Fundamentals of ROS

Carmine Tommaso Recchiuto

What is ROS?

✓ *ROS is an open-source, meta-operating system, for robots.*

It provides the services that you would expect from an operating system, including:

- Hardware abstraction;
- Low-level device control;
- Implementation of commonly-used functionality;
- Message-passing between processes;
- Package management.

Ros Installation

- ✓ Operating system: Unix-based platforms. It is primarly tested on Ubuntu.
- ✓ Installation: follow instructions from *wiki.ros.org.*
- ✓ In the Docker Image, add the line *source /opt/ros/noetic/setup.bash* to the .bashrc file.

ROS Filesystem Level (1)

- Let’s start now to better understand what is ROS and what we can do with it. In the following, we will briefly describe the organization of the ROS framework.

- **Packages:** main unit for organizing software in ROS. A package may contain ROS runtime processes (nodes), a ROS-depented library, datasets, configuration files, or anything else that is usefully organized together.

- **Package Manifests:** Manifests (package.xml) provide metadata about a package, including its name, version, description, license information, dependencies.

ROS Filesystem Level (2)

- **Message (msg) types:** Message description, stored in my_package/msg/MyMessageType.msg, define the data structurs for messages sent in ROS.

- **Service (srv) types:** Service descriptions, stored in my_package/src/MyServiceType.srv, define the request and response data structures for services in ROS.

ROS Computation Graph Level (1)

The Computation Graph is the network of ROS processes that are processing data together

- **Nodes:** Nodes are processes that perform computation. ROS is designed to be modular: a robot control system usually comprises many nodes.

For example, one node controls a laser range-finder, one node control the wheel motors, one node perform localization, one node performs path planning, one node provides a graphical view of the system, and so on. A ROS node is written with the use of a ROS client library, such as roscpp or rospy.

ROS Computation Graph Level (2)

- **Master:** The ROS Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.

- **Messages:** Nodes communicate with each other by passing messages. A message is simply a data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, …) are supported, as well as arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structures).

ROS Computation Graph Level (3)

- **Topics:** Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics.

- **Services:** The publish / subscribe model is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for request / reply interactions, which are often required in a distributed system. Request / reply is done via services, which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply.

ROS – Running a node

- **roscore** (master) is the first thing you should run when using ROS.

- **rosnode** displays information about the ROS nodes that are currently running. The *rosnode list* command lists these active nodes, while the *rosnode info /[node_name]* returns information about a specific node.

- ***rosrun*** allows you to use the package name to directly run a node within a package (without having to know the package path):

  rosrun [package_name] [node_name]

  - e.g. : *rosrun turtlesim turtlesim_node*

ROS - Workspace

- Create your own workspace:
  - - **mkdir –p my_ros/src**
  - - **catkin_make** (in the root folder of your ws. Catkin is the official build system of ROS and the successor to the original ROS build system, rosbuild)

Source the setup file:

  - - *add the line ‘source [ws_path]/devel/setup.bash’ in your .bashrc file.*

ROS – Creating the package

- Create your own package:
  - - **catkin_create_pkg [name] [dependencies]** (inside the src folder of the workspace)

- Example:
  - - **catkin_create_pkg my_first_package std_msgs roscpp rospy**

- Refresh the package list:
  - - **rospack profile**

ROS – Tutorial package

- Now you should have a new folder, called “my_first_package”. Move there with the terminal

- Download the files needed for the exercise
  - **git clone https://github.com/CarmineD8/src.git**
  - **git clone https://github.com/CarmineD8/scripts.git**

- Let’s first analyze the c++ files: publisher.cpp and subscriber.cpp

ROS – Tutorial package

- The same thing can be done in Python: **pub.py** and **sub.py**
- Please notice that usual c++ files are in the *src* folder, while python files are in the *scripts* folder
- But what is the correct procedure for creating nodes and building the package?
- At first we need to make the python scripts executable (**chmod +x <script name>**)
- We need to modify the ***CMakeLists.txt*** file!

Building the project

- Open the CMakeLists.txt
- Check that find_package looks for all dependencies
- Add your cpp node as an executable (uncomment *add_executable*)
- Link with the required libraries (uncomment *target_link_libraries*)
- Call *catkin_make* from the workspace root folder

How to run it?

- rosrun <name_of_the_package> <name_of_the_node>
  - Es: **rosrun my_first_package publisher**
    rosrun my_first_package subscriber

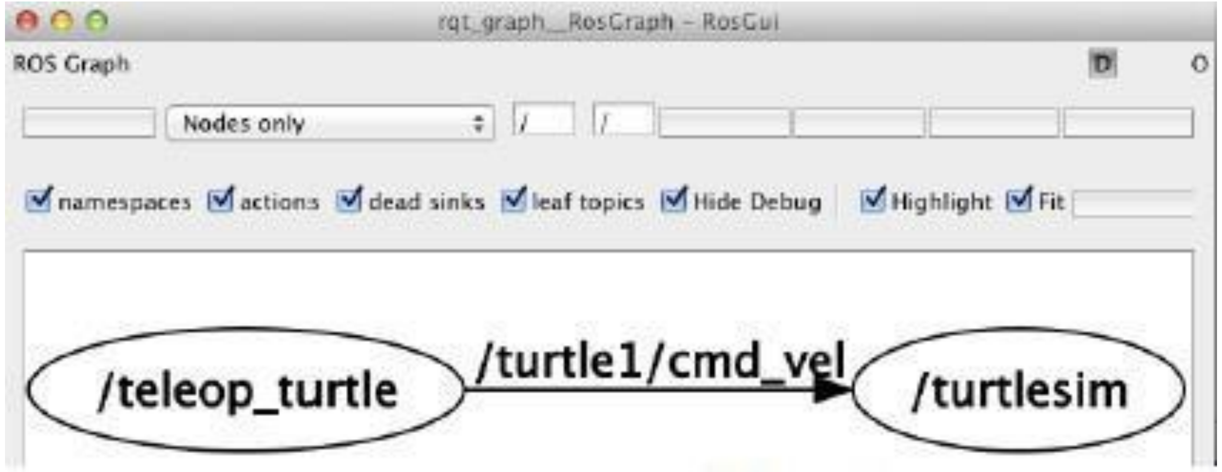
- Useful commands:
  - **rostopic list**
  - **rostopic echo <topic_name>**
  - **rostopic info <topic_name>**

  - **rosmsg show <msg_type>**

Other ROS tools

- **Rqt_graph** shows a dynamic graph of what's going on in the system :

rosrun rqt_graph rqt_graph



Other ROS tools

- Record and play topics:
  - ***Rosbag record chatter***
- It will create in the same folder a .bag file with all the info transmitted on that topic
- The info can be stored played in order to simulate the robot behaviour:
- ***Rosbag play [filename].bag***

Example 2: writing a subscriber for a robot

- Create a new package: **catkin_create_pkg turtlebot_controller turtlesim std_msgs roscpp rospy**

- Create a subscriber.cpp file in the /src folder of your turtlebot_controller package

- Write the source code for the node that will subscribe to the turtle's turtlesim/Pose message and write it into the console

- Required headers for the node.
  - #include *"ros/ros.h"*
  - *#include "turtlesim/Pose.h"*

Example 2: writing a subscriber for a robot

- How may I know the type of message and how the message is structured?

- Start the robot node:
  - **rosrun turtlesim turtlesim_node**

Let’s now check the topics and the messages (**rostopic list, rostopic info** and **rosmsg show**)

Example 2: writing a subscriber for a robot

```
int main (int argc, char **argv)
    {
// Initialize the node, setup the NodeHandle for handling the communication with the ROS
//system
    ros::init(argc, argv, "turtlebot_subscriber");
    ros::NodeHandle nh;
    // Define the subscriber to turtle's posi@on
    ros::Subscriber sub = nh.subscribe("turtle1/pose", 1,turtleCallback);
    ros::spin();
    return 0;
    }
```

Example 2: writing a subscriber for a robot

- Write the callback function turtleCallback that will be called each time a message is published on the turtle1/pose topic (above the main function)

```
void turtleCallback(const turtlesim::Pose::ConstPtr& msg)
{
ROS_INFO("Turtle subscriber@[%f, %f, %f]",
msg->x, msg->y, msg->theta);
}
```

- The message has been passed in a boost_shared_ptr and member of the class being pointed to can be accessed using the dereferencing operator ‘->’

Building the project

- Open the CMakeLists.txt
- Check that find_package looks for all dependencies
- Add your node as an executable (uncomment *add_executable*)
- Link with the required libraries (uncomment *target_link_libraries*)
- Call *catkin_make* from the workspace root folder

Testing the node

- Run the *turtlesim* node

  roscore

  rosrun turtlesim turtlesim_node

  rosrun turtlebot_controller subscriber

- In the terminal your program should be printing the turtle’s pose (position and orientation)

Publish velocity commands

Now we want to add a publisher to our node

- Addt the header related to the velocity message:

  #include "geometry_msgs/Twist.h"

• Above the turtleCallback functiondeclare the publisher

ros::Publisher pub;

• Now setup the publisher within the main function (e.g. just aher the subscriber)

pub = nh.advertise<geometry_msgs::Twist> ("turtle1/cmd_vel", 1);

Message Types

- Keep in mind that the details for the message sent on a topic can be determined using *rostopic type and rosmsg show*
  - ***rostopic type [topic]***
  - ***rosmsg show [topic_type]***

e.g: ***rostopic type /turtle1/cmd_vel***
rosmsg show geometry_msgs/Twist

Publish the velocity

- In the *turtleCallback,* we may actually publish the velocity.

```
geometry_msgs::Twist my_vel;
my_vel.linear.x = 1.0;
my_vel.angular.z = 1.0;
pub.publish(my_vel);
```

Rebuild the node!

- Open the CMakeLists.txt
- Check that find_package looks for all dependencies -> we need to add geometry_msgs!
- Let’s add the dependency on geometry_msgs also on the package.xml file.
- Call *catkin_make* from the workspace root folder

Services

- *The publish/subscribe model (i.e. topic/msgs)* is not always appropriate for RPC **request / reply interactions**, which are likely to occur in a distributed system

- *Request / reply* is done via a **Service,** which is defined with a pair of messages, one for the request and one for the reply.

Services

- A providing ROS node offers a service under a string name, and the client calls the service by sending a request message and awaiting the reply

- Practical usage: ***rosservice*** ***/ rossrv***

- ***rosservice list*** *(list all the available services)*
- ***rosservice call*** *(Call the services with the provided args)*
- ***rosservice type*** *(gives info about the services)*

Services

- E.g. : ***rosservice call clear***
  - *Clear the background of the turtlebot environment*
- The details of the arguments needed from a service can be determined using ***rossrv show [service_type]***
- Example: in order to add a new turtle in a specific point in the environment call the **spawn** service

Example: *spawn* client

Objective: write a node that spawn a turtle in a certain position by using the service "spawn"

1) Check the type of the service (on the terminal ***rosservice type turtle1/spawn***)

2) In the .cpp file, Include the header "***turtlesim/Spawn.h***"

Example: *spawn* client

3)Create a client that sends a Spawn request to the /spawn service

ros::ServiceClient client1 = nh.serviceClient<turtlesim::Spawn>("/spawn");

4) Check the structure of the service (on the terminal ***rossrv show turtlesim/Spawn***)

5) Define a Spawn service message

turtlesim::Spawn srv1;

Example: *spawn* client

6)Add the arguments of the message as defined in the structure of the service*:*

srv1.request.x = 1.0;

srv1.request.y = 5.0;

srv1.request.theta = 0.0;

srv1.request.name = "my_turtle";

7) Call the service:

client1.call(srv1);

Exercise

You can find a starting cpp file in the course material (Teams, Aulaweb) or download it from https://github.com/CarmineD8/ros_ex1

- ✓ Kill the turtle named **turtle1**
- ✓ Spawn a turtle named **rpr_turtle** in the position x = 2.0, y=1.0, theta=0.0
- ✓ Let the turtle move along x, until it reaches the end (x > 9.0)
- ✓ When x > 9.0 or x < 2.0, make it turn in a circular arc
- ✓ Continue until the turtle covers the whole area.
- ✓ Modify the CMakeLists.txt file (if needed) so as to build the program

Exercise