F1/10 Autonomous Racing

ROS - Filesystem and Workspaces

Lab Session CS4501

Madhur Behl madhur.behl@virginia.edu

Course Website: https://linklab-uva.github.io/autonomousracing/

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

Lab objective:

In this lab, a group of concepts are used to explain how ROS is internally formed, the folder structure, and the minimum number of files that it needs to work.

- Section [A]: ROS Namespace
- Section [B]: Publishing and Subscribing in the same node
- Section [C]: ROS Services
- Section [D]: ROS Launch Files

[A] Namespace and remapping:

Any ROS name within a node can be remapped when it is launched at the command-line. This is a powerful feature of ROS that lets you launch the same node under multiple configurations from the command-line. All resource names can be remapped. You can also provide assignment for private node parameters. This feature of ROS allows you to defer complex name assignments to the actual runtime loading of the system..

1. Now start `roscore` and try to run two talker nodes by running the following command twice in two separate sourced terminal instances

```
rosrun beginner_tutorials talker.py
rosrun beginner_tutorials talker.py
```

Notice how one of the talker nodes shuts down as soon as you run the second node. This is due to the fact that they have the same node names. It also demonstrates another way a node can receive a shutdown message

1. One way to fix this problem is to go back to the `talker.py` code and reset the argument `anonymous=False` on line 8, back to `anonymous=True`. Rebuild. Retry two nodes. Upon trying this you will find, that ROS will append the name of the `talker` node with unique numerical identifiers. You can verify this by running `rosnode list` or by launching the `rqt_graph`.

2. However, we need not reset and recompile our code to create multiple instances of the same node. We can do so using `ROS namespaces`. You will learn more about the namespaces in subsequent lectures, but try running the following commands, while roscore is running.

```
rosrun beginner_tutorials talker.py __name:=talker1
rosrun beginner_tutorials talker.py __name:=talker2
rqt_graph
```

And just like that, we have two instances of `talker.py` node with different names, both publishing hello messages on the topic `chatter`.

Any ROS name within a node can be remapped when it is launched at the command-line. This is a powerful feature of ROS that lets you launch the same node under multiple configurations from the command-line. All resource names can be remapped. You can also provide assignment for private node parameters. This feature of ROS allows you to defer complex name assignments to the actual runtime loading of the system.

```
__name

 __name is a special reserved keyword for "the name of the node."
 It lets you remap the node name without having to know its actual name.
 It can only be used if the program that is being launched contains one node.

__log

 __log is a reserved keyword that designates the location that the node's log
file should be written.
Use of this keyword is generally not encouraged --
it is mainly provided for use by ROS tools like roslaunch.

__ip and __hostname

__ip and __hostname are substitutes for ROS_IP and ROS_HOSTNAME.
Use of this keyword is generally not encouraged as it is provided for
special cases where environment variables cannot be set.

__master

__master is a substitute for ROS_MASTER_URI.
Use of this keyword is generally not encouraged as it is provided for
special cases where environment variables cannot be set.

__ns

 __ns is a substitute for ROS_NAMESPACE.
 Use of this keyword is generally not encouraged as it is provided
 for special cases where environment variables cannot be set.
```

[B] Pub/Sub in the same node

Run and examine the following nodes:

In terminal 1:

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

In terminal 2:

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

In terminal 3:

```
madhur@ubuntu:~$ rostopic list
/rand_no
/rosout
/rosout_agg
/sub_pub
```

Echo the messages on the topic `/sub_pub`

```
madhur@ubuntu:~$ rostopic echo /sub_pub
```

Let us checkout the code for the `pub_n_sub.py` node to convice youself that the messages being echoed on the topic `/sub_pub` are indeed correct:

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

import rospy
from std_msgs.msg import Int32

varS=None

def fnc_callback(msg):
    global varS
    varS=msg.data

if __name__=='__main__':
    rospy.init_node('pub_n_sub')

    sub=rospy.Subscriber('rand_no', Int32, fnc_callback)
    pub=rospy.Publisher('sub_pub', Int32, queue_size=1)
    rate=rospy.Rate(5)

    while not rospy.is_shutdown():
        if varS<= 2500:
            varP=0
        else:
             varP=1

        pub.publish(varP)
        rate.sleep()
```

[C] 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

```
#!/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

```
#!/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))
```

[D] ROS Launch

Hopefully by this point you've realized that starting every node individually is not always the best option. Doing so not only takes time, but it also results in a plethora of command terminals. 'roslaunch' is a tool in ROS that allows you to start multiple nodes (which are defined in a launch file) with only one command.

The `beginner_tutorials` package downloaded from the course's git repository includes a subdirectory called `launch` with some examples of `.launch` files.

[D.1] Using roslaunch

`roslaunch` starts nodes as defined in a launch file.

Usage:

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

NOTE: The directory to store launch files don't necessarily have to be named as launch. In fact you don't even need to store them in a directory. roslaunch command automatically looks into the passed package and detects available launch files. However, it is good practice to do so.

[D.2] The Launch File

Let's examine the launch file called `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>
```

Now, let's break the launch xml down.
We start the launch file with the launch tag, so that the file is identified as a launch file.
We start two groups with a namespace tag of turtlesim1 and turtlesim2 with a turtlesim node with a name of sim. This allows us to start two simulators without having name conflicts.
We start the mimic node with the topics input and output renamed to turtlesim1 and turtlesim2. This renaming will cause turtlesim2 to mimic turtlesim1.

[D.3] Additional Renaming Methods

In addition to using `group ns` as described above, there are two other methods for renaming nodes

```
<remap from="/sim" to="george"/>
<node pkg="turtlesim" name="sim" type="turtlesim_node"/>
```

```
<node pkg="turtlesim" name="sim" type="turtlesim_node" ns="george"/>
```

[D.4] roslaunching

Now let's `roslaunch` the launch file:

```
roslaunch beginner_tutorials turtlemimic.launch
```

Two turtlesims will start and in a "'new terminal"' send the `rostopic` command:

```
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`.

[D.5] Examine `chat.launch` and see how it can be used to launch multiple `talker` and `listener` nodes.