

ROS NOTES

version 2/12/2020

Run existing nodes

0. Make sure the catkin setup has been sourced.

```
$ echo $ROS_PACKAGE_PATH #should give something like
>> /home/tm/catkin_ws/src:/opt/ros/kinetic/share
```

**if your catkin_ws is missing, source the catkin setup. This will need to be done in each terminal unless you've done it permanently. Highly recommend doing it permanently.

Permanent catkin source

- open ~/.bashrc in a text editor
- find source /opt/ros/[distro]/setup.bash
- under that add source ~/yourPathTo/catkin_ws/devel/setup.bash
- save

Catkin source in each terminal

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

1. Start ROS

```
$ roscore
```

2. Run node in new terminal

```
$ rosrun [package_name] [node_name] or, list with $ rosrun [package_name] <tab><tab>
    - optional- can rename node like
    $ rosrun [package_name] [node_name] __name:=my_name
```

3. Run as many nodes as needed in new terminals

OR

1. Start ROS

```
$ roscore
```

2. roslaunch to do a defined setup

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

-to set up the launch file, see <http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch>

ROS filesystem and commands

rospack – gets path about packages. **find** returns path to packages

```
$ rospack find roscpp
```

-list nodes in package- use autocomplete. \$ rosrun package_name <tab><tab>

roscd – change directory directly to package or stack (don't need path)

```
$ roscd roscpp or $ roscd roscpp/cmake
```

*roscd and other ros tools will only find ros packages within directories on the
ros_package_path

```
$ echo $ROS_PACKAGE_PATH
```

rosls – list information about package by name instead of by path

```
$ rosls roscpp_tutorials
```

```
>> cmake launch package.xml srv
```

rosls

list – see what is running

```

$ rosnode list
-cleanup-- refreshes rosnode list (after something has been killed)
$ rosnode cleanup
info – gives publications, subscriptions, services, etc of a node
$ rosnode info /[node]      ex. $ rosnode info /rosout
ping – check connection
$ rosnode ping [node_name]

roslaunch – runs node
$ roslaunch turtlesim turtlesim_node (optional: __name:=loggerhead)

rqt_graph – makes bubble chart graph of what is happening in the system.
$ roslaunch rqt_graph rqt_graph

rqt_plot – plot the data being published on a topic in real time.
$ roslaunch rqt_plot rqt_plot

rosmmsg – get info about messages
$ rosmmsg list #see all messages
$ rosmmsg package <package-name> # see all msgs in a package
$ rosmmsg packages # list the packages with msgs

rostopic – get info about topics.
Example topics: /turtle1/cmd_vel, /turtle2/pose, /rosout. Also use autocomplete
$ rostopic -h #explains the rostopic commands
$ rostopic echo [topic] #print messages to the screen
$ rostopic list #list of topics. End with -v to get verbose published and subscriptions
$ rostopic type [topic] # gives message type sent on a topic
    -then to see details of the msg type, do
$ rostopic type [topic] | rosmmsg show # $ rosmmsg show geometry_msgs/Twist

Send ROS message manually
$ rostopic pub [topic] [msg_type] [args]
    # $ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist – '[2.0, 0, 0]' '[0, 0, 1.8]'
    #the -1 means it only sends one message. Use -r to repeat ex. [-r 1] means repeat at 1hz
$ rostopic hz [topic] #show frequency a topic is being published to

rosservice – attach to client/service framework. Can use list, call, type, find, uri
# turtle example services: /clear, /kill, /reset, /spawn
$ rosservice type [service] #tells the type of the argument
    -$ rosservice type [service] | rossrv show #to see the actual arguments
$ rosservice call [service] [args]
    ex. $ rosservice call /spawn 8 6 .5 "" #parameters x, y, theta, name(optional)
    ex. $ rosservice call /clear #this reset the background color and the tracks on the turtle

```

rosparam – lets us store and change data on ROS parameter server. Stores int, float, bool, dict, lists.

```
$ rosparam -h          #show commands
$ rosparam [set or get] [param_name]      # set parameter
$ rosparam get /        #gives all contents of parameter server
$ rosparam dump [file_name] [namespace]
    ex. $ rosparam dump params.yaml
– can load yaml file into a new namespace, for example “copy”
$ rosparam load params.yaml copy
$ rosparam get /copy/background_b
```

ROS Basics and Conventions

<http://wiki.ros.org/ROS/Patterns/Communication>

rospy and roscpp – client libraries, allows ROS to communicate cross-language with python/cpp nodes

Package – software organization unit. Can hold libraries, executables, scripts... Like a folder

- Package names should follow C naming conventions: lower case, start with a letter, use underscore separators, e.g. laser_viewer
- names should be specific enough to identify what the package does. Ex. a motion planner might be called waypoint_planner, but not planner.

Node – executable that uses ROS to communicate with other nodes through topics.

- can have multiple nodes in one package
- a python script. A process that performs computation. An executable in a ROS package
- have ‘**type**’ and ‘**name**’. Type is the name of the executable used to launch the node. Name is what goes to other ROS nodes when it starts. Named when doing rospy.init_node(‘name’,...)
- node type- keep the type name short.
Ex. package= laser_scan, node type= view, \$ rosrn laser_scan view
- node name- want default name of node to be same as executable that launches node. Rename it at startup if needed, not in code
- nodes can **publish, subscribe**, provide/use a **service**...

Topic- nodes publish/subscribe to topics to receive msgs

- Should be used for continuous data stream (sensor data, robot state...)
- many to many connection
- callbacks receive data when it is available
- data can be published/subscribed at any time, independent of senders/receivers
- publisher decides when data is sent

- should follow C naming. ex. laser_scan
- name should be descriptive. Don't call topic 'state', call it 'planner_state'.
- ex. Can have topic named 'out' on A(publisher), topic 'in' on B(subscriber), and as long as the later- defined topic name is the same and they are on same ROS master they can communicate. DON'T try to make topic names match inside nodeA and nodeB!

Message – ROS data type for subscribing/publishing to a topic

- used to auto generate class names. Must name with camelCase ex laserScan.msg

Services – used for remote calls that terminate quickly. For querying state of node or quick calculations.

- not for longer running processes or anything that might be preempted
- blocking. For requesting specific data. Semantically for processing requests

RQT_graph – nodes are circles, topics are lines

Action – actions are built on top of msgs

- should be used for anything that moves robot, or runs for a longer time with feedback during execution
- can be preempted, preemption should always be implemented cleanly by action servers
- can execute toward multiple action goals on the same server (multiple clients)
- action clients request goals
- action servers execute towards goals with function calls and callbacks
- **goal** -sent to actionServer by actionClient.
- **feedback** – sent to actionClient to give incremental progress towards goal
- SimpleActionServer only has one active goal at a time (always the most recent)
- **result** – sent from actionServer to actionClient when goal is completed. Sent only once. Ex. Moving to a location result= 'finished', but for a laser scan result= scanData

TF or tf2 – in general, where you publish coordinate frames or spatial data. There are exceptions

- holds relationships between frames over time. Lets you transform points, vectors, etc at any point in time

Creating catkin workspace

```
-create and build catkin workspace
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
```

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.

To make sure your workspace is properly overlayed by the setup script, make sure ROS_PACKAGE_PATH environment variable includes the directory you're in.

```
$ echo $ROS_PACKAGE_PATH
>> /home/youruser/catkin_ws/src:/opt/ros/kinetic/share
```

ex. finished catkin workspace example (this is not what it looks like immediately after creation)

```
workspace1/      #catkin 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
  scripts/       # only if there are python files
    file1.py
  src/           # only if there are cpp files
    file2.cpp
  msg/           #only if there are custom msgs
    customMsg1.msg
  include/       # header files for cpp
    package_1/
      file2.h
  action/        # only if using action: a special msg file
    action1.action
  launch/        #may be empty if not using launch files
    package_1.launch
...
package_n/
  CMakeLists.txt # CMakeLists.txt file for package_n
  package.xml    # Package manifest for package_n
```

Create a package

Package will have 1. CmakeLists.txt 2. package.xml (with dependency and meta info about the file) 3. be in its own folder (No nested packages). /scripts, /src, etc are used as needed.

1. change to source space directory of workspace

```
$ cd ~/catkin_ws/src
```

2. 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 beginner_tutorials std_msgs rospy roscpp
```

This will create a beginner_tutorials folder which contains a [package.xml](#) and a [CMakeLists.txt](#), which have been partially filled out with the information you gave catkin_create_pkg.

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

catkin_make combines cmake and make from standard cmake workflow

build- where cmake and make are called to configure packages.

devel-- devel space, where my executables and libraries go before installing packages

```
$ cd ~/catkin_ws
```

```
$ catkin_make
```

4. To add the workspace to your ROS environment you need to source the generated setup file:

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

Package dependencies

First order dependencies- ones we provide (std_msgs, rospy, roscpp above). Also stored in package.xml

```
$ rospack depends1 beginner_tutorials
>>   roscpp
      rospy
      std_msgs
```

overall dependencies – ros will recursively find all dependencies.

ex. Will go through roscpp, rospy, and std_msgs to find everything they depend on too

Developing

Editing - \$ rosed [package_name] [file_name] #opens file in nano for editing

debugging with rqt_console

-<http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch>

copy file – \$ roscp [package_name] [path_of_file] [destination_path]

ex. \$ roscp rospy_tutorials AddTwoInts.srv srv/AddTwoInts.srv

Creating msg and srv

<http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv>

*see ‘ROS basics and conventions’ for naming help

msg – text file that describes fields of ROS message. Used to generate source code for messages in different languages. Stored in msg directory of package.

```
$ rosmmsg show geometry_msgs/Twist #do this to see an example
-can be int, float, string, time, array, or other msg files, header
```

```
ex.   Header header
      string child_frame_id
      geometry_msgs/PoseWithCovariance pose
```

Create new msg file – ex. write “int64 num” to file Num.msg in folder msg

```
0.   $ roscd [package] # $ roscd beginner_tutorials
      $ mkdir msg      #make a msg directory/folder (if it doesn't exist)
```

```
$ echo "int64 num" > msg/Num.msg
- [>] overwrites anything currently in the file. Creates new file if necessary.
-Use [>>] to append to the file
```

1. write msg file (see step 0, above)

2. Check package.xml for: (in build_depend and exec_depend)

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

3. Check CMakeLists.txt for:

```
3.1. find_package(catkin REQUIRED COMPONENTS
    ...
    message_generation
)
3.2. catkin_package(
    ...
    CATKIN_DEPENDS message_runtime ...
)
3.3. add_message_files(
    ## adding .msg here manually lets Cmake know when it has to reconfigure
    FILES
    msg_file1.msg                #ex. Num.msg
    ...
)
3.4. generate_messages(
    DEPENDENCIES
    std_msgs
    #plus any other packages with .msg files you use
)
```

4. Now you're ready to generate source files from the msg definition.

srv – srv describes a service. Made of **request** and **response**. Stored in srv directory of package

```
ex.  int64 A      #request
      int64 B      #request
      ---
      int64 Sum    #response
```

```
$ rossrv show [srv_name]           #returns the format of the service
```

Create new srv file--

1. write or copy srv file (see above, "Create new msg file" for help)

2. Exact same as for msg

3. Check CMakeLists.txt for:

```
3.1. Exact same as above
3.2. add_service_files(
    FILES
    srv_file1.srv                #ex. AddTwoInts.srv
    ...
)
3.3. generate_messages(
    DEPENDENCIES
    std_msgs
    #plus any other packages with .msg files you use
)
```

4. Now you're ready to generate source files from the srv definition

Generating Source Files

1. Remake the package

```
# In your catkin workspace
$ roscd [package]           # roscd beginner_tutorials
```

```
$ cd ../../
$ catkin_make install
$ cd -          #cd back to the last spot?
```

Writing publisher and subscriber nodes (Python)

<http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29>

Publisher

```
1 #!/usr/bin/env python                                ***1**
2
3 import rospy                                         ***2**
4 from std_msgs.msg import String
5
6 def talker():
7     pub = rospy.Publisher('chatter', String, queue_size=10)  ***3**
8     rospy.init_node('talker', anonymous=True)                ***4**
9     rate = rospy.Rate(10) # 10hz                             ***5**

#start loop, check flag to see if we should quit
10 while not rospy.is_shutdown():                        ***6**
11     hello_str = "hello world %s" % rospy.get_time()
12     rospy.loginfo(hello_str) #rospy.loginfo(message).
# loginfo does print to screen, writes to node's log file, writes to rosout
# (rosout can be seen with rqt_console, good for debugging)
13     pub.publish(hello_str) # pub.publish(message)
14     rate.sleep() #can use rospy.sleep() to work with simulated time too
15
16 if __name__ == '__main__':                            #standard Python __main__ check
17     try:
18         talker()
19     except rospy.ROSInterruptException:
# above avoids continuing to work when node is shutdown
20         pass
```

0. Make the node executable (should not have to redo after the script is edited and resaved)

```
$ chmod +x file_name.py
```

1. add exactly this to the top of the python script. Every python node will have this.

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

2. import rospy

```
--may want to import other things
```

3. name publisher

```
pub = rospy.Publisher('topic_name', msg_type, queue_size)
```

4. tell rospy the name of our node. Names can't have any "/" in them (must be base name)

```
rospy.init_node('choose_node_name', anonymous=True)
```

```
-anonymous makes sure the node has unique name by adding numbers to the end of it
```

5. rate = rospy.Rate(10) #loops at 10hz

6. Start loop

```
--see comments in code example
```

7. include python __main__ check

```
--calls your function in it
```


8. Run catkin make

```
$ cd ~/catkin_ws
```

```
$ catkin_make
```

9. Before using applications, re-source the catkin setup.bash (might not be necessary with the permanent catkin source).

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

Subscriber- tutorial example

```
__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__     # name 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)
#the subscriber is based on a callback function ("call later") to listen for
messages
__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__':
__23__     listener()
```

Subscriber- Tyler Musgraves example

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

```
import rospy
```

```
from std_msgs.msg import Int64
```

```
class altimeter:
```

```
    #start publisher
```

```
    def pub_altitude(self):
```

```
        self.pub = rospy.Publisher('altitude_m', Int64, queue_size = 5)
```

```
        rospy.init_node('pub_altitude', anonymous=True)
```

```
        rate = rospy.Rate(10)
```

```
        return
```

```
    #callback that gets data from topic, works with it
```

```
    def callback_alt(self, msg_alt):
```

```
        alt_ft = msg_alt.data * 3
```

```
        rospy.loginfo('altitude in m: %i ---- altitude in ft: %i' %(msg_alt.data,
                                                                    alt_ft))
```

```
        return
```

```
    #start subscriber (using callback)
```

```
    def sub_altitude(self):
```

```
        sub = rospy.Subscriber('altitude_m', Int64, self.callback_alt)
```

```

    rospy.spin()    #callbacks get called in spin. If we want to control
                    #period, use spinOnce() and then rate.sleep() or similar
    return

if __name__ == '__main__':
    try:
        plane1 = altimeter() #create object (altimeter)
        plane1.pub_altitude() #start publisher

        while not rospy.is_shutdown():
            plane1.sub_altitude() #run subscriber (and get data from callback)
    except rospy.ROSInterruptException:
        pass
#send messages manually with rostopic pub [topic] [msg_type] [arg]

```

Writing a Simple Service and Client (Python)

<http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28python%29>

Service Node

1. Write script (see example in beginner_tutorials: add_two_ints_server.py)
 - a. declare node
 - ex. `Rospy.init_node('add_two_ints_server')`
 - b. declare service (see example)
 - `S = rospy.Service('service_name', service_type, function)`
 - ex. `S = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)`
2. make executable
 - `$ chmod +x scripts/add_two_ints_server.py`

Client Node

1. Write script (see example in beginner_tutorials: add_two_ints_client.py)

```

__ 1 #!/usr/bin/env python                                     #always need this
__ 2 import sys
__ 3 import rospy
__ 4 from beginner_tutorials.srv import *
__ 5
__ 6
__ 7 def add_two_ints_client(x, y):
__ 8     rospy.wait_for_service('add_two_ints') #blocks the service until
#add_two_ints is available
__ 9     try:
#create a handle for the service
__10         add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
#call the handle like a normal function
__11         resp1 = add_two_ints(x, y)
__12         return resp1.sum
#since the type of the service is AddTwoInts, it generates the AddTwoIntsRequest
(or we can pass in our own). The return is AddTwoIntsResponse object.
__13     except rospy.ServiceException, e:
__14         print "Service call failed: %s"%e
__15
__16 def usage():
__17     return "%s [x y]"%sys.argv[0]
__18

```

```

19 if __name__ == "__main__":
20     if len(sys.argv) == 3:
21         x = int(sys.argv[1])
22         y = int(sys.argv[2])
23     else:
24         print usage()
25         sys.exit(1)
26     print "Requesting %s+%s"%(x, y)
27     print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))

```

- a. Call the service


```
rospy.wait_for_service('service_name')
```
- b. make a handle for the service


```
add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
```
- c. Then use the handle like a normal function


```
resp1 = add_two_ints(x, y)
return resp1.sum
```

Using a Simple Action Server (python)

- see 7.2 <http://wiki.ros.org/actionlib>

-for api details, see

https://docs.ros.org/api/actionlib/html/classactionlib_1_1simple_action_server_1_1SimpleActionServer.html

-Example given that we have defined DoDishes.action in the 'chores' package:

```

#!/usr/bin/env python
import roslib
roslib.load_manifest('my_pkg_name')
import rospy
import actionlib
from chores.msg import DoDishesAction

class DoDishesServer:
    def __init__(self):
#this creates a DoDishes action server named 'do_dishes'
        self.server = actionlib.SimpleActionServer('do_dishes', DoDishesAction,
self.execute, False)
        self.server.start()

    def execute(self, goal):
        # Do lots of awesome groundbreaking robot stuff here
        self.server.set_succeeded()

if __name__ == '__main__':
    rospy.init_node('do_dishes_server')
    server = DoDishesServer()
    rospy.spin()

```

Recording and playing back data (bag file)

<http://wiki.ros.org/ROS/Tutorials/Recording%20and%20playing%20back%20data>

-Record data from a running ROS system into a .bag file, then play back the data to produce similar behavior in a running system.

- commands that are timing-sensitive might not be replayed perfectly. It will get you close, but don't expect too much

0. rosrun everything we want

1. See all running topics

```
$ rostopic list -v
```

2. Make temporary directory

```
$ mkdir ~/bagfiles
```

```
$ cd ~/bagfiles
```

3. Record topics

```
$ rosbag record -a
```

#record all topics

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

#record only topics in
subset.bag

4. Playback topics

```
$ rosbag play <bagfile.bag>
```

#playback at normal speed

```
$ rosbag play -r 2 <bagfile.bag>
```

#playback at double speed

Simulators and etc (next steps)

<http://wiki.ros.org/ROS/Tutorials/WhereNext>

TODO – add a section on actions

Managing your environment

-during installation you have to source one of several setup.*sh files

-check environment variables like ROS_ROOT and ROS_PACKAGE_PATH with

```
$ printenv | grep ROS
```

CmakeLists.txt example

For a package “beginner_tutorials” with:

- 1 service, 1 msg

- multiple python files in /scripts

- no c++ files

```
cmake_minimum_required(VERSION 2.8.3)
```

```
project(beginner_tutorials) #the package name
```

```
find_package(catkin REQUIRED COMPONENTS
```

```
  roscpp
```

```
  rospy
```

```
  std_msgs
```

```
  message_generation
```

```
)
```

```
add_message_files(FILES
```

```

        Num.msg
    )
    add_service_files(FILES
        AddTwoInts.srv
    )
    generate_messages(DEPENDENCIES
        std_msgs
    )
    catkin_package(
        CATKIN_DEPENDS roscpp rospy std_msgs message_runtime
    )
    include_directories(
        ${catkin_INCLUDE_DIRS}
    )

```

package.xml example

```

- for the same package as above
<package format="2">
<name>beginner_tutorials</name>
<version>0.0.1</version>
<description>Tyler's tutorial description</description>
<maintainer email="tm@todo.todo">tm</maintainer>
<license>BSD</license>

<buildtool_depend>catkin</buildtool_depend>

<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<build_export_depend>roscpp</build_export_depend>
<build_export_depend>rospy</build_export_depend>
<build_export_depend>std_msgs</build_export_depend>
<exec_depend>roscpp</exec_depend>
<exec_depend>rospy</exec_depend>
<exec_depend>std_msgs</exec_depend>

<export>
</export>
</package>

```

Linux Notes

apt-cache search ros-kinetic | **grep** python

-grep searches the output for the words "python" and highlights it

pwd – print working directory

cat – show file \$ cat package.xml

Transfer file from one user to another

cp file_name /tmp/

chmod a+r /tmp/file_name

```
sudo -u user_two cp /tmp/file ~user_two  
rm /tmp/file_name
```