TDDE05: Lab 0: Introduction to ROS and Morse

Cyrille Berger

February 2, 2018

Contents

1 Introduction **1**

2 Running **2**
2.1 On Campus 2
2.2 thinlinc 2

3 ROS **2**
3.1 middleware 2
3.2 catkin 2

4 Morse **2**

5 Programming a robot **3**
5.1 Get the base code 3
5.2 Start ROS and the simulator 3
5.3 Get started with the command line 4
5.3.1 rostopic 4
5.4 Create a package 5
5.5 Get started with C++ 5
5.6 Creating user interface with ROS 10
5.7 Get started with Python 10

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.

"*MORSE is a generic simulator for academic robotics. It focuses on realistic 3D simulation of small to large environments, indoor or outdoor, with one to tenths of autonomous robots*" [1].

The objective of this lab is to facilate first contact to ROS framework and MORSE simulator.

2 Running

2.1 On Campus

ROS and MORSE are already installed on IDA computers, you can find them in the directory */home/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-und.ida.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 Morse

MORSE is a simulator based on the blender game engine. World and robots can be designed using the blender user interface. MORSE allow to control simulation from the command line, through python script and provide an interface with ROS. With MORSE

you can define a robot, with actuators (such as a robotic arm) and many different sensors (IMU, camera, lidar...).

5 Programming a robot

5.1 Get the base code

1. First you will need to add TDDE05 module files:

```
module add /home/TDDE05/modulefiles/tdde05
module initadd /home/TDDE05/modulefiles/tdde05
```

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

2. Then create a ROS workspace for your project:

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

3. 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):

```
git remote add origin git@gitlab.ida.liu.se:tdde05-2018/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.ida.liu.se:tdde05-2018/air-labs-XX.git air_labs
```

4. To build the project (do not forget `load_ros` 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`.

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

5.2 Start ROS and the simulator

In two different terminals:

- Start the ROS middleware:

```
load_ros
roscore
```

Figure 1: MORSE Simulator with a Husky robot

- Start the morse simulator with the environment used for the labs:

```
load_ros
roscd air_morse/worlds
morse run lab_0123_world.py
```

The simulator should show a window as displayed on figure 1.

5.3 Get started with the command line

5.3.1 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:

```
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:

```
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
```

5.4 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.

```
load_ros
cd ~/TDDE05/catkin_ws/src/air_labs
catkin_create_pkg air_lab0 roscpp rospy message_generation std_msgs
```

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

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.

5.5 Get started with C++

The goal of this lab is to create a small C++ program that implement a very basic control system and it takes as input a target velocity command, the actual velocity of the robot and try to adjust the commanded velocity of the robot.

```
cd ~/TDDE05/catkin_ws/src/air_labs/air_lab0/src
touch simple_controller.cpp
```

Then you can open *simple_controller.cpp* in your favorite text editor.

First we need to include a few headers:

```
// Include for ROS framework
#include <ros/node_handle.h>
#include <ros/publisher.h>
#include <ros/subscriber.h>

// Include for the messages used in this script
#include <nav_msgs/Odometry.h>
#include <geometry_msgs/Twist.h>
#include <sensor_msgs/JointState.h>
```

Then we will define a class for our simple controller:

```
/**
 * Class that will contains the logic for our simple controller.
 */
class SimpleController
{
public:
  SimpleController(const ros::NodeHandle& _nodeHandle);
public:
  /**
   * This callback is called when a new odometry message is
   * received on the odometry topic
   */
  void odometryCallback(const nav_msgs::OdometryPtr& _odometry);
  /**
   * This callback is called when a new twist message is
   * received on the desired velocity topic
   */
  void desiredVelocityCallback(const geometry_msgs::TwistPtr& _twist);
private:
  /**
   * Function used to compute the controlled velocity and
   * publish it.
   */
  void publishControlledVelocity();
  /**
   * Member used to keep a copy of the last messages received.
   */
  geometry_msgs::TwistPtr m_lastDesiredVelocityMsg;
  nav_msgs::OdometryPtr m_lastOdometryMsg;
  /**
   * ROS's node handle, subscriber and publisher object. See
   * later for an explanation.
   */
  ros::NodeHandle m_nodeHandle;
  ros::Subscriber m_odometrySub, m_desiredVelocitySub;
  ros::Publisher m_wheelCmdPub;
};
```

Then we will add the main function:

```
int main(int argc, char** argv)
{
```

Then we will initiliase ROS.

The ros::init() function needs to see argc and argv so that it can perform any ROS arguments and name remapping that were provided at the command line. For programmatic remappings you can use a different version of init() which takes remappings directly, but for most command-line programs, passing argc and argv is the easiest way to do it. The third argument to init() is the name of the node.

You must call one of the versions of ros::init() before using any other part of the ROS system.

```
49    ros::init(argc, argv, "simple_controller");
```

NodeHandle is the main access point to communications with the ROS system. The first NodeHandle constructed will fully initialize this node, and the last NodeHandle destructed will close down the node.

```
50    ros::NodeHandle n;
51
52    /**
53     * Create an instance of our simple controller.
54     */
55    SimpleController sc(n);
56
57    /**
58     * Start listening for messages and loop for ever.
59     */
60    ros::spin();
61    return 0;
62  }
63
64  SimpleController::SimpleController(const ros::NodeHandle& _nodeHandle)
65    : m_nodeHandle(_nodeHandle)
66  {
```

The subscribe() call is how you tell ROS that you want to receive messages on a given topic. This invokes a call to the ROS master node, which keeps a registry of who is publishing and who is subscribing. Messages are passed to a callback function, here called odometryCallback. subscribe() returns a Subscriber object that you must hold on to until you want to unsubscribe. When all copies of the Subscriber object go out of scope, this callback will automatically be unsubscribed from this topic.

The second parameter to the subscribe() function is the size of the message queue. If messages are arriving faster than they are being processed, this is the number of messages that will be buffered up before beginning to throw away the oldest ones.

```
67    m_odometrySub = m_nodeHandle.subscribe(
68              "odometry", 1, &SimpleController::odometryCallback, this);
69    m_desiredVelocitySub = m_nodeHandle.subscribe(
70              "desired_velocity", 1, &SimpleController::desiredVelocityCallback, thi
```

The advertise() function is how you tell ROS that you want to publish on a given topic name. This invokes a call to the ROS master node, which keeps a registry of who is publishing and who is subscribing. After this advertise() call is made, the master node will notify anyone who is trying to subscribe to this topic name, and they will in turn negotiate a peer-to-peer connection with this node. advertise() returns a Publisher object which allows you to publish messages on that topic through a call to publish(). Once all copies of the returned Publisher object are destroyed, the topic will be automatically unadvertised.

The second parameter to advertise() is the size of the message queue used for publishing messages. If messages are published more quickly than we can send them, the

number here specifies how many messages to buffer up before throwing some away.

```
  m_wheelCmdPub
          = m_nodeHandle.advertise<sensor_msgs::JointState>(
                                    "wheel_cmd", 1);
}
void SimpleController::odometryCallback(
                          const nav_msgs::OdometryPtr& _odometry)
{
  // Keep a copy of the message and publish an update of the
  // controlled velocity
  m_lastOdometryMsg = _odometry;
  publishControlledVelocity();
}

void SimpleController::desiredVelocityCallback(
                           const geometry_msgs::TwistPtr& _twist)
{
  // Keep a copy of the message and publish an update of the
  // controlled velocity
  m_lastDesiredVelocityMsg = _twist;
  publishControlledVelocity();
}

void SimpleController::publishControlledVelocity()
{
  if(m_lastDesiredVelocityMsg and m_lastOdometryMsg)
  {
    sensor_msgs::JointState cmd;
    // This is now up to you, to compute the cmd velocity
    // You can use a simple algorithm and increment the effort until the velocity
    cmd.name = {"left", "right"};
    cmd.effort = {0.0, 0.0};
    m_wheelCmdPub.publish(cmd);
  }
}
```

Then we need to create an executable, open the file *CMakeLists.txt*, after the line *## Build ##* add the following:

```
# Tell to create a simple_controller executable
add_executable(simple_controller src/simple_controller.cpp)
# Tell to link it to roscpp
target_link_libraries(simple_controller
  ${catkin_LIBRARIES}
)
```

Then in a terminal, you can build your new ROS program:

```
cd ~/TDDE05/catkin_ws/src/air_lab0/
catkin build air_lab0
```

Then in a terminal, you can run your new ROS program:

```
rosrun air_lab0 simple_controller
```

At this point, the topics of your new ROS program are not connected to the robot, and your new program should not work, to see what are the potential problem, you can run the command *roswtf*, this should output:

```
 WARNING The following node subscriptions are unconnected:
 * /simple_controller:
   * /odometry
   * /desired_velocity
```

You can run the command *rosnode info simple_controller* to get some information about your ROS node, you should see the list of published and subscribed topics as well as services. You can do the same for the simulator with *rosnode info morse*, you will then notice that the topics for the robot start with */husky0*, this is called a namespace. We can start the simple controller with a namespace, using the *__ns:=/husky0* argument:

```
rosrun air_lab0 simple_controller __ns:=/husky0
```

Now to access the information about your node, you need to run *rosnode info /husky0/simple_controller*. This allow to run multiple simple_controller with the same *roscore*. Now you can also notice that the topics are prefixed with */husky0* which means that the odometry topics are now connected, as you can see with *roswtf*.

/husky0/desired_velocity is also not connected, but this is our input, we will not fix that for now. However, on the simulator we should see that the *velocity_cmd* topic is not connected (if you do not see that, make sure no *rostopic pub* command is running). We need to connect */husky0/controlled_velocity* which is the output of the simple controller to *velocity_cmd* which is what the simulator expect. We can either solve this by changing the name of the topic in the code, or by remapping the topic. We are going to use the later to demonstrate how we can adapt different modules to our hardware.

To remap a topic we can simply add on the command line *controlled_velocity:=velocity_cmd* and this will change the name of the topic output by our simple controller:

```
rosrun air_lab0 simple_controller __ns:=/husky0
                controlled_velocity:=velocity_cmd
```

You can check the new name of the topic with *rosnode info /husky0/simple_controller* and that *roswtf* does not show any warning about */husky0/velocity_cmd* anymore.

You can now send command to your controller with:

```
rostopic pub /husky0/desired_velocity geometry_msgs/Twist
    "{ linear: [0.1, 0, 0], angular: [0.0, 0, 0.1] }"
```

Since typing all those command lines all the time is not so convenient, we can use launch files to define how to launch our ros program.

```
cd ~/TDDE05/catkin_ws/src/air_labs/air_lab0/
mkdir launch
cd launch
touch simple_controller.launch
```

Then in your favorite text editor

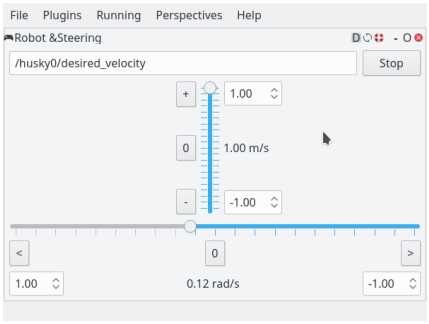

Figure 2: RQT's robot steering

```
<launch>
  <!-- Launch a node of type simple_controller from the package
       air_lab0 give it the name "husky_simple_controller" -->
  <node name="husky_simple_controller" pkg="air_lab0"
        type="simple_controller">
    <remap from="controlled_velocity" to="velocity_cmd" />
  </node>
</launch>
```

Then we can run it using roslaunch *roslaunch*:

```
roslaunch air_lab0 simple_controller.launch __ns:=/husky0
```

It is possible to define the namespace in the launch file, but this would bind the launch file to use with *husky0* for sake of generality, it is better to pass the namespace on the command line.

5.6 Creating user interface with ROS

Since inputing the velocity in the terminal with the rospub command is not convenient, 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 velocity commands, in a terminal, you can launch:

```
rqt
```

This should show an empty application with a menu, in the menu, select *Plugins*, *Robot tool*, *Robot steering*. This should show an interface similar to figure 2. In the editor line you can type the name of the topic where you want to publish a velocity.

5.7 Get started with Python

In this section, we are going to look at how to use python to write scripts for ROS. We are goint to write a small script that estimate how well our controller manage to keep the commanded velocity.

We will publish the performance of our controller in a topic, in a new message. To create a message:

```
cd ~/TDDE05/catkin_ws/src/air_labs/air_lab0/
mkdir msg
cd msg
touch ControllerEvaluationStat.msg
```

Then in your favorite editor:

```
int32 samples           # Number of samples used for computing
                        # the error
float64 last_error      # Last error
float64 average_error   # Average error
```

Then in CMakeLists.txt, find the section with *add_message_files* and add:

```
add_message_files(
  FILES
  ControllerEvaluationStat.msg
)
generate_messages(DEPENDENCIES   std_msgs )
```

Then to generate the code for the message, you need to build the module:

```
cd ~/TDDE05/catkin_ws/src/air_labs/air_lab0/
catkin build air_lab0
```

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

Then in your favorite text editor:

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

import rospy
import nav_msgs.msg
import geometry_msgs.msg
import air_lab0.msg
import math

# C++ API is very similar to the Python API and therefore the
# explanation for C++ apply to Python as well.

class evaluate_controller:
    def __init__(self):
        self.odometry_sub =  rospy.Subscriber("odometry",
                  nav_msgs.msg.Odometry, self.odometry_callback)
        self.velocity_sub =  rospy.Subscriber("desired_velocity",
                  geometry_msgs.msg.Twist,
                  self.desired_velocity_callback)

        self.stat_pub = rospy.Publisher("evaluation_stat",
                  air_lab0.msg.ControllerEvaluationStat)

        self.last_desired_velocity = None
        self.samples        = 0.0
        self.average_error  = 0.0
    def odometry_callback(self, odometry):
        if(self.last_desired_velocity):
          # Use a root square error
          last_error = ?
          self.average_error = ?
          self.samples += 1.0
          msg = air_lab0.msg.ControllerEvaluationStat()
          msg.last_error = ?
          msg.average_error = ?
          msg.samples = int(self.samples)
          self.stat_pub.publish(msg)

    def desired_velocity_callback(self, velocity):
        self.last_desired_velocity = velocity

if __name__ == '__main__':
    rospy.init_node('evaluate_controller', anonymous=False)
    ec = evaluate_controller()
    rospy.spin()
```

References

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

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