Goal

On completion of this worksheet you will have a good understanding of how Topics communicate within a Ros system. Using Publishers and Subscribers you will create your own topics for communicating messages across Ros, aswell as creating your own message type.

Introduction to Topics

Topics as we know are the most common way to exchange data within a Ros system .Topics communicate through peer-to-peer publish /subscriber nodes.
Before nodes can transmit information over topics, Publishers must first announce or advertise the topic name and the message type that are going to be published. Once advertised a subscriber, subscribes to the publisher and receives the information at the rate, the publisher broadcasts it at. The publisher doesn't care who receives the information, nor dose it care how many subscribers are receiving its topic.
The following is a simple Publisher that broadcasts "Hello" over a topic called 'message_hello' with type String.
We are now going to look at the implementation of basic code that will advertise a topic and publish a messages on it but first we need to create a workspace and package.

Creating a workspace and Package

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

$ cd ~/HelloWorld_ws/
$ catkin_make
$ source devel/setup.bash

$ cd src/
$ catkin_create_pkg lab_3 std_msgs rospy
$ cd ~/HelloWorld_ws/
$ catkin_make
```

Break down of Publisher code

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

rospy.init_node('publish_hello')

pub = rospy.Publisher('message_hello',String,queue_size=10)
rate= rospy.Rate(2)

Message_to_send = 'Hello'

while not rospy.is_shutdown():
            pub.publish(Message_to_send)
             rate.sleep()
```

The first line : *#!/usr/bin/env python*
Lets the operating system know that this is a python file and passes it into the python interpreter

The second line : *import rospy*
Appears in all Python nodes and imports all standard function of the node

The third line : *from std_msgs.msg import String*
The message we are going to send is of type String so we need to import that type from our Package library std_msgs. We can import other types such as ints32, here if required.

The fourth line : *rospy.init_node('publish_hello')*
Basically means start the node, don't get confused the program and the node are not the same thing , we are using a program to create a node called (publish_hello)

The fifth line: *pub = rospy.Publisher('message_hello',String,queue_size=10)*
We are advertising. This gives the topic name (message_hello) and the topic a type (String). Behind the scenes the publisher sets up a connection to Roscore. At this point the topic is advertised and ready for other nodes to subscribe to it. The rest of the code is sending the message over the topic.

```
rate= rospy.Rate(2)
Message_to_send = 'Hello'
while not rospy.is_shutdown():
            pub.publish(Message_to_send)
             rate.sleep()
```

First we set the rate at which we want to publish (2) measured in hertz, we define a string called Message_to_send with value 'Hello' .
A while loop is created to continually send the message at 2Hz for as long as the node in active (not rospy.is_shutdown). We then publish the message, then sleep.

Running the Publisher

To run the program, save the code as Publisher_Hello.py within a workspace called Hello_World_ws.

to run the code :

Open new Terminal T1

```
$ roscore
```

Open new Terminal T2

```
$ chmod +x Publisher_Hello.py
$ ./Publisher_Hello.py
```

Ros gives us tool to check if our code is working correctly, rostopic list will display all active topics being broad casted across roscore. Since the name of our topic is message_hello we will check if our topic appears we can then check if it is broadcasting the correct type.

With the node running we can then use rostopic, to review all active nodes ,aswell as gathering information on them such as the nodes name, the name of the topic aswell as the message type.

Open new Terminal T3

```
$ rostopic list
$ rostopic info message_hello
```

The output to RosTopic List and Info should look something like the following

-
  ```
  u170457@HWL-26:~/Hello_World_ws$ rostopic list
  /message_hello
  /rosout
  /rosout_agg
  ```

-
  ```
  u170457@HWL-26:~/Hello_World_ws$ rostopic info message_hello
  Type: std_msgs/String

  Publishers:
  * /publish_hello (http://HWL-26:34154/)

  Subscribers: None
  ```

Break down of Subscriber code

Now we are going to take a look at simple Subscriber that will receive what our publisher is sending and print the results.

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

rospy.init_node('subscriber')

def callback(message_recived):
        print 'I heard ' + message_recived.data

sub = rospy.Subscriber('message_hello',String,callback)

rospy.spin()
```

The first four lines : Are exactly the same as the publisher, we are letting the operating system know that this is a python file, importing all standard functions and message type and we are initializing the node and calling it *('subscriber')*

Things get interesting in the fifth line : *def callback(message_recived):*
print 'I heard ' + message_recived.data

The callback is the method that dose something to the messages as they come in. Since Ros is event driven the callback must be declared before the actual subscription. Once a node is subscribed to a topic every time the message appears the associated callback is called, with the message as its parameter. In this case every time a message is received the subscriber will simply print it.

In the sixth Line : *sub = rospy.Subscriber('message_hello',String,callback)*
We subscribe to the topic. We give the name of the topic, the message type of the topic, and the name of the callback function. The subscriber passes this information on to roscore and tries to make a direct connection with the publishers of this topic.

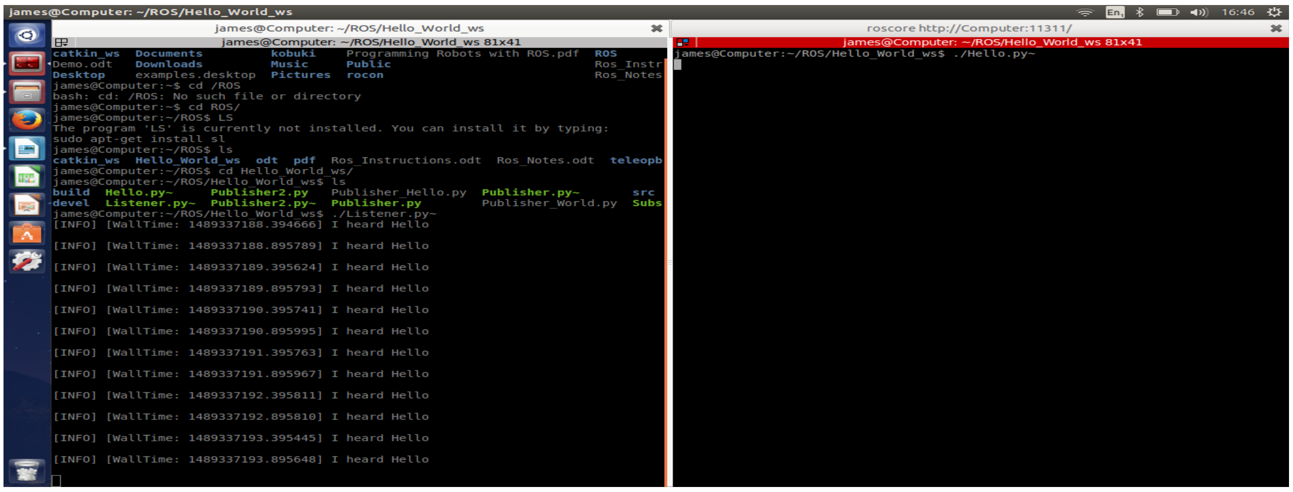
Running the Subscriber

Save the code as Subscriber.py and make sure roscore and Publisher_Hello.py are already running and :

<u>Open new Terminal T4</u>

```
$ chmod +x Subscriber.py
$ ./Subscriper.py
```

With the two nodes running we get a the following results:



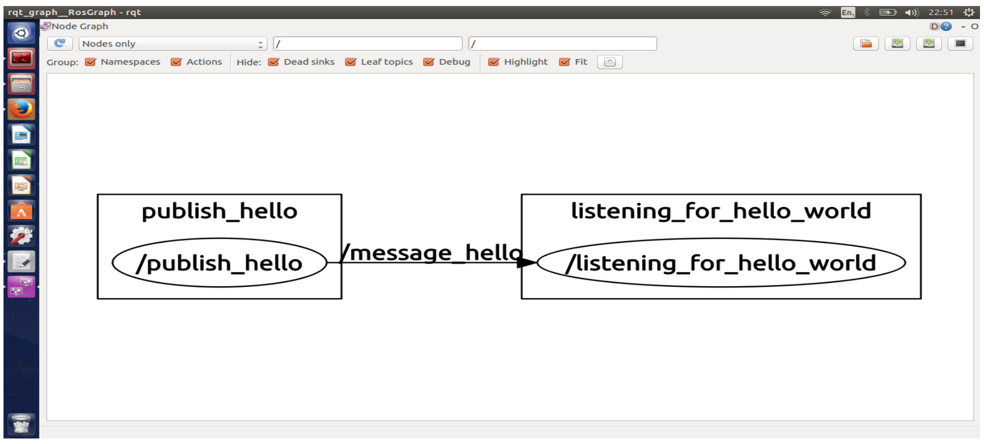
```
james@Computer: ~/ROS/Hello_World_ws 81x41
catkin_ws   Documents         kobuki    Programming Robots with ROS.pdf  ROS
Demo.odt    Downloads         Music     Public                           Ros_Instr
Desktop     examples.desktop  Pictures  rocon                            Ros_Notes
james@Computer:~$ cd /ROS
bash: cd: /ROS: No such file or directory
james@Computer:~$ cd ROS/
james@Computer:~/ROS$ LS
The program 'LS' is currently not installed. You can install it by typing:
sudo apt-get install sl
james@Computer:~/ROS$ ls
catkin_ws  Hello_World_ws  odt  pdf  Ros_Instructions.odt  Ros_Notes.odt  teleopb
james@Computer:~/ROS$ cd Hello_World_ws/
james@Computer:~/ROS/Hello_World_ws$ ls
build  Hello.py~     Publisher2.py   Publisher_Hello.py  Publisher.py~       src
devel  Listener.py~  Publisher2.py~  Publisher.py        Publisher_World.py  Subs
james@Computer:~/ROS/Hello_World_ws$ ./Listener.py~
[INFO] [WallTime: 1489337188.394666] I heard Hello

[INFO] [WallTime: 1489337188.895789] I heard Hello

[INFO] [WallTime: 1489337189.395624] I heard Hello

[INFO] [WallTime: 1489337189.895793] I heard Hello

[INFO] [WallTime: 1489337190.395741] I heard Hello

[INFO] [WallTime: 1489337190.895995] I heard Hello

[INFO] [WallTime: 1489337191.395763] I heard Hello

[INFO] [WallTime: 1489337191.895967] I heard Hello

[INFO] [WallTime: 1489337192.395811] I heard Hello

[INFO] [WallTime: 1489337192.895810] I heard Hello

[INFO] [WallTime: 1489337193.395445] I heard Hello

[INFO] [WallTime: 1489337193.895648] I heard Hello
```

```
roscore http://Computer:11311/
james@Computer: ~/ROS/Hello_World_ws 81x41
james@Computer:~/ROS/Hello_World_ws$ ./Hello.py~
```

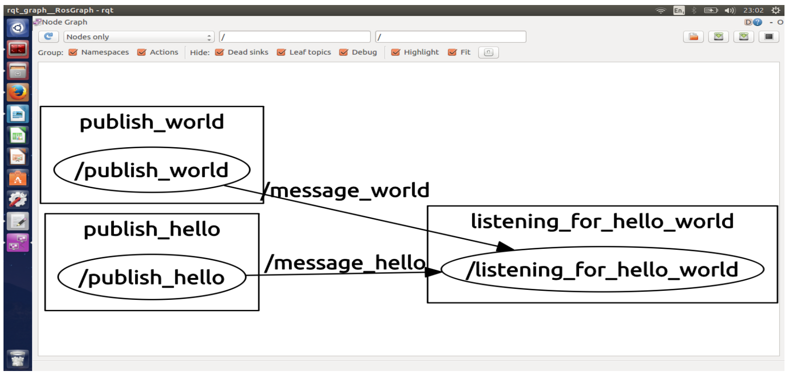
Using Rosrun, we can create a Ros graph of our system. This can help us visualize whats happening, instead of solely depending on what we see in the terminal window.

<u>Open new Terminal T5</u>

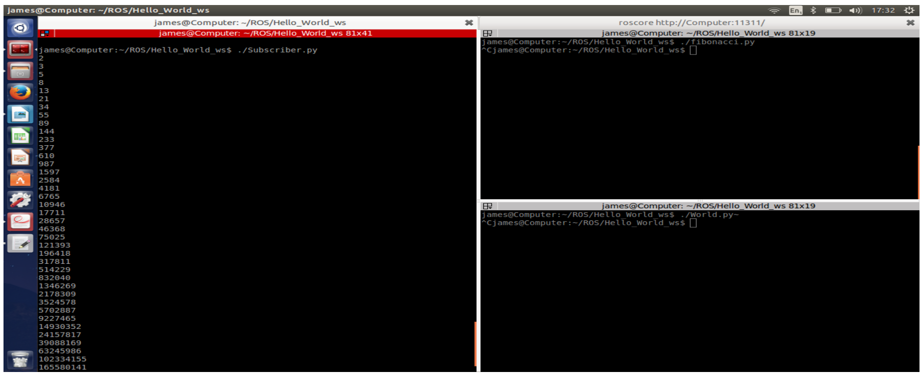
```
$ rosrun rqt_graph rqt_graph
```



Exercise 1 : Create a new publisher called Publisher_World.py which broadcasts a Topic called “message_world” of Type String with a message “World” , alter Subscriber.py so it is subscribed to Publisher_World.py 's node as well as Publisher_Hello.py 's node and prints out both messages . Your Ros Graph should like like the following:


rqt_graph__RosGraph - rqt
Node Graph
Nodes only
Group: Namespaces Actions Hide: Dead sinks Leaf topics Debug Highlight Fit
publish_world
/publish_world
/message_world
publish_hello
/publish_hello
/message_hello
listening_for_hello_world
/listening_for_hello_world

Exercise 2 : Using *“from std_msgs.msg import Int32”* instead of *“from std_msgs.msg import String”* create a node that publishes a sequence of Fibonacci numbers starting at 2. Then either alter or create a new subscriber that listens for the new nodes message, and prints them. Your result should look like the following:

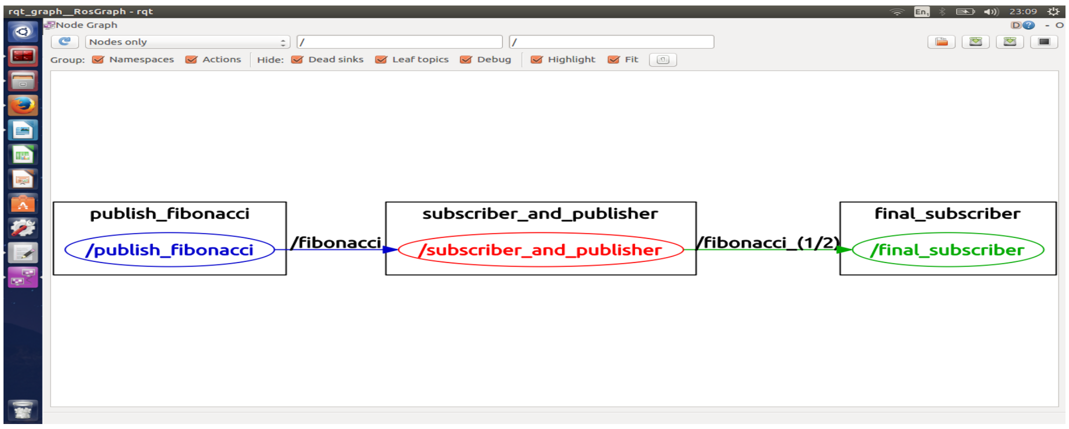


```
james@Computer:~/ROS/Hello_World_ws 81x41
james@Computer:~/ROS/Hello_World_ws$ ./Subscriber.py
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
28657
46368
75025
121393
196418
317811
514229
832040
1346269
2178309
3524578
5702887
9227465
14930352
24157817
39088169
63245986
102334155
165580141
```

```
roscore http://Computer:11311/
james@Computer: ~/ROS/Hello_World_ws 81x19
james@Computer:~/ROS/Hello_World_ws$ ./fibonacci.py
^Cjames@Computer:~/ROS/Hello_World_ws$ 
```

```
james@Computer: ~/ROS/Hello_World_ws 81x19
james@Computer:~/ROS/Hello_World_ws$ ./World.py~
^Cjames@Computer:~/ROS/Hello_World_ws$ 
```

Exercise 3 : Alter the subscriber node from Exercise 2 , so that it takes the Fibonacci values it receives, divides them by 2, then publishes them as a new topic of type Int32. Create a new node that subscribes and prints out the new values. Your result should look like the following:



Messages

We have already looked at sending messages of type int32 and type String which are found in the std_msgs package, which we import at the start of our code.
Now that we have a good feel for how to send messages over topics, we are now going to look at how to send a message that isn't already defined by the Ros std_msgs package

This std_msgs package defines the following primitive types :

ROS TYPE	SERIALIZATION	PYTHON TYPE
Bool	Unsigned 8-bit integer	Bool
int8	Signed 8-bit integer	int
uint8	Unsigned 8-bit integer	int
int16	Signed 16-bit integer	int
uint16	Unsigned 16-bit integer	int
int32	Signed 32-bit integer	int
uint32	Unsigned 32-bit integer	int
int64	Signed 64-bit integer	long
uint64	Unsigned 64-bit integer	long
float32	32-bit IEEE float	float
float64	64-bit IEEE float	float
String	ASCII string	String
time	Secs/nsecs unsigened 32-bit int	rospy.Time

Making a message

There are many times when these message types found in std_msgs arnt enough and we have to define our own. Ros messages are defined by message_deffinition files in the msg directory of a package which we will need to create.

```
$ roscd lab_3
$ mkdir msg
$ cd msg
$ gedit Complex.msg
```

In this new folder create a file called Complex.msg

```
float32 real
float32 imaginary
```

```
$ cd ~/HelloWorld/
$ catkin_make
```

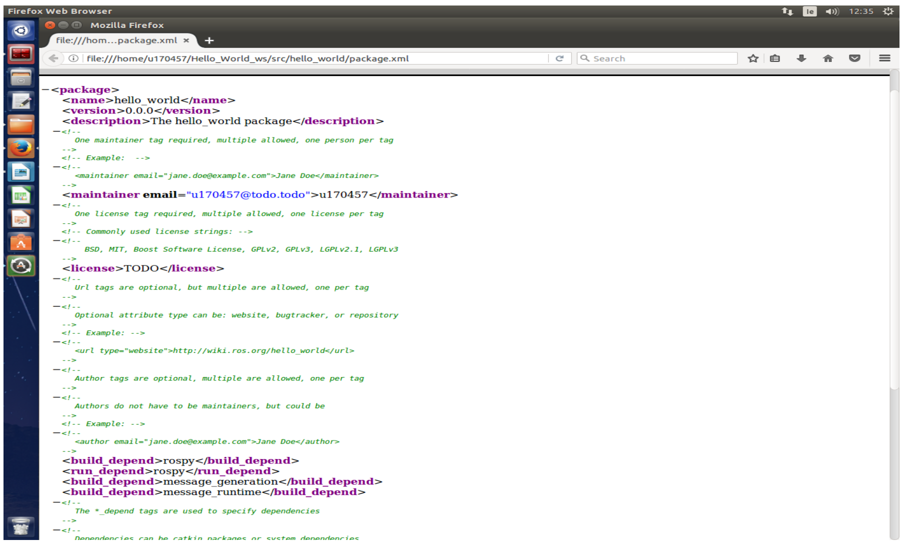
What this will do is create a message file for a new message type that we are calling Complex. It defines two values, one real and one imaginary.

Now for the tricky bit, in order to get to generate the new message type we need to get Ros to generate the message creation code. We have to add the message_creation dependency to our package.xml file. This file was generated when we created our package *$ catkin_create_pkg lab_3 std_msgs rospy* .

The file is located in : /HelloWorld_ws/src/lab_3/package.xml

add the following to the file

<build_depend>message_generation**</build_depend>**
<run_depend>message_runtime**</run_depend>**



Firefox Web Browser
12:35
Mozilla Firefox
file:///hom...package.xml
file:///home/u170457/Hello_World_ws/src/hello_world/package.xml
Search

```
-<package>
   <name>hello_world</name>
   <version>0.0.0</version>
   <description>The hello_world package</description>
 -<!--
     One maintainer tag required, multiple allowed, one person per tag
   -->
   <!-- Example:  -->
 -<!--
     <maintainer email="jane.doe@example.com">Jane Doe</maintainer>
   -->
   <maintainer email="u170457@todo.todo">u170457</maintainer>
 -<!--
     One license tag required, multiple allowed, one license per tag
   -->
   <!-- Commonly used license strings: -->
 -<!--
       BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3
   -->
   <license>TODO</license>
 -<!--
     Url tags are optional, but multiple are allowed, one per tag
   -->
 -<!--
     Optional attribute type can be: website, bugtracker, or repository
   -->
   <!-- Example: -->
 -<!--
     <url type="website">http://wiki.ros.org/hello_world</url>
   -->
 -<!--
     Author tags are optional, multiple are allowed, one per tag
   -->
 -<!--
     Authors do not have to be maintainers, but could be
   -->
   <!-- Example: -->
 -<!--
     <author email="jane.doe@example.com">Jane Doe</author>
   -->
   <build_depend>rospy</build_depend>
   <run_depend>rospy</run_depend>
   <build_depend>message_generation</build_depend>
   <build_depend>message_runtime</build_depend>
 -<!--
     The *_depend tags are used to specify dependencies
   -->
 -<!--
     Dependencies can be catkin packages or system dependencies
```

Now we are going to open CmakeList.txt file, found beside the package.xml.The file requires a few changes so that when we run catkin_make to compile, it knows where to look for the message_generation package. The file has tonnes of comments and is split into different sections. In the first section in the Find_Pakage() method uncomment it and add message_generation so it looks like:

```
find_package(catkin REQUIRED COMPONENTS
rospy
std_msgs
message_generation)
```

In the second section (*Declare Ros messages,services and actions)* uncomment the add_message_files() method and add the name of our file Complex.msg, it should look like

```
add_message_files(
FILES
Complex.msg )
```

Then in the generate_message() method alter it so it looks like

```
generate_messages(
DEPENDENCIES
std_msgs )
```

In the fifth section (*catkin specific configuration)* uncomment the catkin_package method and add CATKIN_DEPENDS message_runtime , it should look like

```
catkin_package(
 CATKIN_DEPENDS message_runtime)
```

Now run catkin_make to complie the message in your workspace root:

```
$ cd ~/HelloWorld/
$ catkin_make
```

A good indicator that everything worked is that catkin_make has a lot of out puts as it complies all the new packages and messages we added. Now we can use the message just like any other message in Ros .
To check if the message complied correctly type:

```
$ rosmsg show Complex
```

Running a message

The following is the code for a publisher and subscriber that will send and receive messages of our new type, Complex.

Publisher

```
#!/usr/bin/env python
import rospy
from lab_3.msg import Complex

rospy.init_node("message_Publisher")
pub = rospy.Publisher('complex_message',Complex,queue_size = 5)
rate = rospy.Rate(2)

while not rospy.is_shutdown():
	msg=Complex()
	msg.real= 6
	msg.imaginary= 9
	pub.publish(msg)
	rate.sleep()
```

Subscriber

```
#!/usr/bin/env python
import rospy
from lab_3.msg import Complex

def callback(msg):
	print 'Real : ',msg.real
	print 'Imaginary : ',msg.imaginary

rospy.init_node('complex_sub')
sub = rospy.Subscriber('complex_message',Complex,callback)
rospy.spin()
```

Make sure you:

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

In each terminal before running the code other wise it will not find your message.

Exercise four : Create a new message Type called Date which consists of a non zero int32 called year, and two Strings called day and month. To do this you will need to create a file called Date.msg in the msg folder aswell as add its dependency to CmakeList.txt. Alter exsisting code to make a publisher subsrciber that will print out todays date.