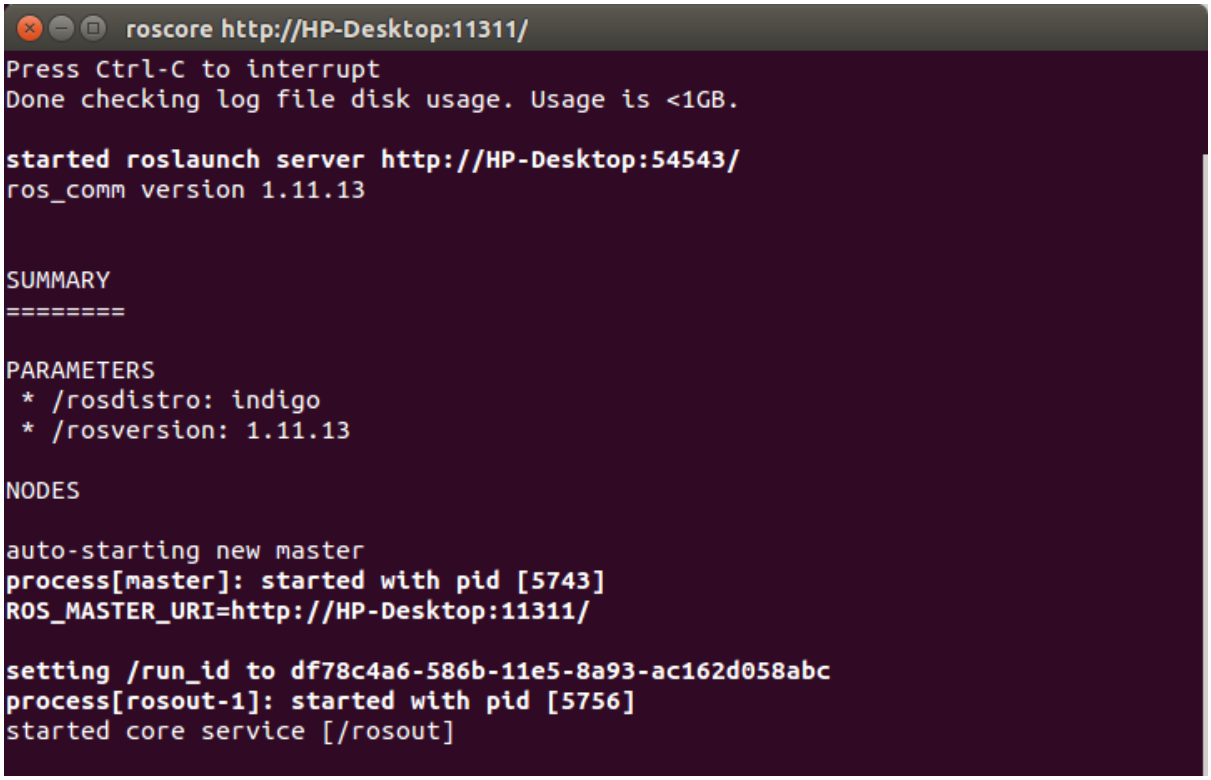
Introduction to ROS -
The Robot Operating System

Version	Date	Author	Comments
1.0	Sept 2014	Arthur Richards	New document
1.1	Sept 2015	Colin Greatwood	Updates for Indigo
1.2	Sept 2017	Arthur Richards	Updates for Kinetic
1.3	Jan 2018	Arthur Richards	Rationalized for Short Course

1 Setting up ROS on the Linux PCs in BRL PC Teaching Room

First steps: open a new terminal window (Ctrl+Alt+T) and type “roscore”.



1.1 Creating your ROS Workspace

Open up a terminal and type the following to create the workspace

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

Packages in your workspace are compiled using the following, test this now

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

To let ROS know where the workspace is you should update your .bashrc with the following command

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

The double chevrons after an echo allow you to append to a file, we can check how this has changed the .bashrc by typing

```
$ less ~/.bashrc
```

Press Q to exit the file preview. Now reopen the terminal or type

```
$ source ~/.bashrc
```

We can check that this all worked by looking at the ROS package path with

```
$ echo $ROS_PACKAGE_PATH
```

which should provide the following output

/home/netlab/USERNAME/catkin_ws/src:/opt/ros/kinetic/share

2 Drive a Turtle

Start a ROS core (the hub of all ROS communications – you'll get "cannot communicate with master" errors if you try and do anything ROS-based without a core running)

```
$ roscore
```

Now in a new terminal or tab (press Ctrl+Shift+T), run the turtle simulator:

```
$ rosrun turtlesim turtlesim_node
```

And now manually send a command to the turtle

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist '{linear:  {x: 1.2, y: 0.0, z: 0.0}, angular: {x: 0.0,y: 0.0,z: 0.2}}'
```

And then use RQT for the same job

```
$ rqt
```

3 Creating a ROS Package

Create a package called "ros_course" for use with all the exercises

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg ros_course std_msgs rospy turtlesim
$ cd ..
$ catkin_make
```

This creates a package called ros_course, which depends on the packages std_msgs, rospy and turtlesim

Check ROS has found your new package by looking for it in the package list:

```
$ rospack list
```

4 A Publisher Node

```
#!/usr/bin/python
import rospy
from geometry_msgs.msg import Twist
import random

# set up a publisher
pub = rospy.Publisher('turtle1/cmd_vel', Twist, queue_size=1)
# start the node
rospy.init_node('driver')
# will be updating at 2 Hz
r = rospy.Rate(2)

while not rospy.is_shutdown():
   # make a blank velocity message
   msg = Twist()
   # pick a random direction
   msg.angular.z = 2*(random.random() - 0.5)
   # constant speed
   msg.linear.x = turtle_speed
   # publish it
   pub.publish(msg)
   # show a message
   rospy.loginfo("New turn rate=%s"%msg.angular.z)
   # wait for next time
   r.sleep()
```

Typically python nodes are stored in the scripts directory within a package, so for our ros_course package your python script would be located in:

`~/catkin_ws/src/ros_course/scripts`

Let's say you called it "drive.py". Make it executable by running "chmod +x drive.py"

Run it by doing each of the following in a fresh terminal window:

1. roscore
2. rosrun turtlesim turtlesim_node
3. ./drive.py (*Note: the dot-slash is important, as Linux will refuse to execute it otherwise*)

You should see a turtle driving around drunkenly. Use "rostopic" and "rqt" to poke around and see what's happening.

5 A subscriber node

```
#!/usr/bin/python
import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
from math import sqrt

# start the node
rospy.init_node('listen')

# callback for pose does all the work
def pose_callback(data):
  rospy.loginfo("x is now %f" % data.x)

# and the subscriber
rospy.Subscriber("turtle1/pose",Pose,pose_callback)
rospy.spin()
```

Follow the three steps in the “publisher” example above and then run this program as well. Don’t forget to chmod it.

6 Closing a Loop

Run each one of these alongside the turtlesim_node as before. You should get the same results each time – a turtle that tries to follow a circle. Note the differences in the code and use "rostopic hz" to check the frequency of the "cmd_vel" topic output.

Method 1: do the work in the callback function. OK for simple stuff, but means you cannot control the rate. Also, if you do too much work in the callback, can you be sure it finishes before the next message comes in?

```
#!/usr/bin/python
import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
from math import sqrt

# start the node
rospy.init_node('bounce')
# set up a publisher
pub = rospy.Publisher('turtle1/cmd_vel', Twist, queue_size=3)

# callback for pose does all the work
def poseCallback(data):
  radius = sqrt((data.x-5.0)**2 + (data.y-5.0)**2)
  rospy.loginfo("radius is now %f" % radius)
  turn_rate = 0.3*(radius - 4.0)
  msg = Twist()
  msg.linear.x = 0.5
  msg.angular.z = turn_rate
  pub.publish(msg)

# and the subscriber
rospy.Subscriber("turtle1/pose",Pose,poseCallback)
rospy.spin()
```

Method 2: store data from the callback but publish in your own loop. More control, but risk of jitter when data gets overlooked. Also, if your loop depends on more than one input topic, you're forced to do something like this. The global variable is ugly – we'll fix that in a moment.

```
#!/usr/bin/python
import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
from math import sqrt

# start the node
rospy.init_node('bounce')
# set up a publisher
pub = rospy.Publisher('turtle1/cmd_vel', Twist, queue_size=3)

# initialize global
radius = 4.0

# callback for pose does all the work
def poseCallback(data):
  global radius
  radius = sqrt((data.x-5.0)**2 + (data.y-5.0)**2)
  rospy.loginfo("radius is now %f" % radius)

# start the subscriber
rospy.Subscriber("turtle1/pose",Pose,poseCallback)

# main control loop
r = rospy.Rate(10)
while not rospy.is_shutdown():
  turn_rate = 0.3*(radius - 4.0)
  msg = Twist()
  msg.linear.x = 0.5
  msg.angular.z = turn_rate
  pub.publish(msg)
  r.sleep()
```

Method 2.1: identical in operation to Method 2, but with better Python style. Note how you can use the method when setting up the "Subscriber". Arguably you could put the "run" stuff in the constructor "__init__" but note that the callbacks will start as soon as the Subscriber is made.

```
#!/usr/bin/python
import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
from math import sqrt

class TurtleControlNode:

  def __init__(self):
    self.radius = 4.0
    # start the node
    rospy.init_node('loop_tidy')
    # set up a publisher
    self.pub = rospy.Publisher('turtle1/cmd_vel', Twist, queue_size=3)
    # rate and control
    self.rate = rospy.Rate(10)
    self.msg = Twist()

  def poseCallback(self,data):
    self.radius = sqrt((data.x-5.0)**2 + (data.y-5.0)**2)
    rospy.loginfo("radius is now %f" % self.radius)

  def run(self):
    # start the subscriber
    rospy.Subscriber("turtle1/pose",Pose,self.poseCallback)
    # main control loop
    while not rospy.is_shutdown():
      turn_rate = 0.3*(self.radius - 4.0)
      self.msg.linear.x = 0.5
      self.msg.angular.z = turn_rate
      self.pub.publish(self.msg)
      self.rate.sleep()

if __name__=='__main__':
  t = TurtleControlNode()
  t.run()
```

7 Parameters

For the next two sections, we need to use a modified form of the original publisher code from Section 4. You need to add the six lines from "# get name for topic" up to "turtle_speed=…" and change the "msg.linear.x=…" line in the main loop.

```
#!/usr/bin/python
import rospy
from geometry_msgs.msg import Twist
import random

# set up a publisher
pub = rospy.Publisher('turtle1/cmd_vel', Twist, queue_size=1)
# start the node
rospy.init_node('driver')
# will be updating at 2 Hz
r = rospy.Rate(2)

# get name for topic
cmd_vel_name = rospy.resolve_name('turtle1/cmd_vel')
# show it
rospy.loginfo("Publishing to topic %s" % cmd_vel_name)

# get speed from parameter
turtle_speed = rospy.get_param('turtle1/turtle_speed',1.0)

while not rospy.is_shutdown():
   # make a blank velocity message
   msg = Twist()
   # pick a random direction
   msg.angular.z = 2*(random.random() - 0.5)
   # constant speed
   msg.linear.x = turtle_speed
   # publish it
   pub.publish(msg)
   # show a message
   rospy.loginfo("New turn rate=%s"%msg.angular.z)
   # wait for next time
   r.sleep()
```

Again, get the roscore and the turtle running.

First start this node as is, and you should see messages go by saying the speed and turn being published and see the turtle move. Now kill the node again.

Type the following, and then run the node again. You should see speed change.

```
$ rosparam set /turtle1/turtle_speed 0.2
```

Play with the speed. Negatives will work. Also try using "rosparam" to delete the parameter.

8 Exercise:

Improve your turtle driver node so that it reverses direction if it hits the wall.

9 Namespaces and Remapping

Notice in the last revision, we added a couple of lines using “resolve_name” to check the name of the topic we were using. We will use this now to explore how ROS nodes can be re-used in a modular way. Before running it, let’s get two turtles going. Start roscore and turtlesim as before (if they’re not already running). Now run a second turtle as follows:

```
$ ROS_NAMESPACE=/t2 rosrun turtlesim turtlesim_node
```

Can you drive the second turtle? Use rqt, rqt_graph and rostopic to poke around.

Or what about this one:

```
$ ROS_NAMESPACE=/t2 rosrun turtlesim turtlesim_node
/t2/turtle1/cmd_vel:=/turtle1/cmd_vel
```

See the output and have a poke around with “rostopic” and “rqt_graph” to see what’s going on.

Exercise: Now, can you use your turtle controller code (i.e. the code that bounced the turtle off the wall) twice to control both our two turtles? ***Don’t touch the source code.*** Did it by remapping? Now do it by namespaces, or vice versa.

10 Launch Files

Tired of opening terminals yet? Put this example launch file into a new "launch" subdirectory of your package (`~/catkin_ws/src/ros_course/launch/`), named something like "myturtle.launch"

```
<launch>
  <node name="turtle1" pkg="turtlesim" type="turtlesim_node" />
  <node name="control1" pkg="ros_course" type="naming.py" />
</launch>
```

Close everything else down, including the roscore window, and run "roslaunch ros_course my.launch". Try using "rqt_console" and "rqt_graph" to monitor what's going on.

Here's an example moving everything to a lower namespace using a launch file:

```
<launch>
  <group ns="bob">
    <node name="turtle1" pkg="turtlesim" type="turtlesim_node" />
    <node name="control1" pkg="ros_course" type="naming.py" />
  </group>
</launch>
```

Again, use rostopic and rqt_graph to see the effect.

We can launch two different turtles and their controllers by "including" the same "my.launch" from above, twice. We'll set the parameter for one of them to make it slower. The "$(find ros_course)" locates your ros_course package, this is required if you call the launch script from a different location to my.launch. Pay close attention to the getparam call in the controller node – it must be relative to the current namespace, not absolute, for this to work, i.e. no leading "/".

```
<launch>
  <group ns="bob">
    <include file="$(find ros_course)/launch/my.launch"/>
  </group>
  <group ns="margaret">
    <param name="turtle1/turtle_speed" value="0.3" />
    <include file="$(find ros_course)/launch/my.launch"/>
  </group>
</launch>
```

Finally a remap example: the turtle will be in a namespace, but the controller will not.

```
<launch>
  <group ns="bob">
    <node name="turtle1" pkg="turtlesim" type="turtlesim_node" />
  </group>
  <node name="control1" pkg="ros_course" type="naming.py">
    <remap from="/turtle1/cmd_vel" to="/bob/turtle1/cmd_vel" />
  </node>
</launch>
```

Exercise: Write a launch file that runs two turtles, each controlled by a wall-bounce controller.

11 Obstacle Avoidance

Without changing any existing nodes, add a capability to avoid an obstacle centred at position (5, 5). This should be embodied in a new node, which intercepts the command velocity between your current controller and the turtle. In its simplest form, it should stop the turtle if it gets too close to the obstacle; otherwise, it passes along the velocity command it received. Put it all together in a launch file, including necessary remaps or namespace changes. ***You should not need to change any existing node source files. Only edit the new node and the new launch file.***

12 Networking

Partner with someone on a PC next to you. We'll call the two PCs "PC A" and "PC B". Make sure you get these clearly identified as you'll need to set them up carefully.

Identify PC A's IP address by running "ifconfig". Somewhere in the output, you'll see an address like "164.11.73.XX". Write it down. Do the same for PC B.

On PC A, run "ping <PC B's IP addr>" and check you get "reply" messages. Repeat vice versa on PC B.

On PC A, make a file named "roscommA.sh" as follows:

```
export ROS_IP=<PC A's IP addr>
```

Then on PC B, make another file named "roscommB.sh" as follows:

```
export ROS_IP=<PC B's IP addr>

export ROS_MASTER_URI=http://<PC A's IP addr>:11311
```

Back on PC A:

1. Open a new window or tab
2. Run "source roscommA.sh"
3. Run roscore
4. Open a new window or tab
5. Run "source roscommA.sh"
6. Run "rosrun turtlesim turtlesim_node"

And then over on PC B:

7. Open a new window or tab
8. Run "source roscommB.sh"
9. Run "rostopic list"
10. Run "rostopic echo /turtle1/pose"
11. Run "rqt" and drive the turtle over on PC A.