Lecture 2 (7/9) - Middleware for Autonomous Systems & Introduction to Robot Operating System

Starts 07 September, 2020 1:00 PM

Middleware for Autonomous Systems & Introduction to ROS

We meet in Zoom at 13:00. Link: https://dtudk.zoom.us/j/63569471741?pwd=emNGMEQxNitXdGFhc3hSZXhqdVFNQT09

Reading Material

A Gentle Introduction to ROS. [link]
Author: Jason M. O'Kane

Time Plan

- 13:00 Introduction to Middleware for Autonomous Systems (AS)
- 14:00 Introduction to ROS
- 15:00 exercises with support on Discord (use link: https://discord.gg/AEsFTp9)

Description

In this lecture we will:

- Explore the different available middleware.
- We will look into the different middleware architectures and discuss pros and cons.
- We will land on the Robot Operating System (ROS) and have a tour around it!
- 13:00 Introduction to Middleware for Autonomous Systems (AS)
- 14:00 Introduction to ROS

66.67 % 2 of 3 topics complete

Lecture Slides

PDF document

updated 9.Sept (11:56)

Exercise 2.1 - Topics, Messages and nodes

Exercise 2.1 - Topics, messages and nodes :

Open a terminal and start the master:

```
roscore
```

Open an other terminal (CTLR+SHIFT+TAB) and type

```
rosrun turtlesim turtlesim_node
```

The following window will appear:

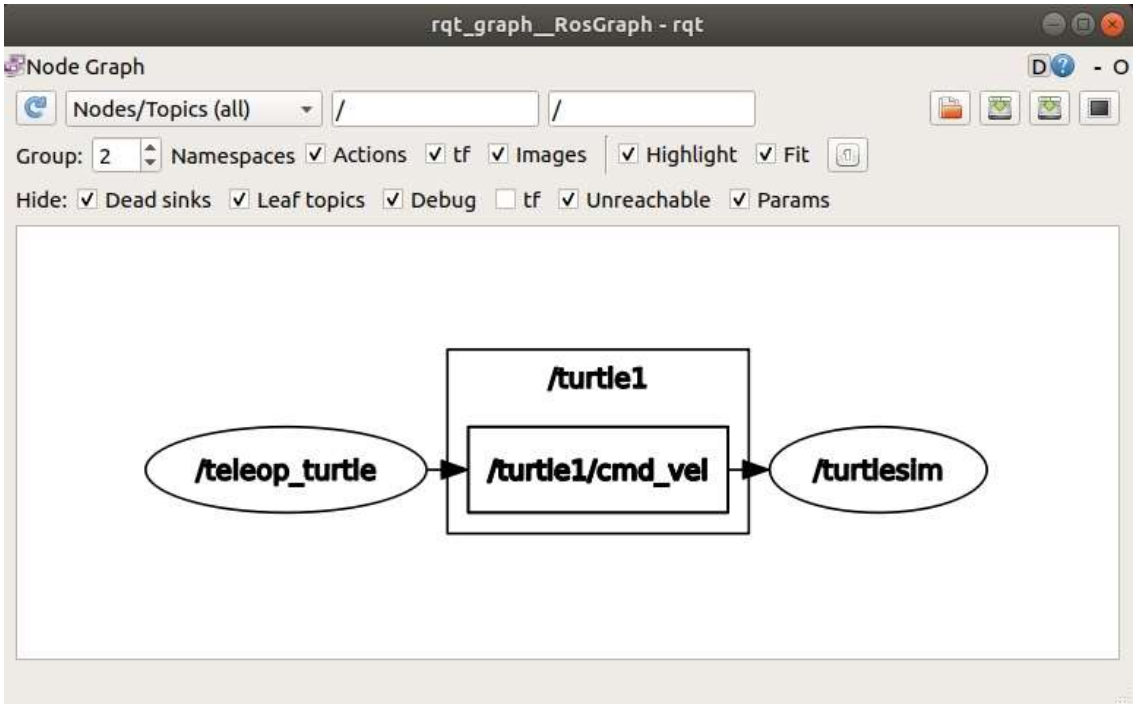
Open an other terminal (CTLR+SHIFT+TAB) and type

```
rosrun turtlesim turtle_teleop_key
```

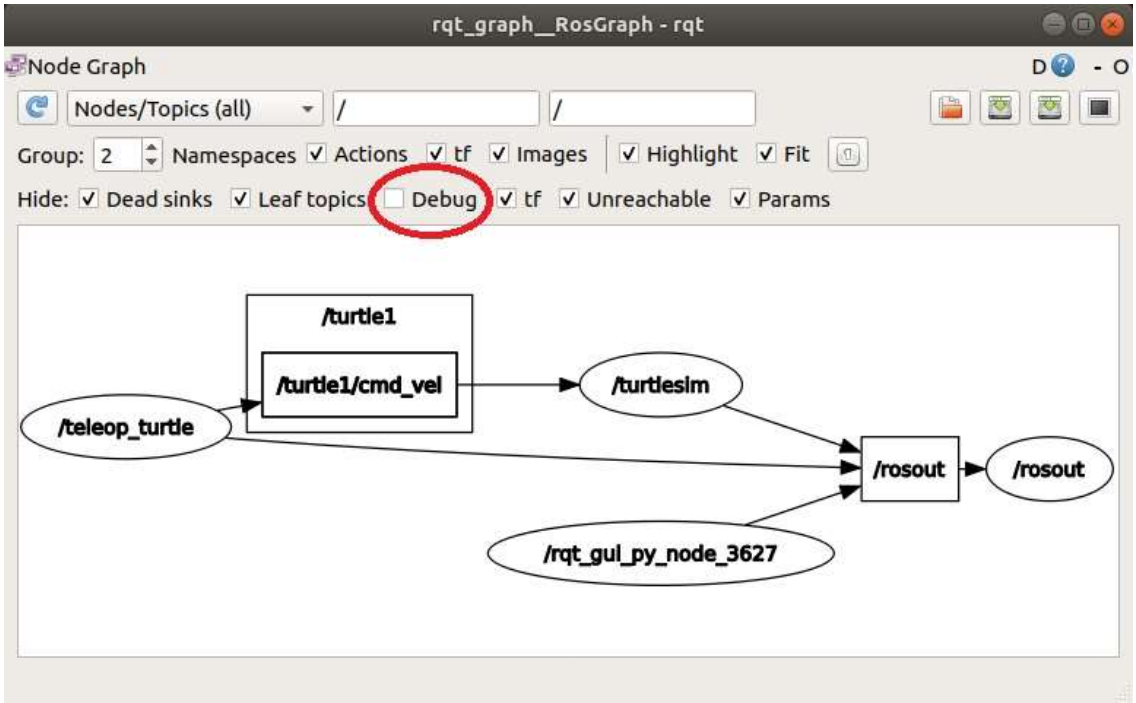
By pressing the arrow keys on the keyboard you can move the turtle around. Note that the terminal with the turtle_teleop_key node must be selected for your input to be registered.

LETS discus what just happened. In other terminal type:

```
rqt_graph
```



To see the full structure with roscore included please uncheck the debug flag:



Let's get some more insight.

The nodes **teleop_turtle** and **turtlesim** communicate via the **/turtle1/cmd_vel** topic.
Let's see what happens there:

Open an other terminal (CTLR+SHIFT+TAB) and type:

```
rostopic list
```

you will see all the available topics including the master:
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose

Let's get more detail on topic **/turtle1/cmd_vel** :
Open an other terminal (CTLR+SHIFT+TAB) and type:

```
rostopic info /turtle1/cmd_vel
```

Type: geometry_msgs/Twist
Publishers:
* /teleop_turtle (http://ubuntu:40946/)

Subscribers:
* /turtlesim (http://ubuntu:33270/)

So now we know that **teleop_turtle** is publishing on it and **turtlesim** is subscribed on it.
It also seems like the message that is sent over the topic is of type: *geometry_msgs/Twist*
But what is the exact message that is sent over this topic?

Let's see:

```
rosmsg show geometry_msgs/Twist
```

geometry_msgs/Vector3 linear
float64 x
float64 y
float64 z
geometry_msgs/Vector3 angular
float64 x
float64 y
float64 z

So this is a message that contains two 3dvectors (one for linear velocity and one for angular velocity).
Let's try to replicate the message:

```
rostopic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist
'[2, 0, 0]' '[0, 0, 0]'
```

Mini exercise: Publish a velocity command, that makes the turtle move on a circle.

Now as we can see we are sending commands through the topic and the **turtlesim** node is receiving them.
Great! lets subscribe to it as well so that we can listen to our own song!

```
rostopic echo /turtle1/cmd_vel
```

By pressing the arrow keys on the keyboard (having the terminal with the teleop_key selected) you can move the turtle around, so that we can listen to the signal. Press the up arrow once.

linear:
x: 2.0
y: 0.0
z: 0.0
angular:
x: 0.0
y: 0.0
z: 0.0

Mini exercise: Which color code is the turtle perceiving?

Congratulations!

Now you understand how topics, messages and nodes work!

Presentation
Web Page

Exercise 2.2 - Create a ROS package

Please note that:

> We define notes with a red vertical line,

> terminal commands with a black one,

> and code with a blue one

We need to create an initial package to work with in our tutorial.

> Based on the lecture (and the cited tutorials) you should know about how nodes need to be below packages, etc..

Navigate in your catkin workspace and create a new package called "hello_ros"

```
cd ~/catkin_ws/src
catkin_create_pkg hello_ros geometry_msgs turtlesim
roscpp rospy
cd hello_ros
mkdir scripts
```

Open sublime and create a file called hello.py in the:
~/catkin_ws/src/hello_ros/scripts/hello.py

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String

def hello_ros():
    rospy.init_node('hello_ros', anonymous=True)
    rate = rospy.Rate(10) # 10hz
```

```
    hello_str = "hello ros!!! %s" % rospy.get_time()
    rospy.loginfo(hello_str)
    while not rospy.is_shutdown():
        rate.sleep()

if __name__ == '__main__':
    try:
        hello_ros()
    except rospy.ROSInterruptException:
        pass
```

Now we need allow the python file to be executable (**~/catkin_ws/src/hello_ros/**):

```
cd scripts
chmod +x hello.py
```

Last step is to compile the project like so:

```
cd ~/catkin_ws
catkin build
```

If you like autocompletion to work you might want to type this command in the terminal:

```
rospack profile
```

Let's run the think and see what happens:

In one terminal start the master:

```
roscore
```

In another terminal run:

```
rosrun hello_ros hello.py
```

Mini exercise: Modify the node, so that it prints the message continuously every 0.2 sec with an updated timestamp.

Exercise 2.3 - Writing a publisher

We want to replicate what the keyboard_teleop was doing before.
That means that we want to publish commands to the /turtle1/cmd_vel topic.

Open sublime and create a file called pubvel.py in the:
~/catkin_ws/src/hello_ros/scripts/pubvel.py:

```
#!/usr/bin/env python
# This program publishes randomly-generated velocity
# messages for turtlesim.
import rospy
import numpy as np # For random numbers


from std_msgs.msg import String
from geometry_msgs.msg import Twist #For geometry_msgs/Twist

#Initialize publisher
p = rospy.Publisher('turtle1/cmd_vel', Twist, queue_size=1000)

# Initialize node
rospy.init_node('publish_velocity')
r = rospy.Rate(2) # Set Frequency

#loop until someone shutds us down..
while not rospy.is_shutdown():
    #Initiate Message with zero values
    t=Twist()
    #Fill in message
```

```
t.angular.z = 2*np.random.rand()-1
t.linear.x = np.random.rand()

#ROS INFO of the commands
rospy.loginfo("Ang.z is%s" % t.angular.z)
rospy.loginfo("Lin.x is%s" % t.linear.x)

#publish the message
p.publish(t)
r.sleep()
```

Now we need allow the python file to be executable

(**~/catkin_ws/src/hello_ros/**):

```
cd scripts
chmod +x pubvel.py
```

If you like autocompletion to work you might want to type this command in the terminal:

```
rospack profile
```

Let's run the thing and see what happens:

In one terminal start the master:

```
roscore
```

In another terminal run:

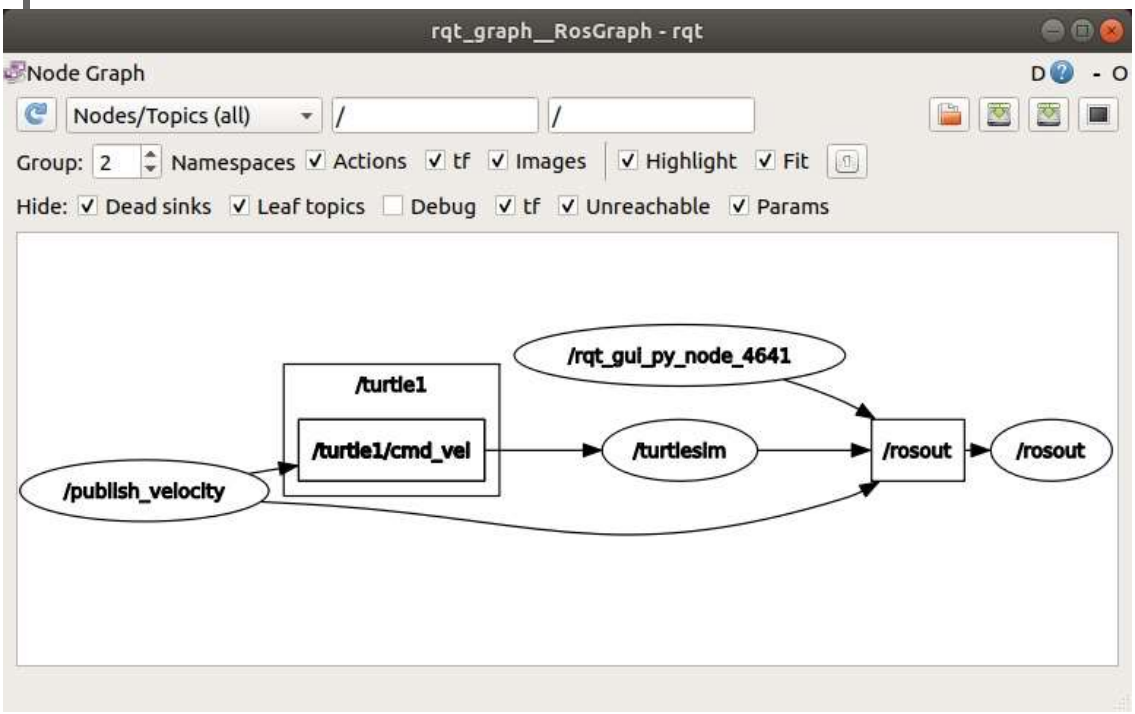
```
rosrun turtlesim turtlesim_node
```

In another terminal run:

```
rosrun hello_ros pubvel.py
```

You should see the turtle moving in random directions. LETS discus what just happened. In other terminal type:

```
rqt_graph
```



Your own node with the name "publish_velocity" publishes messages to the topic "turtle1/cmd_vel", to that again the node "/turtlesim" subscribes to.

Mini exercise: Modify your publisher, so that the turtle moves along a circle.

Exercise 2.4 - Writing a subscriber

Ok now this is getting esoteric... Let's Subscribe to the "/turtle1/Pose" topic to see what we've achieved

Create a file subpose.py and save it to **~/catkin_ws/src/hello_ros/scripts/subpose.py:**

```
#!/usr/bin/env python
# This program publishes randomly-generated velocity
# messages for turtlesim.
```

```
import rospy
import numpy as np # For random numbers


from std_msgs.msg import String
from turtlesim import msg # we need to import hte turtlesim msgs in order
to use them

#Create callback. This is what happens when a new message is received
def sub_cal(msg):
    rospy.loginfo("position=( %f, %f), direction= %f", msg.x, msg.y, msg.theta)

#Initialize publisher
rospy.Subscriber('turtle1/pose', msg.Pose, sub_cal, queue_size=1000)

# Initialize node
rospy.init_node('cmd_vel_listener')
rospy.spin()
rospy.loginfo()
```

Now we need allow the python file to be executable

(**~/catkin_ws/src/hello_ros/**):ros

```
cd scripts
chmod +x subpose.py
```

If you like autocompletion to work you might want to type this command in the terminal:

```
rospack profile
```

Let's run the thing and see what happens:

In one terminal start the master:

```
roscore
```

In another terminal run:

```
rosrun turtlesim turtlesim_node
```

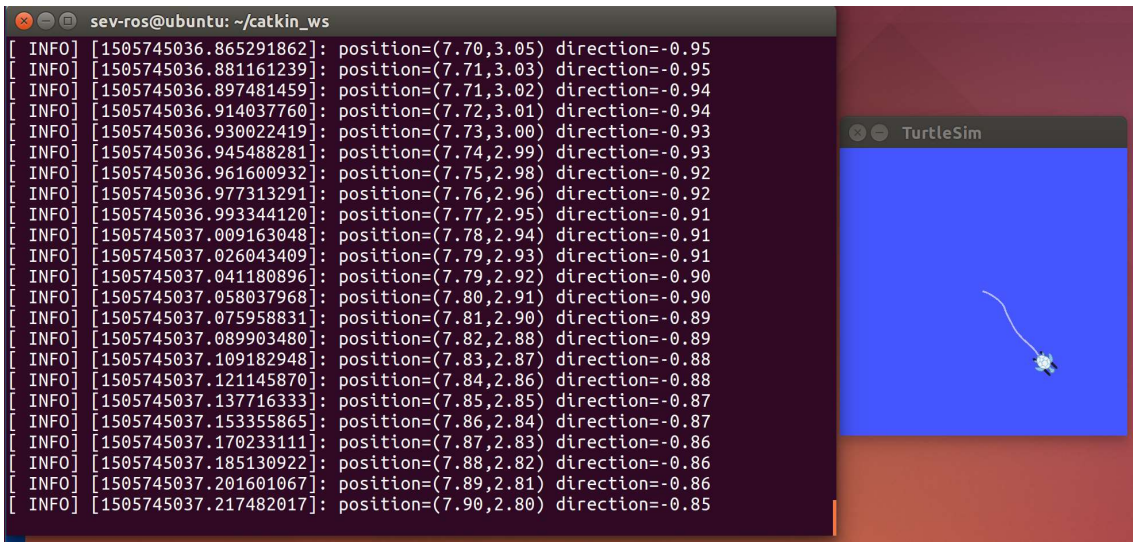
In another terminal run:

```
rosrun hello_ros pubvel.py
```

You should see the turtle moving in random directions as in the previous tutorial.
In a last terminal run the **subpose.py** and see what happens:

```
rosrun hello_ros subpose.py
```

You should see all the messages coming in printed out like this:



Mini exercise: Extend the subscriber, so that it prints the velocity of the turtle.

Exercise 2.5 - Create your own messages

We aim to create our own message and use it to publish some info through a topic

First step is to create the message itself.
Create a file turtle.msg and save it to
~/catkin_ws/src/hello_ros/msg/turtle.msg:
(NOTE: Remember to create the directory **msg** using the command `mkdir`)

```
string name
float32 speed
```

Next we need to let ROS know that there is a new type of message that is defined in our cool package.
To do that we will need to change the two files that are used to "catkin_make" our package:
The CMakeLists.txt and the package.xml

Edit the file **~/catkin_ws/src/hello_ros/package.xml** to contain this extra info

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

Next edit the file **~/catkin_ws/src/hello_ros/CMakeLists.txt** to contain this extra info:

```
find_package(catkin REQUIRED COMPONENTS
  geometry_msgs
  roscpp
  rospy
  turtlesim
  std_msgs
  message_generation

)

add_message_files(
   FILES
   turtle.msg
 )

generate_messages(
   DEPENDENCIES
   std_msgs
   geometry_msgs
 )

catkin_package(
    CATKIN_DEPENDS message_runtime
#  INCLUDE_DIRS include
#  LIBRARIES hello_ros
#  CATKIN_DEPENDS geometry_msgs roscpp rospy turtlesim
#  DEPENDS system_lib
)

include_directories(
# include
  ${catkin_INCLUDE_DIRS}
)
```

OK the message is created. All we have to do is compile the package:

```
roscd
cd ..
catkin build
```

We can see if it's actually there by opening a new terminal and typing the following commands:

```
rosmsg package hello_ros
rosmsg show hello_ros/turtle
```

We can therefore be convinced that our message is correctly created and ROS is aware!

Next, let's use it.
Create a file pubvel_variable.py and save it to
~/catkin_ws/src/hello_ros/scripts/pubvel_variable.py:

```
#!/usr/bin/env python
# This program publishes randomly-generated velocity
# messages for turtlesim.
import rospy
import numpy as np # For random numbers
global speed_var
speed_var= 0.5

from std_msgs.msg import String
from geometry_msgs.msg import Twist #For geometry_msgs/Twist
from hello_ros.msg import turtle #For geometry_msgs/Twist

#Create callback. This is what happens when a new message is received
def sub_cal(msg):
    rospy.loginfo("name= %s, speed= %f", msg.name, msg.speed)
    global speed_var
    speed_var = msg.speed

#Initialize publisher
p = rospy.Publisher('turtle1/cmd_vel', Twist, queue_size=1000)
#Initialise subscriber
rospy.Subscriber('turtle1/turtle_speed', turtle, sub_cal, queue_size=1000)



# Initialize node
```

```
rospy.init_node('publish_variable_velocity')
r = rospy.Rate(2) # Set Frequency

#loop until someone shutds us down..
while not rospy.is_shutdown():

    #Initiate Message with zero values
    t=Twist()
    #Fill in message
    t.angular.z = (2*np.random.rand()-1)*speed_var
    t.linear.x = (np.random.rand())*speed_var

    #ROS INFO of the commands
    #rospy.loginfo("Ang.z is%s" % t.angular.z)
    #rospy.loginfo("Lin.x is%s" % t.linear.x)

    #publish the message
    p.publish(t)
    r.sleep()
```

Now we need allow the python file to be executable

(**~/catkin_ws/src/hello_ros/**):

```
cd scripts
chmod +x pubvel_variable.py
```

If you like autocomplete to work you might want to type this command in the terminal:

```
rospack profile
```

Let's run the thing and see what happens:

In one terminal start the master:

```
roscore
```

In another terminal run:

```
rosrun turtlesim turtlesim_node
```

In another terminal run:

```
rosrun hello_ros pubvel_variable.py
```

You should see the turtle moving in random directions but in **LOW** speed.
Let's try to change this;
by sending a message to the topic we just created:

```
rostopic pub /turtle1/turtle_speed hello_ros/turtle
"name: ''

speed: 10.0"
```

You should see the turtle moving very fast.

Mini exercise: Make the turtle stop using your own message.

Exercise 2.6 - Paramters

In the last tutorial, we changed the execution of a system by passing a message through a topic. However, there exists a much more simple approach concerning the changing of parameters inside nodes.

Let's start 3 terminals and type the following commands:

```
roscore
rosrun turtlesim turtlesim_node
rosparam list
```

We can read the values of parameters and change them like this

```
rosparam get /turtlesim/background_b
rosparam set /turtlesim/background_b 127
```

Due to the **turtlesim_node** inner workings you have to call the clean service for the change to take effect:
rosservice call /clear

Mini exercise: Set the background color to red!

Ok, let's see how we can take benefit of it in our code...

Create a file pubvel_variable_param.py and save it to **~/catkin_ws/src/hello_ros/scripts/pubvel_variable_param.py:**

```
#!/usr/bin/env python
# This program publishes randomly-generated velocity
# messages for turtlesim.
import rospy
import numpy as np # For random numbers
global speed_var
global tmp_speed
speed_var= 1.0
tmp_speed=speed_var
rospy.set_param('speed_of_turtle', speed_var)


from std_msgs.msg import String
from geometry_msgs.msg import Twist #For geometry_msgs/Twist
from hello_ros.msg import turtle #For geometry_msgs/Twist

#Create callback. This is what happens when a new message is received
def sub_cal(msg):
    rospy.loginfo("name= %s, speed= %f", msg.name, msg.speed)
    global speed_var
    speed_var = msg.speed
```

```
    rospy.set_param('speed_of_turtle',speed_var)

#Initialize publisher
p = rospy.Publisher('turtle1/cmd_vel', Twist, queue_size=1000)
#Initialise subscriber
rospy.Subscriber('turtle1/turtle_speed', turtle, sub_cal, queue_size=1000)



# Initialize node
rospy.init_node('publish_variable_velocity')
r = rospy.Rate(2) # Set Frequency

#loop until someone shutds us down..
while not rospy.is_shutdown():

    #Initiate Message with zero values
    t=Twist()
    #Fill in message
    speed_of_turtle = rospy.get_param('speed_of_turtle')
    if speed_of_turtle is not tmp_speed:
        speed_var=speed_of_turtle
        tmp_speed=speed_of_turtle

    t.angular.z = (2*np.random.rand()-1)*speed_var
    t.linear.x = (np.random.rand())*speed_var

    #ROS INFO of the commands
    #rospy.loginfo("Ang.z is%s" % t.angular.z)
    #rospy.loginfo("Lin.x is%s" % t.linear.x)

    #publish the message
    p.publish(t)
    r.sleep()
```

Now we need allow the python file to be executable

(**~/catkin_ws/src/hello_ros/**):

```
cd scripts
chmod +x pubvel_variable_param.py
```

Let's run the thing and see what happens:

In one terminal start the master:

```
roscore
```

In another terminal run:

```
rosrun turtlesim turtlesim_node
```

In another terminal run:

```
rosrun hello_ros pubvel_variable_param.py
```

You should see the turtle moving in random directions but in **LOW** speed.
Let's try to change this;
by sending a message to the topic we just created:

```
rostopic pub --once /turtle1/turtle_speed
hello_ros/turtle "name: '' speed: 10"
```

Now we can also get/set the parameter through the parameter server:

```
rosparam get /speed_of_turtle
rosparam set /speed_of_turtle 100.0
```

You should see the turtle going super fast!!!

Exercise 2.7 - Launch files

Are you tired of all these terminals? Let's replace them with an automated method: **ROSLAUNCH**

The example setup of these familiar terminal commands:

```
rosrun turtlesim turtlesim_node
rosrun turtlesim turtle_teleop_key
rosrun hello_ros subvel.py
```

Can be replaced with the following .launch file:

Create a file **turtlesim_teleop.launch** and save it to **~/catkin_ws/src/hello_ros/launch/turtlesim_teleop.launch:**
(Remember to create the folder **launch** using `mkdir`)

```
<launch>
 <node
  pkg="turtlesim"
  type="turtlesim_node"
  name="turtlesim"
  respawn="true"
 />
 <node
  pkg="turtlesim"
  type="turtle_teleop_key"
  name="teleop_key"
  required="true"
/>
 </launch>
```

Let's run it and see the result:

```
roslaunch hello_ros turtlesim_teleop.launch
```

Mini exercise: Add one of your own nodes to the launch file. E.g. the cmd_vel_listener to print the pose of the turtlebot.

Exercise 2.8 - Services

Ros services offer us (a client node) the ability to directly (or via another node) perform actions on the "server" node.

Let's start with using the turtlesim services

```
roscore
rosrun turtlesim turtlesim_node
rosservice list
```

You can see all the available services. We are interested in the turtlesim ones. Lets call one or two. we have done this before:

```
rosparam get /background_b
rosparam set /background_b 127
rosservice call /clear
```

Lets see something more interesting:

```
rosservice info /turtle1/teleport_relative
rosservice call /turtle1/teleport_relative {"linear:
1.0, angular: 0.0"}
```

Ok, let's see how we can take benefit of it in our code...

Create a file pubvel_client.py and save it to **~/catkin_ws/src/hello_ros/scripts/pubvel_client.py:**

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

import sys
```

```
import rospy
from turtlesim.srv import *

def call_service(x, theta):
    rospy.wait_for_service('/turtle1/teleport_relative')
    try:
        teleport_relative = rospy.ServiceProxy('/turtle1/teleport_relative',
TeleportRelative)
        resp1 = teleport_relative(x, theta)
        return True
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e
        return False

def usage():
    return "%s [x theta]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = float(sys.argv[1])
        theta = float(sys.argv[2])
    else:
        print usage()
        sys.exit(1)
    print "Requesting a relative telepot linear:%s, angular:%s"%(x, theta)
    print "The execution is %s"%call_service(x, theta)
```

Now we need allow the python file to be executable

(**~/catkin_ws/src/hello_ros/**):

```
cd scripts
chmod +x pubvel_client.py
```

Let's run the thing and see what happens:

In one terminal start the master:

```
rosrun hello_ros pubvel_client.py 1.0 0.0
```

You should see the turtle moving. What gives?

Mini exercise: Move the turtle to the bottom left corner of the working area. Hint: Use the service, that teleports the turtle to an absolute position.

Exercise 2.9 - Rosbag

Wouldn't be cool to have instant replay of all these cool things we do in ROS

Ros Bags can be used to record topics and replay them

Let's start with using the same old example

```
roscore
rosrun turtlesim turtlesim_node
rosrun turtlesim turtle_teleop_key
```

Let's record the cmd_vel. In a new terminal:

```
rosbag record -O turtle_commands /turtle1/cmd_vel
```

Now start moving the robot with the keyboard and next press **Ctrl+C** to stop the record

Next lets replay our own data:

```
rosbag play turtle_commands.bag
```

You should see the turtle moving. Why?

Rosbag is very useful during the development of robotic application. Check out the tool here. http://wiki.ros.org/rosbag/Commandline

Mini exercise: Replay your rosbag with double the original speed. Play the rosbag in a loop!

Exercise 2.10 - Putting everything together.

exercise 2.10

Web Page