Quick ROS Intro

CS189 Spring 2019
What is ROS?

- “Robot Operating System”
- Provides a way to communicate with robots
- Allows us to write several programs which work together
- Multilingual support (Can write programs in: C++, Python, LISP, Java, JavaScript, MATLAB, Ruby, Haskell, R, Julia,...)
An Example Problem: Self-Driving Car as a Robot!

- What kind of things do we want our cars to be able to do on our commute?
  - Turn on and back out of the garage
  - Wait for us to get into the car
  - Plan a route to follow
  - Adapt route for traffic changes
  - Avoid potholes, roadkill, or bad drivers
  - Place a phone call
  - Play music
  - Go refuel/recharge when needed

- If each is a program, do they need to be constantly running?
- What kind of sensors and signals would we take in?
ROS Architecture

- Many programs with specific tasks: NODES

<table>
<thead>
<tr>
<th>Task</th>
<th>Info Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Drive_and_Steer</strong></td>
<td>Obstacle position and size (if present)</td>
</tr>
<tr>
<td><strong>Identify_Obstacles</strong></td>
<td>LIDAR data</td>
</tr>
<tr>
<td><strong>Process_LIDAR</strong></td>
<td>Reads in raw LIDAR data and processes it into usable data</td>
</tr>
</tbody>
</table>

**Task**: Sends commands to the car to steer, accelerate, and brake. Safely avoids obstacles if present.

**Task**: Identifies obstacles from LIDAR data. Determines their size and location.

**Task**: Reads in raw LIDAR data and processes it into usable data.
How do Nodes communicate?

- **NODES** communicate by **Publishing** (sending) and **Subscribing** (receiving) messages to a **TOPIC**

  - **Subscribes To:** ‘Obstacles’ topic. Receives messages with obstacle information to determine evasive maneuvers
  - **Publishes To:** ‘Depth Map’ topic. Processes raw LIDAR data from sensor and publishes result

  - **Subscribes To:** ‘Depth Map’ topic. Uses depth map data to identify obstacles
  - **Publishes To:** ‘Obstacles’ topic
More on Publisher/Subscriber

- When a publisher sends a message to a topic, it does not care which node is subscribed to it.

- Likewise, a subscriber will not care which node published to the topic.

- It is possible to have multiple publishers or subscribers to a single topic.
  - When could we require multiple subscribers to the same topic?
  - What about multiple publishers?
Publisher Queues

- By default, a publisher in rospy is **synchronous**; After a message is published, the publisher is blocked from sending another message until:
  - The message has been sent to the topic
  - The topic has sent the message to each of the current subscribers
    - Can you think of why this may not be good?
- It is recommended that we use **asynchronous** publishing, which is defined by `queue_size`.
- For asynchronous, the publisher is still blocked while it is sending the message to the topic, but can publish another message once it is sent
  - A queue of messages can be kept; once it overflows, oldest messages are removed
  - The subscribers can receive the messages from the topic at their own rate
- **Choosing a good `queue_size`** (None = synchronous, Zero = infinite, 1 = Most Recent)
What is ‘roscore’?

- Invisible master that manages communication between nodes
- When a node is started up, it connects to roscore to let it know where it will publish and subscribe to
- roscore only sets up peer-to-peer connections between nodes

![Diagram showing roscore connecting to Process_LIDAR and 'Depth Map']
“So, what will I actually be using....”

- Here are some commands we will use in this class:
  - roscore
    - Starts roscore, which is required for nodes to communicate
  - rosrun
    - Starts a node running
  - roslaunch
    - Starts a collection of specified nodes; if roscore isn’t running, it will start up roscore
  - Crtl + C
    - Stops a program while it is running
Starting and stopping our nodes

- We will be writing our nodes using Python with help of ‘rospy’
- Initializing a node
  - `rospy.init_node("my_node_name")`
- Shutdown sequence
  - `rospy.on_shutdown(self.shutdown)`
    - When the program is shut down, it will run the function described in shutdown
    - For our robot, this may include telling it to stop moving
- Defining a Publisher
  - `pub = rospy.Publisher('topic_name', std_msgs.msg.String, queue_size = 10)`
    - pub can now publish to the topic ‘topic_name’ messages of type String, only keeping 10 most recent messages if they aren’t being received as fast as they are published
    - `pub.publish("Hello World")` #Publishes the message using publisher we defined
- Defining a Subscriber
  - `rospy.Subscriber('topic_name', std_sg.msg.String, process_topic)`
  - When a message is published to ‘topic name’, the information will be processed using the function we define as process_topic
Example Code from Lab 1 (Today):

```python
import rospy

# Import Twist message
from geometry_msgs.msg import Twist

# Initialize node
rospy.init_node('turtlebot_example', anonymous=True)

# Create a publisher
cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)

move = Twist()

# Initial velocity
move.linear.x = 0.5  # drive straight ahead at 0.5 m/s

# Set iteration rate
rate = rospy.Rate(10)  # iterate at 10 Hz

# Main loop
while not rospy.is_shutdown():
    cmd_vel_pub.publish(move)
    rate.sleep()
```

**IF THE PUBLISHER SENDS COMMANDS TOO SLOWLY, THE TURTLEBOT WILL SHUT DOWN AND STOP LISTENING!**
bump_sub = rospy.Subscriber('bumper', BumperEvent, bump_callback)

rate = rospy.Rate(10) # iterate at 10 hz

def bump_callback(data):
    bump = False
    if data.state == BumperEvent.PRESSED:
        bump = True

while not rospy.is_shutdown():
    if bump:
        Move.linear.x = 0 # stop
    rate.sleep()
● Any additional questions:
  ○ Check Canvas for links to documentation resources
  ○ Ask your peers
  ○ Ask a question on Piazza
  ○ Ask your TFs!

Have fun using the Turtlebots and treat them well!