TDDE05: Lab 0: Introduction to ROS

Cyrille Berger

January 25, 2021

Contents

1 Introduction

"*The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms*" [2]. It is commonly use by many robots in research and industry. ROS can transparently interract with real hardware or with a simulator.

The objective of this lab is to facilate first contact to the ROS framework.

2 Running

2.1 On Campus

ROS is already installed on IDA computers, you can find it in the directories */opt/ros/melodic*, */usr* and */courses/TDDE05/software*.

2.2 thinlinc

Running on personnal laptops is currently not supported, however it is possible to use thinlinc to access remotely IDA's computers. To use thinlinc, follow the installation instruction at `http://www.cendio.com/thinlinc/download` and connect to the IDA server: *thinlinc.edu.liu.se*.

3 ROS

ROS is a framework composed of a communication middleware and a set of modules providing different functionnalities for robots.

3.1 middleware

With ROS the various algorithms, interfaces to hardware and simulators all run in different processes. ROS include communication libraries for various programming languages: C++ (roscpp `http://wiki.ros.org/roscpp`) or Python (rospy `http://wiki.ros.org/rospy`). There are libraries for more programming languages, but they are not as well supported and it is recommended that you stick to C++ and Python for this course.

ROS programs mainly communicate through the use of topic. ROS topics follow a multi-publishers/multi-subscribers pattern: ROS programs subscribe to individual topics and they will then receive the messages that are published by other programs on the topic. Topics are associated to a specific type of a ROS message (examples of standard messages can be found at `http://docs.ros.org/kinetic/api/std_msgs/html/index-msg.html`, it is possible to create custom messages to suit your needs).

In addition to topics, ROS programs can offer *service calls*, which is the ROS implementation of Remote Procedure Calls (RPC).

3.2 catkin

ROS also aim to provide a standard method for developing and distributing packages using a tool called *catkin* which relies on the *cmake* build system.

4 Simulator

This year we use *simple simulator*, a very basic 2D simulator, which can run well in thinlinc.

5 Programming a robot

5.1 Get the base code

1. First you will need to add the following to your `.bashrc` file:

```
alias start-tdde05=". /courses/TDDE05/software/bin/start-tdde05.sh"
```

   In every terminal, before issuing a ROS command, you will need to run:

```
start-tdde05
```

2. Then create a ROS workspace for your project:

```
mkdir -p ~/TDDE05/catkin_ws/src
cd ~/TDDE05/catkin_ws/
catkin init
```

   You need to use `~/TDDE05/catkin_ws/` as a directory for the `start-tdde05` script to work properly!

3. Then you need to create a `~/TDDE05/.ros_port` file. This file will be used to set the port number used for communication by ROS. When running on thinlinc, the port number need to be different for each user. In the following command, replace XX with your group number:

```
echo "112XX" > ~/TDDE05/.ros_port
start-tdde5
```

4. Then get the code of your project:

```
cd ~/TDDE05/catkin_ws/src
mkdir air_labs
cd air_labs
git init
```

   Once you have gotten access to your gitlab repository (replace XX with your group number and YY with the correct year number), **the repositories might not be created yet once you start the labs**:

```
git remote add origin git@gitlab.liu.se:tdde05-20YY/air-labs-XX.git
git push -u origin master
```

   Next time you want to access the code from a new account:

```
cd
git clone git@gitlab.liu.se:tdde05-20YY/air-labs-XX.git air_labs
```

5. To build the project (do not forget `start-tdde05` if you are using a new terminal):

```
cd ~/TDDE05/catkin_ws
catkin build
```

   You can run the `catkin build` command from any directory under `\~/TDDE05/catkin_ws`.

6. You are now ready to start programming your robot!

5.2 Start ROS and the simulator

In two different terminals:

- Start the ROS middleware:

```
start-tdde05
roscore
```

- Start the simple simulator:

```
start-tdde05
rosrun air_simple_sim simple_sim.py __ns:=/husky0
```

5.3 RViz

To be able to see what is happening in the simulated world, we can use a tool called `rviz`:

```
start-tdde05
rviz
```

When it opens it shows a window like in figure 1. `rviz` can be used to display the location of a robot and all the sensor data. For now, we will use it to display the location of the robot:

- Press on the button `Add`, at the bottom of the `Displays` panel
- It should show a dialog with a tab called `By display type` (see figure 2)
- Select the item `TF` (circled in red in figure 2)
- Click on `Ok`

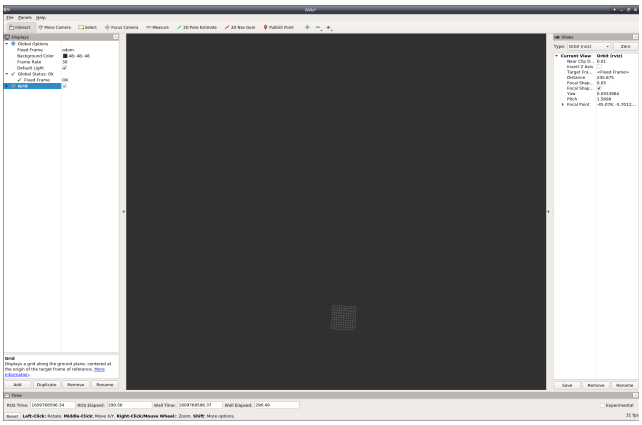

Figure 1: RViz empty window

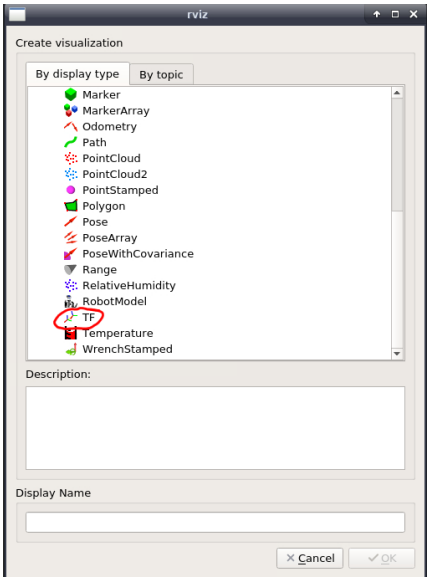

Figure 2: RViz Add dialog

- In the `Displays` panel, make sure that `Fixed Frame` is set to `odom`

The rviz window should look like on figure 3. There are currently two frames, `odom` corresponding to the origin of the world and `husky0/base_footprint` corresponding to the location of the robot.

5.4 Get started with the command line

5.4.1 rostopic

rostopic is a command line tool for interracting with topics. It can be used to get information about a topic, such as the type, publishers and subscribers:

```
1 rostopic info /husky0/odometry
```

rostopic can be used to read what is a published on a topic. The following command can be used to display what the observation of the wheel velocity:

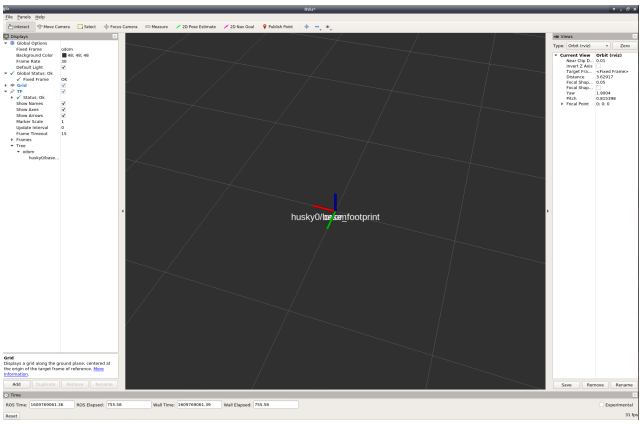

Figure 3: RViz with TF

```
rostopic echo /husky0/odometry
```

rostopic can be used to publish data on a topic:

```
rostopic pub [topicname] [topictype] [value in yaml]
```

For example to command some velocity to your robot, first you can check the topic type with. You can use tab completion to help with finding topicname, tyopictype and formatting the yaml value.

```
rostopic info /husky0/wheel_velocity_cmd
```

The *sensor_msgs/JointState* (`http://docs.ros.org/api/sensor_msgs/html/msg/JointState.html`) message has the following structure:

```
Header header

string[] name
float64[] position
float64[] velocity
float64[] effort
```

The YAML format use a key/value structure, and you do not need to specify all the field for your message, the following command will make your robot move in straight line with the maximum velocity of $1.0 m/s$.

```
rostopic pub /husky0/wheel_cmd sensor_msgs/JointState "name: ['left', 'right']
effort: [1, 1]"
```

It essentially tells the robot to apply full power on the left and right wheels.
You can check the velocity of your robot with:

```
rostopic echo /husky0/odometry
```

You can check the velocity of the wheels of your robot with:

```
rostopic echo /husky0/wheel_velocities
```

5.5 Create a package

We will use the command *catkin_create_pkg* to create a new package for the tool that we are going to write in this lab.

```
start-tdde05
cd ~/TDDE05/catkin_ws/src/air_labs
catkin_create_pkg air_lab0 rospy
```

This will create a package called *air_lab0* which depends on *rospy*. In practice, it creates a directory called *air_lab0* containing two files:

Then after creating a package, even if it is only python, you need to *build it* at least once and reload the environment:

```
catkin build
start-tdde05
```

1. *package.xml*: meta information about your package: authors, license and more importantly ROS packages dependencies
2. *CMakeLists.txt*: this is the file used by the build system to compile your ROS programs. This file contains a lot of documentation on how to use it.

5.6 Saving rviz configuration

You can now save the rviz configuration in your `air_lab0` package. Go in the menu `File` and save your configuration in `~/TDDE05/catkin_ws/src/air_labs/air_lab0/rviz/labs.rviz`.

You can use to start `rviz` with your configuration file:

```
rviz -d `rospack find air_lab0`/rviz/labs.rviz
```

5.7 Basic Husky Controller (in Python)

In this part, you will write a basic controller that takes a string as input and send some simple effort to the simulator.

We will listen on a topic called `/husky0/text_command` using a `std_msgs/String` message. You can use the following command to see the structure of the message:

```
  rosmsg info std_msgs/String
```

```
cd ~/TDDE05/catkin_ws/src/air_labs/air_lab0/src
touch simple_text_controller.py
chmod u+x simple_text_controller.py
```

Then in your favorite text editor:

```
#!/usr/bin/env python

import math

# rosoy is the main API for ROS
import rospy

# import library with ros messages
import sensor_msgs.msg
import std_msgs.msg

#
# Class that contains the logic for our simple controller:
#   - listen to the "text_cmd" topic, accept
#     "forward", "backward", "left", "right" and "stop"
#     as valid commands.
#   - publish a wheel command on the "wheel_cmd" topic
#
class simple_text_controller:
    def __init__(self):
        # Subscribe to the "text_cmd" topic which has
        # "std_msgs/String" as message type.
        # It will call "self.command_callback" every
        # time a new message arrive
        self.command_sub =  rospy.Subscriber("text_cmd",
                  std_msgs.msg.String, self.command_callback)

        # Publisher to the "wheel_cmd" topic which has
        # "sensor_msgs/JointState" as message type.
        self.effort_pub = rospy.Publisher("wheel_cmd",
                  sensor_msgs.msg.JointState, queue_size = 1)

    # Callback called everytime a new command message
    # is received
    def command_callback(self, command_msg):
        # Compute the effort depending on the received
        # command.
        if command_msg.data == "forward":
            left_effort = 0.5
```

```
            right_effort = 0.5
        elif command_msg.data == "left":
            left_effort = -0.5
            right_effort = 0.5
        # TODO backward and right
        else:
            # anything else
            left_effort   = 0.0
            right_effort  = 0.0

        # Fill the effort_msg with the command values
        effort_msg = sensor_msgs.msg.JointState()
        effort_msg.name = ["left", "right"]
        effort_msg.effort = [left_effort, right_effort]
        self.effort_pub.publish(effort_msg)

if __name__ == '__main__':
    # Initialise the ROS sub system
    rospy.init_node('simple_text_controller', anonymous=False)
    # Create an instance of our controller
    ec = simple_text_controller()
    # Start listening to messages and loop forever
    rospy.spin()
```

Expand with command for turning right and going backward.
You can run it with, in a new terminal:

```
start-tdde05
rosrun air_lab0 simple_text_controller.py __ns:=/husky0
```

To send command to it, in a new terminal:

```
start-tdde05
rostopic pub /husky0/text_cmd std_msgs/String "data: 'forward'"
```

5.8 Get started with GNU Screen

It can be annoying to have so many windows and terminals open, a solution is to use GNU Screen, which allows to start all the processes in a single terminal window.

The easiest way to use it is to create a screen configuration file:

1. In the `air_lab0` project:

```
roscd air_lab0
mkdir screen
```

2. Then in that directory you can create a file called *labs*, with the following content (replace the ... accordingly):

```
# Configuration
deflogin on
autodetach on

caption always

bindkey ^w screen
bindkey ^p prev
bindkey ^n next
bindkey ^x quit
bind q quit
bindkey ^l windowlist
bindkey ^e copy

# Pre-defined tabs

screen 0
title "roscore"
stuff "start-tdde05\015"
stuff "roscore\015"

screen 1
title "simple sim - no obstacles"
stuff "start-tdde05; rosrun air_simple_sim simple_sim.py __ns:=/husky0\015"

screen 2
title "rviz"
stuff "start-tdde05; rviz -d `rospack find air_lab0`/rviz/labs.rviz\015"

screen 3
title "simple_text_controller"
stuff "start-tdde05; rosrun air_lab0 simple_...oller.py __ns:=/husky0\015"
```

3. Then to start it:

```
rosscreen air_lab0 labs
```

The following shortkeys can be used:

- `ctrl+w` to create a new command tab
- `ctrl+p` to navigate to the previous tab
- `ctrl+n` to navigate to the next tab

- `ctrl+x` to quit
- `ctrl+l` to show the list of tabs
- `ctrl+e` to scroll in the log, using up/down arrow (page up/down works as well), press `ESC` to stop scrolling

In the screen configuration file:

- *screen ##* indicates a new tab with number *##*
- *title* indicates the name of the tab (as shown in the list of tabs)
- *stuff* indicates the command that is run in the tab, the `\015` at the end indicates whether the command is ran when starting the screen or if you have to start it manually

5.9 Creating user interface with ROS

There is the possibility in ROS to build user interface with a program called *rqt*, which can be used for visualisation and for giving commands.

For this lab we are going to use it to send commands to the text controller and monitor some topics. In a terminal, you can launch:

```
1 rqt
```

This should show an empty application with a menu:

- in the menu, select *Plugins*, *Robot topics*, *Message publisher* and *Topic Monitor*
- in the `Message Publisher` panel, add the `/husky0/text_cmd` topic five times and then set the message in the `expression` column to `forward`, `backward`, `left`, `right` and `stop` respectively.
- Use the `Topic Monitor` panel to monitor the odometry and wheel command topics.

This should show an interface similar to figure 4. In the editor line you can type the name of the topic where you want to publish a velocity.

Add the rqt command to your `screen` file.

5.10 Demonstration

- Show your `screen` file
- Show in `rviz` how your robot respond to the `forward`, `backward`, `left`, `right` and `stop` command using `rqt`.

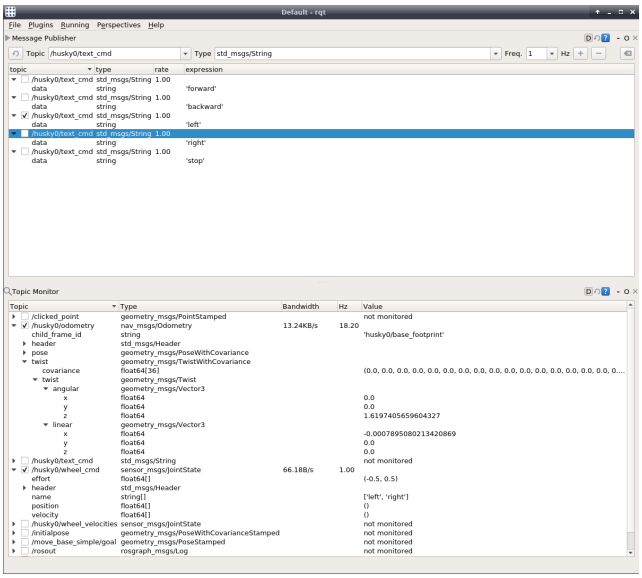

Figure 4: *rqt* with a *Message Publisher* and *Topic Monitor* panel.

References

[1] Modular openrobots simulation engine (morse). `http://morse.openrobots.org/`.

[2] Robot operating system (ros). `http://www.ros.org/`.