F1/10 Autonomous Racing

ROS - Filesystem/Workspaces/Publishing/Subscribing

Lab Session CS4501

Madhur Behl madhur.behl@virginia.edu

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

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

Lab objective:

In this lab, several concepts are used to explain how ROS is internally formed, the folder structure, and the minimum number of files that it
needs to work. We then see and example of ROS publishing and Subscribing.

- Section [A]: Managing a ROS Environment
- Section [B]: Creating a ROS Workspace
- Section [C]: ROS Filesystem Tools
- Section [D]: Creating a ROS Package
- Section [E]: rospy Publisher and Subscriber.
- Section [F]: ROS Namespace
- Section [G]: Publishing and Subscribing in the same node

[A] Managing a ROS Environment

If you are ever having problems finding or using your ROS packages make sure that you have your environment properly setup. A good way to check is to ensure that environment variables like `ROS_ROOT` and `ROS_PACKAGE_PATH` are set:

```
$ printenv | grep ROS
```

The output should look something like this (based on your ROS distribution)

```
madhur@ubuntu:~$ printenv | grep ROS
ROS_ROOT=/opt/ros/melodic/share/ros
ROS_PACKAGE_PATH=/opt/ros/melodic/share
ROS_MASTER_URI=http://localhost:11311
ROS_VERSION=1
ROSLISP_PACKAGE_DIRECTORIES=
ROS_DISTRO=melodic
ROS_ETC_DIR=/opt/ros/melodic/etc/ros
```

If they are not then you might need to 'source' some setup.*sh files.

If you just installed ROS from apt on Ubuntu then you will have setup.*sh files in '/opt/ros//', and you could source them like so:

```
$ source /opt/ros/<distro>/setup.bash
```

If you installed ROS melodic, that would be:

```
$ source /opt/ros/melodic/setup.bash
```

You will need to run this command on every new shell you open to have access to the ROS commands, unless you add this line to your .bashrc. [This process allows you to install several ROS distributions (e.g. indigo and melodic) on the same computer and switch between them.]

To do so:

```
echo "source /opt/ros/<distro>/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

You may recall doing so during the ROS installation. Now you know the reason - to let every shell instance know of the location where ROS is installed on your system and to set the `PATH` environment.

[B] Creating a ROS Workspace

In general terms, the workspace is a folder which contains packages, those packages contain our source files and the environment or workspace provides us with a way to compile those packages. It is useful when you want to compile various packages at the same time and it is a good way of centralizing all of our developments.

Lets create and build a catkin workspace - similar to what we discussed in the lectures.

You can create the workspace in any directory on your machine, **except inside an existing workspace!**. For this example we will just create it in the home directory.

Go ahead and type in these commands one at a time:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
$ cd ..
$ catkin_make
$ echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

The output will look something like:

```
madhur@ubuntu:~$ mkdir -p ~/catkin_ws/src
madhur@ubuntu:~$ cd ~/catkin_ws/src
madhur@ubuntu:~/catkin_ws/src$ catkin_init_workspace
```

```
Creating symlink "/home/madhur/catkin_ws/src/CMakeLists.txt" pointing to "/opt/ros/melodic/share/catkin/cmake/top
madhur@ubuntu:~/catkin_ws/src$
```

```
madhur@ubuntu:~/catkin_ws/src$ cd ..
madhur@ubuntu:~/catkin_ws$ catkin_make
Base path: /home/madhur/catkin_ws
Source space: /home/madhur/catkin_ws/src
Build space: /home/madhur/catkin_ws/build
Devel space: /home/madhur/catkin_ws/devel
Install space: /home/madhur/catkin_ws/install
####
#### Running command: "cmake /home/madhur/catkin_ws/src -DCATKIN_DEVEL_PREFIX=/home/madhur/catkin_ws/devel -DCMAK
####
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
.
.
.
-- Configuring done
-- Generating done
-- Build files have been written to: /home/madhur/catkin_ws/build
####
#### Running command: "make -j2 -l2" in "/home/madhur/catkin_ws/build"
####
```

The catkin_make command is a convenience tool for working with catkin workspaces. Running it the first time in your workspace, it will create a CMakeLists.txt link in your 'src' folder. Additionally, if you look in your current directory you should now have a 'build' and 'devel' folder. Inside the 'devel' folder you can see that there are now several setup.*sh files.

[C] ROS Filesystem Tools

Code is spread across many ROS packages.

Navigating with command-line tools such as ls and cd can be very tedious which is why ROS provides tools to help you.

For this tutorial we will inspect a package in ros-tutorials, please install it using:

```
sudo apt-get install ros-<distro>-ros-tutorials
```

Example output:

```
madhur@ubuntu:~$ sudo apt-get install ros-melodic-ros-tutorials
[sudo] password for madhur:
Reading package lists... Done
Building dependency tree Reading state information... Done
ros-melodic-ros-tutorials is already the newest version (0.7.1-0xenial-20181107-045510-0800).
ros-melodic-ros-tutorials set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 220 not upgraded.
```

[C.1] `rospack`

rospack allows you to get information about packages. In this tutorial, we are only going to cover the find option, which returns the path to package.

Type in:

```
$ rospack find [package_name]
```

Example:

```
$ rospack find roscpp
```

would return:

```
madhur@ubuntu:~$ rospack find roscpp
/opt/ros/melodic/share/roscpp
```

[Q1] Find and report where `turtlesim` is installed ?

[Q2] The package `turtlesim` depends on the `gennodejs` package. Is this statement true or false ?

Hint: Use `rospack help` to figure out which command to use to answer the true/false statement above.

[C.2] `roscd`

roscd allows you to change directory (cd) directly to a package or a stack. [Recall what is a stack ?]

Usage:

```
$ roscd [locationname[/subdir]]
```

Example:

```
$ roscd roscpp
```

Now let's print the working directory using the Unix command `pwd` :

```
$ pwd
```

You should see:

```
YOUR_INSTALL_PATH/share/roscpp
```

Example:

```
madhur@ubuntu:~$ roscd roscpp
madhur@ubuntu:/opt/ros/melodic/share/roscpp$ pwd
/opt/ros/melodic/share/roscpp
```

You can see that YOUR_INSTALL_PATH/share/roscpp is the same path that rospack find gave in the previous example.

roscd can also move to a subdirectory of a package or stack.

Wondering where does `turtlesim` store the images of all the tutrles it renders in the ocean ?

Type

```
$ roscd turtlesim/images/
```

`roscd log` will take you to the folder where ROS stores log files.

Note that if you have not run any ROS programs yet, this will yield an error saying that it does not yet exist.

[C.3] `rosls`

`rosls` allows you to `ls` directly in a package by name rather than by absolute path.

Example:

```
$ rosls roscpp_tutorials
```

Output:

```
madhur@ubuntu:~/.ros/log$ rosls roscpp_tutorials
cmake  launch  package.xml  srv
```

Note

You may have noticed a pattern with the naming of the ROS tools:

rospack = ros + pack(age)

roscd = ros + cd

rosls = ros + ls

This naming pattern holds for many of the ROS tools.

[D] Creating a ROS Package

Similar to an operating system, an ROS program is divided into folders, and these folders have files that describe their functionalities.

- **Packages:** Packages form the atomic level of ROS. A package has the minimum structure and content to create a program within ROS. It may have ROS runtime processes (nodes), configuration files, and so on.

- **Package manifests:** Package manifests provide information about a package, licenses, dependencies, compilation flags, and so on. A package manifest is managed with a file called `package.xml.`

- **Message (msg) types:** A message is the information that a process sends to other processes. ROS has a lot of standard types of messages. Message descriptions are stored in `my_package/msg/MyMessageType.msg`

- **Service (srv) types:** Service descriptions, stored in `my_package/srv/MyServiceType.srv` , define the request and response data structures for services provided by each process in ROS.

For a package to be considered a catkin package it must meet a few requirements:

1. The package must contain a catkin compliant `package.xml` file.
   That `package.xml` file provides meta information about the package.

2. The package must contain a `CMakeLists.txt` which uses `catkin`.

3. Each package must have its own folder. This means no nested packages nor multiple packages sharing the same directory.

The simplest possible package might have a structure which looks like this:

```
my_package/
  CMakeLists.txt
  package.xml
```

Packages usually reside inside ROS workspaces. A trivial workspace might look like this:

```
workspace_folder/        -- catkin ROS WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- 'Toplevel' CMake file, provided by catkin
    package_1/
      CMakeLists.txt     -- CMakeLists.txt file for package_1
      package.xml        -- Package manifest for package_1
    ...
    package_n/
      CMakeLists.txt     -- CMakeLists.txt file for package_n
      package.xml        -- Package manifest for package_n
```

[D.1] Creating a `catkin` Package

Lets change to the source space directory of the catkin workspace you created earlier in Section [B].

```
# You should have created this in the Creating a Workspace Tutorial
$ cd ~/catkin_ws/src
```

Now use the catkin_create_pkg script to create a new package called 'beginner_tutorials' which depends on `std_msgs`, `roscpp`, and `rospy`:

```
$ catkin_create_pkg my_package std_msgs rospy roscpp
```

This will create a `my_package` folder which contains a `package.xml` and a `CMakeLists.txt`, which have been partially filled out with the information you gave `catkin_create_pkg`.

`catkin_create_pkg` requires that you give it a package_name and optionally a list of dependencies on which that package depends:

```
$ catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

The output will look something like this:

```
madhur@ubuntu:~/catkin_ws/src$ catkin_create_pkg my_package std_msgs rospy roscpp
Created file my_package/package.xml
Created file my_package/CMakeLists.txt
Created folder my_package/include/my_package
Created folder my_package/src
Successfully created files in /home/madhur/catkin_ws/src/my_package.</br>
Please adjust the values in package.xml.
madhur@ubuntu:~/catkin_ws/src$
```

[D.2] Building a catkin workspace and sourcing the setup file

Now you need to build the packages in the catkin workspace:

```
$ cd ~/catkin_ws
$ catkin_make
```

The output will look something like this:

```
madhur@ubuntu:~/catkin_ws$ catkin_make
Base path: /home/madhur/catkin_ws
Source space: /home/madhur/catkin_ws/src
Build space: /home/madhur/catkin_ws/build
Devel space: /home/madhur/catkin_ws/devel
Install space: /home/madhur/catkin_ws/install
####
#### Running command: "cmake /home/madhur/catkin_ws/src
.
.
.
```

```
-- +++ processing catkin package: 'my_package'
-- ==> add_subdirectory(my_package)
-- Configuring done
-- Generating done
-- Build files have been written to: /home/madhur/catkin_ws/build
####
#### Running command: "make -j2 -l2" in "/home/madhur/catkin_ws/build"
####
```

Add the workspace to your ROS environment you need to source the generated setup file:

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

[D.3] package.xml: The package manifest

The generated package.xml should be in your new package.
You can go through the new package.xml and touch up any elements that need your attention.

[Q3] Use a ROS command from section C to list the directory and files (e.g. `package.xml`) under the newly created `beginner_tutorial package` ?

[Q4] Change the current working directory to the directory which contains the `package.xml` manifest using a single ROS command.

The `package.xml` would look like this:

```
 1 <?xml version="1.0"?>
 2 <package format="2">
 3   <name>my_package</name>
 4   <version>0.1.0</version>
 5   <description>The my_package package</description>
 6
 7   <maintainer email="you@yourdomain.tld">Your Name</maintainer>
 8   <license>BSD</license>
 9   <url type="website">http://wiki.ros.org/beginner_tutorials</url>
10   <author email="you@yourdomain.tld">Jane Doe</author>
11
12   <buildtool_depend>catkin</buildtool_depend>
13
14   <build_depend>roscpp</build_depend>
15   <build_depend>rospy</build_depend>
16   <build_depend>std_msgs</build_depend>
17
18   <exec_depend>roscpp</exec_depend>
19   <exec_depend>rospy</exec_depend>
20   <exec_depend>std_msgs</exec_depend>
21
22 </package>
```

[E] Python Publisher and Subscriber.

Step 1)

Create a new directory in your home folder to clone the `f1tenth-course-labs` Github 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 folders with ROS packages.
For example, the repository contains a `beginner_tutorials` ROS package, which we will use for illustrating the next step.

Step 3)

Copy the contents of the `f1tenth-course-labs` directory in the github folder to the `src` folder of the catkin workspace.

After copying the contents of the github repo, you will find that a new folder called `scripts` has beed added to the `beginner_tutorials` package.
Make sure all the python scripts are executable:
Run:

```
cp -r ~/github/f1tenth-course-labs/. ~/catkin_ws/src/
cd ~/catkin_ws
find . -name “*.py” -exec chmod +x {} \;
catkin_make
```

Step 4)
Finally, as you may have already guessed, we need to source our new package that was just added to the ROS workspace.

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

Thats it !, you should now be able to run any ROS command that interacts with the `beginner_tutorials` package.

Step 4)
Lets run the two nodes `talker` and `listener` we have just build with `catkin_make`

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 1394915043.55
[INFO] [WallTime: 1394915043.654982] /listener_9056_1394915043253I heard hello world 1394915043.65
[INFO] [WallTime: 1394915043.754936] /listener_9056_1394915043253I heard hello world 1394915043.75
[INFO] [WallTime: 1394915043.854918] /listener_9056_1394915043253I heard hello world 1394915043.85
...
```

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

```
1 #!/usr/bin/env python
2 import rospy
3 from std_msgs.msg import String
4
5 def callback(data):
6     rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
7
8 def listener():
9
10     # In ROS, nodes are uniquely named. If two nodes with the same
11     # node are launched, the previous one is kicked off. The
12     # anonymous=True flag means that rospy will choose a unique
13     # name for our 'listener' node so that multiple listeners can
14     # run simultaneously.
15     rospy.init_node('listener', anonymous=True)
16
17     rospy.Subscriber("chatter", String, callback)
18
19     # spin() simply keeps python from exiting until this node is stopped
20     rospy.spin()
21
22 if __name__ == '__main__':catkin package as we just changed some code
23     listener()
```

[F] 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..

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*

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`.

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.
```

[G] 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()
```