Linee Guida Programmazione in ROS per ARS Control

Davide Ferrari, Andrea Pupa
Università di Modena e Reggio Emilia

Lo scopo di questo documento è quello di fissare delle linee guida per la programmazione in ROS all'interno del laboratorio di ricerca ARS Control.

Imporre che tutti i componenti di un gruppo di ricerca seguino delle linee comuni per l'attività di programmazione porta con sé molteplici vantaggi. In primis tutti coloro che lavorano all'interno del gruppo stesso sanno esattamente come approcciarsi ad un codice creato da un proprio collega, semplificando enormemente il lavoro di ricerca/estensione. In secondo luogo, anche i neofiti della programmazione o del sistema ROS si ritrovano ad avere un schema mentale ben definito che permette loro di velocizzare l'apprendimento iniziale.

Il concetto FONDAMENTALE su cui si basa questa guida è quindi uno: siamo un gruppo, aiutiamoci a vicenda. Scrivere un codice prestante che funziona correttamente non fa di te un buon lavoratore. Creare, invece, un algoritmo meno efficiente ma che più facilmente si presta al riuso da parte dei propri colleghi migliora di gran lungo le prestazioni di tutto il gruppo.

Si tenga presente che questo documento non vuole assolutamente andare a sostituire i tutorial di ROS o quelli di C++, quindi la guida darà per scontato che il lettore sia già familiare con questi due strumenti.

Indice

1 Pacchetti ROS

Dopo aver definito bene gli scopi del proprio progetto e aver capito cosa bisogna implementare, la prima cosa che bisogna fare con il framework ROS è di creare il proprio pacchetto. Diventa quindi scontato andare a definire le prime linee guida proprio per questa fase.

- **Ogni pacchetto deve contenere un solo algoritmo:**
  Un atteggiamento comune è quello di creare un pacchetto generale in cui vengono inseriti tutti i codici sviluppati (es. pacchetto "Tesi" contenente il codice per fare "skeleton tracking", "inquadrare un marker", "pianificare la traiettoria di un robot"). Questo, ovviamente, rende il pacchetto assolutamente impossibile da riutilizzare da altri, a meno che non debbano fare esattamente la stessa cosa.

- **La struttura del pacchetto deve essere quella mostrata in Fig.1:**
  Dove i file .cpp devono essere inseriti all'interno della cartella src e bisogna creare dei launch file per eseguire i propri nodi.

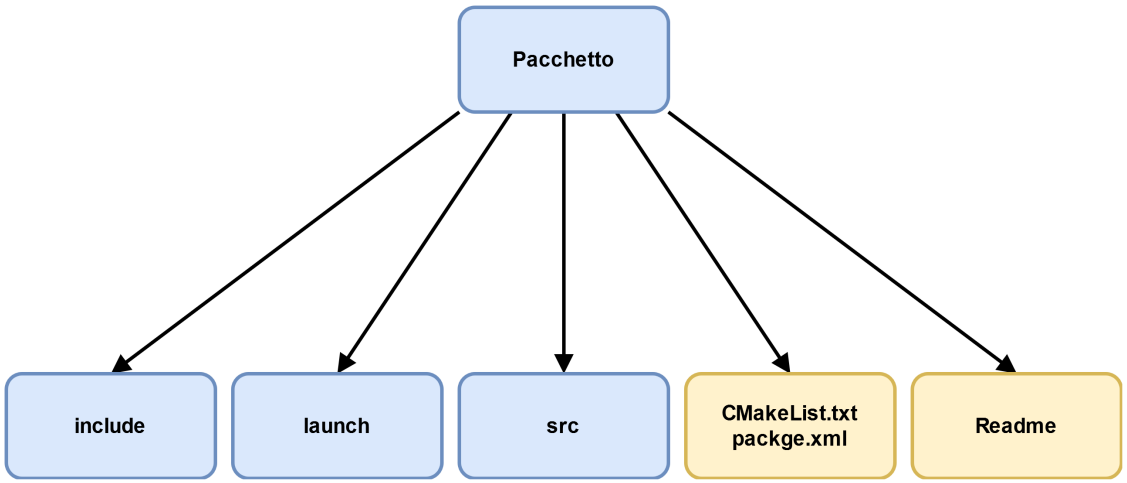

Figura 1: Struttura Pacchetto. In blu le cartelle e in giallo i file.

- **La CMakeList.txt e il package.xml devono essere puliti:**
  Bisogna evitare, soprattutto per le CMakeList.txt di mantenere tutti i commenti superflui che derivano dal template originale (es. nel template è commentata la sezione per generare una libreria, se non la stai generando elimina quella parte).

- **Il README.md deve contenere TUTTE le dipendenze e le versioni:** Molto spesso i Readme sono assenti o a volte scarni (es. questo pacchetto fa questo, fine.). Al di là del fatto che nominando in maniera adeguata il pacchetto molte volte si intuisce già la funzione che svolge, il senso del readme è quello di aiutare chi ottiene il pacchetto a compilarlo e ad eseguirlo sul proprio computer. Deve quindi contenere le informazioni su tutte le librerie esterne che sono state utilizzate con le relative versioni, non è strettamente necessario spiegare come installare i pacchetti esterni a meno che non sia molto problematico. Esempio pratico, nel mio pacchetto utilizzo le librerie Armadillo, che possono essere facilmente installate con una ricerca Google, quindi non è necessario inserire una guida su come compilarle. La versione utilizzata, però, può essere utile perché magari in futuro potrebbero non essere più presenti alcune funzioni e quindi il pacchetto diventa inutilizzabile con la nuova versione delle librerie.

  I file README.md sono dei file di tipo *markdown* (si rimanda alla *Mastering Markdown GitHub Guide* per approfondimenti). Si raccomanda di utilizzare questo tipo di formattazione all'interno dei propri README, in modo da mantenere uno standard di scrittura.

- **Altri File e Cartelle che possono essere presenti in un pacchetto:** Oltre agli elementi fondamentali visti finora, all'interno di un pacchetto è possibile inserire altri file e/o cartelle a seconda delle proprie esigenze. Come verrà mostrato più avanti, se si vogliono creare dei *custom_message* o *services* sarà necessario creare le cartelle *msg* e *srv* nelle quali definire i messaggi/servizi creati. Altre cartelle frequentemente utilizzate sono *config*, dove vengono inseriti file di configurazione, oppure *mesh*, *urdf*, *models* e *robots* nelle quali vengono inseriti i modelli degli oggetti visualizzati negli ambienti simulativi (*Gazebo*, *MoveIt*).

2 Scrivere un Nodo ROS (C++)

Prima di proseguire con la guida si rimanda il lettore ai tutorial ufficiali ROS, disponibili presso il link: "http://wiki.ros.org/it/ROS/Tutorials"

I seguenti paragrafi si affiancheranno ai tutorial ROS (dal 10 al 16) fornendo le indicazioni basilari su come scrivere i propri nodi uniformando così i vari codici del laboratorio. Sono stati creati a supporto di questa guida degli esempi di nodi *publisher-subscriber* e *client-server* all'interno del *simple_ws* reperibile nel drive condiviso *Software ARSControl*

2.1 Struttura Base

In questo paragrafo verrà definita la struttura base, organizzata in classi, che è stata scelta per scrivere un nodo ROS.

Ogni nodo (es. "*mio_nodo*") deve essere composto da tre elementi fondamentali:

- **header file** (mio_nodo.h), inserito nella cartella **include** in cui definisco l'interfaccia della classe con i propri metodi e attributi.

Algorithm 1: mio_nodo.h

```
1  #ifndef MIO_NODO_H
2  #define MIO_NODO_H
3  #include <ros/ros.h>
4  class Mio_Nodo {
5          public:
6                  Mio_Nodo();
7                  void spinner(void);
8          private:
9                  ros::NodeHandle nh;
10                 ros::Publisher pub;
11                 ros::Subscriber sub;
12 };
13 #endif /* MIO_NODO_H */
```

- **source file** (mio_nodo.cpp), inserito nella cartella **src** in cui definisco le implementazioni dei metodi.

Algorithm 2: mio_nodo.cpp

```
1 #include "mio_nodo/mio_nodo.h"
2 Mio_Nodo::Mio_Nodo() {      \\ costruttore
3         ...
4 }
5 void Mio_Nodo::spinner() {      \\ main
6         ...
7 }
```

- **source node** (mio_nodo_node.cpp), inserito nella cartella **src** in cui definisco e instanzio un nuovo oggetto della classe **Mio_Nodo** e richiamo la funzione **spinner**

Algorithm 3: mio_nodo_node.cpp

```
1  #include "mio_nodo/mio_nodo.h"
2  int main(int argc, char** argv) {
3          ros::init(argc, argv, "mio nodo");
4          Mio_Nodo mn;
5          ros::Rate r(10);
6          while(ros::ok()) {
7                  mn.spinner();
8                  r.sleep();
9          }
10         return 0;
11 }
```

2.2 Publisher e Subscriber

Partendo dalla struttura di base decritta, è possibile creare facilmente due nodi che sfruttano due classi importantissime per il funzionamento di ROS: **Publisher e Subscriber**.

Il Publisher ha come scopo quello di pubblicare un certo messaggio su un determinato topic, mentre il Subscriber si occupa di sottoscriversi ad un determinato topic ed eseguire una funzione ogni volta che è presente un nuovo messaggio: questa funzione viene detta **Callback**. Diventa intuitivo capire che, tramite l'ausilio di Publisher e Subscriber, è possibile abilitare la sincronizzazione e la comunicazione tra due nodi molto complessi, ognuno dei quali responsabile dell'esecuzione di un determinato processo. All'interno del workspace fornito insieme alla guida, è possibile trovare una implementazione molto semplice dei due nodi descritti: *simple_subscriber* e *simple_publisher*.

Analizziamo ora nel dettaglio la struttura di un nodo che utilizza un Subscriber:

- All'interno dell'header file dichiariamo un attributo di tipo Subscriber e una funzione di tipo void che sarà la Callback.

Algorithm 4: mio_nodo.h

```
1  #ifndef MIO_NODO_H
2  #define MIO_NODO_H
3  #include <ros/ros.h> #include <std_msgs/Float64.h>
4  class Mio_Nodo {
5          public:
6                  Mio_Nodo();
7                  void spinner(void);
8          private:
9                  void SubCallback(std_msgs::Float64::ConstPtr& msg);
       ros::NodeHandle nh;
10                 ros::Subscriber sub;
11 };
12 #endif /* MIO_NODO_H */
```

- Nel source file invece, all'interno del costruttore della nostra classe, inizializziamo il Subscriber. Inoltre definiamo ciò che deve essere svolto all'interno della Callback.

Algorithm 5: mio_nodo.cpp

```
1  #include "mio_nodo/mio_nodo.h"
2  Mio_Nodo::Mio_Nodo() {      \\ costruttore
3          sub = nh.subscribe("/number", 2, &Subscriber::SubCallback, this);
4          ...
5  }
6  void Mio_nodo::SubCallback(std_msgs::Float64::ConstPtr& msg) {
7          std::cout « "I received " « msg->data « std::endl;
8  }
9  void Mio_Nodo::spinner() {      \\ main
10         ros::spinOnce();
11         ...
12 }
```

NOTA: all'interno del main è stata chiamata la funzione ros::spinOnce() che si occupa di andare a leggere se nei topic richiesti (in questo caso solo "*/number*") sono presenti nuovi messaggi.

- il source node, invece, rimane invariato.

Il funzionamento del nodo può essere spiegato facilmente in questo modo: ad una frequenza ben definita (ros::Rate nel source node) viene chiamata la funzione spinner che controlla se ci sono dei nuovi messaggi sui topic. Se ci sono nuovi messaggi eseguo le rispettive callback, altrimenti vado avanti ed eseguo la parte di codice restante (in questo caso nulla).

2.3 Client e Server

Un altro metodo per permettere a due nodi di comunicare tra loro è quello di utilizzare una struttura Client - Server. In linea generale, il nodo Client si occupa di fare la richiesta di esecuzione di un determinato servizio. Il Server, invece, accoglie la richiesta, esegue il servizio (Callback) ed invia la risposta. La struttura Client - Server è molto simile alla struttura Publisher - Subscriber, con la grossa differenza che Client - Server è asincrona. Quindi il nodo Client rimane in attesa della risposta del Server.

Per eseguire il servizio richiesto, i nodi Client e Service hanno bisogno di un file di tipo .srv che specifica nel dettaglio come è strutturato il messaggio che viene scambiato. La struttura del file è la seguente:

datatype namedata
—
datatype namedata

Dove le prime righe indicano la tipologia e il rispettivo nome delle variabili necessarie a chiamare il servizio, mentre le ultime righe indicano le variabili che vengono restituite come in risposta. In questo caso abbiamo 1 variabile per chiamare il servizio e 1 variabile come risposta.

Nel workspace fornito è possibile trovare due nodi chiamati *simple_client* e *simple_service* che si scambiano un dato di tipo int64.

NOTA: il file srv è presente solo all'interno del nodo *simple_service* a cui sono state aggiunte le dipendenze di *message_generation* e *message_runtime* sia nella CMakeLists.txt che nel package.xml (**Guardare attentamente come sono strutturati**). Il pacchetto *simple_client*, invece, dipende dal pacchetto *simple_service* (sia package.xml che CMakeLists.txt), motivo per cui non è necessario ricreare un file .srv (**fare attenzione all'ultima riga della CMakeLists.txt. È necessaria per l'ordine di compilazione**).

2.4 Custom Message

Tutti i nodi ROS possono utilizzare anche messaggi custom, ovvero messaggi composti da una combinazione di dati o messaggi già esistenti.

La procedura per la creazione di messaggi custom è uguale alla creazione di servizi, con la differenza che è necessario creare dei file di tipo .msg solitamenti inseriti all'interno della certella msg.

Nel worspace allegato è possibile trovare i nodi *simple_publisher_custom_msg* e *simple_publisher_custom_msg*. Il messaggio custom è inserito all'interno del Publisher, mentre il Subscriber è dipendente dal Subscriber stesso.

NOTA: è possibile creare dei custum message che dipendono da altri custom message, stile matrioska.

2.5 CMakeLists.txt e package.xml

Per la scrittura delle CMakeLists.txt e dei package.xml si rimanda alle guide ufficiali ROS:

- CMakeLists.txt
- package.xml

2.6 Launchfile

I **launchfile** forniscono un modo per avviare contemporaneamente il ROS master (roscore) e vari nodi, oltre a permettere di settare altri requisiti di inizializzazione come l'impostazione dei parametri. Attraverso il comando **roslaunch** è possibile richiamare l'esecuzione di un **launchfile**, specificando il pacchetto, seguito dal nome del launchfile, oppure specificandone il percorso:

roslaunch nome_pacchetto launchfile
roslaunch /.../.../.../ launchfile

I launchfile (.launch) sono scritti utilizzando un formato XML specifico. Possono essere posizionati ovunque all'interno del package, ma è buona norma posizionarli in una cartella denominata "launch". Il contenuto di un launchfile deve essere contenuto tra una coppia di tag di avvio:

<launch> ... </launch>

Per avviare un nodo, vengono utilizzati i tag <node> e sono richiesti gli argomenti pkg, type e name nel seguente formato:

<node pkg = "..." type = "..." name = "..." ns = "..." output = "screen"/>

L'argomento "**pkg**" punta al pacchetto associato al nodo che deve essere lanciato; "**type**" si riferisce al nome del file eseguibile del nodo, mentre "**name**" è il nome che verrà visualizzato durante l'esecuzione e sovrascriverà il nome dell'eseguibile. "**ns**" permette di avviare un nodo all'interno di uno specifico **namespace**; ciò è utile quando si utilizzano più istanze dello stesso nodo. Altri argomenti opzionali possono essere trovati all'interno della ROS Wiki.

É anche possibile includere un **launchfile** all'interno di un altro **launchfile** mediante i tag <include>:

<include file = "nome_launchfile.launch"/>

A volte è necessario utilizzare una variabile locale nei launchfile. Questo può essere fatto usando il parametro "**arg**":

<arg name = "..." value = "..."/>

Un altro tag frequentemente utilizzato è <param>, il quale passa al server dei parametri di ROS il valore del parametro desiderato:

<param name="parameter_name" type="double" value="10.0"/>

In caso di parametri multipli, anzichè caricarli singolarmente è possibile creare un file di configurazione in formato "**.yaml**" da caricare nel parameter server:

<rosparam command="load" file="$(find pkg_name)/config/config_file.yaml"/>

Di seguito un esempio di launchfile:

Algorithm 6: mio_launchfile.launch

```
1 <launch>
2     <arg name="arg_1" value="valore"/>
3     <include file="$(find nome_pkg)/launch/altro_launchfile.launch"/>
4     <node pkg="nome_pkg" type="nome_nodo" name="nome">
5         <arg name="arg_2" value="$(arg arg_1)"/>
6         <param name="par_name" type="double" value="10.0"/>
7         <rosparam command="load" file="$(find
    pkg_name)/config/config_file.yaml"/>
8     </node>
9 </launch>
```

3 Strumenti Utili

3.1 .bash.rc e .bash.aliases

.bash.rc e **.bash.aliases** sono due file di configurazione di Linux (sono file nascosti quindi bisogna abilitarne la visualizzazione) situati nella **home** (*/home/user*) dell'utente.

- **.bash.rc:**

  Il file **.bash.rc** è un file di configurazione di Linux letto tutte le volte che si apre una nuova shell (terminale), con CTRL+ALT+T o cliccando sull'apposita icona. Tutti i comandi scritti all'interno di tale file vengono quindi caricati all'apertura del terminale, evitando all'utente di scriverli ogni volta.

  Lavorando con ROS è necessario effettuare il **source** del percorso di installazione ROS e del proprio workspace:

  source /opt/ros/<distro>/setup.bash
  source devel/setup.bash

  Inserendo queste due righe di codice all'interno del **.bash.rc**, ogni qualvolta si apra un terminale il source di ROS verrà fato automaticamente, evitando dunque all'utente di dover sempre scrivere tali righe su ogni nuovo terminale.

- **.bash.aliases** Il file **.bash.aliases** permette la creazione di alias Bash, i quali consentono di impostare un comando di scelta rapida al posto di un comando più lungo. Gli alias di Bash sono essenzialmente scorciatoie che possono risolvere il problema di dover ricordare comandi lunghi ed eliminare una grande quantità di lavoro sulla riga di comando. Ad esempio, è possibile impostare l'alias **cm** come scorciatoia per richiamare il comando per buildare il workspace "**cd ∼/catkin_ws && catkin_make**".

  La creazione di alias in bash è molto semplice. La sintassi è la seguente:

  alias alias_name="command_to_run"

3.2 CATKIN_IGNORE

CATKIN_IGNORE è uno strumento che impedisce al comando **catkin_make** e analoghi di buildare i file contenuti all'interno della cartella dove esso si trova. In questo modo possiamo alleggerire il nostro workspace escludendo quei package che non ci servono per nostro attuale lavoro o che hanno problemi di build o che per qualsiasi altro motivo vogliamo escludere momentaneamente dal workspace senza essere costretti a spostare o eliminare tali file.

Per utilizzare questo strumento è sufficiente creare un nuovo file vuoto, rinominarlo CATKIN_IGNORE e inserirlo all'interno della cartella che vogliamo venga ignorata.

3.3 timed_roslaunch

Il paccheto *timed_roslaunch* permette di inseride dei delay all'interno di un launchfile e ritardare quindi l'avvio di alcuni nodi. Vediamo un esempio in cui si vuole ritardare di *10 secondi* l'avvio di un altro launchfile:

Algorithm 7: timed_roslaunch

```
1 <launch>
2       <node pkg="timed_roslaunch" type="timed_roslaunch.sh"
    args="10 pkg_name nome.launch" name="name" output="screen"/>
3 </launch>
```

3.4 Visual Studio Code

Visual Studio Code è un editor di codice sorgente sviluppato da Microsoft per Windows, Linux e macOS. Include il supporto per debugging, un controllo per Git integrato, Syntax highlighting, IntelliSense, Snippet e refactoring del codice. Può essere usato con vari linguaggi di programmazione, tra cui la famiglia di linguaggi C (C, C++, C#), F#, HTML e altri linguaggi web, tra cui PHP, Java, Ruby e molti altri. Incorpora un insieme di funzioni che variano a seconda del linguaggio che si sta usando, come mostrato nella tabella seguente. Sono personalizzabili il tema dell'editor, le scorciatoie da tastiera e le preferenze. È un software libero e gratuito, anche se la versione ufficiale è sotto una licenza proprietaria.

Molte delle funzioni di Visual Studio Code non sono accessibili attraverso menu o interfacce utente, ma piuttosto attraverso una finestra di comando o un file .json, ad esempio le preferenze dell'utente. Inoltre, attraverso una serie di plugin, è possibile abilitare il supporto all'ambiente ROS, alle CMakeList e avere accesso all'intero workspace direttamente dall'editor.

Questa breve guida mostra come configurare correttamente VSCode per funzionare con ROS, sia utilizzando C++ (roscpp) che Python (rospy).

Il primo step consiste nell'installazione dei plugin (o Extension) dal Marketplace di VSCode a seconda delle proprie esigenze. Di seguito un elenco dei principali plugin da installare:

- C/C++
- C++ Intellisense
- CMakeTools
- Python
- Python for VSCode
- ROS
- ROS Package Variable
- ROS snippets

In seguito è necessario aprire il proprio workspace cliccando su file -> open folder e selezionare il proprio *catkin_ws*. Ora bisogna modificare le impostazioni di VSCode per permettergli di trovare i percorsi di installazione ROS; per fare ciò recarsi nella cartella *catkin_ws* e abilitare la visualizzazione dei file nascosti. Entrare nella cartella *.vscode* e modificare il file *c_cpp_properties.json*:

Algorithm 8: c_cpp_properties.json

```
{
    "configurations": [
        {
            "name": "Linux",
            "browse": {
                "databaseFilename": "",
                "limitSymbolsToIncludedHeaders": true
            },
            "includePath": [
                "${workspaceFolder}/devel/include",
                "/opt/ros/melodic/include",
                "/usr/include/**",
                "${workspaceFolder}/**"
            ],
            "defines": [],
            "intelliSenseMode": "gcc-x64",
            "compilerPath": "/usr/bin/g++",
            "cStandard": "c11",
            "cppStandard": "c++17"
        }
    ],
    "version": 4
}
```

Nella medesima cartella è necessario modificare anche il file *settings.json*:

Algorithm 9: settings.json

```
{
    "cmake.configureOnOpen": true,
    "python.autoComplete.extraPaths": [
        "/home/utente/catkin_ws/devel/lib/python2.7/dist-packages",
        "/opt/ros/melodic/lib/python2.7/dist-packages"
    ],
    "python.pythonPath": "/users/bin/xxx/python3"
}
```

Ora VSCode è configurato con il proprio ambiente ROS ed è possibile iniziare a scrivere codice !