Università degli Studi di Padova
Facoltà di Ingegneria
DTG — Dipartimento di Tecnica e Gestione
——— dei Sistemi Industriali

Corso di Laurea Triennale in Ingegneria Meccanica
e Meccatronica curriculum Meccatronico

Tesi di laurea triennale

Implementazione di cROS per ambiente PLC B&R per applicazioni industriali

cROS Implementation for the B&R PLC Environment for Industrial Applications

23 Novembre 2016

Candidato:
Aldo Marchi
Matricola 1073695

Relatore:
Dr. Stefano Ghidoni [DEI — Dipartimento di Ingegneria dell'Informazione]

Correlatore:
Dr. Nicolò Boscolo [Azienda IT+Robotics]

Anno Accademico 2015–2016

Indice

Prefazione

La seguente tesi è stata seguita dal professor Ghidoni Stefano ed è stata svolta presso IT+Robotics, spin-off dell'Università di Padova.

Il progetto di tesi prevede il porting di cROS su sistemi PLC B&R, in modo da rendere i PLC compatibili con l'ecosistema ROS.

cROS è l'implementazione in Ansi C sviluppata da IT+Robotics compatibile con il protocollo middleware ROS (Robotics Operating System). La libreria è stata rilasciata con licenza opensource ed è disponibile al seguente indirizzo: https://github.com/it-robotics/cros.

Ho iniziato il mio lavoro di tesi documentandomi per capire e imparare ad usare il protocollo ROS in ambiente Linux per poi passare al software cROS. Il primo compito assegnatomi è stato quello di risolvere un bug nell'ecosistema che io stesso avevo riscontrato in una prima fase di test sul sistema operativo Linux. Dopo aver portato a termine il compito con successo, sono passato ad operare in ambiente PLC B&R e, in un primo momento, ho cercato di delineare i problemi e i relativi adattamenti che si sarebbero dovuti compiere per rendere operativo cROS in un PLC.

I principali successi ottenuti sono stati: adattare la parte di codice relativa al protocollo TCP/IP alle librerie fornite da B&R, invertire poi nella logica delle callback per eliminare i puntatori a funzione, e infine approntare i tipi di dati definiti nel software per poterli rendere disponibili all'utente.

Introduzione

0.1 ROS: Robot Operating System

ROS è un framework che permette di scrivere e gestire software destinato al mondo dell'automazione. Esso contiene varie librerie e tool con lo scopo di semplificare la scrittura di codice destinato alla robotica, e perciò è possibile implementare con maggiore facilità programmi robusti e complessi.

I vantaggi del framework sono principalmente dati dalla sua struttura: essenzialmente si compone di un nucleo centrale (core) e un numero variabile di nodi.

Il core si occupa della creazione dei collegamenti (topic) per la comunicazione tra i nodi, i quali non sono altro che moduli eseguibili, in parte scritti dall'utente, e in parte forniti all'interno dei ROS packages; questi svolgono una ben determinata funzione come, ad esempio, il controllo di una specifica area del robot, l'acquisizione di dati da sensori, l'elaborazione, etc... .

Questa struttura permette quindi un approccio "divide et impera", cioè offre la possibilità di gestire il robot separatamente nelle varie parti che lo compongono, e la suddivisione che si può ottenere semplifica notevolmente il controllo permettendo di concentrarsi sulle singole problematiche da risolvere.

Un altro aspetto importante da sottolineare è la modularità che acquisisce la struttura del codice e che rende molto agevole il lavoro in team:

ogni membro può occuparsi di una parte isolata di un ipotetico progetto senza doversi preoccupare troppo dell'integrazione finale nel sistema complesso, cosa che infatti viene svolta da ROS mediante la creazione e la gestione dei topic.

Inoltre, ROS supporta, per la scrittura dei nodi, sia il linguaggio C++ che il linguaggio Pyton, i quali possono essere usati contemporaneamente per due nodi distinti che devono comunicare tra di loro.

Questa compatibilità è resa possibile da ROS, il quale si interpone tra i due nodi occupandosi della loro interazione mediante i topic, come accennato sopra.

Un ulteriore vantaggio dato dal framework è l'alto grado di astrazione dall'hardware; è quindi possibile applicare un modello "black box": lo sviluppatore può concentrarsi sulla programmazione ad alto livello senza preoccuparsi delle particolari caratteristiche del robot a basso livello.

Vorrei accennare ora al concetto di callback nonostante sia un aspetto puramente tecnico di ROS, poiché, durante il mio lavoro di tesi, si è rivelato un ostacolo importante, come descriverò in seguito.

Le callback rendono possibile l'esecuzione parallela dei nodi che compongono ogni pacchetto ROS. Una callback non è altro che un metodo, sempre in attesa di un «determinato» tipo di messaggio (le cui caratteristiche sono definite al suo interno) pubblicato su un topic, ed è gestito parallelamente al resto del codice scritto dall'utente e invocato quando avviene l'evento. Ogni qual volta viene scritto qualcosa sul topic, la callback viene invocata.

Infine, è importante evidenziare il fatto che ROS, nell'ambito della ricerca, sia diventato ormai uno standard nello sviluppo di applicazioni robotiche; inoltre, essendo open source, ogni utente può contribuire alla creazione di nuovi pacchetti e in più può usufruire di quelli già esistenti condivisi da altri fruitori come lui, rendendo il sistema sempre aggiornato e ricco di soluzioni.

0.2 PLC

0.2.1 Sviluppo

I primi PLC nacquero nel 1968 a fronte della necessità di avere dei sostituti elettronici per sistemi relè cablati e, in seguito, ebbero un enorme successo nell'industria grazie alla loro alta versatilità; per questo fin da subito ci furono numerosi investimenti che portarono ad una notevole evoluzione degli stessi.

Negli ultimi anni però ha prevalso l'inerzia tipica del settore industriale che ha generalmente preferito la conservazione di controllori logici diventati ormai obsoleti, ma ancora funzionanti.

Questo è stato reso possibile grazie alla notevole longevità dei PLC, i quali riescono a rimanere operativi anche per vent'anni. Bisogna comunque sottolineare come le potenzialità dei più moderni controllori logico programmabili siano difficilmente sfruttabili dalle aziende principalmente a causa di due fattori: il primo è che un discreto numero di PLC viene gestito ancora da elettricisti, poiché inizialmente furono progettati per la messa a punto da parte di periti elettrici.

Questo dato di fatto, nonostante non sia un male in sé, preclude la possibilità di sfruttare le attuali potenzialità dei PLC: oggigiorno, per una programmazione adeguata di un controllore logico, sono necessarie delle conoscenze informatiche specifiche e non banali.

Il secondo fattore è dovuto proprio alle competenze specifiche richieste; infatti, anche un programmatore esperto ha bisogno di conoscenze incentrate su questo genere di sistemi. Come descriverò in seguito, i PLC hanno dei linguaggi propri ideati dalla casa produttrice che, a differenza di un comune computer, sono real-time; perciò tali sistemi devono non solo portare a compimento delle operazioni logiche, ma devono anche assicurare che queste ultime siano terminate entro un lasso di tempo ben preciso e piuttosto stringente.

0.2.2 Linguaggi di programmazione

Tutti i PLC per poter funzionare hanno bisogno di essere programmati cosa che viene tipicamente effettuata su un PC e solo successivamente il programma viene trasferito sulla macchina.

I linguaggi che si adoperano per la programmazione sono molteplici e dipendono dal PLC: il linguaggio più antico è quello a contatti (ladder diagram) che era anche il più usato fino a pochi anni fa, in quanto era la trasposizione

informatica dei circuiti elettrici usati dagli elettrotecnici.

Gli altri linguaggi grafici, oltre a quello a contatti, sono essenzialmente due: Sequential function chart (detto Diagramma funzionale sequenziale) e Function Block Diagram (detto Diagramma a blocchi funzionali). Esistono due linguaggi testuali: quello a lista di istruzioni e quello a testo strutturato. Il primo è un linguaggio assembler mentre il secondo è simile al C; attraverso questi due è possibile sfruttare appieno le potenzialità dei moderni PLC. Il più famoso linguaggio a lista di istruzioni è sicuramente STEP 7, che fu definito dal leader del mercato Simens, mentre la normativa IEC 1131-3 del 1993 ha standardizzato 5 linguaggi di programmazione, uno per ciascuna tipologia sopra esposta in quanto sono quelle presenti in tutti i PLC forniti dalle principali case produttrici. Tuttora ci sono molte resistenze da parte dei produttori di PLC ad integrare questi linguaggi nei propri sistemi per ostacolare gli acquirenti che decidono di cambiare fornitore, obbligandoli in questo modo a stravolgere il software.

0.2.3 PLC B&R

I PLC B&R sono quelli che ho effettivamente usato durante il mio lavoro di tesi e sono senza dubbio tra i più avanzati che si possano trovare sul mercato; essi supportano tutti i linguaggi della normativa IEC 1131-3 e sono gli unici che permettono la programmazione dei PLC in C e C++ anche se con alcune limitazioni, soprattutto nel caso di C++. I limiti nell'uso del linguaggio C sono strettamente legati a quelli intrinseci del linguaggio IEC Structured Text. Questo significa che, nonostante sia possibile programmare il PLC con C, è comunque sempre necessaria anche la conoscenza dei linguaggi specifici dei controllori logici programmabili.

È evidente come il mondo dei PLC possieda due anime in continuo contrasto; da un lato il progresso tecnologico che ha portato a una radicale evoluzione dei PLC, e dall'altro il mondo industriale che, cercando il più possibile di contenere i costi per poter rimanere competitivo nel mercato, solo in piccola parte è riuscito ad apprezzarne le attuali potenzialità.

0.3 cROS

0.3.1 Descrizione sintetica delle caratteristiche del software e del suo principio di funzionamento

cROS è una libreria scritta in Ansi C che fornisce un unico thread per l'implmentazione delle caratteristiche di base necessarie a realizzare un nodo ROS.

Lo scopo del software è di rendere compatibile l'ambiente ROS con i sistemi PLC. Attualmente l'unico produttore che renda possibile il porting è B&R in quanto è il solo a supportare il linguaggio C oltre a quelli standardizzati dalla normativa IEC 1131-3.

La libreria è stata sviluppata da IT+Robotics sul sistema operativo Linux (Ubuntu) dove sono stati condotti dei test per verificarne le sue funzionalità.

È importante notare come la libreria cROS non renda possibile l'utilizzo del protocollo middleware ROS sui PLC senza la compresenza di un PC linux che abbia il core ROS installato.

Il modus operandi del sistema consiste nell'esecuzione sul PLC della routine di gestione dei nodi il quale, mediante la comunicazione con il PC e quindi con il core ROS, in modo da fornire la possibilità all'utente di usare effettivamente ROS sul PLC.

La routine di gestione dei nodi è implementata dalla funzione void cRosNodeDoEventsLoop (CrosNode *n) della libreria, e il protocollo TCP/IP permette la comunicazione con il PC. La funzione cRosNodeDoEventsLoop viene eseguita ciclicamente e in essa avviene lo scambio di messaggi e di servizi tipica del protocollo ROS.

0.3.2 Lavoro svolto presso IT+Robotics

Il mio lavoro di tesi finalizzato al porting di cROS è iniziato con un'analisi approfondita delle discrepanze e dei limiti imposti dall'ambiente di sviluppo B&R Automation Studio per la creazione di una libreria scritta in C.

Ho constatato fin da subito delle discrepanze generate dalla differente piattaforma su cui si appoggia il software: cROS è stato sviluppato su un PC Linux mentre ora necessita di essere trasposto sul PLC.

I limiti invece, nascono da un'assenza di corrispondenza, almeno per al-

cuni elementi del linguaggio C nel testo strutturato (ST-IEC).

Una prima difformità consiste nelle librerie per il TCP/IP: quelle standard usate in ambiente linux non sono presenti sul PLC, ed è stato quindi mio compito quello di capire e usare le librerie fornite da B&R.

Un altro importante problema di base che ho dovuto affrontare ha riguardato l'impossibilità di usare i puntatori a funzione, poiché tutta la logica di cROS si fondava su questi per sfruttare una funzione definita dall'utente; per questo ho dovuto necessariamente invertire il meccanismo, argomento che però approfondirò in seguito.

Infine, un ultimo aspetto a cui mi sono dedicato, non meno importante dei precedenti, ha riguardato la condivisione dei tipi definiti nella libreria cROS. L'ambiente Automation Studio impone che tutti i tipi di una libreria che devono interagire direttamente con l'utente (es. ogni tipo incluso in una definizione di funzione destinata all'utente) devono essere definiti all'interno di file specifici (.typ) scritti in testo strutturato.

L'attività da me svolta si è rivelata necessaria per portare a compimento il porting, anche se richiederebbe ancora degli aggiustamenti per essere completamente funzionante.

0.3.3 I principali vantaggi con un'analisi dei possibili riscontri pratici

Perché rendere compatibile ROS con i PLC?

Le potenzialità del porting sono molteplici e i riscontri pratici ad esse connesse risultano di notevole impatto.

La prima è quella di fornire la possibilità agli utenti pratici dell'uso di ROS, anche se inesperti nell'uso dei PLC, di poterli programmare senza l'acquisizione di nuove competenze. Difatti, grazie all'astrazione dall'hardware fornita da ROS, la programmazione del PLC diventa più semplice e si avvicina molto di più a quella necessaria per un comune PC.

In secondo luogo, i linguaggi C++ e Pyton si rendono compatibili al 100% dal momento che è possibile creare dei moduli ROS sul computer in comunicazione col PLC; il C++ è supportato attualmente solo in minima parte dai PLC B&R e il Pyton non viene neppure preso in considerazione. Inoltre, ROS è già compatibile con molte piattaforme diverse nel settore dell'automazione industriale.

La conseguente standardizzazione e la notevole riduzione del distacco che separa la programmazione di un PLC da quella classica su PC, favorirà senza dubbio la scelta di aprire nuovi corsi per la programmazione dei PLC in istituti superiori ed universitari; in questo modo aumenterebbe l'offerta nel mercato del lavoro di persone in grado di programmare al meglio i PLC a beneficio delle aziende. Difatti, ancor oggi molte ditte riescono a sfruttare solo in piccola parte le risorse offerte dai PLC per mancanza di personale specializzato.

Un altro aspetto, che a mio avviso si può considerare rivoluzionario, concerne la possibilità di cooperazione nell'ambito dell'automazione industriale. Come ho già in precedenza sottolineato, ROS permette il rilascio di pacchetti usando licenze open source: questo rende possibile fronteggiare quei problemi che richiedono l'impegno di più professionalità.

In un ipotetico progetto entra inevitabilmente in gioco una varietà di competenze che solo un ambiente open source è in grado di mettere assieme: ROS è stato concepito proprio con l'intenzione di unire più competenze, al fine di incoraggiare sempre più la collaborazione nello sviluppo di software per l'automatizzazione.

Porting di cROS in ambiente Automation Studio

1.1 Descrizione del bug riscontrato e della relativa risoluzione

La prima attività da me svolta è stata quella di risolvere un bug che ho riscontrato durante una prima fase di test per prendere confidenza con cROS sul sistema operativo Ubuntu. L'errore non si manifestava in compilazione, bensì durante l'esecuzione, compromettendone il corretto funzionamento. Il programma segnalava un problema al md5sum, un codice di riconoscimento per i file usato all'interno di cROS; dopo una lunga fase di debugging, ho scoperto che l'errore era generato da un bug nella lettura di variabili array all'interno di un messaggio ROS, i quali non venivano riconosciuti come tali e perciò causavano un errore di scrittura nella fase di bufferizzazione.

Per la correzione è stato sufficiente aggiungere una condizione (if) che controllasse se la variabile fosse un array o meno, e si comportasse di conseguenza.

1.1.1 Descrizione della correzione apportata per risolvere il bug

Per porre rimedio a quest'errore, ho inserito nella funzione getMD5Txt (file: cros_message.c), come si può vedere in Figura 1.1, un if per il controllo della variabile in esame al fine di verificare se si tratta di un array, e nel caso positivo, il programma aggiunge al buffer la dimensione di quest'ultimo tra parentesi quadre.

```
commit 28a083281e501c74c7ecd1253d531b476345eaf6
Author: Aldo Marchi <aldo.marchi@studenti.unipd.it>
Date:   Tue Sep 29 17:48:03 2015 +0200

    Fix missing fields_it->is_array check in getMD5Txt()

diff --git a/src/cros_message.c b/src/cros_message.c
index b948e2e..f0b84d0 100644
--- a/src/cros_message.c
+++ b/src/cros_message.c
@@ -472,9 +472,25 @@ void getMD5Txt(cRosMessageDef* msg, DynString* buffer)
     while(fields_it->next != NULL)
     {
         const char *type_decl = getMessageTypeDeclarationField(fields_it);
+
+        // CHECK-ME: Verify processing "fields_it->is_array" for header
+        // and custom types
         if(isBuiltinMessageType(fields_it->type))
         {
             dynStringPushBackStr(buffer, type_decl);
+
+            if(fields_it->is_array)
+            {
+              dynStringPushBackStr(buffer,"[");
+              if(fields_it->array_size != -1)
+              {
+                char num[15];
+                sprintf(num, "%d",fields_it->array_size);
+                dynStringPushBackStr(buffer,num);
+              }
+              dynStringPushBackStr(buffer,"]");
+            }
+
             dynStringPushBackStr(buffer," ");
             dynStringPushBackStr(buffer,fields_it->name);
             dynStringPushBackStr(buffer,"\n");
```

Figura 1.1: Commit con Github della correzione del bug.

1.2 Cambio di librerie per il TCP/IP

Dopo aver corretto il bug, sono passato ad operare in ambiente Automation Studio e già dai primi tentativi di inserimento della libreria in questo ambiente, ho notato che buona parte dell'incompabilità era causata dal TCP/IP, ragion per cui ho deciso di dedicarmi come prima cosa a quest'ultimo.

Il TCP/IP è la suite di protocolli usata da cROS per permettere la comunicazione tra il PLC e un PC Linux per accedere al core ROS installato su quest'ultimo.

cROS è stato sviluppato in ambiente Linux e per questo motivo sono state usate le librerie standard Libnet, ma per farlo funzionare su un PLC ho dovuto cambiarle e usare le librerie fornite da B&R. Purtroppo, l'unico mezzo per capire come usare le librerie B&R era l'help di Automation Studio che ho scoperto essere incompleto; prima di passare ad operare su cROS, ho cercato di realizzare un semplice TCP/IP ponendo come server il PLC e come client il PC windows che operava sotto il simulatore di Automation Studio, ma solo dopo diversi tentativi sono riuscito ad instaurare la comunicazione tra i due. Ho scritto il primo esempio funzionante del TCP/IP, che sono riuscito

ad ottenere, in Structured Text, poiché risultava più semplice rispetto alla scrittura direttamente in C: l'help di Automation Studio era più dettagliato per questo linguaggio dal momento che la libreria B&R è stata scritta col medesimo.

In generale, tutte le librerie principali di Automation Studio sono scritte in ST fornendo l'opportunità agli utenti di usarle anche su file C mediante l'ausilio di file di header generati automaticamente dall'ambiente per permetterne la compatibilità; una volta che si è arrivati a capirne la logica è abbastanza facile "tradurre" un file semplice scritto con ST in uno scritto con C.

Il passo successivo è stato quello di riscrivere la socket di cROS e di modificarne la parte relativa di codice nei vari file (vedi paragrafo 1.2.2). Diversamente da quanto ci si possa aspettare, la socket è in realtà diventata più snella rispetto alla versione precedente; la motivazione di fondo risiede nell'essenza dei PLC: questi sono dispositivi strettamente real-time e di conseguenza non possono accettare alcun processo bloccante, come solitamente il protocollo TCP/IP. Nella socket di cROS per Linux erano presenti diverse righe di codice per evitare che il TCP interrompesse momentaneamente l'esecuzione in attesa di un'informazione in entrata: la struttura di ROS necessita di un'esecuzione ciclica priva di fermate e, per questo motivo, se non c'è alcun messaggio in ingresso, è sufficiente attendere il ciclo successivo per ricontrollare l'arrivo dell'informazione evitandone l'attesa. In questa nuova socket da me progettata non si ha quindi bisogno di scrivere funzioni per evitare che il TCP sia bloccante, perché non lo è più di default.

1.2.1 Analisi del server TCP B&R e della relativa conversione da ST a C

Analizzerò ora il server TCP/IP che ho scritto con la libreria AsTCP di B&R evidenziando alcuni aspetti interessanti che aiutano a comprendere meglio come venga attuata la compatibilità tra una libreria scritta in ST e un file scritto in C.

Come si può notare in Figura 1.2, è possibile mixare dichiarazioni di variabili in Ansi C e in ST (di quest'ultimo le definizioni dei tipi si trovano nel file di header plctypes.h);

```
#include <bur/plctypes.h>
#include <AsTCP.h>
#include <string.h>

#ifdef _DEFAULT_INCLUDES
#include <AsDefault.h>
#endif

UINT counter;
BOOL ioctl_ready;
BOOL serv_ready;
BOOL recv_ready;
BOOL send_ready;
BOOL close_ready;
char buffer[100];
SINT ipAddr[10] = "127.0.0.1";

TcpOpen_typ TcpOpen_1;
TcpServer_typ TcpServer_1;
TcpSend_typ TcpSend_1;
TcpClose_typ TcpClose_1;
TcpRecv_typ TcpRecv_1;

```

Figura 1.2: Dichiarazioni delle variabili per il server TCP/IP B&R.

il secondo blocco di dichiarazioni ha le relative definizioni nel file AsTCP.h: si tratterebbe del file, accennato sopra, generato automaticamente da Automation Studio per tutte le librerie. In questo caso il file servirebbe per rendere compatibili i data block nel linguaggio C, per meglio dire dei blocchi dati (DB): ognuno di essi accompagna una ben precisa funzione permettendole di non perdere i dati memorizzati nei blocchi stessi anche dopo il suo completamento, in particolare contengono tutte le variabili di ingresso e di uscita.

Nel linguaggio C, i DB vengono tradotti con delle struct che contengono tutti i dati che devono essere passati alla funzione associata.

```
void _INIT ProgramInit(void)
{
    TcpOpen_1.enable = 1;
    TcpOpen_1.pIfAddr = ipAddr;
    TcpOpen_1.port = 22011;

    strcpy(buffer, "");
    counter = 0;
    recv_ready = 1;
    serv_ready = 1;
    send_ready = 1;
    close_ready = 1;
    TcpOpen(&TcpOpen_1);


}

```

Figura 1.3: Blocco aciclico INIT.

Gli eseguibili per i PLC non possiedono un main unico, come accade solitamente nei PC, ma ne hanno tre: un INIT, un CYCLIC e un EXIT. Il primo viene eseguito una volta sola, quando il programma entra in funzione, passando poi al CYCLIC, il quale è ripetuto ciclicamente fino al termine; a questo punto viene eseguito il blocco EXIT. L'unico dei tre realmente necessario è il CYCLIC (vedi schema alla pagina seguente)

Nell'INIT (vedi Figura 1.3) inserisco i dati nel DB TcpOpen, cioè gli argomenti della funzione ad esso associata, e invoco il *FB una prima volta per aprire la comunicazione dopo aver inizializzato le variabili booleane e il counter (la loro utilità verrà analizzata in seguito).

Schema di un eseguibile in ambiente Automation Studio

```
#include <bur/plctypes.h>

#ifdef _DEFAULT_INCLUDES
#include <AsDefault.h>
#endif


    dichiarazione di eventuali variabili globali...


void _INIT ProgramInit(void)
{
  /* istruzioni eseguite
     all'avvio dell'eseguibile
     un'unica volta
     ... */
}


void _CYCLIC ProgramCyclic(void)
{
  /* istruzioni eseguite
     ciclicamente
     fino all'interruzione
     ... */
}


void _EXIT: ProgramExit(void)
{
  /* istruzioni eseguite
     prima della chiusura
     dell'eseguibile
     ... */
}
```

Nota: i tre blocchi possono essere posti in file differenti

```
void _CYCLIC ProgramCyclic(void)
{

    if(counter == 2)
    {
        if(TcpOpen_1.status != 0)
        {
            TcpOpen(&TcpOpen_1);
            codex = TcpOpen_1.status;
        }
        else if(serv_ready == 1)          /* if no co
        {
            TcpServer_1.enable = 1;
            TcpServer_1.ident = TcpOpen_1.ident;
            TcpServer_1.backlog = 0;
            TcpServer(&TcpServer_1);           /* tri
            serv_ready = 0;
            codex = 11;
        }
        else if(TcpServer_1.status != 0)
        {
            TcpServer(&TcpServer_1);
            codex = TcpServer_1.status;
        }
        else if(recv_ready == 1)            /* if the
        {
            TcpRecv_1.enable = 1;
            TcpRecv_1.ident = TcpServer_1.identclnt;
            TcpRecv_1.pData = &buffer[0];
            TcpRecv_1.datamax = sizeof(buffer);
            TcpRecv(&TcpRecv_1);            /* recive
            recv_ready = 0;           /* initialize */
            codex = 8;
        }
        else if(TcpRecv_1.status != 0)           /* if
            TcpRecv(&TcpRecv_1);           /* send th
        else if(send_ready == 1)            /* if the
        {
            TcpSend_1.enable = 1;
            TcpSend_1.ident = TcpServer_1.identclnt;
            TcpSend_1.pData = buffer;
            TcpSend_1.datalen = sizeof(buffer);
            TcpSend(&TcpSend_1);           /* send th
            send_ready = 0;           /* initialize */
        }
        else if(TcpSend_1.status != 0)           /* if
            TcpSend(&TcpSend_1);           /* send th
        else if(close_ready == 1)           /* set th
        {
            TcpClose_1.enable = 1;
            TcpClose_1.ident = TcpServer_1.identclnt;
            TcpClose(&TcpClose_1);          /* set the
            close_ready = 0;          /* initialize */
        }
        else if(TcpClose_1.status != 0)      /* if the
            TcpClose(&TcpClose_1);          /* set the
            counter = 0; /* initialize */

    }
    counter++;                    /* increment */

}
```

Figura 1.4: Blocco ciclico CYCLIC.

Nel CYCLIC vengono eseguite tutte le FB necessarie ad aprire/chiudere e a scambiare dati per mezzo del protocollo TCP/IP. Il modo di operare delle varie funzioni è simmetrico: dall'inizializzazione del DB, analogamente al TcpOpen, all'esecuzione poi della funzione. Ogni ciclo del CYCLIC dura 1 ms e lo scopo del counter è quello di ridurre la frequenza a 3 ms: le funzioni possono essere eseguite se e solo se il contatore vale 2. In ogni ciclo utile viene eseguito un solo FB e la relativa funzione viene riinvocata nei cicli utili successivi, fino a quando l'esito non risulti valido ($status \neq 0$, vedi Figura 1.4). L'inizializzazione del DB di ogni FB è eseguita un'unica volta, cioè quando la relativa funzione deve essere invocata per la prima volta; questa unicità è garantita dalle variabili booleane $*\_ready$. Ognuno dei FB, escluso TcpOpen, per funzionare correttamente necessita che l'esecuzione della funzione antecedente sia andata a buon fine.

Il relativo blocco di EXIT è vuoto.

*NOTA: le funzioni che possiedono un DB associato si dicono function block. (FB)

1.2.2 Descrizione della socket

Per completare la compatibilità del TCP/IP con l'ambiente Automation Studio, ho riscritto il socket, e per fare ciò mi sono posto due vincoli: in primo luogo, volendo rimanere molto fedele alla forma della socket precedente, ho cercato di creare funzioni simili in input/output per riuscire a non stravolgere

troppo il resto del codice, e in secondo luogo ho cercato di essere quanto più C-user-friendly nascondendo la maggior parte dei dettagli del testo strutturato. Per adempiere al mio intento, ho creato una struct (TcpipSocket, vedi Appendice A per dettagli) che contenesse le variabili booleane di ready e i vari DB necessari; è da notare che anche nella socket originaria era presente una struct che raggruppava tutto il necessario per la comunicazione con il TCP. Le funzioni sono rimaste essenzialmente le stesse con l'eccezione dell'eliminazione di quelle necessarie all'evitare il blocking (per i motivi riportati sopra), e con l'aggiunta di altre con un'utilità esclusivamente interna al file.

1.3 Inversione del meccanismo di callback

Il problema più grande che ho dovuto affrontare durante il mio lavoro di tesi, si è presentato quando ho scoperto che una libreria scritta in Ansi C non ammetteva l'uso di puntatori a funzione, se questi dovevano essere usati dall'utente.

L'ambiente Automation Studio implementa un sistema che permette all'utente di usare una libreria indipendentemente dal linguaggio che sceglierà di usare tra quelli supportati, cioè non ci sono problemi di compatibilità se l'eseguibile scritto dall'utente è in un linguaggio differente da quello della libreria. Questo bellissimo sistema favorisce molto l'utilizzatore, ma complica l'attività di sviluppo di una libreria nell'ambiente. Il linguaggio prediletto da Automation Studio è il testo strutturato e difatti qualsiasi libreria deve avere dei file "speciali" scritti appositamente con quest'ultimo: in sostanza deve possedere un un file .typ contenente tutte le definizioni dei tipi della libreria che devono interagire con l'utente e un file .fun contenente tutte le dichiarazioni di funzioni che devono essere a disposizione dell'utente. Solitamente i tipi che devono interagire sono quelli di INPUT/OUTPUT delle funzioni a disposizione dell'utente definite nel .fun.

L'impedimento si incontra proprio nel file .typ poiché, dovendo esso scriversi in Structured Text, ne possiede le medesime limitazioni: una di queste riguarda il tipo puntatore a funzione che, guarda caso, non esiste in ST.

L'intera libreria si fondava sull'utilizzo di funzioni definite dall'utente e passate tramite puntatori; è stato perciò necessario trovare un modo alternativo per poter operare. La soluzione vincente tra quelle da me ipotizzate, è stata quella di invertire la logica: adesso cROS, invece di avere in input il puntatore della funzione, ha in output gli argomenti per la funzione dell'utente. In questo modo l'utente non deve più preoccuparsi di fornire a cROS la sua funzione, ma deve occuparsi di assegnare alla propria funzione, prima

della fine del ciclo, tutti gli input che cROS gli fornisce in output al termine della propria routine.

```
/*! Max num published topics */
#define CN_MAX_PUBLISHED_TOPICS 5

#define NEVER_STOP 1

static CallbackResponse user_callback(DynBuffer *buffer, cRosMessage* message);

[...]

while ( NEVER_STOP )
{
    cRosNodeDoEventsLoop ( node );

    for ( i = 0; i < CN_MAX_PUBLISHED_TOPICS; ++i )
        while ( node->pubs[i].queue_pubs != NULL )
            user_callback ( pop(&(node->pubs[i].queue_pubs)), node->pubs[i].msg );
}
```

Figura 1.5: Schema del funzionamento di cROS nel file utente.

Al fine di attuare il cambiamento di logica, ho scelto di ritornare a sviluppare in ambiente Linux perché qui è più semplice eseguire test.

1.3.1 Descrizione tecnica dell'implementazione

Il primo passo che ho compiuto per l'inversione del meccanismo di callback è stato quello di pensare e poi implementare una struttura da fornire all'utente per la condivisione degli input delle ex callback (packet e status a seconda). In particolare un packet è un buffer dinamico, cioè il messaggio standard usato da ROS per scambiare dati tra i nodi (vedi Figura 1.6),

```
typedef struct DynBuffer DynBuffer;
struct DynBuffer
{
  size_t size;                  //! Current buffer size
  size_t pos_offset;            //! Current position indicator
  size_t max;                   //! Max buffer size
  unsigned char *data;         //! buffer data
};
```

Figura 1.6: Struct DynBuffer, file: dyn_buffer.h.

mentre uno status è di tipo CrosNodeStatusUsr, e serve per fornire dei parametri relativi all'XMLRPC (vedi Figura 1.7).

```
typedef struct CrosNodeStatusUsr
{
  // FIXME: this is a work in progress
  // int callid; // This may be useful to track register/unregister
  CrosNodeStatus state; // May be useful to udnerstand what the node, or the particular sub/pub/svc is doing
  int provider_idx;
  const char *xmlrpc_host;
  int xmlrpc_port;
  const char *parameter_key;
  XmlrpcParam *parameter_value;
} CrosNodeStatusUsr;
```

Figura 1.7: Struct CrosNodeStatus.

Gli input variano nel numero ad ogni routine di cROS, e devono essere eseguiti nella relativa funzione un'unica volta; per queste ragioni ho deciso di usare una coda, nonostante si sarebbe potuta usare una pila, poiché non vi è la necessità di processarli in un ordine preciso.

Non esistendo una libreria standard dedicata alle code nel linguaggio C, ho dovuto implementarle: i nodi della coda sono formati dall'input e da un puntatore all'elemento successivo come si può vedere nelle Figure 1.8 e 1.9.

```
typedef struct QueueArgExtFun
{
  DynBuffer* packet;
  struct QueueArgExtFun* next;
}QueueArgExtFun;
```

Figura 1.8: Nodo della coda dei packet.

```
typedef struct QueueStatus
{
  CrosNodeStatusUsr status;
  struct QueueStatus* next;
}QueueStatus;
```

Figura 1.9: Nodo della coda degli status.

Ho realizzato le funzioni di push, pop e release per entrambe le tipologie di coda: il push serve per aggiungere un nuovo nodo alla coda, il pop serve per prenderne uno e eliminarlo dalla coda, e infine il release serve per svuotare la coda.

Successivamente ho eliminato solo i tipi puntatore a funzione delle callback che dovevano essere rimosse. È bene sottolineare che non tutti i puntatori a funzione della libreria cROS sono stati eliminati: molti di loro non vanno ad interagire con l'utente, ragion per cui la loro cancellazione sarebbe stata un inutile complicazione.

I function pointers che ho eliminato sono:

- `typedef CallbackResponse (*PublisherCallback)( DynBuffer *buffer, void* context);`
- `typedef CallbackResponse (*SubscriberCallback)( DynBuffer *buffer, void* context);`
- `typedef CallbackResponse (*ServiceProviderCallback)( DynBuffer *bufferRequest, DynBuffer *bufferResponse, void* context);`

Essi venivano memorizzati all'interno della variabile di tipo CrosNode (vedi Appendice B, è il principale della libreria), la cui funzione è quella di tenere traccia dell'intero sistema di gestione dei nodi implementato da cROS. I tre puntatori sopra elencati erano membri rispettivamente delle struct PublisherNode, SubscriberNode e ServiceProviderNode. Queste struct sono definite come array all'interno di CrosNode perché servono per contenere le quattro principali categorie di nodi: publisher, subscriber, client e server. I primi due si scambiano dati attraverso i messaggi (il publisher invia mentre il subscriber riceve); gli ultimi due, invece, si scambiano servizi (il client richiede e il server esegue). È facile intuire come nel primo caso si tratti di una comunicazione unidirezionale, mentre nel secondo è bidirezionale; infatti un servizio è composto da due parti: una request (del client) ed una reply (del server).

Per continuità ho deciso di inserire le code per la memorizzazione degli input proprio nello stesso posto dove ho tolto le callback (vedi Figure 1.10, 1.11 e 1.12)

```
/*! Structure that define a published topic */
struct PublisherNode
{
  char *topic_name;
  char *topic_type;
  char *md5sum;
  char *message_definition;
  int client_tcpros_id;
  /* @AldoM adattamento automation studio
  void *context;
  PublisherCallback callback;
  NodeStatusCallback status_callback;
  */

  // @AldoM posti in sostituzione alle callback
  QueueArgExtFun* queue_pubs;
  QueueStatus* queue_status;
  cRosMessage* msg;

  int loop_period;    //! Period (in msec) for
};
```

Figura 1.10: Struct PublisherNode, file: cros_node.h.

```
struct SubscriberNode
{
  char *message_definition;
  char *topic_name;
  char *topic_type;
  char *md5sum;
  char *topic_host;
  int   topic_port;
  int   client_xmlrpc_id;
  int   client_tcpros_id;
  int   tcpros_port;
  /* @AldoM adattamento automa
  void *context;
  SubscriberCallback callback;
  NodeStatusCallback status_ca
  */

  // @AldoM posti in sostituzi
  QueueArgExtFun* queue_subs;
  QueueStatus* queue_status;
  cRosMessage* msg;
};
```

Figura 1.11: Struct SubsciberNode, file: cros_node.h.

```
struct ServiceProviderNode
{
  char *service_name;
  char *service_type;
  char *servicerequest_type;
  char *serviceresponse_type;
  char *md5sum;
  /* @AldoM adattamento automation
  void *context;
  ServiceProviderCallback callback;
  NodeStatusCallback status_callba
  */

  // @AldoM posti in sostituzione a
  QueueArgExtFun* queue_services;
  QueueStatus* queue_status;
};
```

Figura 1.12: Struct ServiceProviderNode, file: cros_node.h.

D'ora in poi mi riferirò solo ai publisher poiché le modifiche attuate per questi sono analoghe agli altri casi elencati sopra.

Nella funzione per la registrazione di un nuovo Publisher ho rimosso le callback dalla dichiarazione e le relative operazioni di memorizzazione al suo interno proprio perché l'utente non deve più fornire a cROS alcuna callback.

```
int cRosApiRegisterPublisher(CrosNode *node, const char *topic_name, const char *topic_type, int loop_period)
                                //@AldoM
                               //PublisherApiCallback callback, NodeStatusCallback status_callback, void *context)
```

Figura 1.13: Function cRosApiRegisterPublisher, file: cros_api.h.

Lo stesso è stato fatto per le funzioni di inizializzazione (initPublisherNode, initSubscriberNode, initServiceProviderNode, etc.. : file cros_node.c).

Sono passato poi ad introdurre il codice necessario per salvare i packet per l'utente all'interno della coda definita sopra; ho dovuto attuare le modifiche in più di una funzione, ma il principio applicato è il medesimo.

Ho cercato tutte le funzioni in cui veniva invocata la callback dove entravano in gioco i pacchetti di interesse: ho rimosso tramite commento le parti in cui veniva usata la callback e, al suo posto, ho attuato un push del packet nella coda.

```
414  void cRosMessageParsePublicationPacket( CrosNode *n, int client_idx )
415  {
416    TcprosProcess *client_proc = &(n->tcpros_client_proc[client_idx]);
417    DynBuffer *packet = &(client_proc->packet);
418    int sub_idx = client_proc->topic_idx;
419    /* @AldoM adattamento automation studio
420    void* data_context = n->subs[sub_idx].context;
421    n->subs[sub_idx].callback(packet,data_context);
422    */
423    push(&(n->subs[sub_idx].queue_subs), packet);
424  }
```

Figura 1.14: Esempio di correzione.

Una modifica analoga è stata fatta effettuata anche in altre funzioni come, per esempio, cRosMessagePreparePublicationPacket (file cros_tcpros.c) o cRosApiParseResponse (file cros_node_api.c).

Problemi nel funzionamento e risoluzioni

Nonostante le correzioni apportate, il software non poteva funzionare: durante i primi test, mi sono reso conto che le callback richiedevano, per un corretto funzionamento, un messaggio di tipo cRosMessage (vedi Appendice C) in input. Inizialmente ho tentato di creare direttamente nel file di test un messaggio che potesse soddisfare le esigenze, ma non sono arrivato ad alcun risultato. Alla fine, dopo vari tentativi, sono arrivato a trovare la soluzione per questo annoso problema: ho creato un puntatore al messaggio all'interno della struct PublisherNode; questa struct è definita all'interno di CrosNode come un array e ogni suo elemento è un publisher differente. Il messaggio viene definito durante la registrazione stessa di un nuovo publisher, perciò è stato semplice salvare un puntatore per ognuno di essi nell'array. Adesso, ogni qual volta si attua la registrazione di un nuovo publisher, il programma salva un puntatore al messaggio rendendolo disponibile all'utilizzatore (es: il messaggio del primo publisher registrato sarà accessibile così → *(node.pubs[0].msg) con node di tipo CrosNode).

Il messaggio è creato nella dichiarazione del ProviderContext (l'elemento specifico è outgoing) all'interno della funzione cRosApiRegisterPublisher (vedi Figura 1.15)

```
284  int cRosApiRegisterPublisher(CrosNode *node, const char *topic_name, const char *topic_type, int loop_period)
285                                    //@AldoM
286                                   //PublisherApiCallback callback, NodeStatusCallback status_callback, void *context)
287  {
288    char path[256];
289    cRosGetMsgFilePath(node, path, 256, topic_type);
290    ProviderContext *nodeContext = newProviderContext(path, CROS_PUBLISHER);
291    if (nodeContext == NULL)
292      return -1;
293    /* @AldoM
294    nodeContext->api_callback = callback;
295    nodeContext->status_callback = status_callback;
296    nodeContext->context = context;
297    */
298
299    // NB: Pass the private ProviderContext to the private api, not the user context
300    int rc = cRosNodeRegisterPublisher(node, nodeContext->message_definition, topic_name, topic_type,
301                                   nodeContext->md5sum, loop_period, nodeContext->outgoing);
302                                    /*@AldoM
303                                       cRosNodePublisherCallback,
304                                   status_callback == NULL ? NULL : cRosNodeStatusCallback, nodeContext);
305                                   */
306    return rc;
307  }
```

Figura 1.15: Function cRosApiRegisterPublisher, file: cros_api.c.

```
1631    PublisherNode *pub = &node->pubs[pubidx];
1632    pub->message_definition = pub_message_definition;
1633    pub->topic_name = pub_topic_name;
1634    pub->topic_type = pub_topic_type;
1635    pub->md5sum = pub_md5sum;
1636    // @AldoM
1637    pub->msg = msg;
1638
```

Figura 1.16: memorizzazione del messaggio msg.

L'outgoing è poi passato alla funzione cRosNodeRegisterPublisher dove viene effettivamente salvato nell'array (Figura 1.16).

Le esecuzioni successive del file di test non davano comunque un esito positivo perché si presentava continuamente un errore di bufferizzazione. Dopo una fase di debugging, sono riuscito ad isolare il punto in cui si verificava l'errore; la risoluzione è stata quindi quella di usare una serializzazione proprio in quel punto, come si può vedere in Figura 1.17.

```
/* @AldoM adattamento automation studio
void* data_context = n->pubs[pub_idx].context;
n->pubs[pub_idx].callback( packet, data_context);
*/
push(&(n->pubs[pub_idx].queue_pubs), packet);

// @AldoM essenziale per far funzionare il talker
cRosMessageSerialize(n->pubs[pub_idx].msg, packet);
```

Figura 1.17: cRosMessagePreparePublicationHeader file: cros_tcpros.c.

L'errore in questione nasceva a causa della funzione che veniva invocata subito dopo, la quale richiedeva la dimensione del buffer che non era mai stata inizializzata, poiché non era stata effettuata la sua serializzazione, e di conseguenza il valore letto era insensato.

1.3.2 Descrizione dei file di test

I file di test creati per verificare il corretto funzionamento di cROS dopo l'inversione del meccanismo delle callback sono un talker cROS e un listener ROS.

Il listener è scritto in C++ (vedi Figura 1.18) con le librerie standard di ROS ed è un subsriber la cui funzione è quella di ricevere una stringa dal talker (il publisher). Il talker e il listener si identificano tra i vari nodi utilizzando un identico nome di tipo stringa: in questo caso "/chatter".

```
#include "ros/ros.h"
#include "prova/hello.h"

void chatterCallback(const prova::hello::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->aWord.c_str());
}

int main(int argc, char **argv)
{

  ros::init(argc, argv, "listener");

  ros::NodeHandle n;

  ros::Subscriber sub = n.subscribe("/chatter", 1000, chatterCallback);

  ros::spin();

  return 0;
}
```

Figura 1.18: Listener ROS scritto in C++.

La funzione spin serve per rendere ciclico il funzionamento fino a che il nodo non viene chiuso.

Passando poi al talker scritto in C con le librerie cROS, è possibile vedere in modo più concreto il funzionamento di un ipotetico file scritto da un utente dopo il rovesciamento del meccanismo delle callback.

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

    if (!inizializza(argc, argv, "talker_cros"))
        return EXIT_FAILURE;

    char *new_node_name = "/talker_cros",
        *node_name = new_node_name;
    char *default_host = "127.0.0.1",
        *node_host = default_host,
        *roscore_host = default_host;
    unsigned short roscore_port = 11311;

    char path[1024];
    getcwd(path, sizeof(path));
    strncat(path, "/rosdb", sizeof(path));
    node = cRosNodeCreate(node_name, node_host,
    roscore_host, roscore_port, path, NULL);

    int rc = 0;
```

Figura 1.19: Talker cROS: dichiarazioni.

La funzione "inizializza" controlla semplicemente che argc sia uguale ad 1 e argv sia uguale alla stringa posta in input, ma la sua rilevanza nel talker è trascurabile. Successivamente vengono definite le variabili di input necessarie per inizializzare la variabile globale di gestione dei nodi (node) di tipo CrosNode*. rc è una variabile di controllo usata subito dopo per controllare che la registrazione del publisher sia andata a buon fine.

Si passa poi alla registrazione del publisher; è bene notare che, grazie all'inversione del meccanismo di callback non si ha più la necessità di passare il puntatore di una callback alla funzione.

```
rc = cRosApiRegisterPublisher(node, "/chatter", "prova/hello", 1000);

if (rc == -1)
    return EXIT_FAILURE;
```

Figura 1.20: Talker cROS: function cRosApiRegisterPublisher.

Il cambiamento principale si ha nella routine di cROS: essa non è più automatica, ovvero, prima che il ciclo ricominci, è necessario che l'utente esegua i packet nella funzione di callback.

```
    unsigned char exit = 1;
    int i;
    while(exit)
    {
      cRosNodeDoEventsLoop( node );

for (i = 1; i < 4; ++i)
        while (node->pubs[i].queue_pubs != NULL)
          callback_hello(pop(&(node->pubs[i].queue_pubs)), node->pubs[i].msg);
    }
    cRosNodeDestroy( node );

    return EXIT_SUCCESS;

}// main
```

Figura 1.21: Talker cROS: routine.

Con riferimento alla Figura 1.22, la callback serve per inviare il messaggio "Hello world!!!" al subscriber. Per l'invio di questo messaggio è necessaria la preparazione di una variabile di tipo cRosMessageField (vedi Appendice C) tramite le funzioni cRosMessageGetField e cRosMessageSetFieldValueString. La prima serve per inizializzare la variabile, mentre la seconda per memorizzare la stringa che si desidera trasmettere. La fprintf mi è servita per il debugging, ma non è utile ai fini dell'invio del messaggio. La funzione che permette effettivamente la comunicazione tra il publisher ed il subsriber è cRosMessageSerialize.

```
static CallbackResponse callback_hello(DynBuffer *buffer, cRosMessage* message)
{
    cRosMessageField *StringStatus_field = cRosMessageGetField(message, "aWord");

    cRosMessageSetFieldValueString(StringStatus_field, "Hello world!!!");

    fprintf(stderr, "\n\nHOLA %s\n\n", StringStatus_field->data.as_string);

    /**/
    StringStatus_field->size=20;

    cRosMessageSerialize(message, buffer);

    return 0;
}
```

Figura 1.22: Talker cROS: callback.

Il test, nonostante la sua semplicità, pone in evidenza la capacità di cROS di mettere in comunicazione un suo nodo con uno ROS. Grazie ad esso, si comprende in modo concreto la possibilità di sfruttare anche il linguaggio C++ per un nodo di un ipotetico sistema; infatti, è possibile porre in comunicazione un qualsiasi nodo standard ROS con cROS e quindi con un PLC.

1.4 Redefinizione in testo strutturato dei tipi di cROS da condividere con l'utente

L'ultimo aspetto a cui mi sono dedicato, non meno importante dei precedenti, ha riguardato la condivisione dei tipi definiti nella libreria cROS. L'ambiente Automation Studio impone che tutti i tipi di una libreria che devono interagire direttamente con l'utente devono essere definiti all'interno di file specifici (.typ) scritti in testo strutturato.

Come esposto nel paragrafo 1.3, il testo strutturato presenta meno tipi fondamentali rispetto al linguaggio C e nel caso specifico è stato risolto il problema dell'assenza dei puntatori a funzione attraverso l'inversione del meccanismo di callback. Grazie all'inversione, mi sono potuto dedicare alla stesura dei file .typ dal momento che l'impedimento principale era stato rimosso; purtroppo però, l'assenza dei puntatori non era l'unico vincolo.

1.4.1 Adattamento dei tipi fondamentali del linguaggio ST a quelli del linguaggio C

Il linguaggio Structured Text dispone dei seguenti tipi:

```
SINT    [1 B]  →  char
USINT          →  unsigned char
INT     [2 B]  →  short int
UINT           →  unsigned short int
DINT    [4 B]  →  long int
UDINT          →  unsigned long int
REAL    [4 B]  →  float
LREAL   [8 B]  →  double
BOOL    [1 B]  →  char
STRING[N]      →  char[n]
```

Oltre a questi ce ne sarebbero degli altri, ma la loro funzione è prettamente legata ai PLC e non hanno un corrispondente in C.

Inoltre si possono definire delle strutture, delle enumerazioni, degli array, dei puntatori e delle redefinizioni di nome, nei seguenti modi:

```
struct_name :  STRUCT
element1 :  type;
element2 :  type;
...
END_STRUCT;

enum_name :
(
element1,
element2,
...
);

ARRAY[0..n] OF type; // nota:  arry di n+1 elementi

REFERENCE TO type; // nota:  puntatore

new_typeName :  type;
```

È bene notare che non è possibile definire in ST un array di puntatori, anche se nel mio caso non ce n'erano.

I puntatori multipli non sono ammessi con l'eccezione del doppio puntatore a char (char**) che può essere rappresentato come un ARRAY[0..n] OF STRING[N].

```
TYPE
    _int8_t : SINT;
    _uint8_t : USINT;

    _int16_t : INT;
    _uint16_t : UINT;

    _int32_t : DINT;
    _uint32_t : UDINT;

    _int64_t : DINT;
    _uint64_t : UDINT;

    _size_t : UDINT;

    _int : DINT;
    unsignedInt : UDINT;
    _char : SINT;
    unsignedChar : USINT;
    _short : INT;
    unsignedShort : UINT;
    unsignedLong : UDINT;
    _double : LREAL;
```

Figura 1.23: ST types to C types.

In Figura 1.23 ho ridefinito i tipi di ST con i nomi usati dal C; tuttavia in alcuni casi ho dovuto usare il medesimo tipo di ST per quelli del C, dal momento che non ne esistono di corrispondenti. Prendendo ad esempio int64_t, la sua definizione non è formalmente corretta perché il DINT è costituito solo da 4 B, ma non è previsto un tipo intero in ST con un numero maggiore di bytes che possa andare bene. I cambi di nome sono stati effettuati per rendere di più facile lettura e gestione le definizioni delle struct, che sarei andato a compiere successivamente nei file .typ.

1.4.2 Creazione dei file .typ individuando i tipi da rendere disponibili con l'utente

Quando sono passato alla fase operativa, ho avuto la necessità di "tradurre" la struct principale, cioè CrosNode: questo compito potrebbe sembrare banale, ma la suddetta struttura ha nella sua definizione numerose strutture annidate che a loro volta ne hanno altre che ne hanno altre; Automation Studio non permette di averne una in un file .typ i cui elementi siano di tipi struct parzialmente definiti o definiti in file di diverso genere. Dopo averle tutte rintracciate, giungendo fino a quelle formate solo da tipi elementari, ho creato tanti file .typ quanti erano quelli usati dalla libreria cROS e li ho chiamati con nomi evocativi rispetto ai file d'origine; il lavoro è stato lungo, ma non ha presentato in genere particolari difficoltà. (in Appendice D si trova CrosNode scritto in ST)

Ho avuto un'unica vera difficoltà, di difficile risoluzione, quando ho do-

vuto inserire il tipo cRosMessage, struttura da me introdotta all'interno di CrosNode durante le modifiche per l'inversione del meccanismo di callback (vedi paragrafo 1.3.1, sezione Problemi nel funzionamento e risoluzioni). Il linguaggio C permette di definire una struct anche se al suo interno ci sono elementi di tipo puntatore a struct non realmente ancora definite, ma solo dichiarate attraverso un typedef (vedi Figura 1.25); tuttavia ciò non è possibile nel testo strutturato: per definire cRosMessageField sarebbe necessario definire prima cRosMessage (vedi Figura 1.24), ma per farlo, sarebbe necessario definire prima cRosMessageField.

Inoltre, in entrambe le definizioni sono presenti dei doppi puntatori, altro elemento irrappresentabile mediante il linguaggio ST. Avevo ipotizzato di salvare il puntatore al messaggio in una variabile di tipo UDINT (unsigned long int) all'interno della libreria cROS e di renderla disponibile all'utente; quest'ultimo poi la restituirebbe alla libreria alla fine della routine di cROS come di consueto (vedi paragrafo 1.3.1, Problemi e ...) e, infine, si dovrebbe realizzare un casting all'interno della libreria per riottenere il messaggio cRosMessage di partenza. Questo sarebbe un escamotage per evitare che l'utente abbia direttamente a che fare con il suddetto messaggio, evitando così la necessità di dover dichiarare la struct cRosMessage nei file .typ. L'utente, non avendo bisogno di usare effettivamente il messaggio, funge solo da ponte per questo che esce e rientra nella libreria.

```
struct cRosMessage
{
    cRosMessageField **fields;
    cRosMessageDef* msgDef;
    char* md5sum;
    int n_fields;
};
```

Figura 1.24: struct cRosMessage.

```
typedef struct cRosMessage cRosMessage;

struct cRosMessageField
{
    char *name;

    union data
    {
      uint8_t opaque[8];
      int8_t as_int8;
      uint8_t as_uint8;
      int16_t as_int16;
      uint16_t as_uint16;
      int32_t as_int32;
      uint32_t as_uint32;
      int64_t as_int64;
      uint64_t as_uint64;
      float as_float32;
      double as_float64;
      char *as_string;
      cRosMessage *as_msg;
      int8_t *as_int8_array;
      uint8_t *as_uint8_array;
      int16_t *as_int16_array;
      uint16_t *as_uint16_array;
      int32_t *as_int32_array;
      uint32_t *as_uint32_array;
      int64_t *as_int64_array;
      uint64_t *as_uint64_array;
      float *as_float32_array;
      double *as_float64_array;
      char **as_string_array;
      cRosMessage **as_msg_array;
      void *as_array;
    } data;
    int size;
    int is_const;
    int is_array;
    int is_fixed_array;
    int array_size;
    int array_capacity;
    CrosMessageType type;
    char *type_s;
};
```

Figura 1.25: struct cRosMessageField.

Conclusioni

Il lavoro di tesi si è concluso con un approccio all'hardware del PLC per capire come renderlo operativo da un computer Linux tramite VM Windows.

Ho portato a compimento le fasi più importanti per permettere il porting della libreria cROS in ambiente Automation Studio per PLC B&R.

Per completare il lavoro sarà necessario:

1. sviluppare il mio ultimo progetto al fine di risolvere il problema generato dalla struct cRosMessage (vedi paragrafo 1.4.2), e

2. risolvere alcuni bug di compilazione generati da alcune parole chiave del C usate nella libreria cROS (es. typedef, static) la cui causa è ignota poiché l'ambiente Automation Studio non dovrebbe porre limiti su queste parole; probabilmente questi errori sono dovuti alla versione non aggiornata dell'ambiente che avevo a disposizione (la penultima).

In ultima analisi la tesi è stata di grande utilità pratica e teorica per la mia formazione in quanto ho potuto mettermi alla prova in un ambiente lavorativo e spero che il mio lavoro potrà in qualche modo essere utile nel mondo industriale.

Ringraziamenti

In primis vorrei fare un ringraziamento speciale al professor Stefano Ghidoni, il mio relatore, per avermi seguito con pazienza e costanza in tutte le fasi della tesi.

Ringrazio infine la Spin-off dell'Università di Padova IT+Robotics e in particolar modo Nicolò Boscolo per la collaborazione e l'amicizia dimostratami durante il lavoro di tesi.

Appendice A

TcpipSocket.h

```
#ifndef TCPIPSOCKET_H
#define TCPIPSOCKET_H

#include <AsTcp.h>
#include <bur/plctypes.h>
#include <string.h>
#include <dyn_buffer.h>
#include <dyn_string.h>

typedef enum
{
TCPIPSOCKET_FAILED = 0,
TCPIPSOCKET_IN_PROGRESS,
TCPIPSOCKET_DISCONNECTED,
TCPIPSOCKET_DONE,
TCPIPSOCKET_UNKNOWN
} TcpipSocketState;

typedef struct TcpipSocket
{
BOOL open_ready;
BOOL serv_ready;
BOOL recv_ready;
BOOL send_ready;
BOOL close_ready;
size_t maxLength;
SINT ipAddr[10];
```

```
TcpOpen_typ TcpOpen_1;
TcpServer_typ TcpServer_1;
TcpSend_typ TcpSend_1;
TcpClose_typ TcpClose_1;
TcpRecv_typ TcpRecv_1;
} TcpipSocket;

void TcpipSocketInit(TcpipSocket* s, char ipAddress[10],
int port, size_t maxLen);

int TcpipSocketOpen(TcpipSocket* s);

TcpipSocketState TcpipSocketConnect(TcpipSocket* s);

TcpipSocketState TcpipSocketReadBuffer(TcpipSocket* s,
DynBuffer* d_buf);

TcpipSocketState TcpipSocketReadBufferEx(TcpipSocket* s,
DynBuffer* d_buf, size_t max_size, size_t *n_reads);

TcpipSocketState TcpipSocketReadString(TcpipSocket* s,
DynString* d_str);

TcpipSocketState TcpipSocketWriteBuffer (TcpipSocket* s,
DynBuffer *d_buf);

TcpipSocketState TcpipSocketWriteString (TcpipSocket* s,
DynString *d_str);

int TcpipSocketClose(TcpipSocket* s);

UINT tcpIpSocketGetPort(TcpipSocket *s);

int isTcpipSocketOpen (TcpipSocket *s)

int isTcpipSocketConnected (TcpipSocket *s);

void checkAddressAndChangeIfNecessary (TcpipSocket* s,
char ipAddress[10], int port);

#endif
```

Appendice B

CrosNode type

```
typedef struct CrosNode CrosNode;
struct CrosNode
{
  char *name;
  //! The node name: it is the absolute name,
  //i.e. it includes the namespace
  char *host;
  //! The node host (ipv4, e.g. 192.168.0.2)
  unsigned short xmlrpc_port;
   //! The node port for the XMLRPC protocol
  unsigned short tcpros_port;
  //! The node port for the TCPROS protocol
  unsigned short rpcros_port;
  //! The node port for the RPCROS protocol

  uint64_t select_timeout;
   //! Select max timeout (in ms)
  int pid;
  //! Process ID
  int roscore_pid;
  //! Roscore PID

  char *roscore_host;
  //! The roscore host (ipv4, e.g. 192.168.0.1)
  unsigned short roscore_port;
   //! The roscore port

  char *message_root_path;
```

```
 //! Directory with the message register

CrosLogLevel log_level;
CrosLogQueue* log_queue;
uint32_t log_last_id;

unsigned int next_call_id;
ApiCallQueue master_api_queue;
ApiCallQueue slave_api_queue;

//! Manage connections for XMLRPC calls
// from this node to others
XmlrpcProcess xmlrpc_client_proc[CONST_1];
// CONST_1 = CN_MAX_XMLRPC_CLIENT_CONNECTIONS
XmlrpcProcess xmlrpc_listner_proc;
//! Accept new XMLRPC connections from
// roscore or other nodes
/*! Manage connections for XMLRPC calls
// from roscore
or other nodes to this node */
XmlrpcProcess xmlrpc_server_proc[CONST_2];
// CONST_2 = CN_MAX_XMLRPC_SERVER_CONNECTIONS

//! Manage connections for TCPROS calls
// from this node to others
TcprosProcess tcpros_client_proc[CONST_3];
// CONST_3 = CN_MAX_TCPROS_CLIENT_CONNECTIONS
TcprosProcess tcpros_listner_proc;
//! Accept new TCPROS connections
// from roscore or other nodes

/*! Manage connections for TCPROS
 between this and other nodes  */
TcprosProcess tcpros_server_proc[CONST_4];
// CONST_4 = CN_MAX_TCPROS_SERVER_CONNECTIONS

//! Manage connections for RPCROS calls
// from this node to others
TcprosProcess rpcros_listner_proc;
//! Accept new TCPROS connections
// from roscore or other nodes
```

```
  /*! Manage connections for RPCROS
   between this and other nodes  */
  TcprosProcess rpcros_server_proc[CONST_5];
  // CONST_5 = CN_MAX_RPCROS_SERVER_CONNECTIONS

  PublisherNode pubs[CONST_6];
  // CONST_6 = CN_MAX_PUBLISHED_TOPICS
  //! All the published topic, defined
  // by PublisherNode structures
  SubscriberNode subs[CONST_7];
  // CONST_7 = CN_MAX_SUBSCRIBED_TOPICS
  //! All the subscribed topic, defined
  // by PublisherNode structures
  ServiceProviderNode services[CONST_8];
  // CONST_8 = CN_MAX_SERVICE_PROVIDERS
  //! All the services to register
  ParameterSubscription paramsubs[CONST_9];
  // CONST_9 = CN_MAX_PARAMETER_SUBSCRIPTIONS

  int n_pubs;
  //! Number of node's published topics
  int n_subs;
  //! Number of node's subscribed topics
  int n_services;
  //! Number of registered services
  int n_paramsubs;

};
```

Appendice C

cRosMessage type

```
typedef enum CrosMessageType
{
  CROS_CUSTOM_TYPE = 0,
  CROS_STD_MSGS_INT8,
  CROS_STD_MSGS_UINT8,
  CROS_STD_MSGS_INT16,
  CROS_STD_MSGS_UINT16,
  CROS_STD_MSGS_INT32,
  CROS_STD_MSGS_UINT32,
  CROS_STD_MSGS_INT64,
  CROS_STD_MSGS_UINT64,
  CROS_STD_MSGS_FLOAT32,
  CROS_STD_MSGS_FLOAT64,
  CROS_STD_MSGS_STRING,
  CROS_STD_MSGS_BOOL,
  CROS_STD_MSGS_TIME,
  CROS_STD_MSGS_DURATION,
  CROS_STD_MSGS_HEADER,
  // deprecated
  CROS_STD_MSGS_CHAR,
  CROS_STD_MSGS_BYTE
} CrosMessageType;

typedef struct cRosMessageField cRosMessageField;
typedef struct cRosMessage cRosMessage;
```

```
struct cRosMessageField
{
    char *name;
    union data
    {
      uint8_t opaque[8];
      int8_t as_int8;
      uint8_t as_uint8;
      int16_t as_int16;
      uint16_t as_uint16;
      int32_t as_int32;
      uint32_t as_uint32;
      int64_t as_int64;
      uint64_t as_uint64;
      float as_float32;
      double as_float64;
      char *as_string;
      cRosMessage *as_msg;
      int8_t *as_int8_array;
      uint8_t *as_uint8_array;
      int16_t *as_int16_array;
      uint16_t *as_uint16_array;
      int32_t *as_int32_array;
      uint32_t *as_uint32_array;
      int64_t *as_int64_array;
      uint64_t *as_uint64_array;
      float *as_float32_array;
      double *as_float64_array;
      char **as_string_array;
      cRosMessage **as_msg_array;
      void *as_array;
    } data;
    int size;
    int is_const;
    int is_array;
    int is_fixed_array;
    int array_size;
    int array_capacity;
    CrosMessageType type;
    char *type_s;
};
```

```
typedef struct t_msgDef cRosMessageDef;

struct cRosMessage
{
    cRosMessageField **fields;
    cRosMessageDef* msgDef;
    char* md5sum;
    int n_fields;
};
```

Appendice D

CrosNode ST

```
CrosNode : STRUCT
name : String;
host : String;
xmlrpc_port : unsignedShort;
tcpros_port : unsignedShort;
rpcros_port : unsignedShort;

select_timeout : _uint64_t;
pid : _int;
roscore_pid : _int;

roscore_host : String;
roscore_port : unsignedShort;

message_root_path : String;

log_level : CrosLogLevel;
log_queue : REFERENCE TO CrosLogQueue;
log_last_id : _uint32_t;

next_call_id : unsignedInt;
master_api_queue : ApiCallQueue;
slave_api_queue : ApiCallQueue;

xmlrpc_client_proc : ARRAY[0..5] OF XmlrpcProcess;
(*ARRAY_LENGTH = CN_MAX_XMLRPC_CLIENT_CONNECTIONS*)

xmlrpc_listner_proc : XmlrpcProcess;
```

```
xmlrpc_server_proc: ARRAY[0..4] OF XmlrpcProcess;
(*ARRAY_LENGTH = CN_MAX_XMLRPC_SERVER_CONNECTIONS*)

tcpros_client_proc : ARRAY[0..5] OF TcprosProcess;
(*ARRAY_LENGTH = CN_MAX_TCPROS_CLIENT_CONNECTIONS*)

tcpros_listner_proc : TcprosProcess;

tcpros_server_proc : ARRAY[0..4] OF TcprosProcess;
(*ARRAY_LENGTH = CN_MAX_TCPROS_SERVER_CONNECTIONS*)

rpcros_listner_proc : TcprosProcess;

rpcros_server_proc : ARRAY[0..7] OF TcprosProcess;
(*ARRAY_LENGTH = CN_MAX_RPCROS_SERVER_CONNECTIONS*)

pubs : ARRAY[0..4] OF PublisherNode;
(*ARRAY_LENGTH = CN_MAX_PUBLISHED_TOPICS*)

subs : ARRAY[0..4] OF SubscriberNode;
(*ARRAY_LENGTH = CN_MAX_SUBSCRIBED_TOPICS*)

services : ARRAY[0..7] OF ServiceProviderNode;
(*ARRAY_LENGTH = CN_MAX_SERVICE_PROVIDERS*)

paramsubs : ARRAY[0..19] OF ParameterSubscription;
(*ARRAY_LENGTH = CN_MAX_PARAMETER_SUBSCRIPTIONS*)

n_pubs : _int;
n_subs : _int;
n_services : _int;
n_paramsubs : _int;
```

Bibliografia

[1] Ros industrial, URL: http://rosindustrial.org

Sito di presentazione di ROS-Industrial:
ROS-Industrial is an open-source project that extends the advanced capabilities of ROS software to manufacturing.

[2] cros, URL: https://github.com/ros-industrial/cros

Sito di accesso al software open-source cROS.

[3] Ros wiki, URL: http://wiki.ros.org/it.

Sito in aggiunta al [4] per aiutare i fruitori di ROS.

[4] Ros, URL: http://www.ros.org.

Sito ufficiale contenente tutte le informazioni necessarie relative alla struttura e al funzionamento di ROS.

[5] Dr. Stefano Vitturi Dr. Federico Tramarin. Dispense corso di reti di comunicazione industriali, 2016

Dispense usate per l'approfondimento del protocollo TCP/IP e dei PLC.

[6] Kim N. King. *Programmazione in C.* Apogeo, Maggioli Editore, 2014

Manuale completo per la programmazione in C.

[7] Walter Savitch. *Absolute C++.* Pearson, Addison-Wesley, 5th edition, 2013

Manuale completo per la programmazione in C++.