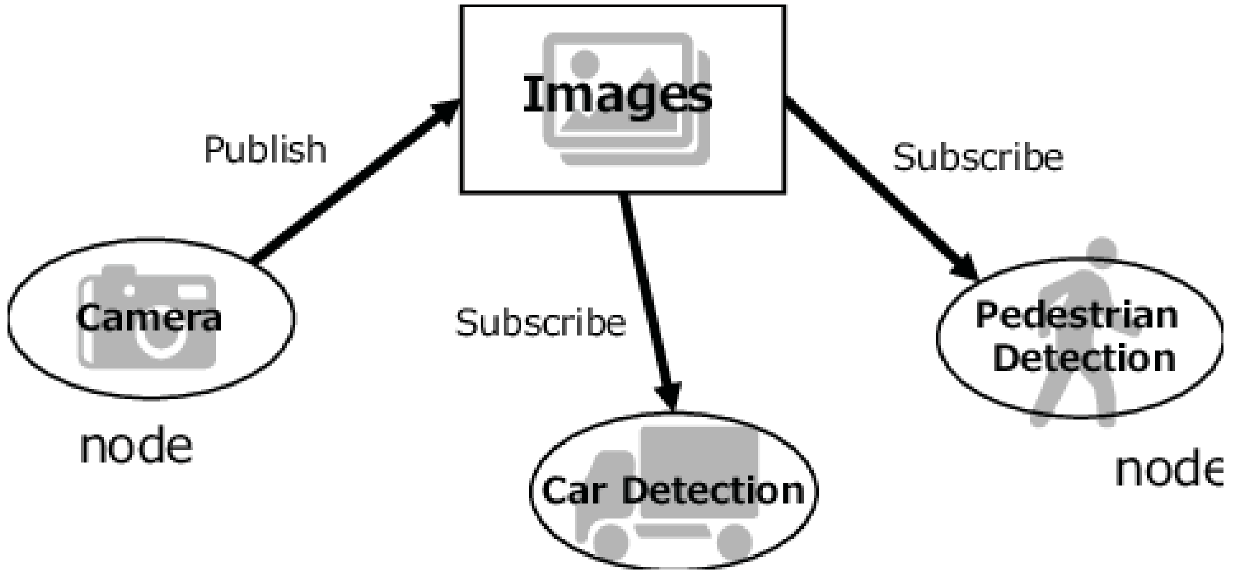
PUBLISHER & SUBSCRIBER IMPLEMENTATION + CUSTOM MESSAGE

Prepared by: Eng. Miriam Hani

REMEMBER…

- Publisher node: A ROS node that generates information is called a publisher. A publisher sends information to nodes through topics.

- Subscriber node: A ROS node that receives information is called a subscriber. It's subscribed to information in a topic and uses topic callback functions to process the received information.



PUBLISHER & SUBSCRIBER USING PYTHON:

- Objective:
  - Write a python file that initializes a talker node and publishes on it the string msg “Hello World”
  - Write a python file that initializes a listener node and subscribes to the talker node and prints out the msg received

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

import rospy
from std_msgs.msg import String

def talker():

    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz

    while not rospy.is_shutdown():
        hello_str = "hello world"
        pub.publish(hello_str)
        rate.sleep()

if __name__== '_main_':

    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

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

def callback(data):
       print ("I heard " + data.data)

def listener():

     rospy.init_node('listener', anonymous=True)

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

     # spin() simply keeps python from exiting until this node is stopped
     rospy.spin()

if __name__== '_main_':
     listener()
```

LET'S PRACTICE

- Implement publisher-subscriber communication between two nodes to send numbers.

PUBLISHER CODE:

#include "ros/ros.h“

• Include that includes all the headers necessary to use the most common public pieces of the ROS system.

#include "std_msgs/String.h“

• This includes the std_msgs/String message, which resides in the std_msgs package. This is a header generated automatically from the String.msg file in that package.

ros::init(argc, argv, "talker");

• Initialize ROS.

ros::NodeHandle nh;

• Create a handle to this process' node.

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

• Tell the master that we are going to be publishing a message of type std_msgs/String on the topic chatter. This lets the master tell any nodes listening on chatter that we are going to publish data on that topic. The second argument is the size of our publishing queue. In this case if we are publishing too quickly it will buffer up a maximum of 1000 messages before beginning to throw away old ones.

• NodeHandle::advertise() returns a ros::Publisher object, which serves two purposes:

1) it contains a publish() method that lets you publish messages onto the topic it was created with, and
2) when it goes out of scope, it will automatically unadvertise.

ros::Rate loop_rate(10);

• allows you to specify a frequency that you would like to loop at

std_msgs::Int32 count;

Create a variable count of type Int32

Count.data=0

Initialize count variable

```
while (ros::ok())
 {
```
- Loop untill Ctrl+C handling.

```
chatter_pub.publish(count);
```
- Now we actually broadcast the message to anyone who is connected.

```
ros::spinOnce();
```
- For trigering callbacks, not needed in this program.

```
loop_rate.sleep();
```
- Now we use the ros::Rate object to sleep for the time remaining to let us hit our 10hz publish rate.

```
++count.data;
}
Return 0;
}
```

SUBSCRIBER CODE:

```
#include <ros/ros.h>
#include <std_msgs/Int32.h>
 void counterCallback(const std_msgs::Int32::ConstPtr& msg)
```

Define a function called 'callback' that receives a parameter named 'msg'

```
{
ROS_INFO("%d" , msg->data);
```

Print the value 'data' inside the 'msg' parameter

```
 }
int main(int argc, char **argv)
{
ros::init(argc, argv, "topic_subscriber");
```

Initiate a Node called 'topic_subscriber’

```
ros::NodeHandle nh;
ros::Subscriber sub = nh.subscribe("counter" , 1000, counterCallback);
```

Create a Subscriber object that will listen to the /counter topic and will call the 'callback' function each time it reads something from the topic

```
 ros::spin();
```

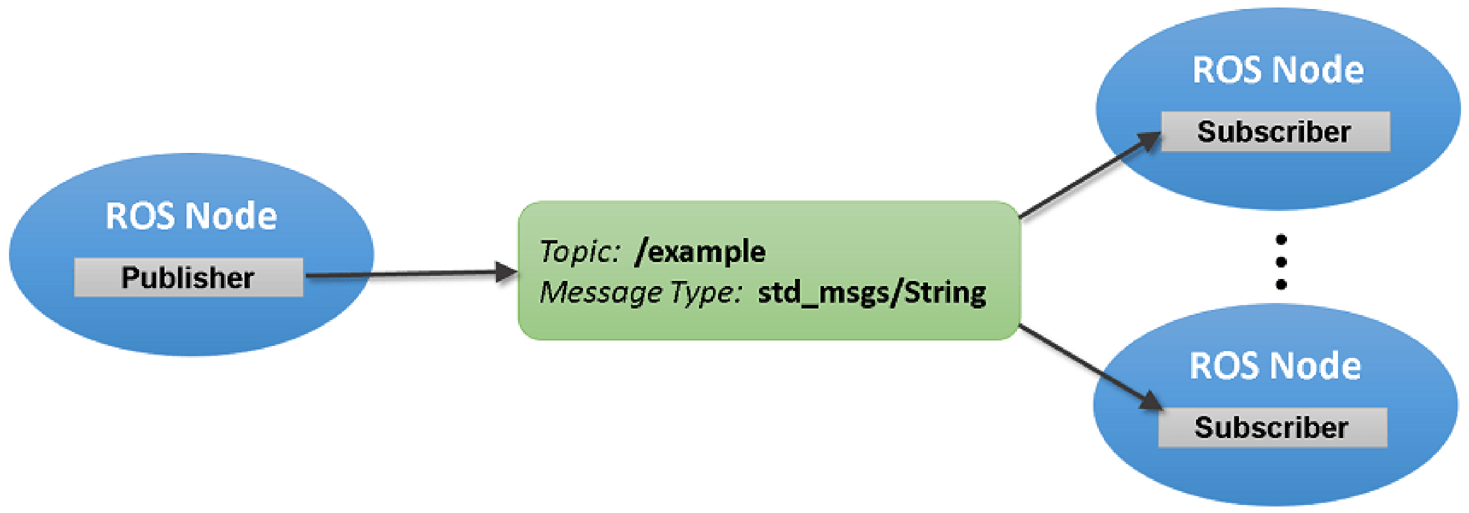
Create a loop that will keep the program in execution

```
return 0;
}
```

TASK:

- Implement a publisher-subscriber code using cpp to send “hello”
- Modify the code to let the publisher send “hello <your name> “

MESSAGES:

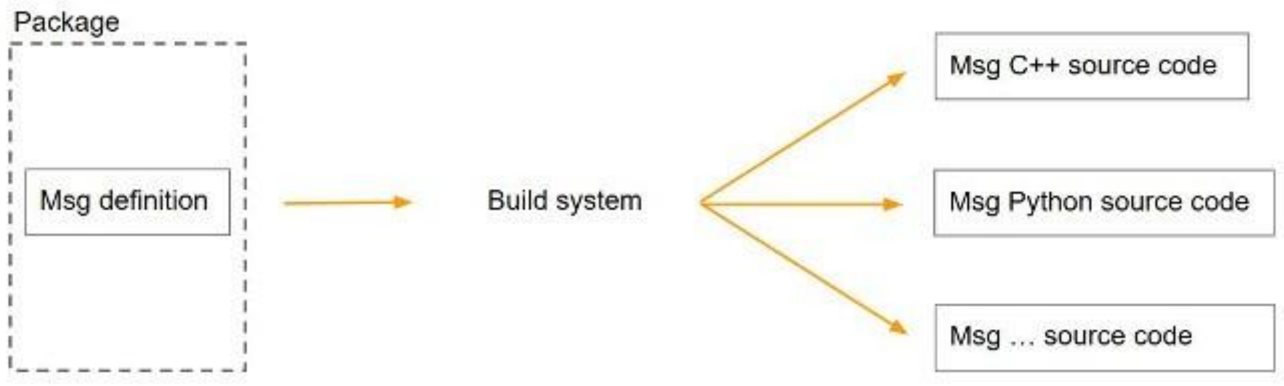


- msg files are simple text files that describe the fields of a ROS message. They are used to
- generate source code for messages in different languages. msgs are just simple text files with a
- field type and field name per line. The field types you can use are:

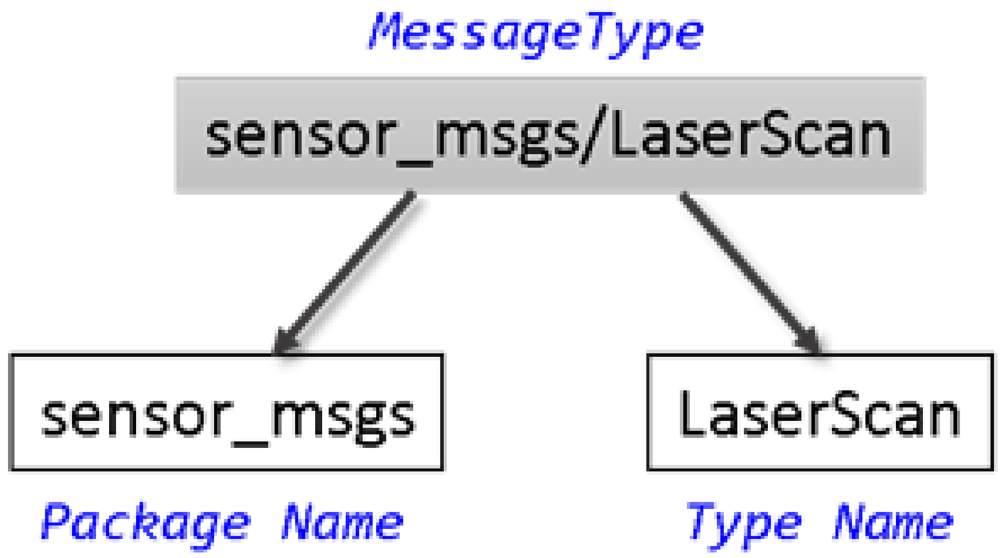
- • int8, int16, int32, int64 (plus uint*)
- • float32, float64
- • string
- • time, duration
- • other msg files
- • variable-length array and fixed-length array

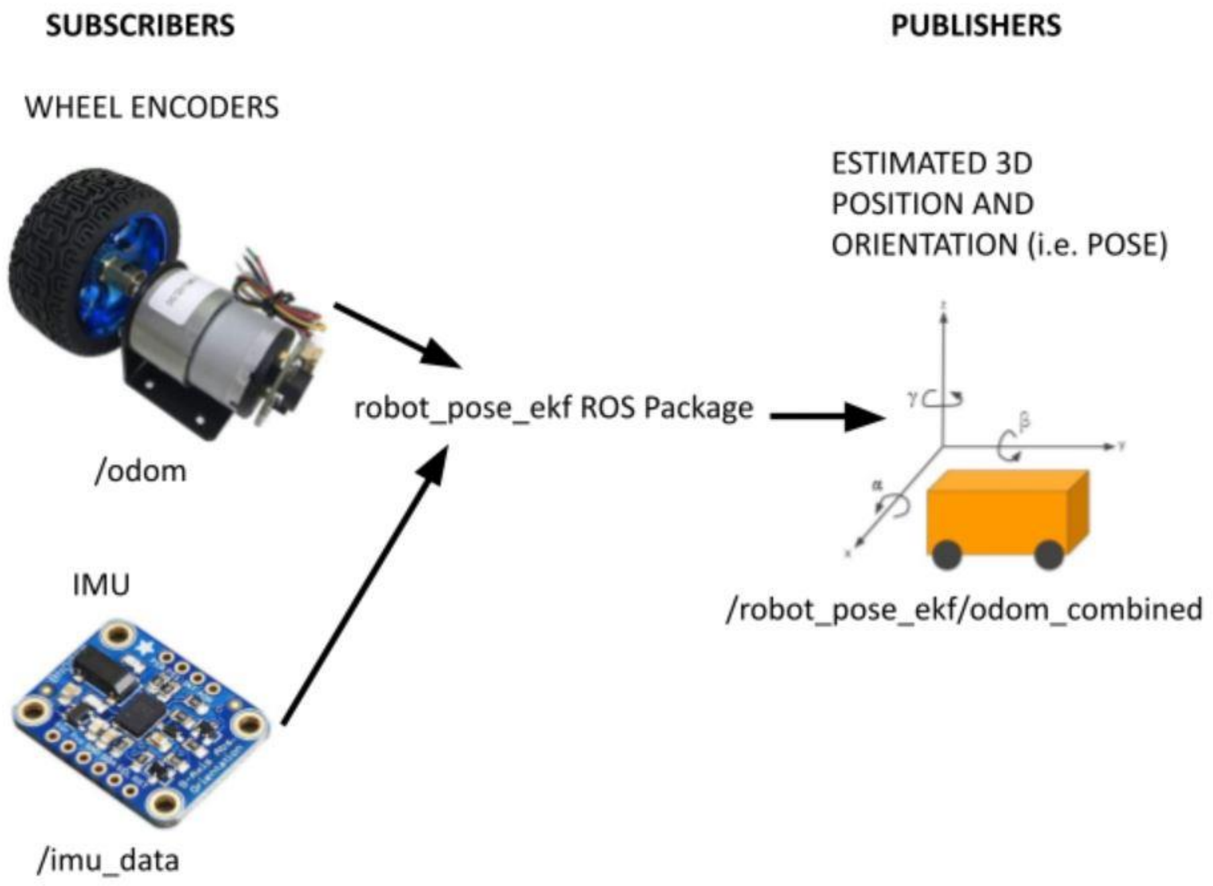
CUSTOM MESSAGE

- Sometimes, you will need to customize a certain message for your application.



LET'S EXPLORE SOME EXAMPLES OF CUSTOM MESSAGES

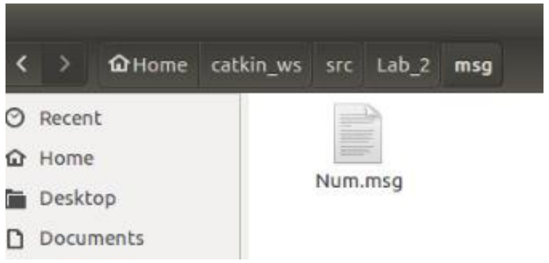



SUBSCRIBERS
WHEEL ENCODERS
/odom
IMU
/imu_data
robot_pose_ekf ROS Package
PUBLISHERS
ESTIMATED 3D POSITION AND ORIENTATION (i.e. POSE)
/robot_pose_ekf/odom_combined

- Odom message in nav_msgs is one of the most significant message types in autonomous robots.
- Odometry message stores an estimate of position and velocity of robot in free space.

STEPS TO CREATE A ROS MESSAGE:

- Create a message folder in your package
- Then create a message file in the package with extension .msg
- Edit the message file accordingly
- Update the dependencies accordingly
  - In package.xml
  - and CmakeLists.txt
- Compile the package using the catkin_make
- Make sure that your message is created using the rosmsg show



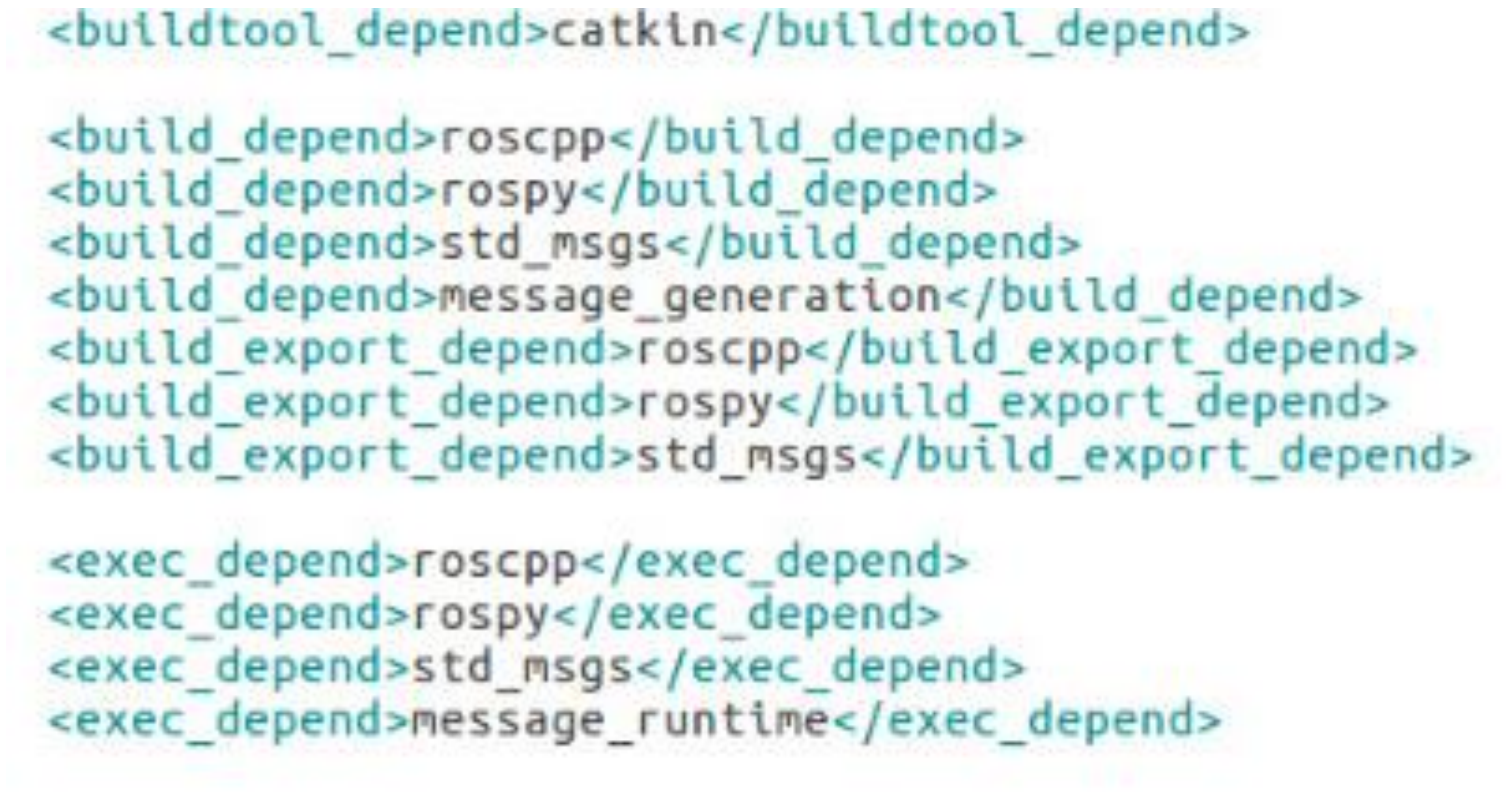
A MESSAGE CONSISTS OF:

- Fieldtype + fieldname
- A fieldtype may be one of the built-in types mentioned above or a name of another message defined by its one, for examples in geometry_msgs package.
- A field name is the name given to the message to enable it to be called anywhere else.
- For example:

1. string child_id
2. geometry_msgs/PoseWithCovariance pose
3. Header header
4. float32 years

- In package.xml, you should modify the following:

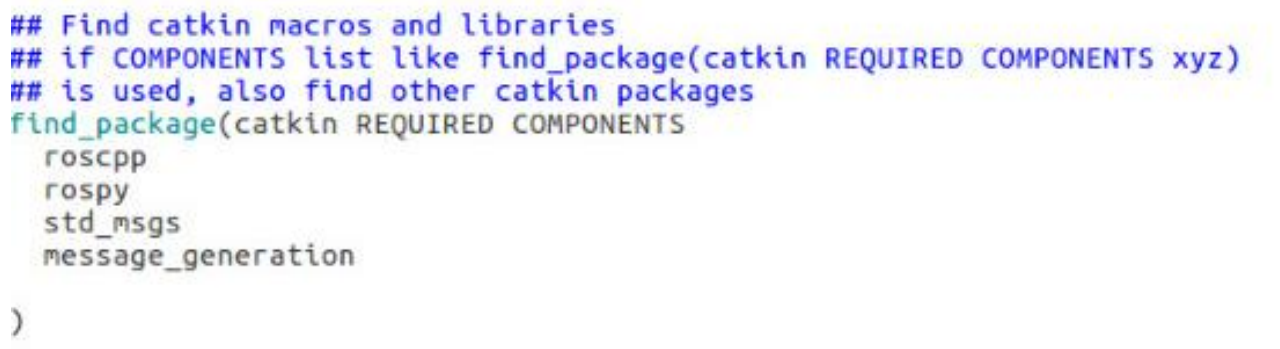
1. <build_depend>message_generation</build_depend>
2. <exec_depend>message_runtime</exec_depend>



```
<buildtool_depend>catkin</buildtool_depend>

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

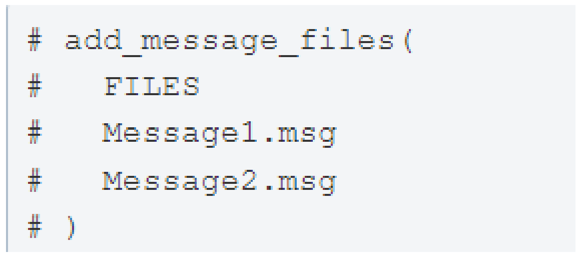
- In CMakeLists:

1. Add the message generation dependency to the find package call which already exists in your CMakeLists.txt so that you can generate messages. You can do this by simply adding message generation to the list of COMPONENTS such that it looks like this:



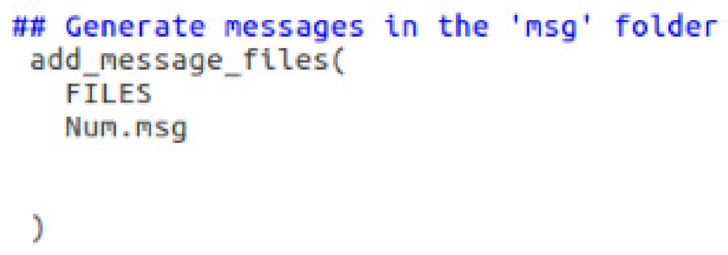
```
## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation

)
```

2. Find the following block of code:



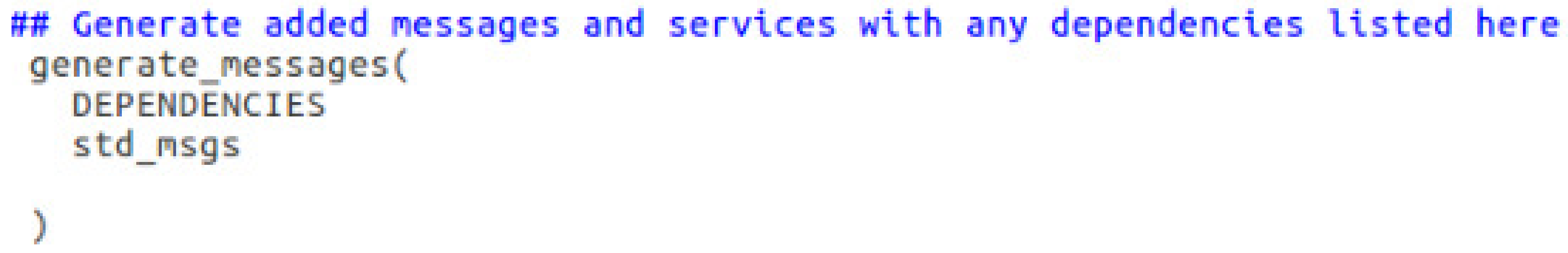
```
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )
```

3. Uncomment it by removing the # symbols and then replace the message's file name in Message1.msg, Message2.msg,…. files
with your .msg file, such that it looks like this:



```
## Generate messages in the 'msg' folder
 add_message_files(
   FILES
   Num.msg


 )
```

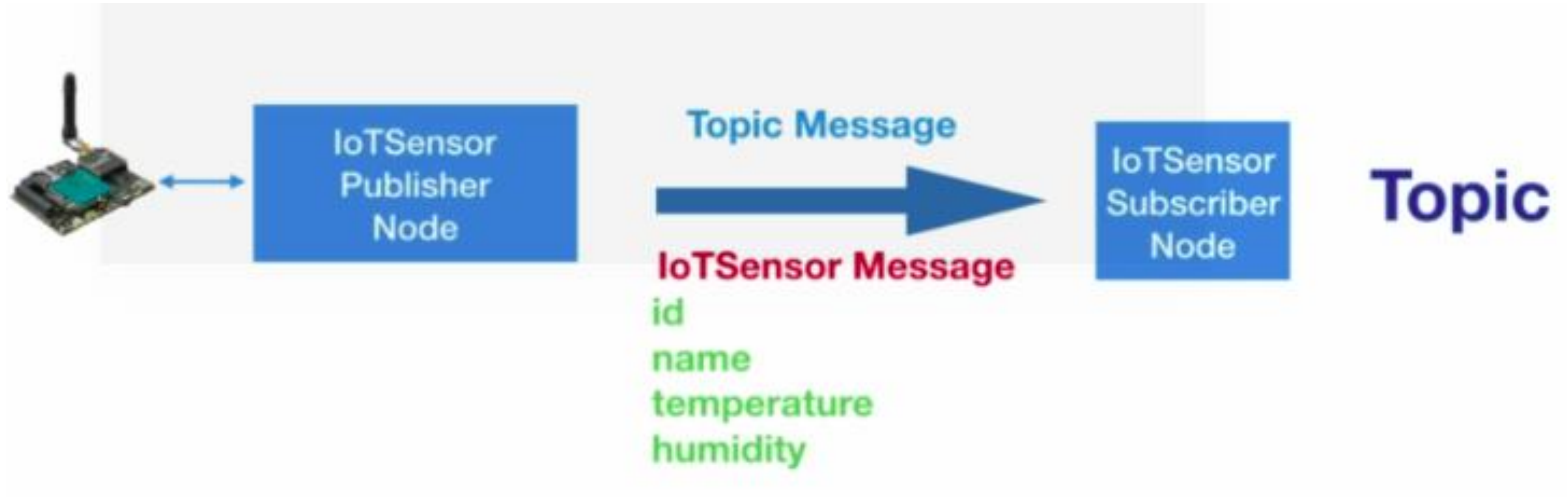
4. Don't forget the add the message package consisting the standard messages you used in your customized message file.
For example, if we used fieldtypes string, float,…..etc; then we should include in generate messages section std_msgs.
If we use Twist or Pose datatypes, then we should include in generate messages geometry_msgs



```
## Generate added messages and services with any dependencies listed here
 generate_messages(
   DEPENDENCIES
   std_msgs

 )
```

TASK:

- Create a custom message for a certain sensor which contains : id, name, temperature & humidity.
- Create the message file, build your message file by making the required modifications on CMakeLists and package.XML file
- Use rosmsg show to make sure your message is successfully created.


IoTSensor Publisher Node
Topic Message
IoTSensor Message
id
name
temperature
humidity
IoTSensor Subscriber Node
Topic