Workshop “ROS for Engineers” - part 1

Author (dutch): Eric Dortmans (e.dortmans@fontys.nl)

Translation (english): Kris Piters (k.piters@fontys.nl)

Helpful tips before you start

You can open a new terminal window with the key combination: *CTRL-ALT-T*.

All commands support *tab-completion*, that is, as soon as you press the *TAB* key, Ubuntu or ROS tries to complete what you have already typed.

You can walk through previously entered commands using the *arrow keys*. A previously entered command can be modified and/or re-executed by pressing the *ENTER* key.

You can stop a program by tapping *CTRL-C* in the relevant terminal window or close the entire terminal window.

Copy-pasting text from and to a terminal can be done using *CTRL-SHIFT-C* and *CTRL-SHIFT-V*.

Some commonly used Linux commands when working with ROS:

- *cd* or *cd ~* (go to your home directory)
- *cd ~/catkin_ws* (go to the top of your workspace, here you have to do *catkin_make* to build your ros packages)
- *cd ~/catkin_ws/src* (go to the src directory of your workspace, here are your packages)

ROS installation & update

If all went well, you’ll have the Kinetic version of ROS installed in Ubuntu. Open a new terminal window and enter the following command to check your ROS version:

```
1 rosversion -d
```

Update Ubuntu and ROS as follows:

```
1 sudo apt-get update && sudo apt-get -y upgrade
```

Now install `git`, a version management tool used for downloading ROS projects (GitHub):

```
1 sudo apt-get -y install git
```

You are now ready to start your ROS exercises.

Creating a ROS workspace

When installing ROS you'll automatically have a nice number of standard packages installed. Often you'd want to create new packages yourself or reuse packages by others. You therefore need a *workspace* for creating, downloading and building those packages

Create a ROS workspace, for example *catkin_ws* (the name of the workspace can be freely chosen but the examples and commands in this exercise assume the default catkin_ws). If you don't have a workspace yet, you can create one as follows:

```
source /opt/ros/kinetic/setup.bash

mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
cd ~/catkin_ws
catkin_make
```

Make this workspace your default workspace (because you can have multiple workspaces):

```
source ~/catkin_ws/devel/setup.bash
```

To have this command executed automatically in every new terminal window you open, you can add it to your *.bashrc* file as follows (it is in your home directory):

```
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
```

If you want to know the contents of your *.bashrc* file you can print the file to your terminal window:

```
cat ~/.bashrc
```

You can also open this file in a text editor (Ubuntu default: gedit):

```
gedit ~/.bashrc
```

To see the values of the ROS environment variables use:

```
env | grep ROS
```

Downloading and installing ROS packages

A lot of packages are installed by default when installing ROS. One of them is *turtlesim*, a 2D simulation package made for educational purposes. Let's see if that package is indeed available.

Is the package `turtlesim` installed?

```
rosversion turtlesim
```

Where is it installed?

```
rospack find turtlesim
```

Some packages aren't installed by default. They are available as a binary download (so-called debian packages).

Which packages are available to extend your ROS installation?

```
apt-cache search ros-kinetic-
```

There are a lot of them! You can find an overview here.

Let's download and install the *turtlebot-teleop* package as an example. Currently we are not interested in the sources, but only want to install and use the package. Let's see if the relevant ROS package is available as a Debian (.deb) installation package. Which debian packages are available with the text *turtlebot* in their name?

```
apt-cache search ros-kinetic- | grep turtlebot
```

Apparently the relevant Debian package is called *ros-kinetic-turtlebot-teleop*. Now that we know this, installing it is a breeze:

```
sudo apt-get install ros-kinetic-turtlebot-teleop
```

Check whether it was successfully installed:

```
rospack find turtlebot_teleop
```

As you can see, this package has been added to the ROS installation and not to your workspace.

Sometimes we'd like to download ROS packages that are only available in source form, or whose sources we'd like to view and perhaps modify. We do not want to add these packages to our ROS installation in binary form, but download them in source form to our workspace.

Let's download some helpful ROS sample packages from GitHub to your workspace:

```
    cd ~/catkin_ws/src
    git clone
   https://github.com/dortmans/ros_examples.git
    cd ~/catkin_ws
    catkin_make
```

If all goes well, ROS can find those packages:

```
rospack find agitr
```

You can also navigate to the directory of the package using roscd(to make changes for example):

```
roscd agitr
```

Nodes, topics, messages

Try starting the node *listener* from the package *beginner_tutorials*:

```
rosrun beginner_tutorials listener
```

Why doesn't it work? How can you fix it?

Open a new terminal window (ctrl-alt-t) and enter the following command:

```
rosnode
```

Running the rosnode command without arguments will list all the possible arguments for the command.

Check which **nodes** are running:

```
rosnode list
```

Which **topics** have been created?

```
rostopic list
```

Take a look at the ROS Graph:

```
rqt_graph
```

Next we are going to explore topics using the rostopic command:

```
rostopic
```

To which topic is the *listener* node subscribed, i.e. to which topic is it listening?

What kind of message can be published on this topic?

```
rostopic type <topic_name>
```

What information is contained inside this message?

```
rosmsg show <message_type>
```

In Linux you can pass the output of a command to another command by means of a *pipe* sign (|):

```
rostopic type <topic_name> | rosmsg show
```

Try to manually publish a message to this topic:

```
rostopic pub <topic_name> <message_type>
    <message_content>
```

Tip: use the *TAB* key after typing the message type.

What happened in the terminal from where you started the *listener* node?

ROS is made to control robots and not to make chat programs. In practice, messages are more complex and nodes will advertise and publish on several topics. Moreover, nodes often have parameters and can be controlled via services in addition to messages via topics. Now let's take a look at a node that a little bit more complex: a 2D robot simulator.

Start the node *turtlesim_node* from the package *turtlesim*

- Which nodes are running?
- Which topics have been created?

Analyse the ROS Computation Graph.

- To which topics is the node *turtlesim* subscribed? To which topics does it publish?
- Which messagetype is published to the topic */turtle1/pose* ?

With the rostopic command you can do much more. Use this command to address the following questions:

- At which frequency (rate) are messages published to this topic?

Analyze the messages that are currently being published on this topic.

- What do you think the published data means?
- Which message type is associated with the topic */turtle1/cmd_vel*?
- What happens when using the following command?

```
rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist
    "{linear: {x: 2.0}}"
```

- And what about this command?

```
rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist
    "{angular: {z: 1.57}}"
```

  Draw a square with the turtle (approximately).

- What happens when you execute the following command?

```
rostopic pub -r 10 /turtle1/cmd_vel geometry_msgs/Twist
   "{linear: {x: 2.0}, angular: {z: 1.8}}"
```

- How does it compare to this command?

```
rostopic pub -r 10 /turtle1/cmd_vel geometry_msgs/Twist
   '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
```

So far we only used one application with only one node. In practice, an application will consist of several nodes. We moved the turtle using ROS commands from the terminal. That is enough to test the node, but of course it is not an integrated application. Suppose we want to control the turtle with the arrow keys of the keyboard. We will use another node to make this possible. This new node will convert input from the keyboard into control commands for the turtle.

Use a new terminal to start the *turtle_teleop_key* node from the *turtlesim* package. You will probably know how to do that by now.

Use the computation graph to see how the *turtle_teleop_key* node is linked to the *turtlesim_node*:

Try using *rostopic echo* to analyze the messages that are being published by *turtle_teleop_key* when you press a certain arrow key.

Services

Topics aren't the only way for nodes to exchange information. A node can also call a service offered by another node (or offer a service itself). We can test a service from a terminal by using the rosservice command.

Which services are offered by the *turtlesim* node?

```
rosservice list
```

What are the arguments for the set_pen service?

```
rosservice args /turtle1/set_pen
```

For more detail:

```
rosservice type /turtle1/set_pen | rossrv show
```

Call the set_pen service (using *rosservice call*) to change the color of the pen to red.

Move the turtle to see if it has worked.

Disable the pen; move the turtle to see if it worked.

Reset turtlesim:

```
rosservice call /reset
```

Parameters

Nodes can also read parameters. These parameters have to be published before starting the node, this usually happens by using startup scripts (launch files).

Which parameters are currently defined?

```
rosparam list
```

Dump all the parameters to a file in YAML format:

```
rosparam dump dump.yaml
```

Let's take a look at that file:

```
cat dump.yaml
```

Change the value of the parameter *background_b*:

```
rosparam set /background_b 0
```

Clear the turtlesim node to make it reread the parameters:

```
rosservice call /clear
```

Add a new turtle to the sim:

```
rosservice call /spawn 2 2 1.57 "r2d2"
```

Which topics are available after doing this?

```
rostopic list
```

Make the newly added turtle move.

Recording and playing back messages

Published messages can be recorded to a *bag* file. Bag files can be used to play back messages at a later time. This is very useful for testing!

Let's start recording messages on the *turtle1/cmd_vel* topic using rosbag:

```
rosbag record -O /tmp/turtle1 /turtle1/cmd_vel
```

Make *turtle1* move.

To stop recording hit *CTRL-C* in the terminal running rosbag.

Reset *turtlesim* to start with a clean slate:

```
rosservice call /reset
```

Play back the recorded messages stored in the *bag file*:

```
rosbag play /tmp/turtle1.bag
```

What happens?

Launch files

Up until now we had to start each node individually from a new terminal window (and *roscore* of course). Starting your nodes this way can be an inconvenience. Luckily ROS has a mechanism to start a whole collection of nodes at once: roslaunch. Your collection of nodes (and their associated parameters) have to be specified for use with the *roslaunch* command. Such a specification is called a *launch file*.

Analyse the content of the launch files in the *agitr* package. The folder containing the launch files can be opened with Ubuntu's filemanager (*nautilus*) using the following command:

```
nautilus `rospack find agitr`/launch
```

Try to launch them with the *roslaunch* command:

```
roslaunch agitr <launchfile_name>
```

Next we are going to make a useful launch file to launch the nodes needed to show a realtime image from a webcam on our screen.

Install a USB camera driver package (usb_cam):

```
sudo apt-get -y install ros-kinetic-usb-cam
```

This package contains a node for controlling the webcam.

We also need a node that is able to read the images and show them on a screen. For this we will use the *image_view* node included in the image_view package. This package should be installed by default.

We want to be able to simultaneously start both nodes with their required parameters. Let's make a launch file to help us achieve this goal. First we'll create a new package that will contain our launch file. Let's call it *usb_camera*.

```
cd ~/catkin_ws/src
catkin_create_pkg usb_camera std_msgs rospy roscpp
cd ~/catkin_ws
catkin_make
```

Let's check if ROS can find our new package:

```
rospack find usb_camera
```

In our still empty package we'll make a directory named *launch*. This directory will contain our launch file.

```
roscd usb_camera
mkdir launch
```

It's probably useful to open this directory with *Nautilus*:

```
nautilus `rospack find usb_camera`/launch
```

Create a new file named *usb_camera.launch* in the *launch* directory and with the following content:

```
<launch>
  <node name="usb_cam" pkg="usb_cam" type="usb_cam_node"
      output="screen" >
    <param name="video_device" value="/dev/video0" />
    <param name="image_width" value="640" />
    <param name="image_height" value="480" />
    <!--<param name="pixel_format" value="mjpeg" /> -->
    <param name="pixel_format" value="yuyv" />
    <param name="camera_frame_id" value="usb_cam" />
    <param name="io_method" value="mmap"/>
  </node>
  <node name="image_view" pkg="image_view"
      type="image_view" respawn="false" output="screen">
    <!-- <remap from="image" to="/usb_cam/image_raw"/>
        -->
    <param name="autosize" value="true" />
  </node>
</launch>
```

PS: Usually a webcam device is registered as */dev/video0* in Ubuntu, in some cases it might be numbered differently eg: */dev/video1*.

PPS: Not every camera is the same. You might have to change the *pixel_format* depending on the camera.

Test the launchfile:

```
roslaunch usb_camera usb_camera.launch
```

Analyze the computation graph:

```
rqt_graph
```

What structural problem can you identify? Hint: play around with rqt_graph's settings to get more information in your graph.

Fix the problem by modifying the launch file. If you succeed you can nicely look at yourself ;-)

rqt

Many things you have done so far (using terminal commands) can also be done using *rqt*. This program provides a graphical user interface (GUI) and can be used to visualize and control many things in the ROS system (messages for example).

Start the *rqt* GUI:

```
1 rqt
```

Explore the available plugins.

Some plugins you already know as we have used them before (rqt_graph, rqt_image_view). *rqt* nicely integrates all of these plugins into a single GUI.

Controlling a real mobile robot

Up until now all nodes have been run on our own laptop. Usually a part of the ROS nodes run on the robot itself. The nodes on our laptop will have to use a network connection (ea. Wifi) to communicate with the nodes running on the robot.

ROS is designed to operate in a distributed way. For this to work you need to know on which machine the ROS Master node (roscore) is running. Usually it'll run on the robot itself. It is sufficient to set the ROS environment variables for ROS_IP and ROS_MASTER_URI to enable communication between your laptop and the robot. ROS_IP (on your laptop) has to be set to the ip address of your laptop.

Make sure the robot and your laptop are both connected to the same network. Then use the following command to view your ip addresses:

```
1 hostname -I
```

ROS_IP will have to be set to the address that partly matches the robot's address (eg 192.168.x.x). If only one address is returned you can simply reuse the hostname command:

```
1 export ROS_IP=`hostname -I`
```

If not, you'll have to set the ip address manually:

```
1 export ROS_IP=<IP_address_of_your_laptop>
```

Next you'll have to set the ROS_MASTER_URI value. This will allow the nodes running on your laptop to access to the remote ROS Master on the robot.

```
export ROS_MASTER_URI=http://<ip_address_of_robot>:11311
```

These exports can also be added to your *.bashrc* file, this way they are automatically executed when opening a new terminal window (but keep mind to disable the lines (by removing or commenting them) when not using a remote master).

```
gedit ~/.bashrc
```

Check if you can communicate with the remote ROS Master (now running on the robot):

```
rostopic list
```

Controlling the robot works similary to the simulated robot in *turtlesim*.

Make the robot drive around in a circle using the *rostopic pub* command.

References

- A Gentle Introduction to ROS
- ROS Tutorials
- YAML on the ROS command line
- rosparam
- roslaunch
- Testing: ROS USB Camera drivers
- usb_cam
- image_view
- rqt