Robotics Lab - Lecture 1

Introduction to Robot Operating System (ROS)

Jonathan Cacace

February 28, 2020

Contents

Preface

This document contains the lecture notes for the *Robotics Lab* class taken at University of Naples Federico II for automation engineering master degree. The aim of this course is to give an overview of the fundamental tools and techniques used to program advanced robotics systems (both industrial and mobile). After a brief introduction of the technologies commonly used to program robots (e.g. Linux, c++, git), the Robot Operating System (ROS) framework is introduced and deeply studied. Simulation software will help the course attenders to implement and test state-of-art robotic algorithms and their own robot control software.

In this lesson, the Robot Operating System (ROS) is introduced in order to provide an initial knowledge of its core underlying concepts.

1

Robot Operating System

Robot Operating System (ROS) represents a flexible framework, providing various tools and libraries to write robotic software. It offers several powerful features such as message passing, distributing computing, code reusing, and implementation of state-of-the-art algorithms for robotic applications.

1.0.1 History of ROS

The first version of ROS was release in 2007 by Willow Garage, a robotics research laboratory located in California. In that year, Willow Garage was developing PR2 robot (see Fig.1.1), one of the first robot running ROS as programming framework.

ROS was born as an open source software and several developers outside Willow Garage participated to its development. One of the main reason motivating the development of ROS was to simplify the role of robotic programmers. In particular, robotic applications are typically composed by similar part of software. The most common problem of robotics at the time was that they spent too much time to re-implement the software infrastructure required to build complex robotics algorithms (basically, drivers to the sensors and actuators, and communications between different programs inside the same robot) and too little time dedicated to build intelligent robotics programs that were based on that infrastructure. For this reason, to overcome this problem ROS offers different features, like an easy process communication, code reuse and software modularity. These features made ROS particularly suitable for pr2 programming since it was composed by several heterogeneous sensors (e.g. sonar, lidars, mobile base, etc...) and hardware components. Nowadays ROS is managed by *OSRF* (Open Source Robotic Foundation) and it is released under BSD License.

Today, ROS represents the standard for robot programming and it is already integrated in many robots and used by many universities and companies.

Figure 1.1: PR2 Willow Garage robot. One of the first ROS-enabled robot.

1.0.2 ROS Distributions

ROS updates are released with new ROS distributions. A new distribution of ROS is composed by an updated version of its core software and a set of new/updated ROS packages. ROS follows the same release cycle of Ubuntu Operating System: a new version of ROS is released every six months. Typically, for each Ubuntu LTS (Long Time Support) version, an LTS version of ROS is released. LTS stands for *Long Term Support* and means that the released software will be maintained for long period time (5 year in case of ROS and Ubuntu). The current LTS version of ROS at writing time is *Melodic Morenia*, and it's supported by Ubuntu 18.04. The list of recent ROS distribution is shown in Fig. 1.2.

1.0.3 Robot Operating System

ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes and package management. It also provides tools and libraries for building, writing and running code across multiple computers. Actually, the correct definition for ROS is a robotic *middleware*, a software that connects different software components or applications, as shown in Fig. 1.3.

ROS officially runs on Unix-based platform. Some experimental versions have been released for Windows and MacOS. The suggested version for Linux is Ubuntu. Before to discuss the features and working principle of ROS, let's take an overview of the main elements of a ROS software. The computation in ROS is done using a network of processes called ROS

Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Noetic Ninjemys	May, 2020 (planned, see Upcoming Releases)	TBA	TBA	May, 2025 (planned)
ROS Melodic Morenia (**Recommended**)	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

Figure 1.2: Recent ROS release.

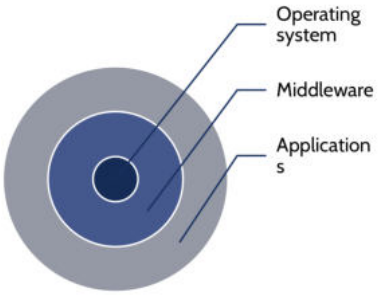

Figure 1.3: Middleware concept.

nodes. This computation network along with additional functionalities can be called *computation graph*. The main concepts in ROS computation graph are *Nodes, Master, Parameter server, Messages, Topics, Services, and Bags*. Each concept in the graph contributes in different ways. Let's focus on the two main elements that are the nodes and the master:

- **Nodes**: Nodes are the processes that perform computation (the executable). Each ROS node is written using ROS client libraries implementing different ROS functionalities, such as the communication methods between nodes, which is particularly useful when different nodes of our robot must exchange information between them. Using the ROS communication methods, they can communicate with each other and exchange data. One of the aims of ROS nodes is to build simple processes rather than a large process with all the functionality (modularity).

- **Master**: The ROS Master is a special ROS node that provide the name registration and lookup to the rest of the nodes. Nodes will not be able to find each other, exchange messages, or invoke services without a ROS Master. In a distributed system, we should run the master on one computer, and other remote nodes can find each other by communicating with this master.

The typical structure of ROS applications is shown in Fig. 1.4. In particular, in each ROS system only one Master node can be active. All the other nodes exchange information between them thanks to the ROS master. In practice, a ROS node is nothing more that a program written by developers in one of the supported programming languages. Currently the supported languages are C++, Python, Matlab and Java. The ROS functionalities in all the programming languages are the same. The ROS Master is much like a DNS

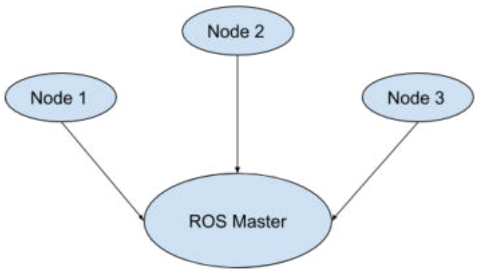

Figure 1.4: ROS application structure.

server, associating unique names and IDs to ROS elements active in our system. When a new node is launched in the ROS system, it will start looking for the ROS Master and register the name of the node in it. So, the ROS Master handles the details of all the nodes currently running on the ROS system.

Now that the basic idea of ROS has been discussed, let's start to better detail its components.

The elements of ROS are summarized in Fig. 1.5 and are described in the following:

- **Plumbing**: ROS allows communication between processes (nodes). In particular, it provides publish-subscribe messaging infrastructure designed to support the quick and easy construction of distributed (local and remote) computing systems. For example, consider that your application uses data from a camera, you can use the ROS node deployed by the vendor of your camera to use the data in your own application.

- **Tools**: ROS provides an extensive set of tools to configure, manage, debug, visualize data, log and test your robotic application.

- **Capabilities**: ROS provides a broad collection of libraries that implement useful robot functionalities, like manipulation, control, and perception. In addition, ROS can be connected to other external software like OpenCv, PCL, and so on, thanks to proper wrappers (i.e. developers can avoid to re-invent the wheel).

- **Ecosystem**: ROS is supported and improved by a large community, with a strong focus on integration and documentation. On ROS webpage: `ros.org` you can find basic and advanced tutorial to learn how to program in ROS, while di Q&A website (`answers.ros.org`) allow you to directly ask you solution for your own problems (and contains thousand of question already answered).

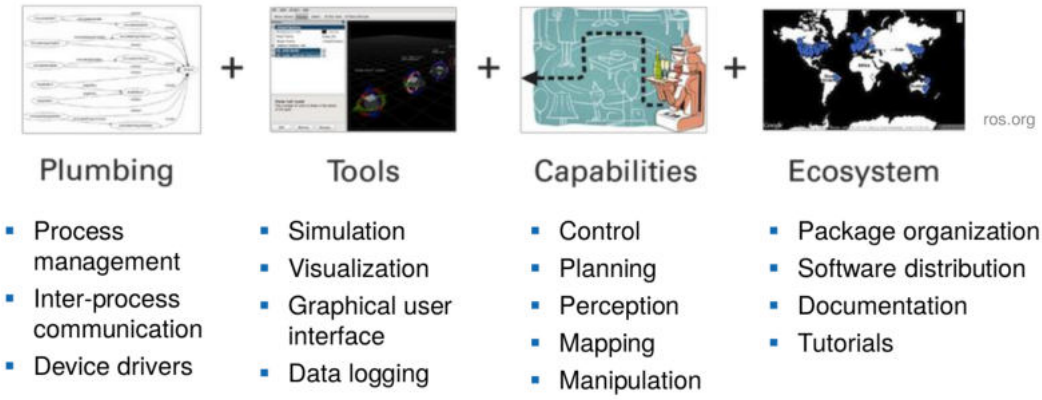

Figure 1.5: ROS components.

At this point we are ready to discuss the philosophy of ROS.

The philosophy of ROS is that you have several individual programs (modules) implemented in your robotic system (these programs can be located on the same machine or **distributed** over the network) and are able to communicate each other using defined API like ROS messages, services and others. Each module can be written in any preferred programming language supported by ROS (at current stage: C++, Python, Matlab and Java). ROS is a free software and its core is open source.

Following this philosophy you can be easily able to design modular software with several independent interchangeable modules solely responsible of a small task in the overall software. The advantages the modularity in your code are many. First of all, debug small part of code and functionalities is easier. In addition, will be more easy update your software substituting only the legacy part of your software.

Let's discuss an example of a simple robotic application programmed with ROS. We can consider the navigation of a mobile robot that have to

track and follow a given visual target. Such application can be quite complex to program from scratch and can be composed by several modules, as depicted in Fig. 1.6. In this context, we can classify three application layers. One responsible to handle robot sensors, like the vision sensor (*Camera*), the laser scanner (*Lidar*) and the *Encoders* of the wheels. The second layer instead, is responsible of the robot navigation. To perform such task, the robot must be able to *Localize* itself into the environment, to *Map* the obstacles and to generate a control strategy to accomplish a given task considering sensor data. Finally, the lower level of your application consists in the integration with the hardware of the robot. Thanks to the large number of versatile ROS modules the developers can concentrate on the programming of the control algorithm. In fact, several sensors are already supported by ROS, like standard usb cameras, depth sensor (from kinect, asus or intel), laser scanner and so on. Similarly, state-of-art mapping and localization algorithms are already deployed to receive the data from the sensors. One important element of this software architecture is that to exchange information between multiple modules (plumbing) in ROS a set of *standard* messages is used. In this context, another cause for reflection regards the maintainability of the code. In fact, update or change sensors or piece of codes in this architecture is very easy since the communication interface between the modules will remain the same.

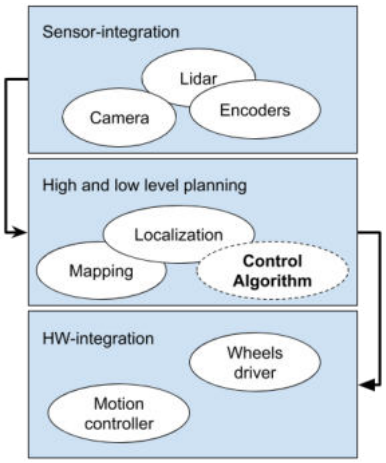

Figure 1.6: Example of ROS node in a robotic mobile navigation task.

Now, let's talk more concretely about ROS software infrastructure starting with the *message-passing*. In the following, two kind of communication protocols are described: the publish-subscribe and services.

Publish-subscribe

Two processes (ROS Nodes) can communicate in different ways. The first communication protocol discussed here is an asynchronous communication protocol based on the publish/subscribe paradigm in which a process streams a series of data that can be read by one or more processes. This communication relies on an entity called **topic**. In particular, each message in ROS is transported using named buses called topics. When a node sends a message through a topic, then we can say the node is publishing a topic, while when a node receives a message through a topic, then we can say that the node is subscribing to a topic. The publishing node and subscribing node are not aware of each other's existence, in fact we can even subscribe a topic that might not have any publisher. In short, the production of information and consumption of it are decoupled. The publish/subscribing communication is described in Fig. 1.7. In this context, the publisher and subscriber nodes register to the ROS Master. The publisher node creates a topic specifying its name that must be unique in the ROS system and the type of the message to publish. Differently, the subscriber node request the data from the topic as long as it specifies the correct message type. The ROS publish/subscribe communication protocol is useful especially when a node must share a continuous stream of information. For example, a node that must grab data from a camera sensor should broadcast the sequence of the images taken from the sensor using a ROS publisher. In this case we are not interested in the of the communication bridge.

ROS Messages

ROS communication relies on a set of standard and custom data structures called ROS messages. Datatypes are described using a simplified message description language called ROS messages. These datatype descriptions can be used to generate source code for the appropriate message type in different target languages. The message definition consists in a typical data structure composed by two main types: fields and constants. The field is split into field types and field names. The field type is the data type of the transmitting message and field name is the name of it. The constants define a constant value in the message file. In the following, an example of message definition to share the pose (position and orientation) of the robot is shown. It is a `geometry_msgs::PoseStamped` message.

```
1 std_msgs/Header header
2         uint32 seq
3         time stamp
4         string frame_id
5 geometry_msgs/Pose pose
6         geometry_msgs/Point position
```

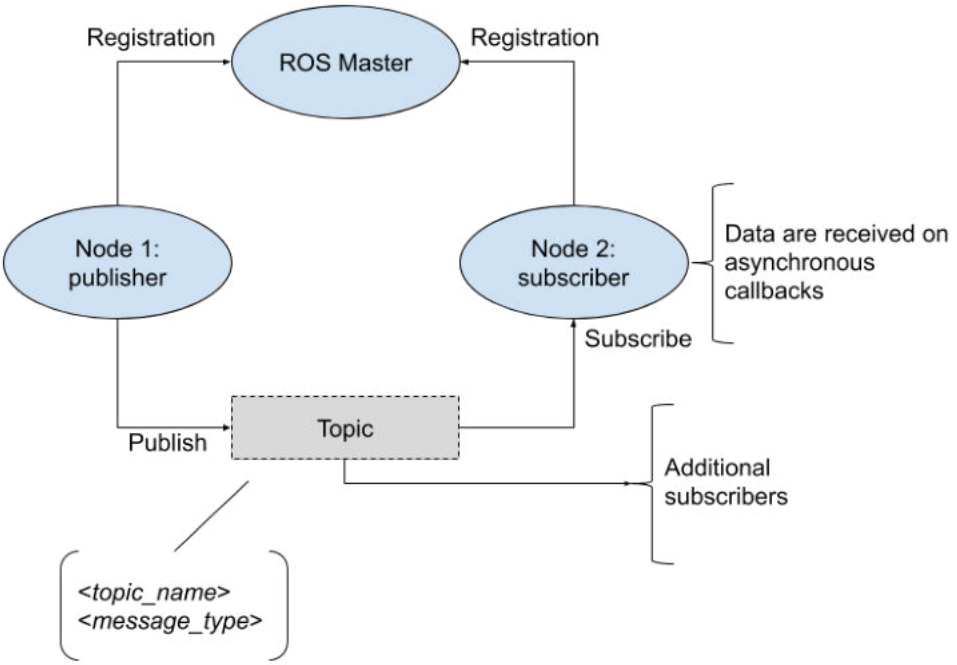

Figure 1.7: ROS publish/subscribe protocol.

```
7                 float64 x
8                 float64 y
9                 float64 z
10          geometry_msgs/Quaternion orientation
11                float64 x
12                float64 y
13                float64 z
14                float64 w
```

This example represents the definition of a structured data to stream. The first part of the message is present in several ROS messages and represents an *header* containing information about the publishing time of the message and its reference frame (i.e. the fixed or dynamic reference frame with respect to the pose of the object is specified). The rest of the message contains information about the 6D position of the robot. As you can easily see, the structure of the message is composed using basic data types (string, float and so on). Table 1.1 shows some of the built-in field types that we can use in our message.

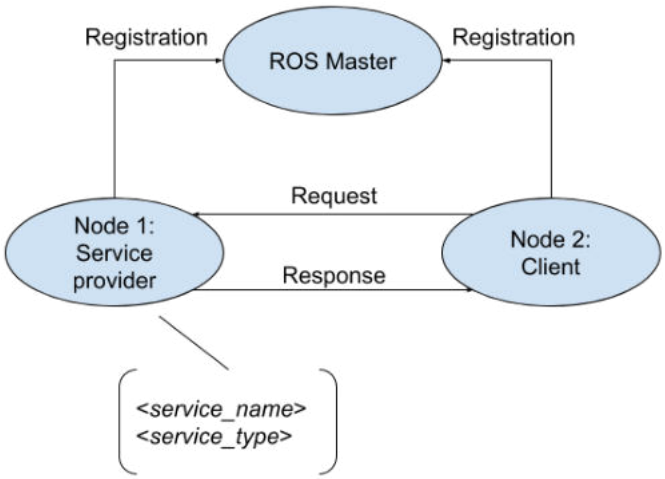

Figure 1.8: ROS service communication.

Primitive type	Serialization	C++	Python
bool(1)	Unsigned 8-bit int	uint8_t(2)	bool
int8	Signed 8-bit int	int8_t	int
uint8	Unsigned 8-bit int	uint8_t	int (3)
int16	Signed 16-bit int	int16_t	int
uint16	Unsigned 16-bit int	uint16_t	int
int32	Signed 32-bit int	int32_t	int
uint32	Unsigned 32-bit int	uint32_t	int
int64	Signed 64-bit int	int64_t	long
uint64	Unsigned 64-bit int	uint64_t	long
float32	32-bit IEEE float	float	float
float64	64-bit IEEE float	double	float
string	ascii string(4)	std::string	string
time	secs/nsecs unsigned 32-bit ints	ros::Time	rospy.Time
duration	secs/nsecs signed 32-bit ints	rospy.Time ros::Duration	rospy.Duration

Table 1.1: Built-in type for ROS messages

ROS Services

ROS also supports a synchronous communication protocol: ROS services. ROS Services establish a request/response communication between nodes. One node will send a request and wait until it gets a response from the other. Similar to the message definitions we have to define the service definition. A service description language is used to define the ROS service types.

An example of service description format is as follows:

```
1 #Request message type
2 string str
3 ---
4 #Response message type
5 string str
```

The first section is the message type of the `request` that is separated by $---$ and in the next section is the message type of the `response`. In this example, both Request and Response are strings. Of course, the definition of

a service could contain a structured ROS message (like the one used for the pose of the robot) as well. This type of communication protocol should be used when a node is specialized in doing a specific tasks, like some complex calculation. For example, this node could be responsible to calculate the inverse of a matrix. In this way, all the other modules of your system that need to invert a given matrix could directly use this service, without replicate the inversion operation in their source code.

Visualization

ROS provides different tools to support developers in their work. Among these tools, Ros Visualization (RViz) is very useful. It is a 3D visualizer to graphically display the contents of a topic using `viualization_markers` messages. RViz can visualize robot models, the environments they work in, and sensor data directly from ROS topics. Built-in and custom plugins can be loaded on RViz to add additional functionalities like motion planning or motion control.

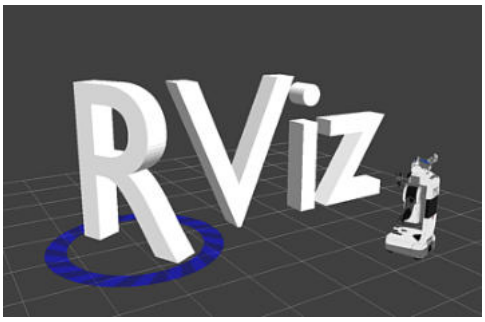

Figure 1.9: ROS Visualization (RViz).

Simulation

Another important feature of ROS is the simulation. In particular, ROS is strictly integrated with Gazebo simulator `http://gazebosim.org/`, a multi-robot simulator for complex indoor and outdoor robotic simulation. In the Gazebo environment we can simulate complex robots, robot sensors, and a variety of 3D objects. Gazebo already has simulation models of popular robots, sensors, and a variety of 3D objects in their repository. In addition, several plugin already exists to interact with the simulated using ROS. Using Gazebo we can simulate the real sensor mounted on our robots receiving exactly the same stream of data (thanks to the same ROS message definition between the real and simulated sensors). An example of the Gazebo interface is shown in Fig. 1.10, where a wheeled mobile robot is simulated.

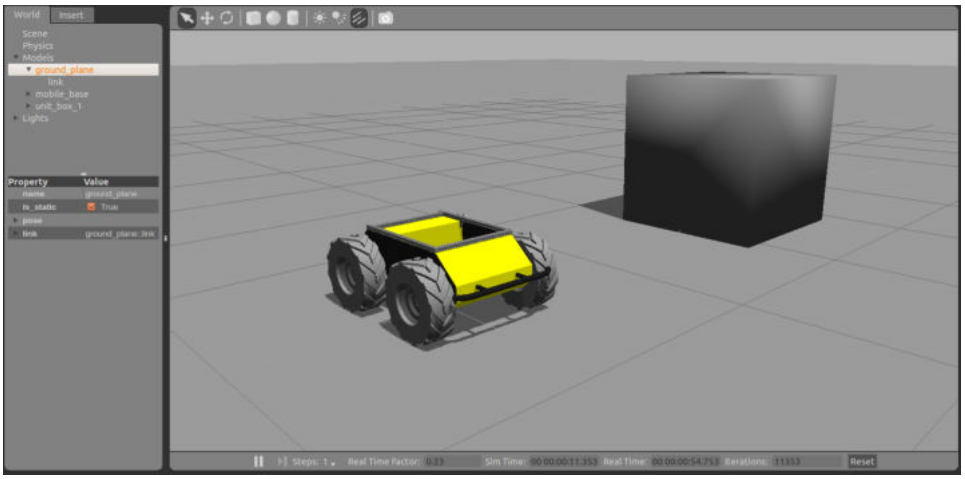

Figure 1.10: Gazebo ROS simulator.

Summary

In this lesson, we introduced Robot Operating System detailing its history and the motivation that brought to its development. The main element of the ROS computation graph have been discussed: the ROS master node and the executable nodes. Finally, two communication protocols allowing ROS nodes to exchange messages each other have been introduced. Beyond all its benefits, there are some disadvantages in using ROS. In particular, main issues of ROS concern the its reliability and safety. Firstly, the communication between multiple nodes can be unstable, especially with a big amount of data or with a distributed robotic system. In addition, the ROS master will respond to requests from any device on the network (or host) that can connect to it. Any host on the network can publish or subscribe topics, list or modify parameters, and so on. If an unauthorized user can connect to the ROS master, they could leak sensitive information (such as data from sensors or cameras), or even send commands to move a robot, which creates both a privacy and a safety risk. These and other issues bring to the development or ROS 2, the second version of ROS that focus on the use of ROS in real industrial scenarios.

In the next lesson, an overview of the technologies used to program robot will be provided. In particular, we will learn the basic usage of a unix-based operating system, the basic concept of C++ programming and the compilation and an overview of version control tools like git.