Mini-Guida (incompleta) **a ROS (www.ros.org)**

Terminologia di base:

- **Package:** cartella di lavoro di base, in cui si trovano (oltre ai sorgenti dei nodi contenuti nel package) alcuni file che indicano le dipendenze del pacchetto, nome, makefile ed eseguibili
- **Stack:** insieme di Packages che forniscono una funzione di alto livello (ad esempio una libreria)
- **Node:** Eseguibile che svolge una parte della computazione nel modello publish/subscribe
- **Topic:** un nodo può pubblicare messaggi su un topic, oppure può sottoscriverlo, in modo da poter leggere i messaggi in ingresso
- **Messages:** tipo di dato utilizzato nel topic.
- **Master:** Nodo che si occupa di tenere i topic e di indirizzare i Subscribers verso i Publishers
- **Services:** Messaggi di tipo client server che si aspettano una risposta a fronte di un dato inviato

Creare Packages

1. Eseguire il comando
   `roscreate-pkg [nome del pacchetto] [dipendenze da pacchetti ROS]`
2. modificare manifest.xml per introdurre la descrizione e la licenza
3. Aggiungere all'XML le dipendenze da pacchetti esterni a ros con la riga:
   `<rosdep name="[nome dipendenza esterna]" />`

Comandi per conoscere le dipendenze di un pacchetto:

- **`rospack depends1`** (solo le dipendenze dirette)
- **`rospack depends`** (anche quelle indirette)

Compilare Packages

1. Installare tutte le dipendenze esterne con il comando:
   `rosdep install [nome package]`
2. Aggiungere al file CMakeLists.txt una riga per ogni eseguibile:
   `rosbuild_add_executable([nome eseguibile] [sorgenti da compilare...])`
3. Dare il comando:
   `rosmake [nome package] [altro package da compilare] [etc...]`
   che sfrutterà il makefile del (o dei) pacchetto per compilare il pacchetto e le dipendenze.

Utility da riga di comando

Node e Master

- Per lanciare il processo Master usare il comando:
  `roscore`
- Per conoscere quali nodi sono in esecuzione:
  `rosnode list`
- Per avere info su un nodo in esecuzione:
  `rosnode info [nome nodo]`
- Per eseguire un nuovo nodo:
  `rosrun [nome pacchetto] [nome nodo] __name:=[nome nodo alternativo]`
- Per verificare la connettività di un nodo, si può usare un comando di ping:
  `rosnode ping [nomenodo]`
- Per vedere quali nodi sono attivi, e come interagiscono con loro si può lanciare il comando:
  `rxgraph`

Topics

Per gestire i Topics da linea di comando, si può usare il comando `rostopic`, che ha diversi parametri:

- **`rostopic bw`** mostra la banda usata dal topic
- **`rostopic echo`** stampa i messaggi del topic a schermo
- **`rostopic hz`** mostra la frequenza di aggiornamento del topic
- **`rostopic list`** elenca i topic attivi
- **`rostopic pub`** pubblica un messaggio su un topic
- **`rostopic type`** mostra il tipo di topic (che tipo di messaggio pubblica)

Si può stampare in un grafico i dati che arrivano ai topic col comando

`rxplot [grafico1curva1,grafico1curva2] [grafico2curva1...]`

Services e Parameters

Oltre che col modello publish/subscribe, ROS può sfruttare il modello client/server sia per scambiare dati tra i nodi (services) sia per impostare i parametri di una applicazione (parameters) attraverso un parameter server.

Per gestire i servizi da terminale si può usare il comando rosservice, che ha diversi parametri:

- **`rosservice list`** Stampa le informazioni sui servizi attivi
- **`rosservice call`** Chiama i servizi con gli argomenti richiesti
- **`rosservice type`** mostra il tipo di servizio
- **`rosservice find`** cerca i servizi in base al tipo
- **`rosservice uri`** stampa l'uri ROSRPC

Per gestire i parametri si può usare il comando *rosparam*, che ha diversi parametri in ingresso:

- **`rosparam set`** Imposta un parametro
- **`rosparam get`** stampa il valore corrente del parametro
- **`rosparam load`** carica parametri su file
- **`rosparam dump`** salva i parametri su file
- **`rosparam delete`** cancella un parametro
- **`rosparam list`** lista i nomi dei parametri

Debug con ROS

Per debuggare con ROS si può usare la comoda console integrata.
la console può essere chiamata col comando:

`rxconsole`

Per filtrare la priorità dei messaggi, si può attivare la finestra apposita col comando

`rxloggerlevel`

RosBash

Ros fornisce comodi comandi bash-style per poter agire rapidamente sui packages:

- per spostarsi direttamente nella cartella di un pacchetto:
  `roscd [nomepacchetto]`
- per copiare un file da un pacchetto all'altro:
  `roscp [nome pacchetto] [percorso sorgente] [percorso destinazione]`
- per elencare tutti i file in un pacchetto:
  `rosls [nome pacchetto]`
- Per editare un file in un pacchetto:
  `rosed [nome pacchetto] [nome file]`

Creare messaggi e servizi

Tipi predefiniti

- Messaggi corrispondenti ai tipi di base (interi, stringhe, …) si trovano nel package `std_msgs`.
- Nei pacage dello stack `common_msgs` si trovano le definizioni di alcuni messaggi più complessi utilizzati frequentemente in applicazioni robotiche, ad esempio immagini, punti, vettori, ...
- Nel package `std_srvs` si trova la definizione di un servizio vuoto

Messaggi

1. Spostarsi nel pacchetto e creare una cartella `msg`
2. Creare un file `.msg`, che deve contenere in ogni riga un tipo e un nome campo
3. Decommentare `rosbuild_genmsg()` dal file CMakeLists.txt del pacchetto

I tipi che si possono usare sono:

- `int8`, `int16`, `int32`, `int64` (più `uint*` per gli unsigned)
- `float32`, `float64`
- `bool`
- `string`
- `time`, `duration`
- altri file `.msg`
- array, sia a lunghezza variabile`[]` che a lunghezza fissa`[C]`
- Header (è un timestamp di ROS)

Il comando da terminale per gestire i messaggi è **`rosmsg`**

- `rosmsg show [nomemessaggio]` Mostra la descrizione del messaggio
- `rosmsg users [nomemessaggio]` Trova i file che usano il messaggio
- `rosmsg md5 [nomemessaggio]` Mostra l'hash md5sum del messaggio
- `rosmsg package [nomepacchetto]` Elenca i messaggi del pacchetto
- `rosmsg packages [nomemessaggio]` Elenca i pacchetti contenenti il messaggio

Servizi

1. Spostarsi nel pacchetto e creare una cartella `srv`
2. Creare un file `.srv`
3. il file srv è composto da due parti, separate dalla linea "`---`" (senza virgolette). Ogni parte ha per ogni linea un tipo e un nome campo, come nei messaggi. La prima parte rappresenta i dati richiesti in richiesta, la seconda i dati restituiti come risposta
4. Decommentare `rosbuild_gensrv()` dal file CMakeLists.txt del pacchetto

Il comando da terminale per gestire i servizi è **rossrv**:

- `rossvr show [nomemessaggio]` Mostra la descrizione del servizio
- `rossvr users [nomemessaggio]` Trova i file che usano il servizio
- `rossvr md5 [nomemessaggio]` Mostra l'hash md5sum del servizio
- `rossvr package [nomepacchetto]` Elenca i servizi del pacchetto
- `rossvr packages` Elenca i pacchetti che contengono servizi

Esempi di publisher/subscriber e service/client: C++

Publisher

```
#include "ros/ros.h" // header base per ros
#include "std_msgs/String.h" // contiene la dichiarazione del messaggio standard String
                             // NB: per i messaggi non standard è analogo, ad esempio
                             // per il messaggio Results definito nel package Vision:
                             // #include "Vision/Results.h"
#include <sstream>

int main(int argc, char **argv)
{
	ros::init(argc, argv, "talker"); // inizializza ros e da il nome al nodo
	ros::NodeHandle n; /*crea un identificatore per il nodo. il primo identificatore
	crea anche il nodo, l'ultimo identificatore rilasciato lo distrugge*/

	/* Rende noto al nodo *master che il processo pubblicherà sul topic chatter, e la
	coda messaggi massima sarà di 1000 *messaggi.*/
	ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
	ros::Rate loop_rate(10); //indica la frequenza in hertz di ciclo del publisher
	int count = 0;
	while (ros::ok()) //ritorna falso se il nodo viene interrotto
	{
		std_msgs::String msg;
		std::stringstream ss;
		ss << "hello world " << count;
		msg.data = ss.str(); // scrive nel campo data del messaggio
		ROS_INFO("%s", msg.data.c_str()); // rimpiazzo di cout
		chatter_pub.publish(msg); // invia effettivamente il messaggio
		ros::spinOnce(); // fa in modo che i callback vengano eseguiti
		loop_rate.sleep(); // addormenta il processo in modo da rispettare il rate
		di publishing
		++count;
	  }
	  return 0;
}
```

Subscriber

```
#include "ros/ros.h"
#include "std_msgs/String.h"

//callback eseguito alla chiamata
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
	ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
	// inizializza il nodo, passando gli argomenti e impostando un nome
	ros::init(argc, argv, "listener");
	ros::NodeHandle n; //c rea l'identificatore per il nodo
	/* sottoscrive il topic chatter, assegna una dimensione massima alla coda e
	collega l'evento al callback */
	ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
	/* entra nel loop di attesa in cui legge i messaggi e chiama i callback
	corrispondenti. esce dal loop in caso che il processo venga terminato da crtl-c o
	dal master */
	ros::spin();
	return 0;
}
```

Server

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h" // header file generato dal messaggio svr
                                           // (analogo al caso dei messaggi)

//callback del servizio
bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res )
{
       res.sum = req.a + req.b;
       ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
       ROS_INFO("sending back response: [%ld]", (long int)res.sum);
       return true; //ritorna true se il servizio termina correttamente
}

//main del servizio
int main(int argc, char **argv)
{
       ros::init(argc, argv, "add_two_ints_server"); //inizializzazione nodo server
       ros::NodeHandle n;
       ros::ServiceServer service = n.advertiseService("add_two_ints", add); /*dichiara
       i servizi che può offrire*/
       ROS_INFO("Ready to add two ints.");
       ros::spin(); //entra nel loop di attesa
       return 0;
}
```

Client

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
#include <cstdlib>

int main(int argc, char **argv)
{
	ros::init(argc, argv, "add_two_ints_client");
	if (argc != 3)
	{
		ROS_INFO("usage: add_two_ints_client X Y");
		return 1;
	}

	ros::NodeHandle n;
	//crea un client per il servizio
	ros::ServiceClient client =
	n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
	beginner_tutorials::AddTwoInts srv;  //crea un'istanza della classe servizio
	srv.request.a = atoll(argv[1]); //setta un campo del messaggio di servizio
	srv.request.b = atoll(argv[2]); //setta un campo del messaggio di servizio
	if (client.call(srv)) //chiama effettivamente il servizio, ritorna una volta
	effettuato.
	{
		ROS_INFO("Sum: %ld", (long int)srv.response.sum);
	}
	else
	{
		ROS_ERROR("Failed to call service add_two_ints");
		return 1
	}
	return 0;
}
```

Esempi di publisher/subscriber e service/client: Python

Publisher

```
#!/usr/bin/env python
# legge manifest.xml e aggiunge dipendenze nel PYTHONPATH
import roslib; roslib.load_manifest('beginner_tutorials')
import rospy
from std_msgs.msg import String # importa codice generato dal file .msg

def talker():
   # il nodo pubblica messaggi di tipo String sul topic ‘chatter’
   pub = rospy.Publisher('chatter', String)
   rospy.init_node('talker') # inizializzazione + nome del nodo
   while not rospy.is_shutdown():
       str = "hello world %s"%rospy.get_time()
       rospy.loginfo(str)
       pub.publish(String(str)) # pubblica il messaggio str
       rospy.sleep(1.0)

if __name__ == '__main__':
   try: talker()
   # l’eccezione può essere lanciata da rospy.sleep()
   except rospy.ROSInterruptException: pass
```

Subscriber

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('beginner_tutorials')
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_name()+"I heard %s",data.data)

def listener():
    rospy.init_node('listener', anonymous=True)
    # sottoscrive il topic chatter e collega l’evento al callback
    rospy.Subscriber("chatter", String, callback)
    rospy.spin() # loop di attesa

if __name__ == '__main__':
    listener()
```

Service

```
import roslib; roslib.load_manifest('beginner_tutorials')
from beginner_tutorials.srv import * # importa codice generato dal file .srv
import rospy

def handle_add_two_ints(req):
    # riceve l'oggetto req contenente i dati inviati al servizio,
    # ritorna un'istanza di AddTwoIntsResponse contenente la risposta
    print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server') # inizializza nodo
    # dichiara che puo' offire il servizio add_two_ints, di tipo AddTwoInts, e
    # che l'handler del servizio e' la funzione handle_add_two_ints()
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print "Ready to add two ints."
    rospy.spin()  # ciclo di attesa...

if __name__ == "__main__":
    add_two_ints_server()
```

Client

```
import roslib; roslib.load_manifest('beginner_tutorials')
import sys
import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints') # attendi che il servizio sia pronto
    try: # ServiceProxy: crea un handler per la chiamata al servizio
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)   # chiama effettivamente il servizio
        return resp1.sum
    except rospy.ServiceException, e: # puo' essere lanciata in caso di errori
        print "Service call failed: %s"%e

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else: sys.exit(1)
    print "Requesting %s+%s"%(x, y)
    print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

Compilare i pacchetti: i file CMakeLists.txt

I file `CMakeLists.txt` sono file per CMake, un programma cross platform per compilare eseguibil, che genera automaticamente Makefile nativi per il sistema utilizzato.

ROS sfrutta CMake per semplificare la compilazione cross-platform degli eseguibili.

Per usare le macro predefinite da ros il file deve avere innanzitutto la riga:

```
include($ENV{ROS_ROOT}/core/rosbuild/rosbuild.cmake)
```

Se si vuole che il progetto non prenda il nome della cwd, porre a true la variabile:

```
ROSBUILD_DONT_REDEFINE_PROJECT
```

verrà dunque usato il nome del package definito nel'XML
in seguito dichiarare

```
rosbuild_init()
```

questa macro compie le seguenti operazioni:

- Configura le cartelle di default per l'output:
  - eseguibili in `${dir}`
  - librerie in `${dir}/lib`
- Aggiunge `${dir}/include` al percorso per la ricerca degli header
- Chiama `rospack` per settare i flag di compilazione e linking nelle seguenti variabili:
  - `-I` flag di compilazione in `${pkg}_INCLUDE_DIRS`
  - altri flag di compilazione in `${pkg}_CFLAGS_OTHER`
  - `-L` flag di linking in `${pkg}_LIBRARY_DIRS`
  - `-l` flag di linking in `${pkg}_LIBRARIES`
  - altri flag di linking in `${pkg}_LDFLAGS_OTHER`

  Non è necessario accedere direttamente alle variabili se si usano le apposite macro di ROS.

cambiare (eventualmente) il percorso di output per i file binari e per le librerie con:

```
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/libraries)
```

Per aggiungere un file eseguibile da compilare usare

```
rosbuild_add_executable(exe src1 src2...)
```

Per aggiungere una libreria da compilare, usare:

```
rosbuild_add_library(lib src1 src2...)
```

NB: non chiamare gli eseguibili o le librerie "test"!

Per abilitare la generazione del codice per messaggi e servizi aggiungere:

```
rosbuild_gensrv()
rosbuild_genmsg()
```

Per aggiungere o togliere flag di compilazione o linking:

```
rosbuild_add_compile_flags(target flags)
rosbuild_remove_compile_flags(target flags)
rosbuild_add_link_flags(target flags)
rosbuild_remove_link_flags(target flags)
```

Per aggiungere le librerie boost:

```
rosbuild_link_boost(target lib1 lib2 lib3 ...)
```

Per trovare un package o uno stack:

```
rosbuild_find_ros_package(pkg)
rosbuild_find_ros_stack(stack)
```

Per includere codice da un altro CMakeLists.txt:

```
rosbuild_include(package module)
```

Per includere le librerie Qt4: (l'esempio utilizza le QtCore, QtGui e QtXml, oltre alle QtMultimediaKit, che sono parte delle QtMobility. Omettere le parti relative ai componenti non utilizzati, e compila tutti I file – sorgenti in src e header in include – in un unico eseguibile)

```
find_package(Qt4 COMPONENTS QtCore QtXml QtGui REQUIRED)
include(${QT_USE_FILE})
include_directories(${CMAKE_CURRENT_BINARY_DIR})
include_directories(${QT_QTXML_INCLUDE_DIR})
include_directories("${QT_INCLUDE_DIR}/../QtMobility")
include_directories("${QT_INCLUDE_DIR}/../QtMultimediaKit")

set (QT_USE_QTXML TRUE)
set(QT_USE_QTMULTIMEDIA TRUE)
set(QT_LIBRARIES "${QT_LIBRARIES};-lQtMultimediaKit")

file(GLOB_RECURSE QT_MOC RELATIVE ${CMAKE_CURRENT_SOURCE_DIR} FOLLOW_SYMLINKS
include/*.h)
QT4_WRAP_CPP(QT_MOC_HPP ${QT_MOC})

#decommentare le linee seguenti se si usano file .ui oppure le risorse
#file(GLOB QT_FORMS RELATIVE ${CMAKE_CURRENT_SOURCE_DIR} ui/*.ui)
#file(GLOB QT_RESOURCES RELATIVE ${CMAKE_CURRENT_SOURCE_DIR} resources/*.qrc)
#QT4_ADD_RESOURCES(QT_RESOURCES_CPP ${QT_RESOURCES})
#QT4_WRAP_UI(QT_FORMS_HPP ${QT_FORMS})

file(GLOB_RECURSE QT_SOURCES RELATIVE ${CMAKE_CURRENT_SOURCE_DIR} FOLLOW_SYMLINKS
src/*.cpp)

rosbuild_add_executable(eseguibile ${QT_SOURCES} ${QT_RESOURCES_CPP} $
{QT_FORMS_HPP} ${QT_MOC_HPP})
target_link_libraries(eseguibile ${QT_LIBRARIES})
```

Launcher ROS

I launcher ROS consentono di lanciare più nodi contemporaneamente, in base a quanto specificato in appositi file di configurazione .launch
Una volta configurato il file .launch, lanciare il comando:

```
roslaunch [package] [filename.launch]
```

i file launch sono file xml contenenti i seguenti tag:

- `<launch>` e `</launch>` all'inizio e alla fine del file
- Uno o più nodi che si vogliono lanciare possono essere racchiusi dal tag `<group ns="[namespace]">` e `</group>`
  in modo da impostare il namespace voluto
- `<node pkg="[nomepackage]" name="[nome nodo]" type="[tipo nodo]"/>` per lanciare un determinato nodo da un package

Si può inoltre lanciare un nodo cambiando i nomi dei topic (remap) in questo modo:

```
<node pkg="[nomepackage]" name="[nome nodo]" type="[tipo nodo]">
      <remap from="[original topic1]" to="[new topic1]"/>
      <remap from="[original topic2]" to="[new topic2]"/>
</node>
```

Legal stuff