Robotic Systems Programming - BOT
Practical Session 1 (TP-1)
Introduction to ROS

Panagiotis PAPADAKIS, Lucia BERGANTIN

1 Introduction

The objective of TP-1 is to introduce to you the **Robot Operating System** (ROS), the internationally established middle-ware in robotics programming engineering. While the objective of this session is dedicated to ROS, getting accustomed to ROS functionalities and usage will also be implicitly performed all along the TP series through exercises of varying complexity and scope.

What is ROS? ROS[1] is a project with a very recent history whose origin in time traces back to late 2000s (2008 and onwards[2]). It resulted from the need of robotics research community for establishing a **common framework** allowing the **collaboration**, **functionality exchange** and **standardization** of practices in the programming of diverse robotic architectures. From its initial steps predominantly organized and supported by *Willow Garage*[3], a United States based robotics companies incubator and Stanford University, it has today become an internationally renowned, active, open-source, community-supported project. It is composed of a collection of software tools, libraries[4] and conventions that allow the development of complex behaviours across diverse robotic platforms[5].

Why ROS? Performing a seemingly simple (from the human point of view) robot operation often requires the combination of diverse robot skills and human

[1] `http://www.ros.org/`

[2] Quigley M. et al., "*ROS: an open-source Robot Operating System*", IEEE International Conference on Robotics and Automation, Workshop on Open Source Software, 2009

[3] `https://en.wikipedia.org/wiki/Willow_Garage`

[4] `https://index.ros.org/packages/`

[5] `http://robots.ros.org/`

expertise. Consequently, effectual robotic projects require team work where often each team masters a specific functionality of the entire system. Agreeing upon and using a common framework for interfacing and integrating functionalities is therefore crucial. ROS allows to build upon other's work, by serving as the common ground among different hardware and software components that need to communicate with each other and coordinate.

2 ROS philosophy

Communication As highlighted earlier, a robot operation is most often performed by several processes that communicate with each other in a coordinated manner. ROS enables this via a **peer-to-peer** (p2p) communication mechanism between processes, meaning that messages are exchanged at a local level (between the peers). The peer-to-peer paradigm is adopted in ROS due to the nature of most robotic systems wherein the main volume of information needs to be exchanged within subnets (subgroups) of the processes of the entire system.

The peer-to-peer philosophy is in contrast to the *client-server* communication paradigm where all processes transmit their messages to a central server which then dispatches the messages to the appropriate recipients. Such a philosophy is not suited for robotic systems because if all messages would need to be centrally communicated, the network channel would be easily overwhelmed with data from various sources (RGB images, 3D point clouds, sound, video, occupancy maps, range scans, etc).

The peer-to-peer communication requires nevertheless a minimal level of centralization via a dedicated, *master* process (see section 3.1) that allows peers to look-up for each other and establish the communication. ROS allows processes to communicate both asynchronously (via *topics* 3.3) and synchronously (via *services* 3.4).

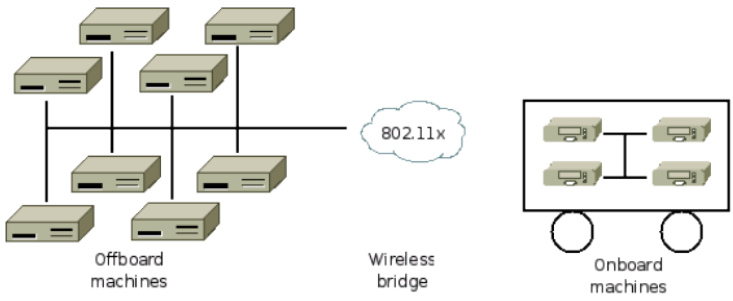

Figure 1: Typical ROS network configuration [cf. footnote[2]].

3 ROS computation

3.1 Nodes

In ROS a computer process that performs an elementary operation is called *node*, in other words, it corresponds to one of the peers mentioned earlier. A typical robotic system is composed of many nodes communicating with each other and can be schematically visualized as a graph whose edges correspond to the established p2p communications. An example of a simple ROS node graph is given in Fig. 2.

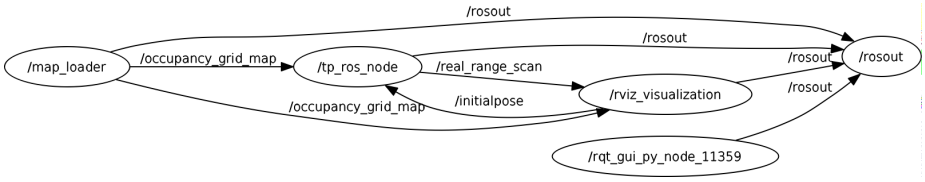

Figure 2: Typical ROS node graph. The ellipses represent ROS nodes performing some elementary operation. The oriented edges denote established peer-to-peer (node-to-node) communication, from a sender versus the recipient.

Nodes that perform very closely related tasks can be grouped into a ROS *package*. For example a single ROS package may contain a node that allows a robot to detect a 3D plane and another node to detect a 3D ball and altogether could be part of a ROS package for detection of shape primitives. Another ROS package may offer detection of solid objects while another package detection of articulated objects.

The *master* process The ROS master process allows nodes to communicate with each other by providing the list of names for node subscribers, node publishers and services. In any ROS system, running the master processes is mandatory and is run before any other node can be launched. The master process further serves as a *parameter* server in the sense that it maintains a list of parameters that may be shared by ROS nodes. With reference to Fig. 2, the master process is represented by the `/rosout` node.

3.2 Messages

Nodes communicate with each other through appropriately defined messages, i.e. data structures. Messages can be composed on the basis of **built-in ROS fields types**[6] (e.g. `float, char, int`, etc), allowing the composition of arbitrarily complex new messages (arrays, messages of messages, etc). Messages are defined using a simple description language whose syntax does not follow the rules of a particular programming language.

[6] `http://wiki.ros.org/msg`

The structure of a ROS message is defined in a text file of the form `.msg` based on which, code (C++, Python, Lisp, etc) is then generated by ROS that is responsible for following the syntax of the respective language. In the following example, the structure of the ROS message `geometry_msgs/Transform` is shown:

```
geometry_msgs/Vector3 translation
  float64 x
  float64 y
  float64 z
geometry_msgs/Quaternion rotation
  float64 x
  float64 y
  float64 z
  float64 w
```

3.3 Topics

Messages are exchanged between ROS nodes via a **publish/subscribe** mechanism by routing messages to communication channels called **topics**. A ROS topic has a name which is used by a node to refer to the communication channel where it can publish messages that other nodes can receive (listen) if they subscribe to that topic.

Referring to Fig. 2, the node `/tp_ros_node` publishes messages to a topic named `/real_range_scan` to which the node `/rviz_visualization` has subscribed to. Likewise, the node `/tp_ros_node` is listening to messages in the topic named `/initialpose` where the node `/rviz_visualization` is publishing messages.

Nodes can publish or subscribe to topics without restriction as long as they define the name of the topic and the type of the expected message. Communication in a pair of nodes can be established through the ROS master process which is the only node aware of topic publishers and topic subscribers. ROS supports TCP/IP-based (the default) and optionally UDP-based message transport.

3.4 Services

Complementary to asynchronous communication via topics, ROS further allows synchronous communication between nodes via *services*. A ROS service allows nodes to request an information and receive a reply. A service is defined by:

- A message definition that has to be respected for the request
- A message definition that has to be given in the reply
- The name of the service

4 Exercises

Important notes
- The `%` symbol means a linux terminal session.
- Avoid copy-paste from the .pdf to the terminal, as it does not always work as expected
- Be careful when setting variable values. A single character error will always lead to failure

4.1 Start-up of your virtual machine

From the menu on the left of your Linux session, start-up the virtual machine (VM) software and choose the VM named "TP-BOT".

4.2 ROS invocation

Remote access In this exercise you will connect remotely via *ssh* to a machine that is already running the ROS master process. Follow carefully the next steps:

1. Open a terminal and connect to the machine that hosts the ROS system:
   `% ssh client-stud@ip_of_server`

   where `ip_of_server` should be given to you by the teacher . The password for the user `client-stud` is: elu613ros

2. Once connected, execute a typical ROS command from the terminal, e.g.:
   `% rosnode list`

   The above command should **list all nodes** that are part of the ROS system (if such a system is already running). If you obtain a list of (>= 1) node names after execution, it means that you have successfully established remote (yet direct) access to the underlying ROS system.

   By convention, all command-line tools based on ROS are preceded by the predicate "ros". If you type `ros` in the terminal followed by 2 consecutive TABs, you should be able to get the exhaustive list of ros commands.

3. You can obtain documentation on the use of a particular ROS command by typing: `% ros<command> -h`

   Retrieve the documentation of the `rosnode` command and invoke it so that it outputs information related to an active ROS node. Likewise, refer to the documentation of the command `rostopic` and use it to retrieve the ROS topics that are active.

4. Use the `rostopic` command to publish a message of type `std_msgs/String` to a topic named as your lastname (e.g. `smith`). In another terminal (repeat again step 1.), use `rostopic` to listen to the messages that you are

publishing. You should be able to experiment by listening to the topics created by your classmates and exchange ROS messages between each other[7].

Local access In this exercise you will introduce your virtual machine to the remote ROS system that is already up and running. This assumes that your machine has already an installation of ROS, which is true for the used virtual machine. To verify, try to run any ROS command and see if the command is found and executed as expected. For example, in a new local terminal session, try to run:

```
% rosversion -d
```

to see the version of the installed ROS distribution

You can introduce your machine to an already running ROS system and executally locally all ROS related operations thanks to the *master* ROS service. To do so, ensure that your virtual machine is connected to the network and follow carefully the next steps:

1. Every machine that wants to be member of a ROS ecosystem needs to have a kind of identity that the other members will use to communicate with it. Since members communicate through TCP/IP, each member is identified by its IP in the network. From a terminal of your virtual machine (local), retrieve your IP:

   ```
   % ip addr show | grep 'inet '
   ```

   With the above command, you should be able to retrieve the IP of your machine (the one starting with `inet 10.`).

   For the other members/machines of the ROS system to communicate with you, you have to set a ROS environment variable via:

   ```
   % export ROS_IP=your_ip_address
   ```

   Verify that the environment variable has been properly set by typing:

   ```
   % echo $ROS_IP
   ```

   For you to be able to communicate with the other members/machines of the ROS ecosystem, you only need to be able to communicate with the ROS master service. After knowing the IP of the ROS master (`ip_of_server` given earlier to you by the teacher), you have to set another ROS environment variable via:

[7] If you wish to break the execution of a ros command, simply press `Ctrl+C`

```
% export ROS_MASTER_URI=http://ip_of_server:11311
```

Verify the content of the `ROS_MASTER_URI` variable .

2. You can verify that your computer is successfully integrated into the ROS active system by trying to list, for example, the active topics or nodes (see previous exercise).

   Note that whatever ROS operation you are launching now runs on your own machine, as opposed to what you were doing before when you remotely connected to the server machine and every command that you started was running there. Try to publish and listen messages among your colleagues, to validate that you are both visible in the ROS ecosystem, i.e. that the ROS *master* service is fulfilling its job (you can use the command `rostopic pub`[8] for this purpose).

3. Open a new terminal and before trying anything else, try to communicate in the ROS system (e.g. through listing topics). What do you notice? Can you explain and fix the problem?

4. You will now create your own isolated ROS system that should be visible ONLY to your own machine. Isolating yourself means that the remote ROS master service should NOT be able to access you. As a consequence, you will need to initiate your **own** ROS master service.

   Local ROS system To start your own ROS ecosystem, you should make sure that the newly initiated ROS master service does not conflict with the existing ROS master service running remotely. By default, i.e. after the opening of a new terminal, your machine is considered as the host of the ROS master service (simply invoke `% echo $ROS_MASTER_URI` in a new terminal to verify this). Follow the next steps:

5. Open a new terminal and initiate your new ROS master service by:
   `% roscore`

6. Open another terminal and test some of the ROS commands that you have learned so far to see that you have indeed created an isolated ROS system.

4.3 ROS Package creation

You have so far performed a very basic use of ROS functionalities using pre-built utilities in order to understand ROS's communication infrastructure. The last objective of this TP is to guide you through the process of creating your first ROS **Hello World** package. To create a ROS package, you need to follow a

[8] `http://wiki.ros.org/ROS/YAMLCommandLine`

number of conventions when creating and placing source files in directories and when building your project.

Creating ROS workspace ROS uses a custom build system to create packages, called `catkin`[9]. The role of `catkin` is to produce targets (executables, libraries, interfaces, etc) from raw source code (C++, Python, LISP), analogously to, for example, the GNU `Make` tool or `cmake`. In reality, `catkin` extends `cmake` to manage ROS packages dependencies. `catkin` is meant to make project building transparent to the operating system, by solely basing itself on `CMake` and `Python` that are both portable.

To create a ROS package, a `catkin` workspace needs to be created (if it has not been already created). Do the following:

1. Download the unix script file for creating the workspace from `https://moodle.imt-atlantique.fr/pluginfile.php/32285/mod_folder/content/0/catkin_workspace_create.sh?forcedownload=1`, open it and read its content to understand what it will do.

2. From the directory wherein you downloaded it, execute it in a terminal via :

   ```
   % .  catkin_workspace_create.sh
   ```

   Validate that it executed as expected

3. Go to `~/student-catkin-ws` and *source* the newly created setup file `setup.sh`:

   ```
   % source devel/setup.sh
   ```

   This way your newly created workspace will be visible in the entire ROS installation, but only for the current terminal session. To make your workspace persistently visible every time that you open a new terminal session, do the following:

   ```
   % echo "source ~/student-catkin-ws/devel/setup.sh" >> ~/.bashrc
   ```

   This way the sourcing of the workspace setup file will be added in your `.bashrc` file and will be performed every time that a new terminal session is created.

Creating ROS package The ROS workspace you just created will host your new ROS package that you are about to create. Again, we will be using ROS tools that automatize a part of the necessary work. Follow the next steps:

[9] `http://wiki.ros.org/catkin`

1. Go to the `~/student-catkin-ws/src` folder and execute:
   `% catkin_create_pkg tp1_ros_package std_msgs rospy`

   The ROS tool `catkin_create_pkg` creates for you an empty ROS package named "`tp1_ros_package`" and a number of necessary files and folder tree. The two last arguments `std_msgs rospy` refer to other ROS packages to which your package will depend on. Obviously, all dependencies may not be known in the beginning but more dependencies can be easily added later if necessary.

   The most important files at this point are the:

   (a) `package.xml`[10]

   (b) `CMakeLists.txt`

   located in the ROS package's home folder. Examine the contents of these files to understand their purpose by opening them as standard text files. In short, `package.xml` is meant to provide meta-information of your package and `CMakeLists.txt` serves for automatizing the building process.

2. You have currently a minimal ROS package that you can attempt to build. Execute the following:

   ```
   % cd ~/student-catkin-ws/; catkin_make
   ```

   This will invoke the building process for the entire `~/student-catkin-ws/` workspace which contains your ROS package. Since your ROS package contains no source code, no targets will be built.

Publisher node You will now start to introduce actual source code into your newly created `tp1_ros_package` ROS package. The goal is to create a typical "Hello world" package in ROS. Follow the next steps:

3. Download the Python script containing the "Hello world" source code from `https://moodle.imt-atlantique.fr/pluginfile.php/32285/mod_folder/content/0/my_first_ros_node.py?forcedownload=1`. Create a folder named `scripts` inside the `~/student-catkin-ws/src/tp1_ros_package` folder and place the Python script within the folder.

   NOTE: You should give execution rights to that file otherwise the ROS package will not be able to use it. Execute:

   ```
   % chmod +x my_first_ros_node.py
   ```

[10] `http://wiki.ros.org/catkin/package.xml`

inside the `scripts` folder. Open the Python script and examine its contents. For all documentation related to the API of rospy, refer to `http://wiki.ros.org/rospy/Overview`

Running ROS node You should now be ready to run your ROS package within a ROS ecosystem. Do the following:

4. Launch the master service:

   ```
   % roscore
   ```

5. In a new terminal, type:

   ```
   % rosrun tp1_ros_package my_first_ros_node.py
   ```

   The `rosrun` command will ask the ROS master service to register your node to the active ROS system and will start the corresponding script. Inspect the list of ROS topics and ROS nodes that are now active using the appropriate commands.

Subscriber node Download the Python script containing the source code responsible for listening to the messages that are published by the first ROS node, from `https://moodle.imt-atlantique.fr/pluginfile.php/32285/mod_folder/content/0/my_listener_ros_node.py?forcedownload=1`. Examine its contents to understand its function. Refer to the `rospy` documentation given earlier when appropriate.

Finally, copy the file to the scripts folder, give it execution rights and run it within the ROS ecosystem.

5 Toy turtle robot simulation

6. Execute the following command:

   ```
   % rosrun turtlesim turtlesim_node
   ```

   which should launch a simulator of a turtle as shown in Figure 5.

   The simulator is supposed to make the turtle to move if the ROS node receives the appropriate ROS message. Try to make the turtle move, using the ROS tool `rostopic pub`[11] and the appropriate message type.

[11] `http://wiki.ros.org/ROS/YAMLCommandLine`

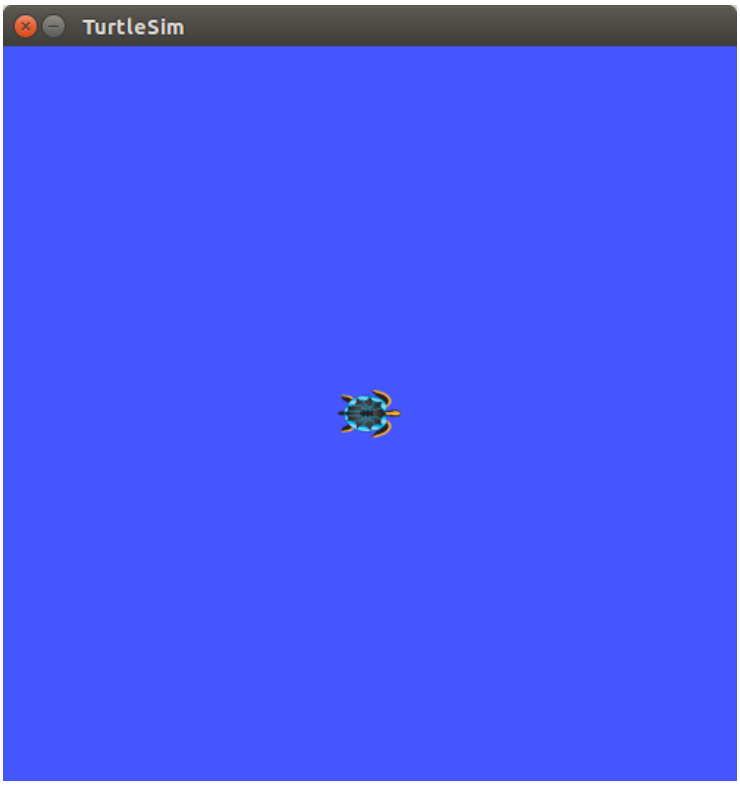

Figure 3: Window of turtlesim ROS package.

ATTENTION: the turtle can move by combining linear (forward/backward) and angular (clock/left) velocities. The x-axis is assumed pointing forward from the head of the turtle and the z-axis is parallel to gravity.

Execute the following command to control the turtle with your keyboard:

```
rosrun turtlesim turtle_teleop_key
```

Using the arrow keys on your keyboard, move the simulated turtle in the simulation environment.

2. In another terminal, execute the following command:

   ```
   rqt
   ```

   This command opens the `rqt` interface[12]. The `rqt` interface in ROS is a GUI tool that provides plugins for monitoring and debugging ROS sys-

[12] https://wiki.ros.org/rqt

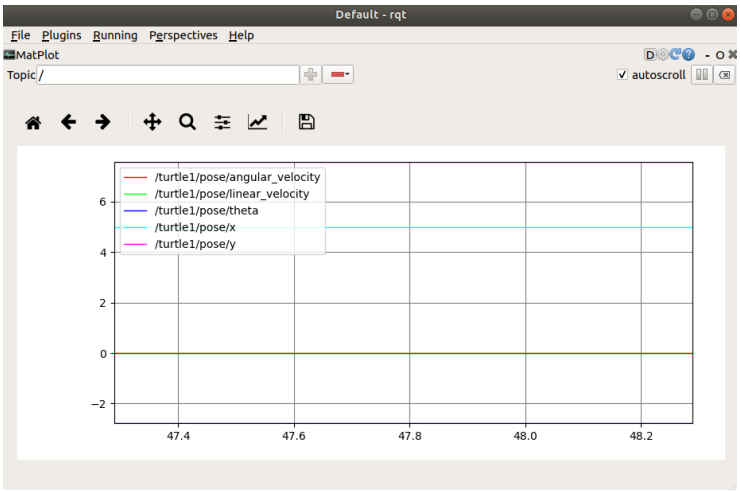

Figure 4: Window of `rqt` interface.

tems. It allows users to visualize topics, inspect node graphs, plot data, and manage parameters in an intuitive way.

In the interface, select `Plugins > Topics > Topic Monitor`. Check the list of topics, select `/turtle1/pose`, and open the topic. Move the turtle around using the keyboard and observe what happens. Then, select `Plugins > Visualization > Plot`. Move the turtle around using the keyboard and observe what happens on the plot on the right side. Now, select `Plugins > Introspection > Node Graph`. Observe the graph of the ROS system to understand the relationships between the nodes currently running. What happens in `rqt` if you kill the `turtlesim_node` node?

`rqt` is a powerful tool, but sometimes we need shortcuts to access certain information. If you killed the turtlesim simulation, restart the `turtlesim_node` and the node enabling keyboard control. To show the topics plot, execute the following command:

```
rqt_plot
```

To plot the ROS system graph, execute the following command:

```
rqt_graph
```

3. In ROS, it is possible to record and replay ROS messages using rosbags[13]. This allows data from topics to be captured in a bag file (.bag), which can later be played back for analysis, debugging, or testing without needing real-time data sources.

[13] https://wiki.ros.org/rosbag

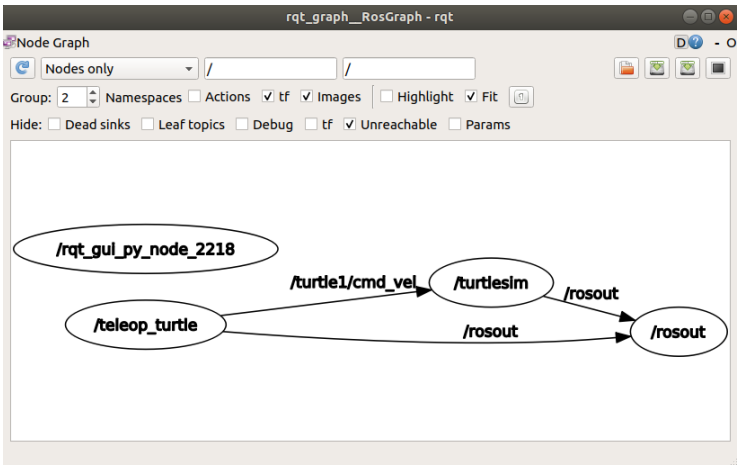

Figure 5: Window of `rqt_graph`.

Create a folder named bagfiles where you will save your rosbag files. To start recording all currently published topics, execute the following command:

```
rosbag record -a
```

Move the turtle around using the keyboard while the rosbag is being recorded. When you are done, stop the recording using Ctrl + C. Check that the rosbag has been correctly saved in the bagfiles folder.

To play the rosbag, fist restart the turtlesim simulation and then execute the following command:

```
rosbag play <name bagfile>
```

The turtle should move in the exact same way.

We can also record only a specific topic. To do so, run again turtlesim_node and the node enabling keyboard control. To start recording, execute the following command:

```
rosbag record <topic name>
```

Which topic should you record? Use `rqt_graph` to find out. Now, play the rosbag back and check if the turtle moves correctly.