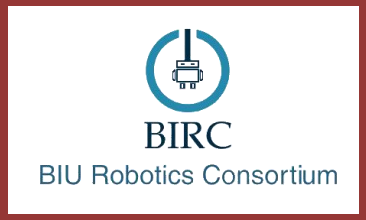
October 2016



ROS – Lecture 3

ROS topics
Publishers and subscribers
roslaunch
Custom message types

Lecturer: Roi Yehoshua
roiyeho@gmail.com

ROS Communication Types

Type	Best used for
Topic	One-way communication, especially if there might be multiple nodes listening (e.g., streams of sensor data)
Service	Simple request/response interactions, such as asking a question about a node's current state
Action	Most request/response interactions, especially when servicing the request is not instantaneous (e.g., navigating to a goal location)

(C)2016 Roi Yehoshua

ROS Topics

- Topics implement a *publish/subscribe* communication mechanism
  - one of the more common ways to exchange data in a distributed system.
- Before nodes start to transmit data over topics, they must first announce, or *advertise,* both the topic name and the types of messages that are going to be sent
- Then they can start to send, or *publish,* the actual data on the topic.
- Nodes that want to receive messages on a topic can *subscribe* to that topic by making a request to roscore.
- After subscribing, all messages on the topic are delivered to the node that made the request.

(C)2016 Roi Yehoshua

ROS Topics

- In ROS, all messages on the same topic *must be of the same data type*
- Topic names often describe the messages that are sent over them
- For example, on the PR2 robot, the topic /wide_stereo/right/image_color is used for color images from the rightmost camera of the wide-angle stereo pair

(C)2016 Roi Yehoshua

Topic Publisher

- Manages an advertisement on a specific topic
- Created by calling **NodeHandle::advertise()**
  - Registers this topic in the master node
- Example for creating a publisher:

```
ros::Publisher chatter_pub = node.advertise<std_msgs::String>("chatter", 1000);
```

  - First parameter is the topic name
  - Second parameter is the queue size
- Once all the publishers for a given topic go out of scope the topic will be unadvertised

(C)2016 Roi Yehoshua

Topic Publisher

- To publish a message on a topic call **publish()**
- Example:

```
std_msgs::String msg;
chatter_pub.publish(msg);
```

  - The message's type must agree with the type given as a template parameter to the advertise<>() call

(C)2016 Roi Yehoshua

Talker and Listener

- We'll now create a package with two nodes:
  - *talker* publishes messages to topic "chatter"
  - *listener* reads the messages from the topic and prints them out to the screen
- First create the package chat_pkg

```
$ cd ~/catkin_ws/src
catkin_create_pkg chat_pkg std_msgs rospy roscpp
```

- Open the package source directory in Eclipse and add a C++ source file named Talker.cpp
- Copy the following code into it

(C)2016 Roi Yehoshua

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
```

(C)2016 Roi Yehoshua

Subscribing to a Topic

- To start listening to a topic, create a **Subscriber** object
- Example for creating a subscriber:

```
rospy.Subscriber("chatter", String, callback)
```

  - 1st parameter is the topic name
  - 2nd parameter is the message type
  - 3rd parameter is the callback function

(C)2016 Roi Yehoshua

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     # 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)
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()
```

(C)2016 Roi Yehoshua

rospy.spin()

- **rospy.spin()** gives control over to ROS, which allows it to call all user callbacks
  - will not return until the node has been shutdown, either through a call to rospy.shutdown() or a Ctrl-C.

(C)2016 Roi Yehoshua

Building the Nodes

- Now build the package and compile all the nodes using the catkin_make tool:

```
cd ~/catkin_ws
catkin_make
```

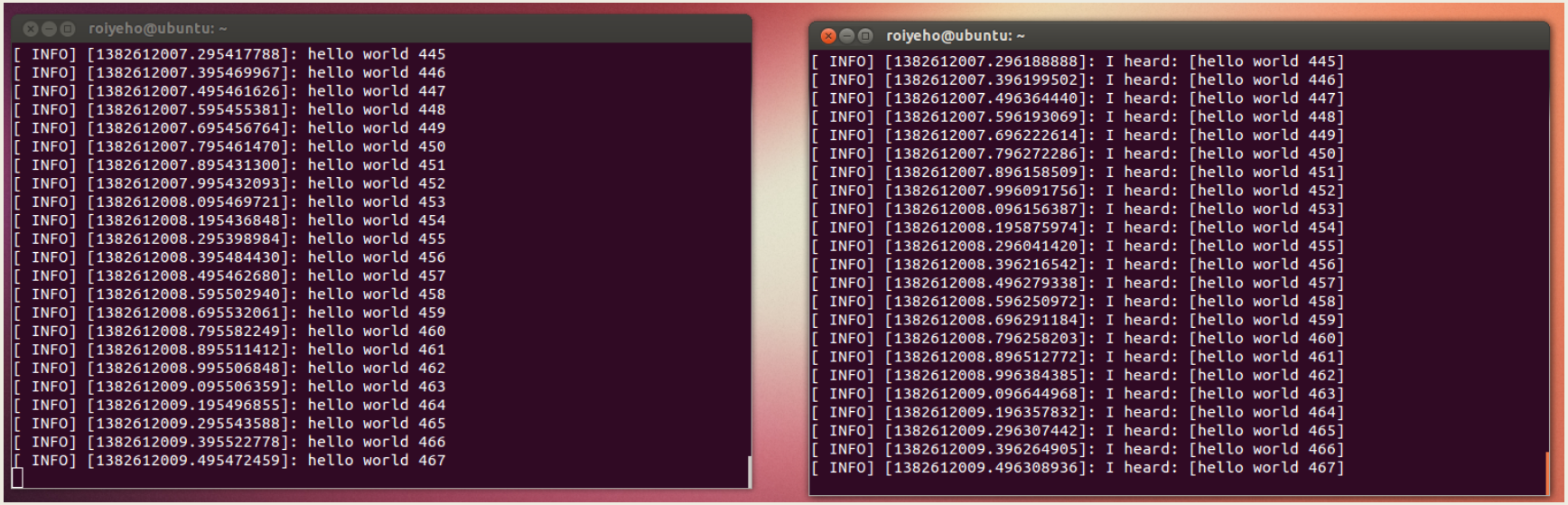
- This will create two executables, talker and listener, at ~/catkin_ws/devel/lib/chat_pkg

(C)2016 Roi Yehoshua

Running the Nodes From Terminal

- Run roscore
- Run the nodes in two different terminals:

```
$ rosrun chat_pkg talker.py
$ rosrun chat_pkg listener.py
```



(C)2016 Roi Yehoshua

Running the Nodes From Terminal

- You can use rosnode and rostopic to debug and see what the nodes are doing
- Examples:

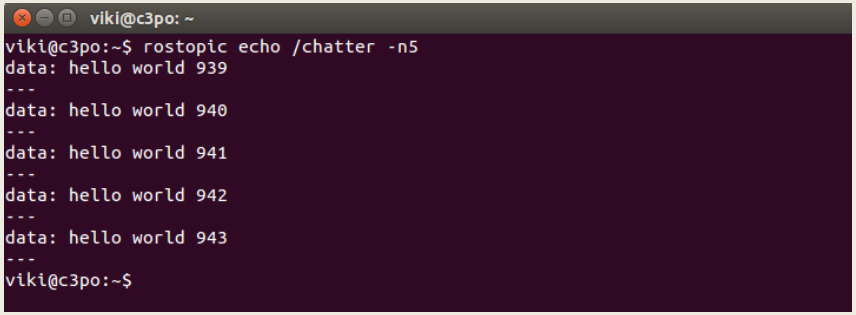
  $rosnode info /talker

  $rosnode info /listener

  $rostopic list

  $rostopic info /chatter

  $rostopic echo /chatter

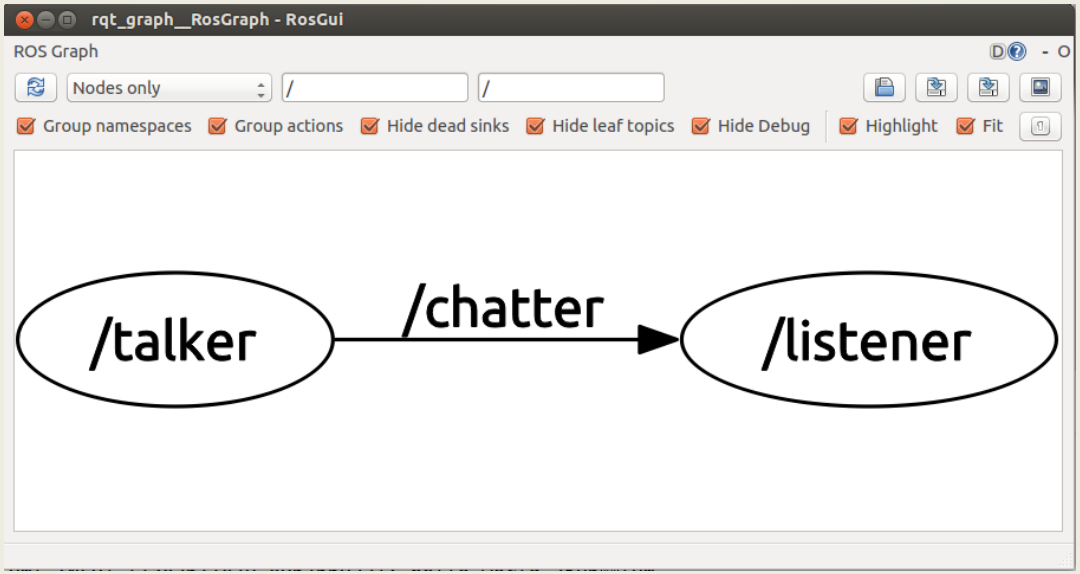


(C)2016 Roi Yehoshua

rqt_graph

- rqt_graph creates a dynamic graph of what's going on in the system
- Use the following command to run it:

```
$ rosrun rqt_graph rqt_graph
```



(C)2016 Roi Yehoshua

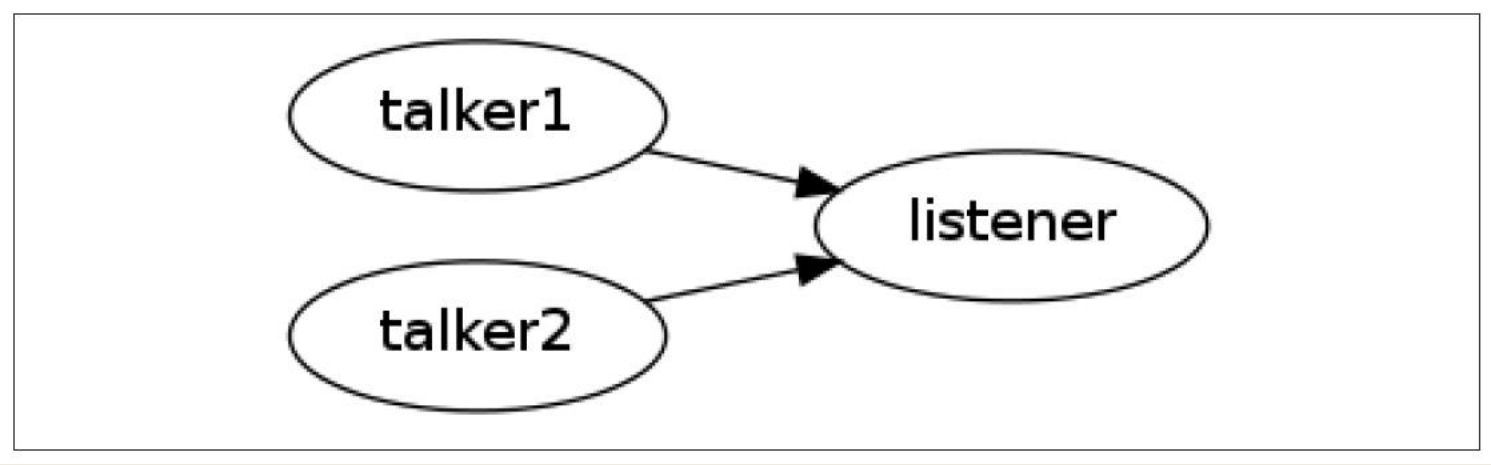
ROS Names

- ROS names must be unique
- If the same node is launched twice, roscore directs the older node to exit
- To change the name of a node on the command line, the special **__name** remapping syntax can be used
- The following two shell commands would launch two instances of talker named talker1 and talker2

```
$ rosrun chat_pkg talker __name:=talker1
$ rosrun chat_pkg talker __name:=talker2
```

(C)2016 Roi Yehoshua

ROS Names



Instantiating two talker programs and routing them to the same receiver

(C)2016 Roi Yehoshua

roslaunch

- a tool for easily launching multiple ROS nodes as well as setting parameters on the Parameter Server
- roslaunch operates on launch files which are XML files that specify a collection of nodes to launch along with their parameters
  - By convention these files have a suffix of .launch
- Syntax:

```
$ roslaunch PACKAGE LAUNCH_FILE
```

- roslaunch automatically runs roscore for you

(C)2016 Roi Yehoshua

Launch File Example

- Launch file for launching the talker and listener nodes:

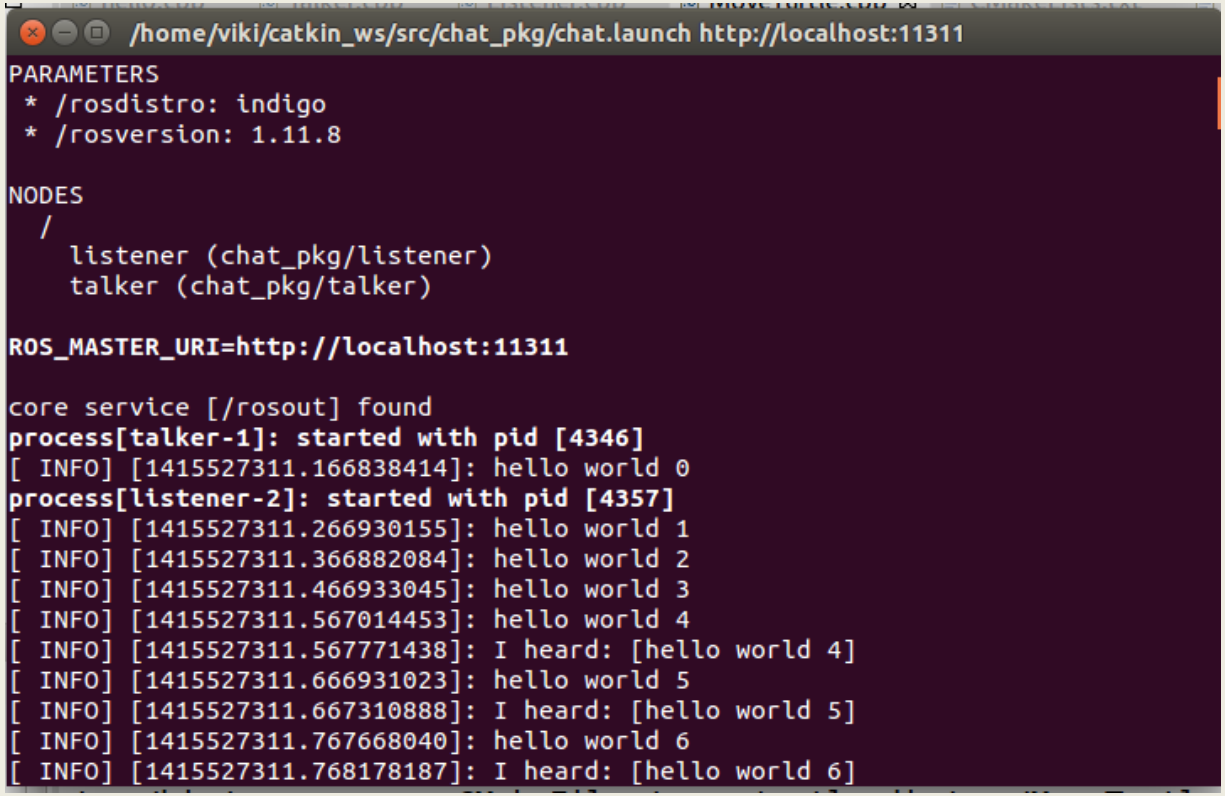
```
<launch>
  <node name="talker" pkg="chat_pkg" type="talker" output="screen"/>
  <node name="listener" pkg="chat_pkg" type="listener"
output="screen"/>
</launch>
```

- Each <node> tag includes attributes declaring the ROS graph name of the node, the package in which it can be found, and the type of node, which is the filename of the executable program
- output="screen" makes the ROS log messages appear on the launch terminal window

(C)2016 Roi Yehoshua

Launch File Example

```
$ roslaunch chat_pkg chat.launch
```



(C)2016 Roi Yehoshua

Velocity Commands

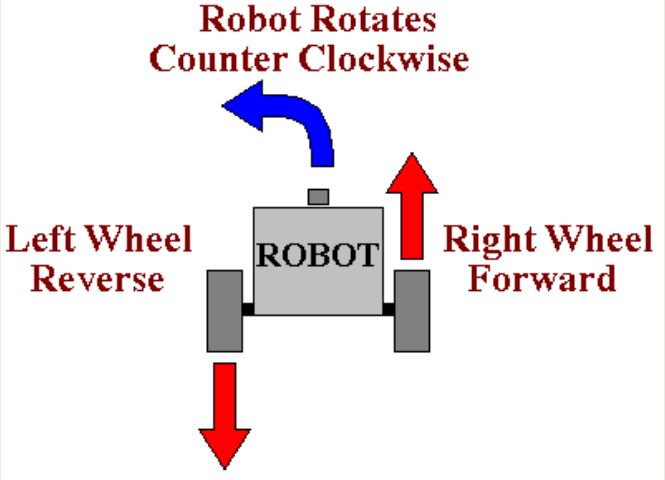
- To make a robot move in ROS we need to publish **Twist** messages to the topic **cmd_vel**
- This message has a linear component for the (x,y,z) velocities, and an angular component for the angular rate about the (x,y,z) axes

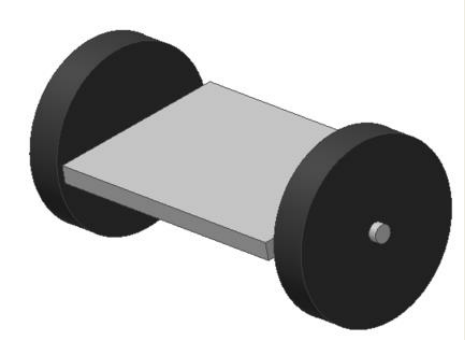
```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

(C)2016 Roi Yehoshua

Differential Drive Robots

- A differential wheeled robot consists of two independently actuated wheels, located on its two sides
- The robot moves forward when both wheels turn forward, and spins in place when one wheel drives forward and one drives backward.





(C)2016 Roi Yehoshua

Differential Drive Robots

- A differential drive robot can only move forward/backward along its longitudinal axis and rotate only around its vertical axis
  - The robot cannot move sideways or vertically
- Thus we only need to set the linear **x** component and the angular **z** component in the Twist message
- An omni-directional robot would also use the linear y component while a flying or underwater robot would use all six components

(C)2016 Roi Yehoshua

A Move Turtle Node

- For the demo, we will create a new ROS package called my_turtle

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg my_turtle std_msgs rospy roscpp
```

- In Eclipse add a new source file to the package called Move_Turtle.cpp
- Add the following code

(C)2016 Roi Yehoshua

Launch File

- Add move_turtle.launch to your package:

```
<launch>
  <node name="turtlesim_node" pkg="turtlesim" type="turtlesim_node" />
  <node name="move_turtle" pkg="my_turtle" type="move_turtle"
output="screen" />
</launch>
```

- Run the launch file:

```
$ roslaunch my_turtle move_turtle.launch
```

(C)2016 Roi Yehoshua

Creating Custom Messages

- ROS offers a rich set of built-in message types
- The std_msgs package defines the primitive types:

Built-in types:

Primitive Type	Serialization	C++	Python
bool (1)	unsigned 8-bit int	`uint8_t` (2)	`bool`
int8	signed 8-bit int	`int8_t`	`int`
uint8	unsigned 8-bit int	`uint8_t`	`int` (3)
int16	signed 16-bit int	`int16_t`	`int`
uint16	unsigned 16-bit int	`uint16_t`	`int`
int32	signed 32-bit int	`int32_t`	`int`
uint32	unsigned 32-bit int	`uint32_t`	`int`
int64	signed 64-bit int	`int64_t`	`long`
uint64	unsigned 64-bit int	`uint64_t`	`long`
float32	32-bit IEEE float	`float`	`float`
float64	64-bit IEEE float	`double`	`float`
string	ascii string (4)	`std::string`	`string`
time	secs/nsecs signed 32-bit ints	ros::Time	rospy.Time
duration	secs/nsecs signed 32-bit ints	ros::Duration	rospy.Duration

(C)2016 Roi Yehoshua

Creating Custom Messages

- These primitive types are used to build all of the messages used in ROS
- For example, (most) laser range-finder sensors publish sensor_msgs/LaserScan messages

```
# Single scan from a planar laser range-finder

Header header
# stamp: The acquisition time of the first ray in the scan.
# frame_id: The laser is assumed to spin around the positive Z axis
# (counterclockwise, if Z is up) with the zero angle forward along the x axis

float32 angle_min # start angle of the scan [rad]
float32 angle_max # end angle of the scan [rad]
float32 angle_increment # angular distance between measurements [rad]

float32 time_increment # time between measurements [seconds] - if your scanner
# is moving, this will be used in interpolating position of 3d points
float32 scan_time # time between scans [seconds]

float32 range_min # minimum range value [m]
float32 range_max # maximum range value [m]

float32[] ranges # range data [m] (Note: values < range_min or > range_max should be
discarded)
float32[] intensities # intensity data [device-specific units]. If your
# device does not provide intensities, please leave the array empty.
```

(C)2016 Roi Yehoshua

Creating Custom Messages

- Using standardized message types for laser scans and location estimates enables nodes can be written that provide navigation and mapping (among many other things) for a wide variety of robots
- However, there are times when the built-in message types are not enough, and we have to define our own messages

(C)2016 Roi Yehoshua

msg Files

- ROS messages are defined by special message-definition files in the *msg* directory of a package.
- These files are then compiled into language-specific implementations that can be used in your code
- Each line in the file specifies a type and a field name

(C)2016 Roi Yehoshua

Message Field Types

Field types can be:

- a built-in type, such as "float32 pan" or "string name"
- names of Message descriptions defined on their own, such as "geometry_msgs/PoseStamped"
- fixed- or variable-length arrays (lists) of the above, such as "float32[] ranges" or "Point32[10] points“
- the special Header type, which maps to std_msgs/Header

(C)2016 Roi Yehoshua

Creating Custom Messages

- As an example we will create a new ROS message type that each robot will publish when it is ready to perform some task
- Message type will be called **RobotStatus**
- The structure of the message will be:

```
Header header
int32 robot_id
bool is_ready
```

- The header contains a timestamp and coordinate frame information that are commonly used in ROS

(C)2016 Roi Yehoshua

Message Header

```
#Standard metadata for higher-level flow data types
#sequence ID: consecutively increasing ID
uint32 seq
#Two-integer timestamp that is expressed as:
# * stamp.secs: seconds (stamp_secs) since epoch
# * stamp.nsecs: nanoseconds since stamp_secs
# time-handling sugar is provided by the client library
time stamp
#Frame this data is associated with
string frame_id
```

- stamp specifies the publishing time
- frame_id specifies the point of reference for data contained in that message

(C)2016 Roi Yehoshua

Creating a Message Type

- First create a new package called

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg custom_messages std_msgs rospy roscpp
```

- Now create a subdirectory msg and the file RobotStatus.msg within it

```
$ cd ~/catkin_ws/src/custom_messages
$ mkdir msg
$ gedit msg/RobotStatus.msg
```

(C)2016 Roi Yehoshua

Creating a Message Type

- Add the following lines to RobotStatus.msg:

```
Header header
int32 robot_id
bool is_ready
```

- Now we need to make sure that the msg files are turned into source code for C++, Python, and other languages

(C)2016 Roi Yehoshua

Creating a Message Type

- Open package.xml, and add the following two lines to it

```
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<build_depend>message_generation</build_depend>
<run_depend>roscpp</run_depend>
<run_depend>rospy</run_depend>
<run_depend>std_msgs</run_depend>
<run_depend>message_runtime</run_depend>
```

- Note that at build time, we need "message_generation", while at runtime, we need "message_runtime"

(C)2016 Roi Yehoshua

Creating a Message Type

- In CMakeLists.txt add the message_generation dependency to the find package call so that you can generate messages:

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

- Also make sure you export the message runtime dependency:

```
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES multi_sync
    CATKIN_DEPENDS roscpp rospy std_msgs message_runtime
#  DEPENDS system_lib
)
```

(C)2016 Roi Yehoshua

Creating a Message Type

- Find the following block of code:

```
## Generate messages in the 'msg' folder
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )
```

- Uncomment it by removing the # symbols and then replace the stand in Message*.msg files with your .msg file, such that it looks like this:

```
add_message_files(
  FILES
  RobotStatus.msg
)
```

(C)2016 Roi Yehoshua

Creating a Message Type

- Now we must ensure the generate_messages() function is called
- Uncomment these lines:

```
# generate_messages(
#   DEPENDENCIES
#   std_msgs
# )
```

- So it looks like:

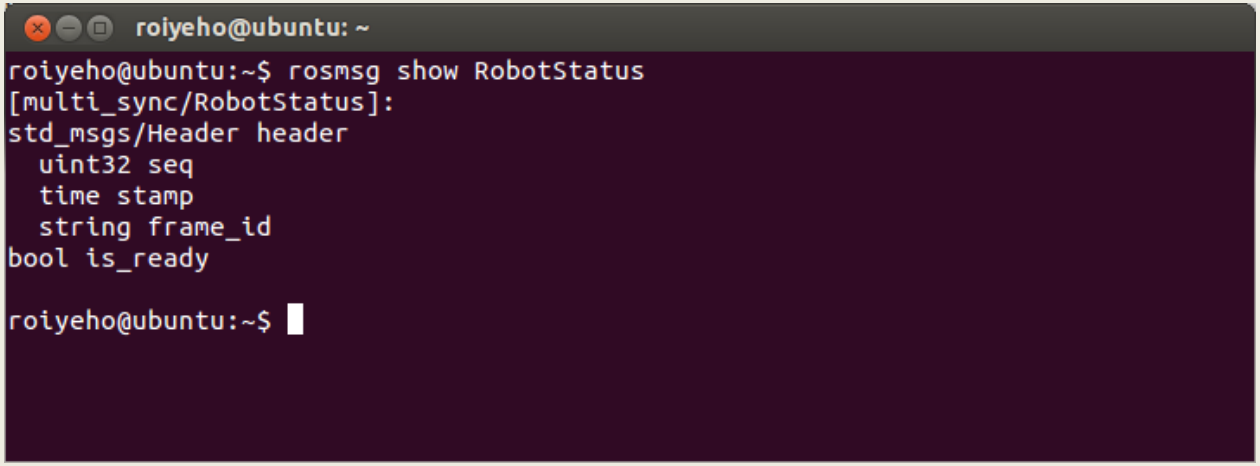
```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

(C)2016 Roi Yehoshua

Using rosmsg

- That's all you need to do to create a msg
- Let's make sure that ROS can see it using the rosmsg show command:

```
$ rosmsg show [message type]
```



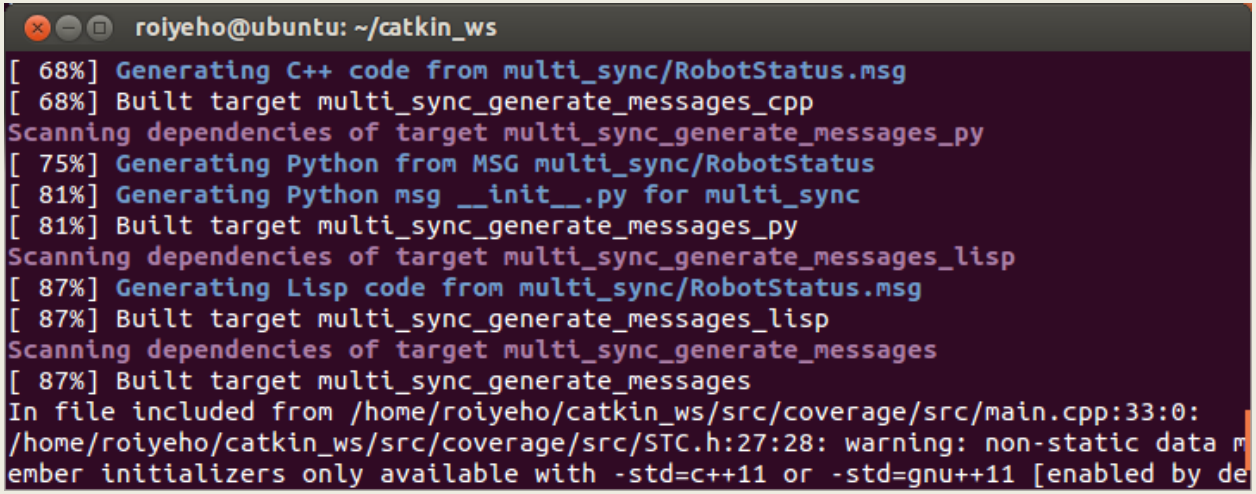
(C)2016 Roi Yehoshua

Building the Message Files

- Now we need to make our package:

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

- During the build, source files will be created from the .msg file:



(C)2016 Roi Yehoshua

Building the Message Files

- Any .msg file in the msg directory will generate code for use in all supported languages.
- The message header file will be generated in ~/catkin_ws/devel/include/custom_messages/

(C)2016 Roi Yehoshua

Importing Messages

- You can now import the RobotStatus header file by adding the following line:

```
from custom_messages import RobotStatus
```

- Note that messages are put into a namespace that matches the name of the package

(C)2016 Roi Yehoshua

Ex. 3

- Write a program that moves the turtle 1m forward from its current position, rotates it 45 degrees and then causes it to stop
- Print the turtle's initial and final locations

(C)2016 Roi Yehoshua