ROS & GAZEBO: Deployment on a line of a multi-robot system

If you want to make the LAB1 in ROS and Gazebo, you can use as a starting point or an example the files provided via moodle.

Note: files tested under Ubuntu 16.04 LTS, ROS Kinetic and Gazebo (either in virtual box or in a partition). Some students have reported success also in Ubuntu 18 and 20 without apparently too many problems. If you use other combinations of Ubuntu/ROS, or other language, things may or may not be like here (it is at your own risk…)

In case you need it, you have here several interesting tutorials on ROS:
http://wiki.ros.org/ROS/Tutorials

Gazebo files to simulate turtlebot3

Instead of dots acting as robots, in this session we are going to use a simulator, Gazebo, and a model of a real robot (a turtlebot3 waffle_pi).
Here, we are including the associated packages and will test that everything works fine.

You can download them and configure them (or instal them) as it is commented here:
https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/
Recall that if you use other combinations of Ubuntu, ROS, etc., it is at your own risk.

The key steps you need to make are:
From https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/
Download the souces and compile them.
Also in this folder, download and compile the source of the package for making the simulations of the turtlebot3, as it is explained here:
https://emanual.robotis.com/docs/en/platform/turtlebot3/simulation/

In the example, we will be using waffle_pi turtlebot robots
$ export TURTLEBOT3_MODEL=waffle_pi

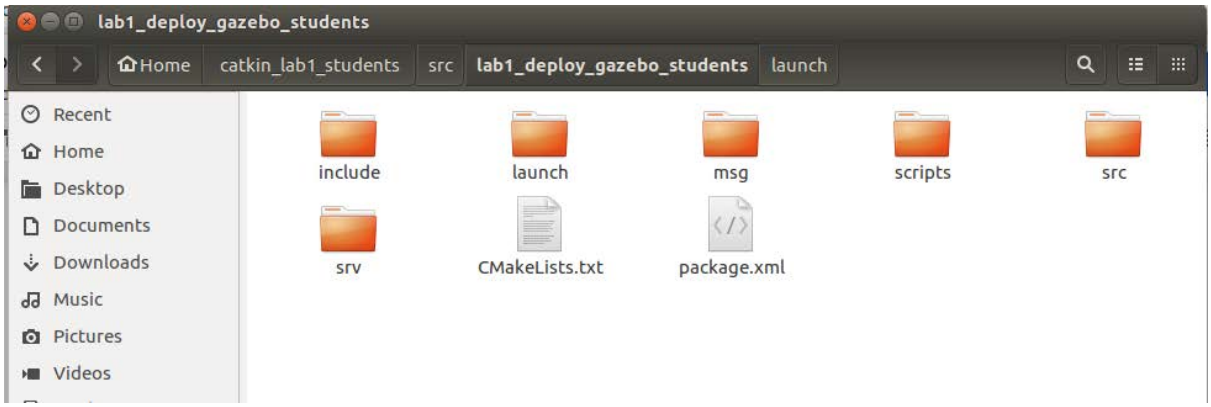
Now, to test that everything works fine, we will use some basic examples extracted from the tutorials.
https://emanual.robotis.com/docs/en/platform/turtlebot3/simulation/

We will load in Gazebo a room, will place a robot there, and will make the robot move randomly. In separate terminals, do:
$ roscore
$ roslaunch turtlebot3_gazebo turtlebot3_house.launch
$ roslaunch turtlebot3_gazebo turtlebot3_simulation.launch
$ roslaunch turtlebot3_gazebo turtlebot3_gazebo_rviz.launch

If things go fine, you should see a turtlebot 3 robot moving around a hose, randomly and avoiding obstacles, both in Gazebo simulation and in the RVIZ visualizer.

Naive turtlebots (draft files provided)

To make things easier, we have prepared a package **lab1_deploy_gazebo_students.** You can spend some time observing the provided files (launch, msg, srv, scripts..). The files CMakeLists.txt and package.xml are already prepared for working with these files. Your catkin folder should look like this:



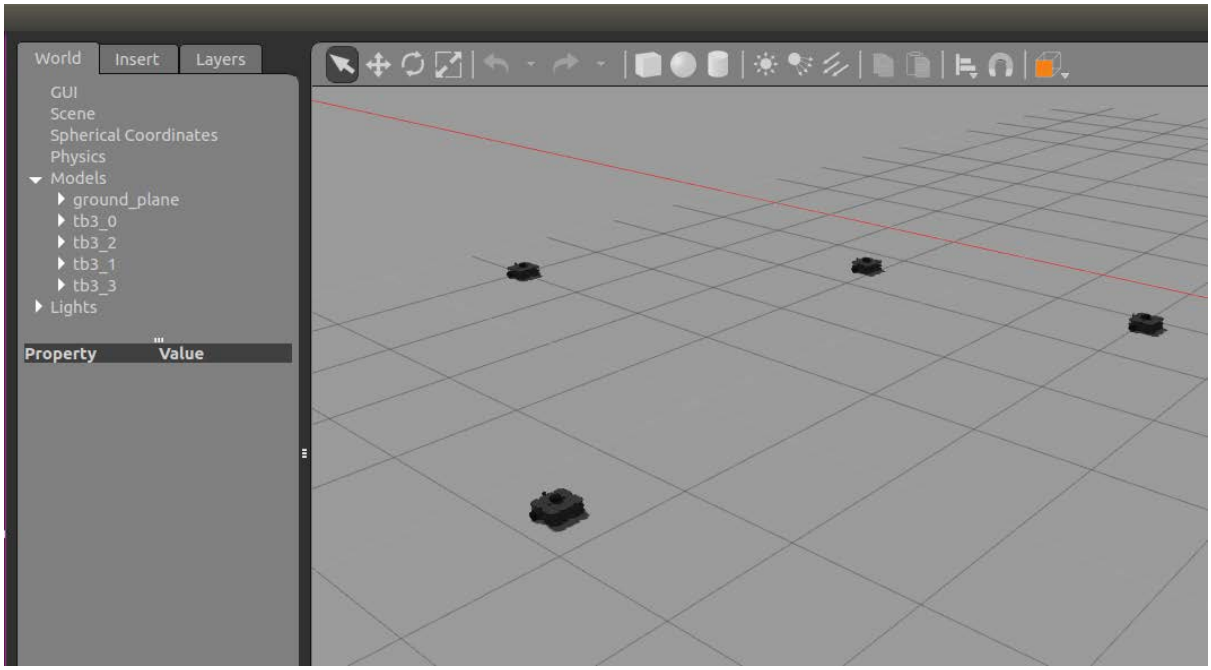
The launch file **multi_turtlebot3_4.launch** is provided. It creates the world in **Gazebo**, and spawns several turtlebot3 robots in Gazebo, each one with its own namespace. These robots are equipped with several sensors, and the measurements are published in different topics. The launch file **multi_turtlebot_goTogoal.launch** starts several nodes of type "**goTogoal_turtlebot.py**". These nodes listen to a topic associated with information about the goal x,y position for a particular turtlebot3. Then, they implement a local navigation method for making the gazebo robots to drive towards this goal position. Finally, the launch file **bunch_of_depolyment_turtles.launch** starts several nodes of type "**deployment_turtlebot.py**". In these nodes, you will have to implement the deployment strategy. (Feel free to modify all the msg and srv message types, to pass any additional argument you may need, and to make changes to any python script / launch file as desired.)

To check that everything works fine, you can run in different terminals:

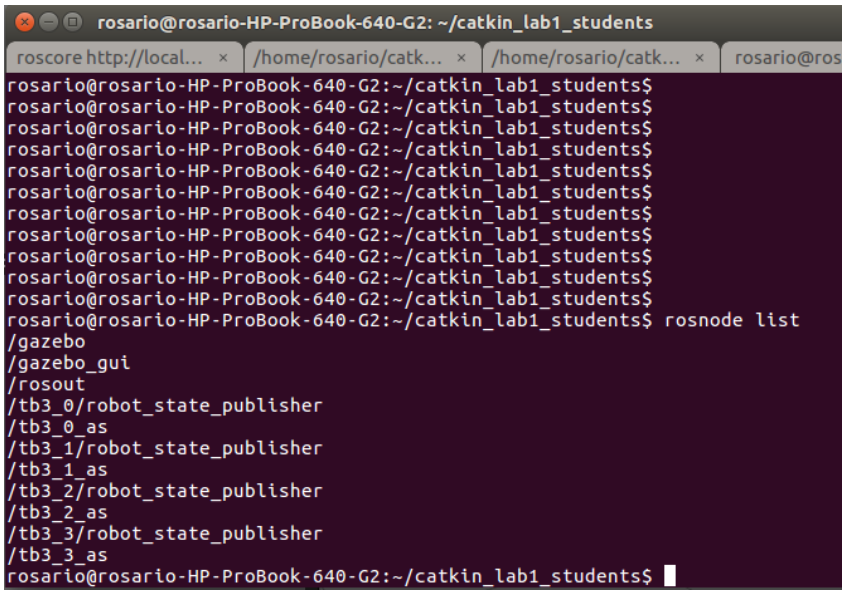
```
$ roscore
$ roslaunch lab1_deploy_gazebo_students multi_turtlebot3_4.launch
```

(Sometimes this fails, especially the first time it is run after making changes. So just wait until Gazebo starts, and the robots are spawned. If it fails, just kill this and launch it again)

You should see something like this (four robots placed in Gazebo):

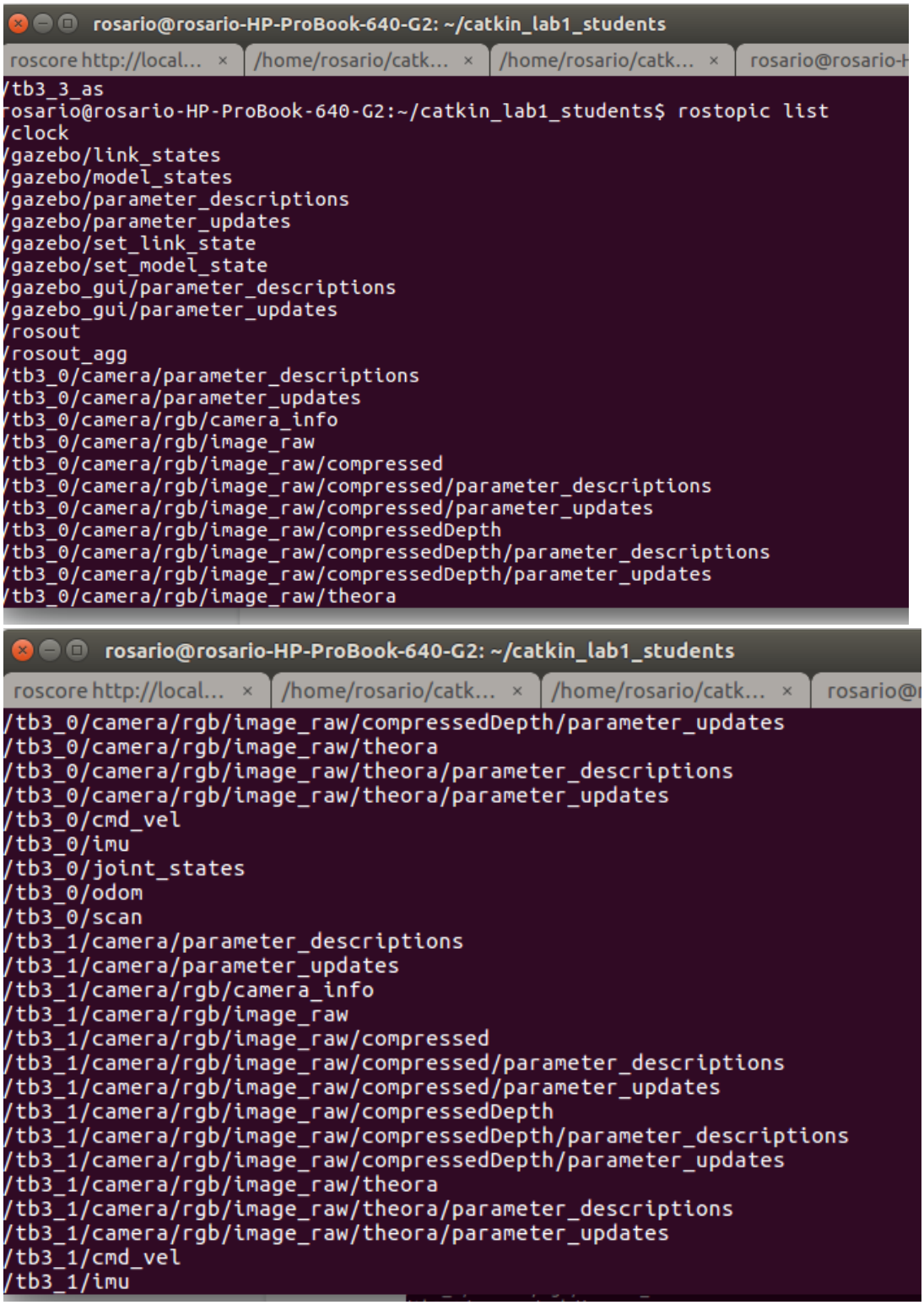


In a new terminal, launch **multi_turtlebot_goTogoal.launch**
If everything went well, you should now see that new nodes, topics, etc. appeared. Take a look at the code and launch files, and establish connections between the nodes, services, topics etc. that are available and how they were defined in the implemented nodes.



Some of these elements, are provided by Gazebo (sensor readings, robot ground truth position, etc.) For instance, you will find the following nodes:
tb3_n/robot_state_publisher: to provide the state of the robot
tb3_n_as: nodes in charge of the local level navigation

Topics:


rosario@rosario-HP-ProBook-640-G2: ~/catkin_lab1_students
roscore http://local... ×
/home/rosario/catk... ×
/home/rosario/catk... ×
/tb3_3_as
rosario@rosario-HP-ProBook-640-G2:~/catkin_lab1_students$ rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/gazebo_gui/parameter_descriptions
/gazebo_gui/parameter_updates
/rosout
/rosout_agg
/tb3_0/camera/parameter_descriptions
/tb3_0/camera/parameter_updates
/tb3_0/camera/rgb/camera_info
/tb3_0/camera/rgb/image_raw
/tb3_0/camera/rgb/image_raw/compressed
/tb3_0/camera/rgb/image_raw/compressed/parameter_descriptions
/tb3_0/camera/rgb/image_raw/compressed/parameter_updates
/tb3_0/camera/rgb/image_raw/compressedDepth
/tb3_0/camera/rgb/image_raw/compressedDepth/parameter_descriptions
/tb3_0/camera/rgb/image_raw/compressedDepth/parameter_updates
/tb3_0/camera/rgb/image_raw/theora
rosario@rosario-HP-ProBook-640-G2: ~/catkin_lab1_students
roscore http://local... ×
/home/rosario/catk... ×
/home/rosario/catk... ×
/tb3_0/camera/rgb/image_raw/compressedDepth/parameter_updates
/tb3_0/camera/rgb/image_raw/theora
/tb3_0/camera/rgb/image_raw/theora/parameter_descriptions
/tb3_0/camera/rgb/image_raw/theora/parameter_updates
/tb3_0/cmd_vel
/tb3_0/imu
/tb3_0/joint_states
/tb3_0/odom
/tb3_0/scan
/tb3_1/camera/parameter_descriptions
/tb3_1/camera/parameter_updates
/tb3_1/camera/rgb/camera_info
/tb3_1/camera/rgb/image_raw
/tb3_1/camera/rgb/image_raw/compressed
/tb3_1/camera/rgb/image_raw/compressed/parameter_descriptions
/tb3_1/camera/rgb/image_raw/compressed/parameter_updates
/tb3_1/camera/rgb/image_raw/compressedDepth
/tb3_1/camera/rgb/image_raw/compressedDepth/parameter_descriptions
/tb3_1/camera/rgb/image_raw/compressedDepth/parameter_updates
/tb3_1/camera/rgb/image_raw/theora
/tb3_1/camera/rgb/image_raw/theora/parameter_descriptions
/tb3_1/camera/rgb/image_raw/theora/parameter_updates
/tb3_1/cmd_vel
/tb3_1/imu

(...)

```
/tb3_3/imu
/tb3_3/joint_states
/tb3_3/odom
/tb3_3/scan
/tf
/tf_static
/topic_GoToGoal_goal0
/topic_GoToGoal_goal1
/topic_GoToGoal_goal2
/topic_GoToGoal_goal3
rosario@rosario-HP-ProBook-640-G2:~/catkin_lab1_students$
```

The dummy implementation given at "**deployment_turtlebot.py**" does nothing useful (just take a look at the code).

```
#!/usr/bin/env python
# license removed for brevity
import rospy
import actionlib
from std_msgs.msg import String
from lab1_deploy_gazebo_students.srv import gossip_update, gossip_updateResponse
from lab1_deploy_gazebo_students.msg import GoToGoal_goal, queue_position_plot
import sys
from math import radians, copysign, sqrt, pow, pi, atan2
import numpy as np
import tf

class Robot():
	def __init__(self, robot_id):
	# Class attributes
		self.robot_id = robot_id
		self.x = 0.0
		self.y = 0.0
		self.pub = rospy.Publisher('topic_queue_position_plot',
queue_position_plot, queue_size=10)
		self.pub_goTogoal =
rospy.Publisher('topic_GoToGoal_goal'+str(self.robot_id), GoToGoal_goal,
queue_size=10)
		self.ser = rospy.Service('gossip_update'+str(self.robot_id),
gossip_update, self.handle_gossip_update)
		self.act_dummy()

	def act_dummy(self):

		rate = rospy.Rate(1/2.0) # once every two seconds
		if (self.robot_id == 0):
			neig_id = 1
			rospy.wait_for_service('gossip_update'+str(neig_id)) # ask for
service gossip_update
			try:
				print("I request a service")
				service_gossip_update =
rospy.ServiceProxy('gossip_update'+str(neig_id), gossip_update)
				resp1=service_gossip_update(self.robot_id, 0.0, 1.0)
				print('Reply received: (', resp1.avg_x, ',',
resp1.avg_y)

			except rospy.ServiceException as e:
```

```
                        print("Service call failed: %s"%e)

                    print("I publish at topic_queue_position_plot")
                    my_pos_plot=queue_position_plot()
                    my_pos_plot.robot_id=self.robot_id
                    my_pos_plot.x=3.0
                    my_pos_plot.y=4.0
                    self.pub.publish(my_pos_plot)

                    print("I publish a navigation goal at
topicGoToGoal_goal"+str(self.robot_id))
                    next_pos = GoToGoal_goal()
                    next_pos.goal_coords=[-1.0, 4.0]
                    next_pos.goal_z=0.0 #currently, not used
                    next_pos.speed=0.0 #currently, not used
                    self.pub_goTogoal.publish(next_pos)
            #Up the here, endif

            while not rospy.is_shutdown():
                    hello_str = "hello world %s" % rospy.get_time()
                    rospy.loginfo(hello_str)
                    rate.sleep()

      def handle_gossip_update(self, req):
            # Gossip and use of bx_ij, by_ij for deploying on a line
            print("I am robot "+str(self.robot_id)+" and I received a
gossip_update request: "+str(req.x)+","+str(req.y))
            myResponse=gossip_updateResponse()
            myResponse.avg_x=10.3
            myResponse.avg_y=14.6
            return myResponse


if __name__ == '__main__':
      #Input arguments (robot id, x0, y0, Tlocal, neig1, neig2... neign)
      sysargv = rospy.myargv(argv=sys.argv) # to avoid problems with __name:=
elements.
      num_args=len(sysargv)
      if (num_args >= 1):
            robot_id = int(sysargv[1])
      else:
            robot_id=0
      try:
            rospy.init_node('deploy_'+str(robot_id), anonymous=False)
            my_naive_robot=Robot(robot_id)
            print('Finished!')
      except rospy.ROSInterruptException:
            pass
```

Once you test the behavior of this node, e.g., with rosrun:

```
$ roscore
$ roslaunch lab1_deploy_gazebo_students multi_turtlebot3_4.launch
$ roslaunch lab1_deploy_gazebo_students multi_turtlebot_goTogoal.launch
$ roslaunch lab1_deploy_gazebo_students bunch_of_depolyment_turtles.launch
```

You will see that one of the nodes publishes some data and requests a dummy service. In addition, you will see that **one of the robots starts moving along the scene towards a position**. You will have to change this node to implement your deployment strategy, and you will have to change other files, e.g., is associated launch file accordingly.

Note that the code provided for the other nodes is ***deliberately quite simple.*** The plotter is quite naive as well, and it may miss messages, not plotting some of the positions as a result. The motion controllers do not include obstacle avoidance, they do not rely on sensors, or maps, and they can be even implemented in a better way. The aim is that you get a basic set of files that you can then adapt or substitute, depending on your needs. You will also find examples on how to use topics, services, or start nodes passing arguments.