# F1/10 Autonomous Racing

## ROS - Filesystem and Workspaces

### Lab Session 3 - CS4501/SYS4582 - Spring 2019

**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]: Python publisher and Subscriber.
- Section [B]: ROS Namespace
- Section [C]: Publishing and Subscribing in the same node
- Section [D]: ROS Services

---

# [A] Python Publisher and Subscriber.

**The following instructions assume that you created a `catkin_ws` folder on your machine, using the instructions discussed during the previous lab sessions**

---

**Step 1)**

Make sure to create a ROS workspace on your machine. Usually this can be done in the home directory. You would have already doen this in a previous lab session but here we recap all the steps. You can do this by tying the following coommands in the terminal, one at a time. If you already have a ROS workspace on your machine, you can skip directly to Step 2, and use your ROS workspace name instead of the name catkin_ws.

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
cd ..
catkin_make
source ~/catkin_ws/devel/setup.bash
```

**Step 2)**

Next create a new directory in your home folder to clone this git repository.

```
mkdir ~/github
cd ~/github
git clone https://github.com/linklab-uva/f1tenth-course-labs.git
```

This will create a `f1tenth-course-labs` folder inside the `github` directory you just created.

The git repository, will usually contain a folder with the same name as the lab handout issued. For example, the repository contains a folder 'beginner_tutorials', which we will use for illustrating the next step. But you may want to replace the folder name with the appropriate name as indicated by the lab handout.

**Step 3)**

Create a symbolic link between the folder of interest, e.g. `beginner_tutorials` and the src folder in your catkin workspace. Note that you may already have a `beginner_tutorials` package that you may have created in a previous lab. We will remove that folder and symbolically link our git repo with the workspace.

Depending on your ROS workspace. First remove the beginner_tutorials directory and its contents.

```
madhur@ubuntu:~$ cd ~/catkin_ws/src/
madhur@ubuntu:~/catkin_ws/src$ rm -rf beginner_tutorials/
```

In this example we assume that the root folder for the workspace source does not contain a `beginner_tutorials` directory. We will make sure all the python scripts are executable: Run:

```
ln -s ~/github/f1tenth-course-labs/beginner_tutorials ~/catkin_ws/src/
cd ~/catkin_ws
find . -name “*.py” -exec chmod +x {} \;
catkin_make
source ~/catkin/devel/setup.bash
```

Thats it !, you should now be able to run any ROS command that interacts with packages. **Step 4)** Finally, as you may have already guessed, we need to source our new package that was just added to the ROS workspace.

```
source ~/catkin_ws/devel/setup.bash
```

**Step 4)** Lets run these the two nodes `talker` and `listener`

Run:

```
roscore
rosrun beginner_tutorials talker.py
rosrun beginner_tutorials listener.py
```

Talker should begin outputting text similar to:

```
[INFO] [WallTime: 1394915011.927728] hello world 1394915011.93
[INFO] [WallTime: 1394915012.027887] hello world 1394915012.03
[INFO] [WallTime: 1394915012.127884] hello world 1394915012.13
[INFO] [WallTime: 1394915012.227858] hello world 1394915012.23
...
```

And listener should begin outputting text similar to:

```
[INFO] [WallTime: 1394915043.555022] /listener_9056_1394915043253I heard hello world 13
[INFO] [WallTime: 1394915043.654982] /listener_9056_1394915043253I heard hello world 13
[INFO] [WallTime: 1394915043.754936] /listener_9056_1394915043253I heard hello world 13
[INFO] [WallTime: 1394915043.854918] /listener_9056_1394915043253I heard hello world 13
...
```

Now run

```
rostopic echo chatter
```

`chatter` is the name of the topic on which the messages are being published, and subscribed. You should see an output similar to what is shown below:

```
data: hello world 1394915083.35
---
data: hello world 1394915083.45
---
data: hello world 1394915083.55
---
data: hello world 1394915083.65
---
data: hello world 1394915083.75
---
...
```

**Step 5)** For you reference, the talker.py and listener.py code is shown below:

*talker.py*

```
1 #!/usr/bin/env python
2 # license removed for brevity
3 import rospy
4 from std_msgs.msg import String
5
6 def talker():
7     pub = rospy.Publisher('chatter', String, queue_size=10)
8     rospy.init_node('talker', anonymous=True)
9     rate = rospy.Rate(10) # 10hz
10     while not rospy.is_shutdown():
11         hello_str = "hello world %s" % rospy.get_time()
12         rospy.loginfo(hello_str)
13         pub.publish(hello_str)
14         rate.sleep()
15
16 if __name__ == '__main__':
17     try:
18         talker()
19     except rospy.ROSInterruptException:
20         pass
```

*listener.py*

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():

    # In ROS, nodes are uniquely named. If two nodes with the same
    # node are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("chatter", String, callback)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

if __name__ == '__main__':catkin package as we just changed some code
    listener()
```

### Useful Tip: Fixing the problem at its source !

Since you will be mostly working with the `catkin_ws` ROS workspace, it can become quite cumbersome to repeatedly source the environment every time you start a new shell instance. To fix this, it is recommended to add the line `source ~/catkin_ws/devel/setup.bash` to your `.bashrc` file.

To do so, you can type:

```
sudo echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

# Exercise:

We know that ROS down not allow two nodes to have the same name, but let us verify this with the `talker.py` and `listener.py` nodes that we just installed and executed.

1. Open a new terminal instance.
2. type `roscd beginner_tutorials/scripts` and hit enter
3. Next, we will edit the `rospy.init_node` command in `talker.py` and change the argument `anonymous=True` on line 8 to `anonymous=False`.
4. You can use your favorite file editor to do this. Just be sure, to save your changes.
5. Lets rebuild the catkin package as we just changed some code.
6. You know the drill:

# [B] 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

__log

    __log is a reserved keyword that designates the location that the node's log file s

__ip and __hostname

    __ip and __hostname are substitutes for ROS_IP and ROS_HOSTNAME. Use of this keywor

__master

    __master is a substitute for ROS_MASTER_URI. Use of this keyword is generally not e

__ns

    __ns is a substitute for ROS_NAMESPACE. Use of this keyword is generally not encour
```

# [C] 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()
```

# [D] 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()
```