingStepRaw, startingStep, endingStep e clusterCount (ereditati da ArRangeDevice) a seconda della risoluzione angolare del dispositivo.

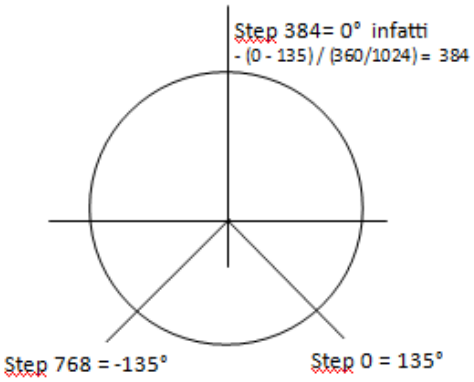
L'hokuyo UHG-08LX suddivide l'angolo giro (360°) in 1024 sezioni cosicché la sua risoluzione angolare è $\frac{360}{1024} = 0{,}3515625$.

In altre parole, le misurazioni hanno fra loro una distanza angolare di 0,3515625°; allo stesso modo potremmo dire che per ogni grado di scostamento angolare il sensore effettua fra le 2 e le 3 letture (2,844).

Il metodo dovrà quindi operare la seguente trasformazione come riportato nel foglio che descrive lo standard di comunicazione SCIP 2.0:

```
double startingStepRaw;
double endingStepRaw;
startingStepRaw = -(startingDegrees - 135.0) / .3515625;
endingStepRaw = -(endingDegrees - 135.0) / .3515625;
```

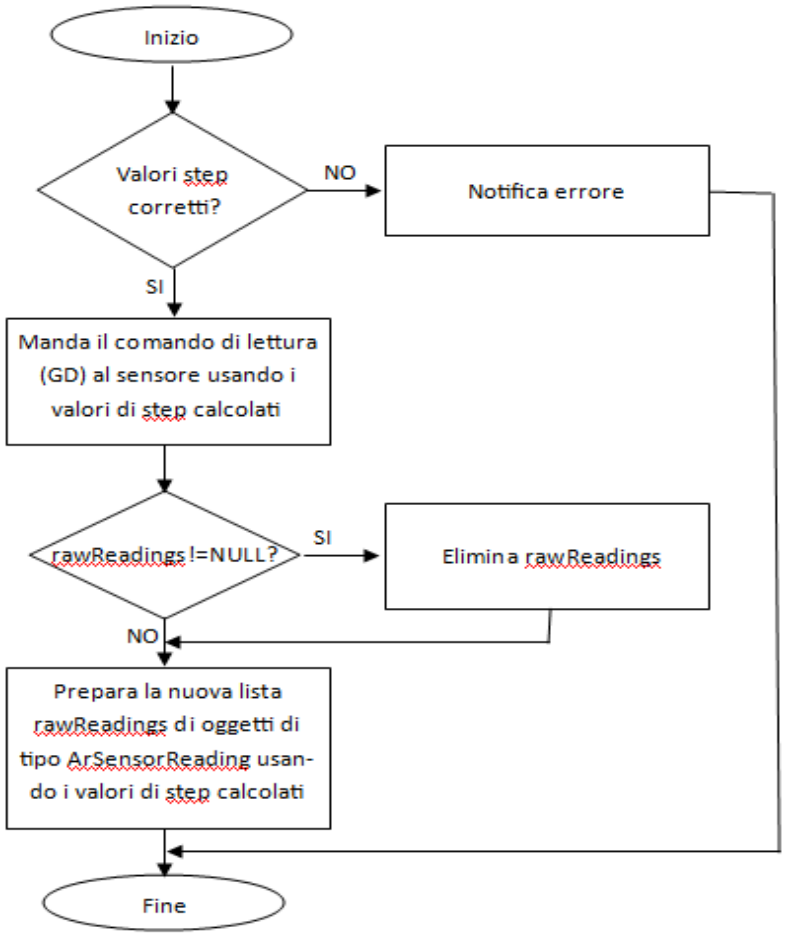
Il seguente schema chiarisce meglio la trasformazione effettuata:



Il metodo setParams, infine, richiama a sua volta il metodo setParamsByStep utilizzando i nuovi valori calcolati per gli step.

3.1.3. Il metodo setParamsByStep

Questo metodo utilizza i valori calcolati dal metodo setParams per mandare l'opportuno comando di acquisizione delle letture al sensore. A titolo puramente illustrativo riportiamo nel seguente diagramma di flusso le operazioni svolte dal metodo.



Il comando 'G' del protocollo SCIP 2.0 è del tutto equivalente al comando 'G' della versione 1.1: quando il sensore riceve questo comando, fornisce l'ultima misurazione effettuata all'host. Tuttavia, come abbiamo già avuto modo di vedere, il formato dei comandi da mandare al sensore è leggermente diverso rispetto alla versione 1.1 del protocollo di comunicazione SCIP. Al comando G, infatti, deve essere aggiunto un secondo carattere per stabilire il tipo di codifica da usare per le letture:

- GD: codifica a 3 caratteri
- GS: codifica a 2 caratteri

Noi abbiamo scelto di utilizzare la codifica a 3 caratteri. Il seguente estratto di codice mostra la vera e propria formattazione del comando da mandare al sensore.

```
sprintf(myRequestString, "GD%04d%04d%02d\n", myStartingStep, endingStep, clusterCount);
```

Si noti che i parametri startingStep ed endingStep sono lunghi 4 byte mentre il campo clusterCount è lungo 2 byte.

Per quanto riguarda il concetto di cluster gestito dal sensore, non sono state apportate modifiche. Tra l'altro, i programmi di test predefiniti e quello da noi realizzato non fanno uso del concetto di cluster.

3.1.4. Il metodo blockingConnect

Quando il metodo viene invocato, viene dapprima invocato il metodo blockingConnectSCIP1 che cerca di mandare al sensore dei comandi utilizzando la versione 1.1 del protocollo di comunicazione SCIP. Se però il sensore è abilitato a utilizzare la versione 2.0 del protocollo, la comunicazione avverrà con degli errori e il metodo ritornerà il valore booleano FALSE. A questo punto viene fatto un secondo tentativo di comunicazione invocando il metodo blockingConnectSCIP2 che usa la versione 2.0 del protocollo. Se entrambi i tentativi falliscono il metodo ritorna il valore booleano FALSE. Si noti che a seguito di una

risposta positiva di uno dei due tentativi, viene valorizzata anche la variabile SCIPVersion al corrispondente valore che identifica la versione di protocollo utilizzato dal sensore.

```
AREXPORT bool ArUrg::blockingConnect(void)
{
if (blockingConnectSCIP1())
{
	printf ("Found a SCIP 1 laser\n");
	SCIPVersion=1;
	return true;
}

if (blockingConnectSCIP2())
{
	printf ("Found a SCIP 2 laser\n");
	SCIPVersion=2;
	return true;
}
return false;
}
```

3.1.5. Il metodo blockingConnectSCIP2

In questo metodo viene effettuato il tentativo di comunicazione con il sensore utilizzando il protocollo SCIP 2.0.

Dopo aver fatto gli opportuni controlli sui parametri del sensore e sulla connessione del sensore all'host (questa parte non è stata modificata), il metodo invoca a sua volta il metodo setParams già descritto in precedenza. Noi abbiamo aggiunto un'istruzione che stampi a video i parametri passati in questa invocazione per assicurarci che essi fossero passati correttamente:

```
setParams(getStartDegrees(), getEndDegrees(), getIncrement(), getFlipped());
printf("Start deg: %f, End deg: %f\n",myStartDegrees, myEndDegrees);
```

Successivamente viene effettuato il tentativo di invio del comando "VV" (richiesta di informazioni di versione) e la corrispondente lettura della risposta.

Se non viene ricevuto risposta o se la risposta non avviene secondo le regole del protocollo illustrate nel capitolo precedente, è possibile che la velocità di comunicazione sulla porta seriale non sia stata impostata al valore corretto e quindi viene eseguito un nuovo tentativo di invio del comando "VV" dopo aver auto-impostato la velocità di comunicazione. Se ancora non vengono ricevute risposte o se riceviamo risposte errate il metodo tenta ancora di cambiare il baud rate; se invece non esiste la connessione seriale "si arrende".

```
// send for the version info
if (!writeString("VV\n"))
{
failedToConnect();
return false;
}
// send for the version info
if (!readLine(buf, sizeof(buf), 10000) ||
	strncasecmp(buf, "VV", strlen("VV")) != 0)
{
	// if we didn't get it, try it at what the autobaud rate is
	if (serConn != NULL)
	{
		serConn->setBaud(atoi(getAutoBaudChoice()));
		ArUtil::sleep(1000);
		// send for the version info again at the new baud rate
		if (!writeString("VV\n"))
		{
			failedToConnect();
			return false;
		}
		// if we still didn't get it, just give up
		if (!readLine(buf, sizeof(buf), 10000) ||
		 strncasecmp(buf, "VV", strlen("VV")) != 0)
		{
```

```
                ArLog::log(ArLog::Normal, "%s: '%s'\n", getName(), buf);
                ArLog::log(ArLog::Normal, "%s::blockingConnectSCIP2: Did not get
                back version response (even after switching to autobaudchoice)",
                getName());
                failedToConnect();
                return false;
                }
        }
        // if we don't have a serial port, then we can't change the baud,
        // so just fail
        else
        {
        ArLog::log(ArLog::Normal, "%s: '%s'\n", getName(), buf);
        ArLog::log(ArLog::Normal,
                "%s::blockingConnectSCIP2: Did not get back version response",
                getName());
        failedToConnect();
        return false;
        }
}
```

Ora che il comando “VV” dovrebbe esser stato spedito con successo, l’host dovrebbe ricevere una risposta che, come illustrato nel precedente capitolo, dovrebbe contenere il codice “00” indicante che la trasmissione del comando è avvenuta senza errori.

```
// get the status from that request
if (!readLine(buf, sizeof(buf), 10000) ||
strncasecmp(buf, "00", strlen("00")) != 0)
{
ArLog::log(ArLog::Normal,
        "%s::blockingConnectSCIP2: Bad status on version response",
        getName());
failedToConnect();
return false;
}
```

A questo punto il metodo cerca di cambiare il baud rate per la comunicazione utilizzando il comando “SS” unito ai 6 byte necessari per indicare la nuova velocità di comunicazione (acquisita tramite il metodo getAutoBaudChoice()) ed effettua un nuovo invio del comando “VV”.

```
if (serConn != NULL)
{
// now change the baud...
sprintf(buf, "SS%06d\n", atoi(getAutoBaudChoice()));
if (!writeString(buf))
        {
        failedToConnect();
        return false;
        }
ArUtil::sleep(1000);

//serConn->setBaud(115200);
serConn->setBaud(atoi(getAutoBaudChoice()));
// wait a second for the baud to change...
ArUtil::sleep(1000);

// send for the version info
if (!writeString("VV\n"))
        {
        failedToConnect();
        return false;
        }

// send for the version info
if (!readLine(buf, sizeof(buf), 10000) ||
strncasecmp(buf, "VV", strlen("VV")) != 0)
        {
        ArLog::log(ArLog::Normal, "%s: '%s'\n", getName(), buf);
        ArLog::log(ArLog::Normal, "%s::blockingConnectSCIP2: Did not get back
        version response after baud change",
         getName());
        failedToConnect();
        return false;
```

```
        }
}
```

Infine il metodo legge le informazioni di versione ricevute dal sensore e le stampa a video. Visto che la risposta è composta da varie righe terminanti con il carattere LF (line feed) e che il termine della risposta è indicato da un singolo carattere LF (a tale proposito si consulti il capitolo precedente), il seguente ciclo while legge le varie linee ricevute fino a quando non viene letta una stringa avente nella prima posizione il carattere \n.

```
while (readLine(buf, sizeof(buf), 10000))
{
        if (buf[0] == '\n' || buf[0] == '\r')
        break;
        if (strlen(buf) == 0)
        {
                ArLog::log(ArLog::Normal,
                "%s::blockingConnectSCIP2: Bad information in version response",
                getName());
                failedToConnect();
                return false;
        }
// chop off the \n or \r and the checksum **************
buf[strlen(buf) - 2] = '\0';
if (strncasecmp(buf, "VEND:", strlen("VEND:")) == 0)
        myVendor = &buf[5];
else if (strncasecmp(buf, "PROD:", strlen("PROD:")) == 0)
        myProduct = &buf[5];
else if (strncasecmp(buf, "FIRM:", strlen("FIRM:")) == 0)
        myFirmwareVersion = &buf[5];
else if (strncasecmp(buf, "PROT:", strlen("PROT:")) == 0)
        myProtocolVersion = &buf[5];
else if (strncasecmp(buf, "SERI:", strlen("SERI:")) == 0)
        mySerialNumber = &buf[5];
myStat="SCIP 2 doesn't return a status."; //SCIP 2 doesn't return a status
}

if (myVendor.empty() || myProduct.empty() || myFirmwareVersion.empty() ||
myProtocolVersion.empty() || mySerialNumber.empty())
{
ArLog::log(ArLog::Normal,
        "%s::blockingConnectSCIP2: Missing information in version response",
        getName());
        failedToConnect();
        return false;
}
```

Dopo aver stampato le informazioni di versione, procediamo con l'accensione del laser mandando al sensore il comando "BM". Il seguente estratto di codice mostra anche i controlli effettuati:

```
// send for the laser illumination
if (!writeString("BM\n"))
{
        failedToConnect();
        return false;
}

// get laser status
if (!readLine(buf, sizeof(buf), 10000) ||
strncasecmp(buf, "BM", strlen("BM")) != 0)
{
        ArLog::log(ArLog::Normal,
        "%s::blockingConnectSCIP2: Did not get laser illumination control",
        getName());
        failedToConnect();
        return false;
}
if (!readLine(buf, sizeof(buf), 10000) ||
(strncasecmp(buf, "00", strlen("00")) != 0 &&
strncasecmp(buf, "02", strlen("02")) != 0))
{
        ArLog::log(ArLog::Normal,
```

```
        "%s::blockingConnectSCIP2: Bad status on laser illumination response",
        getName());
        failedToConnect();
        return false;
}
// Print the laser status
if(strncasecmp(buf, "00",strlen("00"))==0)
        printf ("Command received without any error.\n");
else if(strncasecmp(buf, "02",strlen("02"))==0)
        printf ("Laser is already on.\n");
else if (strncasecmp(buf, "01",strlen("01"))==0)
        printf ("Unable to control due to laser malfunction.\n");

if (!readLine(buf, sizeof(buf), 10000) ||
(buf[0] != '\n' && buf[0] != '\r'))
{
        ArLog::log(ArLog::Normal,
      "%s::blockingConnectSCIP2: Extra stuff on laser illumination response",
        getName());
        failedToConnect();
        return false;
}
```

3.1.6. Il metodo runThread

Questo metodo viene ripetuto per tutta l'esecuzione del programma in un thread a se stante: esso è costituito dal seguente ciclo while:

```
while (getRunning())
  { … }
```

All'interno del ciclo, la prima cosa che viene controllata è che sia stata eseguita la connessione iniziale al sensore controllando il valore della variabile booleana myStartConnect. Nel nostro caso, la variabile viene valorizzata a false dal metodo blockingConnect che viene a sua volta chiamato dal metodo connectLasers e quindi questa parte di codice non viene mai eseguita.

Visto che il metodo blockingConnect effettua due tentativi di comunicazione con il sensore utilizzando le diverse versioni del protocollo di comunicazione SCIP e solo alla fine del tentativo che ha successo valorizza la variabile SCIPVersion, siamo stati costretti a inserire un'istruzione che impedisse l'esecuzione delle istruzioni di comunicazione con il sensore fintanto che non sia stato stabilito il protocollo da usare:

```
if(SCIPVersion==0) continue;
```

Una volta stabilita la versione di protocollo da usare, il metodo controlla periodicamente la presenza di connessione al sensore e manda il comando di lettura "GD" e ricevendo la relativa risposta:

```
if (!writeString(myRequestString))
    {
      ArLog::log(ArLog::Terse, "Could not send request distance reading to
urg");
      continue;
    }

    if (!readLine(buf, sizeof(buf), 10000) ||
       strncasecmp(buf, "GD", strlen("GD")) != 0)
    {
      ArLog::log(ArLog::Normal, "%s: Did not get distance reading response",
              getName());
      continue;
    }

    if (!readLine(buf, sizeof(buf), 10000) ||
       strncasecmp(buf, "00", strlen("00")) != 0)
         {
      ArLog::log(ArLog::Normal,  "%s::blockingConnectSCIP1:  Bad  status  on
distance reading response",
              getName());
      continue;
         }
```

La lettura comprende un timestamp (che per prova abbiamo anche stampato e, dopo esserci accertati della sua correttezza, abbiamo deciso di commentare) e un campo “sum” per il controllo degli errori (che per ora viene ignorato):

```
if(!readLine(buf, sizeof(buf), 10000))
                {  ArLog::log(ArLog::Normal,  "%s::  Cannot  read  Timestamp
string",getName());
                   continue;
                }
// Il Timestamp e' di 4 byte (Four -character Encoding; sum viene ignorato,
per ora)
       buf[4]='\0';
       int ts[4];
         for (int i=0; i<4; i++)
         {  ts[i]=buf[i] - 0x30;
//            printf("%d ",ts[i]);
         }
//     printf("\n");
```

Infine viene effettuata la vera e propria lettura delle distanze restituite dal sensore:

```
while (readLine(buf, sizeof(buf), 10000))
    {
      if (buf[0] == '\n' || buf[0] == '\r')
      {
       myReadingMutex.lock();
       myReadingRequested = readingRequested;
       myReading = reading;
       myReadingMutex.unlock();
       if (myRobot == NULL)
         sensorInterp();
       break;
      }

      // chop off the \n or \r and the checksum
      if (buf[0] != '\0')
      {
       buf[strlen(buf) - 2] = '\0';
       reading += buf;
      }
    }
```

Come si capisce dal codice precedente, il carattere LF e il checksum presenti in coda alla risposta ricevuta dal sensore vengono eliminati.

3.1.7. Il metodo sensorInterpr

Questo metodo si preoccupa di convertire le misure di distanza ricevute dal sensore in formato ASCII secondo le regole stabilite dal protocollo SCIP 2.0 e le salva nella variabile range. Di seguito viene riportato solo l’estratto di codice significativo e modificato per essere adattato alla codifica a 3 caratteri:

```
for (it = myRawReadings->rbegin(), i = 0;
it != myRawReadings->rend() && i < len - 1;
it++, i += 3)
{
       ignore = false;
       big = reading[i] - 0x30;
       center = reading[i+1] - 0x30;
       little = reading[i+2] - 0x30;
       range = (big << 6 | center);
       range = (range << 6 | little);
       if (range < 20)
       {
             range = 4096;
       }
       sReading = (*it);
       sReading->newData(range, pose, encoderPose, transform, counter,
             time, ignore, 0);
}
```

Esempio: supponiamo che una delle letture restituite dal sensore sia rappresentata dalla stringa ASCII “1Dh”. L’equivalente esadecimale di questa stringa (carattere per carattere) è 31_h 44_h 68_h. Le regole del

protocollo SCIP 2.0 (e anche del protocollo SCIP 1.1) stabiliscono che a tale valore debba essere sottratto 30_h quindi i nuovi valori saranno 1_h 14_h 38_h. L'equivalente binario di questi valori è 000001_2 010100_b 111000_b che, dopo aver fatto gli opportuni scorrimenti a sinistra di 6 bit, possono essere uniti ottenendo il seguente valore: 00000101010011000_b. L'equivalente binario di questa misurazione è 5432mm.

Il fatto che se una misura è più piccola di 20mm essa venga forzata a 4096mm è stato ereditato dalla versione originale della classe ArUrg.cpp.

3.2. Demo.cpp

Per comprendere e testare appieno la classe ArUrg modificata, abbiamo esaminato il codice del programma demo.cpp (si trova in /usr/local/Aria/examples) di cui illustriamo velocemente le parti salienti che riguardano il sensore laser.

```
88: ArLaserConnector laserConnector(&parser, &robot, &robotConnector)
```

viene creato l'oggetto laserConnector di classe ArLaserConnector passando 3 parametri: in particolare con il primo parametro passiamo al costruttore gli argomenti specificati nella riga di comando quando lanciamo il programma demo (che riguardano proprio il sensore laser). Per permettere al programma di collegarsi correttamente al laser, infatti è necessario avviare il programma demo aggiungendo i seguenti argomenti:

```
./demo -laserType urg -lp /dev/ttyACM0 -lpt serial –connectLaser
```

Tali argomenti riguardano il tipo di laser, la porta attraverso la quale è collegato al computer a bordo di MORGUL, il tipo di porta (seriale o tcp).

La classe ArLaserConnector serve proprio a stabilire una connessione con il sensore basandosi sugli argomenti della riga di comando.

```
129: if(!laserConnector.connectLasers(false,false,true))
                { printf("Could not connect to lasers… exiting\n");
                   Aria::exit(2); }
```

Il metodo connectLasers viene usato per il collegamento automatico dei sensori laser al robot e restituisce false in caso di errori.

```
152: ArModeLaser laser(&robot, "laser",'l',L''):
```

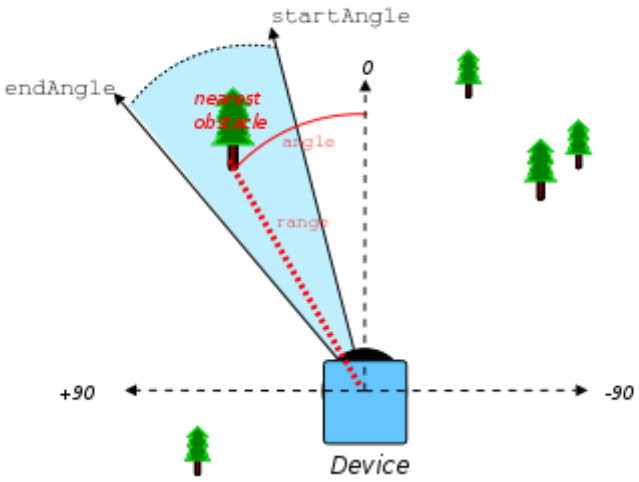
Questo oggetto aggiunge una modalità di utilizzo al programma "demo" basata sul laser. Il costruttore della classe ArModeLaser fondamentalmente ricerca il sensore laser collegato al robot e crea un binding fra due tasti della tastiera che permettono di selezionare il modo di utilizzo del laser; il programma demo fornisce inoltre la possibilità di 2 modi di utilizzo delle letture laser:

1) Modalità "far reading" che dovrebbe restituire la distanza dell'ostacolo più vicino e dell'ostacolo più lontano e le loro rispettive posizioni angolari;
2) Modalità "middle reading with reflectivity" che dovrebbe restituire la distanza dell'ostacolo posto frontalmente al robot dando anche una valutazione della sua riflettività (se il sensore è un sensore in grado di farne il rilevamento).

Esaminando attentamente il codice del metodo ArModeLaser.userTask si capisce che le misure restituite dal sensore vengono pre-elaborate in modi diversi a seconda della modalità scelta.

Nella modalità "far-reading" la misura di distanza minima (close) viene acquisita invocando il metodo currentReadingPolar della classe ArRangeDevice; tale metodo restituisce la distanza dell'ostacolo più vicino nella regione polare passata come argomenti. Per esempio:

```
double angle;
double range = laser.currentReadingPolar(startAngle, endAngle, &angle);
```



restituisce la distanza dell'ostacolo più vicino all'interno della regione angolare compresa tra "startAngle" ed "EndAngle" (in senso antiorario); nella variabile "Angle" viene salvato il corrispondente angolo. Tale misurazione tiene conto del posizionamento del robot (pose).

La misura della distanza dall'ostacolo più lontano, invece, viene acquisita semplicemente leggendo dal buffer del dispositivo il valore più grande senza tener conto del posizionamento del robot.

Nella modalità "middle reading with reflectivity", la stampa delle misure di distanza avviene utilizzando direttamente le letture acquisite dal buffer del sensore (raw readings). Il funzionamento della modalità è molto semplice: il sensore acquisisce le distanze degli ostacoli rilevati nell'intera area di cui è in grado di eseguire la scansione e le salva nel suo buffer interno; il metodo getRawReadings della classe ArLaser legge questo buffer e inserisce le varie letture di distanza in una lista di oggetti di tipo ArSensorReading.

Dalla lista di oggetti viene scelto quello che rappresenta la lettura più piccola corrispondente all'ostacolo più vicino; le informazioni stampate a video oltre, naturalmente, alla distanza sono l'angolo corrispondente e la riflettività dell'ostacolo rilevato (se il sensore lo permette). Come misura "middle" (cioè quella riferita alll'ostacolo posto frontalmente al sensore) viene presa quella corrispondente all'oggetto centrale della lista (assumendo che il sensore acquisisca un'area simmetrica rispetto alla sua posizione frontale).

Abbiamo rilevato che questa seconda modalità, a differenza della prima, restituisce dei valori corretti e quindi abbiamo deciso di scrivere il nostro programma provalaser.cpp andando a leggere i dati acquisiti dal sensore tramite il metodo getRawReadings() della classe ArLaser.

3.3. Il programma provalaser.cpp

I primo test di funzionamento delle modifiche apportate alla classe ArUrg sono stati effettuati utilizzando il programma Demo.cpp. Tuttavia, a causa delle diverse modalità di acquisizione e manipolazione delle misure di distanza operate dai metodi della classe ArModes, si è ritenuto più opportuno creare un semplice programma che stampasse a video le misure "raw" effettuate dal laser. Di questo programma ne sono state create due versioni: la prima versione (che non verrà esaminata in dettaglio in questo documento), stampa a monitor le misure di distanza lette dal sensore sull'intero arco di circonferenza coperto (-135°, +135°); l'output dei questo prima versione è di difficile lettura in quanto le misurazioni sono davvero tante. Per ovviare a questo inconveniente è stata creata una seconda versione del programma che stampa a monitor solo 7 misure di distanza distribuite sull'arco di circonferenza che va da -90° a +90° distanziate da archi di circa 30°.

Di seguito riportiamo il codice sorgente del programma provaLaser2 contenuto nel file prova.cpp.

La parte iniziale del codice è del tutto simile a quella del programma "caramelle.cpp": Aria viene inizializzato, viene creato il collegamento il robot e lanciato il metodo runAsync().

```
#include <string>
#include "provalaser.h"

/* main function */
int main(int argc, char **argv)
{
// robot and devices
       ArRobot robot;

// initialize aria and aria's logging destination and level
       Aria::init();
       ArArgumentParser parser(&argc, argv);
       parser.loadDefaultArguments();
       ArLog::init(ArLog::StdErr, ArLog::Normal);

// connector nuovo al posto del simple
   ArRobotConnector robotConnector(&parser, &robot);

//metodo nuovo di tentativo di connessione
if (!robotConnector.connectRobot())
  {
    // Error connecting:
    // if the user gave the -help argumentp, then just print out what
happened,
    // and continue so options can be displayed later.
    if (!Aria::parseArgs() || !parser.checkHelpAndWarnUnparsed())
    {
      ArLog::log(ArLog::Terse, "Could not connect to robot, will not have
parameter file so options displayed later may not include everything");
    }
    // otherwise abort
    else
    {
      ArLog::log(ArLog::Terse, "Error, could not connect to robot.");
      Aria::logOptions();
      Aria::exit(1);
       }
  }

robot.runAsync(true);
```

Come suggerito nella documentazione Aria a proposito della classe ArLaserConnecttor, dopo aver invocato il metodo runAsync dellaclasse ArRobot, viene affettuato il collegamento al laser. Si ricorda che il metodo connectLasers della classe ArLaserConnector richiama a sua volta il metodo blockingConnect quindi non c'è bisogno di una seconda chiamata esplicita di questo metodo.

```
ArLaserConnector laserConnector(&parser, &robot,
&robotConnector,true,ArLog::Verbose);
  if (!laserConnector.connectLasers(false, false, true))
  {
    printf("Could not connect to lasers... exiting\n");
    Aria::exit(2);
  }

// cerco il laser sul robot
       ArLaser *laser;
       for (int i = 1; i <= 10; i++)
       {
          if(robot.findLaser(i) != NULL) {
                laser=robot.findLaser(i);
                break; }
       }
       // Mappa dei laser installati nel robot
       std::map<int, ArLaser *>* laserMap = robot.getLaserMap();

       // Scrivo il nome del laser così sono sicuro che laser punti al
dispositivo giusto!
       printf("Nome del laser: %s\n",laser->getName());
       printf("Numero di laser installati (dim vettore laserMap): %d\n",
laserMap->size());

ArLaserConnector::connectLasers()
```

```
	// non serve richiamare la blockingConnect()... lo ha già fatto il
metodo connectLasers().
	//laser->blockingConnect();
```

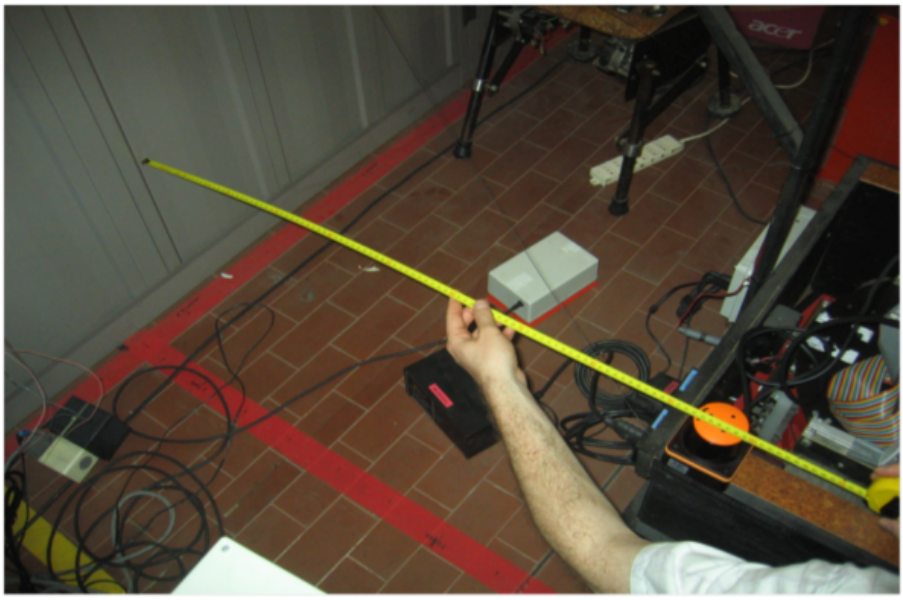
L'estratto di codice seguente recupera dal buffer del sensore le misure di distanza in corrispondenza degli angoli che stanno a circa -90°, -60°, -30°, 0°, 30°, 60°, 90°. Gli angoli di acquisizione non sono precisi perché viene presa la scelta del dato da stampare avviene tramite l'indice della lista di letture e non tramite l'angolo di acquisizione: gli scostamenti si introducono a causa della risoluzione angolare del sensore. Tra le varie prove effettuate, ne sono state eseguite alcune in cui venivano stampate anche le misure di distanza che il metodo getIgnoreThisReading (che semplicemente legge il valore di un attributo booleano dell'oggetto di classe ArSensorReading rappresentante una lettura) ci suggeriva di ignorare. Abbiamo riscontrato che anche le letture che dovrebbero essere ignorate in realtà sono corrette.

```
	double dist, angle;
    // getRawReadings restituisce una lista di oggetti di tipo
ArSensorReading.
    const std::list<ArSensorReading *> *rawReadings;
    // per navigare nella lista c'è bisogno di un iterator
    std::list<ArSensorReading *>::const_iterator rawIt;

	// VETTORE ArPoseWithTIme
	const std::list<ArPoseWithTime *> *readings;
    std::list<ArPoseWithTime *>::const_iterator it;

	while(robot.isRunning()) {
		ArUtil::sleep(500);
		robot.lock();
		if (!laser->isConnected())
		  {
		  ArLog::log(ArLog::Terse, "\n\nLaser mode lost connection to
the laser.");
		  ArLog::log(ArLog::Terse, "Select that laser or laser mode
again to try reconnecting to the laser.\n");
		  }
		laser->lockDevice();
		rawReadings = laser->getRawReadings();
		readings = laser->getCurrentBuffer();

		// salvataggio di 7 letture (compresa la frontale) sia per gli
		// angoli che per misure di distanza. Letture da -3 a +3
		double distSubsetReadings[7];
		double angleSubsetReadings[7];
		int indexStep = 30;
		// indice misura frontale ( circa 0°)
		int middleIndex = rawReadings->size() / 2;
		if (rawReadings->begin() != rawReadings->end())
		{
			int i; int indexToRead = -3;
			int ignoreCount = 0;
			for (rawIt = rawReadings->begin(), i=0; rawIt !=
rawReadings->end(); rawIt++, i++)
			{
// getIgnoreThisReading dive se la lettura è avvenuta con errori e quindi
// DOVREBBE essere ignorata!
				if ((*rawIt)->getIgnoreThisReading()) {
					ignoreCount++;
			//		continue;
				}

				if(i == (middleIndex + indexToRead*indexStep)) {

					distSubsetReadings[indexToRead+3] =
(*rawIt)->getRange();
					angleSubsetReadings[indexToRead+3] =
(*rawIt)->getSensorTh();
					if(indexToRead == 3)
						break;
					else
						indexToRead++;
				}
			}
```

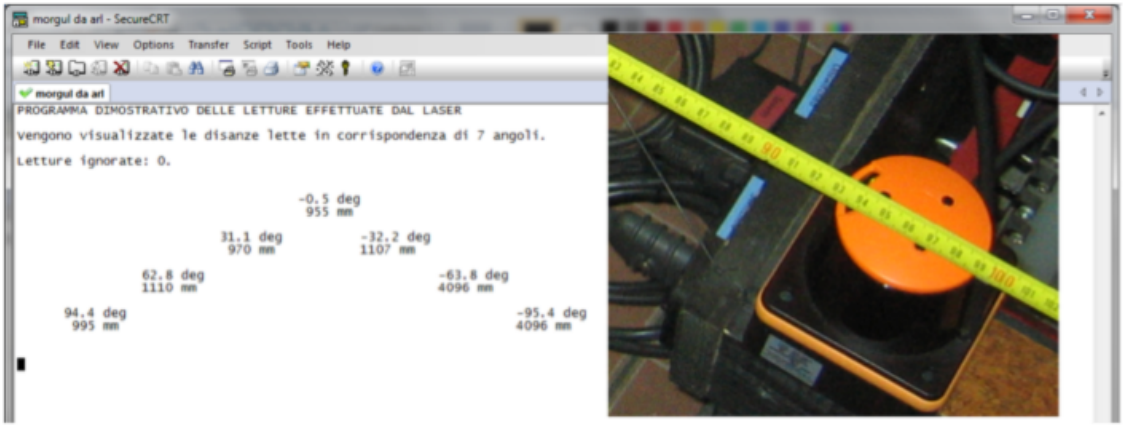
Il rimanente estratto di codice serve solo alla vera e propria stampa a video delle letture implementando un minimo di formattazione dell'output.

```
                        system("clear");
                        printf("PROGRAMMA DIMOSTRATIVO DELLE LETTURE EFFETTUATE
DAL LASER\n\n");
                        printf("Vengono visualizzate le disanze lette in
corrispondenza di 7 angoli.\n\n");
// printf("Letture ignorate: %d.\n\n\n", ignoreCount);
                        int centerSpaces = 40;
                        //stampo lettura centrale
                        printf("%*.1f deg\n%*.0f mm\n\n", centerSpaces,
angleSubsetReadings[3], centerSpaces, distSubsetReadings[3]);

                        // stampo le coppie di letture successive (una a sx, una
a dx)
                        int dxSpaces, sxSpaces;
                        for(int i=1; i<4; i++) {
                                sxSpaces = centerSpaces - i*10;
                                dxSpaces = centerSpaces + i*10 - 5;
                                printf("%*.1f deg%*.1f deg\n", sxSpaces,
angleSubsetReadings[3-i], dxSpaces-sxSpaces, angleSubsetReadings[3+i]);
                                printf("%*.0f mm%*.0f mm\n", sxSpaces,
distSubsetReadings[3-i], dxSpaces-sxSpaces, distSubsetReadings[3+i]);
                                printf("\n");
                        }
                        printf("\n");
                }
                laser->unlockDevice();
                robot.unlock();
        }// while
        printf("Dimensione letture rawreadings: %d\n", rawReadings->size());
        printf("Dimensione letture arPoses: %d\n", readings->size());
        Aria::shutdown();
        return 0;
}
```

A puro titolo di esempio riportiamo una fotografia della misura di distanza acquisita dal nostro programma confrontata con la misura "reale" acquisita tramite il tradizionale metro[1]: come si vede, a fronte di una misura "reale" di circa 95/96 cm il sensore rileva una distanza di 955 mm che riteniamo più che soddisfacente.



[1] Durante la misura il sensore era stato posizionato sul bordo superiore della stazione di ricarica, rivolto verso il lato est.



3.4. Adattamento delle ArActionAvoidFront per l'uso del laser

L'analisi fatta nei due paragrafi precedenti ci ha permesso di sollevare un ulteriore problema che affligge la libreria ARIA, ovvero l'errata interpretazione dei dati (corretti) rilevati dal laser. Ciò è particolarmente evidente quando tali dati vengono utilizzati dal robot per spostarsi nell'ambiente.

Durante le prove effettuate con il programma "demo.cpp", oltre ad aver rilevato la completa inaffidabilità delle letture del laser nella modalità "far-reading" rispetto alla modalità "middle" (vedi par. 3.2), si poteva notare che il programma non si comportava bene nemmeno nella modalità "wander mode". In tale modalità il robot dovrebbe spostarsi nello spazio di lavoro cercando di evitare gli ostacoli; tuttavia nel nostro caso il robot persisteva in un movimento rotatorio attorno al proprio asse senza compiere nessun movimento, nonostante lo spazio circostante fosse libero da ostacoli.

Abbiamo pensato che questo comportamento fosse causato da una cattiva gestione del laser, quindi abbiamo eseguito un'ulteriore prova con il programma "caramelle.cpp". Tale programma, che era stato impiegato con successo sul robot Tobor (che monta un Laser Hokuyo di tipo URG-04LX che usa il protocollo SCIP1.1), permette al robot di vagare per l'ambiente evitando gli ostacoli frontali e laterali. Il risultato sul campo è stato il medesimo di quello riscontrato con il programma demo: il robot effettuava solo una rotazione su se stesso.

Abbiamo quindi analizzato il codice contenuto nel file caramelle.cpp. Innanzitutto abbiamo ritrovato le stesse chiamate per connessione al laser che erano presenti nel programma demo, ovvero la costruzione dell'oggetto ArLaserConnector e la chiamata al metodo connectLasers. È mportante a questo punto far notare che la costruzione di un "laser connector" implica anche la costruzione di un oggetto ArRangeDevice (nel nostro caso un ArUrg), che va ad aggiungersi alla lista dei "range device" a disposizione del robot. Oltre ai laser, il programma aggiunge altri dispositivi di lettura di distanza, come riportato nelle seguenti linee di codice:

```
morgul.addRangeDevice(&sonar);
morgul.addRangeDevice(&bumpers);
morgul.addRangeDevice(&ir);
```

Vengono aggiungi i sonar, i bumper e il dispositivo ad infrarossi. Di seguito riportiamo le istruzioni che permettono al robot di evitare gli ostacoli:

```
ArActionAvoidFront avoidFront("avoidFront",450,200,15);
[…]
morgul.addAction(&avoidFront,79);
```

La action "ArActionAvoidFront" permette al robot di evitare gli ostacoli forzando il robot ad eseguire una curva rispetto alla direzione che stava seguendo. Nei parametri del costruttore è possibile specificare a quale distanza dall'ostacolo iniziare a sterzare, l'angolo di sterzata e la velocità con cui effettuare la curva. Consultando la documentazione di Aria, abbiamo avuto la conferma che la action utilizza le letture di qualsiasi sensore di distanza disponibile che sia stato precedentemente aggiunto al robot. Dopo aver ispezionato il codice, abbiamo notato che anche il programma demo.cpp in modalità wander usa la medesima action per evitare gli ostacoli. Per cercare di isolare il problema, abbiamo provato a

modificare il programma in modo che non venisse aggiunto il device laser; eseguendolo in questa configurazione si è visto che il robot procedeva evitando correttamente gli ostacoli.

Per cercare quindi di capire come risolvere questo problema abbiamo deciso di addentrarci nell'analisi del comportamento interno della action. Di seguito riportiamo la parte iniziale del metodo ArActionAvoidFront::fire:

```
double dist, angle;
if (currentDesired.getDeltaHeadingStrength() >= 1.0)
  myTurning = 0;
myDesired.reset();
dist = (myRobot->checkRangeDevicesCurrentPolar(-70, 70, &angle)
     - myRobot->getRobotRadius());
```

Lo scopo del metodo fire è di restituire una ArActionDesired che contenga il comportamento (in termini di spostamenti) che il robot dovrà tenere per evitare l'eventuale ostacolo. Dall'estratto di codice è interessante notare che nella variabile reale "dist" viene salvata la distanza dall'ostacolo più vicino, mentre in "angle" l'angolo a cui è stato rilevato tale ostacolo. Nel prosieguo del metodo verrà stabilito, in base alla soglia passata al costruttore, se e in che modo il robot dovrà sterzare per evitare l'ostacolo.

Abbiamo ipotizzato che il problema fosse da ricercare nella funzione ArRobot::checkRangeDevicesCurrentPolar: la documentazione di Aria a tal proposito dice che la funzione trova la lettura di distanza più vicina a partire dalle letture correnti di tutti i sensori di distanza, all'interno di una regione polare definita dall'intervallo angolare passato come parametro. In particolare itera fra i sensori di distanza registrati, richiama, per ognuno di essi, il metodo ArRangeDevice::currentReadingPolar() per effettuare la misurazione ed infine restituisce la distanza minima rilevata.

Il metodo currentReadingPolar() è lo stesso metodo usato per restituire le letture del laser nella modalità ArModeLaser (letture che abbiamo dimostrato essere inaffidabili nelle prove effettuate sul programma demo.cpp). Siamo andati quindi a curiosare nell'implementazione di tale metodo, percorrendo la catena di chiamate che si rincorrono fra i componenti della libreria. Il punto d'arrivo è il metodo ArPoseWithTime::findDistanceTo(), il quale dovrebbe restituire la distanza letta dal sensore in corrispondenza di una certa posizione angolare, includendo informazioni riguardanti la posizione del robot e il tempo di rilevamento. La lista di tali letture "filtrate" è mantenuta nell'attributo ArRangeDevice::myCurrentBuffer e costruita, per ogni lettura del dispositivo, a partire dalla lista di letture non filtrate ArSensorReading, mantenuta nell'attributo ArRangeDevice::myRawReadings. Nel caso particolare del sensore laser, la classe ArLaser è responsabile del popolamento della lista di letture filtrate a partire dalle "raw".

La nostra ipotesi è che il filtraggio delle letture provenienti dal laser non venga eseguito correttamente; ciò è sufficiente per preferire l'uso delle letture non filtrate. Del resto gli stessi progettisti di Aria nella loro documentazione specificano che la recente introduzione dei metodi di accesso alle letture raw è utile soprattutto per dispositivi laser. A riprova di quanto detto abbiamo effettuato una verifica eseguendo il programma Provalaser.cpp sulle letture filtrate. I risultati sperimentali hanno dimostrato che l'insieme di letture, oltre ad essere fortemente rimaneggiato rispetto all'originale, contiene anche alcuni valori decisamente fuorvianti.

A questo punto del lavoro eravamo in possesso di letture "raw" corrette provenienti dal laser UHG-08LX, ma il robot non era in grado di evitare gli ostacoli poiché le action prendevano in considerazione le letture filtrate, le quali contenevano errati valori di distanza. La soluzione più immediata ci è sembrata quella di intervenire nelle porzioni del codice Aria in cui viene fatta la ricerca della distanza minima nella regione polare di interesse: è necessario differenziare la modalità di acquisizione della distanza nel caso del dispositivo laser, per il quale si deve forzare l'utilizzo delle letture non filtrate, rispetto a tutti gli altri.

Si tratta quindi di stabilire a quale livello della gerarchia della classi è opportuno effettuare le modifiche. Inizialmente eravamo interessati ad avere una soluzione comoda e funzionante, quindi abbiamo modificato direttamente la action interessata, in particolare abbiamo modificato il comportamento del metodo ArAction::fire() (per i dettagli si veda il prossimo paragrafo).

Successivamente abbiamo pensato ad una soluzione migliore dal punto di vista dell'ingegneria del software, usando codice più generico e riusabile, ma che comporta modifiche ad una profondità

maggiore nella libreria Aria. Tali modifiche interessano la classe ArRobot, oltre che la action, e verranno maggiormente dettagliate nel paragrafo 3.4.2.

Per le modalità di installazione messa in opera di entrambe le modalità si rimanda al paragrafo 4.2.4.

3.4.1. ArActionAvoidFrontLaser.cpp

Come già anticipato, il compito della "action avoid front" è di ritornare al robot la prossima azione da intraprendere (action desired) che è opportunamente calibrata sulla base dell'eventuale ostacolo da evitare. A tale scopo viene invocato periodicamente il metodo virtual ArAction::fire(), che ha il compito di leggere i dati dai sensori e di regolare di conseguenza le azioni da intraprendere.

Abbiamo creato il file ArActionAvoidFront**Laser**.cpp, con il relativo header file ArActionAvoidFrontLaser.h che rispecchia la struttura della classe originale. Rispetto al costruttore originale abbiamo aggiunto due parametri relativi agli angoli che delimitano la regione polare che ci interessa valutare per la ricerca di ostacoli. Sono parametri opzionali e assumono valore di default pari a (-70.0,70.0). Tale valore viene poi copiato negli attributi di classe myStartAngle e myEndAngle, anch'essi aggiunti rispetto alla classe originale.

```
public:
  // Costruttore
  AREXPORT ArActionAvoidFrontLaser(
              const char *name = "avoid front obstacles",
              double obstacleDistance = 450,
              double avoidVelocity = 200,
              double turnAmount = 15,
              bool useTableIRIfAvail = true,
              double startAngle = -70,
              double endAngle = 70);
```

Ci siamo quindi concentrati sul metodo ArActionAvoidFront::fire() e le nostre modifiche hanno interessato solo la parte iniziale che acquisisce dai dispositivi le informazioni di distanza e angolo dell'ostacolo più vicino e salvarle nelle variabili dist e angle. Da lì in poi il comportamento della action rimane quello originale: se la distanza letta è sotto la soglia vengono calcolati l'angolo di sterzata e la velocità con cui il robot dovrà evitare l'ostacolo, e infine ritornata la action desiderata.

Nella parte iniziale del metodo ci sono dichiarazioni e inizializzazioni di variabili. Degna di nota è la variabile closest: essa tiene traccia della distanza dell'oggetto più vicino letta dal sensore corrente e viene confrontata con l'ultima lettura effettuata (ed eventualmente aggiornata) ad ogni iterazione del ciclo; essa è inizializzata ad un valore arbitrario molto grande (rispetto alla portata dei range device), in modo che la prima lettura sia sicuramente salvata come la più vicina. La variabile *it* conterrà l'iteratore della lista di sensori di distanza disponibili sul robot.

```
myDesired.reset();

double dist; double *angle;
double closest = 32000; double closeAngle;
double tempDist, tempAngle;
std::list<ArRangeDevice *>::const_iterator it;
ArRangeDevice *device;
bool foundOne = false;
const ArRangeDevice *closestRangeDevice = NULL;
```

Prima di entrare nel ciclo che passa in rassegna i vari sensori scandendo la lista `ArRobot::myRangeDeviceList`, è necessario salvare il nome del sensore laser in modo da poterlo riconoscere all'interno del ciclo trattarlo come un caso particolare rispetto agli altri sensori. Per fare ciò ci siamo serviti del metodo `ArRobot::findLaser(int)`, la quale restituisce la stringa identificativa di un dispositivo laser. Si assume che ogni laser venga aggiunto al robot con un indice identificativo intero, anche se nel nostro caso il laser è unico. Con un semplice ciclo a conteggio si salva in una variabile la stringa corrispondente al nostro laser (ovvero "URG_1").

```
ArLaser *laser;
char* laserName;
for (int i = 1; i <= 10; i++)
  {
    if ((laser = myRobot->findLaser(i)) != NULL)
    {
```

```
                laserName = new char[strlen(laser->getName())+1];
                strcpy(laserName, laser->getName());
            }
        }

        std::list<ArRangeDevice *>* deviceList = myRobot->getRangeDeviceList();
        int j =0;
        for (it = deviceList->begin(); it != deviceList->end(); ++it)
           {
               device = (*it);
               device->lockDevice();
```

Una volta entrati nel ciclo che considera i vari sensori si fa subito un controllo sul nome del sensore corrente. Se si tratta del dispositivo laser allora si controlla la lista di rawRadings del device e si estrae il valore minimo di distanza. Notare che con l'istruzione condizionale

```
(tempDist = (*rawIt)->getRange()+ getRobotRadius()) < closest)
```

si controlla se la i-esima misura letta dal laser è inferiore al minimo attuale closest, aggiungendo però la quantità getRobotRadius() che rappresenta l'ingombro radiale del robot. Dopo alcune prove sperimentali ci siamo resi conto che è consigliabile tenere conto di tale quantità per uniformare le rilevazioni del laser con quelle degli altri dispositivi (che hanno come riferimento il centro geometrico del robot). Prima della fine del corpo dell'if più esterno c'è l'istruzione `continue`, in modo da passare al device successivo (che quindi non sarà un laser).

```
if(laserName != NULL && if(!strcmp(laserName,device->getName()))
        {       // codice eseguito solo se il device è un laser
        const std::list<ArSensorReading *> *rawReadings;
        std::list<ArSensorReading *>::const_iterator rawIt;

        rawReadings = device->getRawReadings();

        if (rawReadings->begin() != rawReadings->end())
              {
              for (rawIt = rawReadings->begin(); rawIt != rawReadings->end();
                    rawIt++)
                    {
                    if (rawIt == rawReadings->begin() ||
                       (tempDist = (*rawIt)->getRange()+ getRobotRadius())
                           < closest)
                           {
                           if((tempAngle=(*rawIt)->getSensorTh()) >
                           myStartAngle && tempAngle<myEndAngle)
                                  {
                                  closest = tempDist;
                                  closeAngle = tempAngle;
                                  foundOne = true;
                                  closestRangeDevice = device;
                                  }
                           }
                     }
                    }
                    printf("Closest letta dal laser: %f\n", closest);
                    device->unlockDevice();
                    continue;
              }

        }
```

La seconda parte del ciclo for principale viene eseguita solo se l'iteratore è posizionato su un device diverso dal laser. Questa parte di codice è rimasta invariata rispetto alla versione originaria della action e non fa altro che leggere la distanza minima sfruttando il metodo `ArRangeDevice::currentReadingPolar()`.

```
        tempDist = device->currentReadingPolar(startAngle, endAngle,
        &tempAngle);
        if (!foundOne ||tempDist < closest)
            {

              if (!foundOne)
              {
```

```
                closest = device->currentReadingPolar(startAngle,
endAngle, &closeAngle);
                closestRangeDevice = device;
        }
        else
        {
                closest = tempDist;
                closeAngle = tempAngle;
                closestRangeDevice = device;
        }
        foundOne = true;
}
device->unlockDevice();
```

3.4.2. Programma dimostrativo wander.cpp

Al fine di testare il corretto funzionamento della nuova action si è deciso di scrivere il programma *wander.cpp*, che si basa interamente sull'esempio di *caramelle.cpp* precedentemente citato ma che ne mantiene solo le parti essenziali al comportamento desiderato. In output viene stampate, ed aggiornate ad ogni chiamata della action, unicamente le informazioni di distanza e posizione angolare dell'oggetto più vicino al robot. Riportiamo di seguito la riga di codice che fa riferimento alla costruzione della nuova ArActionAvoidFrontLaser:

```
ArActionAvoidFrontLaser avoidFront("avoidFront",450,200,15, true, -70,
70);
```

Per quanto riguarda lo header file del nostro programma wander.cpp, oltre all'inclusione della libreria Aria, è necessario includere lo header file della nuova action. Il file sorgente della Action si trova nella stessa directory del programma che la usa.

```
#include "ArActionAvoidFrontLaser.h"
```

Il programma dimostrativo funziona egregiamente e il robot evita gli ostacoli come richiesto. Abbiamo effettuato la prova anche escludendo i sonar, in modo che il robot usasse unicamente il laser per rilevare gli ostacoli. Anche in questo caso la navigazione non ha avuto problemi, eccetto in presenza di ostacoli posizionati ad una quota troppo bassa per essere visti dal laser.

3.4.3. Estensione del comportamento di ArRobot.cpp

Una volta appurato che l'approccio adottato con ArActionAvoidFrontLaser è corretto, si è pensato di implementare il comportamento di lettura di distanza polare ad un livello più basso della gerarchia, in modo che possa essere riusato più facilmente da altre action o programmi utente. In altre parole anziché integrare nella action il comportamento desiderato, sarebbe più utile modificare il comportamento del metodo `ArRobot::checkRangeDevicesCurrentPolar()` direttamente nella classe ArRobot.

In tal modo il codice della action (e di tutti gli altri programmi che accedono a tali funzionalità) guadagna in termini di leggerezza, genericità e flessibilità. Tuttavia per ovvie ragioni di retro-compatibilità dell'intera libreria abbiamo preferito mantenere il vecchio metodo ed aggiungerne uno nuovo, ovvero `ArRobot::checkRangeDevicesLaserCurrentPolar()`. Tale metodo ha la stessa firma del metodo originale, ma nell'implementazione abbiamo replicato fedelmente i passaggi descritti nel paragrafo precedente per il metodo fire, a meno di qualche adattamento di scope. Il prototipo è stato aggiunto all'header file ArRobot.h.

Per avere modo di testare entrambi gli approcci in modo separato, abbiamo creato un'ulteriore action (ArActionAvoidFrontLaser1) che nel proprio metodo fire effettua la chiamata al nuovo metodo della classe robot.

```
dist = (myRobot->checkRangeDevicesLaserCurrentPolar(-70, 70, &angle) -
myRobot->getRobotRadius());
```

4. Modalità operative

In questo paragrafo vengono illustrati tutti i passi necessari per poter utilizzare con successo il laser range scanner.

4.1. Componenti necessari

4.1.1. Laser Range Scanner Hokuyo UHG-08LX

Il laser scanner UHG-08 ha le seguenti caratteristiche:

- ✓ intervallo di rilevamento: 30÷11000mm;
- ✓ angolo di rilevamento: 270°;
- ✓ angolo di risoluzione: 0.36°;
- ✓ tempo di scansione: 67msec/scansione;
- ✓ interfaccia: USB 2.0;
- ✓ protocollo: SCIP 2.0;
- ✓ alimentazione: 12V DC tramite alimentatore.

4.1.2. Libreria Aria

La libreria Aria più recente (attualmente la 2.7.2) è scaricabile dal sito del produttore Mobile Robots tramite comando:

$wget http://robots.mobilerobots.com/ARIA/download/current/ARIA-2.7.2.tgz

4.1.3. Calcolatore

Il lavoro svolto è stato pensato e sviluppato per funzionare in ambiente Linux. Il calcolatore deve pertanto essere dotato di una delle sue distribuzioni, in particolare si fa riferimento alla distribuzione Debian (installata sul calcolatore del robot MORGUL).

4.1.4. Il codice sviluppato

Per quanto riguarda le modifiche alla libreria Aria sono necessari i file sorgenti ArUrg.cpp e ArRobot.cpp (con il relativo header file ArRobot.h). Gli esempi sviluppati invece richiedono i file inclusi nelle sottodirectory "ProvaLaser" e "Wander", presenti nell'archivio relativo a questo lavoro. L'archivio del pacchetto sviluppato ha la seguente struttura di directory:

ModificheAria

ProvaLaser

Wander

L'archivio è scaricabile da http://www.ing.unibs.it/arl/ nella sezione relativa ai lavori conclusi dagli studenti.

4.2. Modalità di installazione

Per poter utilizzare correttamente l'applicazione è necessario svolgere i passi riportati di seguito.

4.2.1. Installazione ambiente di sviluppo

Installazione del compilatore gcc e dell'utility make, che permette una compilazione automatica di file sorgente:

$ sudo apt-get install gcc make

4.2.2. Installazione libreria Aria

Per l'installazione della libreria Aria si rimanda al lavoro svolto dai colleghi Ceriani, Gatti, Caffi, Tomson [4], in particolare al paragrafo 3.1 che spiega passo per passo come installare e compilare tale libreria. Si noti, infine, che tale lavoro è aggiornato alla versione 2.7.0 di Aria, mentre in questa sede si è utilizzata la 2.7.2.

4.2.3. Modifiche alla libreria Aria

Sostituzione delle classi ArUrg.cpp, ArRobot.cpp e dell'header file ArRobot.h presenti nella libreria Aria con quelle da noi modificate:

$ cp ModificheAria/ArUrg.cpp /usr/local/Aria/src

$ cp ModificheAria/ArRobot.cpp /usr/local/Aria/src

$ cp ModificheAria/ArRobot.h /usr/local/Aria/include

$ cd /usr/local/Aria/

$ make clean

$ make

4.2.4. Esempi dimostrativi

Per l'installazione dei due esempi acclusi al pacchetto è sufficiente portarsi nelle rispettive sottodirectory e digitare:

$ make clean

$ make

Si noti che il codice relativo alle action modificate è incluso nella directory *Wander*.

5. Conclusioni e sviluppi futuri

Grazie al lavoro svolto è ora possibile sfruttare le funzionalità della libreria Aria al fine di utilizzare il laser range scanner Hokuyo UHG-08LX, montato sul robot MORGUL, in modo da ottenere letture corrette che permettono al robot di evitare ostacoli presenti di fronte a sé.

Il nuovo metodo introdotto nella classe ArRobot facilita la realizzazione di action che necessitano di interfacciarsi con il nuovo laser di casa Hokuyo.

L'obiettivo principale, ovvero l'adattamento della classe ArUrg, è stato dunque raggiunto. L'intento iniziale tuttavia era quello di riuscire a creare una classe che fosse in grado di mantenere una retro-compatibilità con il vecchio protocollo, ma, date le difficoltà a cui siamo andati incontro, siamo stati costretti a rinunciarvi. La retro-compatibilità, peraltro, si potrebbe ottenere facilmente duplicando parecchie righe di codice ma, dal punto di vista dell'ingegneria del software, tale soluzione non è ideale. D'altra parte, progettare una gerarchia di classi che risolvesse il dilemma sarebbe stato troppo dispendioso in termini di tempo e fuorviante dagli scopi del lavoro. Pertanto, lo sviluppo di una struttura che riesca allo stesso tempo a mantenere la retro-compatibilità e a soddisfare i requisiti di buona progettazione rientra negli sviluppi futuri del lavoro.

Bibliografia

[1] Mori, Maejima, Santosh, Kawata: "URG Series Communication Protocol Specification", Hokuyo Automatic, 2 Febbraio 2004.

[2] Mori, Maejima, Kawata: "Communication Protocol Specification For SCIP2.0 Standard", Hokuyo Automatic, 10 Ottobre 2006.

[3] Aria Developer's Reference Manual: http://www.ing.unibs.it/~arl/docs/documentation/Aria%20documentation/Current/Aria-Reference/

[4] Ceriani L., Caffi A., Gatti S., Tomson N. M. .: “Guida all’installazione di Aria versione 2.7.0 su Acer Aspire One e collaudo su robot Tobor”, giugno 2009.

Indice

SOMMARIO **1**

1. INTRODUZIONE **1**

2. IL PROBLEMA AFFRONTATO **1**
2.1. Confronto tra i dispositivi URG-04LX e UHG-08LX **1**
2.2. Confronto tra i protocolli SCIP1.1 e SCIP2.0 **2**
2.2.1. Differenze generali 2
2.2.2. Comando di Informazioni di Versione 3
2.2.3. Comandi di abilitazione e disabilitazione del laser 4
2.2.4. Comando per impostare la velocità di comunicazione 5
2.2.5. Comandi di acquisizione dati 6

3. LA SOLUZIONE ADOTTATA **8**
3.1. Modifica della classe ArUrg **8**
3.1.1. Costruttore 9
3.1.2. setParams 9
3.1.3. Il metodo setParamsByStep 10
3.1.4. Il metodo blockingConnect 11
3.1.5. Il metodo blockingConnectSCIP2 12
3.1.6. Il metodo runThread 15
3.1.7. Il metodo sensorInterpr 16
3.2. Demo.cpp **17**
3.3. Il programma provalaser.cpp **18**
3.4. Adattamento delle ArActionAvoidFront per l'uso del laser **22**
3.4.1. ArActionAvoidFrontLaser.cpp 24
3.4.2. Programma dimostrativo wander.cpp 26
3.4.3. Estensione del comportamento di ArRobot.cpp 26

4. MODALITÀ OPERATIVE **27**
4.1. Componenti necessari **27**
4.1.1. Laser Range Scanner Hokuyo UHG-08LX 27
4.1.2. Libreria Aria 27
4.1.3. Calcolatore 27
4.1.4. Il codice sviluppato 27
4.2. Modalità di installazione **27**
4.2.1. Installazione ambiente di sviluppo 27
4.2.2. Installazione libreria Aria 28
4.2.3. Modifiche alla libreria Aria 28
4.2.4. Esempi dimostrativi 28

5. CONCLUSIONI E SVILUPPI FUTURI **28**

BIBLIOGRAFIA **28**

INDICE **30**