

UNIVERSITÀ POLITECNICA DELLE MARCHE
INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE
**Interfacciamento del cobot TM5-900 con
ROS per la gestione degli ostacoli**



Corso di
DYNAMICS AND CONTROL OF INTELLIGENT ROBOTS AND VEHICLES
Anno accademico 2020-2021

Studenti:
Christian Centorame
Maria Cristina Giannini

Professore:
Andrea Bonci
Dottorando:
Giacomo Nabissi



Dipartimento di Ingegneria dell'Informazione

Indice

Introduzione	1
1 Robot collaborativo TM5-900	2
1.1 Hardware	2
1.1.1 Manipolatore	3
1.1.2 Control Box	4
1.1.3 Telecomando	5
1.2 TMflow	6
1.2.1 Creare ed eseguire un programma TMflow	6
1.2.1.1 Avviare il robot	6
1.2.1.2 Avviare TMflow	7
1.2.1.3 Creare un progetto TMflow	8
1.2.1.4 Eseguire un progetto TMflow	9
1.2.2 Ethernet Slave e External Script	10
1.2.2.1 Ethernet Slave	11
1.2.2.2 External Script	13
1.2.2.3 Comandi in modalità External Script	16
2 ROS	21
2.1 Concetti fondamentali e comandi	22
2.1.1 Nodo master	23
2.1.2 Nodi	23
2.1.3 Topic e messaggi	24
2.1.4 Servizi	28
2.1.5 Azioni	30
2.1.6 File di launch	31
2.2 RViz e MoveIt	32
2.3 Gazebo	33
3 Driver ROS	34
3.1 Utilizzo driver ROS	34
3.1.1 File di launch	35
3.1.2 Demo	37
3.2 Nodo driver	38
3.2.1 Servizio <i>SendScript</i>	39

3.2.2 Azione <i>FollowJointTrajectory</i>	39
4 Setup hardware e software	43
4.1 Configurazione macchina virtuale Ubuntu 18.04	43
4.2 Installazione e configurazione ROS Melodic	45
4.3 Download e compilazione del repository GitHub	46
4.4 Connessione fisica robot-pc	47
4.5 Configurazione TMflow	48
4.5.1 Configurazione di rete	48
4.5.2 Ethernet Slave	48
4.5.3 Programma TMflow	50
4.5.4 Test connessione	52
5 Driver ROS modificato	53
5.1 Gestione ostacoli	53
5.1.1 Problema	53
5.1.2 Soluzione	53
5.2 Gestione ostacoli con traiettoria pianificata	55
5.3 Gestione ostacoli con traiettoria predefinita	56
6 Utilizzo driver ROS modificato	61
6.1 test_functions	61
6.2 loop_trajectory	63
6.3 pick_place_trajectory	63
6.4 moveit_trajectory	64
7 Sviluppi futuri	65
7.1 Gestione reale degli ostacoli	65
7.2 Modbus	65
7.3 Problemi aperti	66
7.4 ROS2	67
Appendici	67
A Simulazione con MoveIt e Gazebo	68
B Codice gestione ostacoli con traiettoria pianificata	70
C Codice gestione ostacoli con traiettoria predefinita	75

Introduzione

L'obiettivo del progetto è quello di capire come poter interfacciare il robot collaborativo TM5-900 della OMRON TM con l'ambiente ROS (Robot Operating System) per poter gestire la comparsa di ostacoli nello spazio di lavoro del robot. Nello specifico l'idea è quella di utilizzare un sistema di telecamere esterne per rilevare le potenziali collisioni tra il cobot e l'ambiente e di sfruttare quindi tale informazione per ripianificare la traiettoria del manipolatore in modo che possa portare a termine il task assegnatogli evitando gli ostacoli.

Attualmente il robot, pur essendo a tutti gli effetti di tipo collaborativo, non è in grado di rilevare la presenza di ostacoli e di evitarli. Non possiede infatti né sensori di prossimità né sistemi di visione adatti a questo (è dotato soltanto di una fotocamera per il riconoscimento di oggetti). Il manipolatore si accorge quindi di un eventuale ostacolo solo nel momento della collisione a seguito della quale si blocca e deve essere riavviato. È quindi fondamentale che a monte venga effettuata una valutazione del rischio per configurare correttamente le impostazioni di sicurezza del robot, limitando in maniera opportuna velocità, coppia e forza dei giunti e dell'end-effector.

Per poter implementare una qualunque logica di rilevamento degli ostacoli e prevenzione degli urti occorre quindi necessariamente ricorrere a dei sensori aggiuntivi esterni al robot. In particolare l'idea è di affidarsi ad un sistema di visione che possa rilevare le potenziali collisioni e ricostruire l'ambiente 3D in cui il robot opera. Partendo da tali dati è possibile pensare di ripianificare la traiettoria del manipolatore considerando la presenza dell'ostacolo. Tutto questo non può però essere gestito direttamente da TMflow che è il software che permette di configurare e programmare i robot collaborativi della OMRON attraverso un'interfaccia grafica. Tali funzionalità sono invece implementabili con un framework per la robotica come ROS.

Questo progetto si concentra sulla prima parte del problema, ovvero su come interfacciare il robot con ROS. Non è stata quindi gestita tutta la parte di rilevamento degli ostacoli e di ripianificazione della traiettoria. Ci si è invece limitati a simulare via software la presenza di un generico ostacolo e a gestirlo semplicemente fermando il robot e facendolo tornare alla posizione di home.

Per fare questo si è partiti da un repository GitHub ancora in fase di sviluppo che implementa un driver per l'interfacciamento del TM5-900 con ROS. Questo repository è stato prima testato per verificare la correttezza del setup e poi clonato e modificato per ottenere un'applicazione ROS per la gestione degli ostacoli.

Capitolo 1

Robot collaborativo TM5-900

Il manipolatore utilizzato nel presente progetto è il TM5-900. Si tratta di un robot collaborativo della serie TM prodotta dalla Techman Robot ma venduta come prodotto co-branded attraverso la rete di distribuzione della OMRON [1].

Tutti i manipolatori di tale linea hanno la medesima struttura e si differenziano soltanto in termini di dimensioni e portata (Figura 1.1). La sigla TM5-900 si riferisce proprio al valore di tali parametri. I cobot TM5 hanno infatti una portata massima di circa 5kg, nello specifico 6 kg il TM5-700 e 4 kg il TM5-900. Il TM5-900 ha poi un raggio d'azione di 900mm.



Figure 1.1: Robot collaborativi OMRON TM

In questo capitolo vengono riportati sinteticamente tutti gli aspetti relativi all'hardware e al software del cobot TM5-900 che è opportuno conoscere per la comprensione del progetto. Per ulteriori informazioni e approfondimenti si rimanda alla documentazione ufficiale del robot.

1.1 Hardware

Il TM5-900 utilizzato è costituito dalle seguenti componenti hardware fondamentali:

- Manipolatore

- Control box
- Telecomando

Tali componenti sono disposti all'interno di una workstation mobile come mostrato in Figura 1.2.



Figure 1.2: *Hardware*

1.1.1 Manipolatore

Il TM5-900 mostrato in Figura 1.3 è un manipolatore costituito da 6 giunti rotoidali che connettono tra loro 7 link, uno dei quali è la base fissa. Si tratta quindi di un manipolatore a 6 gradi di libertà.

Il robot è inoltre dotato di un sistema di visione integrato costituito da una fotocamera a colori.

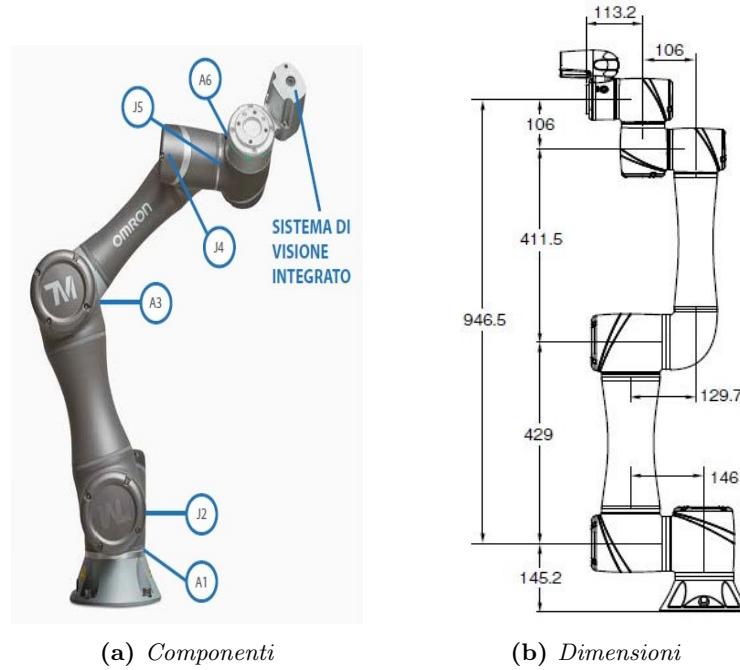


Figure 1.3: TM5-900

Per maggiori informazioni sulle specifiche del robot si faccia riferimento al datasheet [2].

1.1.2 Control Box

La control box è il computer che si occupa del controllo del robot. È possibile programmare il manipolatore con TMflow anche collegando semplicemente monitor, tastiera e mouse alla control box, sfruttando il client TMflow già installato su di essa.

In alternativa si può utilizzare un computer esterno installandovi TMflow (il software può essere scaricato dal sito ufficiale dopo essersi registrati). Il computer va poi collegato alla control box con una connessione cablata o wireless. Una delle soluzioni possibili è quella di collegare fisicamente computer e control box con un cavo Ethernet (RJ45). Il cavo di rete può essere collegato a una qualunque delle tre porte LAN della control box. È però opportuno utilizzare una delle 2 porte GigE (Gigabit Ethernet) in quanto la porta LAN deve essere utilizzata per l'interfacciamento con ROS come si capirà meglio in seguito.

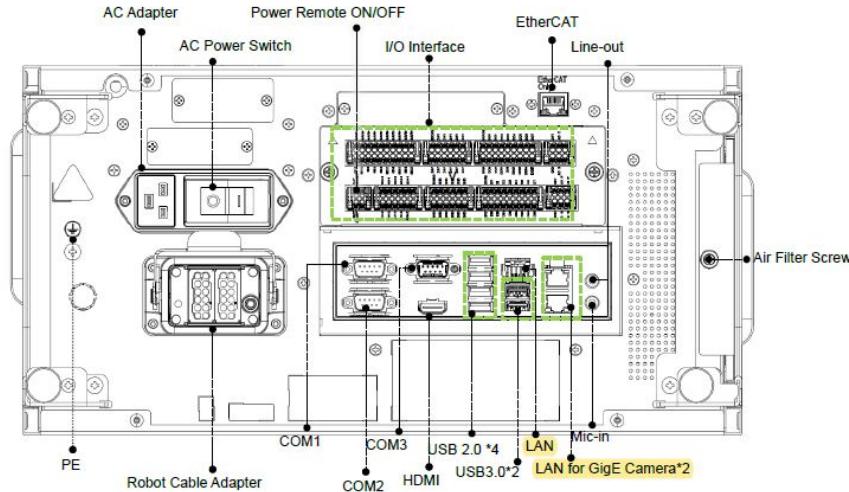


Figure 1.4: Interfacce della control box

Per maggiori dettagli sulle diverse possibilità per l'utilizzo di TMflow si faccia riferimento al Capitolo 2.2.4 *TM Robot HMI TMflow Operation* del *Software Manual TMflow* [3].

1.1.3 Telecomando

Il manipolatore è dotato di un telecomando connesso alla control box e mostrato in Figura 1.5.



Figure 1.5: Telecomando

Il telecomando offre le seguenti funzioni:

- **Emergency Switch:** Pulsante di emergenza per bloccare il robot;
- **Power Button:** Pulsante per accendere (singola pressione) e spegnere (tenendo premuto) il robot;
- **M/A Mode Switch Button:** Pulsante per cambiare modalità (Manuale/Automatica);
- **Play/Pause Button:** Pulsante per avviare il progetto o metterlo in pausa se è già in esecuzione

- **Stop Button:** Pulsante per interrompere il progetto;
- **+- Button:** Pulsanti per modificare la velocità del progetto in modalità Manuale;
- **Power Indicator:** Indicatore luminoso dello stato del robot (spento/startup/acceso);
- **Mode Indicator Lights:** Indicatori luminosi della modalità corrente del robot (Manuale/Automatica);
- **Speed Indicator:** Indicatori luminosi della velocità del progetto corrente.

Qualche informazione aggiuntiva sul telecomando si può trovare al capitolo 4.2.3.1 *Robot Stick* dell'*Hardware Installation Manual* [4].

1.2 TMflow

TMflow è l’interfaccia uomo-macchina grafica che permette di configurare il robot e di programmarlo in modo semplice e intuitivo mediante diagrammi di flusso.

In questo capitolo vengono riportati solo gli aspetti del software che è necessario conoscere ai fini del progetto. Per una descrizione dettagliata delle funzionalità di TMflow si rimanda al manuale dedicato [3].

La versione del software installata sulla control box del robot utilizzato per il seguente progetto è la 1.82.5100. Si fa quindi notare che, nel caso in cui si volesse utilizzare TMflow da un computer esterno, occorre necessariamente scaricarvi tale versione.

1.2.1 Creare ed eseguire un programma TMflow

Di seguito sono riportati in dettaglio gli step da seguire per avviare TMflow, creare un nuovo programma e mandarlo in esecuzione.

1.2.1.1 Avviare il robot

Per avviare il robot seguire i seguenti passi:

1. Alimentare il robot;
2. Verificare che il manipolatore sia in una configurazione sicura. Durante l’avvio infatti i giunti si muovono leggermente (massimo 5° per giunto) in quanto ne viene eseguita automaticamente la calibrazione;
3. Verificare che l’interruttore di emergenza sul telecomando non sia premuto;
4. Avviare il robot premendo il pulsante di alimentazione. Il manipolatore si avvia in modalità startup;
5. Attendere la fine della fase di startup (impiega qualche minuto). Durante l’avvio l’indicatore di alimentazione sul telecomando e l’anello luminoso sull’end effector del robot lampeggiano in rosso. Al termine l’indicatore di alimentazione diventa rosso fisso mentre l’anello luminoso diventa blu fisso ad indicare che il robot è in modalità automatica.

Gli step precedentemente descritti presuppongono che il TM5-900 sia già stato correttamente installato. La procedura di installazione si può però trovare in [4].

Per quanto riguarda invece lo spegnimento del robot ci sono i seguenti 2 metodi:

- Da TMflow: Accedere alla sezione *Shutdown* da \equiv ;
- Da telecomando: Tenere premuto il pulsante *Power* del telecomando finché il robot non si spegne.

1.2.1.2 Avviare TMflow

Come già accennato nella Sezione 1.1.2, TMflow può essere avviato in modalità diverse. Nel presente progetto sono state testate le seguenti 2 modalità:

- **TMflow locale:** Il client TMflow è in esecuzione sulla control box;
- **TMflow remoto con connessione cablata:** il client TMflow è in esecuzione su un computer esterno collegato alla control box mediante cavo Ethernet.

Gli step per avviare TMflow con la prima modalità sono i seguenti:

1. Collegare schermo, tastiera e mouse alla control box;
2. Sul monitor si aprirà automaticamente la schermata iniziale di TMflow dove sarà visualizzato il TM5-900;
3. Eseguire il login accedendo a \equiv . Di default si può accedere con ID administrator e PW vuota come mostrato in Figura 1.6;

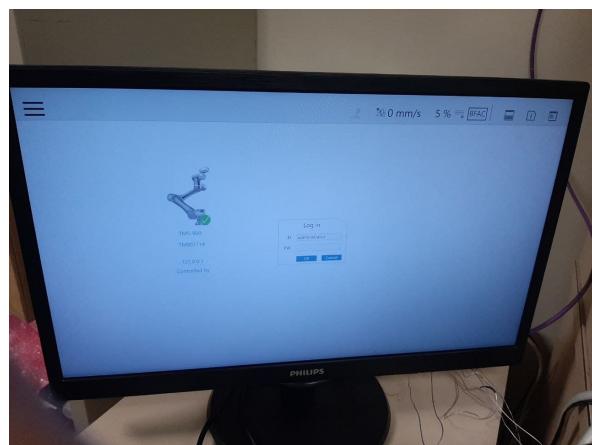


Figure 1.6: Schermata iniziale di TMflow (locale)

4. Ottenere il controllo del robot con *Get Control*.

Per poter invece utilizzare TMflow su un computer remoto i passi sono i seguenti:

1. Installare TMflow sul computer assicurandosi di scaricare la stessa versione del software della control box;
2. Collegare fisicamente computer e control box con un cavo di rete facendo attenzione ad utilizzare una porta LAN GigE (Sezione 1.1.2);
3. Non è necessario modificare le impostazioni di rete né lato computer né lato control box;
4. Da computer aprire TMflow e attendere che il robot venga rilevato e appaia sulla schermata iniziale. Se necessario fare un refresh con l'apposito pulsante per aggiornare la lista dei robot disponibili;
5. Eseguire il login analogamente alla precedente modalità;
6. Ottenere il controllo del robot come per la precedente modalità.

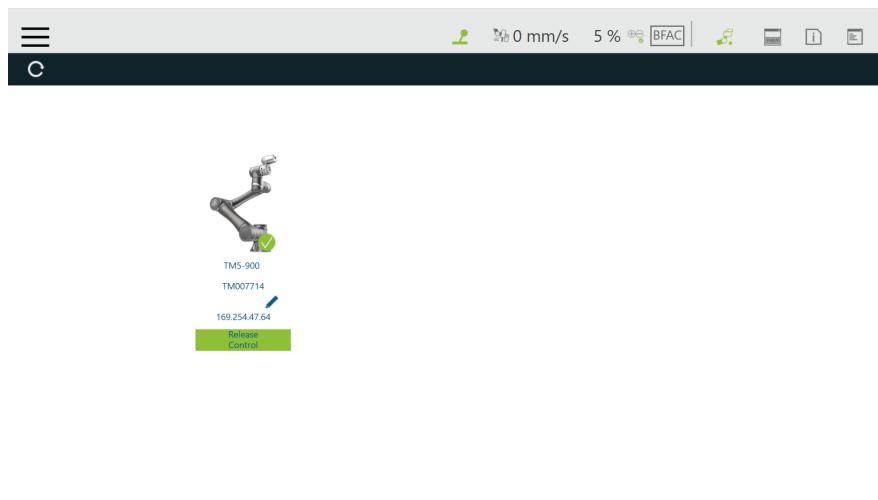


Figure 1.7: Schermata iniziale di TMflow (remoto)

Per poter cambiare modalità di utilizzo di TMflow occorre prima rilasciare il controllo del robot con *Release Control* ed effettuare il logout.

Si fa notare che, indipendentemente dalla modalità adottata, è comunque necessario il robot per poter utilizzare TMflow. Il software è infatti pensato come un'interfaccia uomo-macchina e non permette di svolgere alcuna funzione offline, neppure la scrittura di un programma TMflow.

1.2.1.3 Creare un progetto TMflow

Dopo aver avviato il robot e il software, gli step da seguire per poter creare un nuovo progetto TMflow sono:

1. Assicurarsi che il robot sia in modalità manuale. La maggior parte delle funzioni di TMflow sono infatti accessibili solo in tale modalità. La modalità operativa del robot si può vedere sia sul telecomando sia dall'anello luminoso dell'end effector: la luce verde indica la modalità manuale, la luce blu quella automatica;
2. Se il robot è in modalità automatica passare in manuale semplicemente premendo il pulsante M/A dedicato sul telecomando;
3. Su TMflow accedere alla sezione *Project* da \equiv (la sezione non è accessibile se il robot è in modalità automatica);
4. Dalla nuova schermata è possibile creare un nuovo progetto o aprire un progetto esistente. Si fa notare che i progetti salvati sono memorizzati sulla control box, non sono quindi accessibili offline;
5. Creare un progetto TMflow. Un progetto TMflow è semplicemente un diagramma di flusso che può essere creato connettendo dei nodi predefiniti ma parametrizzati;
6. Salvare il progetto.

In questo progetto non è stata approfondita la programmazione del robot in TMflow in quanto il controllo del robot è stato fatto interamente da ROS. I dettagli sulla programmazione si possono però trovare in [3].

1.2.1.4 Eseguire un progetto TMflow

I progetti TMflow salvati sono riportati nella sezione *Run Setting* accessibile sempre da \equiv e mostrata in Figura 1.8.

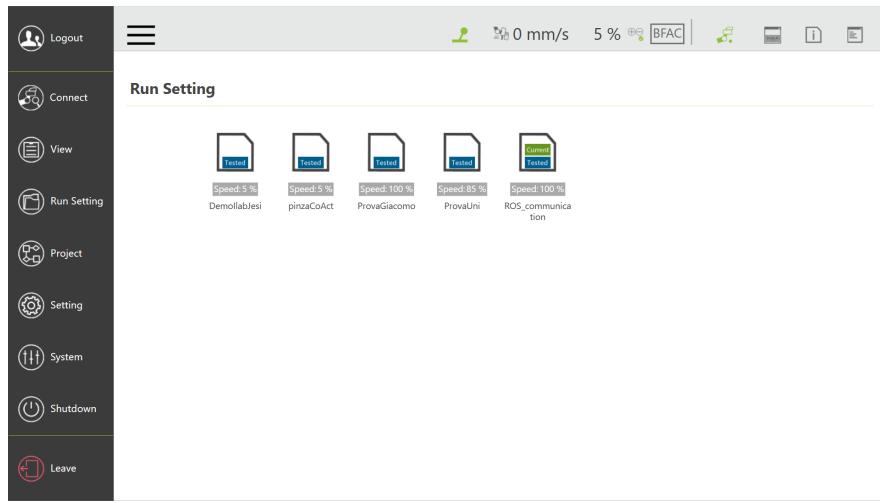


Figure 1.8: Schermata di Run Setting

Il progetto etichettato come *Current* è il progetto correntemente selezionato. I progetti etichettati come *Tested* sono i programmi già testati in modalità manuale e che quindi possono essere eseguiti in modalità automatica.

Un progetto quindi può sempre essere eseguito in modalità manuale mentre per andare in esecuzione in modalità automatica deve prima essere testato. Gli step per testare un nuovo programma TMflow sono i seguenti:

1. Aprire il progetto dalla sezione *Project*;
2. Con il programma aperto premere il pulsante *Play/Pause* del telecomando per mandare in esecuzione il progetto in modalità manuale;
3. La modalità manuale serve per testare il codice e regolarne la velocità per l'esecuzione in modalità automatica. Il programma va in esecuzione con una velocità di progetto iniziale del 5%. L'utente può modificare la velocità durante l'esecuzione con gli appropriati pulsanti +- del telecomando. Il valore corrente della velocità è visibile sia sul telecomando sia su TMflow;
4. Una volta scelta la velocità, per memorizzarla e completare la fase di test occorre tenere premuto il pulsante M/A e successivamente premere i pulsanti +- del telecomando nell'ordine "+-++-";
5. Tale procedura permette di etichettare il progetto come *Tested* e va ripetuta se si desidera modificare la velocità del progetto.

Una volta testato il programma è possibile mandarlo in esecuzione in modalità automatica con la velocità scelta. Per farlo è sufficiente seguire i seguenti step:

1. Accedere alla sezione *Run Setting*;
2. Verificare che il progetto corrente sia quello che si desidera eseguire; altrimenti selezionare il progetto corretto per renderlo il progetto corrente;
3. Passare in modalità automatica tenendo premuto il pulsante M/A e successivamente premendo i pulsanti +- del telecomando nell'ordine "+-++-";
4. Mandare in esecuzione il programma premendo il pulsante *Play/Pause* del telecomando.

Il programma TMflow in esecuzione può essere messo in pausa o interrotto rispettivamente con i pulsanti *Play/Pause* e *Stop*.

1.2.2 Ethernet Slave e External Script

Per il presente progetto non sono state approfondite tutte le funzionalità di TMflow. In questa prima fase si è scelto infatti di implementare su ROS tutta la logica, dalla definizione della traiettoria iniziale alla gestione dell'ostacolo fino alla ripianificazione. Non è stato quindi necessario sviluppare un vero e proprio programma su TMflow ma ci si è limitati a utilizzare le funzionalità esposte dai socket TCP *Ethernet Slave* e *External Script* che TMflow mette a disposizione.

In Figura 1.9 sono visibili le porte su cui i 2 socket server si mettono in ascolto.

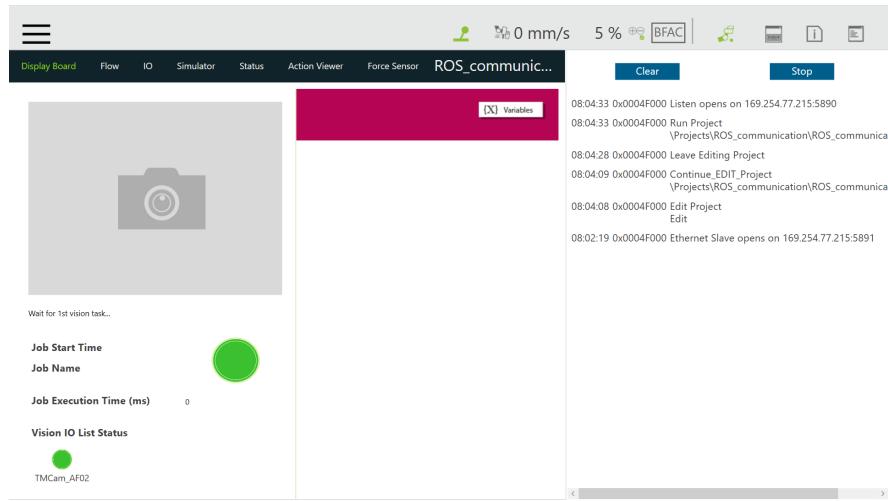


Figure 1.9: Porte di ascolto dei socket TCP Ethernet Slave e External Script

In sintesi l'uso combinato di questi 2 socket permette di controllare il robot dall'esterno collegando un computer alla control box con un cavo di rete. Tramite *Ethernet Slave* un dispositivo esterno può infatti acquisire informazioni sullo stato del manipolatore, mentre con la funzionalità *External Script* può inviargli dei comandi.

Di seguito tali funzionalità vengono descritte in dettaglio. Per maggiori informazioni il riferimento è il manuale *Expression editor and listen node* [5].

1.2.2.1 Ethernet Slave

L'*Ethernet Slave* è un socket server che, una volta abilitato, si attiva sulla porta 5891 della control box. In sintesi questo socket permette ai dispositivi che vi si connettono di leggere e scrivere il valore delle variabili contenute in una *Data Table* che l'utente può personalizzare.

Per abilitare l'*Ethernet Slave* occorre accedere da \equiv alla sezione *Setting* \rightarrow *Connection* \rightarrow *Ethernet Slave* mostrata in Figura 1.10.

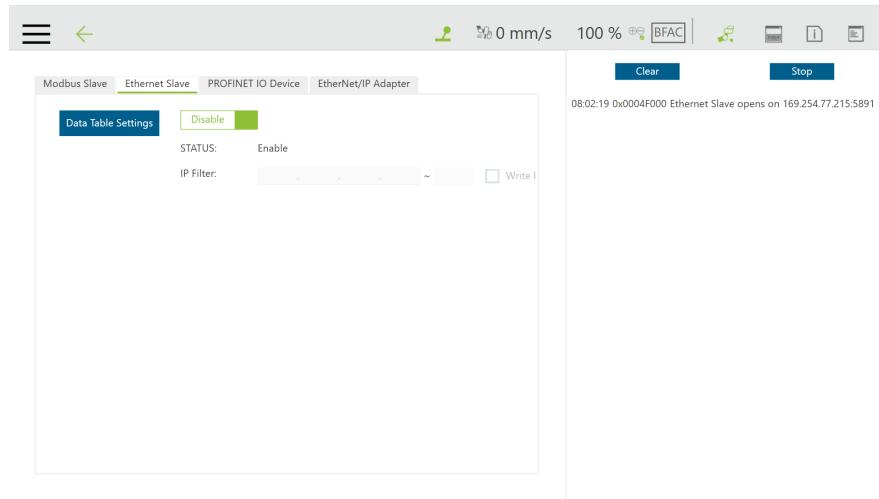


Figure 1.10: Schermata per attivare Ethernet Slave

Da qui è possibile attivare e disabilitare il socket con il pulsante dedicato. È anche possibile aggiungere dei filtri per limitare in lettura e/o scrittura gli indirizzi IP che possono utilizzare l'*Ethernet Slave*.

Con il pulsante *Data Table Settings* si accede alla schermata per definire il contenuto della Data Table (Figura 1.11). In dettaglio l'utente può creare un *Transmit File* selezionando le variabili da esporre all'esterno in lettura e/o scrittura tramite *Ethernet Slave*. È possibile selezionare non soltanto le variabili predefinite ma anche nuove variabili definite dall'utente. L'utente può anche scegliere il formato con cui sono trasmessi i dati della Data Table tra binario, stringa e JSON.

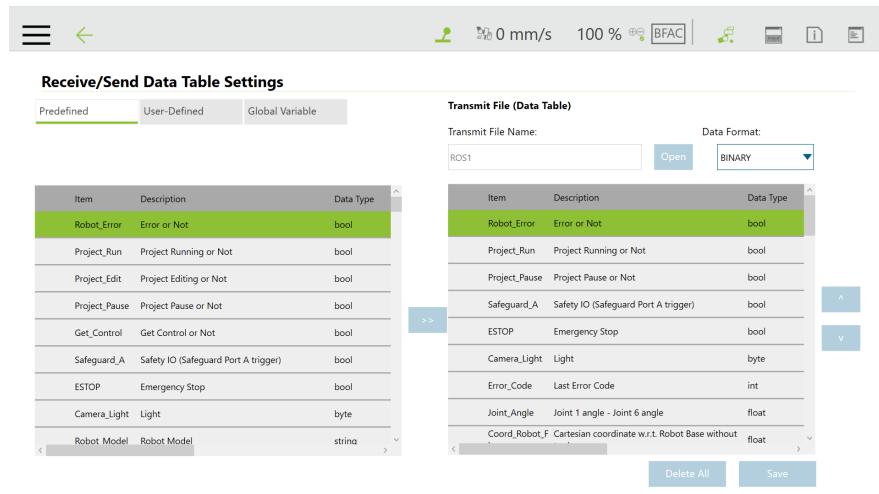


Figure 1.11: Schermata per definire la Data Table

Tutte le variabili inserite nella Data Table vengono infatti trasmesse periodicamente dal server a tutti i client connessi all'*Ethernet Slave*. I client possono però anche richiedere al server il valore corrente di specifiche variabili o modificarne il valore.

Tutte queste funzionalità sono implementate nel protocollo applicativo TMSVR (Figura 1.12) che è alla base dell’*Ethernet Slave*.

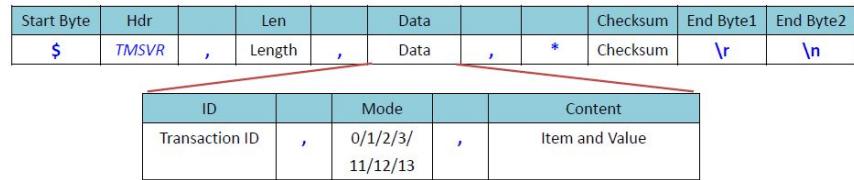


Figure 1.12: Protocollo TMSVR

Nel presente progetto l’*Ethernet Slave* è stato utilizzato soltanto per ricevere periodicamente lo stato corrente del manipolatore e non è stato necessario approfondire il TMSVR. Ulteriori informazioni sull’*Ethernet Slave* e sul protocollo TMSVR si possono però trovare al Capitolo 10 *TM Ethernet Slave* del manuale *Expression editor and listen node* [5].

1.2.2.2 External Script

Il secondo socket server TCP utilizzato è il *Listen* che si attiva invece sulla porta 5890 della control box. In sintesi questo socket permette ai dispositivi che vi si connettono di inviare dei comandi al manipolatore e di verificarne lo stato di esecuzione.

Per abilitare il *Listen* occorre agire direttamente a livello di programma TMflow inserendo un nodo di tipo *Listen* (Figura 1.13). Soltanto finché il flusso del programma rimane su tale nodo, la modalità *External Script* rimane attiva ed è possibile comandare il robot dall'esterno.

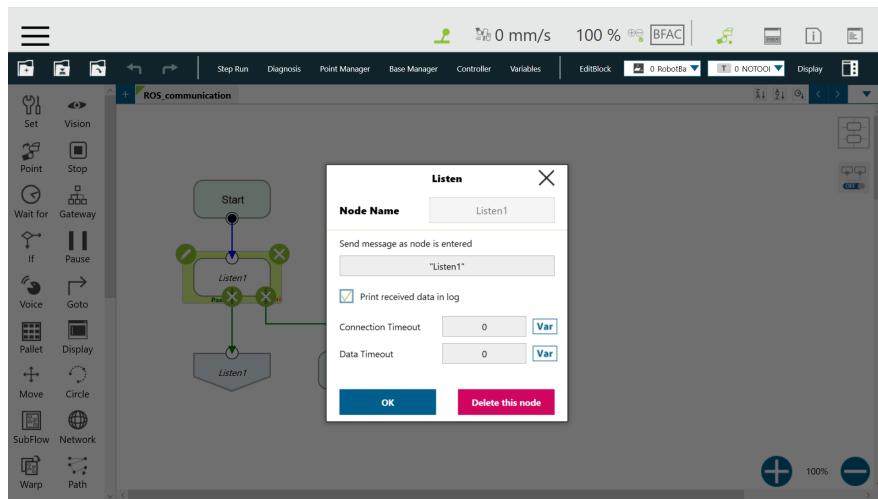


Figure 1.13: Nodo Listen

Quando il processo entra nel nodo Listen vi rimane finché non si verifica una delle seguenti condizioni di uscita:

- *Pass*: se viene inviato il comando *ScriptExit()* per interrompere la modalità *External Script* o se si interrompe il programma;

- *Fail*: se si verificano le condizioni di *Connection timeout* o *Data timeout* (sulla base dei parametri definiti alla creazione del nodo) o se il processo entra nel nodo *Listen* prima che il socket server sia disponibile.

Da qui si può già comprendere un limite della modalità *External Script* che non permette al robot di rimanere in ascolto di comandi esterni mentre esegue un altro programma TMflow. Ed è proprio per questo motivo che non è stato possibile gestire la traiettoria iniziale del robot con TMflow ma è stato necessario implementarla sempre tramite *External Script*. Tale limite e una sua potenziale soluzione sono trattati nella Sezione 7.2.

Finché il processo rimane sul nodo *Listen* il robot è in modalità *External Script* ed esegue i comandi che gli vengono inviati dall'esterno. L'esecuzione di tali comandi può però non essere istantanea. Un comando di assegnazione di una variabile è ad esempio immediato mentre un comando di movimento richiede del tempo per essere eseguito. Per questo motivo i comandi che il robot riceve vengono messi in coda ed eseguiti in ordine. I comandi in coda vengono eseguiti anche se il robot esce dalla modalità *External Script*.

La modalità *External Script* si basa essenzialmente sui seguenti 2 protocolli applicativi:

- TMSCT: per inviare dei comandi al robot dall'esterno e avere un feedback sulla correttezza del comando inviato;
- TMSTA: per avere informazioni sullo stato di esecuzione dei comandi inviati con TMSCT.

Per quanto riguarda il protocollo TMSCT la struttura dei pacchetti è riportata in Figura 1.14.

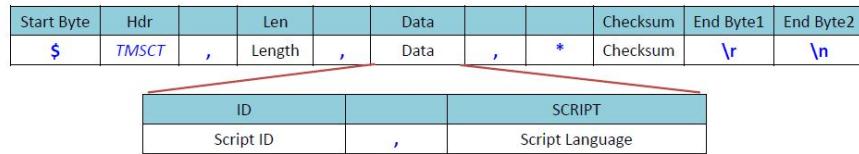


Figure 1.14: Protocollo TMSCT

In particolare la sezione *Data* di ogni pacchetto TMSCT che viene inviato dal client esterno al robot è formata dalle seguenti parti:

- ID: identificativo dello script che può essere scelto arbitrariamente. Serve per specificare a quale script si riferisce il messaggio di ritorno restituito dalla control box;
- SCRIPT: Script con i comandi che il robot deve eseguire. I singoli comandi devono essere separati dal separatore \r\n.

I pacchetti TMSCT vengono ricevuti dal nodo *Listen* che ne verifica la correttezza ed invia un pacchetto TMSCT di analoga struttura in risposta. I pacchetti di risposta contengono l'id dello script a cui si riferiscono e l'informazione sulla correttezza o meno dello script ricevuto (script valido/script valido con warning/script non valido).

Nel Listato 1.1 è riportato un esempio di scambio di pacchetti tra client esterno e robot con protocollo TMSCT. In questo esempio il client invia uno script costituito da più comandi che viene etichettato con ID = 2. Il robot risponde a tale script confermandone la validità.

```
Client -> robot
    $TMSCT,64,2,ChangeBase("RobotBase")\r\n
    ChangeTCP("NOTOOL")\r\n
    ChangeLoad(10.1),*68\r\n
Robot -> client
    $TMSCT,4,2,OK,*5F\r\n
```

Listing 1.1: Esempio di TMSCT

Il protocollo TMSCT è stato ampiamente utilizzato in questo progetto in quanto il controllo del robot è stato fatto totalmente in modalità *External Script*. Nella Sezione 1.2.2.3 verranno poi descritte in dettaglio le funzioni che sono state utilizzate per controllare il robot dall'esterno tramite ROS.

L'altro protocollo previsto dalla modalità *External Script* è il TMSTA riportato in Figura 1.15.

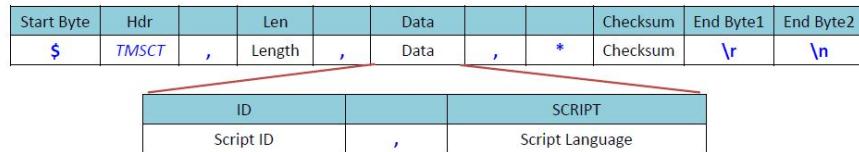


Figure 1.15: Protocollo TMSTA

Questo protocollo si differenzia dal protocollo TMSCT in quanto non richiede che il flusso del programma sia sul nodo *Listen* e supporta funzionalità diverse. La funzione del protocollo TMSTA utilizzata in questo progetto è la *TMSTA SubCmd 01* che permette al client di ottenere informazioni sullo stato di esecuzione dei comandi inviati con il protocollo TMSCT che sono stati etichettati. È possibile infatti richiedere e/o ottenere tale informazione soltanto per i comandi di cui si è richiesto di monitorare lo stato di esecuzione al momento dell'invio con TMSCT utilizzando i comandi *QueueTag* o *WaitQueueTag*. Le funzioni che il TMSCT supporta sono descritte in dettaglio nella Sezione 1.2.2.3 ma in sintesi sono dei particolari comandi che associano ad un certo altro comando un tag per poterlo monitorare.

La struttura dei pacchetti TMSTA per questa funzionalità è riportata in Figura 1.16 e prevede che il campo *SubCmd* della sezione *Data* abbia valore 01 .

Send (Robot→External Device)

SubCmd		Tag Number		Status
01	,	01 .. 15	,	true/false/none

Receive (Robot←External Device)

SubCmd		Tag Number
01	,	01 .. 15

Figure 1.16: Protocollo TMSTA con SubCmd 01

Con il *TMSTA SubCmd 01* il client può quindi richiedere al robot lo stato di esecuzione di un comando specificando il tag con cui è stato etichettato al momento dell'invio. Il robot risponde sempre con un pacchetto *TMSTA SubCmd 01* in cui riporta il tag e il relativo stato che può essere uno dei seguenti:

- *true*: il comando etichettato con quel tag è stato eseguito;
- *false*: il comando etichettato con quel tag non ha ancora completato la sua esecuzione;
- *none*: il tag richiesto non esiste.

Per avere informazioni sull'esecuzione di un comando non è però necessario che il client lo richieda esplicitamente. Il robot infatti invia automaticamente un pacchetto *TMSTA SubCmd 01* con valore dello stato *true* quando un comando etichettato viene completato.

Un esempio di pacchetto *TMSTA SubCmd 01* che il robot invia automaticamente al completamento di un comando etichettato con tag 08 è riportato nel Listato 1.2.

```
Robot -> client
$TMSTA,10,01,08,true,*6D\r\n
```

Listing 1.2: Esempio di TMSTA SubCmd 01

Il protocollo TMSTA è stato utilizzato nel presente progetto in quest'ultimo modo per poter ottenere una traiettoria eseguita in modo ciclico in modalità *External Script*. Nello specifico si anticipa che si è scelto di inviare la traiettoria da eseguire in loop in un unico pacchetto TMSCT etichettandola con la funzione *QueueTag*. Per inviare nuovamente il pacchetto con la traiettoria si attende quindi che venga ricevuto il pacchetto *TMSTA SubCmd 01* che notifica il completamento dell'esecuzione della traiettoria richiesta.

1.2.2.3 Comandi in modalità External Script

Il protocollo TMSCT permette ad un client di inviare comandi al manipolatore in modalità *External Script*. Una descrizione dettagliata dei comandi disponibili si può trovare al Capitolo 8 *Robot Motion Functions* del manuale *Expression editor and listen node* [5]. Di seguito vengono descritte brevemente soltanto le funzioni utilizzate nel progetto.

QueueTag() La funzione *QueueTag()* permette di etichettare un comando di cui si vuole monitorare lo stato di esecuzione con *TMSTA SubCmd 01*.

Una delle possibili sintassi della funzione è riportata nel Listato 1.3.

```
bool QueueTag(int tag)
```

Listing 1.3: Sintassi funzione QueueTag()

La funzione prende in ingresso il tag da associare al comando da monitorare che la precede nello script e restituisce in una variabile booleana l'esito del tag. Il tag deve essere un intero nel range [01,15]. Quando il comando etichettato viene eseguito il robot invia un pacchetto *TMSTA SubCmd 01* al client per informarlo che il comando richiesto è stato portato a termine.

Un esempio di scambio di pacchetti in cui si utilizza la *QueueTag()* è riportato nel Listato 1.4.

```
Client -> Robot
    $TMSCT,172,2,float[] targetP1= {0,0,90,0,90,0}\r\n
    PTP("JPP",targetP1,10,200,0,false)\r\n
    QueueTag(1)\r\n
    float[] targetP2 = {0,90,0,90,0,0}\r\n
    PTP("JPP",targetP2,10,200,10,false)\r\n
    QueueTag(2)\r\n
    ,*49\r\n
Robot -> client
    Dopo aver verificato la correttezza dello script ricevuto:
    $TMSCT,4,2,OK,*5F\r\n
    Dopo aver eseguito il primo comando PTP()
    $TMSTA,10,01,01,true,*64\r\n
    Dopo aver eseguito il secondo comando PTP()
    $TMSTA,10,01,02,true,*67\r\n
```

Listing 1.4: Esempio di QueueTag()

Tale funzione è stata utilizzata nel progetto per gestire una traiettoria ciclica.

StopAndClearBuffer() La funzione *StopAndClearBuffer()* (Listato 1.5) interrompe il movimento corrente del robot e cancella i comandi in coda che sono ancora in attesa di essere eseguiti. Si fa notare che la funzione non mette in pausa né interrompe il programma TMflow in esecuzione sul robot ed è quindi poi possibile inviare nuovi comandi con TMSCT.

```
bool StopAndClearBuffer()
```

Listing 1.5: Sintassi funzione StopAndClearBuffer()

Tale funzione è stata fondamentale nel progetto per gestire la comparsa di ostacoli nello spazio di lavoro del robot. Quando un ostacolo viene rilevato il robot deve infatti fermarsi e modificare la propria traiettoria. La funzione *StopAndClearBuffer()* è stata utilizzata proprio per fermare il manipolatore la cui traiettoria iniziale è data sempre in modalità *External Script*.

PTP() La funzione *PTP()* permette di inviare al robot un comando di movimento di tipo Point To Point (PTP). Si tratta di una modalità di movimento presente anche su TMflow che prevede che il manipolatore raggiunga un certo punto target seguendo il percorso più breve

in spazio giunti.

La sintassi della funzione è riportata nel Listato 1.6.

```
bool PTP(string format, float[] target, int speed, int time,
         int blending, bool precise_positioning)
```

Listing 1.6: Sintassi funzione PTP()

Gli argomenti della funzione sono i seguenti:

- *format*: Stringa che definisce il formato dello stato target, della velocità e del blending. L'unico formato disponibile per velocità e blending è quello percentuale ("P"). Per il target invece è possibile impostare "J" per definire lo stato in spazio giunti o "C" per definirlo nello spazio cartesiano;
- *target*: vettore di float che definisce lo stato target. Se lo stato è definito nello spazio giunti è dato dal valore degli angoli dei singoli giunti espressi in gradi. Se lo stato è invece definito in spazio cartesiano è formato dalle coordinate cartesiane del TCP (Tool Center Point) espresse in mm e dall'orientamento espresso come rotazione attorno agli assi in gradi;
- *speed*: velocità espressa in percentuale;
- *time*: intervallo di tempo per accelerare alla massima velocità espresso in ms;
- *blending*: blending espresso in percentuale. Definisce la tolleranza nel passaggio per il punto;
- *precise-positioning*: variabile booleana per abilitare (false) / disabilitare (true) il posizionamento preciso. Abilitandolo il robot aspetta di stabilizzarsi sul punto prima di eseguire il comando successivo.

Per maggiori informazioni sul significato di tali argomenti il riferimento è [3] in quanto il comando *PTP()* ha il proprio analogo anche su TMflow. Si anticipa però che anche su questo manuale non è del tutto esplicito il significato di alcuni argomenti (Sezione 7.3).

Nel Listato 1.7 è riportato un esempio di comando *PTP()*.

```
PTP("JPP", 0, 0, 90, 0, 90, 0, 35, 200, 0, false)
```

Listing 1.7: Esempio funzione PTP()

In dettaglio si tratta del comando per muovere il robot in modalità PTP per raggiungere il punto [0,0,90,0,90,0] in spazio giunti con una velocità al 35% e tempo per raggiungerla di 200 ms, senza blending e posizionamento preciso.

Questa funzione è stata utilizzata per definire la traiettoria del manipolatore in modalità *External Script*.

Move_PTP() La funzione *Move_PTP()* è del tutto analoga alla *PTP()*. La sintassi è la stessa così come il significato di quasi tutti gli argomenti. La differenza principale è lo stato target che non rappresenta più la posizione assoluta del manipolatore (in spazio giunti o cartesiano) ma definisce lo spostamento relativo (sempre esprimibile in spazio giunti o cartesiano).

Anche questa funzione è stata utilizzata per definire la traiettoria del manipolatore in modalità *External Script*

PVTPoint() L'ultima funzione utilizzata è la *PVTPoint()* che permette di controllare il robot in modalità PVT (Position-Velocity-Time). Questo comando si differenzia da tutti gli altri in quanto non ha il proprio equivalente su TMflow ma esiste solo in modalità *External Script*.

Per poter utilizzare tale funzione sono necessari altri 2 comandi ausiliari:

- *PVTEnter()*: comando per entrare in modalità PVT. Tale funzione ha un unico argomento (*PVT Mode*) per specificare il formato in cui sarà definita la traiettoria (0 = spazio giunti, 1 = spazio cartesiano);
- *PVTExit()*: comando per uscire dalla modalità PVT.

Il comando *PVTPoint()* permette invece di definire i singoli punti della traiettoria che il manipolatore deve eseguire. Per ogni punto viene specificata posizione, velocità e tempo che il robot deve impiegare per raggiungere quella posizione con quella velocità. La sintassi della funzione è riportata nel Listato 1.8.

```
bool PVTPoint(float[] target_position, float[] target_velocity,
              float duration)
```

Listing 1.8: Sintassi funzione PVTPoint()

Gli argomenti della funzione sono i seguenti:

- *target_position*: posizione target espressa in spazio giunti o in spazio cartesiano (in base al valore dell'argomento *PVT Mode* della *PVTEnter()*);
- *target_velocity*: velocità target espressa in spazio giunti o in spazio cartesiano (sempre in base al valore dell'argomento *PVT Mode* della *PVTEnter()*);
- *duration*: tempo richiesto per raggiungere lo stato target (posizione + velocità) espresso in ms.

Un esempio di utilizzo del comando *PVTPoint()* è riportato nel Listato 1.9.

```
PVTEnter(1)
PVTPoint(467.5,-122.2,359.7,180,0,90,50,50,0,0,0,0,0,0.5)
PVTPoint(467.5,-72.2,359.7,180,0,90,-50,50,0,0,0,0,0,0.5)
PVTPoint(417.5,-72.2,359.7,180,0,90,0,0,0,0,0,0,0,0.5)
PVTPoint(417.5,-122.2,359.7,180,0,90,50,50,0,0,0,0,0,0.5)
PVTPoint(417.5,-122.2,359.7,180,0,60,50,50,0,0,0,0,0,0.3)
PVTPoint(417.5,-122.2,359.7,180,0,90,50,50,0,0,0,0,0,0.3)
PVTExit()
```

Listing 1.9: Esempio funzione PVTPoint()

Tale funzione è stata utilizzata nel progetto per far eseguire al robot la traiettoria pianificata da ROS mediante MoveIt. Nello specifico la funzione non è stata introdotta da noi ma era

già utilizzata nel repository GitHub [6] che è stato il punto di partenza di tale progetto. Si anticipa però già qui che ci sono però stati dei problemi di unità di misura nell'utilizzo di tale funzione che sono stati soltanto parzialmente risolti (Sezione 7.3). Non è infatti stato possibile reperire sufficienti informazioni sul significato degli argomenti della *PVTPoint()* dalla documentazione ufficiale.

Capitolo 2

ROS

ROS (Robot Operating System) è un framework open-source per la programmazione di robot. Non è quindi un vero e proprio sistema operativo anche se svolge alcune delle sue funzioni quali astrazione dell'hardware, gestione dei package, controllo dei dispositivi tramite driver e comunicazione tra processi. Oltre a questo fornisce anche strumenti e librerie per facilitare la programmazione di applicazioni robotiche. Dietro a ROS c'è infatti una vasta community grazie alla quale il framework è in continua espansione.

ROS è pensato per operare in simbiosi con sistemi operativi Linux, in particolare il principale sistema operativo su cui ne viene portato avanti lo sviluppo è Ubuntu. Ogni anno infatti viene pubblicata una nuova versione di ROS che segue il rilascio della nuova versione di Ubuntu. In Figura 2.1 sono riportate le ultime versioni di ROS o meglio di ROS1.

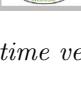
Distro	Release date	Poster	Turtle, turtle in tutorial	EOL date
ROS Noetic NixieMys (Recommended)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017
ROS Indigo Igloo	July 22nd, 2014			April, 2019 (Trusty EOL)
ROS Hydro Medusa	September 4th, 2013			May, 2015

Figure 2.1: Ultime versioni di ROS1

Si può infatti notare che la tabella si interrompe con ROS Noetic, rilasciata nel 2020 e supportata fino al 2025. Questo perché nel corso degli anni gli sviluppatori di ROS si sono resi conto che il framework richiedeva delle modifiche sostanziali che lo avrebbero reso troppo instabile. Per questo motivo a partire dal 2015 è stata sviluppata una versione completamente nuova del framework che è ROS2 e che è destinata a soppiantare la versione originaria rinominata quindi ROS1. Le versioni di ROS2 finora rilasciate sono riportate in Figura 2.2.

Distro	Release date	Logo	EOL date
Humble Hawksbill	May 23rd, 2022		
Galactic Geochelone	May 23rd, 2021		November 2022
Foxy Fitzroy	June 5th, 2020		May 2023
Eloquent Elusor	November 22nd, 2019		November 2020
Dashing Diademata	May 31st, 2019		May 2021
Crystal Clemmys	December 14th, 2018		December 2019
Bouncy Bolson	July 2nd, 2018		July 2019
Ardent Apalone	December 8th, 2017		December 2018
beta3	September 13th, 2017		December 2017
beta2	July 5th, 2017		September 2017
beta1	December 19th, 2016		Jul 2017
alpha1 - alpha8	August 31th, 2015		December 2016

Figure 2.2: Versioni di ROS2

Al momento della scrittura della presente relazione si è quindi ancora in una fase di passaggio tra ROS1 e ROS2 in cui ROS2 è già utilizzabile ma non supporta ancora tutte le funzionalità implementate in ROS1.

Per questo motivo si è scelto di rimanere nell'ambito di ROS1 (Sezione 7.4), utilizzando nello specifico la versione Melodic Morenia di ROS1 che nasce per Ubuntu 18.04 (Bionic).

2.1 Concetti fondamentali e comandi

Una delle caratteristiche principali di ROS è l'architettura distribuita. Un'applicazione sviluppata con tale framework non è altro che un insieme di moduli software, detti nodi, interconnessi tra loro.

Ogni nodo è un processo a sé stante che svolge uno specifico compito. I nodi possono andare

in esecuzione sulla stessa macchina ma anche su macchine diverse. In entrambi i casi è ROS a gestire la comunicazione tra i nodi che è quindi totalmente trasparente al programmatore. I nodi di un'applicazione ROS possono scambiarsi messaggi attraverso dei canali di comunicazione detti topic. Possono anche offrire e/o richiedere servizi agli altri nodi realizzando un'architettura di tipo client-server. Infine possono anche accedere in lettura e/o scrittura ai dati contenuti in un database comune detto server dei parametri.

Di seguito vengono descritti brevemente i componenti fondamentali di un'applicazione ROS che è necessario conoscere per la comprensione del progetto. Per maggiori dettagli sul framework il riferimento è la guida online di ROS1 [7].

2.1.1 Nodo master

Alla base di ogni applicazione ROS c'è un particolare nodo detto master che si occupa di gestire tutti gli altri nodi permettendone l'esecuzione parallela e la comunicazione. Tale nodo è unico e deve essere il primo ad essere mandato in esecuzione e l'ultimo a terminare. Nel Listato 2.1 è riportato il comando da terminale che permette di mandare in esecuzione il nodo master (oltre al server dei parametri e allo standard output).

```
$ roscore
```

Listing 2.1: Comando per lanciare il nodo master

Se si utilizzano i file di launch non è però necessario ricorrere a tale comando in quanto il master è automaticamente mandato in esecuzione prima degli altri nodi.

2.1.2 Nodi

I nodi sono al centro della programmazione in ROS. Ogni nodo è infatti un programma in esecuzione che viene registrato dal nodo master. I nodi possono essere scritti in C++ o in Python che sono i 2 linguaggi di programmazione supportati da ROS.

Il comando da terminale che permette di mandare in esecuzione un nodo è riportato nel Listato 2.2. Si ricorda che tale comando può essere utilizzato soltanto dopo aver mandato in esecuzione il nodo master.

```
$ rosrun package_name executable_name
```

Listing 2.2: Comando per lanciare un nodo

Questo comando si limita a mandare in esecuzione l'eseguibile *executable_name* del pacchetto *package_name*. La creazione del nodo ROS e la sua registrazione sul master con il proprio nome avviene poi all'interno del programma con le istruzioni riportate nel Listato 2.3 per i nodi scritti in C++.

```
1 int main(int argc, char **argv){
2     //Creazione del nodo
3     ros::init(argc, argv, "name_node");
4     ros::NodeHandle nh;
5 }
```

Listing 2.3: Istruzioni per creare un nodo in C++

Si fa notare che non è necessario che il nome del nodo assegnato nella `init()` coincida con il nome dell'eseguibile. È infatti anche possibile lanciare più nodi che eseguono il medesimo programma assegnandogli nomi diversi.

Nel Listato 2.4 è riportato un utile comando che permette di ottenere la lista dei nodi attualmente in esecuzione. Si fa notare che nella lista che si ottiene è sempre presente il nodo `\rosout`. Tale nodo è infatti avviato automaticamente insieme al nodo master ed è l'equivalente ROS dello standard output.

```
$ rosnode list
```

Listing 2.4: Comando per visualizzare i nodi in esecuzione

2.1.3 Topic e messaggi

Il principale meccanismo utilizzato dai nodi ROS per comunicare è lo scambio di messaggi. Si tratta di una modalità di comunicazione asincrona che avviene attraverso dei canali di comunicazione detti topic.

Lo schema di funzionamento è riportato in Figura 2.3.

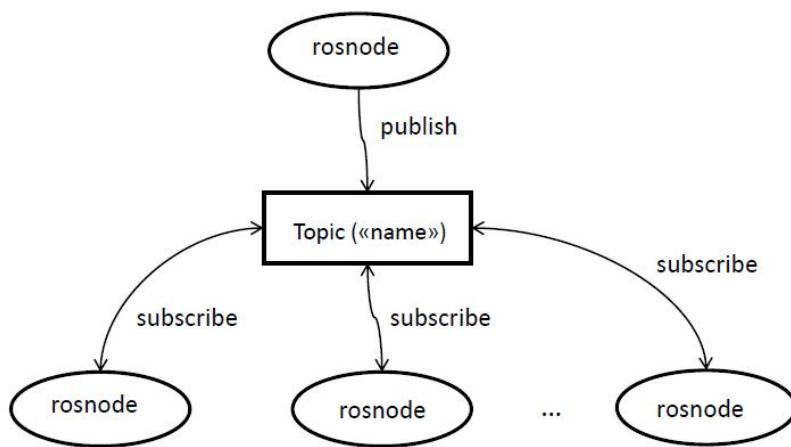


Figure 2.3: Schema di funzionamento dello scambio di messaggi

Il nodo che deve condividere delle informazioni pubblica dei messaggi su un certo topic, divenendo un publisher. Il nodo che vuole ricevere tali informazioni deve invece iscriversi al medesimo topic, divenendone un subscriber. La comunicazione viene gestita dal nodo master che si assicura che i nodi publisher e i nodi subscriber dello stesso topic possano trovarsi.

Ogni topic deve avere un nome univoco per poter essere identificato. Inoltre ogni topic permette lo scambio soltanto di una certa tipologia di messaggio che va dichiarato al momento della creazione del topic. La tipologia di messaggio associata al topic può essere uno dei tipi predefiniti di messaggio di ROS o essere definita dall'utente con un file msg.

Un file msg è un semplice file di testo che viene utilizzato per generare automaticamente il codice sorgente del messaggio nei diversi linguaggi. Nel file vengono specificati i campi del messaggio ROS, riportando per ciascuno il tipo di dato del campo e il nome. Tra i tipi di

dato disponibili ci sono ad esempio bool, string, int8, float32, Header e altri file msg.
Un esempio di file msg è riportato nel Listato 2.5.

```
# ScrResponse.msg
Header header
string id
string script
```

Listing 2.5: Esempio file msg

Un utile comando da terminale per visualizzare graficamente le relazioni publisher/subscriber tra i nodi dell'applicazione ROS è riportato nel Listato 2.6.

```
$ rqt_graph
```

Listing 2.6: Comando rqt_graph

Un altro utile strumento utilizzato nel presente progetto è il tool da linea di comando *rostopic* che mette a disposizione diversi comandi per ottenere informazioni sui topic. Alcuni di questi comandi sono riportati nel Listato 2.7.

```
# Comando per stampare la lista dei topic
$ rostopic list
# Comando per stampare i messaggi che transitano su un certo topic
$ rostopic echo /topic_name
```

Listing 2.7: Comandi rostopic

Per quanto riguarda infine l'implementazione pratica dello scambio di messaggi è opportuno descrivere come creare un nodo publisher/subscriber in C++.

Nel Listato 2.8 è riportato e commentato il codice di un nodo che pubblica un singolo messaggio su un topic.

```
1 int main(int argc, char **argv) {
2     //Creazione del nodo
3     ros::init(argc, argv, "nome_nodo");
4     ros::NodeHandle nh;
5     // Registrazione del nodo come publisher di un topic
6     // occorre specificare il nome univoco del topic,
7     // il tipo di messaggio che vi transita,
8     // il numero massimo di messaggi in attesa di essere pubblicati
9     ros::Publisher pub = nh.advertise<tipo_msg>("nome_topic", lunghezza_coda);
10    // Attesa finche' il publisher non e' pronto
11    //((altrimenti il messaggio e' perso)
12    while (pub.getNumSubscribers() < 1) {
13        ros::WallDuration sleep_t(0.5); //secondi
14        sleep_t.sleep();
15    }
16    //Creazione del messaggio da inviare
17    tipo_msg msg;
18    msg.campo = valore;
19    //Pubblicazione del messaggio
20    pub.publish(msg);
21    // Attesa per dare tempo al nodo di inviare il messaggio
22    //((altrimenti il messaggio e' perso)
23    ros::Duration(1).sleep(); //secondi
24    return 0; }
```

Listing 2.8: Nodo publisher in C++

Si fa notare che nel codice ci sono delle istruzioni di attesa che in generale non sono necessarie ma che lo diventano se il nodo si limita a pubblicare un messaggio prima di terminare. In particolare la seconda istruzione di attesa si comprende facendo riferimento allo schema in Figura 2.4.

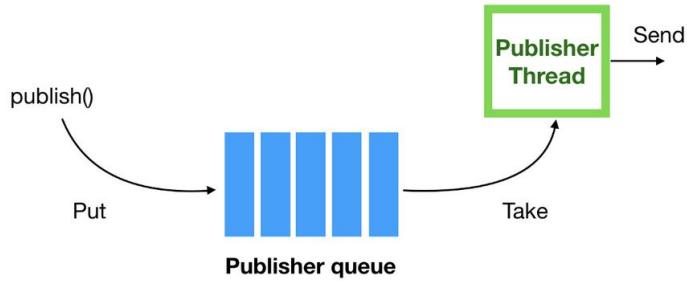


Figure 2.4: Schema pubblicazione messaggi

La funzione *publish()* non è direttamente responsabile dell'invio del messaggio ma si limita ad aggiungere il nuovo messaggio alla coda dei messaggi da pubblicare. C'è poi un diverso thread (*Publisher Thread*) che si occupa di prendere i messaggi dalla coda e inviarli. Da qui si comprende anche il significato del parametro *lunghezza_coda* (riga 9 del Listato 2.8): se la funzione *publish()* è chiamata più velocemente di quanto il thread riesca ad inviare i messaggi la coda si riempie fino a raggiungere la lunghezza massima oltre la quale i messaggi vengono sovrascritti.

Nel Listato 2.9 è invece riportato il codice di un nodo che si mette in ascolto di un certo topic.

```

1 //Funzione richiamata quando un messaggio e' pubblicato sul topic
2 void fCallback(const tipo_msg::ConstPtr& msg) {
3     ...
4 int main(int argc, char **argv){
5     //Creazione del nodo
6     ros::init(argc, argv, "nome_nodo");
7     ros::NodeHandle nh;
8     // Registrazione del nodo come subscriber di un topic
9     // Occorre specificare il nome univoco del topic,
10    // il numero massimo di messaggi in attesa di lettura,
11    // la funzione di callback
12    ros::Subscriber sub = nh.subscribe<tipo_msg>("nome_topic", lunghezza_coda,
13        fCallback);
14    // Mantiene il nodo attivo per eseguire le callback
15    ros::spin();
16    return 0;}

```

Listing 2.9: Nodo subscriber in C++ con spin()

Da evidenziare è il meccanismo con cui sono gestite le funzioni di callback, schematizzato in Figura 2.5.

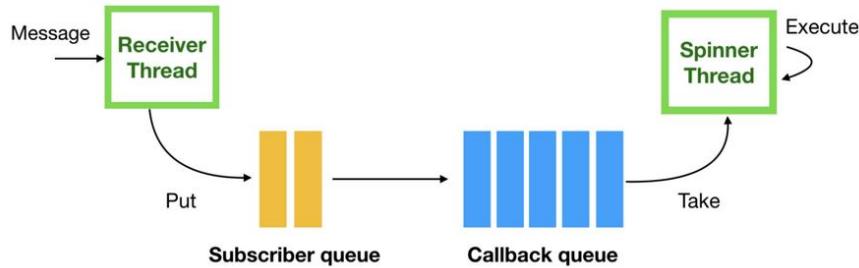


Figure 2.5: Schema ricezione messaggi

Ogni nodo ha un thread (*Receiver Thread*) che si occupa di ricevere i messaggi e metterli nella coda dei messaggi ricevuti (*Subscriber queue*). Quando un nuovo messaggio arriva la corrispondente funzione di callback viene inserita in un'altra coda (*Callback queue*). Anche in questo caso c'è un parametro *lunghezza_coda* (riga 13 Listato 2.9) che rappresenta il numero massimo di messaggi ricevuti ma ancora in attesa di elaborazione che possono accumularsi prima di essere sovrascritti. Tutto questo meccanismo viene gestito automaticamente da ROS in background. C'è poi un altro thread (*Spinner thread*) che si occupa invece di prelevare le funzioni dalla coda delle callback e di eseguirle. Questo thread va in esecuzione solo quando è esplicitamente richiesto dal codice del nodo utilizzando una delle seguenti funzioni:

- *spin()*: manda in esecuzione lo *Spinner thread* che rimane attivo fino alla terminazione del nodo. Il nodo non può eseguire altro codice ma rimane in attesa di nuovi messaggi per lanciare le corrispondenti funzioni di callback al loro arrivo;
- *spinOnce()*: manda temporaneamente in esecuzione lo *Spinner thread* per eseguire le funzioni di callback pendenti. Una volta svuotata la *Callback queue* il controllo è restituito al nodo;

Nell'esempio del Listato 2.9 è stata utilizzata la funzione *spin()* in quanto l'unico compito del nodo è quello di ricevere i messaggi pubblicati sul topic che sottoscrive. Se invece il nodo deve svolgere anche altre funzioni occorre utilizzare la *spinOnce()* inserendola all'interno di un loop (Listato 2.10). In quest'ultimo caso occorre fare attenzione a dimensionare correttamente la *Callback queue* in base al rapporto tra la frequenza di arrivo dei messaggi e la frequenza di esecuzione delle callback.

```

1 void fCallback(const tipo_msg::ConstPtr& msg) {
2     ...
3 int main(int argc, char **argv){
4     ros::init(argc, argv, "nome_nodo");
5     ros::NodeHandle nh;
6
7     ros::Subscriber sub = nh.subscribe<tipo_msg>("nome_topic", lunghezza_coda,
8         fCallback);
9     //Gestione loop a 5 Hz
10    ros::Rate loop_rate(5); //Hz
11    while (ros::ok()){
12        //Altro codice del nodo
13        ...
14        //Spinner thread
  
```

```

14     ros::spinOnce();
15     //Attesa per avere loop a 5 Hz
16     loop_rate.sleep();
17 }
18 return 0;

```

Listing 2.10: Nodo subscriber in C++ con spinOnce()

Nel presente progetto il meccanismo di scambio di messaggi è stato utilizzato per notificare la rilevazione di un ostacolo al nodo che gestisce la conseguente modifica della traiettoria del robot.

2.1.4 Servizi

Un'altra tipologia di comunicazione tra nodi è quella che avviene attraverso i servizi che permettono di implementare un'architettura di tipo client-server. Lo schema di funzionamento è riportato in Figura 2.6.

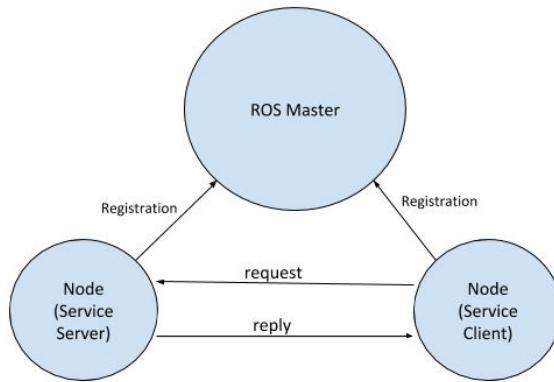


Figure 2.6: Schema di funzionamento dei servizi

Si tratta di un meccanismo di comunicazione bidirezionale che estende quello dei messaggi offrendo ai nodi la possibilità di inviare comandi e di ricevere una risposta al termine dell'esecuzione del comando richiesto. L'idea è quella di avere un nodo server in grado di offrire un certo servizio e un nodo client che richiede tale servizio. Il nodo client fa una richiesta di servizio attraverso un messaggio *Request* e rimane in attesa di ricevere un messaggio *Response* dal nodo server al termine dell'esecuzione del servizio richiesto.

Ad ogni servizio sono quindi associati 2 messaggi che vengono descritti in un file srv che definisce il tipo di servizio del tutto analogo al file msg per la definizione dei messaggi. Si tratta infatti di un file di testo in cui sono specificati tipo di dato e nome dei singoli campi del messaggio di *Request* e del messaggio di *Response*. Da tale file viene generato automaticamente il codice sorgente del servizio nei diversi linguaggi. Nel Listato 2.11 è riportato un esempio di file srv.

```

# SendScript.srv
# Request msg
string id
string script
---
```

```
# Response msg
bool ok
```

Listing 2.11: Esempio file srv

Ad ogni servizio deve poi essere associata una funzione che il nodo server manda in esecuzione quando riceve un messaggio di *Request* dal nodo client. Nel Listato 2.12 è riportato il generico prototipo di tale funzione.

```
bool fservice(Request_msg &req, Response_msg &res)
```

Listing 2.12: Prototipo funzione eseguita dal servizio

Ogni funzione di callback di un servizio prende come input un puntatore al messaggio di *Request* e un puntatore al messaggio di *Response* e restituisce un valore booleano che indica se la chiamata al servizio ha avuto o meno successo.

Per quanto riguarda l'implementazione pratica dei servizi è opportuno descrivere come creare un nodo client e un nodo server in C++.

Nel Listato 2.13 è riportato e commentato il codice di un nodo che offre un servizio.

```
1 bool serviceCallback(Request_msg &req, Response_msg &res){
2     //Codice del servizio
3     //che usa Request e valorizza Response
4     ...
5     return true;}
6 int main(int argc, char **argv){
7     //Creazione del nodo
8     ros::init(argc, argv, "nome_nodo");
9     ros::NodeHandle nh;
10    //Registrazione del servizio
11    ros::ServiceServer service = nh.advertiseService("nome_servizio",
12        serviceCallback);
13    //Mantiene il nodo attivo per offrire il servizio
14    ros::spin();
15    return 0;}
```

Listing 2.13: Nodo server in C++

Nel Listato 2.14 è invece riportato il codice di un nodo che richiede il servizio offerto dal precedente nodo.

```
1 int main(int argc, char **argv){
2     //Creazione del nodo
3     ros::init(argc, argv, "nome_nodo");
4     ros::NodeHandle nh;
5     // Creazione del client per richiedere il servizio
6     ros::ServiceClient client = nh.serviceClient<tipo_srv>("nome_servizio");
7     //Creazione del msg Request
8     tipo_srv srv;
9     srv.request.campo = valore;
10    //Richiesta del servizio
11    if (client.call(srv)) { //Codice se il servizio e' stato erogato
12        correttamente
13        ...
14    } else {
15        //Codice se il servizio non e' stato erogato correttamente
16        ...}
```

```

16     return 1;
17 ...

```

Listing 2.14: Nodo client in C++

Si fa notare che la richiesta del servizio vera e propria effettuata con la funzione `call()` è bloccante e restituisce il controllo al nodo soltanto al termine della callback del servizio invocato. La funzione restituisce l'esito della richiesta: *true* se il servizio è stato erogato correttamente e il client può quindi utilizzare il messaggio di Response, *false* altrimenti.

Nel progetto i servizi sono stati utilizzati per inviare comandi al robot in modalità *External Script*.

2.1.5 Azioni

Un'ultima tipologia di comunicazione tra nodi da approfondire è quella che utilizza le azioni. In sintesi le azioni sono semplicemente un'evoluzione dei servizi che permettono di implementare un'interazione client-server asincrona invece che sincrona. Lo schema di funzionamento è riportato in Figura 2.7.

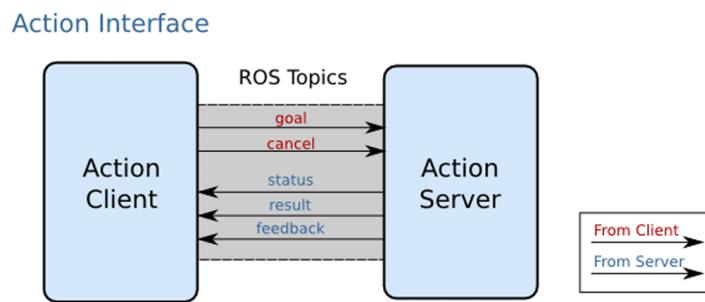


Figure 2.7: Schema di funzionamento delle azioni

Esattamente come nei servizi c'è un nodo server (*Action Server*) che offre un servizio e un nodo client (*Action Client*) che lo richiede che comunicano scambiandosi dei messaggi. Azioni e servizi sono quindi semplicemente due diversi protocolli di più alto livello che specificano come un insieme di topic devo essere utilizzato. La differenza è che le azioni sono pensate per gestire servizi che richiedono molto tempo per essere eseguiti ed offrono quindi al client la possibilità di cancellare la richiesta durante la sua esecuzione e di ottenere un feedback periodico sullo stato di esecuzione del servizio.

Ad ogni azione è associato un file .action del tutto simile al file .srv utilizzato per i servizi. In questo file di testo è specificata infatti la struttura dei 3 messaggi che caratterizzano un'azione:

- *Goal*: messaggio inviato dal client al server per richiedere un certo servizio, è del tutto analogo al messaggio *Request* dei servizi;
- *Result*: messaggio inviato dal server al client al termine dell'esecuzione del servizio, è del tutto analogo al messaggio *Response* dei servizi;
- *Feedback*: messaggio inviato periodicamente dal server al client durante l'esecuzione del servizio per aggiornarlo sui progressi del goal richiesto;

Nel Listato 2.15 è riportata la struttura di un generico file .action.

```
# Goal msg
Tipo_campo nome_campo
---
# Result msg
Tipo_campo nome_campo
---
# Feedback msg
Tipo_campo nome_campo
```

Listing 2.15: Struttura file action

Ci sono poi altri 2 messaggi necessari per implementare un’azione ma standard e quindi non definiti nel file .action:

- *Cancel*: messaggio inviato dal client al server per richiedere la cancellazione del goal;
- *Status*: messaggio inviato dal server al client con lo stato di tutti i goal che il server sta gestendo.

Per ulteriori informazioni sulle azioni e sulla loro implementazione il riferimento è sempre la guida online di ROS1 [7].

Per il presente progetto non è stato necessario gestire direttamente le azioni. È però opportuno conoscerne il meccanismo base in quanto vengono utilizzate nel repository GitHub [6] per la pianificazione della traiettoria da MoveIt.

2.1.6 File di launch

Un ultimo concetto base di ROS da approfondire è quello dei file di launch. Si tratta di un utile strumento che permette di mandare in esecuzione contemporaneamente il nodo master ed altri nodi con un unico comando da terminale. Senza tale strumento mandare in esecuzione un’applicazione ROS sarebbe piuttosto macchinoso dal momento che ogni nodo, compreso il master, dovrebbe essere lanciato singolarmente da un diverso terminale.

Nel Listato 3.1.1 è riportato il comando che permette di lanciare un file di launch.

```
$ roslaunch nome_package nome_file.launch
```

Listing 2.16: Comando per lanciare un file di launch

Nella pratica un file di launch non è altro che un file XML che descrive attraverso una serie di tag cosa e come deve essere mandato in esecuzione. Nel Listato 2.17 è riportato un esempio molto semplice di file di launch che manda in esecuzione 2 nodi e definisce un parametro.

```
<launch>
  <node name="nome_nodo" pkg="nome_pacchetto" type="nome_tipo_nodo" output="screen">
    <param name="nome_param" type="tipo_param" value="valore_param" />
  </node>
  <node name="nome_nodo" pkg="nome_pacchetto" type="nome_tipo_nodo" respawn="false" output="screen" />
  ...
</launch>
```

Listing 2.17: Esempio di file di launch

Per maggiori informazioni sui tag che possono essere utilizzati nei file di launch il riferimento è sempre [7].

2.2 RViz e MoveIt

RViz è il visualizzatore 3D di ROS che permette di visualizzare all'interno di uno spazio tridimensionale il modello del robot, l'ambiente in cui opera e i dati provenienti dai sensori. Questo tool offre inoltre un'interfaccia grafica semplice e intuitiva per selezionare cosa rendere visibile e analizzabile.

MoveIt è invece un plugin di RViz per la pianificazione della traiettoria e la sua esecuzione su un robot reale o simulato.

Per il presente progetto non sono state approfondite tutte le funzionalità offerte da MoveIt ma ci si è limitati ad utilizzarlo per pianificare la traiettoria da eseguire sul robot reale senza modificarne le impostazioni. Di seguito viene quindi descritto soltanto come accedere alle principali funzioni dell'interfaccia grafica. Si fa però notare che il plugin offre anche altre opzioni quali poter modificare il tipo di pianificatore o gestire la presenza di ostacoli nell'ambiente.

In Figura 2.8 è riportata l'interfaccia di RViz quando MoveIt è abilitato.

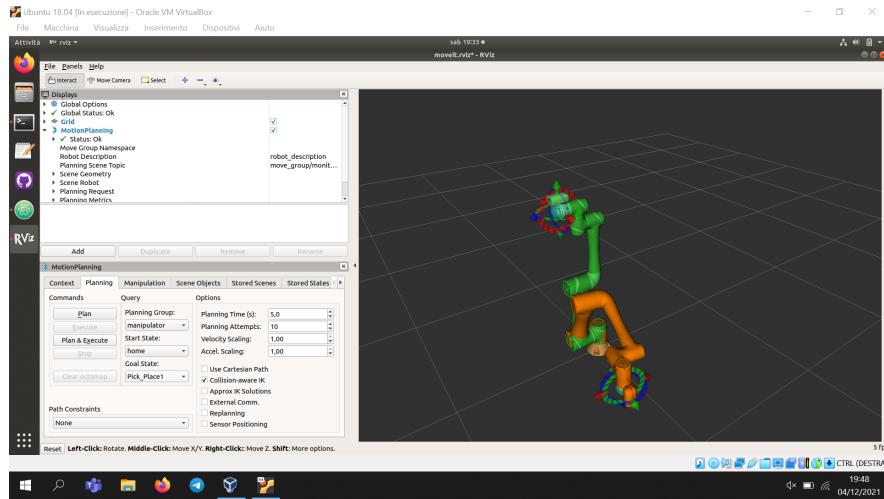


Figure 2.8: Interfaccia di MoveIt

L'interfaccia offre una sezione *Displays* che permette all'utente di selezionare ciò che RViz deve visualizzare. Nell'ambito della pianificazione del moto è possibile ad esempio scegliere se e come visualizzare stato iniziale, stato finale e traiettoria pianificata.

C'è poi una sezione *Motion Planning* che è interamente dedicata a MoveIt. Da qui, accedendo alla sottosezione *Planning*, è possibile richiedere la pianificazione e/o l'esecuzione della traiettoria con i pulsanti *Plan*, *Execute* e *Plan & Execute*. La traiettoria pianificata viene visualizzata da RViz sulla finestra di visualizzazione. Dalla stessa sottosezione è anche possibile selezionare stato iniziale e stato finale del robot da un menù a tendina. È però anche

possibile spostare manualmente lo stato iniziale/finale del robot agendo direttamente sulla finestra di visualizzazione.

2.3 Gazebo

Gazebo è un simulatore 3D open-source per la robotica. Si tratta di un software stand-alone il cui corretto funzionamento è garantito soltanto sotto Linux. Gazebo è anche perfettamente integrato con l'ambiente ROS di cui è il simulatore di riferimento.

Tale simulatore non è però stato approfondito nel presente progetto. Ci si limita quindi a riportare in Figura 2.9 la sua interfaccia.

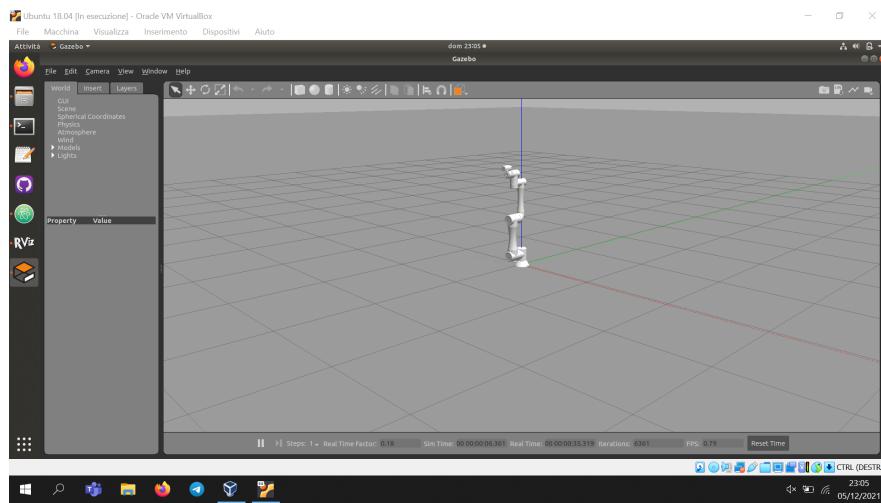


Figure 2.9: *Interfaccia di Gazebo*

Capitolo 3

Driver ROS

Il punto di partenza di tale progetto è il repository GitHub riportato in [6]. Si tratta di un repository ancora in fase di sviluppo e gestito direttamente dalla Techman Robot per l’interfacciamento dei robot della serie TM con ROS1. Nello specifico le versioni di ROS1 supportate sono ROS Melodic (master) e ROS Noetic (branch). Si fa inoltre notare che è in fase di sviluppo un progetto analogo per ROS2 in [8]. Per il presente progetto si è però fatto riferimento soltanto al driver per ROS Melodic in quanto allo stato attuale risulta essere il più completo.

In sintesi questo repository implementa mediante un nodo ROS un vero e proprio driver che gestisce la comunicazione mediante cavo Ethernet tra il robot controllato da TMflow e un computer esterno dove è in esecuzione un’applicazione ROS. Nello specifico il nodo driver implementa i protocolli di comunicazione per la connessione con i socket TCP *Ethernet Slave* e *Listen* che permettono all’utente di ottenere informazioni sullo stato del robot e di controlarlo in modalità *External Script* (Capitolo 1.2.2).

Di seguito sono descritte le funzionalità che tale driver offre e come accedervi, concentrando l’attenzione su quelle che sono state poi sfruttate nella modifica del repository per la gestione degli ostacoli.

3.1 Utilizzo driver ROS

Per capire le funzionalità offerte dal repository il punto di partenza sono i file di launch e soprattutto le demo descritte nel README del progetto GitHub [6]. I file di launch sono delle vere e proprie applicazioni ROS che l’utente può mandare in esecuzione per controllare il robot senza dover apportare alcuna modifica al codice. Le demo sono invece dei semplici nodi ROS che mostrano all’utente come utilizzare le singole funzionalità offerte dal nodo driver. Per testarle occorre quindi manualmente mandare in esecuzione il nodo demo di interesse insieme al nodo driver e al master.

In entrambi i casi per effettuare tali test occorre però a monte configurare l’intero sistema seguendo quanto descritto nel Capitolo 4.

3.1.1 File di launch

I file di launch che il repository mette a disposizione permettono di pianificare una traiettoria con MoveIt e di eseguirla su un robot della serie TM reale o simulato. Nel presente progetto non è però stata approfondita la parte di simulazione che è stata soltanto vista parzialmente e da un diverso repository GitHub (Appendice A). Nello specifico è stato esaminato soltanto il file di launch per il controllo del robot reale TM5-900 tramite MoveIt che può essere mandato in esecuzione con il comando da terminale riportato nel Listato 3.1.

```
$ rosrun tm5_900_moveit_config tm5_900_moveit_planning_execution.launch
  sim:=False robot_ip:=<robot_ip_address>
```

Listing 3.1: Comando per lanciare il file di launch per controllare il TM5-900 con MoveIt

Questo file di launch richiede di specificare i seguenti 2 argomenti:

- *sim*: argomento booleano per specificare se si vuole operare sul robot reale o simulato (si fa però notare che il file di launch per la simulazione non è attualmente implementato completamente);
- *robot_ip*: argomento richiesto solo per il controllo del robot reale (quindi quando l'argomento *sim* ha valore *false*), il suo valore deve essere l'indirizzo IP del robot che si vuole controllare (Capitolo 4).

Lanciando questo comando si apre automaticamente la schermata di RViz con MoveIt abilitato mostrata in Figura 3.1 (Capitolo 2.2).

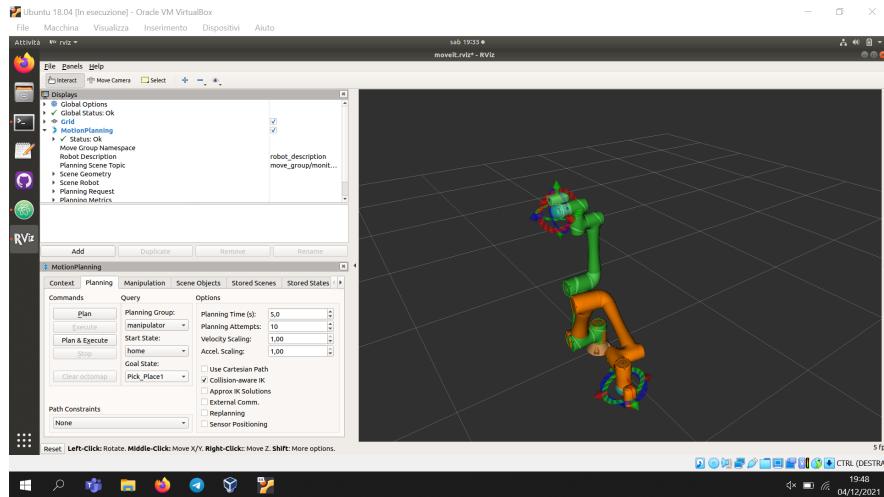


Figure 3.1: Interfaccia di MoveIt

Nella finestra di visualizzazione viene mostrata la posa corrente del manipolatore (in grigio), lo stato di partenza (in verde) e lo stato goal (in arancione). Se lo stato di partenza e/o lo stato goal non vengono mostrati occorre abilitarne la visualizzazione dalla sezione *Displays*, spuntando rispettivamente i campi *Query Start State* e *Query Goal State* della sottosezione *Planning Request* mostrati in Figura 3.2.

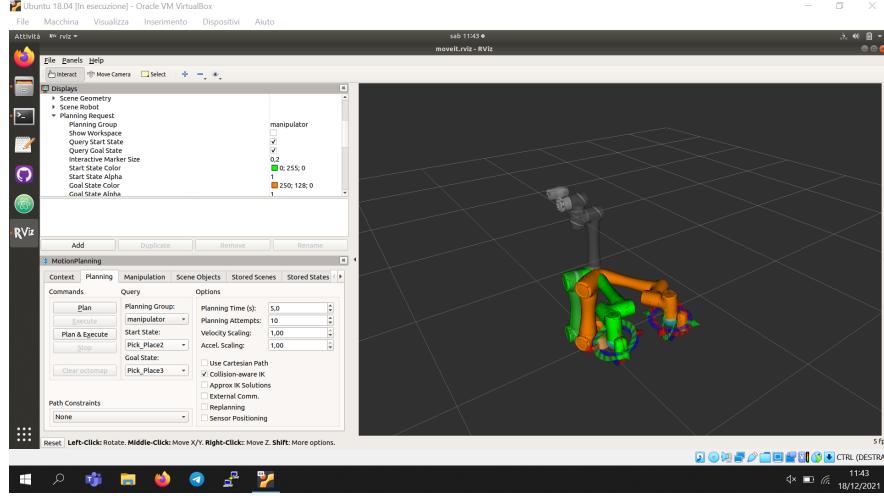


Figure 3.2: Interfaccia di MoveIt: visualizzazione stati

La pianificazione e l'esecuzione della traiettoria desiderata può essere fatta dalla sezione *MotionPlanning* e in particolare dalla sottosezione *Planning*. Da qui l'utente deve selezionare lo stato goal desiderato scegliendolo da un menù a tendina. Gli stati selezionabili sono definiti nel file *NOME_PROGETTO/tm5_900_moveit_config/config/tm5_900.srdf* in formato XML specificando il nome dello stato e il valore di ogni variabile di giunto in radianti. Per aggiungere nuovi stati è quindi sufficiente modificare tale file come mostrato in Figura 3.3.

```

<?xml version="1.0" encoding="UTF-8"?>
<robot name="tm5_900">
  <!-- STATE GOALS -->
  <group_state name="J1_90" group="manipulator">
    <joint name="elbow_joint" value="0" />
    <joint name="shoulder_1_joint" value="1.5708" />
    <joint name="wrist_1_joint" value="0" />
    <joint name="wrist_2_joint" value="0" />
    <joint name="wrist_3_joint" value="0" />
  </group_state>
  <group_state name="J3_90" group="manipulator">
    <joint name="elbow_joint" value="1.5708" />
    <joint name="shoulder_1_joint" value="0" />
    <joint name="wrist_1_joint" value="0" />
    <joint name="wrist_2_joint" value="0" />
    <joint name="wrist_3_joint" value="0" />
  </group_state>
  <group_state name="J1_180" group="manipulator">
    <joint name="elbow_joint" value="1.5708" />
    <joint name="shoulder_1_joint" value="0.1899" />
    <joint name="shoulder_2_joint" value="1.5708" />
    <joint name="wrist_1_joint" value="0.3339" />
    <joint name="wrist_2_joint" value="1.6129" />
    <joint name="wrist_3_joint" value="0" />
  </group_state>
  <group_state name="Pick Place1" group="manipulator">
    <joint name="elbow_joint" value="1.5125" />
    <joint name="shoulder_1_joint" value="0.1899" />
    <joint name="shoulder_2_joint" value="1.5708" />
    <joint name="wrist_1_joint" value="0.3339" />
    <joint name="wrist_2_joint" value="1.6129" />
    <joint name="wrist_3_joint" value="0" />
  </group_state>
  <group_state name="Pick Place2" group="manipulator">
    <joint name="elbow_joint" value="1.5125" />
  </group_state>
</robot>

```

Figure 3.3: File tm5_900.srdf

In alternativa l'utente può anche spostare manualmente lo stato finale dalla finestra di visualizzazione. Lo stato iniziale va invece necessariamente mantenuto al valore di default *current* per non avere errori in fase di esecuzione.

Dalla stessa sottosezione l'utente può richiedere la pianificazione della traiettoria e/o l'esecuzione della traiettoria pianificata con i pulsanti *Plan*, *Execute* e *Plan & Execute*. In particolare con il pulsante *Plan* MoveIt pianifica la traiettoria dallo stato iniziale allo stato

goal selezionato e la mostra con un'animazione in loop nella finestra di visualizzazione. Con il pulsante *Execute* invece la traiettoria pianificata viene fatta eseguire dal robot reale sfruttando la modalità *External Script*.

Infine si fa notare che lo stato corrente del robot è aggiornato in tempo reale sulla finestra di visualizzazione sfruttando il socket *Ethernet Slave*. Questi aspetti implementativi sono però approfonditi nella Sezione 3.2.

3.1.2 Demo

Il repository GitHub [6] mette a disposizione diverse demo che mostrano come poter utilizzare le funzionalità fornite dal nodo driver per comunicare con il robot sulla base dei protocolli descritti nella Sezione 1.2.2. Come già anticipato per il presente progetto sono stati utilizzati soltanto il protocollo TMSCT per controllare il robot in modalità *External Script* e il protocollo TMSTA con *SubCmd 01* per gestire traiettorie da eseguire in modo ciclico. Per fare questo sono state quindi approfondite soltanto le seguenti 2 demo:

- *NOME_PROGETTO/demo/src/demo_send_script.cpp*: codice sorgente di un nodo ROS in C++ che invia un comando al robot in modalità *External Script* sfruttando il servizio SendScript offerto dal nodo driver (Sezione 3.2.1);
- *NOME_PROGETTO/demo/get_status_demo_src/demo_get_sta_response.cpp*: codice sorgente di un nodo ROS in C++ che si mette in ascolto dei messaggi di tipo StaResponse pubblicati dal nodo driver.

Nel Listato 3.2 e nel Listato 3.3 sono riportati e commentati i codici sorgenti dei 2 nodi.

```

1 // Header file standard
2 #include <ros/ros.h>
3 #include <std_msgs/String.h>
4 #include <iostream>
5 #include <cstdlib>
6 // Header file per utilizzare il servizio SendScript
7 #include "tm_msgs/SendScript.h"
8
9 int main(int argc, char **argv){
10     //Comando da inviare al robot
11     std::string cmd = "PTP(\"JPP\",0,0,90,0,90,0,35,200,0,false)";
12     //Creazione del nodo
13     ros::init(argc, argv, "demo_send_script");
14     ros::NodeHandle nh_demo;
15     //Creazione del client per richiedere il servizio SendScript
16     ros::ServiceClient client = nh_demo.serviceClient<tm_msgs::SendScript>(
17         "tm_driver/send_script");
18     //Creazione del msg Request
19     tm_msgs::SendScript srv;
20     srv.request.id = "demo";
21     srv.request.script = cmd;
22     //Richiesta del servizio SendScript
23     if (client.call(srv)){
24         //se il servizio e' stato correttamente erogato
25         //Verifica risposta del robot sulla correttezza del comando richiesto
26         if (srv.response.ok) ROS_INFO_STREAM("Sent script to robot");
27         else ROS_WARN_STREAM("Sent script to robot , but response not yet ok ");
28 }
```

```

27 }else{
28 //Se il servizio non e' stato erogato correttamente
29 ROS_ERROR_STREAM("Error send script to robot");
30 return 1;
31 return 0;

```

Listing 3.2: Sorgente del nodo *demo-send-script*

```

1 // Header file standard
2 #include <ros/ros.h>
3 #include <std_msgs/String.h>
4 // Header file per messaggi di tipo StaResponse
5 #include "tm_msgs/StaResponse.h"
6
7 //Funzione di callback quando un messaggio StaResponse
8 // e' pubblicato sul topic
9 void StaResponseCallback(const tm_msgs::StaResponse::ConstPtr& msg){
10 //Stampa sul terminale del messaggio ricevuto
11 ROS_INFO_STREAM("StaResponse: subcmd is = " << msg->subcmd << ", subdata is "
12 << msg->subdata);}
13
14 int main(int argc, char **argv){
15 //Creazione del nodo
16 ros::init(argc, argv, "StaResponse");
17 ros::NodeHandle nh_demo_get;
18 ros::Subscriber sub = nh_demo_get.subscribe("tm_driver/sta_response", 1000,
19 StaResponseCallback);
20 //Mantiene il nodo attivo per eseguire la callback
21 ros::spin();
22 return 0;

```

Listing 3.3: Sorgente del nodo *demo-get-sta-response*

3.2 Nodo driver

Il nodo driver è il componente alla base dell'applicazione ROS in quanto gestisce l'interfacciamento con il robot; tutti gli altri nodi che intendono comunicare con il TM5-900 devono fare affidamento alle interfacce (servizi, azioni e messaggi) fornite dal driver.

Per il presente progetto sono state approfondite solo alcune delle interfacce messe a disposizione, in particolare:

- Topic */joint_states*: il nodo driver mediante il protocollo TMSVR (Capitolo 1.2.2) riceve informazioni sullo stato corrente del robot che poi pubblica su tale topic; quest'ultimo è sottoscritto da MoveIt che ha così accesso allo stato del TM5-900;
- Topic */tm_driver/sta_response*: il nodo driver pubblica su questo topic il contenuto dei pacchetti TMSTA (Capitolo 1.2.2) ricevuti dal robot, rendendolo accessibile ad ogni nodo che sottoscrive tale topic;
- Servizio */tm_driver/send_script*: servizio offerto dal nodo driver che permette agli altri nodi di inviare comandi al robot in modalità *External Script*;

- Azione */follow_joint_trajectory*: azione offerta dal nodo driver e utilizzata da MoveIt per richiedere l'esecuzione della traiettoria pianificata sul robot reale.

Nelle sezioni successive sono approfonditi alcuni aspetti implementativi relativi alle interfacce *SendScript* e *FollowJointTrajectory* di particolare interesse per la comprensione del progetto.

3.2.1 Servizio *SendScript*

Il servizio *SendScript* è definito nel file *SendScript.srv* riportato nel Listato 3.4.

```
# Request msg
string id
string script
---
# Response msg
bool ok
```

Listing 3.4: File *SendScript.srv*

Questo servizio permette ad un generico nodo di inviare comandi al robot in modalità *External Script* attraverso il protocollo TMSCT (Capitolo 1.2.2). È il nodo driver che si occupa di implementare tale protocollo costruendo il pacchetto TMSCT da inviare al robot. Il nodo client deve soltanto specificare lo script da eseguire compilando i seguenti campi nel messaggio di *Request*:

- id: identificativo dello script inviato al robot; il valore di questo campo viene assegnato all'omonimo campo ID del pacchetto TMSCT inviato dal nodo driver al robot;
- script: script con i comandi che il robot deve eseguire; il valore del campo è assegnato all'omonimo campo SCRIPT del pacchetto TMSCT.

3.2.2 Azione *FollowJointTrajectory*

L'azione *FollowJointTrajectory* è predefinita in ROS nel file *control_msgs/FollowJointTrajectory.action* riportato e commentato nel Listato 3.5.

```
# Goal msg
# Traiettoria da eseguire
trajectory_msgs/JointTrajectory trajectory
# Tolleranze sui singoli punti della traiettoria
# se superate il goal fallisce e restituisce l'errore PATH_TOLERANCE_VIOLATED
JointTolerance[] path_tolerance
# Tolleranze sullo stato finale e sul tempo per raggiungerlo
# se superate il goal fallisce con l'errore GOAL_TOLERANCE_VIOLATED
JointTolerance[] goal_tolerance
duration goal_time_tolerance
---
# Result msg
# Esito dell'esecuzione della traiettoria
int32 error_code
int32 SUCCESSFUL = 0
int32 INVALID_GOAL = -1
int32 INVALID_JOINTS = -2
int32 OLD_HEADER_TIMESTAMP = -3
```

```

int32 PATH_TOLERANCE_VIOLATED = -4
int32 GOAL_TOLERANCE_VIOLATED = -5
string error_string
---

#Feedback msg
# contiene informazioni sulla posizione desiderata, attuale e sul relativo
# errore
Header header
string[] joint_names
trajectory_msgs/JointTrajectoryPoint desired
trajectory_msgs/JointTrajectoryPoint actual
trajectory_msgs/JointTrajectoryPoint error

```

Listing 3.5: File *FollowJointTrajectory.action*

Tale azione viene utilizzata da MoveIt per richiedere l'esecuzione della traiettoria pianificata sul robot reale o simulato. Per l'esecuzione della traiettoria sul robot reale è il nodo driver a fare da server per questa azione trasformando il messaggio di Goal nello script da inviare al robot mediante protocollo TMSCT.

Nello specifico il nodo driver riceve la traiettoria pianificata nel formato specificato nel Listato 3.6.

```

Header header
#Nome e ordine dei giunti
string[] joint_names
#Punti della traiettoria (compresa la posizione iniziale)
JointTrajectoryPoint[] points

```

Listing 3.6: File *trajectory-msgs/JointTrajectory.msg*

Il formato dei singoli punti della traiettoria è invece riportato nel Listato 3.7.

```

#Per ogni punto della traiettoria si specifica:
#tempo necessario per raggiungerlo dalla posizione iniziale
#posizione, velocita' e accelerazione (o posizione e coppia)
#dei singoli giunti
#nell'ordine indicato nel campo joint_names di JointTrajectory.msg
float64[] positions
float64[] velocities
float64[] accelerations
float64[] effort
duration time_from_start

```

Listing 3.7: File *trajectory-msgs/JointTrajectoryPoint.msg*

Per visualizzare il goal richiesto da MoveIt il comando da terminale da utilizzare è quello specificato nel Listato 3.8.

```
$ rostopic echo /joint_trajectory_action/goal
```

Listing 3.8: Comando per visualizzare /joint_trajectory_action/goal

Tale comando visualizza i messaggi di goal come riportato in Figura 3.4.

```
chris@chris-VirtualBox:~$ rostopic echo /joint_trajectory_action/goal
header:
  seq: 0
  stamp:
    secs: 1635429595
    nsecs: 507832682
    frame_id: ''
goal_id:
  stamp:
    secs: 1635429595
    nsecs: 507833801
  id: "/move_group-1-1635429595.507833801"
goal:
  trajectory:
    header:
      seq: 0
      stamp:
        secs: 0
        nsecs: 0
      frame_id: "world"
    joint_names: [elbow_joint, shoulder_1_joint, shoulder_2_joint, wrist_1_joint, wrist_2_joint, wrist_3_joint]
points:
  positions: [-2.13708730193467e-06, 2.1971078993153716e-06, 2.07272223217907e-06, 3.0824069023065556e-06, -1.9310518646326507e-08, -3.2259019298186395e-06]
  velocities: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
  accelerations: [0.0, 3.70062462e0108428e-05, 0.0, 0.0, 0.0, 0.0]
  effort: []
  time_from_start:
    secs: 0
    nsecs: 0
  positions: [0.1934854761570236, 9.595502465189187e-06, -1.0139971535248817e-05, 9.780349154024526e-06, -0.19633767177681793, -1.752775237797468e-06]
  velocities: [0.534856958753646, 0.2053119874556946e-05, -3.32669085178193e-05, 1.824513234425928e-05, -0.5348214585697931, 4.012943278949471e-06]
  accelerations: [0.0071768440913345, 3.794991478867846e-05, -6.264245602236703e-05, 3.435698935402181e-05, -1.007109995788351, 7.556681251289535e-06]
  effort: []
  time_from_start:
    secs: 0
    nsecs: 63233351
  positions: [0.3926992322621349, 1.4993897831063e-05, -2.235221529371394e-05, 1.6478297405742406e-05, -0.3926753024231172, -2.795805177408554e-07]
  velocities: [0.866487002736793, 2.264577799881481e-05, -5.388712308549376e-05, 2.955582436727759e-05, -0.8663494976043133, 6.5085884115538e-06]
  accelerations: [0.9316572613278798, 3.51043747638714e-05, -5.79454367743536e-05, 3.178085238637239e-05, -0.931595425407755, 6.998607314469721e-06]
  effort: []
  time_from_start:
    secs: 0
    nsecs: 890963204
```

Figure 3.4: Messaggio */joint_trajectory_action/goal*

A partire da questo messaggio di goal il nodo driver genera ed invia al robot un pacchetto TMSCT come quello riportato in Figura 3.5.

```
[ INFO] [1635429595.509495699]: $TMSCT,1012,PvtTraj,PVTEnter()
PVTPoint(0.00055, 0.00058, 11.24994, 0.00056, -11.24932, -0.00010, 0.00115, -0.00191, 30.64505, 0.00105, -30.64301, 0.00023, 0.63233)
PVTPoint(0.00097, 0.00128, 22.56001, 0.00094, -22.49864, -0.00002, 0.00187, -0.00389, 49.64146, 0.00169, -49.63817, 0.00037, 0.25863)
PVTPoint(0.00140, 0.00198, 33.75007, 0.00133, -33.74796, 0.00007, 0.00230, -0.00379, 66.92482, 0.00208, -66.92078, 0.00046, 0.26167)
PVTPoint(0.00182, 0.00268, 45.00014, 0.00171, -44.99728, 0.00015, 0.00246, -0.00407, 65.41458, 0.00223, -65.41024, 0.00049, 0.17029)
PVTPoint(0.00225, 0.00338, 56.25021, 0.00210, -50.24660, 0.00024, 0.00226, -0.00373, 59.99778, 0.00205, -59.99380, 0.00045, 0.17371)
PVTPoint(0.00267, 0.00468, 67.50602, 0.00249, -67.49591, 0.00032, 0.00185, -0.00366, 49.14996, 0.00168, -49.14676, 0.00037, 0.20369)
PVTPoint(0.00309, 0.00478, 78.75034, 0.00286, -78.74523, 0.00041, 0.00115, -0.00190, 30.49825, 0.00104, -30.49623, 0.00023, 0.26122)
PVTPoint(0.00352, 0.00548, 96.00040, 0.00325, -89.99455, 0.00049, 0.00000, 0.00000, 0.00000, 0.00000, 0.00000, 0.00000, 0.00000, 0.62750)
PVTExit(),*7d
```

Figure 3.5: Comando di traiettoria inviato al robot

Per il comando di traiettoria viene quindi utilizzata la funzione *PVTPoint()* in spazio giunti (Sezione 1.2.2.3): ad ogni punto della traiettoria pianificata che il nodo driver riceve nel messaggio di goal viene associato il corrispondente PVTPoint. Si fa notare che nella trasformazione vengono apportate le seguenti modifiche:

1. Ordinamento dei giunti: nel PVTPoint i giunti devono essere specificati nell'ordine [shoulder_1_joint, shoulder_2_joint, elbow_joint, wrist_2_joint, wrist_3_joint] mentre nel goal seguono la sequenza [elbow_joint, shoulder_1_joint, shoulder_2_joint, wrist_1_joint, wrist_2_joint, wrist_3_joint];
2. Conversione dei giunti: nel PVTPoint i giunti sono espressi in gradi mentre nel goal sono in radianti;
3. Eliminazione punto iniziale: nel goal viene specificata anche la posizione iniziale del robot che non è richiesta nella modalità PVT di controllo del robot;
4. Calcolo dell'argomento *duration*: il parametro *duration* di ogni PVTPoint viene calcolato come differenza tra i parametri *time_from_start* del corrispondente punto e del precedente punto nella traiettoria del goal.

Come già anticipato (Sezione 1.2.2.3) sono però state riscontrate delle problematiche nell’implementazione dell’azione appena descritta. Il robot reale infatti eseguiva correttamente la traiettoria pianificata da MoveIt ma molto lentamente. È stato quindi necessario modificare tale implementazione come descritto nella Sezione 5.2.

Capitolo 4

Setup hardware e software

In questo capitolo vengono descritti gli step da seguire per poter configurare correttamente l'intero sistema (lato pc e lato robot) per interfacciare il TM5-900 con un'applicazione ROS. I passi da seguire sono gli stessi per testare sia il repository GitHub di partenza [6] sia il repository da noi creato [9].

4.1 Configurazione macchina virtuale Ubuntu 18.04

Per poter lavorare con ROS1 come già accennato occorre disporre di un ambiente Linux e in particolare del sistema operativo Ubuntu; se non si ha a disposizione una macchina Linux, è necessario ricorrere all'utilizzo di una macchina virtuale.

Nel progetto è stato utilizzato il sistema di virtualizzazione VirtualBox della Oracle con cui è stata creata una macchina virtuale con Ubuntu 18.04 (Bionic). Si ricorda infatti che si è scelto di partire dal repository GitHub [6] pensato per ROS Melodic e che tale versione è compatibile con Ubuntu Bionic (Figura 2.1).

Gli step seguiti per creare una macchina virtuale con VirtualBox sono riportati in [10]. Si consiglia di dare alla macchina virtuale almeno la metà della RAM disponibile sul proprio dispositivo e di riservare circa 30GB per l'hard disk virtuale. Inoltre si suggerisce di seguire anche le istruzioni per l'installazione delle *VirtualBox Guest Additions* per migliorare le prestazioni e le funzionalità della macchina virtuale.

In Figura 4.1 sono riportate le specifiche della macchina virtuale utilizzata per il presente progetto.

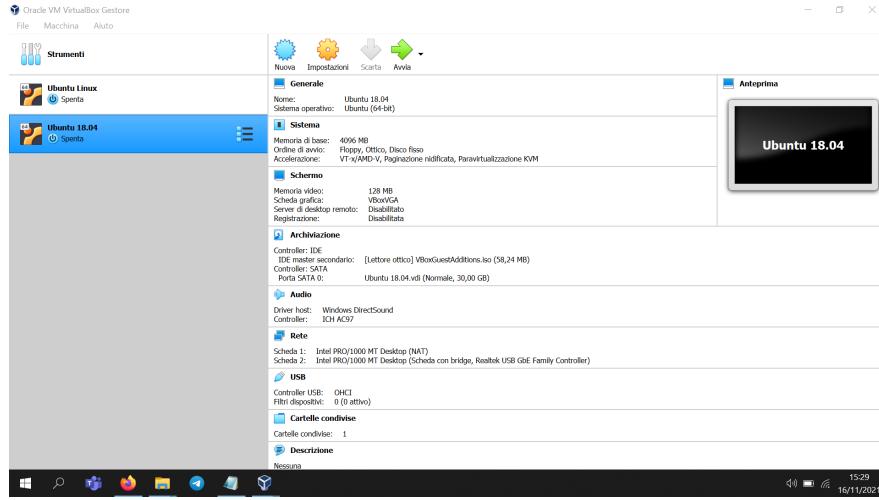


Figure 4.1: Specifiche macchina virtuale

Dopo aver creato la macchina virtuale, occorre anche modificarne le impostazioni di rete di default per permettere la comunicazione TCP/IP tra robot e macchina virtuale. La scheda di rete di default è infatti di tipo NAT (Network Address Translation). Questa modalità consente alla macchina virtuale di accedere a reti esterne (può quindi connettersi ad Internet), ma non la rende direttamente accessibile dall'esterno. Per quest'ultima funzionalità occorre quindi aggiungere una nuova scheda di rete di tipo Bridged. In questa modalità i pacchetti sono inviati e ricevuti direttamente dalla scheda di rete della macchina virtuale che è quindi accessibile da ogni dispositivo connesso fisicamente ad essa.

In Figura 4.2 è riportato come impostare correttamente le schede di rete.

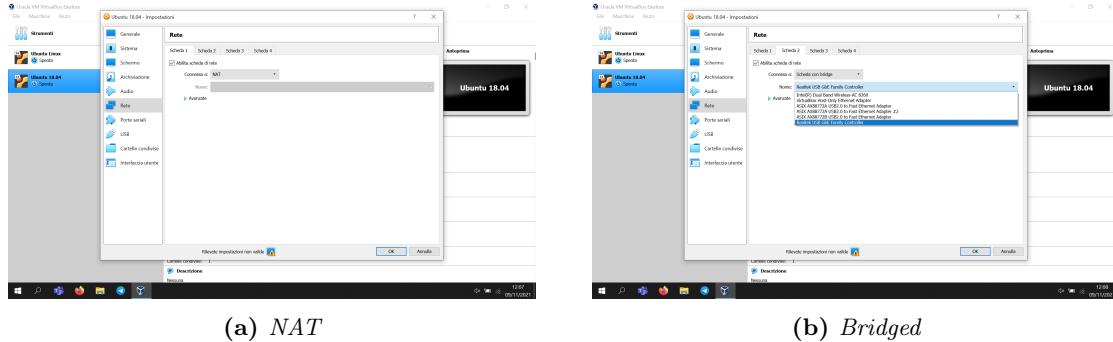


Figure 4.2: Schede di rete

Si fa notare che nella scheda di rete Bridged occorre selezionare la corretta interfaccia di rete (l'adattatore USB-Ethernet utilizzato per la connessione fisica pc-robot).

Infine si deve assegnare alla macchina virtuale un indirizzo IP statico appropriato in modo che la macchina virtuale e il robot appartengano alla stessa rete privata.

Nel presente progetto l'indirizzo IP del robot è di classe B ed è quello riportato in Figura 4.3.

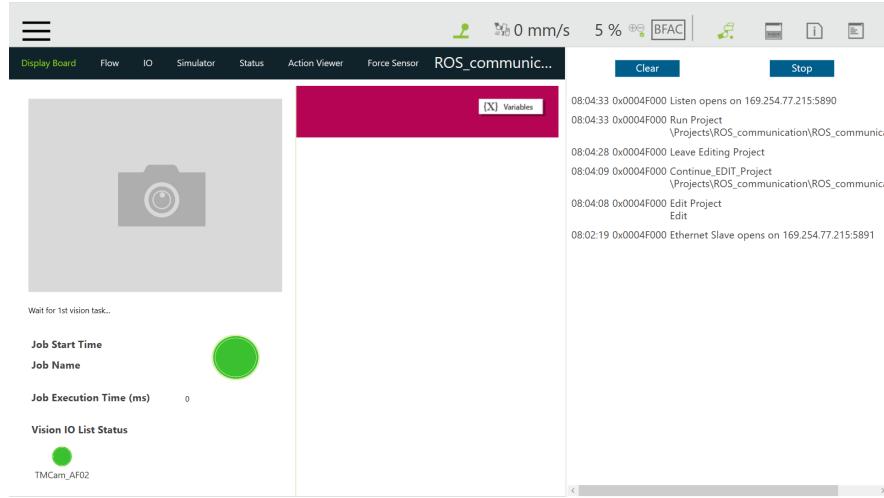


Figure 4.3: Indirizzo IP del robot

Di conseguenza l'indirizzo della macchina virtuale va scelto tra quelli disponibili per questa classe. La Figura 4.4 indica l'indirizzo IP assegnato alla macchina virtuale nel presente progetto e mostra come configurarlo specificando modalità, indirizzo e maschera.

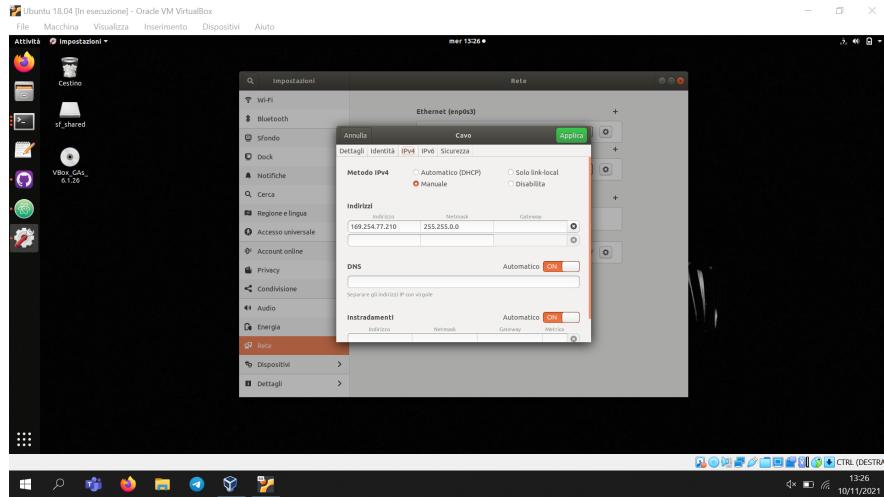


Figure 4.4: Indirizzo IP della macchina virtuale

4.2 Installazione e configurazione ROS Melodic

Lo step successivo è l'installazione di ROS Melodic sulla macchina virtuale precedentemente creata. I passaggi da seguire per l'installazione si possono trovare in [11] (è sufficiente seguire i passi da 1.1 a 1.5). Si consiglia di scegliere la versione *Desktop-Full* per evitare di dover installare manualmente dei pacchetti mancanti.

Nonostante questo ci potrebbero essere alcuni pacchetti mancanti (come MoveIt) che portano

ad errori in fase di compilazione. Il problema può essere risolto installandoli da terminale con il comando riportato nel Listato 4.1.

```
$ sudo apt install ros-melodic-PACKAGE
```

Listing 4.1: Comando per installare il pacchetto "PACKAGE"

Nel caso in cui ROS Melodic sia l'unica versione di ROS attualmente utilizzata si suggerisce di modificare il file nascosto `.bashrc` per aggiungere automaticamente le variabili d'ambiente di ROS alla sessione corrente ogni volta che un nuovo terminale è lanciato (step 1.5 di [11]). Queste variabili sono necessarie per poter utilizzare da terminale i comandi di ROS. Per apportare tale modifica occorre lanciare da terminale i comandi indicati nel Listato 4.2.

```
$ echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

Listing 4.2: Comandi per le variabili d'ambiente di ROS

Altrimenti si deve configurare l'ambiente ROS su ogni nuovo terminale lanciato con il comando riportato nel Listato 4.3.

```
$ source /opt/ros/melodic/setup.bash
```

Listing 4.3: Comando per l'ambiente ROS

4.3 Download e compilazione del repository GitHub

Dopo aver scaricato il repository GitHub che si vuole testare, è necessario decomprimerlo e copiare soltanto la cartella `src` nella cartella del proprio progetto.

Successivamente, dopo essersi posizionati nella cartella `src` da terminale, occorre compilare il workspace ROS con il comando nel Listato 4.4.

```
$ catkin_make
```

Listing 4.4: Comando di compilazione

Al termine della compilazione vengono create le cartelle `build` e `devel`. Si suggerisce di modificare il file `.bashrc` come mostrato in Figura 4.5 per aggiungere automaticamente il workspace alla sessione corrente ogni volta che un nuovo terminale viene lanciato. Questo passaggio è necessario per poter mandare in esecuzione l'applicazione ROS.

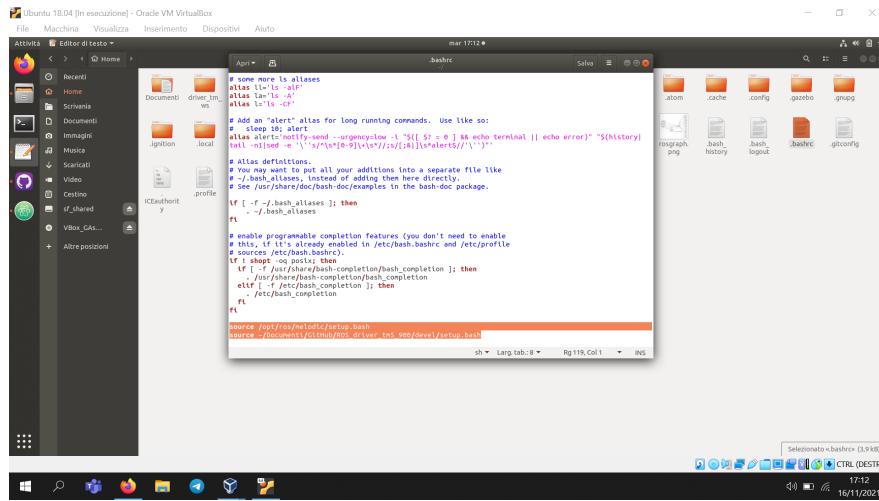


Figure 4.5: Modifica del file .bashrc

In alternativa alla modifica del file `.bashrc` occorre aggiungere il workspace ad ogni nuovo terminale con il comando nel Listato 4.5.

```
$ source [PATH]/[WORKING_DIRECTORY_NAME]/devel/setup.bash
```

Listing 4.5: Comando per configurare il workspace

4.4 Connessione fisica robot-pc

Il robot TM5-900 e il computer su cui è in esecuzione la macchina virtuale devono essere collegati fisicamente via cavo Ethernet (RJ45). È importante connettere il cavo di rete alla porta LAN della control box (come in Figura 4.6) e non alle porte GigE. Si è riscontrato infatti che soltanto utilizzando la porta LAN il manipolatore è accessibile dall'esterno. Utilizzando una delle due porte GigE i socket TCP sono disponibili soltanto sul localhost 127.0.0.1 rendendo impossibile la comunicazione robot-pc. Si fa notare però che è presente una discussione aperta sul repository GitHub [6] relativa a tale problematica in cui la soluzione proposta è invece quella di utilizzare una porta GigE. Se si riscontrano problemi analoghi si suggerisce quindi di provare a cambiare porta sulla control box.

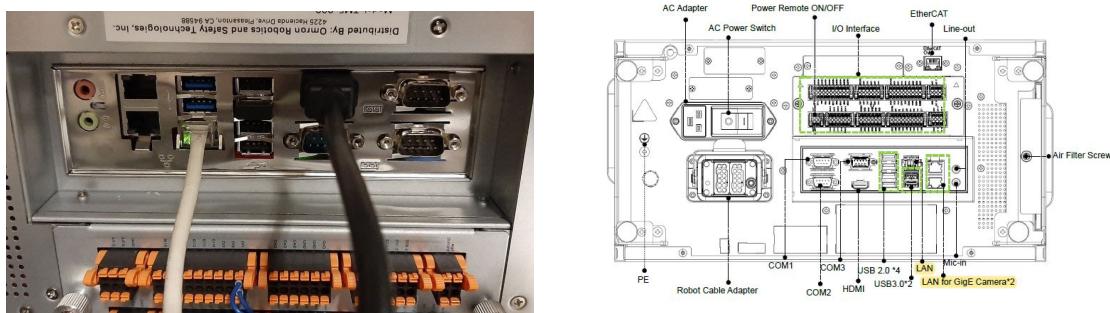


Figure 4.6: Connessione fisica robot-pc

4.5 Configurazione TMflow

L'ultimo step da seguire per interfacciare il TM5-900 con l'applicazione ROS è la configurazione del robot in TMflow. Come già anticipato nella Sezione 1.2.2 per il presente progetto non è necessario sviluppare un vero e proprio programma su TMflow, ma è sufficiente abilitare i socket TCP *Ethernet Slave* e *External Script*. Di seguito è spiegato in dettaglio come abilitare queste funzionalità.

4.5.1 Configurazione di rete

Per rendere il robot TM5-900 accessibile dall'esterno non è necessario modificarne le impostazioni di rete diversamente da quanto descritto nel README di [6]. È sufficiente utilizzare la porta LAN della control box per il collegamento fisico con il computer ed assegnare alla macchina virtuale un indirizzo IP statico compatibile con quello del robot.

L'indirizzo IP del manipolatore è visibile o dalla schermata di log (Figura 4.7) o dalle impostazioni di rete accessibili da \equiv nella sezione *System → Network*.

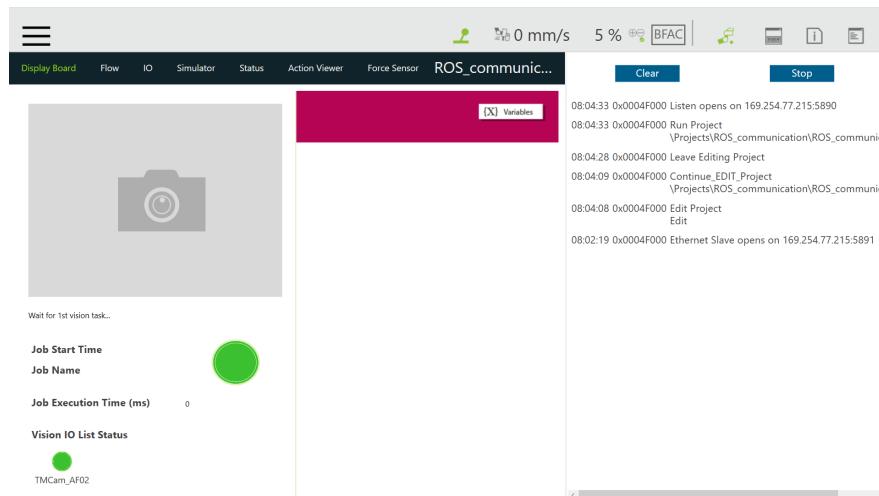


Figure 4.7: Indirizzo IP del manipolatore nella schermata di log

4.5.2 Ethernet Slave

Per abilitare il socket TCP *Ethernet Slave*, che permette di leggere e scrivere dall'esterno le variabili della *Data Table*, occorre accedere da \equiv nella sezione *Setting → Connection → Ethernet Slave*. Da qui è possibile attivare il socket con il pulsante dedicato come mostrato nella Figura 4.8.

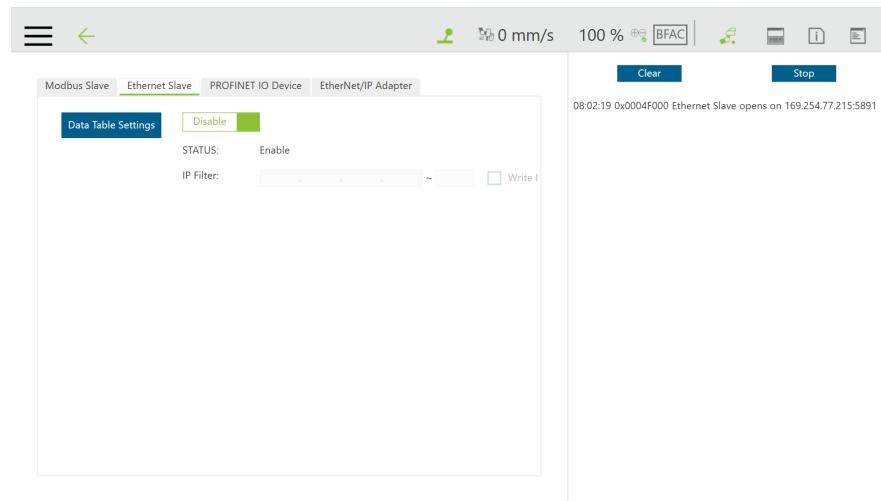


Figure 4.8: Schermata per attivare l'Ethernet Slave

Prima di attivare il socket occorre definire il contenuto della *Data Table* in formato binario, selezionando le seguenti variabili a cui l'applicazione ROS del presente progetto deve avere accesso:

- ✓ Robot_Error
- ✓ Project_Run
- ✓ Project_Pause
- ✓ Safeguard_A
- ✓ ESTOP
- ✓ Camera_Light
- ✓ Error_Code
- ✓ Joint_Angle
- ✓ Coord_Robot_Flange
- ✓ Coord_Robot_Tool
- ✓ TCP_Force
- ✓ TCP_Force3D
- ✓ TCP_Speed
- ✓ TCP_Speed3D
- ✓ Joint_Speed
- ✓ Joint_Torque

- ✓ Project_Speed
- ✓ MA_Mode
- ✓ Robot Light
- ✓ Ctrl_DO0-DO7
- ✓ Ctrl_DI0-DI7
- ✓ Ctrl_AO0
- ✓ Ctrl_AI0-AI1
- ✓ END_DO0-DO3
- ✓ END_DI0-DI2
- ✓ END_AI0

La schermata per definire la Data Table è riportata in Figura 4.9; da qui è possibile o creare un nuovo *Transmit File* selezionando le variabili di interesse o importarlo direttamente seguendo le istruzioni del README del repository GitHub [12].

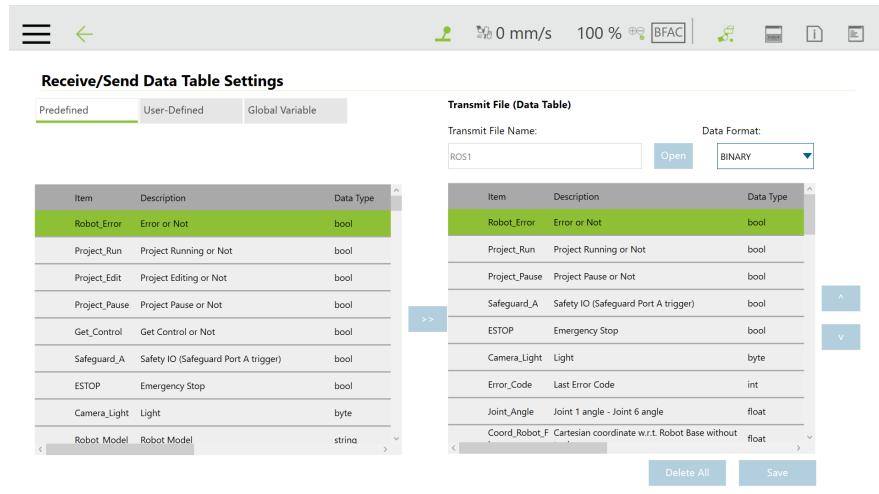


Figure 4.9: Schermata per definire la Data Table

4.5.3 Programma TMflow

Per abilitare il socket TCP *External Script*, che permette di comandare il robot dall'esterno, occorre agire direttamente a livello di programma TMflow.

Il progetto TMflow da creare e da mandare in esecuzione sul manipolatore è riportato in Figura 4.10.

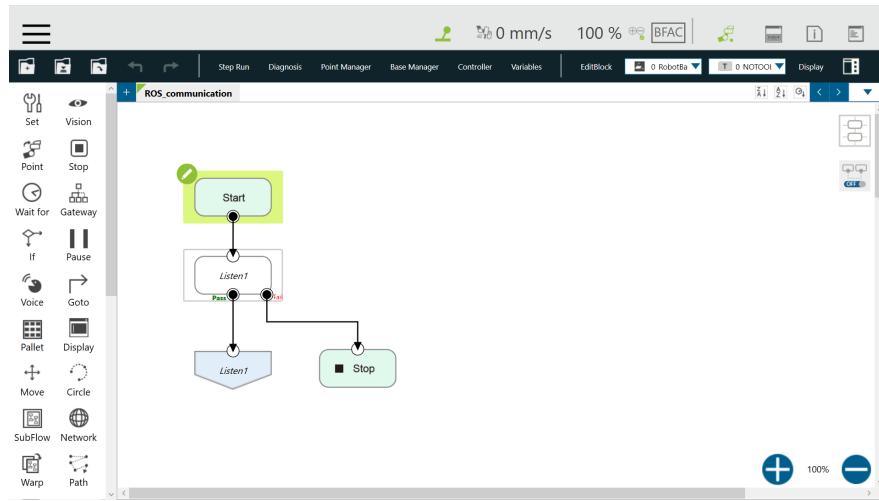


Figure 4.10: Programma TMflow per External Script

Il progetto TMflow consiste in un semplice loop infinito che mantiene il programma sul nodo *Listen*; si ricorda infatti che occorre mantenere il flusso del programma su tale nodo affinché la modalità *External Script* rimanga attiva e sia possibile comandare il robot dall'esterno. Per creare il programma è sufficiente trascinare il nodo *Listen* dal menù dei nodi sulla sinistra, lasciando i suoi parametri ai loro valori di default (Figura 4.11).

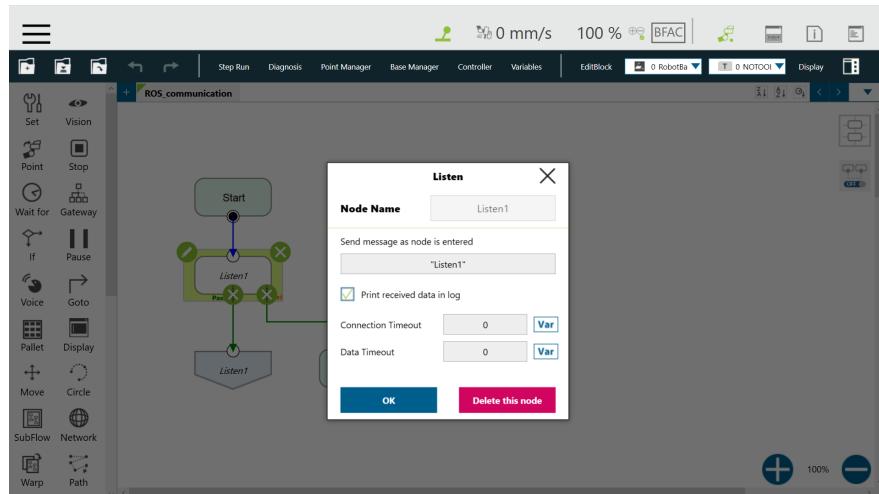


Figure 4.11: Nodo Listen

La condizione di uscita *Fail* del nodo *Listen* va collegata ad un nodo *Stop* in modo da interrompere il programma se si verifica una condizione di errore. La condizione di uscita *Pass* va invece connessa ad un nodo *Goto* che riporta il flusso sul nodo *Listen*.

Per creare il progetto TMflow si possono seguire i passi indicati nella Sezione 1.2.1.3. Il programma TMflow deve poi essere mandato in esecuzione come descritto nella Sezione 1.2.1.4.

4.5.4 Test connection

Dopo aver configurato il robot e la macchina virtuale si consiglia di verificare la corretta connessione tra i due con un ping da terminale. Per eseguire questo test occorre lanciare il comando riportato nel Listato 4.6 specificando l'indirizzo IP del robot.

```
$ ping IP_ROBOT
```

Listing 4.6: Test della connessione con ping

In Figura 4.12 viene mostrato l'output restituito dal terminale in caso di corretta connessione manipolatore-computer.

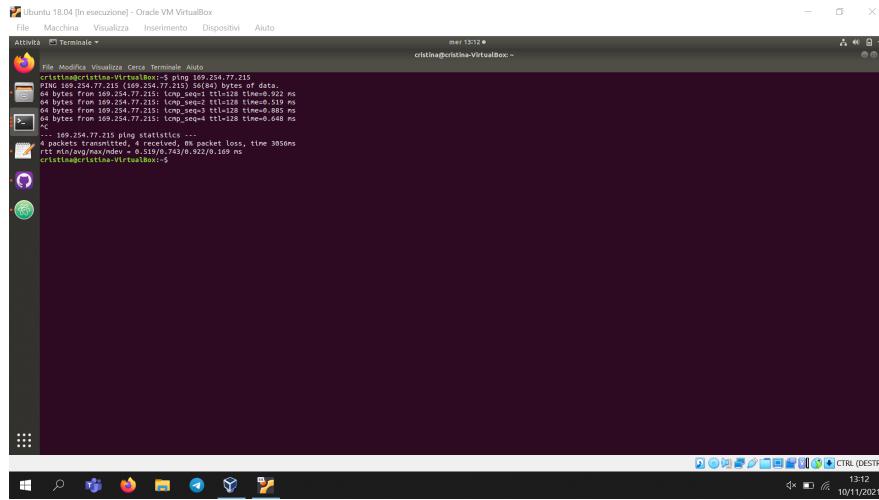


Figure 4.12: Verifica connessione robot-pc con ping

Capitolo 5

Driver ROS modificato

In questo capitolo vengono illustrate le modifiche apportate al driver ROS descritto nel Capitolo 3 per la gestione degli ostacoli.

Come già anticipato, inizialmente è stato testato il corretto funzionamento del repository GitHub [6]. Si è verificato che il driver permette la corretta connessione robot-pc; tuttavia è stato riscontrato un problema relativo all'esecuzione sul robot reale della traiettoria pianificata con MoveIt. Di conseguenza è stata effettuata una modifica per correggere tale problematica che verrà approfondita nella Sezione 5.2.

Successivamente è stata implementata la logica per la simulazione del rilevamento degli ostacoli e per la loro gestione.

5.1 Gestione ostacoli

In questo paragrafo viene descritto in dettaglio il problema che il presente progetto affronta e la relativa soluzione proposta.

5.1.1 Problema

Come già accennato l'obiettivo del progetto è quello di gestire l'eventuale comparsa di ostacoli nell'ambiente di lavoro del robot; nello specifico l'idea è di utilizzare un sistema di telecamere esterne per rilevare le potenziali collisioni e sfruttare tale informazione per ripianificare la traiettoria del manipolatore in modo che possa portare a termine il task assegnatogli evitando gli ostacoli.

In questo progetto non sono stati gestiti il rilevamento degli ostacoli e la ripianificazione della traiettoria, ma ci si è limitati a simulare via software la presenza di un generico ostacolo e a gestirlo semplicemente fermando il robot e facendolo tornare alla posizione di home.

5.1.2 Soluzione

La soluzione proposta è schematicamente rappresentata in Figura 5.1.



Figure 5.1: Schema della soluzione proposta

Per quanto riguarda la simulazione del rilevamento dell’ostacolo viene utilizzato il nodo ROS **Obstacle Detection Naive** che si limita soltanto a pubblicare un messaggio di **Obstacle Detected**; quest’ultimo è a sua volta sottoscritto dal nodo ROS **Obstacle Avoidance Naive** che si occupa di ripianificare la traiettoria. È sufficiente quindi mandare in esecuzione il nodo **Obstacle Detection Naive** per simulare la presenza di un ostacolo nell’ambiente di lavoro del robot. Tuttavia tale nodo va ovviamente modificato per il rilevamento reale delle collisioni in modo che il messaggio di **Obstacle Detected** venga pubblicato soltanto quando il manipolatore rischia di urtare contro un ostacolo.

Il messaggio di **Obstacle Detected** non è predefinito in ROS, ma è customizzato al fine di contenere tutte le informazioni fornite dal nodo **Obstacle Detection Naive** e necessarie al nodo **Obstacle Avoidance Naive** per la gestione dell’ostacolo rilevato. Attualmente il messaggio contiene soltanto un flag booleano per segnalare che il robot rischia una collisione in quanto non viene gestita una vera e propria ripianificazione. Quindi anche tale messaggio deve essere modificato per la gestione di ostacoli reali.

Rimane infine da trattare il nodo **Obstacle Avoidance Naive** che nel presente progetto non effettua propriamente una ripianificazione della traiettoria che il robot deve eseguire, ma si limita ad arrestarlo e farlo tornare alla posizione di home. A tal fine il nodo si mette in ascolto di un messaggio di **Obstacle Detected** per eseguire una funzione di callback al suo arrivo. Nella callback il nodo utilizza il servizio *SendScript* offerto dal nodo driver per inviare comandi al robot in modalità *External Script* attraverso il protocollo TMSCT (Sezione 1.2.2). Nello specifico viene mandato al manipolatore il comando *StopAndClearBuffer()* per interrompere il movimento corrente del TM5-900 e cancellare i comandi in coda che sono ancora in attesa di essere eseguiti. Si è scelto inoltre di inviare successivamente un comando per riportare il robot alla posizione di home utilizzando la funzione *PTP()* in spazio giunti; questa operazione ha generato alcune problematiche che verranno successivamente dettagliate nella Sezione 7.3. Anche la funzione di callback va ovviamente modificata includendovi la logica per la ripianificazione della traiettoria.

Questa soluzione può essere utilizzata sia quando il robot esegue una traiettoria pianificata da MoveIt, sia quando la traiettoria effettuata è predefinita in ROS. In entrambi i casi infatti la traiettoria è fornita da ROS al manipolatore in modalità *External Script*. Non è invece possibile adottare tale soluzione se si vuole programmare il movimento del robot in TMflow ma occorre cambiare approccio. Una possibile soluzione alternativa è accennata nella Sezione 7.2.

Di seguito è illustrata in dettaglio la soluzione proposta nel presente progetto applicata alle due diverse definizioni di traiettoria precedentemente citate.

5.2 Gestione ostacoli con traiettoria pianificata

Per applicare la soluzione proposta al caso di traiettoria definita da MoveIt si è partiti dal file di launch fornito dal repository [6] e già descritto nella Sezione 3.1.1. Per includere la gestione degli ostacoli in questa applicazione ROS è sufficiente aggiungere nel file di launch il nodo ***Obstacle Avoidance Naive*** descritto nella precedente sezione e utilizzare il nodo ***Obstacle Detection Naive*** per simulare l'ostacolo.

Nell'appendice B sono riportate tutte le modifiche apportate al codice (file di launch e i due nodi aggiuntivi).

Tuttavia, come già accennato, testando l'applicazione ROS di partenza è stato riscontrato che il robot reale esegue correttamente la traiettoria pianificata da MoveIt ma molto lentamente. Per risolvere questa problematica è stato necessario modificare il codice che si occupa di convertire il goal dell'azione *FollowJointTrajectory* richiesta da MoveIt nel pacchetto TMSCT da inviare al robot affinché esso esegua la traiettoria pianificata. In particolare è stato modificato il metodo *TmCommand::set_pvt_traj*, che si trova nel file *NOME_PROGETTO/src/tm_driver/src/tm_command.cpp*, come mostrato nel Listato 5.1.

```

1 std::string TmCommand::set_pvt_traj(const TmPvtTraj &pvts, int precision) {
2     std::stringstream ss;
3     ss << std::fixed << std::setprecision(precision);
4     if (pvts.mode == TmPvtMode::Joint) {
5         ss << "PVTEnter(0)\r\n";
6         for (auto &point : pvts.points) {
7             ss << "PVTPoint(";
8             for (auto &value : point.positions) { ss << deg(value) << ","; }
9             // CORRECTION to overcome robot's slow motion with PVTPoint()
10            //for (auto &value : point.velocities) { ss << deg(value) << ","; }
11            int k = 6;
12            for (auto &value : point.velocities) { double velocity_scaled = k*deg(
13                value); ss << velocity_scaled << ","; }
14            //ss << point.time << ")\r\n";
15            double time_scaled = point.time/k;
16            ss << time_scaled << ")\r\n";
17        }
18    } else {
19        ss << "PVTEnter(1)\r\n";
20        for (auto &point : pvts.points) {
21            ss << "PVTPoint(";
22            auto pv = mmdeg_pose(point.positions);
23            for (auto &value : pv) { ss << value << ","; }
24            auto vv = mmdeg_pose(point.velocities);
25            for (auto &value : vv) { ss << value << ","; }
26            ss << point.time << ")\r\n";
27        }
28    }
29    ss << "PVTExit()";
30    //ROS_INFO_STREAM("Comando: "<<ss.str());
31    return ss.str();

```

Listing 5.1: Modifica del metodo *TmCommand::set_pvt_traj*

Si evidenzia subito che non è stato possibile risolvere completamente il problema data la carenza di informazioni della documentazione ufficiale di TMflow (Sezione 7.3). Infatti probabilmente esso è dovuto ad una incongruenza nelle unità di misura che si è venuta a creare a seguito di un eventuale cambiamento del significato assegnato agli argomenti del comando *PVTPoint()*. È stata quindi adottata una soluzione temporanea introducendo un fattore di scala (k nel listato) calcolato empiricamente ed utilizzato per scalare sia le velocità (moltiplicandole per k) sia i tempi (dividendoli per k) dei singoli PVTPoint.

5.3 Gestione ostacoli con traiettoria predefinita

Per applicare la soluzione proposta al caso di traiettoria predefinita in ROS è necessario creare un nuovo file di launch che manda in esecuzione il nodo driver e il nodo ***Obstacle Avoidance Naive*** precedentemente descritti oltre ad un nuovo nodo ***Trajectory***. Quest'ultimo si occupa di inviare al robot comandi in modalità *External Script* in modo che esso esegua ciclicamente la traiettoria desiderata. Per gestire l'esecuzione della traiettoria in loop si utilizza la funzione *QueueTag()* descritta nella Sezione 1.2.2.3 per etichettare l'ultimo comando della traiettoria in modo che il robot invii un pacchetto *TMSTA SubCmd 01* al termine dell'esecuzione. Il nodo ***Trajectory*** quindi invia al TM5-900 un comando di traiettoria e, soltanto dopo aver ricevuto il pacchetto *TMSTA SubCmd 01* ad esso associato, ripete il loop inviando nuovamente lo stesso comando.

Per gestire questa logica viene utilizzato il flag booleano *flag_done* che è inizializzato a false e che viene valorizzato a true dalla funzione di callback *StaResponseCallback()* riportata nel Listato 5.2.

```

1 // Callback executed when a TM_STA message is received to implement a loop
2 // trajectory
3 void StaResponseCallback(bool* flag_done_p, const tm_msgs::StaResponse::
4     ConstPtr& msg){
5     //Check if it is the TMSTA message in response to the QueueTag of the
6     // trajectory commands
7     ROS_INFO_STREAM("TMSTA message: subcmd is = " << msg->subcmd << ", subdata
8     is " << msg->subdata);
9     std::string cmd = "01"; //QueueTag
10    std::string data = "01,true"; // Tag number = 01, Status = true
11    if((msg->subcmd.compare(cmd)==0)&&(msg->subdata.compare(data)==0)) {
12        ROS_INFO_STREAM("PICK: Trajectory executed!");
13        *flag_done_p = true;
14    }
15 }
```

Listing 5.2: StaResponseCallback()

Si fa notare che questa callback viene richiamata ogni volta che il robot invia un pacchetto *TMSTA* ed è quindi necessario verificare che il pacchetto ricevuto sia quello generato dall'esecuzione del comando *QueueTag()* della traiettoria.

La presenza del nuovo nodo ***Trajectory*** e la differente definizione di traiettoria adottata in questo caso richiedono di modificare leggermente la logica che sottende la gestione degli

ostacoli. Nel momento in cui viene rilevato un ostacolo non è sufficiente inviare il comando di *StopAndClearBuffer()* come nell'implementazione con traiettoria pianificata da MoveIt ma occorre anche arrestare il loop facendo terminare l'esecuzione del nodo **Trajectory**.

Il messaggio **Obstacle Detected** deve quindi essere sottoscritto dai seguenti nodi:

- nodo **Trajectory**: alla ricezione del messaggio viene eseguita una funzione di callback che termina l'esecuzione di tale nodo;
- nodo **Obstacle Avoidance Naive**: la funzione di callback manda il comando di *StopAndClearBuffer()* per interrompere immediatamente il movimento del robot; dopodiché attende la terminazione del nodo **Trajectory** per inviare nuovamente un comando di *StopAndClearBuffer()* e un comando *PTP()* per riportare il robot alla posizione di home.

La terminazione del nodo **Trajectory** viene gestita con il flag booleano *flag_loop* che viene inizializzato a true e valorizzato a false nella callback *KillCallback()* riportata nel Listato 5.3.

```

1 // Callback executed when ObstacleDetected message is received to kill the
2 // node
3 void KillCallback(bool* flag_loop_p, ros::NodeHandle &nh, const tm_msgs::
4   ObstacleDetected::ConstPtr& msg){
5   std::string flag = msg->obstacle_detected ? "true" : "false";
6   ROS_INFO_STREAM("PICK: Received message ObstacleDetected: " << flag );
7
8   // Kill the node by setting the flag_loop variable to false
9   ROS_INFO_STREAM("PICK: pick_place_trajectory_node killing... ");
10  *flag_loop_p = false;
11  //ros::shutdown();
12 }
```

Listing 5.3: KillCallback()

Nel Listato 5.4 viene invece riportata la nuova funzione di callback *ObstacleDetectedCallback()* relativa al nodo **Obstacle Avoidance Naive**.

```

1 // Callback executed when ObstacleDetected message is received
2 void ObstacleDetectedCallback(ros::NodeHandle &nh, const tm_msgs::
3   ObstacleDetected::ConstPtr& msg){
4   std::string flag = msg->obstacle_detected ? "true" : "false";
5   ROS_INFO_STREAM("PICK: Received message ObstacleDetected: " << flag );
6   // First command needed to stop immediately the trajectory
7   ROS_INFO_STREAM("PICK: First clear");
8   // Create the client to request the service SendScript
9   ros::ServiceClient client = nh.serviceClient<tm_msgs::SendScript>(
10     "tm_driver/send_script");
11   // Create the request with the commands to stop the trajectory
12   std::string cmd2 = "StopAndClearBuffer()";
13   tm_msgs::SendScript srv;
14   srv.request.id = "obs1";
15   srv.request.script = cmd2;
16   // Service call
17   if (client.call(srv)){
18     if (srv.response.ok) ROS_INFO_STREAM("PICK: Sent script to robot");
19     else ROS_WARN_STREAM("Sent script to robot but response not yet ok");
```

```

18     } else { ROS_ERROR_STREAM("PICK: Error send script to robot"); }
19     //IMPORTANT: Wait for pick_place_trajectory_node to stop
20     ROS_INFO_STREAM("PICK: Wait for pick_place_trajectory_node to stop");
21     bool waiting = true;
22     std::string node = "/pick_place_trajectory_node";
23     while(waiting) {
24         ros::V_string v;
25         ros::master::getNodes(v);
26         waiting = false;
27         for (int i=0; i < v.size(); i++) {
28             if (v[i].compare(node)==0) {
29                 ROS_INFO_STREAM("PICK: still exists" << v[i] << "node...wait...");
30                 waiting = true;
31             }
32         }
33         ros::Duration(0.5).sleep();
34     }
35     ROS_INFO_STREAM("PICK: pick_place_trajectory_node terminated, send clear
36 and home...");
37     // Create the client to request the service SendScript
38     ros::ServiceClient client1 = nh.serviceClient<tm_msgs::SendScript>("tm_driver/send_script");
39     // Create the request with the commands to stop the trajectory and return
40     // to home state
41     std::string cmd = "StopAndClearBuffer()\r\nPTP(\"JPP
42 \\",0,0,0,0,0,0,5,500,0,false)"; //stop+home
43     tm_msgs::SendScript srv1;
44     srv1.request.id = "obs2";
45     srv1.request.script = cmd;
46     // Service call
47     if (client1.call(srv1)){
48         if (srv1.response.ok) ROS_INFO_STREAM("PICK: Sent script to robot");
49         else ROS_WARN_STREAM("Sent script to robot, but response not yet ok");
50     }else{ ROS_ERROR_STREAM("PICK: Error send script to robot"); }
51 }
```

Listing 5.4: ObstacleDetectedCallback()

Si fa notare che la callback è analoga a quella utilizzata nel caso di traiettoria pianificata da MoveIt; infatti anch'essa utilizza il servizio *SendScript* offerto dal nodo driver per inviare comandi al robot in modalità *External Script*. Da evidenziare è soltanto la funzione *getNodes()* messa a disposizione da ROS ed utilizzata nel presente progetto per verificare l'avvenuta terminazione del nodo **Trajectory**. Tale funzione infatti restituisce la lista dei nodi attualmente in esecuzione nell'applicazione ROS; è quindi sufficiente attendere che il nodo **Trajectory** non sia più presente nella lista per verificarne la terminazione.

Nel presente progetto, come verrà meglio chiarito nella Sezione 6, sono state realizzate due diverse implementazioni del nodo **Trajectory** che differiscono esclusivamente per il comando di traiettoria. Nel Listato 5.5 è mostrato il main di una delle due implementazioni.

```

1 int main(int argc, char **argv){
2     //Flag initialization
3     bool flag_loop = true; // loop trajectory on
4     bool flag_done = false; // trajectory commands still not executed
5     //Node initialization
```

```

6  ros::init(argc, argv, "loop_trajectory_node");
7  ros::NodeHandle nh;
8  //IMPORTANT: Wait for TM_SCT connection (otherwise the node tries to send
9  //script before the connection is established)
10 ROS_INFO_STREAM("LOOP: Wait for TM_SCT connection...");
11 ros::Duration(1).sleep(); //seconds
12 // Create subscriber to receive ObstacleDetected messages
13 ros::Subscriber sub = nh.subscribe<tm_msgs::ObstacleDetected>("tm_driver/
   obstacle_detected", 5, boost::bind(&KillCallback,&flag_loop, boost::ref(nh
), _1));
14 // Create subscriber to receive StaResponse messages
15 ros::Subscriber sub2 = nh.subscribe<tm_msgs::StaResponse>("tm_driver/
   sta_response", 1000, boost::bind(&StaResponseCallback,&flag_done,_1));
16 // Create the client to request the service SendScript
17 ros::ServiceClient client = nh.serviceClient<tm_msgs::SendScript>("tm_driver/send_script");
18 // Create the request with the first commands for the loop trajectory
19 tm_msgs::SendScript srv;
20 std::string cmd = "Move_PTP(\"JPP\",90,0,0,0,0,0,5,200,0,false)\r\nMove_PTP
   (\\"JPP\\",-90,0,0,0,0,0,5,200,0,false)\r\nQueueTag(1)";
21 srv.request.id = "loop";
22 srv.request.script = cmd;
23 ROS_INFO_STREAM("LOOP: First command...");
24 // Service call
25 if (client.call(srv)){
26     if (srv.response.ok) ROS_INFO_STREAM("LOOP: Sent script to robot");
27     else ROS_WARN_STREAM("Sent script to robot , but response not yet ok ");
28 } else{
29     ROS_ERROR_STREAM("LOOP: Error send script to robot");
30     return 1;
31 }
32 // Trajectory loop
33 while(flag_loop) {
34     if(flag_done){
35         //Send new trajectory command
36         ROS_INFO_STREAM("LOOP: New trajectory command");
37         //Check for messages
38         ros::spinOnce();
39         if (client.call(srv))
40         {
41             if (srv.response.ok) ROS_INFO_STREAM("LOOP: Sent script to
   robot");
42             else ROS_WARN_STREAM("Sent script to robot , but response not
   yet ok ");
43         } else {
44             ROS_ERROR_STREAM("LOOP: Error send script to robot");
45             return 1;
46         }
47         flag_done = false;
48     } else {// Wait TM_STA message of trajectory executed
49         ROS_INFO_STREAM("LOOP: Wait trajectory executed");
50         //Check for messages
51         ros::spinOnce();
52         ros::Duration(1).sleep();
53     }
54 //Check for messages

```

```
54     ros::spinOnce();  
55 } // Node shutdown for obstacle detection  
56 ROS_INFO_STREAM("LOOP: loop trajectory killed");  
57 return 0;}
```

Listing 5.5: Main del nodo Trajectory

Del codice nel Listato 5.5 si fa notare che alle righe 8-10 viene inserita un’istruzione per attendere l’avvenuta connessione robot-pc affinché il primo comando di traiettoria venga correttamente ricevuto dal manipolatore.

Nel ciclo while alle righe 31-55 viene gestito il loop relativo alla traiettoria grazie all’utilizzo dei due flag *flag_loop* e *flag_done* precedentemente descritti: come già anticipato, finché non viene rilevato un ostacolo, la traiettoria continua ad essere eseguita inviando un nuovo comando ogni volta che il precedente è stato eseguito dal robot.

Nell’appendice C è riportato interamente il codice utilizzato per la gestione degli ostacoli con traiettoria predefinita da ROS (file di launch e nodi).

Capitolo 6

Utilizzo driver ROS modificato

In questo capitolo viene spiegato come mandare in esecuzione i programmi ROS descritti nel Capitolo 5 e il loro funzionamento. Per ognuno di questi programmi è stato creato un file di launch che l'utente può lanciare attraverso il seguente comando:

```
$ roslaunch obstacle_avoidance [LAUNCH_FILE_NAME]
```

Listing 6.1: Comando per lanciare un file di launch

Per poter utilizzare tale comando è necessario effettuare prima tutti i passi descritti nel Capitolo 4 per configurare correttamente l'intero sistema.

Si ricorda che l'indirizzo IP del robot di default è 169.254.77.215; se il robot è associato ad un indirizzo differente, si deve aggiornare il parametro *robot.ip_address* del nodo driver all'interno del file di launch con il valore di tale indirizzo. I file di launch creati nel presente progetto sono i seguenti:

- loop_trajectory.launch
- pick_place_trajectory.launch
- moveit_trajectory.launch
- test_functions.launch

Nelle sezioni successive viene brevemente descritto come utilizzarli.

6.1 test_functions

Questo file di launch è un programma ausiliario aggiuntivo che permette all'utente di testare i comandi che possono essere inviati al robot in modalità *External Script*.

Per testare un nuovo comando è sufficiente creare una variabile di tipo stringa, valorizzarla con il comando che si desidera mandare in esecuzione sul robot e assegnarla al campo *srv.request.script* del messaggio di richiesta del servizio *SendScript*. Quest'ultima operazione è mostrata alla riga 50 del Listato 6.2.

```
1 //Node to test external script functions
2 #include <ros/ros.h>
3 #include <std_msgs/String.h>
```

```

4
5 #include <sstream>
6 #include <cstdlib>
7
8 #include "tm_msgs/SendScript.h"
9
10 int main(int argc, char **argv)
11 {
12     // Commands tested
13     std::string cmd = "StopAndClearBuffer()"; //stop
14     std::string cmd2 = "StopAndClearBuffer()\r\nResume()\r\nPTP(\"JPP
15         \",0,0,0,0,0,0,35,200,0,false)"; //stop+home
16     std::string cmd3 = "Move_PTP(\"JPP\",90,0,0,0,0,0,5,200,0,false)"; // J1
17     std::string cmd4 = "Move_PTP(\"JPP\",0,0,0,0,0,180,5,200,0,false)"; // J6
18     std::string cmd5 = "PVTEnter(1)\r\nPVTPoint
19         (517.5,-236.6,574.10,90,90,0,0,0,0,0,0,0,1)\r\nQueueTag(1)\r\nPVTExit()";
20         //cartesian
21     std::string cmd6 = "PTP(\"CPP\",517.5,-236.6,574.10,90,90,0,10,200,0,false)
22         ";
23     std::string cmd7 = "PVTEnter(0)\r\nPVTPoint(90,0,0,0,0,0,0,0,0,0,0,0,0.125)
24         \r\nQueueTag(1)\r\nPVTExit()"; //joint
25
26     std::stringstream ss;
27     ss << "PVTEnter(0)\r\n";
28     ss << "PVTPoint
29         (0.00055,-0.00058,11.24994,0.00056,-11.24932,-0.00010,0.00115,-0.00191,
30         30.64505,0.00105,-30.64301,0.00023,0.316165)\r\n";
31     ss << "PVTPoint
32         (0.00097,-0.00128,22.50001,0.00094,-22.49864,-0.00002,0.00187,-0.00309,
33         49.64146,0.00169,-49.63817,0.00037,0.129315)\r\n";
34     ss << "PVTPoint
35         (0.00140,-0.00198,33.75007,0.00133,-33.74796,0.00007,0.00230,-0.00379,
36         60.92482,0.00208,-60.92078,0.00046,0.100835)\r\n";
37     ss << "PVTPoint
38         (0.00182,-0.00268,45.00014,0.00171,-44.99728,0.00015,0.00246,-0.00407,
40         65.41458,0.00223,-65.41024,0.00049,0.085145)\r\n";
41     ss << "PVTPoint
42         (0.00225,-0.00338,56.25021,0.00210,-56.24660,0.00024,0.00226,-0.00373,
44         59.99778,0.00205,-59.99380,0.00045,0.086855)\r\n";
45     ss << "PVTPoint
46         (0.00267,-0.00408,67.50027,0.00248,-67.49591,0.00032,0.00185,-0.00306,
48         49.14996,0.00168,-49.14670,0.00037,0.101845)\r\n";
49     ss << "PVTPoint
50         (0.00309,-0.00478,78.75034,0.00286,-78.74523,0.00041,0.00115,-0.00190,
52         30.49825,0.00104,-30.49623,0.00023,0.13061)\r\n";
53     ss << "PVTPoint
54         (0.00352,-0.00548,90.00040,0.00325,-89.99455,0.00049,0.00000,0.00000,
56         0.00000,0.00000,0.00000,0.31375)\r\n";
57     ss << "QueueTag(1)\r\nPVTExit()";
58     std::string cmd8 = ss.str();
59     //ROS_INFO_STREAM("Comando: "<<cmd8);
60
61     std::string cmd9 = "Move_PTP(\"JPP\",-270,0,0,0,0,0,5,1000,0,false)";
62
63     //Node initialization
64     ros::init(argc, argv, "send_script_node");

```

```

39 ros::NodeHandle nh;
40
41 //IMPORTANT: Wait for TM_SCT connection (otherwise the node tries to send
42 //script before the connection is established)
42 ROS_INFO_STREAM("NAIVE: Wait for TM_SCT connection...");  

43 ros::Duration(1).sleep();
44
45 // Create the client to request the service SendScript
46 ros::ServiceClient client = nh.serviceClient<tm_msgs::SendScript>("tm_driver/send_script");
47 // Create the request with the desired commands to test
48 tm_msgs::SendScript srv;
49 srv.request.id = "obs";
50 srv.request.script = cmd9;
51
52 // Service call
53 if (client.call(srv))
54 {
55     if (srv.response.ok) ROS_INFO_STREAM("Sent script to robot");
56     else ROS_WARN_STREAM("Sent script to robot , but response not yet ok ");
57 } else{
58     ROS_ERROR_STREAM("Error send script to robot");
59 }
60
61 return 0;

```

Listing 6.2: send_script_node.cpp

Si fa notare che nel Listato 6.2 sono presenti alcuni comandi già pronti per essere eseguiti. In particolare se si lancia il file senza modifiche il comando che viene inviato al manipolatore è quello riportato alla riga 35, il quale fa ruotare la base (giunto shoulder_1_joint) di 270° in senso orario.

6.2 loop_trajectory

Questo programma permette di testare la gestione degli ostacoli da noi implementata quando il robot esegue una traiettoria predefinita in ROS che si limita a far ruotare la base del manipolatore in loop avanti e indietro di 90°. Per simulare la presenza di un ostacolo l'utente deve mandare in esecuzione il nodo precedentemente descritto ***Obstacle Detection Naive*** attraverso il comando riportato nel Listato 6.3:

```
$ rosrun obstacle_avoidance obstacle_detection_naive_node
```

Listing 6.3: Comando per simulare la presenza di un ostacolo

Lanciando questo nodo il TM5-900 si arresta senza terminare l'esecuzione della traiettoria e torna alla posizione di home.

6.3 pick_place_trajectory

Questo programma è analogo al precedente; anche in questo caso il robot esegue una traiettoria predefinita in ROS, ma viene simulata un'operazione di pick and place eseguita in loop. Per simulare la presenza di un ostacolo va mandato in esecuzione il medesimo nodo

del programma ***loop_trajectory*** con lo stesso comando. L'effetto di tale comando è sempre quello di arrestare il manipolatore e farlo tornare alla posizione di home.

6.4 moveit_trajectory

Questo programma permette di testare la gestione degli ostacoli implementata nel presente progetto quando il robot esegue una traiettoria pianificata da MoveIt.

Il funzionamento del programma è analogo a quello del file di launch del repository GitHub [6] descritto nella Sezione 3.1.1; l'unica differenza è che in qualunque momento dell'esecuzione della traiettoria l'utente può simulare la presenza di un ostacolo mandando in esecuzione il nodo ***Obstacle Detection Naive*** come per i programmi ***loop_trajectory*** e ***pick_place_trajectory***. Anche in questo caso il robot interrompe l'esecuzione della traiettoria e torna alla posizione di home.

Capitolo 7

Sviluppi futuri

In questo capitolo sono riportati i problemi aperti riscontrati nella realizzazione del presente progetto e gli eventuali sviluppi futuri che possono nascere da questo lavoro.

7.1 Gestione reale degli ostacoli

In questo progetto ci si è limitati a simulare via software la presenza di un generico ostacolo e a gestirlo semplicemente fermando il robot e facendolo tornare alla posizione di home. Non è infatti stata gestita né la parte di rilevamento degli ostacoli né la ripianificazione della traiettoria. Il presente lavoro rappresenta quindi soltanto un punto di partenza per una vera e propria gestione degli ostacoli. L'idea iniziale era quella di affidarsi ad un sistema di visione esterno al manipolatore per rilevare le potenziali collisioni e ricostruire l'ambiente 3D in cui robot opera; da qui si può pensare di utilizzare MoveIt per ripianificare la traiettoria in modo che il manipolatore possa portare a termine il task assegnatogli evitando gli ostacoli.

7.2 Modbus

Come già anticipato nella Sezione 5.1.2 il principale limite della soluzione proposta nel presente progetto è che può essere utilizzata soltanto quando la traiettoria eseguita dal manipolatore è data in modalità *External Script*. Questo limite è dovuto al fatto che per utilizzare il socket TCP *External Script* su cui si basa la soluzione proposta il flusso del programma TMflow deve rimanere sul nodo Listen impedendo l'esecuzione di altre istruzioni. Di conseguenza se si vuole programmare il movimento del robot in TMflow occorre adottare un approccio diverso.

Una possibile soluzione è quella di ricorrere al protocollo di comunicazione Modbus per interfacciare il robot con il pc su cui è in esecuzione un'applicazione ROS. Questo protocollo permette infatti di inviare comandi al robot anche quando è in esecuzione un generico programma TMflow. L'idea potrebbe quindi essere quella di definire la traiettoria del manipolatore direttamente con un programma TMflow ed utilizzare Modbus per interrompere tale programma e mandarne in esecuzione un altro nel momento in cui viene rilevato un ostacolo. Si fa notare però che il nodo driver utilizzato nel presente progetto per la comunicazione robot pc non supporta Modbus. Di conseguenza per poter adottare questa soluzione alternativa il

primo problema da risolvere è capire come interfacciare il manipolatore e ROS con questo diverso protocollo.

7.3 Problemi aperti

Nel presente progetto gli obiettivi prefissati sono stati raggiunti, tuttavia nella sua realizzazione sono stati riscontrati alcuni problemi aperti che non sono stati completamente risolti. La prima problematica rimasta in sospeso è quella descritta nella Sezione 5.2 relativa all'esecuzione lenta da parte del manipolatore della traiettoria pianificata da MoveIt. Come già accennato, vista la carenza di informazioni fornite dalla documentazione ufficiale di TMflow, il problema è stato risolto adottando una soluzione temporanea che consiste nell'introduzione di un fattore di scala calcolato empiricamente. Per risolvere propriamente tale questione irrisolta è necessario approfondire il comando *PVTPoint()* in modo da comprendere il significato preciso dei suoi argomenti ed applicare poi un corretto fattore di scala. Quest'ultima considerazione nasce infatti dall'ipotesi che la motivazione della problematica sia un'incongruenza nelle unità di misura degli argomenti di tale comando rispetto ad una sua precedente implementazione.

Il secondo problema aperto (accennato nella Sezione 5.1.2) è invece relativo all'utilizzo del comando *PTP()* per riportare il robot alla posizione di home. In particolare dai test effettuati è emerso che talvolta il robot non riesce a portare a termine tale operazione arrestandosi prima di raggiungere la posa di partenza di default e segnalando l'errore riportato in Figura 7.1.

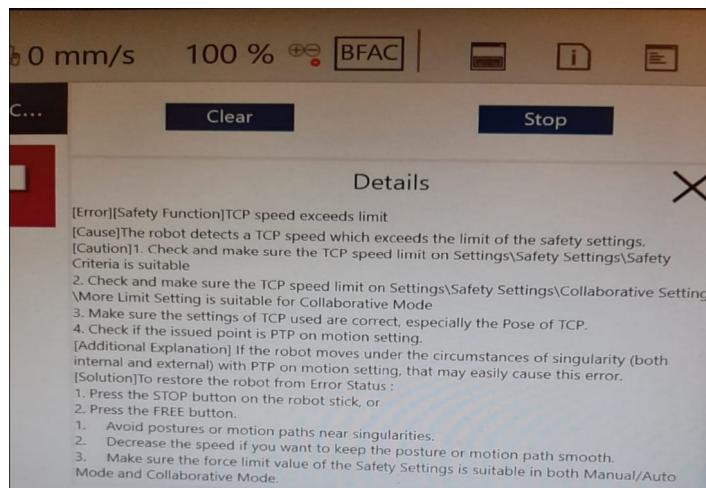


Figure 7.1: Messaggio di errore sul log di TMflow

Nello specifico si è osservato che l'errore si verifica quando per raggiungere la posizione di home partendo dalla posa attuale il manipolatore deve effettuare un movimento ampio. Tale problematica è descritta anche nel Capitolo 9.2.2 *Speed of PTP Motion* del manuale *Software Manual TMflow* [3] in cui si suggerisce di non utilizzare la modalità PTP in queste circostanze, ma di ricorrere alle altre funzioni di movimento a disposizione.

Nel presente progetto il problema è stato aggirato modificando gli argomenti *speed* e *time*

del comando *PTP()* in modo da rallentare il movimento del robot; tale procedimento è stato ripetuto fino a che non è stata raggiunta una condizione in cui l'errore non si verifica più indipendentemente dal momento in cui viene lanciato il nodo ***Obstacle Detection Naive*** nei programmi ROS implementati. Questa problematica non è stata risolta dal momento che, per una gestione reale degli ostacoli, il robot non deve tornare alla posizione di home, ma eseguire la traiettoria ripianificata.

7.4 ROS2

Per il presente progetto si è scelto di utilizzare ROS1, in quanto come già anticipato nel Capitolo 3, il repository GitHub di partenza [6] risulta essere più completo e aggiornato rispetto all'analoga versione per ROS2. Tuttavia alla luce del fatto che ROS1 è destinato a diventare obsoleto, è opportuno replicare il progetto e proseguirne lo sviluppo utilizzando ROS2.

Appendice A

Simulazione con MoveIt e Gazebo

Nelle fasi iniziali del progetto è stato testato anche il repository GitHub [13] che permette di pianificare una generica traiettoria con MoveIt e di eseguirla sul robot simulato in Gazebo. Questo repository è pensato per la versione Kinetic di ROS1, ma il suo corretto funzionamento è stato testato anche su ROS Melodic.

Tra le funzionalità offerte dal repository vi è un file di launch che permette di simulare il robot TM5-700 utilizzando il seguente comando:

```
$ rosrun tm_gazebo tm700_gazebo_moveit.launch
```

Listing A.1: Comando per la simulazione del TM5-700

Il funzionamento di questo programma è analogo a quello descritto nella Sezione 3.1.1 con la differenza che la traiettoria pianificata da MoveIt (Figura A.1) non è eseguita dal robot reale ma simulata su Gazebo come visibile in Figura A.2.

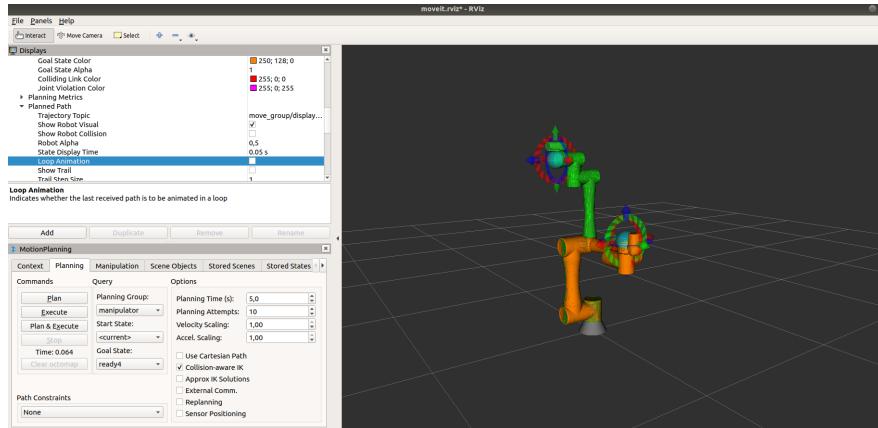


Figure A.1: Pianificazione su MoveIt

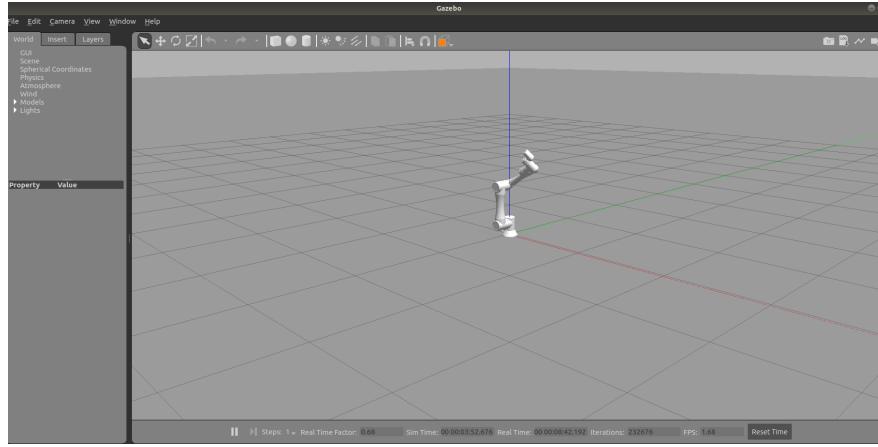


Figure A.2: Esecuzione su Gazebo

Questo file di launch è stato realizzato soltanto per il robot TM5-700, ma esplorando il contenuto del repository si è visto che il presente programma può essere adattato facilmente per la simulazione del TM5-900 in quanto il progetto GitHub contiene anche il modello di questa versione del manipolatore. In particolare è sufficiente creare il nuovo file di launch *NOME_PROGETTO/src/tm_gazebo/launch/tm900_gazebo_moveit.launch* il cui contenuto è mostrato nel Listato A.2.

```
<launch>
  <!-- Modified for tm5-900 -->
  <include file="$(find tm_gazebo)/launch/tm900.launch"/>

  <!-- Remap oint_trajectory_action -->
  <remap from="/joint_trajectory_action" to="/arm_controller/
    follow_joint_trajectory"/>
  <include file="$(find tm900_moveit_config)/launch/move_group.launch">
    <arg name="publish_monitored_planning_scene" value="true" />
  </include>
  <include file="$(find tm900_moveit_config)/launch/moveit_rviz.launch">
    <arg name="config" value="true"/>
  </include>
</launch>
```

Listing A.2: tm900_gazebo_moveit.launch

Per lanciare il file di launch creato appositamente per il manipolatore TM5-900 occorre quindi utilizzare il seguente comando:

```
$ roslaunch tm_gazebo tm900_gazebo_moveit.launch
```

Listing A.3: Comando per la simulazione del TM5-900

Appendice B

Codice gestione ostacoli con traiettoria pianificata

In questa appendice viene riportato e brevemente commentato il codice utilizzato per la gestione degli ostacoli con traiettoria pianificata da MoveIt. Nello specifico sono riportati i seguenti codici:

- *moveit_trajectory.launch*
- *obstacle_avoidance_naive_moveit_node.cpp*
- *obstacle_detection_naive_node.cpp*
- *obstacle_detected.msg*

moveit_trajectory.launch

Questo file di launch è stato ottenuto modificando il file di launch fornito dal repository GitHub [6] per l'esecuzione di una traiettoria pianificata da MoveIt. È stato sufficiente semplificare il codice, riducendolo all'essenziale, e aggiungere il nodo *obstacle_avoidance_naive_moveit_node* per la gestione degli ostacoli.

Si fa notare che in questo caso l'indirizzo IP del robot è specificato all'interno del Listato B.1; pertanto, nel caso in cui sia necessario modificarlo, occorre correggerlo all'interno del file di launch.

```
<!-- Launch file: Obstacle avoidance (stop+home) for MoveIt trajectory-->
<launch>
    <!-- The planning and execution components of MoveIt! configured to run -->
    <!-- using the ROS-Industrial interface. -->
    <!-- Non-standard joint names:
        - Create a file tm5_900_moveit_config/config/joint_names.yaml
          controller_joint_names: [joint_1, joint_2, ... joint_N]
        - Update with joint names for your robot (in order expected by rbt
          controller)
        - and uncomment the following line: -->
    <rosparam command="load" file="$(find tm5_900_moveit_config)/config/
      joint_names.yaml"/>
```

```

<!-- load the robot_description parameter before launching ROS-I nodes -->
<include file="$(find tm5_900_moveit_config)/launch/planning_context.launch
  " >
  <arg name="load_robot_description" value="true" />
</include>
<!-- run the "real robot" interface nodes -->
<!-- - this typically includes: robot_state, motion_interface, and
    joint_trajectory_action nodes -->
<!-- - replace these calls with appropriate robot-specific calls or
    launch files -->
<include file="$(find tm_driver)/launch/tm5_900_bringup.launch" >
  <arg name="robot_ip" value="169.254.77.215"/>
</include>
<include file="$(find tm5_900_moveit_config)/launch/move_group.launch">
  <arg name="publish_monitored_planning_scene" value="true" />
</include>
<include file="$(find tm5_900_moveit_config)/launch/moveit_rviz.launch">
  <arg name="rviz_config" value="$(find tm5_900_moveit_config)/launch/
    moveit.rviz"/>
</include>
<!-- New node for obstacle detection-->
<node name="obstacle_avoidance_naive_moveit_node" pkg="obstacle_avoidance"
  type="obstacle_avoidance_naive_moveit_node" output="screen" />
</launch>

```

Listing B.1: moveit_trajectory.launch

obstacle_avoidance_naive_moveit_node.cpp

Nel Listato B.2 è riportato il codice del nodo *obstacle_avoidance_naive_moveit_node* che permette la gestione degli ostacoli quando il robot esegue la traiettoria pianificata da MoveIt. Tale nodo, alla ricezione del messaggio di *ObstacleDetected* inviato dal nodo *obstacle_detection_naive_node*, interrompe il movimento corrente del manipolatore e lo fa tornare alla posizione di home.

```

1 // Node for obstacle avoidance with trajectory planned in Moveit
2 // when ObstacleDetected message is received
3 // the trajectory stops and the robot returns to home state
4
5 #include <ros/ros.h>
6 #include <std_msgs/String.h>
7
8 #include <iostream>
9 #include <cstdlib>
10
11 #include "tm_msgs/SendScript.h"
12
13 // Custom message ObstacleDetected
14 #include "tm_msgs/ObstacleDetected.h"
15
16 #include <boost/bind.hpp>
17
18 // Callback executed when ObstacleDetected message is received

```

```

19 void ObstacleDetectedCallback(ros::NodeHandle &nh, const tm_msgs::
20   ObstacleDetected::ConstPtr& msg)
21 {
22   std::string flag = msg->obstacle_detected ? "true" : "false";
23   ROS_INFO_STREAM("MOVEIT: Received message ObstacleDetected: " << flag );
24
25   ROS_INFO_STREAM("MOVEIT: Send stop + home");
26
27   // Create the client to request the service SendScript
28   ros::ServiceClient client = nh.serviceClient<tm_msgs::SendScript>("tm_driver/send_script");
29
30   // Create the request with the desired commands to handle obstacle
31   // detection
32   std::string cmd = "StopAndClearBuffer()\r\nPTP(\"JPP
33   \",0,0,0,0,0,0,3,200,0,false)"; //stop+home
34   std::string cmd1 = "StopAndClearBuffer()"; //stop
35
36   tm_msgs::SendScript srv;
37
38   srv.request.id = "Obs";
39   srv.request.script = cmd;
40
41   // Service call
42   if (client.call(srv))
43   {
44     if (srv.response.ok) ROS_INFO_STREAM("MOVEIT: Sent script to robot");
45     else ROS_WARN_STREAM("Sent script to robot , but response not yet ok
46   ");
47   }
48 }
49
50 int main(int argc, char **argv)
51 {
52   //Node initialization
53   ros::init(argc, argv, "obstacle_avoidance_naive_moveit_node");
54   ros::NodeHandle nh;
55
56   // Create subscriber to receive ObstacleDetected messages
57   ros::Subscriber sub = nh.subscribe<tm_msgs::ObstacleDetected>("tm_driver/
58     obstacle_detected", 5, boost::bind(&ObstacleDetectedCallback, boost::ref(
59       nh), _1));
60   // Keep the node active to receive messages
61   ros::spin();
62
63   return 0;
64 }
```

Listing B.2: obstacle_avoidance_naive_moveit_node.cpp

obstacle_detection_naive_node.cpp

Nel Listato B.3 è riportato il codice del nodo *obstacle_detection_naive_node* che deve essere mandato in esecuzione per simulare la presenza di un ostacolo nell'ambiente di lavoro del manipolatore.

```
1 // Node to simulate an obstacle detection
2 // Run this node to send a message of ObstacleDetected
3
4 #include <ros/ros.h>
5 #include <std_msgs/String.h>
6
7 #include <iostream>
8 #include <cstdlib>
9
10 // Custom message ObstacleDetected
11 #include "tm_msgs/ObstacleDetected.h"
12
13 int main(int argc, char **argv)
14 {
15     //Node initialization
16     ros::init(argc, argv, "obstacle_detection_naive_node");
17     ros::NodeHandle nh;
18
19     // Create publisher for ObstacleDetected message
20     ros::Publisher pub = nh.advertise<tm_msgs::ObstacleDetected>("tm_driver/
21         obstacle_detected", 1);
22
23     // IMPORTANT: Wait for the publisher to be ready (otherwise message is lost
24     // )
25     while (pub.getNumSubscribers() < 1) {
26         ros::WallDuration sleep_t(0.5); //seconds
27         sleep_t.sleep();
28         ROS_INFO_STREAM("OB DETECTED: Wait for publisher");
29     }
30
31     //Create ObstacleDetected message
32     tm_msgs::ObstacleDetected msg;
33     msg.obstacle_detected = true;
34
35     //Publish ObstacleDetected message
36     pub.publish(msg);
37     ROS_INFO_STREAM("OB DETECTED: Sent message obstacle_detected");
38
39     //IMPORTANT: Wait to give the node time to send the message (otherwise
40     // message is lost)
41     ros::Duration(1).sleep(); //seconds
42
43     return 0;
44 }
```

Listing B.3: *obstacle_detection_naive_node.cpp*

obstacle_detected.msg

Nel Listato B.4 è infine riportato il file msg che definisce il messaggio custom *obstacle_detected* inviato dal nodo *obstacle_detection_naive_node* per simulare la presenza di un ostacolo.

```
# Custom naive message of obstacle detected
bool obstacle_detected      # flag
```

Listing B.4: obstacle_detected.msg

Appendice C

Codice gestione ostacoli con traiettoria predefinita

In questa appendice viene riportato e brevemente commentato il codice utilizzato per la gestione degli ostacoli con traiettoria predefinita in ROS che simula un'operazione di pick and place eseguita in loop. Nello specifico per la sua implementazione sono necessari i seguenti codici:

- *pick_place_trajectory.launch*
- *obstacle_avoidance_naive_pick_place_node.cpp*
- *pick_place_trajectory_node.cpp*
- *obstacle_detection_naive_node.cpp*
- *obstacle_detected.msg*

Gli ultimi due codici elencati sono gli stessi utilizzati per la gestione degli ostacoli con traiettoria pianificata da MoveIt e riportati nell'Appendice B.

pick_place_trajectory.launch

Il Listato C.1 contiene il file di launch che fa eseguire una traiettoria di pick and place al robot, tenendo in considerazione l'eventuale presenza di ostacoli nell'ambiente di lavoro del manipolatore. Tale file manda in esecuzione il nodo driver, il nodo *pick_place_trajectory_node* e il nodo *obstacle_avoidance_naive_pick_node*.

Si fa notare che anche in questo caso l'indirizzo IP del TM5-900 è indicato all'interno del file di launch.

```
<?xml version="1.0"?>
<!-- Launch file: Obstacle avoidance (stop+home) for pick and place
     trajectory -->
<launch>
  <node name="tm_driver" pkg="tm_driver" type="tm_driver" output="screen">
    <param name="robot_ip_address" type="str" value="169.254.77.215" />
  </node>
```

```

<node name="pick_place_trajectory_node" pkg="obstacle_avoidance" type="pick_place_trajectory_node" respawn="false" output="screen" />
<node name="obstacle_avoidance_naive_pick_node" pkg="obstacle_avoidance" type="obstacle_avoidance_naive_pick_node" respawn="false" output="screen" />
</launch>

```

Listing C.1: pick_place_trajectory.launch

obstacle_avoidance_naive_pick_place_node.cpp

Nel Listato C.2 è mostrata l'implementazione del nodo *obstacle_avoidance_naive_pick_node* per la gestione degli ostacoli quando il robot esegue una generica traiettoria ciclica inviata in modalità *External Script*.

```

1 // Node for obstacle avoidance with pick and place trajectory
2 // when ObstacleDetected message is received
3 // the trajectory stops and the robot returns to home state
4
5 #include <ros/ros.h>
6 #include <std_msgs/String.h>
7
8 #include <iostream>
9 #include <cstdlib>
10
11 #include "tm_msgs/SendScript.h"
12 // Custom message ObstacleDetected
13 #include "tm_msgs/ObstacleDetected.h"
14
15 #include "ros>this_node.h"
16 #include "ros/master.h"
17 #include <boost/bind.hpp>
18
19 // Callback executed when ObstacleDetected message is received
20 void ObstacleDetectedCallback(ros::NodeHandle &nh, const tm_msgs::ObstacleDetected::ConstPtr& msg)
21 {
22     std::string flag = msg->obstacle_detected ? "true" : "false";
23     ROS_INFO_STREAM("PICK: Received message ObstacleDetected: " << flag );
24
25     // First command needed to stop immediately the trajectory
26     ROS_INFO_STREAM("PICK: First clear");
27
28     // Create the client to request the service SendScript
29     ros::ServiceClient client = nh.serviceClient<tm_msgs::SendScript>("tm_driver/send_script");
30     // Create the request with the commands to stop the trajectory
31     std::string cmd2 = "StopAndClearBuffer()";
32     tm_msgs::SendScript srv;
33     srv.request.id = "obs1";
34     srv.request.script = cmd2;
35
36     // Service call
37     if (client.call(srv))

```

```

38     {
39         if (srv.response.ok) ROS_INFO_STREAM("PICK: Sent script to robot");
40         else ROS_WARN_STREAM("Sent script to robot , but response not yet ok
41     ");
42     }
43     {
44         ROS_ERROR_STREAM("PICK: Error send script to robot");
45     }
46
47     //IMPORTANT: Wait for pick_place_trajectory_node to stop
48     ROS_INFO_STREAM("PICK: Wait for pick_place_trajectory_node to stop");
49     bool waiting = true;
50     std::string node = "/pick_place_trajectory_node";
51     while(waiting) {
52         ros::V_string v;
53         ros::master::getNodes(v);
54         waiting = false;
55         for (int i=0; i < v.size(); i++) {
56             if (v[i].compare(node)==0) {
57                 ROS_INFO_STREAM("PICK: still exists " << v[i] <<"node... wait
58 ...");
59                 waiting = true;
60             }
61         }
62         ros::Duration(0.5).sleep();
63     }
64     ROS_INFO_STREAM("PICK: pick_place_trajectory_node terminated, send clear
and home...");
65
66     // Create the client to request the service SendScript
67     ros::ServiceClient client1 = nh.serviceClient<tm_msgs::SendScript>("tm_
driver/send_script");
68     // Create the request with the commands to stop the trajectory and return
to home state
69     std::string cmd = "StopAndClearBuffer()\r\nPTP(\"JPP
\",0,0,0,0,0,5,500,0,false)"; //stop+home
70     tm_msgs::SendScript srv1;
71     srv1.request.id = "obs2";
72     srv1.request.script = cmd;
73
74     // Service call
75     if (client1.call(srv1))
76     {
77         if (srv1.response.ok) ROS_INFO_STREAM("PICK: Sent script to robot");
78         else ROS_WARN_STREAM("Sent script to robot , but response not yet ok ")
79     }
80     else{
81         ROS_ERROR_STREAM("PICK: Error send script to robot");
82     }
83 }
84
85 int main(int argc, char **argv)
86 {
87     //Node initialization
88     ros::init(argc, argv, "obstacle_avoidance_naive_pick_node");

```

```

87     ros::NodeHandle nh;
88
89     // Create subscriber to receive ObstacleDetected messages
90     ros::Subscriber sub = nh.subscribe<tm_msgs::ObstacleDetected>("tm_driver/
91         obstacle_detected", 5, boost::bind(&ObstacleDetectedCallback, boost::ref(
92             nh), _1));
93     // Keep the node active to receive messages
94     ros::spin();
95
96     return 0;
97 }
```

Listing C.2: obstacle_avoidance_naive_pick_place_node.cpp

pick_place_trajectory_node.cpp

Nel Listato C.3 è infine riportato il codice del nodo ***pick_place_trajectory_node.cpp*** che si occupa di inviare al manipolatore comandi *External Script* affinché esso esegua in loop una traiettoria di pick and place. Nello specifico quest'ultima prevede che il manipolatore, partendo dalla posizione di home, raggiunga il punto di pick (posizione 1), si sollevi (posizione 2) per poi ruotare (raggiungendo la posizione 3) e infine scendere per il place (posizione 4); da qui il TM5-900 si innalza e ruota al fine di ritornare alla posizione 2, passando per la 3. Dopodiché il ciclo si ripete partendo direttamente dalla posizione 2. La traiettoria descritta può essere ottenuta in diversi modi utilizzando i diversi comandi in modalità *External Script* specificati nella Sezione 1.2.2.3.

Nel presente listato si è scelto di utilizzare il comando *Move_PTP()* che permette di specificare esclusivamente gli spostamenti relativi che il robot deve eseguire per muoversi da una posizione all'altra. Fa eccezione soltanto il movimento iniziale dalla posizione di home alla posizione 1 che è stato pianificato con MoveIt utilizzando l'applicazione riportata nell'Appendice B per ottenere i corrispondenti *PVTPoint*.

```

1 //Node for pick and place trajectory with external script
2 #include <ros/ros.h>
3 #include <std_msgs/String.h>
4
5 #include <iostream>
6 #include <cstdlib>
7 // Custom message ObstacleDetected
8 #include "tm_msgs/ObstacleDetected.h"
9
10 #include "tm_msgs/SendScript.h"
11 #include "tm_msgs/StaResponse.h"
12 // Callback executed when ObstacleDetected message is received to kill the
13 // node
14 void KillCallback(bool* flag_loop_p,ros::NodeHandle &nh, const tm_msgs::
15     ObstacleDetected::ConstPtr& msg){
16     std::string flag = msg->obstacle_detected ? "true" : "false";
17     ROS_INFO_STREAM("PICK: Received message ObstacleDetected: " << flag );
18
19     // Kill the node by setting the flag_loop variable to false
20     ROS_INFO_STREAM("PICK: pick_place_trajectory_node killing...");
```

```

20     *flag_loop_p = false;
21     //ros::shutdown();
22 }
23 // Callback executed when a TM_STA message is received to implement a loop
24 void StaResponseCallback(bool* flag_done_p, const tm_msgs::StaResponse::ConstPtr& msg)
25 {
26     //Check if it is the TMSTA message in response to the QueueTag of the
27     // trajectory commands
28     ROS_INFO_STREAM("TMSTA message: subcmd is = " << msg->subcmd << ", subdata
29     is " << msg->subdata);
30     std::string cmd = "01"; //QueueTag
31     std::string data = "01,true"; // Tag number = 01, Status = true
32
33     if((msg->subcmd.compare(cmd)==0)&&(msg->subdata.compare(data)==0)) {
34         ROS_INFO_STREAM("PICK: Trajectory executed!");
35         *flag_done_p = true;
36     }
37
38 int main(int argc, char **argv)
39 {
40     //Flag initialization
41     bool flag_loop = true; // loop trajectory on
42     bool flag_done = false; // trajectory commands still not executed
43
44     //Node initialization
45     ros::init(argc, argv, "pick_place_trajectory_node");
46     ros::NodeHandle nh;
47
48     //IMPORTANT: Wait for TM_SCT connection (otherwise the node tries to send
49     // script before the connection is established)
50     ROS_INFO_STREAM("PICK: Wait for TM_SCT connection...");
51     ros::Duration(1).sleep(); //seconds
52
53     // Create subscriber to receive ObstacleDetected messages
54     ros::Subscriber sub = nh.subscribe<tm_msgs::ObstacleDetected>("tm_driver/
55     obstacle_detected", 5, boost::bind(&KillCallback,&flag_loop, boost::ref(nh
56     ), _1));
57     // Create subscriber to receive StaResponse messages
58     ros::Subscriber sub2 = nh.subscribe<tm_msgs::StaResponse>("tm_driver/
59     sta_response", 1000, boost::bind(&StaResponseCallback,&flag_done,_1));
60
61     // Create the client to request the service SendScript
62     ros::ServiceClient client = nh.serviceClient<tm_msgs::SendScript>("tm_
63     driver/send_script");
64     // Create the request with the commands for the pick and place trajectory
65     tm_msgs::SendScript srv;
66
67     // 1: Pick and place trajectory with PTP
68     std::stringstream ss;
69     ss << "PTP(\"JPP\",-10.88,23.75,86.66,-18.56,92.41,0,5,200,0,false)\r\n";
70     //1 pick
71     ss << "PTP(\"JPP\",-10.88,3.23,86.66,-4.20,92.41,0,5,200,0,false)\r\n"; //2
72     ss << "PTP(\"JPP\",28.38,3.23,86.66,-4.20,92.41,0,5,200,0,false)\r\n"; //3

```

```

66 ss << "PTP(\"JPP\",28.38,23.75,86.66,-18.56,92.41,0,5,200,0,false)\r\n"; //4 place
67 ss << "PTP(\"JPP\",28.38,3.23,86.66,-4.20,92.41,0,5,200,0,false)\r\n"; //3
68 ss << "PTP(\"JPP\",-10.88,3.23,86.66,-4.20,92.41,0,5,200,0,false)\r\n"; //2
69 ss << "QueueTag(1)";
70 std::string cmd2 = ss.str();
71 //ROS_INFO_STREAM("PICK Cmd: " << cmd2);
72
73 // 2: Pick and place trajectory with PVTPoint (from Moveit) and Move_PTP
74 std::stringstream ss3;
75 ss3 << "PVTEnter(0)\r\n";
76 ss3 << "PVTPoint(-1.08811,2.37447,8.66625,-1.85596,9.24145,-0.00043,
77 -19.42402,42.37978,154.68622,-33.13028,164.95078,-0.00436,0.09520)\r\n";
78 ss3 << "PVTPoint(-2.17636,4.74882,17.33262,-3.71210,18.48290,-0.00067,
79 -31.63423,69.02029,251.92411,-53.95644,268.64111,-0.00710,0.03969)\r\n";
80 ss3 << "PVTPoint(-3.26460,7.12316,25.99900,-5.56825,27.72435,-0.00092,
81 -39.25025,85.63710,312.57549,-66.94659,333.31715,-0.00881,0.03035)\r\n";
82 ss3 << "PVTPoint(-4.35284,9.49751,34.66537,-7.42439,36.96580, -0.00116,
83 -45.10179, 98.40412, 359.17509,-76.92717,383.00898,-0.01012,0.02552)\r\n";
84 ss3 << "PVTPoint(-5.44108,11.87186,43.33174,-9.28053,46.20725, -0.00141,
85 -46.88508,102.29494,373.37658,-79.96881,398.15284,-0.01052,0.02288)\r\n";
86 ss3 << "PVTPoint(-6.52932,14.24621,51.99812,-11.13667,55.44870,-0.00165,
87 -43.49639,94.90143,346.39028,-74.18895,369.37581, -0.00976,0.02355)\r\n";
88 ss3 << "PVTPoint(-7.61756,16.62056,60.66449,-12.99281,64.69015,-0.00189,
89 -37.26279,81.30081, 296.74802,-63.55670,316.43941,-0.00836, 0.02669)\r\n";
90 ss3 << "PVTPoint(-8.70580,18.99491,69.33086,-14.84895,73.93160,-0.00214,
91 -30.57855,66.71699, 243.51705,-52.15583,259.67618, -0.00686,0.03225)\r\n";
92 ss3 << "PVTPoint(-9.79404,21.36925,77.99723,-16.70509,83.17305,-0.00238,
93 -19.41983,42.37064, 154.65286,-33.12314,164.91521,-0.00436, 0.03970)\r\n";
94 ss3 << "PVTPoint(-10.88228,23.74360,86.66361,-18.56123,92.41450,-0.00263,
95 0.00000,0.00000, 0.00000,0.00000,0.00000, 0.09520)\r\n"; //1 pick
96 ss3 << "PVTExit()\r\n";
97 ss3 << "Move_PTP(\"JPP\",0,-20.52,0,14.36,0,0,10,200,0,false)\r\n"; // 2
98 ss3 << "Move_PTP(\"JPP\",39.26,0,0,0,0,0,10,200,0,false)\r\n"; // 3
99 ss3 << "Move_PTP(\"JPP\",0,20.52,0,-14.36,0,0,10,200,0,false)\r\n"; // 4
100 ss3 << "place
101 if (client.call(srv))
102 {
103     if (srv.response.ok) ROS_INFO_STREAM("PICK: Sent script to robot");
104     else ROS_WARN_STREAM("Sent script to robot , but response not yet ok ");
105 }
106 else
107 {
108     ROS_ERROR_STREAM("PICK: Error send script to robot");
109     return 1;

```

```

110 }
111
112 // Create the request with the commands for the loop pick and place
113 // trajectory
114 tm_msgs::SendScript srv1;
115 std::stringstream ss4;
116 ss4 << "Move_PTP(\"JPP\",0,20.52,0,-14.36,0,0,10,200,0,false)\r\n"; // 2->1
117 ss4 << "Move_PTP(\"JPP\",0,-20.52,0,14.36,0,0,10,200,0,false)\r\n"; // 2
118 ss4 << "Move_PTP(\"JPP\",39.26,0,0,0,0,0,10,200,0,false)\r\n"; // 3
119 ss4 << "Move_PTP(\"JPP\",0,20.52,0,-14.36,0,0,10,200,0,false)\r\n"; // 4
120 // place
121 ss4 << "Move_PTP(\"JPP\",0,-20.52,0,14.36,0,0,10,200,0,false)\r\n"; // 3
122 ss4 << "Move_PTP(\"JPP\",-39.26,0,0,0,0,0,10,200,0,false)\r\n"; // 2
123 ss4 << "QueueTag(1)";
124 std::string cmd4 = ss4.str();
125 //ROS_INFO_STREAM("PICK: Loop commands " << cmd4);
126 srv1.request.id = "pick";
127 srv1.request.script = cmd4;
128
129 // Pick and place trajectory loop
130 while(flag_loop) {
131     if(flag_done){
132         //Send new trajectory command
133         ROS_INFO_STREAM("PICK: New trajectory command");
134         // Check for messages
135         ros::spinOnce();
136         if (client.call(srv1))
137         {
138             if (srv.response.ok) ROS_INFO_STREAM("PICK: Sent script to
139             robot");
140             else ROS_WARN_STREAM("Sent script to robot , but response not
141             yet ok ");
142         }
143         else
144         {
145             ROS_ERROR_STREAM("PICK: Error send script to robot");
146             return 1;
147         }
148         flag_done = false;
149     } else {
150         // Wait TM_STA message of trajectory executed
151         ROS_INFO_STREAM("PICK: Wait trajectory executed");
152         // Check for messages
153         ros::spinOnce();
154         ros::Duration(0.5).sleep();
155     }
156     // Check for messages
157     ros::spinOnce();
158 }
159
160 // Node shutdown for obstacle detection
161 ROS_INFO_STREAM("PICK: loop trajectory killed");
162
163 return 0;
164 }
```

Listing C.3: pick_place_trajectory_node.cpp

Bibliografia

- [1] “Robot collaborativi omron brochure.” https://github.com/GMCristina/ROS_driver_tm5_900/blob/main/src/documents/tm_collaborative_robot_brochure.pdf.
- [2] “Datasheet robot collaborativi.” https://github.com/GMCristina/ROS_driver_tm5_900/blob/main/src/documents/collaborative_robots_datasheet.pdf.
- [3] “Manuale software tmflow.” https://github.com/GMCristina/ROS_driver_tm5_900/blob/main/src/documents/tm_flow_software_manual.pdf.
- [4] “Manuale di installazione hardware.” https://github.com/GMCristina/ROS_driver_tm5_900/blob/main/src/documents/hw_installation_manual.pdf.
- [5] “Manuale nodo listen.” https://github.com/GMCristina/ROS_driver_tm5_900/blob/main/src/documents/tm_expression_editor_and_listen_node_manual.pdf.
- [6] “Github repository del driver ros1.” https://github.com/TechmanRobotInc/tmr_ros1.
- [7] “Ros wiki.” <http://wiki.ros.org/>.
- [8] “Github repository del driver ros2.” https://github.com/TechmanRobotInc/tmr_ros2.
- [9] “Github repository del progetto.” .
- [10] “Guida per l’installazione e la configurazione di una macchina virtuale con ubuntu 18.04.” <https://www.toptechskills.com/linux-tutorials-courses/how-to-install-ubuntu-1804-bionic-virtualbox/>.
- [11] “Guida per l’installazione e la configurazione di ros melodic su ubuntu 18.04.” <http://wiki.ros.org/melodic/Installation/Ubuntu>.
- [12] “Guida per importare il transmit file.” https://github.com/TechmanRobotInc/TM_Export.
- [13] “Github repository del driver ros1 per la simulazione.” <https://github.com/viirya/ros-driver-techman-robot>.