Robot Programming with ROS
Custom Messages and Services / Launch Files

Carmine Tommaso Recchiuto

Exercise

- ✓ Kill the turtle named **turtle1**
- ✓ Spawn a turtle named **rpr_turtle** in the position x = 2.0, y=1.0, theta=0.0
- ✓ Let the turtle move along x, until it reaches the end (x > 9.0)
- ✓ When x > 9.0 or x < 2.0, make it turn in a circular arc
- ✓ Continue until the turtle covers the whole area.
- ✓ Modify the CMakeLists.txt file (if needed) so as to build the program

Please find the source code updated here: https://github.com/CarmineD8/ros_ex1

You can also find the turtlebot controller package here: https://github.com/CarmineD8/turtlebot_controller

Example: client in *python*

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

import rospy

from turtlesim.msg import Pose
from turtlesim.srv import Spawn
from geometry_msgs.msg import Twist

pub = rospy.Publisher('/turtle1/cmd_vel', Twist,
queue_size=1000)

def positionCallback(msg):
    rospy.loginfo("Robot position [%f, %f]", msg.x, msg.y)
    vel = Twist()
    vel.linear.x = 0.1
    vel.angular.z = 0.1
    pub.publish(vel)

def control():
    rospy.init_node('turtlebot_controller_py')
    rospy.Subscriber("/turtle1/pose", Pose, positionCallback)
    client = rospy.ServiceProxy('/spawn', Spawn)
    client(x=1.0, y=1.0)
    rospy.spin()

if __name__ == '__main__':
    try:
        control()
    except rospy.ROSInterruptException:
        pass
```

Defining Custom Messages

- **Msg**: msg files are simple text files that describe the structure of a ROS message.
- Msg files are stored in the msg directory of a package.

- A message can be composed of
  - Int8, int16, int32, int64 (or uint)
  - Float 32, float 64
  - String
  - Time, Duration,..
  - Other msg files (i.e. geometry_msgs package)
  - Variable length array[] and fixed-length array[C]

Defining Custom Messages

- Create a folder named ***msg*** in your package
- Create a **.msg** file with the definition of the message structure:

e.g. string first_name

string last_name

uint8 age

uint32 score

Defining Custom Messages

For this example, let's keep working with the package turtlebot_controller

Inside the folder msg, we define a new .msg file (Vel.msg), with the following structure:

string name
float32 vel

Defining Custom Messages

- Modify the CMakeLists.txt in your package:

– Add ***message_generation*** to the list of components

i.e.: find_package (catkin REQUIRED COMPONENTS
roscpp
std_msgs
message_generation*)*

Also, you need to modify the ***package.xml*** *file*

Defining Custom Messages

- Uncomment and modify the lines (in the CMakeLists.txt)

```
#add_message_files (
# FILES
#Message1.msg
#Message2.msg
)
```

and

```
#generate_messages  (
#DEPENDENCIES
#std_msgs
#)
```

Defining Custom Messages

If you are going to use custom messages in another package:

- Add the header (es. #include *“turtlebot_controller/Vel.h”)*
- Add the dependency (CMakeLists.txt and package.xml)
- In the CmakeLists.txt, add:

```
add_dependencies(<node_name>
${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
```

In the example (turtlebot_controller), we try do define a second publisher using the custom message just defined, which publishes the string «linear» and the actual linear velocity.

Defining Custom Services

- As well as messages, service files are simple text files

- Services are composed by two parts:
  - **Request**
  - **Response**

- Example:

```
string first_name
string last_name
---
uint32 last_score
```

Defining Custom Services

- Services files should be defined in the **srv** directories of the package
- Cmakelists.txt and package.xml should be eventually modified by adding the message_generation dependency
- Let's create a new package, my_srv, with the dependencies **message_generation**, roscpp and *std_msgs*
- Here, we define a custom *service message (Velocity.srv)* with the following structure:

```
float32  min
float32 max
---
float32 x
float32 z
```

Defining Custom Services

- Uncomment and modify the lines (in the CMakeLists.txt)

  #add_service_files (
  # FILES
  #Service1.srv
  #Service2.srv
)

  and

  #generate_messages (
  #DEPENDENCIES
  #std_msgs
  #)

Writing a Service Node

- Definition of the service: name of the service and callback
  - *Ros::ServiceServer service=n.advertiseService("/position", random);*

- Example: service node that sends two random floats.

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, "position_server");
  ros::NodeHandle n;
  ros::ServiceServer service= n.advertiseService("/velocity",
myrandom);
  ros::spin();

  return 0;
}
```

Writing a Service Node

The service callback will be something like this:

```
double randMToN(double M, double N)
{     return M + (rand() / ( RAND_MAX / (N-M) ) ) ; }

bool myrandom (my_srv::Velocity::Request &req, my_srv::Velocity::Response &res){
      res.x = randMToN(req.min, req.max);
      res.z = randMToN(req.min, req.max);
      return true;
}
```

Writing a Service Node

In python, the stucture of a service node is something like:

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

from __future__ import print_function

from beginner_tutorials.srv import AddTwoInts,AddTwoIntsResponse
import rospy

def handle_add_two_ints(req):
    print("Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b)))
    return AddTwoIntsResponse(req.a + req.b)
```

```
def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts,
handle_add_two_ints)
    print("Ready to add two ints.")
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

General Remarks

Using custom messages and services, remember to:

- add the necessary headers
- add the dependency on the package where the custom messages and services are defined in the CMakeLists.txt and package.xml
- (if you are coding in cpp) In the CmakeLists.txt, add:

```
add_dependencies(<node_name>  ${${PROJECT_NAME}_EXPORTED_TARGETS}
        ${catkin_EXPORTED_TARGETS})
```

Example: Turtlebot controller

```
#include "my_srv/Velocity.h"

ros::Publisher pub;
ros::Publisher pub2;
ros::ServiceClient client2;
int count = 10;
```

In the main function, you should add

```
client2 = n.serviceClient<my_srv::Velocity>("/velocity");
```

```
void positionCallback(const turtlesim::Pose::ConstPtr& msg)
{
  ROS_INFO("Robot position [%f, %f]", msg->x, msg->y);
  my_srv::Velocity rec_vel;
  if (count==10){
		count=0;
		rec_vel.request.min=0.0;
		rec_vel.request.max=1.0;
		client2.call(rec_vel);
		geometry_msgs::Twist vel;
		vel.linear.x=rec_vel.response.x;
    vel.angular.z=rec_vel.response.z;
    pub.publish(vel);
    turtlebot_controller::Vel mymsg;
		mymsg.name = "linear";
    mymsg.vel = vel.linear.x;
    pub2.publish(mymsg);
  }
  count++;
}
```

Exercise

- Create a custom srv, which takes a float as request (x position), and replies with a float (x velocity)
- Create a Service which set a velocity depending from the position (from 2.0 to 9.0), with an armonic oscillator behaviour -> vel = 0.1 + 2*sin (pi*x/7 -2*pi/7)
- In the turtlebot controller, set the velocity computed by the service when x is between 2.0 and 9.0

ROS – Parameter Server

The ROS parameter server is a shared, multi-variate dictionary that is accessible via network APIs. Nodes use this server to store and retrieve parameters at runtime. As it is not designed for high-performance, it is best used for static, non-binary data such as configuration parameters.

It is meant to be globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary.

The Parameter Server uses standard data types for parameter values, which include:

- 32-bit integers
- booleans
- Strings
- doubles
- iso8601 dates (November 26, 2020)
- lists
- …

ROS – Parameter Server

The rosparam command-line tool enables you to query and set parameters on the Parameter Server using YAML* syntax.

rosparam set <parameter-name>	set parameter
rosparam get <parameter-name>	get parameter
rosparam load <yaml file>	load parameters from file
rosparam delete <parameter-name>	delete parameter
rosparam list	list all parameters name
rosparam list /namespace	list all parameters in a particular namespace

* YAML stands for Yet Another Markup Language, that supports all parameter types

ROS – Parameter Server

Example:

- rosrun turtlesim turtlesim_node
- rosparam get /background_b

Parameters may be retrieved (or modified) also within a node:

- in python, by using rospy.get_param or (rospy.set_param)
- in cpp, by using ros::param::get (or ros::param::set).

How to launch a node together with some parameters?

ROS – Launch Files

roslaunch is a tool for easily launching multiple ROS nodes locally, as well as setting parameters on the Parameter Server. It includes options to automatically respawn processes that have already died.

roslaunch takes in one or more XML configuration files (with the .launch extension) that specify the parameters to set and nodes to launch

The roslaunch package contains the roslaunch tools, which reads the roslaunch .launch/XML format. It also contains a variety of other support tools to help you use these files.

Many ROS packages come with "launch files", which you can run with:

```
$ roslaunch <package_name> <file.launch>
```

ROS – Launch Files

Some flags:

--wait

Delay the launch until a roscore is detected. By default, roslaunch will also launch the roscore master, the option “wait” may be added to force the launcher to wait for a master to be executed.

--local

Launch of the local nodes only. Nodes on remote machines will not be run.

--screen

Force all node output to screen. Useful for node debugging.

-v

Enable verbose printing. Useful for tracing roslaunch file parsing.

ROS – Launch Files

roslaunch .launch files are written in the XML format

roslaunch evaluates the XML file in a single pass. Includes are processed in depth-first traversal order. Tags are evaluated serially and the last setting wins. Thus, if there are multiple settings of a parameter, the last value specified for the parameter will be used.

Minimal example:

```
<launch>
        <node name="talker" pkg="rospy_tutorials" type="talker" />
</launch>
```

The **<node>** tag specifies a ROS node that you wish to have launched. This is the most common roslaunch tag as it supports the most important features: bringing up and taking down nodes.

ROS – Launch Files

Within the <node> tag, we may have additional (optional) attributs:

args= “arg1 arg2 arg3” -> arguments to be passed to the node.

respawn = “true” (default is false) -> restart the node automatically if it quits

required = “true” (default is false) -> if node dies, kill entire roslaunch

output = “screen” (default is log) -> if screen, stdout/stderr and ROS_INFO will be visualized on the terminal. If ‘log’, the stdout/stderr will be sent to a log file in $ROS_HOME/log.

cwd = “node” (default is ROS_HOME) -> if ‘node’, the working directory of the node will be set to the same directory as the node’s executable.

ROS – Launch Files

The <include> tag enables you to import another roslaunch XML file into the current file.

```
<launch>
        <include file=$(find pkg-name)/path/filename.xml />
        <node name="talker" pkg="rospy_tutorials" type="talker" />
</launch>
```

Roslaunch tag attributes can make use of *substitution args*, which roslaunch will resolve prior to launching nodes.

e.g. $(find pkg-name):

The filesystem path to the package directory will be substituted inline. Use of package-relative paths is highly encouraged as hard-coded paths inhibit the portability of the launch configuration.

ROS – Launch Files

Another example:

```
<launch>
	<arg name="a" default = 1 />
	<arg name="b" default = 2 />
	<arg name= "test" value = "true" />
	<node name="add_two_ints_client" pkg="beginner_tutorials"
	type="add_two_ints_client" args="$(arg a) $(arg b) $(arg test)" />
</launch>
```

The first two arguments may be passed as a command-line argument, while the third one (test) may not be overridden.

Es. roslaunch my_package my_launch_file a:=3 b:=7

ROS – Launch Files

<remap>

Remapping allows you to "trick" a ROS node so that when it thinks it is subscribing to or publishing to /some_topic it is actually subscribing to or publishing to /some_other_topic, for instance.

Sometimes, you may need a message on a specific ROS topic which normally only goes to one set of nodes to also be received by another node. If able, simply tell the new node to subscribe to this other topic. However, you may also do some remapping so that the new node ends up subscribing to /needed_topic when it thinks it is subscribing to /different_topic.

This could be accomplished like so:

<remap from="/different_topic" to="/needed_topic"/>

The remap tag can be used within a <node> tag, and in that case it will remap will apply just to that specific node, or generally in the launch file, and in this case it will apply to the lines following the remap.

ROS – Launch Files

How to set parameters in the launch file?

✓ Tags <param> and <rosparam>

The tag <param> defines a parameter to be set on the Parameter Server. The <param> tag can be put inside of a <node> tag, in which case the parameter is treated like a private parameter.

Es.

```
<param name="publish_frequency" type="double" value="10.0" />
```

Type may be str, int, double, bool or yaml.

ROS – Gazebo and Rviz

How to set parameters in the launch file?

✓ Tags <param> and <rosparam>

Similarly, the tag <rosparam> gives the possibility of loading or deleting parameters from the ROS Parameter Server.

```
<rosparam command="load" file="$(find rosparam)/example.yaml" />
<rosparam command="delete" param="my/param" />
```

Example

Example: package parameters: https://github.com/CarmineD8/parameters ->
roslaunch parameters param.launch

```
int main(int argc, char **argv)
{
ros::init(argc, argv, "getting_params");

int int_var;
double double_var;
std::string string_var;

ros::param::get("/my_integer", int_var);
ros::param::get("/my_float", double_var);
ros::param::get("/my_string", string_var);

ROS_INFO("Int: %d, Float: %lf, String: %s",
int_var, double_var, string_var.c_str());
}
```

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

import rospy

int_var = rospy.get_param("/my_integer")
float_var = rospy.get_param("/my_float")
string_var = rospy.get_param("/my_string")
print("Int: "+str(int_var)+", Float: "+str(float_var)+"
String: "+str(string_var))
```

ROS – Robotic simulations

- ✓ We are ready to start our first simulation of a robot in a 3D simulation environment. Download the package robot_description: [CarmineD8/robot_description: Package for working with mobile robot simulations with ROS and Gazebo (github.com)]() and switch to the branch Noetic (if needed):

  - git checkout noetic

```
├── CMakeLists.txt
├── config
│   ├── sim.rviz
│   └── sim2.rviz
├── include
│   └── robot_description
├── launch
│   ├── sim.launch
│   ├── sim2.launch
│   └── sim_w1.launch
├── package.xml
├── src
├── urdf
│   ├── robot2.gazebo
│   ├── robot2.xacro
│   ├── robot2_laser.gazebo
│   └── robot2_laser.xacro
└── worlds
    ├── world01.world
    └── world02.world
```

Let's take the package robot_description. As first step, let's see the structure of the package.

config -> configuration files for simulation
launch -> roslaunch files
urdf -> robot description files
worlds -> environments for simulation
scripts -> ros nodes, we will see it later

ROS – Rviz and Gazebo

- ✓ roslaunch robot_description sim.launch

We have now two windows, Rviz and Gazebo

Rviz is a tool for ROS Visualization. It's a 3-dimensional visualization tool for ROS. It allows the user to view the simulated robot model, log sensor information from the robot's sensors, and replay the logged sensor information. By visualizing what the robot is seeing, thinking, and doing, the user can debug a robot application from sensor inputs to planned (or unplanned) actions.

Gazebo is the 3D simulator for ROS

The robot may be controlled using ROS topics (/cmd_vel) (a nice tool is teleop_twist_keyboard, which may be launched with rosrun teleop_twist_keyboard teleop_twist_keyboard.py). When moving the robot around, information coming from sensors may be visualized in Rviz (ex: odom, or cameras).

ROS – Rviz and Gazebo

Let’s check more carefully the launch file.

- ✓ We add the robot description in the ROS parameter server
- ✓ We launch the simulation in an empty world
- ✓ We launch the node RVIZ, together with some additional nodes
- ✓ We spawn our robots in the simulation

ROS – Rviz and Gazebo

More details about steps 2 and 3!

Gazebo

- Dynamic simulation based on various physics engines (ODE, Bullet, Simbody and DART)
- Sensors (with noise) simulation
- Plugin to customize robots, sensors and the environment
- Realistic rendering of the environment and the robots
- Library of robot models
- ROS integration

Advanced features

- Remote & cloud simulation
- Open source

ROS – Rviz and Gazebo

Gazebo is composed by:

- A server gzerver for simulating the physics, rendering and sensors
- A client gzclient that provides a graphical interface to visualize and interact with the simulation

The client and the server communicate using the gazebo communication library

This may be seen by analyzing the launch file included (empty_world.launch in the gazebo_ros package)

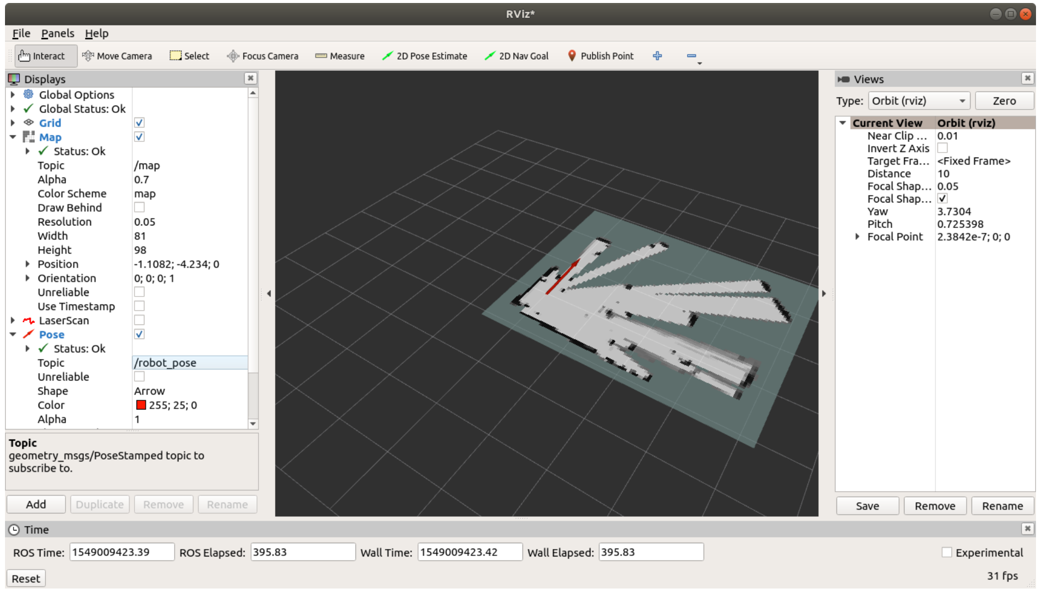
- Two different nodes are started, one for the GzServer, and one for the GzClient
- You may also notice all parameters defined in the launch file

ROS – Rviz

Rviz, abbreviation for ROS visualization, is a powerful 3D visualization tool for ROS. It allows the user to view the simulated robot model, log sensor information from the robot's sensors, and replay the logged sensor information. By visualizing what the robot is seeing, thinking, and doing, the user can debug a robot application from sensor inputs to planned (or unplanned) actions.

Rviz displays 3D sensor data from stereo cameras, lasers, Kinects, and other 3D devices in the form of point clouds or depth images. 2D sensor data from webcams, RGB cameras, and 2D laser rangefinders can be viewed in rviz as image data.

If an actual robot is communicating with a workstation that is running rviz, rviz will display the robot's current configuration on the virtual robot model. ROS topics will be displayed as live representations based on the sensor data published by any cameras, infrared sensors, and laser scanners that are part of the robot's system. This can be useful to develop and debug.



ROS – Rviz and Gazebo

When launching Rviz, three nodes are actually executed:

- joint_state_publisher
- robot_state_publisher
- rviz

• *joint_state_publisher*: the package reads the robot_description parameter from the parameter server, finds all of the non-fixed joints and publishes a JointState message with all those joints defined. If GUI is present, the package displays the joint positions in a window as sliders.

• *robot_state_publisher*: the package uses the URDF specified by the parameter robot_description and the joint positions from the topic joint_states to calculate the forward kinematics of the robot and publish the results via *tf*.

ROS – Rviz and Gazebo

- ✓ Rviz is executed by specifying a configuration file, which sets the elements that we want to display in the simulation.
- ✓ In the example, we specify the fixed frame (odom) and that we want to visualize the robot structure and the output of the camera.
- ✓ Topics or visualization elements may be added by selecting them from the add menu.
- ✓ By selecting “odom” as fixed frame, we may visualize the movement of the robot also in Rviz. This may be more evident, by adding the visualization of the tf

ROS – Rviz and Gazebo

- ✓ The sim2.launch roslaunch file corresponds to the same simulation, but with a slightly different robot: it has a laser sensor instead of a camera.

- ✓ The launch file is thus similar to the previous one, but we are now loading a different urdf file as robot_description parameter in the ROS parameter server, and we are starting Rviz with a different configuration file: indeed, we are going to visualize the laser sensor instead of the camera output.

- ✓ Please notice that, differently from images, the laser output may be seen directly in the corresponding frame

ROS – Rviz and Gazebo

- ✓ Finally sim_w1.launch uses a different environment for the simulation (environments have been stored in the folder worlds).

- ✓ Here in the launch file we explicitly launch the gazebo client and the server (we cannot include anymore the empty_world.launch).

- ✓ The world has been defined with a default value, so this may be overridden when launching the simulation (es. roslaunch robot_description sim_w1.launch world:=world01)

ROS – Exercise

- ✓ Launch sim_w1.launch
- ✓ Check what are the available topics for controlling the robot.
- ✓ Try to adapt the turtlebot_controller that you have developed for this mobile robot and Gazebo