F1/10 Autonomous Racing

ROS Services, Launch, Bags, and Parameters

Lab Session 3

Instructor: Madhur Behl madhur.behl@virginia.edu

Course Website: https://www.f1tenth.racing/

Git repo: https://github.com/linklab-uva/f1tenth-course-labs

Lab objectives:

In this lab, we will understand how to create and use `.launch` files; how to create and play `rosbags`, and how to invoke turtlesim services.

- Section [A]: ROS Services rospy API
- Section [B]: Using `roslaunch`
- Section [C]: ROS Bags: Recording and playing back data
- Section [D]: ROS Parameters

Update your git repo first

The following instructions assume

- You created a `catkin_ws` folder on your machine, using the instructions discussed during the previous lab sessions.
- You created a `github` folder in your home directory and cloned the `f1tenth-course-labs` repository.

Step 1)

Pull the latest code from the `f1tenth-course-labs` git repo:

```
cd ~/github
git pull
```

Ensure that any additional files after the git pull are copied into the appropirate directory under the smae package in the `catkin_ws`

[A] Getting familair with rospy service and client

Run and examine the following nodes:

In terminal 1:

```
madhur@ubuntu:~$ rosrun beginner_tutorials add_two_ints_server.py
```

In terminal 2:

```
madhur@ubuntu:~$ rosrun beginner_tutorials add_two_ints_client.py 1234 5678
Requesting 1234+5678
1234 + 5678 = 6912
```

- Go over the package manifest to see what is enabled for service and message generation
- Go over CMakeList.txt to see the paths to msg and srv files.

add_two_ints_server.py

```python
#!/usr/bin/env python
import roslib; roslib.load_manifest('beginner_tutorials')

from beginner_tutorials.srv import *
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()
```

add_two_ints_client.py

```python
#!/usr/bin/env python
import roslib; roslib.load_manifest('beginner_tutorials')

import sys

import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
```

```
        y = int(sys.argv[2])
    else:
        print usage()
        sys.exit(1)
    print "Requesting %s+%s"%(x, y)
    print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

[B] Using `roslaunch`

roslaunch starts nodes as defined in a launch file.

```
roslaunch [package] [filename.launch]
```

[B.1] >> `chat.launch`

Launch and examine the `chat.launch` file.

You can simply open up a terminal and type:

```
roslaunch beginner_tutorials chat.launch
```

You should see two talkers and one listener node spawn where the talkers are appropriate named as `talker1` and `talker2` , and the listener is named `listener1` . You should verify this using `rosnode list`

>> NOTE: Everything is runnig in the same terminal session as the one where you ran the `roslaunch` command. i.e. all three nodes are running int the same session.

>> You can kill `roscore` and relaunch `chat.launch` again and everythign will still work.

Even if roscore is not running, ROS will start roscore for you when you launch a launch file.

The launch file for `chat.launch` contians the following commands:

```
<launch>
  <node name="talker1" pkg="beginner_tutorials" type="talker.py" output="screen"/>
  <node name="listener1" pkg="beginner_tutorials" type="listener.py" output="screen"/>
  <node name="talker2" pkg="beginner_tutorials" type="talker.py" output="screen"/>
</launch>
```

we are simply runing 3 nodes in sequence (using the `<node>` tag) and using the `name` keyword to name the nodes.

[B.2] >> `turtlemimic.launch`

Lets examine another launch file: `turtlemimic.launch`

```
<launch>

  <group ns="turtlesim1">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
    <node pkg="turtlesim" name="teleop" type="turtle_teleop_key" launch-prefix="gnome-terminal -e"/>
  </group>

  <group ns="turtlesim2">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <node pkg="turtlesim" name="mimic" type="mimic">
    <remap from="input" to="turtlesim1/turtle1"/>
    <remap from="output" to="turtlesim2/turtle1"/>
  </node>

</launch>
```

Two turtlesims will start and in a new terminal the teleop interface will show up. You you issue commands for one of the turtle to move, you will find that the second turtle will mimic those commands.

Now lets move the turtle in the `turtlesim1` instance using `rostopic pub`

```
rostopic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'
```

You will see the two turtlesims start moving even though the publish command is only being sent to turtlesim1.

Notice how you didnt need to start `roscore` but it was started when you launched the launch file.

>> NOTE:

When we want a node to launch in a separate terminal window from the launch file we can use the `launch-prefix="gnome-terminal -e"` option in the `<node>` tag of the file.

[C] ROS Bags: Recording and playing back data

Description: This part of the tutorial will teach you how to record data from a running ROS system into a `.bag` file, and then to play back the data to produce similar behavior in a running system.

[C.1] Recording data (creating a bag file)

Lets pull up the `turtlesim` ocean and the teleop node. First, execute the following commands in separate terminals:

Terminal 1:

```
roscore
```

Terminal 2:

```
rosrun turtlesim turtlesim_node
```

Terminal 1:

```
rosrun turtlesim turtle_teleop_key
```

OR use `roslaunch` to launch both the turtlesim nodes at the same time.

```
roslaunch beginner_tutorials turtlemimic.launch
```

This will start two separate turtlesim nodes and a `mimic` node that allows for the keyboard control of both turtlesim's by mapping the output of `turtlesim1` to the input of `turtlesim2`.

[C.2] Recording all published topics

Examine the full list of topics that are currently being published in the running system. To do this, open a new terminal and execute the command:

```
rostopic list -v
```

This should yield the following output:
We now will record the published data. Open a new terminal window. In this window run the following commands:

```
mkdir ~/bagfiles
cd ~/bagfiles
rosbag record -a
```

Here we are just making a temporary directory to record data and then running rosbag record with the option -a, indicating that all published topics should be accumulated in a bag file.

Move back to the terminal window with turtle_teleop and move the turtle around for 10 or so seconds.

In the window running rosbag record exit with a Ctrl-C. Now examine the contents of the directory ~/bagfiles. You should see a file with a name that begins with the year, date, and time and the suffix .bag. This is the bag file that contains all topics published by any node in the time that rosbag record was running.

```
madhur@ubuntu:~/bagfiles$ ls
2019-02-12-12-18-13.bag
```

[C.3] Examining and playing the bag file

Now that we've recorded a bag file using rosbag record we can examine it and play it back using the commands rosbag info and rosbag play. First we are going to see what's recorded in the bag file. We can do the info command -- this command checks the contents of the bag file without playing it back. Execute the following command from the bagfiles directory:

```
rosbag info <your bagfile>
```

You should see something like:

```
madhur@ubuntu:~/bagfiles$ rosbag info 2019-02-12-12-18-13.bag
path:        2019-02-12-12-18-13.bag
version:     2.0
duration:    16.1s
start:       Feb 12 2019 12:18:13.10 (1549991893.10)
end:         Feb 12 2019 12:18:29.22 (1549991909.22)
size:        150.4 KB
messages:    2007
compression: none [1/1 chunks]
types:       geometry_msgs/Twist [9f195f881246fdfa2798d1d3eebca84a]
             rosgraph_msgs/Log   [acffd30cd6b6de30f120938c17c593fb]
             turtlesim/Color     [353891e354491c51aabe32df673fb446]
             turtlesim/Pose      [863b248d5016ca62ea2e895ae5265cf9]
topics:      /rosout                              4 msgs    : rosgraph_msgs/Log   (2 connections)
             /turtlesim1/turtle1/cmd_vel         19 msgs    : geometry_msgs/Twist
             /turtlesim1/turtle1/color_sensor   992 msgs    : turtlesim/Color              /turtlesim1/turtle1/pc
```

The next step in this tutorial is to replay the bag file to reproduce behavior in the running system. First kill the teleop program that may be still running from the previous section - a Ctrl-C in the terminal where you started turtle_teleop_key. Leave turtlesim running. In a terminal window run the following command in the directory where you took the original bag file:

```
rosbag play <your bagfile>
```

In this window you should immediately see something like:

```
madhur@ubuntu:~/bagfiles$ rosbag play 2019-02-12-12-18-13.bag
[ INFO] [1549996933.935808161]: Opening 2019-02-12-12-18-13.bag

Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to step.
[DELAYED]  Bag Time: 1549991893.101184   Duration: 0.000000 / 16.123553   Delay
[RUNNING]  Bag Time: 1549991893.101184   Duration: 0.000000 / 16.123553
[RUNNING]  Bag Time: 1549991893.101184   Duration: 0.000000 / 16.123553
[RUNNING]  Bag Time: 1549991893.102541   Duration: 0.001357 / 16.123553
```

In its default mode rosbag play will wait for a certain period (.2 seconds) after advertising each message before it actually begins publishing the contents of the bag file. Waiting for some duration allows any subscriber of a message to be alerted that the message has been advertised and that messages may follow. If rosbag play publishes messages immediately upon advertising, subscribers may not receive the first several published messages. The waiting period can be specified with the -d option.

Eventually the topic ~/turtle1/cmd_vel~ will be published and the turtle should start moving in turtlesim in a pattern similar to the one you executed from the teleop program. The duration between running rosbag play and the turtle moving should be approximately equal to the time between the original rosbag record execution and issuing the commands from the keyboard in the beginning part of the tutorial. You can have rosbag play not start at the beginning of the bag file but instead start some duration past the beginning using the -s argument. A final option that may be of interest is the -r option, which allows you to change the rate of publishing by a specified factor. If you execute:

```
rosbag play -r 2 <your bagfile>
```

You should see the turtle execute a slightly different trajectory - this is the trajectory that would have resulted had you issued your keyboard commands twice as fast.

[C.4] Recording a subset of the data

If any turtlesim nodes are running exit them and relaunch the keyboard teleop launch file:

```
rosrun turtlesim turtlesim_node
rosrun turtlesim turtle_teleop_key
```

In your bagfiles directory, run the following command:

```
rosbag record -O subset /turtle1/cmd_vel /turtle1/pose
```

The -O argument tells rosbag record to log to a file named subset.bag, and the topic arguments cause rosbag record to only subscribe to these two topics. Move the turtle around for several seconds using the keyboard arrow commands, and then Ctrl-C the rosbag record.

You may have noted that the turtle's path may not have exactly mapped to the original keyboard input - the rough shape should have been the same, but the turtle may not have exactly tracked the same path. The reason for this is that the path tracked by turtlesim is very sensitive to small changes in timing in the system, and rosbag is limited in its ability to exactly duplicate the behavior of a running system in terms of when messages are recorded and processed by rosrecord, and when messages are produced and processed when using rosplay. For nodes like turtlesim, where minor timing changes in when command messages are processed can subtly alter behavior, the user should not expect perfectly mimicked behavior.

[D] ROS Parameters

A 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.

[D.1] Parameter Types

The Parameter Server uses the following data types for parameter values:

-32-bit integers
-booleans
-strings
-doubles
-iso8601 dates
-lists
-base64-encoded binary data

You can also store dictionaries (i.e. structs) on the Parameter Server, though they have special meaning. The Parameter Server represents ROS namespaces as dictionaries. For example, imagine you `set` the following three parameters:

~~~#!/usr/bin/env bash
/gains/P = 10.0
/gains/I = 1.0
/gains/D = 0.1

```
You can either `get` them back separately, i.e. retrieving `/gains/P` would return `10.0`, or you can retrieve `/
```

#!/usr/bin/env bash
{ 'P': 10.0, 'I': 1.0, 'D' : 0.1 }

```
### [D.2] Run the parameter server demo

Open a shell instance and laucnh the following:
```

#!/usr/bin/env bash
roslaunch rospy_tutorials param_talker.launch

```
`rospy_tutorials` is installed by default when you downloaded ROS but in case this package is missing for you you
```

#!/usr/bin/env bash
sudo apt-get install ros-melodic-ros-tutorials

```
To understadn the ourput of the `param_talker.launch`, let us quickly examine this file:
```

xml

```
<!-- set /foo/utterance -->
<param name="utterance" value="Hello World" />

<param name="to_delete" value="Delete Me" />

<!-- a group of parameters that we will fetch together -->
<group ns="gains">
  <param name="P" value="1.0" />
  <param name="I" value="2.0" />
  <param name="D" value="3.0" />
</group>

<node pkg="rospy_tutorials" name="param_talker" type="param_talker.py" output="screen">

  <!-- set /foo/utterance/param_talker/topic_name -->
  <param name="topic_name" value="chatter" />

</node>
```
~~~

```
As can be seen, this launch files `sets` a parameter `global_example` followed by declaring a namespace called `f
`gains` itself is a subgroup of `foo` and has three parameters `P`, `I`, and 'D'.

Finally a `param_talker.py` node is launched as part of the `rospy_tutorials` package.

If we go through the `param_talker.py` script, you will find the following code:

** You can use `roscd rospy_tutorials/006_parameters` to go to the directory where the script is located.
```

python
import rospy
from std_msgs.msg import String

def param_talker():
rospy.init_node('param_talker')

```
# Fetch values from the Parameter Server. In this example, we fetch
# parameters from three different namespaces:
#
# 1) global (/global_example)
# 2) parent (/foo/utterance)
# 3) private (/foo/param_talker/topic_name)

# fetch a /global parameter
global_example = rospy.get_param("/global_example")
rospy.loginfo("%s is %s", rospy.resolve_name('/global_example'), global_example)

# fetch the utterance parameter from our parent namespace
utterance = rospy.get_param('utterance')
rospy.loginfo("%s is %s", rospy.resolve_name('utterance'), utterance)

# fetch topic_name from the ~private namespace
topic_name = rospy.get_param('~topic_name')
rospy.loginfo("%s is %s", rospy.resolve_name('~topic_name'), topic_name)

# fetch a parameter, using 'default_value' if it doesn't exist
default_param = rospy.get_param('default_param', 'default_value')
rospy.loginfo('%s is %s', rospy.resolve_name('default_param'), default_param)

# fetch a group (dictionary) of parameters
gains = rospy.get_param('gains')
p, i, d = gains['P'], gains['I'], gains['D']
rospy.loginfo("gains are %s, %s, %s", p, i, d)

# set some parameters
rospy.loginfo('setting parameters...')
rospy.set_param('list_of_floats', [1., 2., 3., 4.])
rospy.set_param('bool_True', True)
rospy.set_param('~private_bar', 1+2)
rospy.set_param('to_delete', 'baz')
rospy.loginfo('...parameters have been set')

# delete a parameter
if rospy.has_param('to_delete'):
    rospy.delete_param('to_delete')
    rospy.loginfo("deleted %s parameter"%rospy.resolve_name('to_delete'))
else:
    rospy.loginfo('parameter %s was already deleted'%rospy.resolve_name('to_delete'))

# search for a parameter
param_name = rospy.search_param('global_example')
rospy.loginfo('found global_example parameter under key: %s'%param_name)
```

```
# publish the value of utterance repeatedly
pub = rospy.Publisher(topic_name, String, queue_size=10)
while not rospy.is_shutdown():
    pub.publish(utterance)
    rospy.loginfo(utterance)
    rospy.sleep(1)
```

if **name** == '**main**':
try:
param_talker()
except rospy.ROSInterruptException: pass

```
### [D.3] rospy: Getting parameters

The command is:
```

rospy.get_param(param_name)

```
In this node, this is used at several points:
```

python
global_example = rospy.get_param("/global_example")
utterance = rospy.get_param('utterance')
private_param = rospy.get_param('~private_name')
default_param = rospy.get_param('default_param', 'default_value')

fetch a group (dictionary) of parameters

gains = rospy.get_param('gains')
p, i, d = gains['p'], gains['i'], gains['d']

```
You can optionally pass in a default value to use if the parameter is not set.
Names are resolved relative to the node's namespace.

While the node is still running, you can inspect the parameters from the command line using `rosparam`

Try
```

#!/usr/bin/env bash
rosparam get /foo/gains

```
and try,
```

#!/usr/bin/env bash
rosparam get /foo/gains/P

```
### [D.4] rospy: Setting parameters

As described earlier, you can set parameters to store strings, integers, floats, booleans, lists, and dictionarie
```

rospy.set_param(param_name, param_value)

```
Examples:
```

python

Using rospy and raw python objects

rospy.set_param('a_string', 'baz')
rospy.set_param('~private_int', 2)
rospy.set_param('list_of_floats', [1., 2., 3., 4.])
rospy.set_param('bool_True', True)
rospy.set_param('gains', {'p': 1, 'i': 2, 'd': 3})

```
### [D.5] Parameter Server of turtlesim

The Parameter Server maintains a dictionary of the parameters that are used to configure the screen of turtlesim.

Use the `help` option to determine the form of the `rosparam` command:
```

#!/usr/bin/env bash
rosparam -- help

```
Output of the preceding command will contain the following:
```

#!/usr/bin/env bash
rosparam is a command-line tool for getting, setting, and deleting parameters from the ROS Parameter Server. Commands:
rosparam set set parameter
rosparam get get parameter
rosparam list list parameter names

```
To list the parameters for the ``/turtlesim` node, we will use the following command:
```

#!/usr/bin/env bash
$ rosparam list

```
Output of the preceding code is as follows:
```

#!/usr/bin/env bash
/background_r
/background_g
/background_b
/rosdistro
/roslaunch/uris/host_d125_43873__51759
/rosversion
/run_id

```
Note that the last four parameters were created by invoking the Master with the roscore command, as discussed pre

#### Change parameters for the color of the turtle's background

To change the color parameters for turtlesim, let's change the turtle's background to red. To do this, make the b
Note that the `clear` option from rosservice must be executed before the screen changes color.

The default turtle screen is blue. You can use `rosparam get /` to show the data contents of the entire Parameter
```

#!/usr/bin/env bash
$ rosparam get /

```
Output:
```

#!/usr/bin/env bash
background_b: 255
background_g: 86
background_r: 69
rosdistro: 'indigo
roslaunch:
uris: {host_d125_43873__60512: 'http://D125-43873:60512/'}
rosversion: '1.11.13
run_id: 2429b792-d23c-11e4-b9ee-3417ebbca982

```
You can change the colors of the turtle's screen to a full red background using the `rosparam set` command:
```

#!/usr/bin/env bash
$ rosparam set background_b 0
$ rosparam set background_g 0
$ rosparam set background_r 255
$ rosservice call /clear
~~~

You will see a red background on the turtle screen. To check the numerical results, use the `rosparam get /` command.
~~~