UAVs@Berkeley

ROS Workshop

Suneel Belkhale, Trey Fortmuller

The Plan

- ROS Intro
  - What is ROS/why ROS?
  - ROS concepts
- Working with ROS
  - Project structure
  - Command line tools
- **Interactive Implementation Demo**
- Extras
  - roslaunch
  - Parameter server and services
- ROS References and Takeaways
- *Soft* Prereqs: Linux command line, Python
- Content for the workshop: https://github.com/UAVs-at-Berkeley/ros_workshop

What is ROS?

What is ROS?

“The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools, ROS has what you need for your next robotics project. And it's all open source.” - ros.org

What is ROS?

- A “meta” operating system, robotic middleware
  - Not like windows or Mac OS, but provides similar capability
  - Runs in Linux (esp. Ubuntu), Linux-like systems
- Supports numerous programming languages (mostly C++ and Python)
- Communication paradigms (agent based, nodes)
  1. **Publish/Subscribe**
  2. Services
  3. Parameter Server

ROS Philosophy

- Peer to peer
  - Individual programs communicate over defined API (ROS messages, services, etc.).
- Distributed
  - Programs can be run on multiple computers and communicate over the network.
- Multi-lingual
  - ROS modules can be written in any language for which a client library exists (C++, Python, MATLAB, Java, etc.)
- Light-weight
  - Stand-alone libraries are wrapped around with a thin ROS layer.
- Free and open-source
  - Most ROS software is open-source and free to use.

Why do we use it?

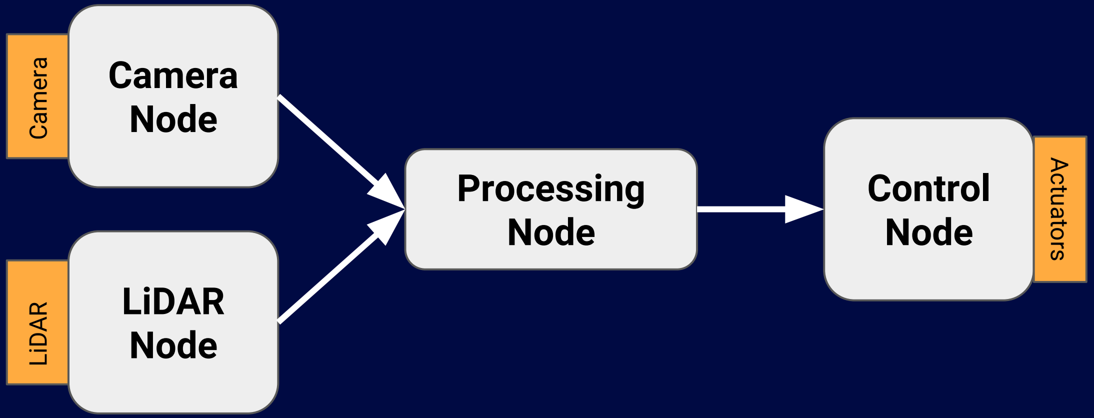
- Flexible and extensible communication
- Abstracts away:
    - Asynchronicity
    - Threading
    - Communication protocols
- Supports wide range of 3rd Party packages
    - Simulation (e.g. Gazebo), SLAM algorithms, image processing, sensor interfacing, etc.
- Great logging and debugging functionality
- Focus on what you care about in robotics research or implementation

ROS Concepts

Publish/Subscribe Communication

Nodes

- One instantiable Agent or piece of your code, an executable
  - Could be a Python script or C++ program, etc.
  - Exists on its own process
- Can receive and send messages from other nodes
- Example Robot:
  - Node for camera
  - Node for LiDAR
  - Node for processing
  - Node for control

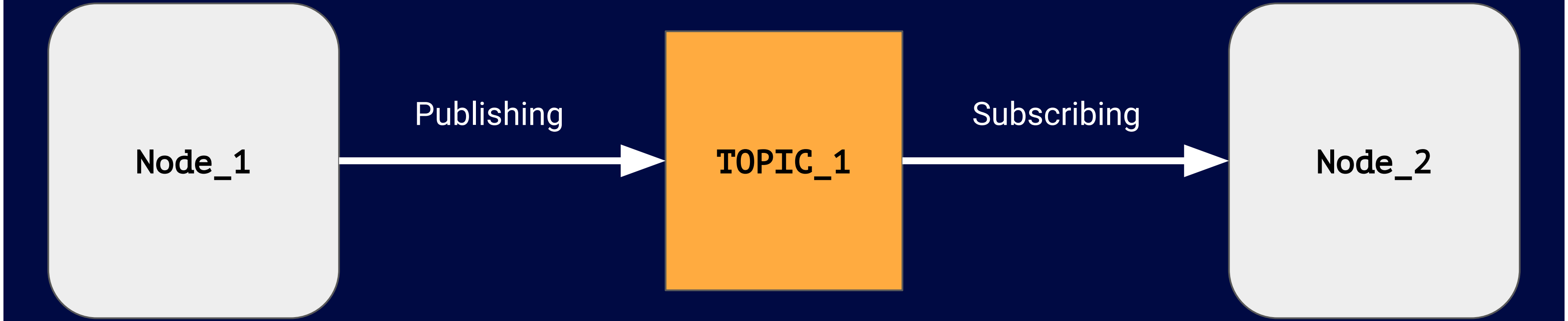


Topics & Messages

- ROS lingo
  - A **node** sends a **message** by **publishing** to a **topic**
  - A **node** receives a **message** by **subscribing** to a **topic**
- Publishing
  - Just publish to a topic whenever you have a message you need to send
- Subscription
  - Accomplished via “callback” functions
    - Callback function is called whenever a new message is received on that topic
    - Frequency agnostic
- Messages
  - Lots of built-in message types
  - Defined as C structs (stored in .msg files), very easy to make custom ones

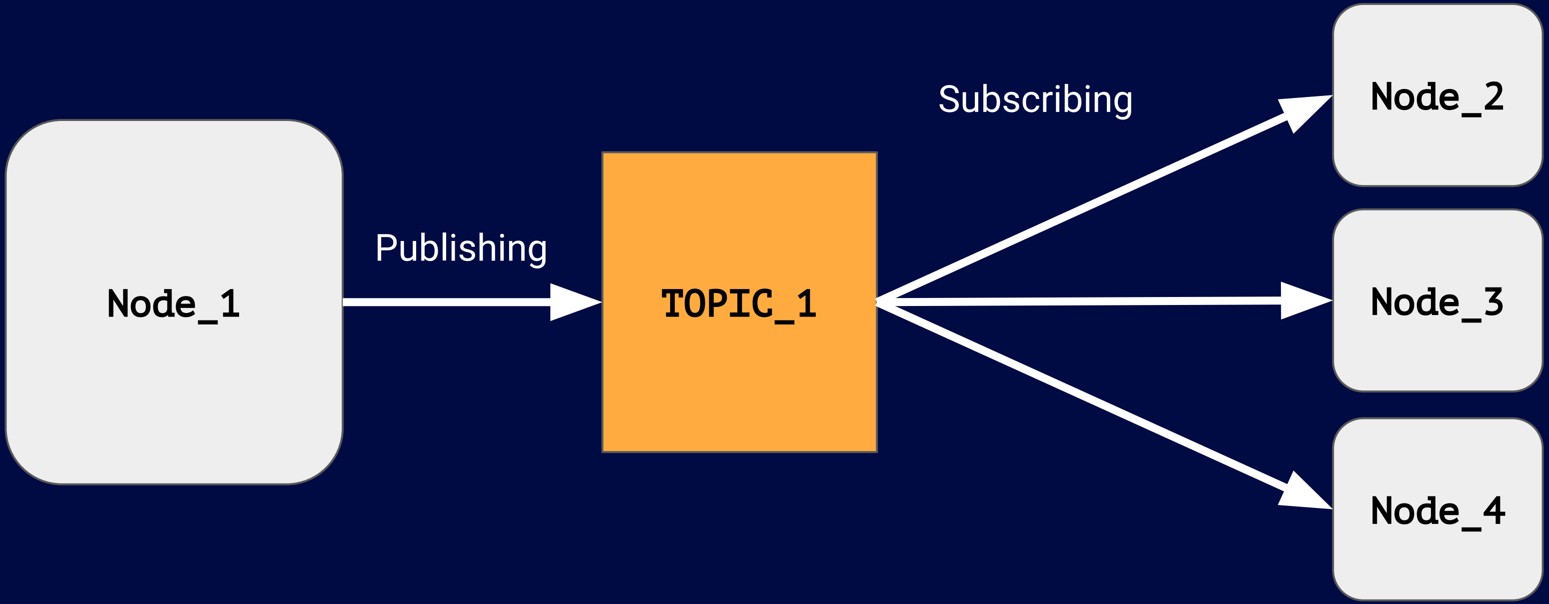
Topics: One-to-One

- Each topic corresponds to only one message type



Topics: One-to-Many

- Same message to each subscriber



Node_1

Publishing

TOPIC_1

Subscribing

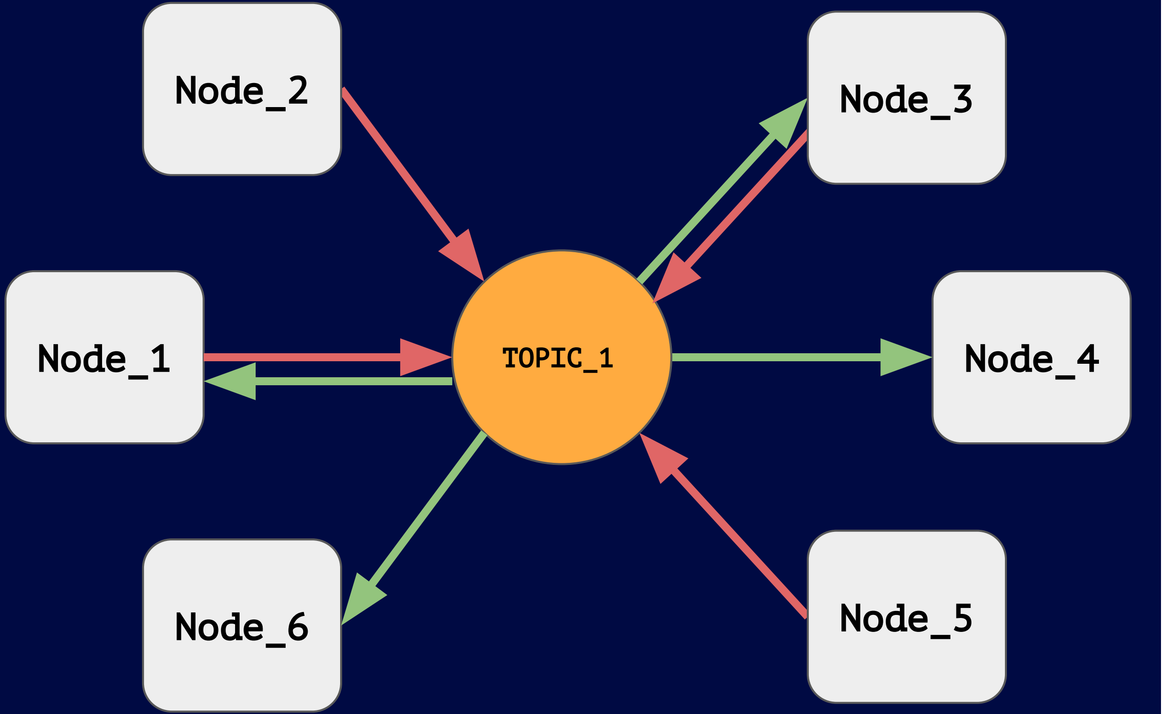
Node_2

Node_3

Node_4

Topics: Two-way

- Time Synchronized
  - Timestamps in each message
- Each published message is broadcasted to all subscribers
  - Includes self if two-way



Roscore

- Think of this as the centralized “manager”
- Must exist before creating nodes and topics
- How to instantiate roscore?
  - Type `roscore` into terminal
- How to close roscore?
  - Ctrl-C in terminal where roscore is running
- Can you have multiple roscore’s open on the same machine?
  - Yes, but they must be on different ports (-p option)
  - Don’t do this unless you have a legitimate need for it
- URI and ports are configured as environment variables for networked communication

Topics & Messages (again)

- Where do topics exist?
  - They exist “globally”, but can be created from inside a node
- How can you delete a topic?
  - You really can’t unless you stop roscore
  - stopping publishing is equivalent to “deleting”
- Naming convention:
  - /camera/image_raw
  - /drone_0/control_seq
  - etc...

Some Common Message Types

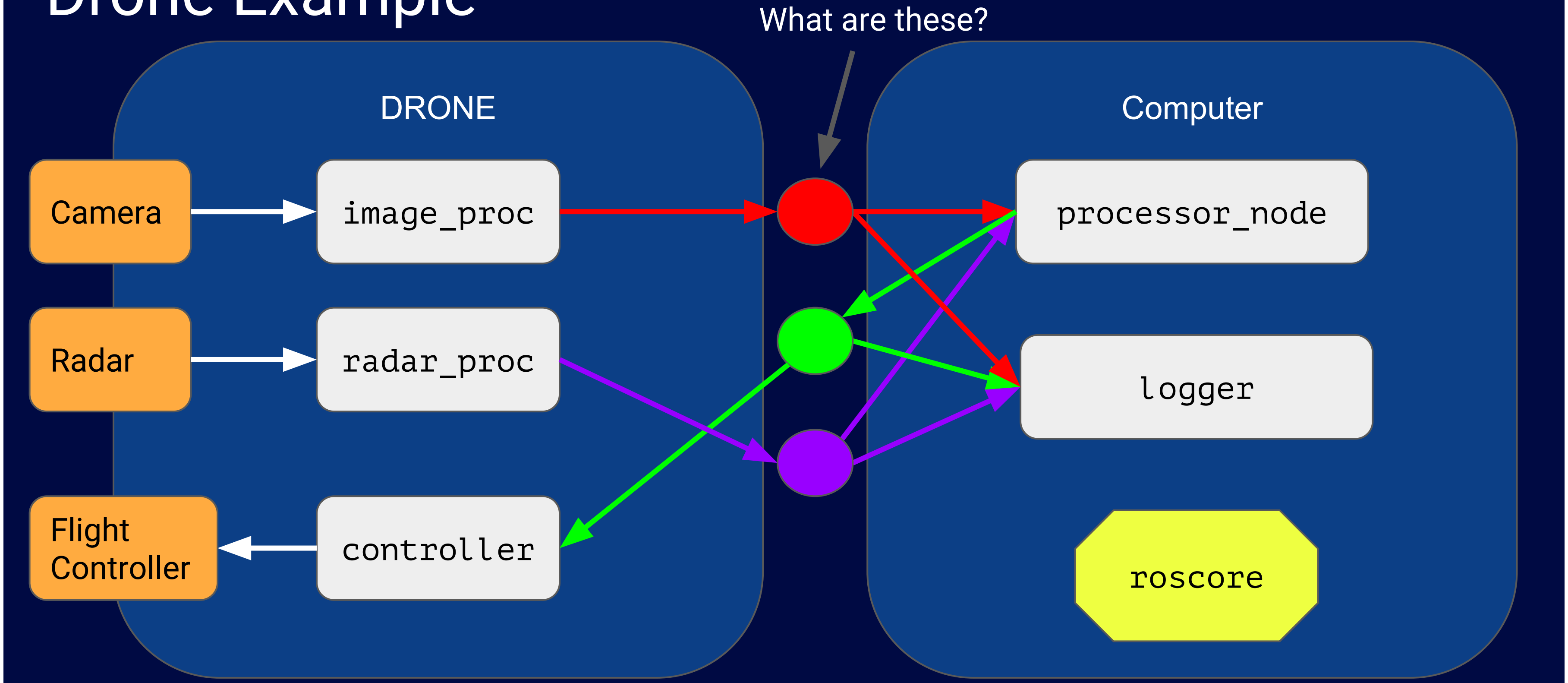
- std_msgs
  - byte, bool, int64, float32, char, etc..
- sensor_msgs
  - image, compressedImage, imu, joy, pointCloud2, temperature
- geometry_msgs
  - pose, twist, quaternion, point, accel, vector3

Example File: **sensor_msgs/Joy.msg**

```
# Reports the state of a joysticks axes and buttons.
Header header                # timestamp for received time
float32[] axes          # the axes measurements from a joystick
int32[] buttons         # the buttons measurements from a joystick
```

UAVs@Berkeley

Drone Example



Working with ROS

Directory Structure, `catkin_ws`

```
-   catkin_ws/
    -   CMakeLists.txt # cmake top level file
    -   devel/         # includes automatically generated setup scripts
    -   build/         # build files for each package after catkin_make
    -   src/           # all the source files for each package go
        -   <my_package 1>/
            -   src
            -   nodes
            -   launch
            -   msgs
            -   srv
        -   <my_package 2>/
```

- To build your package, you add it to the top level `CMakeLists.txt` and run “`catkin_make`” from within catkin_ws

Basic Command Line Tools

- `roscore`
- `rosrun`
  - `“rosrun <package> <node> <args>”`
- `rostopic`
  - `“rostopic hz <topic>”`, `“rostopic list”`
  - `“rostopic echo <topic>”`, `“rostopic type <topic>”`
- `roslaunch`
  - `“roslaunch <package> <launch file> <args>” or “roslaunch <local_launch_file>”`
    - Args in the form: `<arg_name>:=<arg_value>`, each separated by space
  - XML format launch file: Allows you to launch a set of nodes at once
- More tools
  - http://wiki.ros.org/ROS/CommandLineTools

Interactive Demo

ROS Development Studio

Extras

Example roslaunch file:

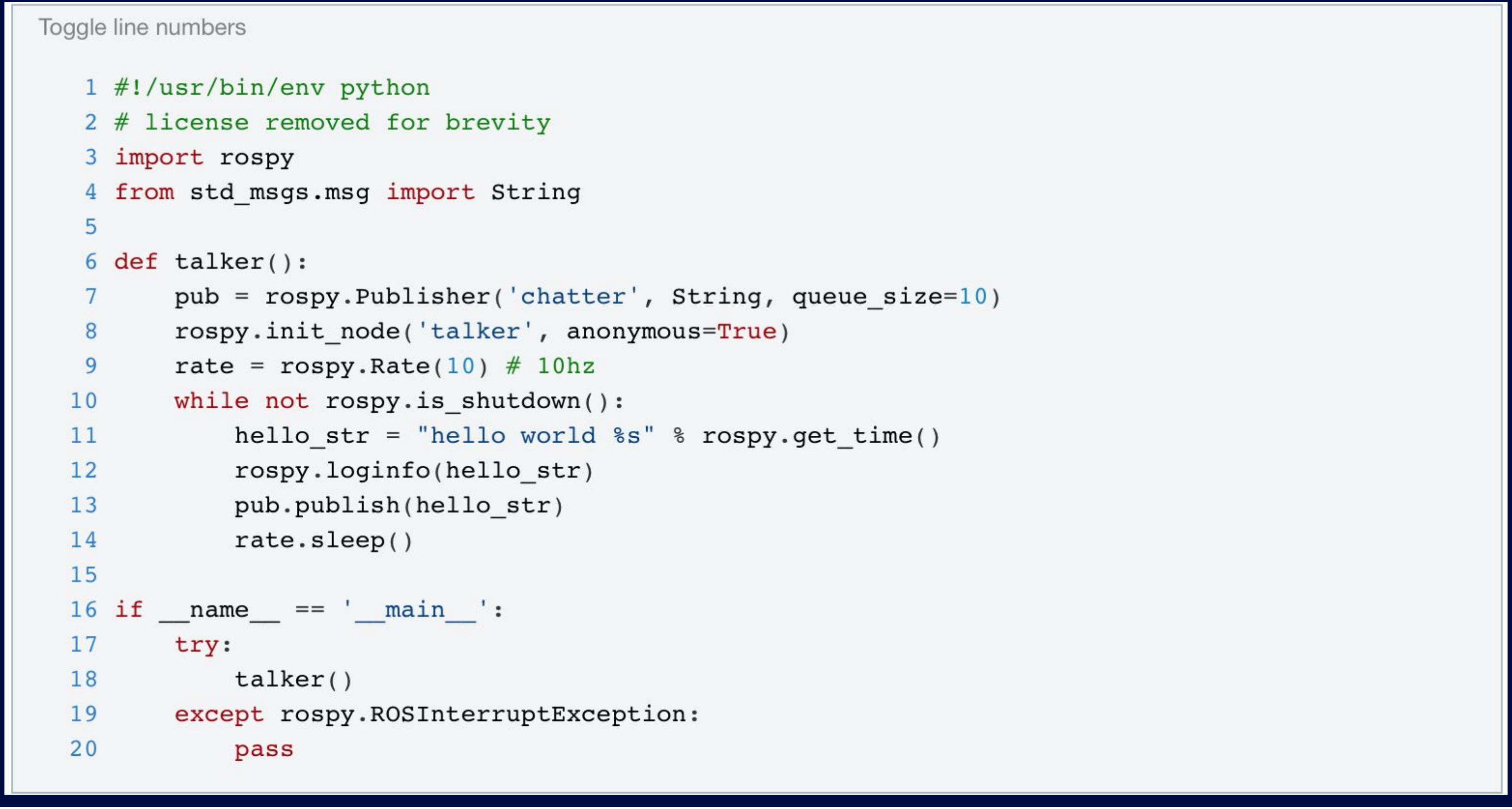
example.launch

```
<launch>

  <!-- This is how you comment →\>

  <node name="add_two_ints_server" pkg="beginner_tutorials" type="add_two_ints_server" />
  <node name="add_two_ints_client" pkg="beginner_tutorials" type="add_two_ints_client" args="$(arg
a) $(arg b)" />

</launch>
```

More Complicated Roslaunch

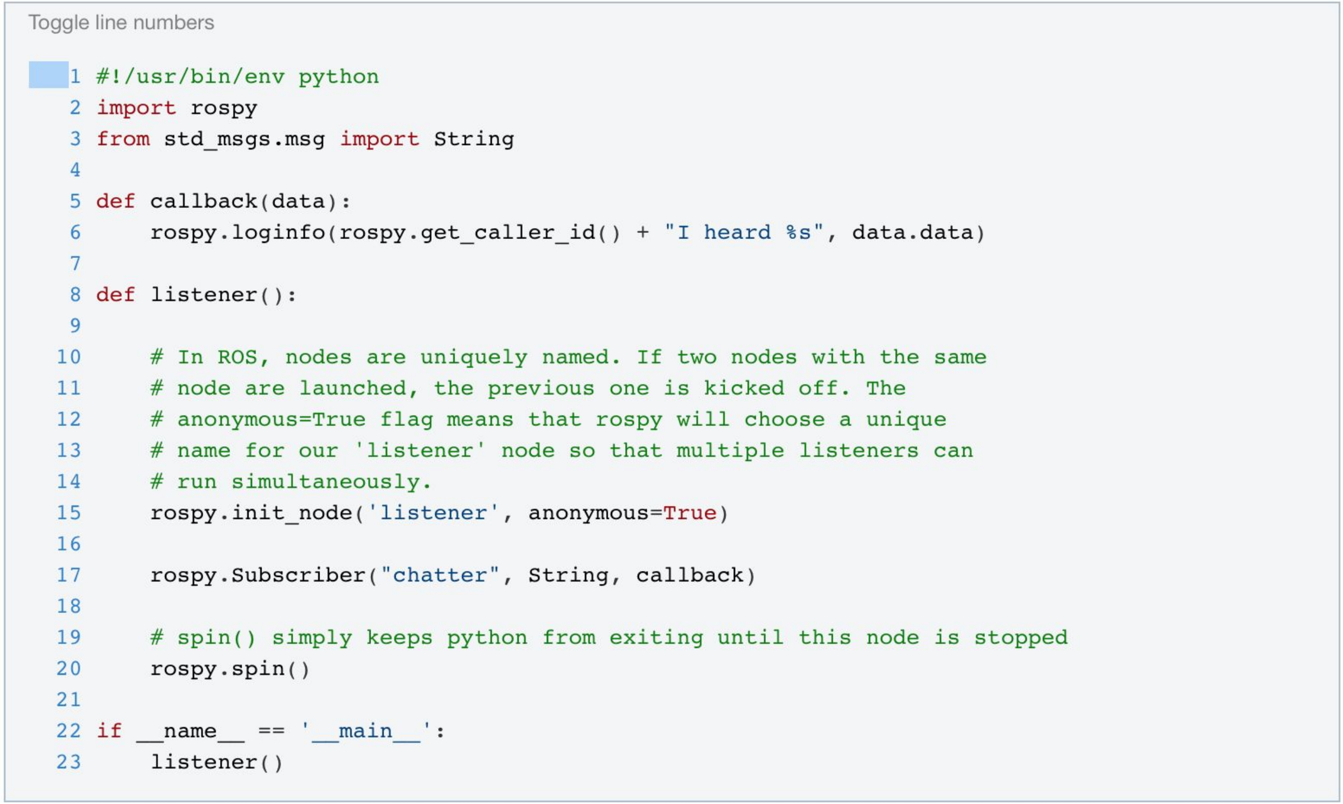
```
<launch>
  <!-- local machine already has a definition by default, This tag overrides the default definition with specific ROS_ROOT and
ROS_PACKAGE_PATH values -->
  <machine name="local_alt" address="localhost" default="true" ros-root="/u/user/ros/ros/" ros-package-path="/u/user/ros/ros-pkg" />
  <!-- 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" />
  <!-- start a group of nodes in the 'wg2' namespace -->
  <group ns="wg2">
    <!-- remap applies to all future statements in this scope. -->
    <remap from="chatter" to="hello"/>
    <node pkg="rospy_tutorials" type="listener" name="listener" args="--test" respawn="true" />
    <node pkg="rospy_tutorials" type="talker" name="talker">
      <!-- set a private parameter for the node -->
      <param name="talker_1_param" value="a value" />
      <!-- 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>
  </group>
</launch>
```

Sample Python ROS Publisher Script



```python
#!/usr/bin/env python
# license removed for brevity
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
```

Sample Python ROS Subscriber Script



```python
#!/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():

    # In ROS, nodes are uniquely named. If two nodes with the same
    # node are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    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()
```

C++

- Code doesn’t fit on these slides, so refer to this link for the tutorial:
    - http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29
- Main takeaways:
    - It’s longer than the python code
    - Use C++ only when your target application benefits from this
        - (i.e. ease of integration or depends heavily on speed)

Services

- Within Node...
- Send **REQUEST** → receive **RESPONSE**
- Think of these as functions that you ask someone else to compute
  - Allows paired messages, essentially
- Example: add 2 ints
  - Call service with the 2 ints as parameters
  - Receive the sum as a response
- Command-line:
  - rosservice
    - “rosservice list”
    - “rosservice call <service> <args>”
    - “rosservice type <service>”

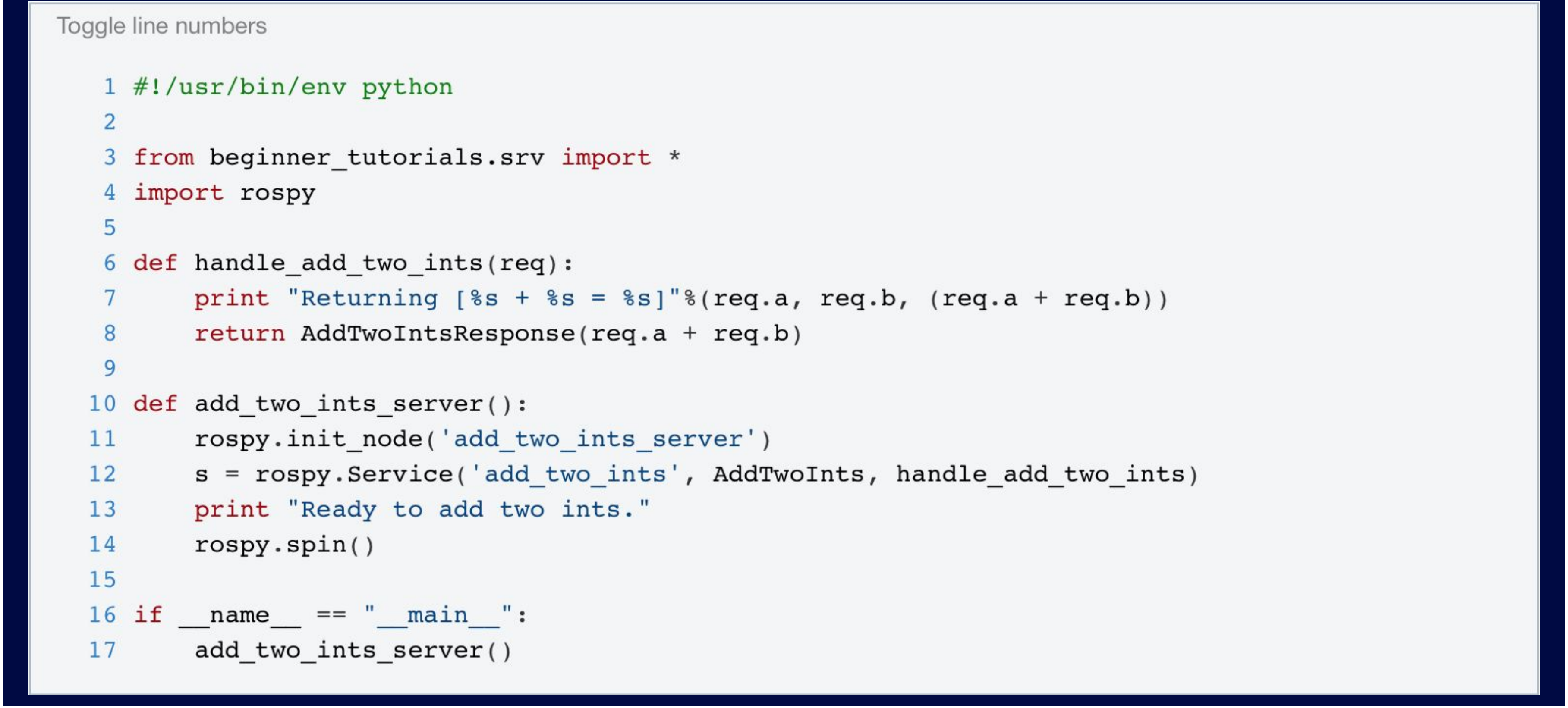
Creating a Service File

- Full tutorial:
  - http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv#Creating_a_srv

AddTwoInts.srv (also creates **AddToIntsResponse.srv**)

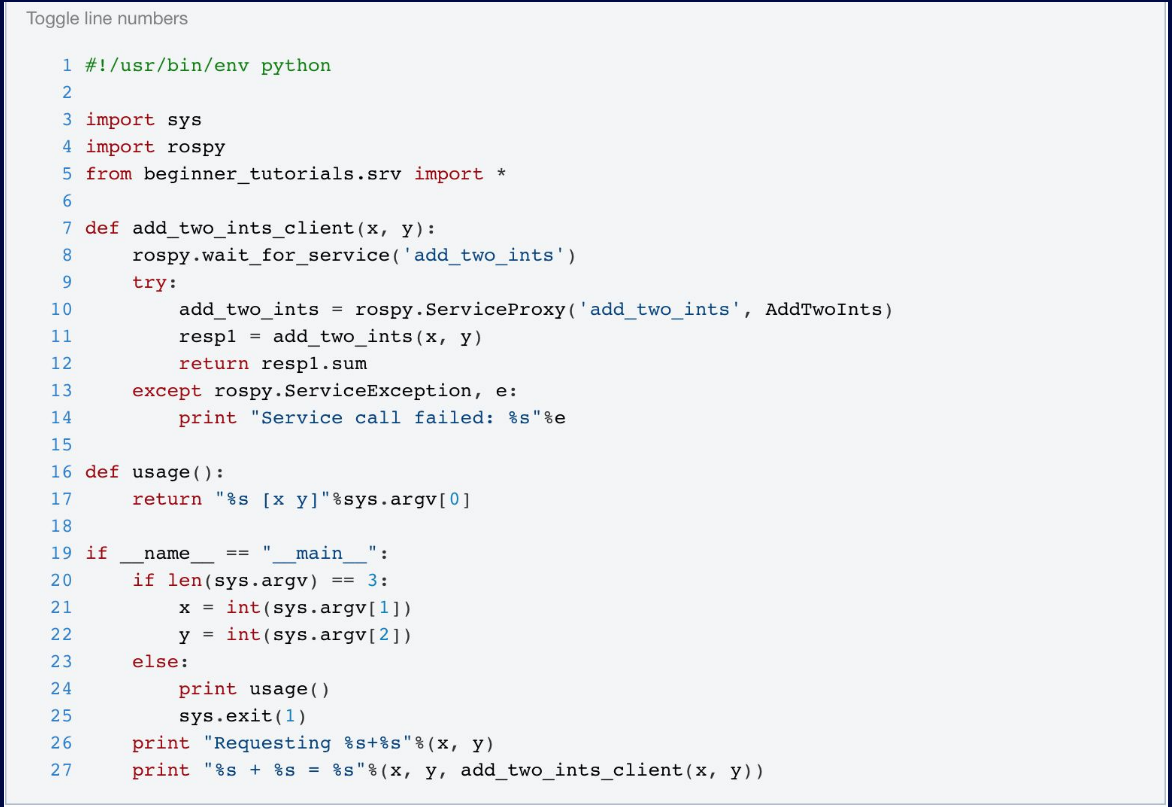
```
int64 a
int64 b
---
int64 sum
```

Example Python Service Node



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

from beginner_tutorials.srv import *
import rospy

def handle_add_two_ints(req):
    print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print "Ready to add two ints."
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

Example Python Service Client Node



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

import sys
import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e

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

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print usage()
        sys.exit(1)
    print "Requesting %s+%s"%(x, y)
    print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

Command Line Example:

```
rosrun <package> add_two_ints_server.py

rosrun beginner_tutorials add_two_ints_client.py 1 3
```

<or>

```
rosrun <package> add_two_ints_server.py

rosservice call /add_two_ints 1 3
```

ROS Takeaways

- Fairly simple framework for communication over multiple networks
- Easy to make your own processing nodes and packages
- Features:
  - Nodes
  - Topics
  - Services
- Awesome command line support
- Works with python and c++
  - Easy to integrate into current programs
- Great Documentation!!!!
- Other readings/Tutorials:
  - http://wiki.ros.org/ROS/Tutorials

Next Steps with ROS

- ROS wiki
  - http://wiki.ros.org
- ROS Environment Variables
  - http://wiki.ros.org/ROS/EnvironmentVariables
- ROS tutorials
  - http://wiki.ros.org/ROS/Tutorials
- Local installation
  - http://wiki.ros.org/ROS/Installation
- Gazebo simulation
  - http://gazebosim.org/tutorials