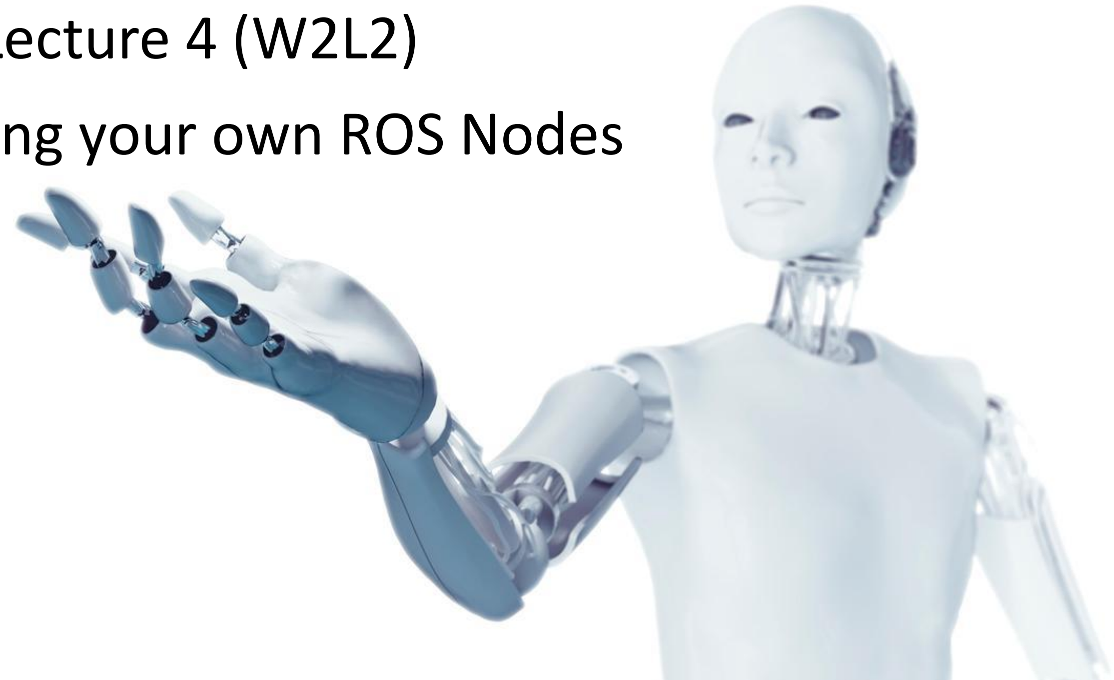
ROBOTICS

Lecture 4 (W2L2)

Developing your own ROS Nodes



Learning outcomes for today

After attending this lecture and doing any associated reading you should be able to:

- Understand what a ROS Node/Nodelet is and how it fits into ROS's distributed architecture
- Understand and use Publishers, Subscribers and Callbacks
- Develop your own ROS nodes/Nodelets
- Visualise your ROS Nodes/Nodelets
- Understand Services and Action Servers

What is a ROS Node?

- Recall: **ROS** is a peer-to-peer network made up of a Master, ***Nodes***, Messages and Services.
- **Nodes:** Independent processes that "do the actual work". Types of nodes include:
  - **Low-level:** "firmware" for sensors/actuators like LiDAR, Cameras, IMUs, Wheels, etc.
  - **Mid-Level:** "middleware" for things like pointcloud filtering, image processing, etc.
  - **High-level:** "software" like sensor fusion, navigation, localisation, etc.

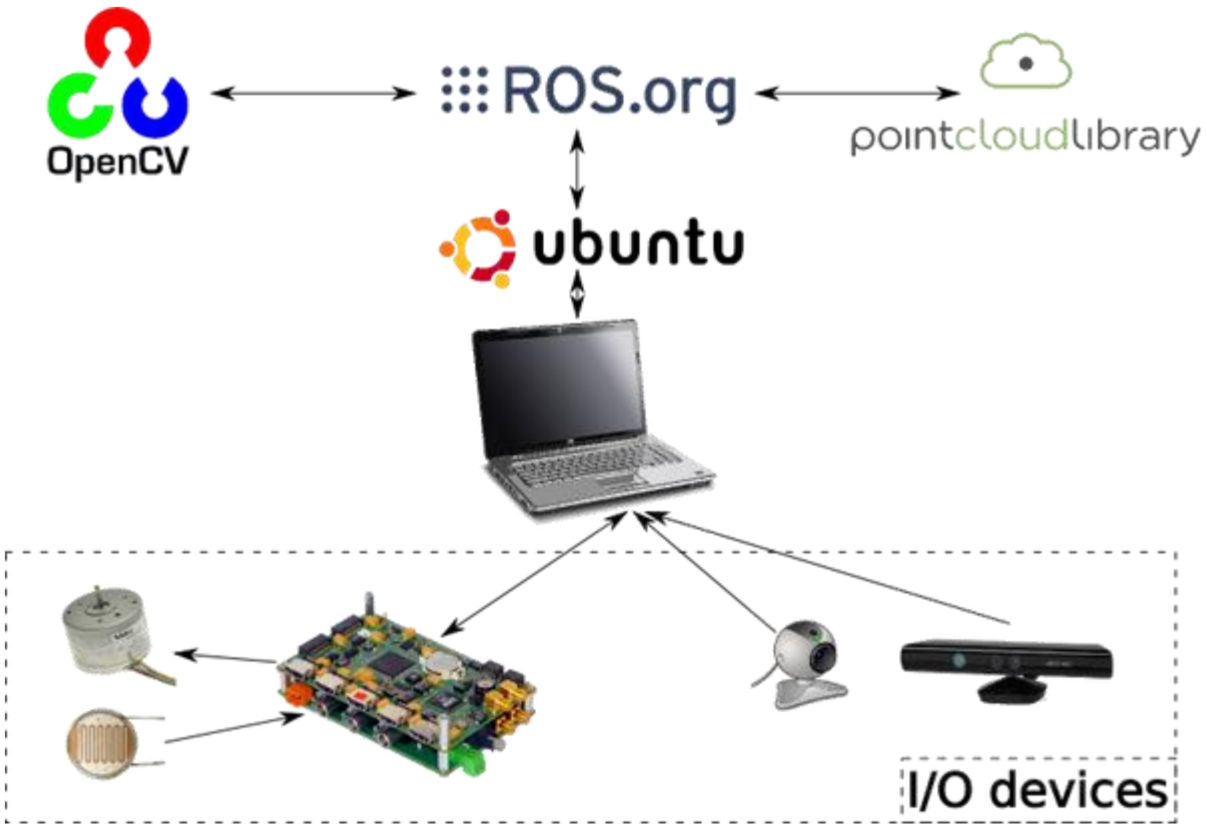
Why Nodes?

Modular

- Each node has a single job
- Development is easier
- Enforces good design principles

Distributed

- Each node is its own process
- No single point of failure
- Nodes can restart automatically



http://www.intorobotics.com/wp-content/uploads/2014/05/rsz_robot_working_sgrhjnhgfbdert54yyjuhgfbdetryujmh0005.png

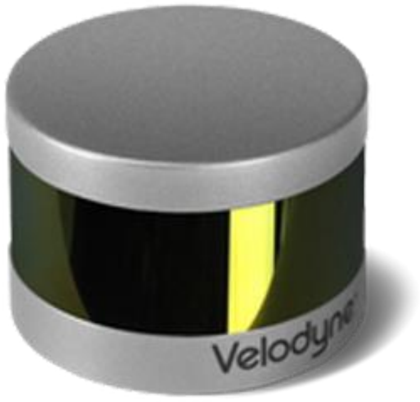
Low-Level Nodes

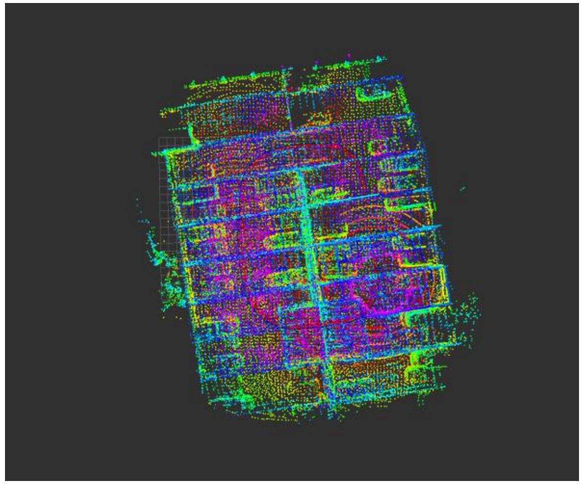
- Interface with Sensors
- Normally a ROS “wrapper” on the sensor’s SDK
- “Real-Time”
- Perform low-level operations:
  - Get Sensor Data
  - Parse into ROS standards
  - Articulate Motors
- Have clear boundaries between nodes

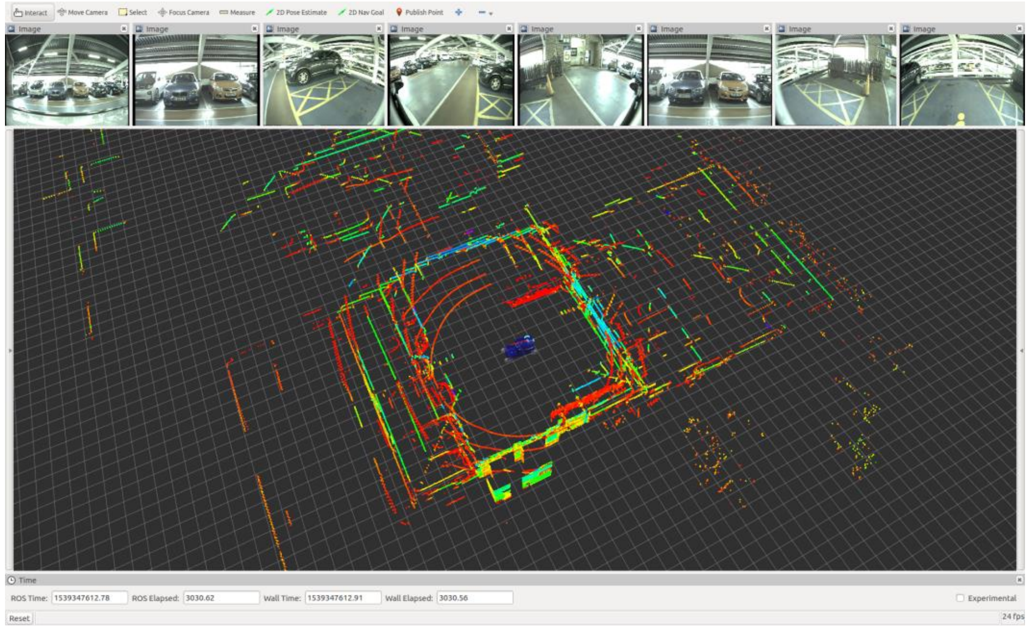
Can you explain why?

Node Example: Low-Level Driver

- Velodyne LiDAR Driver
  - Wrapper for Velodyne SDK
  - Exposes basic parameters
  - Converts raw data to ROS Messages (sensor_msgs/Scan, sensor_msgs/Pointcloud2)

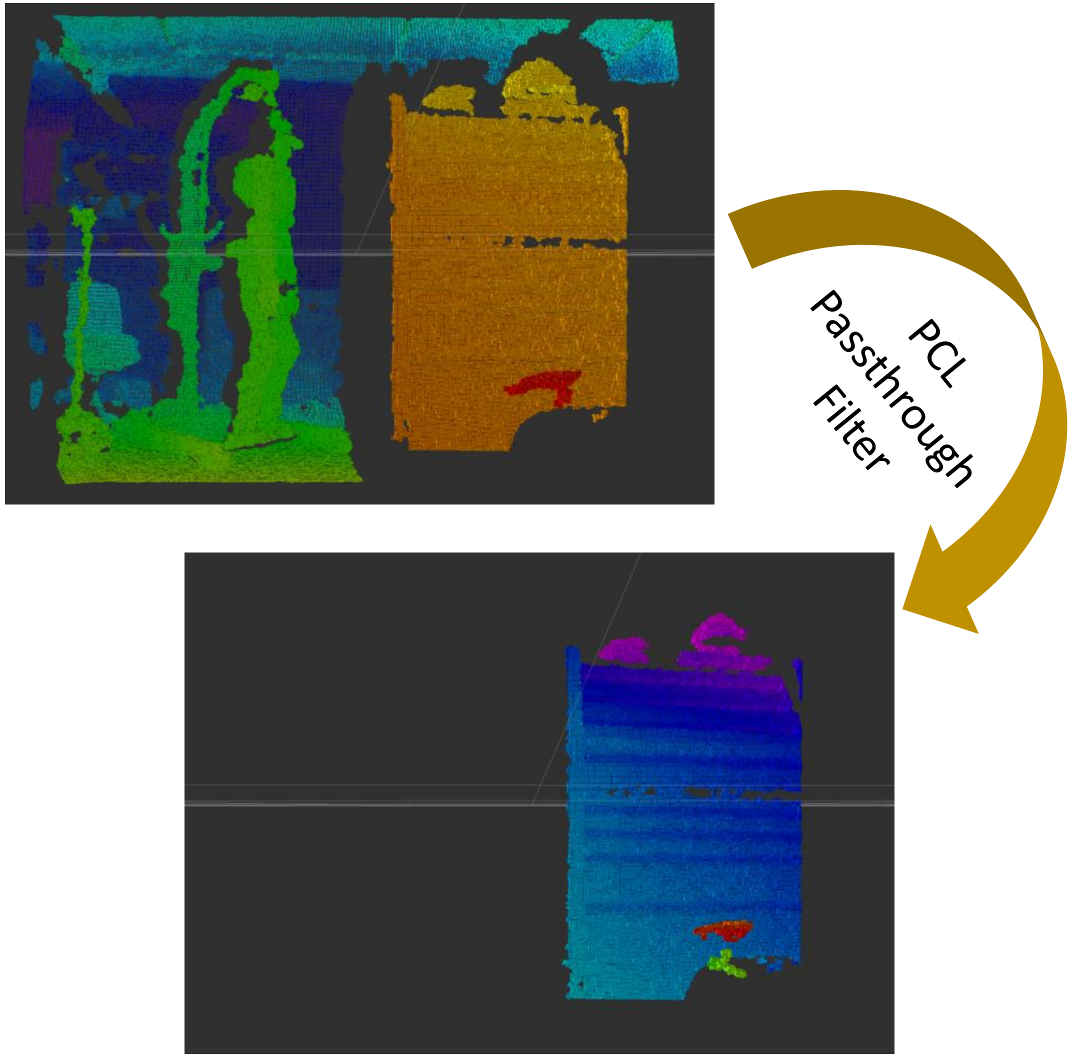






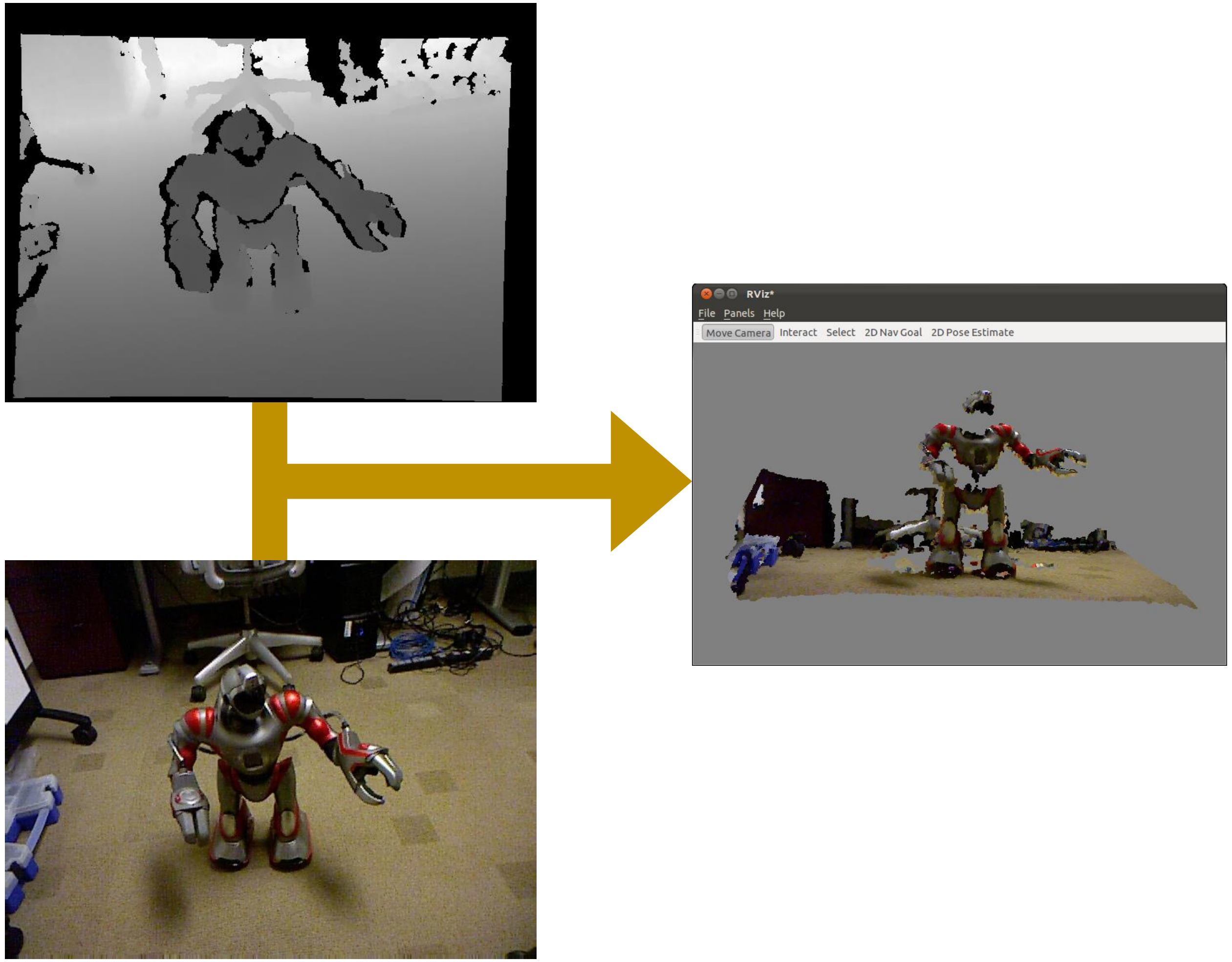
Mid-Level Nodes

- “Glue” between other Nodes (low-high and high-high)
- Perform basic, well-defined operations:
    - Image Processing
    - Perception
    - Sensor Fusion
- No sensor interfaces, heavy processes
- “Real-Time”
- Ideal for Nodelets!



Node Example: Kinect Processing

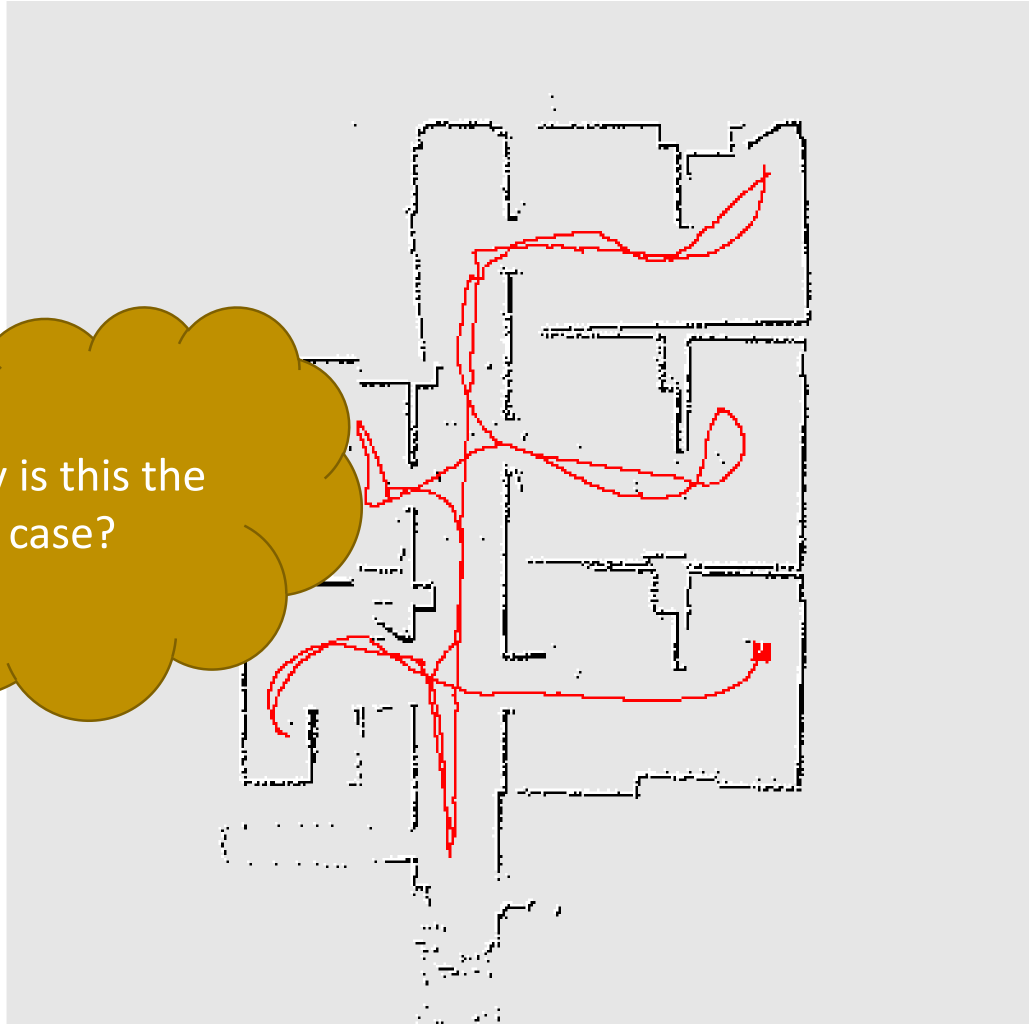
- Receive from Low-Level node:
  - Colour Image
  - Depth Image
- Mid-Node does processing:
  - Take depth Image
  - Match Pixels to Colour
  - Convert to 3D Pointcloud



High-Level Nodes

- Perform more complex operations like:
  - Robot Position Estimation
  - Grasping
  - Pathplanning
- Have less well-defined node boundaries

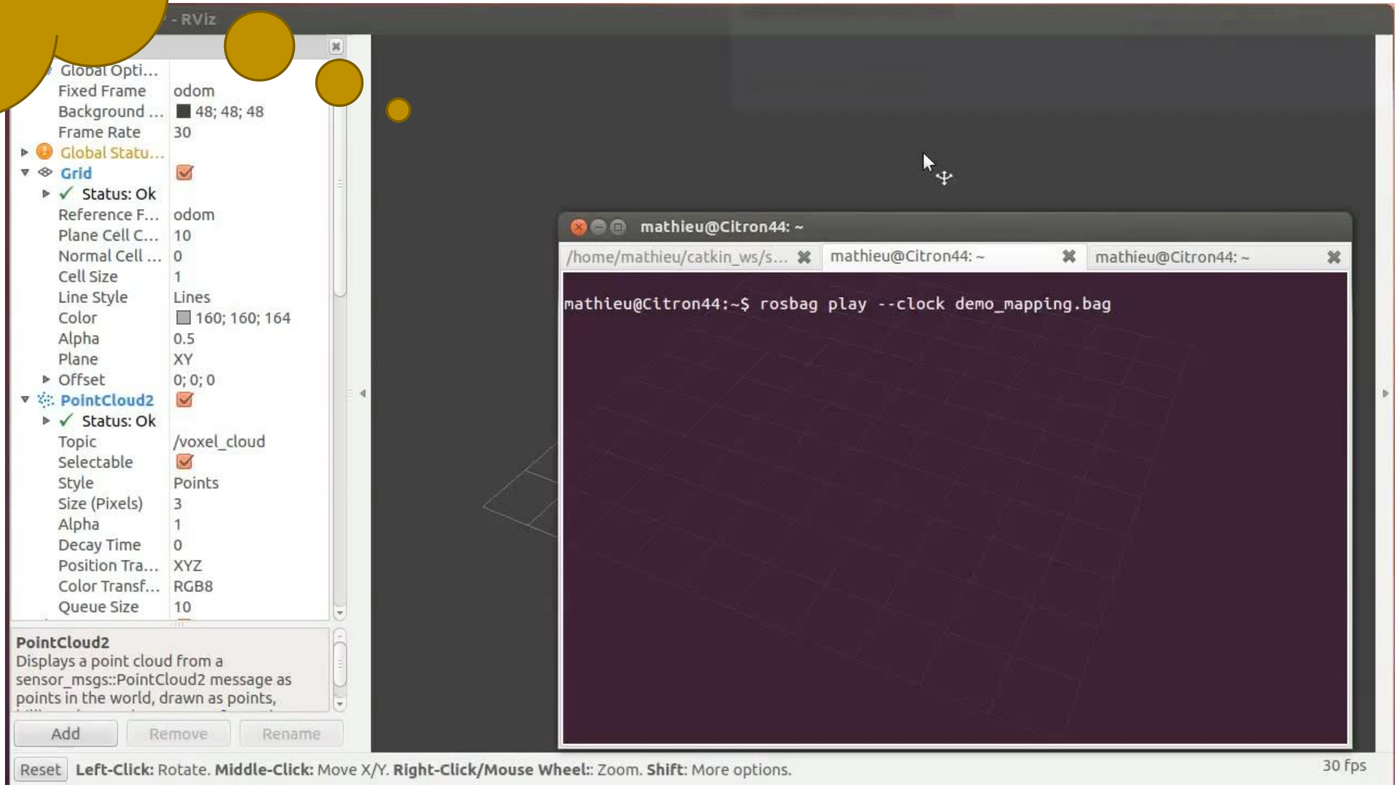
Why is this the case?



Node Example: High-Level Processing

Could we split this into multiple nodes?

- RTabMap
  - Huge node that doe
  - Receives Data from:
    - Low-Level Nodes:
      - Camera
      - IMU
      - LiDAR
    - Mid-Level Nodes:
      - Image Processing
      - Sensor Fusion
      - Transform Tree
    - High-Level Nodes:
      - Pathplanning
      - Exploration



What is a ROS Node

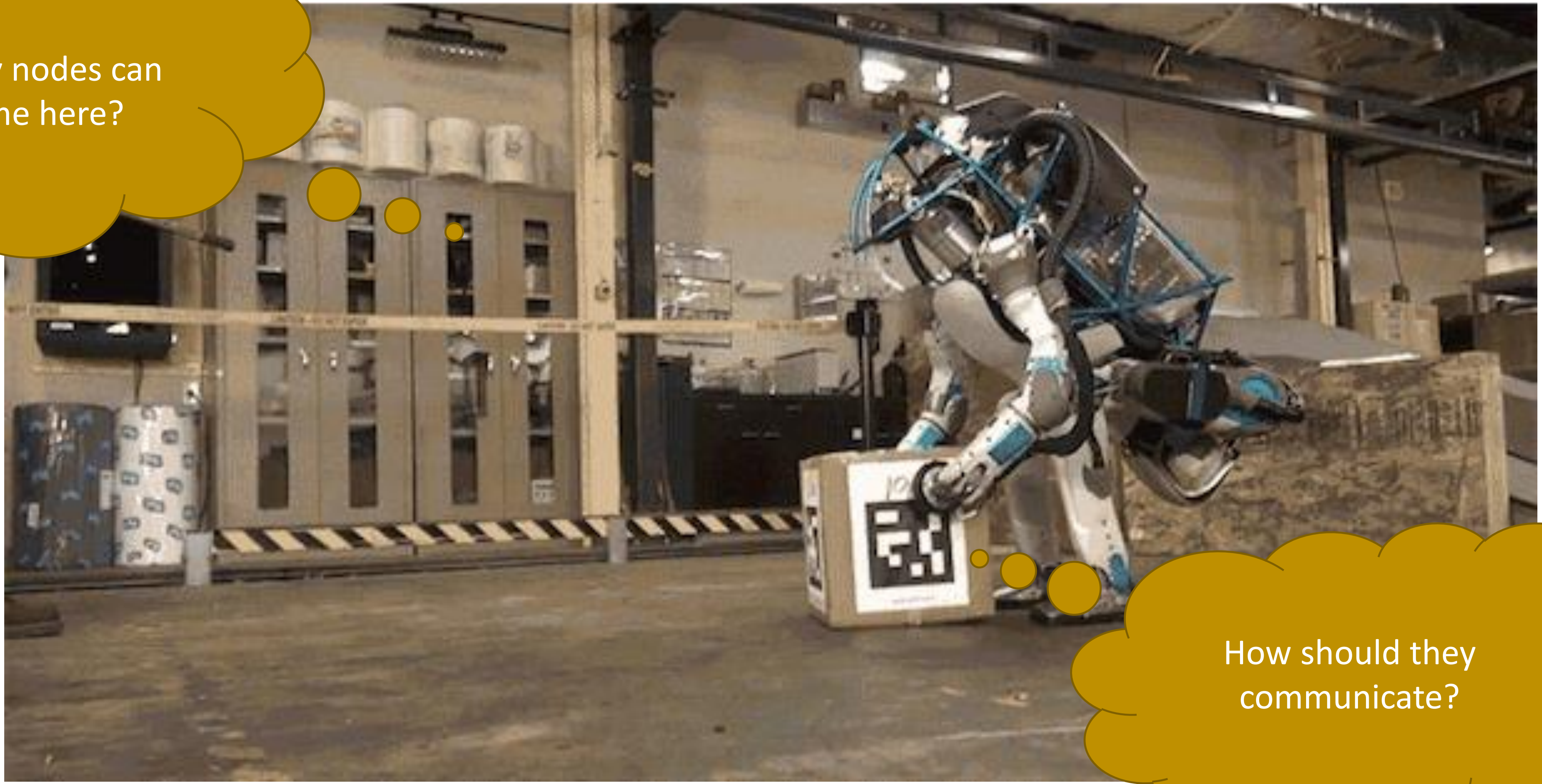
- **Nodes** exist in "**packages**"
- A Package is:
  - The software organization unit of ROS code.
  - A collection of nodes that perform similar or related tasks.
  - Libraries, scripts, and/or other task-related artefacts (messages, services, etc.)
- Each node is a separate executable in the package

Can you think of an example?

What is the point of a package?

Node Overview

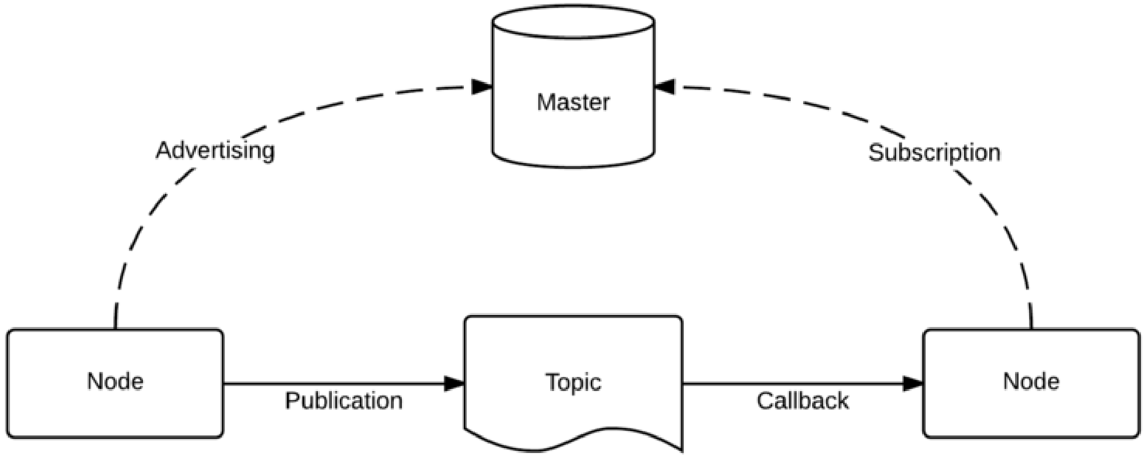
How many nodes can you name here?



How should they communicate?

Communicating between Nodes

- ROS uses a Publish-Subscribe model.
- Typical Communication:
    1. Node A advertises topic
    2. Node B subscribes to topic
    3. Master handles setup (checking datatypes, establishing link, etc.)
    4. Node A publishes message to topic
    5. Message triggers callback in Node B

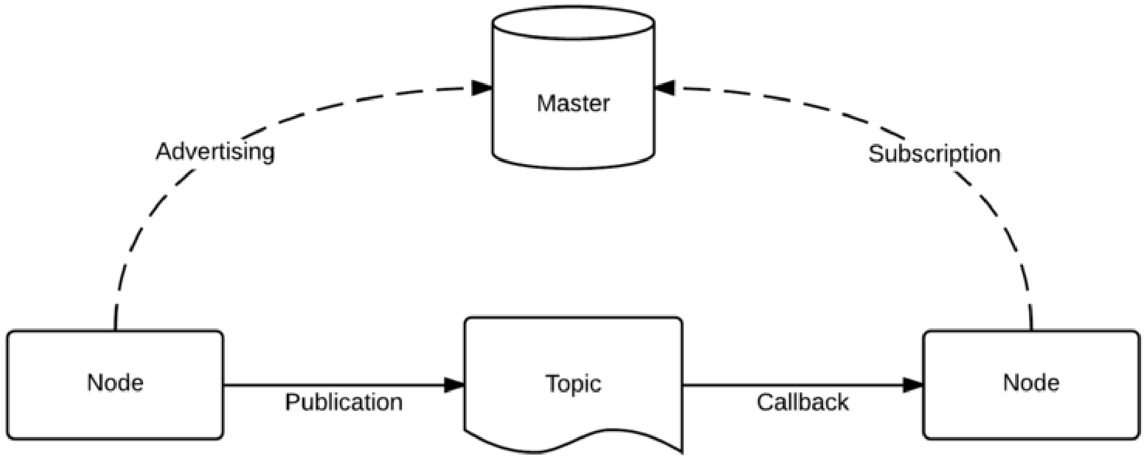


https://commons.wikimedia.org/wiki/File:ROS-master-node-topic.png

Communicating between Nodes

Message Examples:

- geometry_msgs
  - Represent common geometric primitives (Point, Quaternion, Pose, Twist, etc.)
- sensor_msgs
  - Represent Sensor Data (Image, JointState, Pointcloud, etc.)
- nav_msgs
  - Messages for Navigation (Odometry, Path, etc.)
- actionlib_msgs
  - Messages that represent actions (GoalID, GoalStatus, etc.)
- Custom Messages
  - You can define your own messages
- ...etc

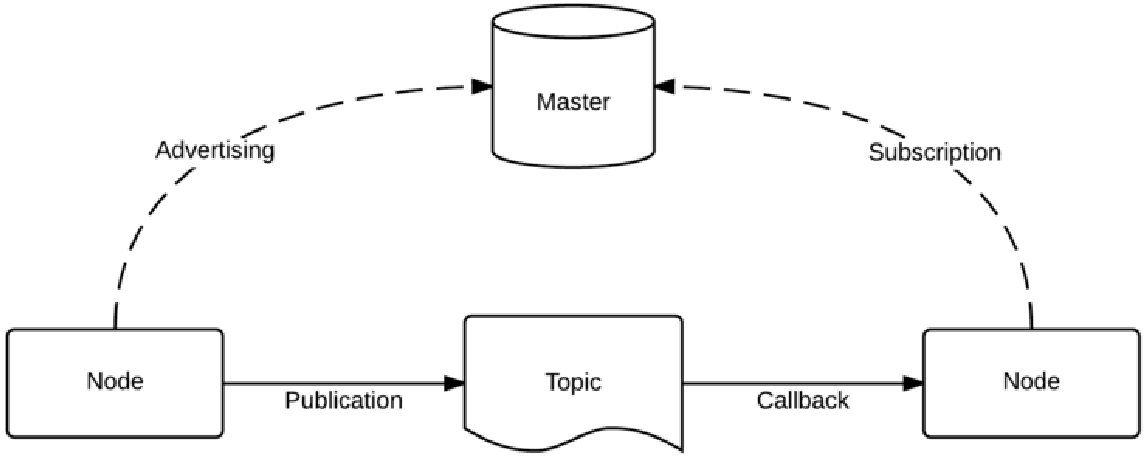


https://commons.wikimedia.org/wiki/File:ROS-master-node-topic.png

Communicating between Nodes

ROS Provides other ways of passing information:

- Services
  - Allow nodes to send a **request** and receive a **response**
- Parameter Server
  - Allows parameters to be stored globally
- Action Server
  - Allows execution of long-running goals that can be cancelled, overwritten or introspected.



https://commons.wikimedia.org/wiki/File:ROS-master-node-topic.png

Nodes in Code

How do I actually write a node?

Node Basics

- Creating a workspace:

```
% mkdir -p ~/ros_ws/src
% cd ~/ros_ws/src
```

- Creating a Package:

```
% catkin_create_pkg [package_name] rospy roscpp [other dependencies]
```

- Compiling Workspace (from ws root):

```
% cd ~/ros_ws
% catkin_make
```

- Running a Node:

```
% rosrun [package_name] [node_name]
```

Developing a Node

- Now our workspace is complied, we can begin coding!
- Lets work through an example…

Node Example: Chatter C++

Publisher

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(10);

  int count = 0;
  while (ros::ok())
  {
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());

    chatter_pub.publish(msg);
    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

Subscriber

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("chatter", 1000,
chatterCallback);

  ros::spin();

  return 0;
}
```

Code Breakdown: Publisher

- ROS specific includes, “**ros/ros.h**” is the main ROS header file all ROS nodes include it.
- **“std_msgs/String.h”** is the header file for the message type used here

- The **“ros::init()”** function needs to see argc and argv so that it can perform any ROS arguments and name remapping that were provided at the command line.
- The third argument to init() is the **name of the node**. You must call ros::init() before using any other part of the ROS system.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(10);

  int count = 0;
  while (ros::ok())
  {
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());

    chatter_pub.publish(msg);
    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

Code Breakdown: Publisher

- **NodeHandle** is the main access point to communications with the ROS system.
- The first NodeHandle constructed will fully initialize this node, and the last NodeHandle destructed will close down the node.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(10);

  int count = 0;
  while (ros::ok())
  {
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());

    chatter_pub.publish(msg);
    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

Code Breakdown: Publisher

- The **advertise()** function is how you tell ROS that you want to publish on a given topic name.
- After this advertise() call is made, the **master node** will notify anyone who is trying to subscribe to this topic name, and they will in turn negotiate a **peer-to-peer** connection with this node.
- advertise() returns a **Publisher** object which allows you to publish messages on that topic through a **call to publish().**
- Once all copies of the returned Publisher object are destroyed, the topic will be automatically unadvertised.
- The **second parameter** to advertise() is the **size of the message queue** used for publishing messages. If messages are published more quickly than we can send them, the number here specifies how many messages to buffer up before throwing some away.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(10);

  int count = 0;
  while (ros::ok())
  {
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());

    chatter_pub.publish(msg);
    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

Code Breakdown: Publisher

- A **ros::Rate** object allows you to specify a frequency that you would like to loop at.
- It will keep track of how long it has been since the last call to **Rate::sleep(),** and sleep for the correct amount of time.
- The constructor parameter is the frequency in Hz.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(10);

  int count = 0;
  while (ros::ok())
  {
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());

    chatter_pub.publish(msg);
    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

Code Breakdown: Publisher

- ros::ok() monitors that state of the ROS node.
- If the node gets a **shutdown call** (from either an internal or an external source), this returns **FALSE**.
- Otherwise, returns **TRUE**.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(10);

  int count = 0;
  while (ros::ok())
  {
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());

    chatter_pub.publish(msg);
    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

Code Breakdown: Publisher

- This is a **message object**.
- You put your **data** here and **publish** it using the publisher
- Each message object has a **header file**.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(10);

  int count = 0;
  while (ros::ok())
  {
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());

    chatter_pub.publish(msg);
    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

Code Breakdown: Publisher

- The **publish()** function is how you **send messages**.
- The **parameter** is the **message object**.
- The type of this object must agree with the type given as a template parameter to the advertise<>() call, as was done in the constructor above.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(10);

  int count = 0;
  while (ros::ok())
  {
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());

    chatter_pub.publish(msg);
    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

Code Breakdown: Publisher

- Checks for any outstanding ROS operations.
- Not doing anything at the moment
- Will trigger “callbacks” when we have a subscriber.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(10);

  int count = 0;
  while (ros::ok())
  {
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());

    chatter_pub.publish(msg);
    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

Code Breakdown: Subscriber

- Subscribe to the **chatter topic** with the **master**.
- **ROS will call** the **chatterCallback() function** whenever a new message arrives.
- The 2nd argument is the queue size, in case we are not able to process messages fast enough we will start throwing away old messages as new ones arrive.
- **NodeHandle::subscribe()** returns a **ros::Subscriber** object, that you must hold on to until you want to unsubscribe. When the Subscriber object is destructed, it will automatically unsubscribe from the chatter topic.
- There are versions of the NodeHandle::subscribe() function which allow you to specify a class member function, or even anything callable by a Boost.Function object.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("chatter", 1000,
chatterCallback);

  ros::spin();

  return 0;
}
```

Code Breakdown: Subscriber

- **ros::spin()** will enter a loop, pumping **callbacks**.
- With this version, **all callbacks** will be called from within this thread (the main one).
- ros::spin() will exit when Ctrl-C is pressed, or the node is shutdown by the master.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("chatter", 1000,
chatterCallback);

  ros::spin();

  return 0;
}
```

Code Breakdown: Subscriber

- This is the **callback function** that will get called when a new message has arrived on the **chatter topic**.
- The message is passed in a **boost shared_ptr**, which means you can store it off if you want, without worrying about it getting deleted underneath you, and without copying the underlying data.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("chatter", 1000,
chatterCallback);

  ros::spin();

  return 0;
}
```

Compiling C++ Code

- All C++ code needs these three lines to compile:

```
add_executable(<exe_name> src/<source_file>.cpp)

add_dependencies(<exe_name> ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS}
<other_dependencies>)

target_link_libraries(<exe_name>
  ${catkin_LIBRARIES}
  <other_libraries>
)
```

Creates **executable** to be compiled

Adds “**dependencies**” to executable, these are things that must be compiled **BEFORE** the executable

Links **libraries** to executable

```
$ rosrun <package_name> <exe_name>
```

Node Example: Chatter Python

Can we find the same elements as the C++ here?

Publisher

talker.py

```
#!/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 %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

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

```
$ chmod +x talker.py
$ rosrun <package_name> talker.py
```

Subscriber

listener.py

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

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():

    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
    rospy.spin()

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

```
$ chmod +x listener.py
$ rosrun <package_name> listener.py
```

Launch Files: Basic

Assume we have a “rospy_tutorials” package, with a “listener” node:

```
$ rosrun rospy_tutorials listener
```

OR

~/catkin_ws/rospy_tutorials/launch/listener.launch

```xml
<launch>
  <!-- a basic listener node -->
  <node name="listener-1" pkg="rospy_tutorials" type="listener" />
</launch>
```

```
$ roslaunch rospy_tutorials listener.launch
```

Launch Files: Basic

```
<launch>
  <!-- a basic listener node -->
  <node name="listener-1" pkg="rospy_tutorials" type="listener" />

  <!-- pass args to the listener node -->
  <node name="listener-2" pkg="rospy_tutorials" type="listener" args="-foo arg2" />

  <!-- a respawn-able listener node -->
  <node name="listener-3" pkg="rospy_tutorials" type="listener" respawn="true" />

  <!-- start listener node in the 'wg1' namespace -->
  <node ns="wg1" name="listener-wg1" pkg="rospy_tutorials" type="listener" respawn="true" />

</launch>
```

Launch Files: Arguments, Params and Remaps

```
<launch>
  <!-- declare arg to be passed in (with default value)-->
  <arg name="my_arg" default="my_value"/>

  <!-- a basic listener node with parameters -->
  <node name="listener-1" pkg="rospy_tutorials" type="listener" respawn="true">

        <!-- read value of arg into private parameter -->
        <param name="my_param" value="$(arg my_arg)"/>

        <!-- nodes can have their own remap args -->
        <remap from="chatter" to="hello-1"/>

        <!-- you can set environment variables for a node -->
       <env name="ENV_EXAMPLE" value="some value" />

  </node>

</launch>
```

More At: http://wiki.ros.org/roslaunch/XML

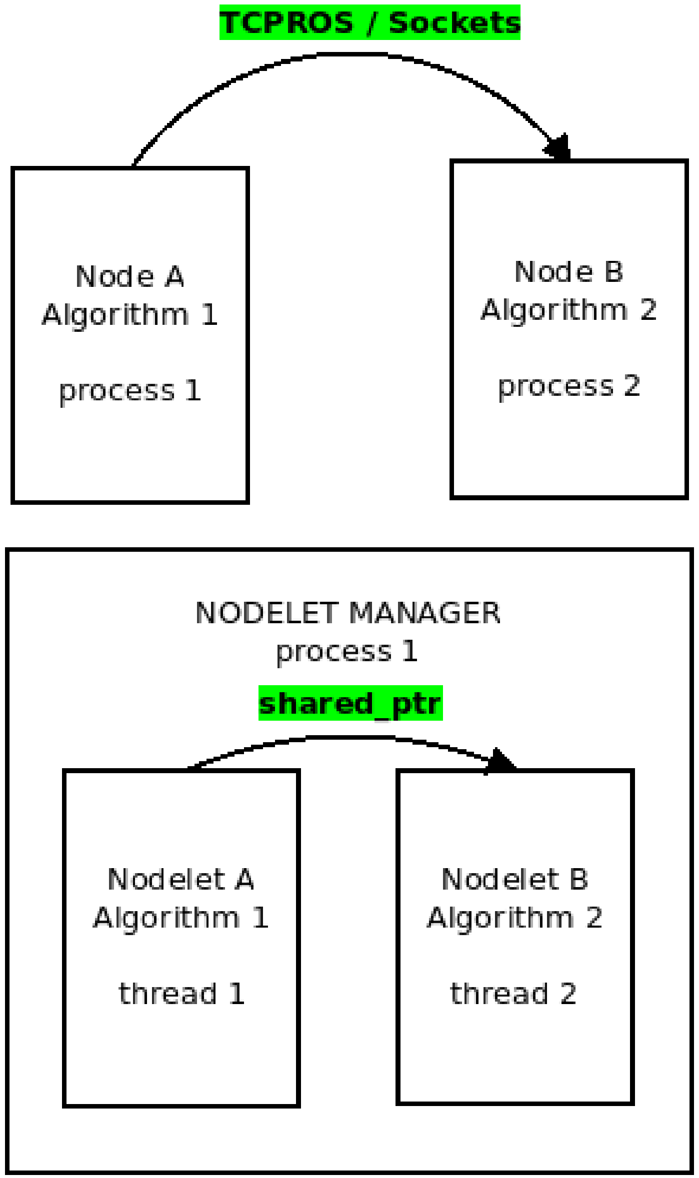
Limitations of Node Paradigm

Why are nodes a BAD idea?

- cy

Nodelets

- Cost-Free Message Passing
- Single Process with Shared Memory
- Good for closely related tasks
- Nodes can be ported to nodelets
- More Info: http://wiki.ros.org/nodelet/Tutorials

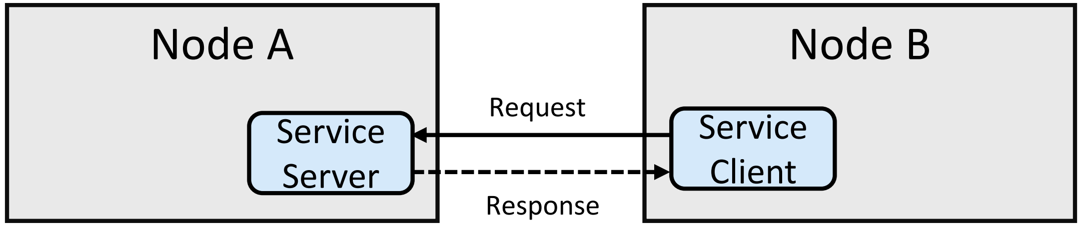


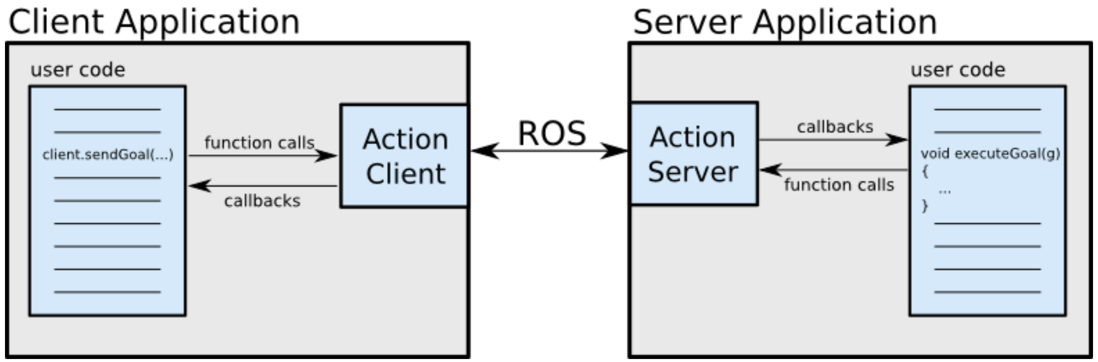
Limitations of Publish/Subscribe Model

Why are messages a BAD idea?

Services and Actions

- ROS provides alternative communication options:
  - **Services**:
    - Simple Request/Response interface
  - **Actions:**
    - More complex interface that includes cancellation requests, periodic feedback, etc.
    - Used for time consuming and complex services



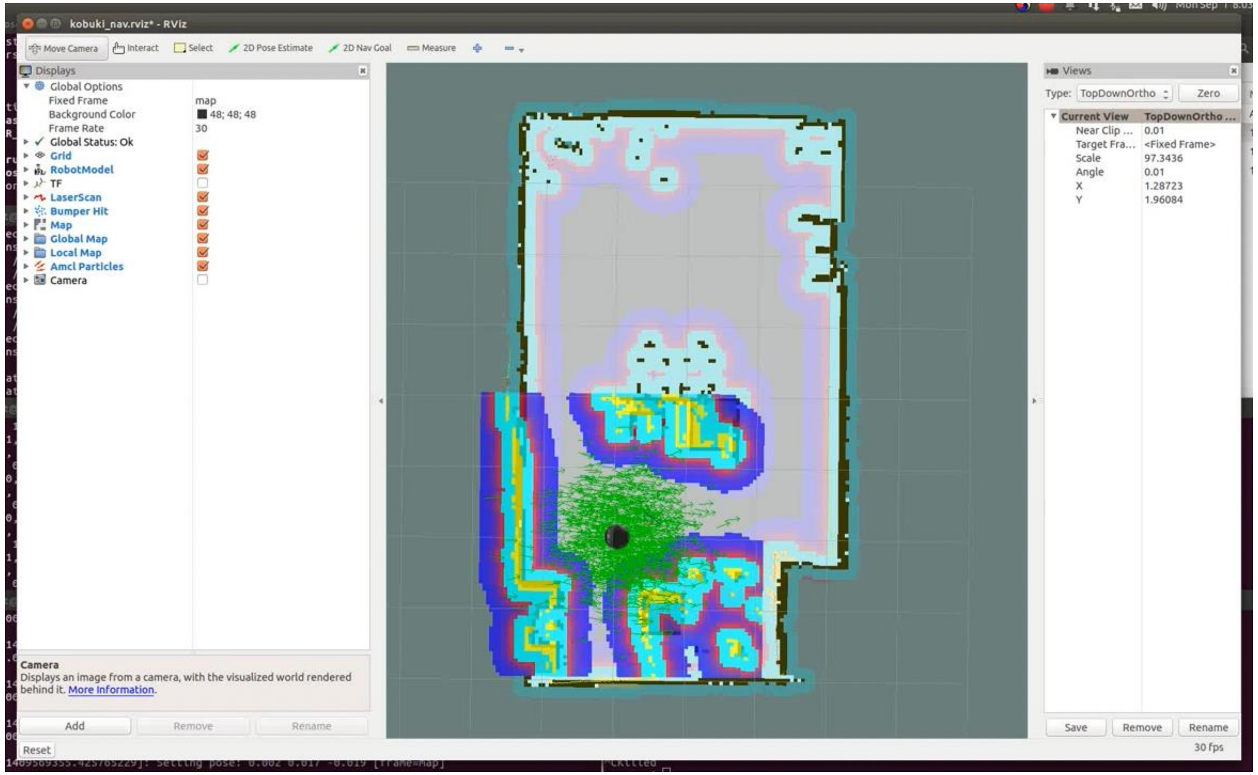


Visualising in RViz

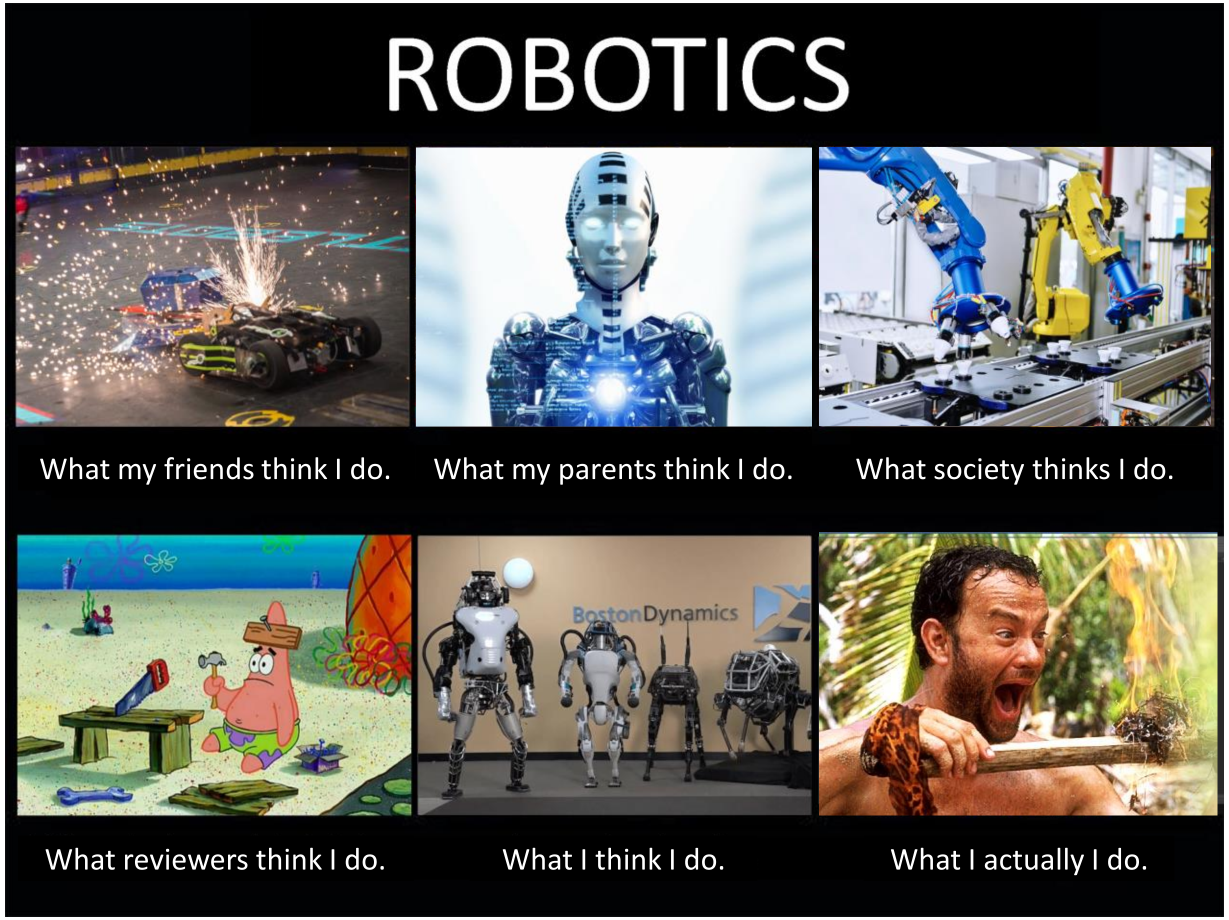
- What is RVIZ?
  - ROS's standard visualisation tool
- What can it do?
  - Visualise most default ROS messages (std_msgs, geometry_msgs, nav_msgs, etc.)
- Common Problems:
  - TF Errors: The frame_id of your message has to be in the TF tree for rviz to work.
- Should I use it?
  - **YES!** It's one of ROS's best features

- Running Rviz:

```
% rosrun rviz rviz
```



In case you are feeling overwhelmed…

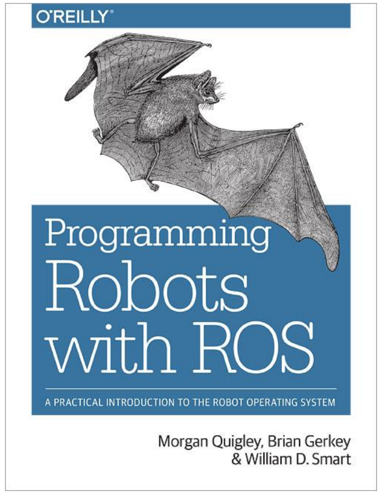


Summary

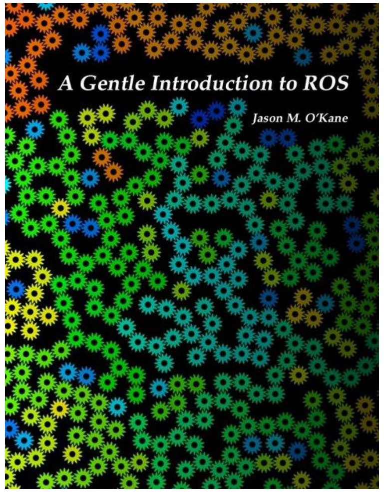
After attending this lecture and doing any associated reading you should be able to:

- Understand what a ROS Node/Nodelet is and how it fits into ROS's distributed architecture
- Understand and use Publishers, Subscribers and Callbacks
- Develop your own ROS nodes/Nodelets
- Visualise your ROS Nodes/Nodelets
- Understand Services and Action Servers

Further reading



- Ch. 2 – Preliminaries
- Ch. 3 – Topics
- Ch. 4 – Services
- Ch. 5 - Actions



- Ch. 2.6 - 2.7 – Nodes, Topics and Messages
- Ch. 3 – Writing ROS Programs
- Ch. 6 – Launch Files

- Core Tutorials – http://wiki.ros.org/ROS/Tutorials
- ActionLib Tutorials - http://wiki.ros.org/actionlib_tutorials/Tutorials
- Nodelet Tutorials - http://wiki.ros.org/nodelet/Tutorials