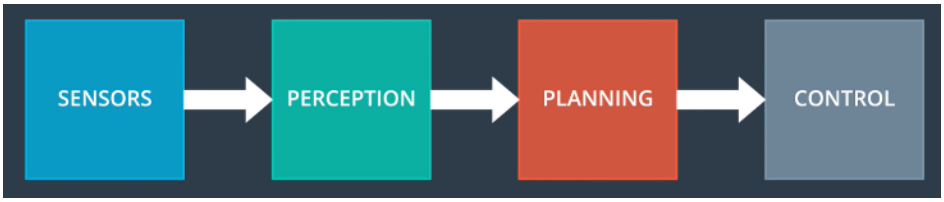


QUIZ QUESTION

Which of the following would most likely belong to the sensor subsystem?

- [x] GPS
- [ ] Lane detection
- [x] IMU (Inertial measurement unit)
- [x] Ultrasonic sensors
- [x] Radar
- [ ] Sensor fusion

QUESTION 1 OF 2

The object detection component of the perception subsystem might process data from which of the following sensors?

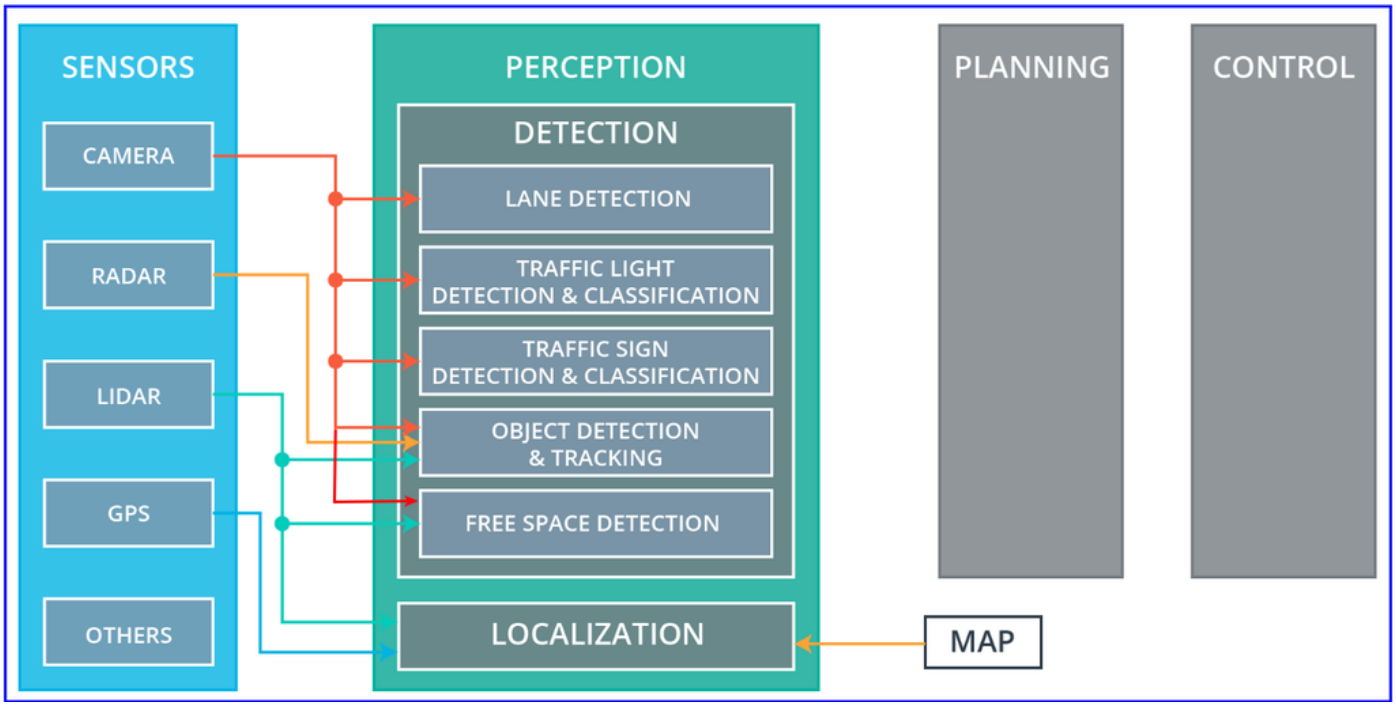
- [x] Camera
- [x] Lidar
- [x] Radar
- [ ] IMU
- [ ] GPS

QUESTION 2 OF 2

The localization component primarily uses data from which of the following sensors?

- [ ] Camera
- [x] Lidar
- [ ] Radar
- [ ] IMU
- [ ] GPS

Localization for fully autonomous vehicles is commonly performed using lidar and a map, although all of these sensors could be used for localization (see here and here for interesting examples).



Planning Subsystem Components

The major components of the planning subsystem components are route planning, prediction, behavioral planning, and path planning.

Route planning

The route planning component is responsible for high-level decisions about the path of the vehicle between two points on a map; for example which roads, highways, or freeways to take. This component is similar to the route planning feature found on many smartphones or modern car navigation systems.

Prediction

The prediction component estimates what actions other objects might take in the future. For example, if another vehicle were identified, the prediction component would estimate its future trajectory.

Behavioral planning

The behavioral planning component determines what behavior the vehicle should exhibit at any point in time. For example stopping at a traffic light or intersection, changing lanes, accelerating, or making a left turn onto a new street are all maneuvers that may be issued by this component.

Trajectory planning

Based on the desired immediate behavior, the trajectory planning component will determine which trajectory is best for executing this behavior.

QUESTION 1 OF 2

Which of the following perception components would provide useful information for the prediction component?

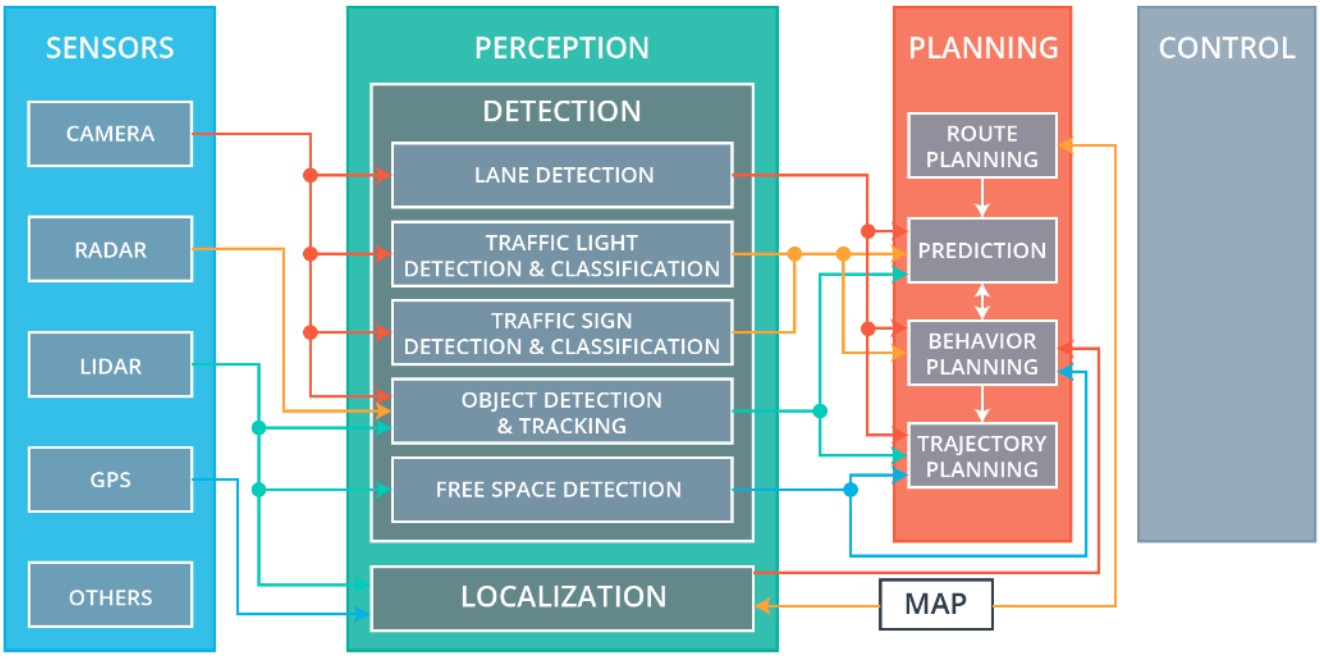
- [ ] Lane detection
- [ ] Free space detection
- [x] Vehicle detection
- [x] Traffic sign classification
- [x] Localization

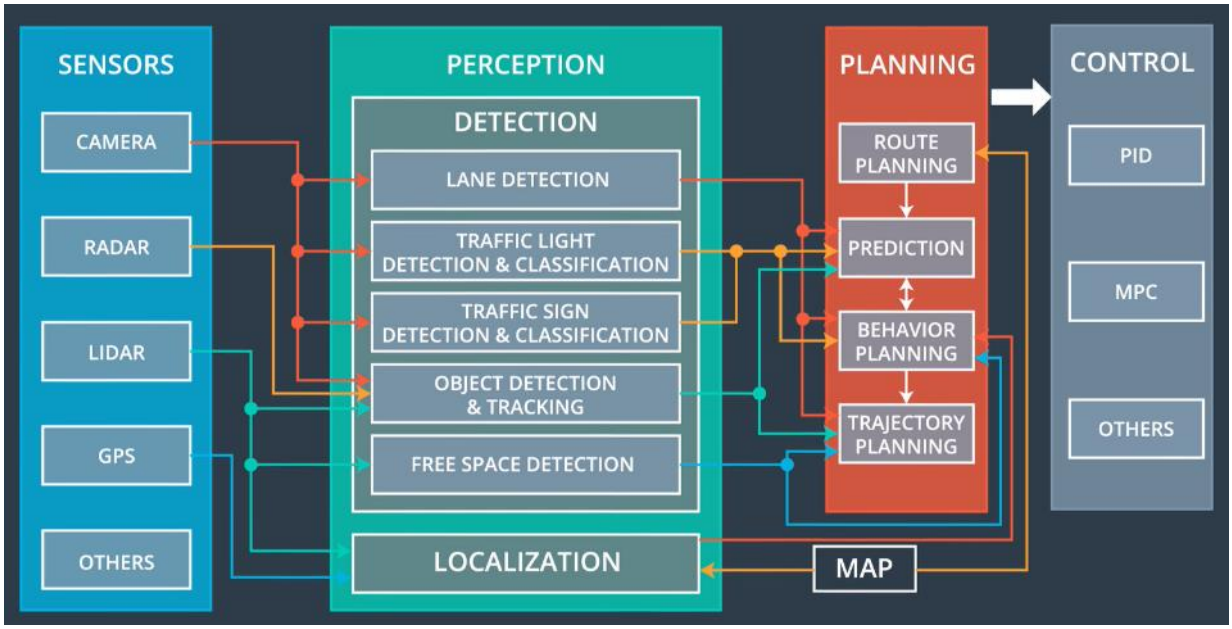
Although localization is unlikely to be used, many of the detection and classification components are needed for a vehicle to make informed predictions, including lane detection, free space detection, vehicle detection, and traffic sign classification.

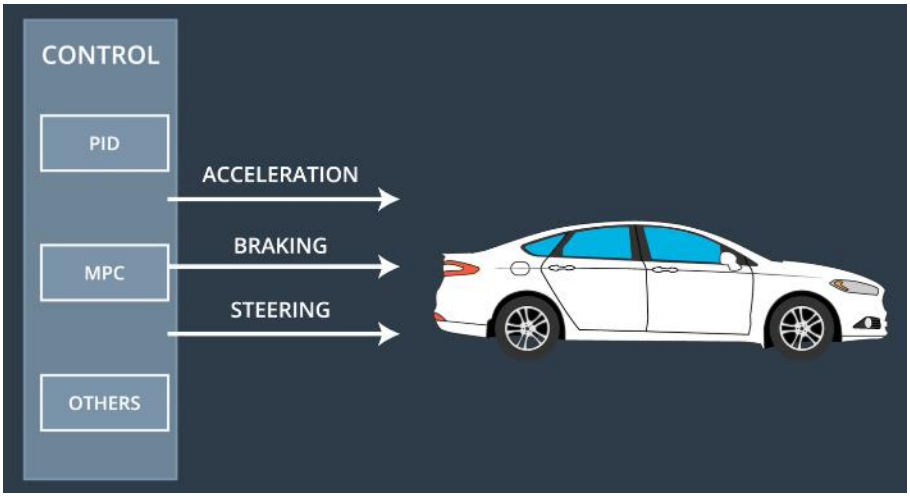
QUESTION 2 OF 2

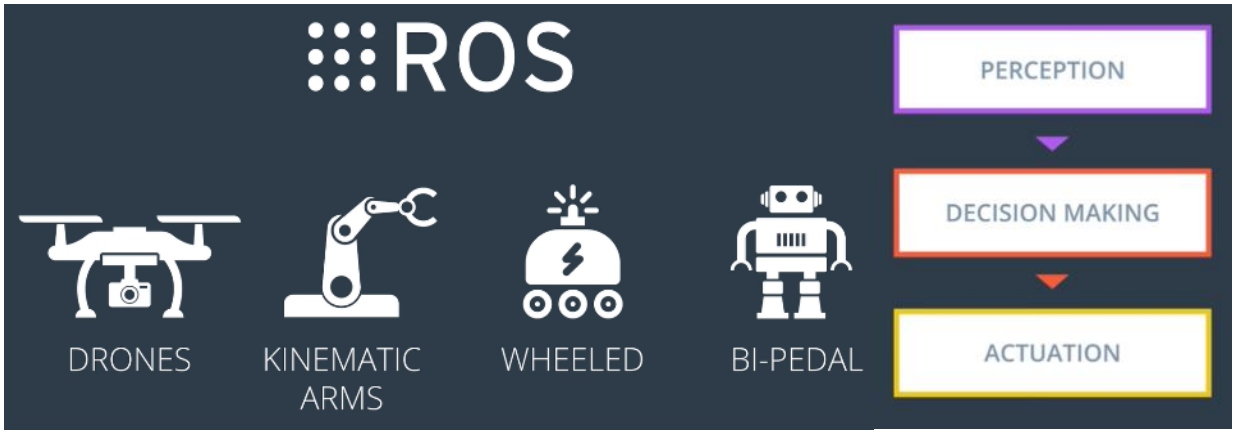
Which of the following perception components would provide useful information for behavioral planning?

- [x] Lane detection
- [x] Free space detection
- [x] Vehicle detection
- [x] Traffic sign classification
- [x] Localization

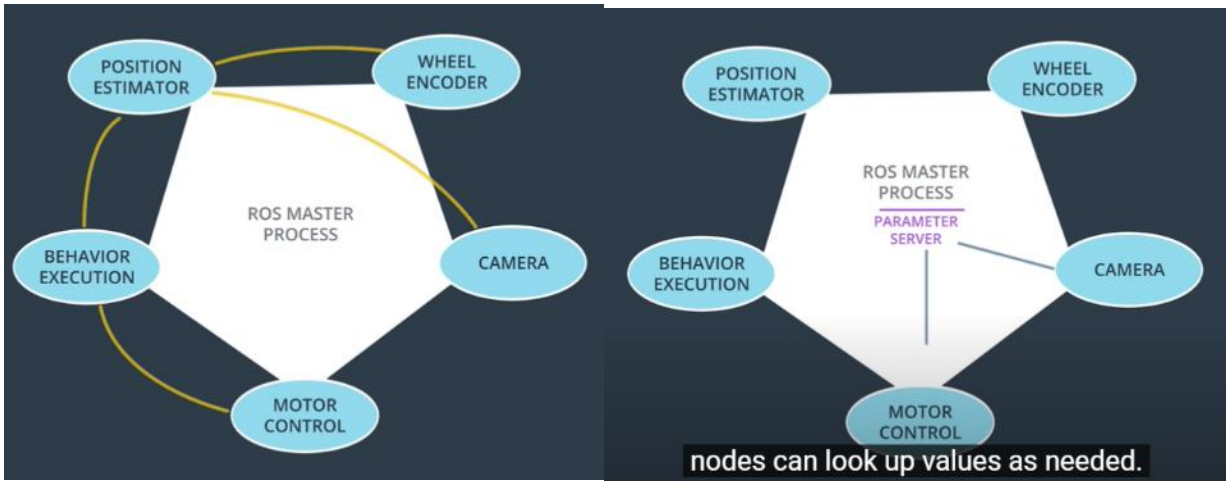



SENSORS
CAMERA
RADAR
LIDAR
GPS
OTHERS
PERCEPTION
DETECTION
LANE DETECTION
TRAFFIC LIGHT DETECTION & CLASSIFICATION
TRAFFIC SIGN DETECTION & CLASSIFICATION
OBJECT DETECTION & TRACKING
FREE SPACE DETECTION
LOCALIZATION
PLANNING
ROUTE PLANNING
PREDICTION
BEHAVIOR PLANNING
TRAJECTORY PLANNING
MAP
CONTROL
PID
MPC
OTHERS

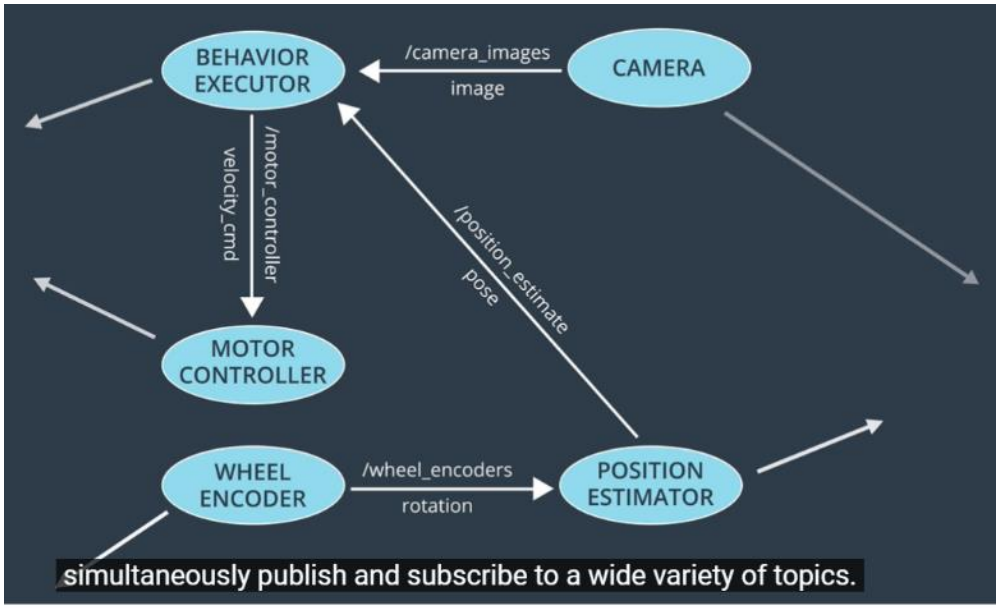

CONTROL
PID
MPC
OTHERS
ACCELERATION
BRAKING
STEERING


ROS
DRONES
KINEMATIC ARMS
WHEELED
BI-PEDAL
PERCEPTION
DECISION MAKING
ACTUATION


PERCEPTION
DECISION MAKING
ACTUATION
WHEEL ENCODER
CAMERA
POSITION ESTIMATOR
BEHAVIOR EXECUTION
MOTOR CONTROL



From publishers to subscribers:



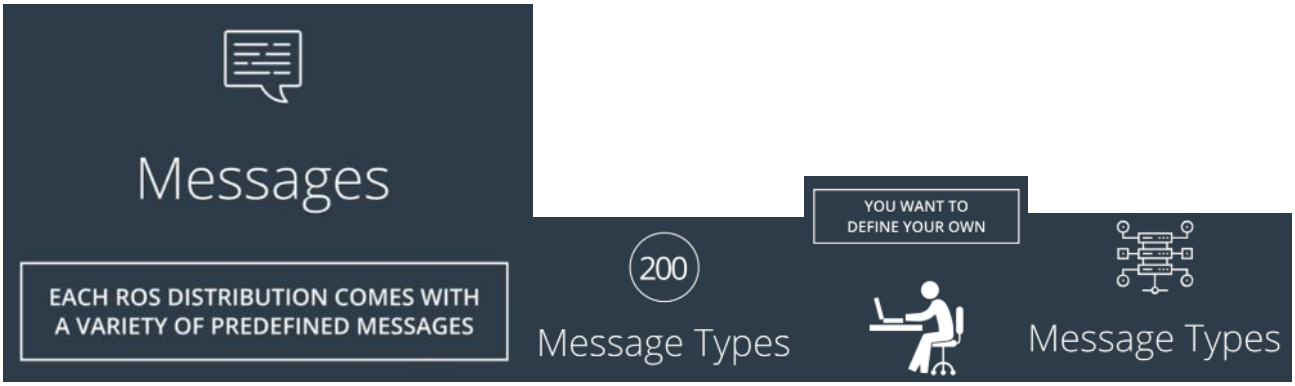
QUIZ QUESTION

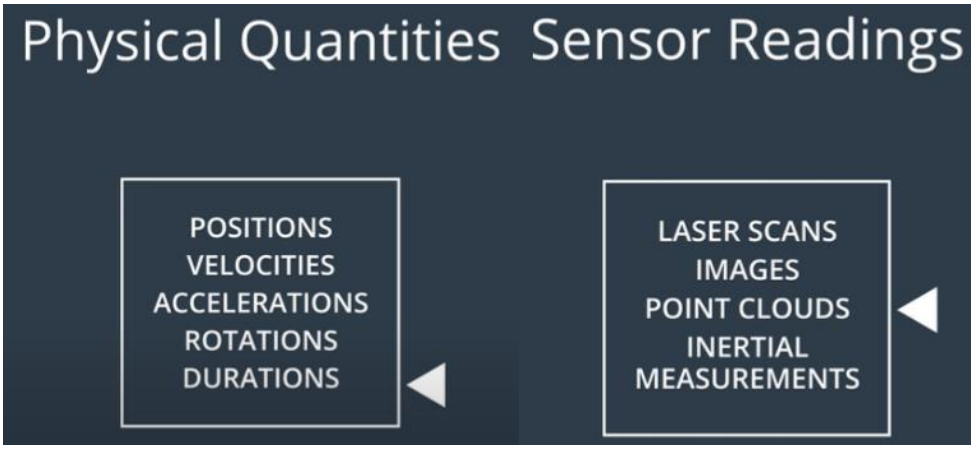
Check the box next to all of the **True** statements

- [x] Robots may be very different in form and function, but they all perform the same high-level tasks of perception, decision making, and actuation.
- [ ] Nodes on a robotic system all communicate through the ROSmaster.
- [x] The parameter server acts as a central repository where nodes on a system can look up parameter values as needed.
- [x] Nodes pass messages to one another via topics, which you can think of as a named bus connecting two nodes.
- [ ] A "pub-sub" architecture is one in which the ROSmaster publishes messages on topics and nodes subscribe to receive them.
- [x] A single node may simultaneously publish and subscribe to many topics.

The ROSmaster allows nodes to locate one another, but after that they connect with each other directly.

A "pub-sub" architecture is one in which each node on the system may publish and subscribe to topics.

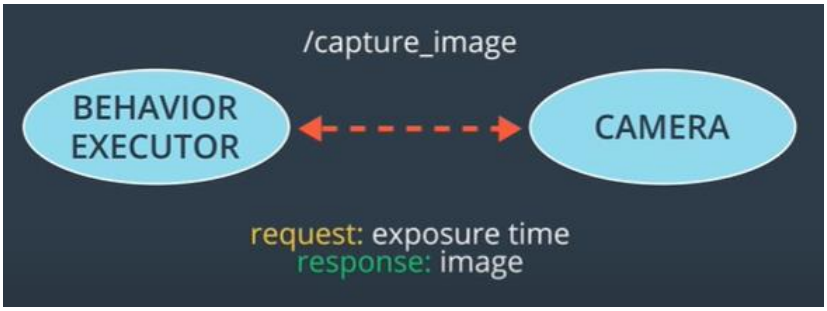




QUIZ QUESTION

Which of the following statements about ROS messages are correct?

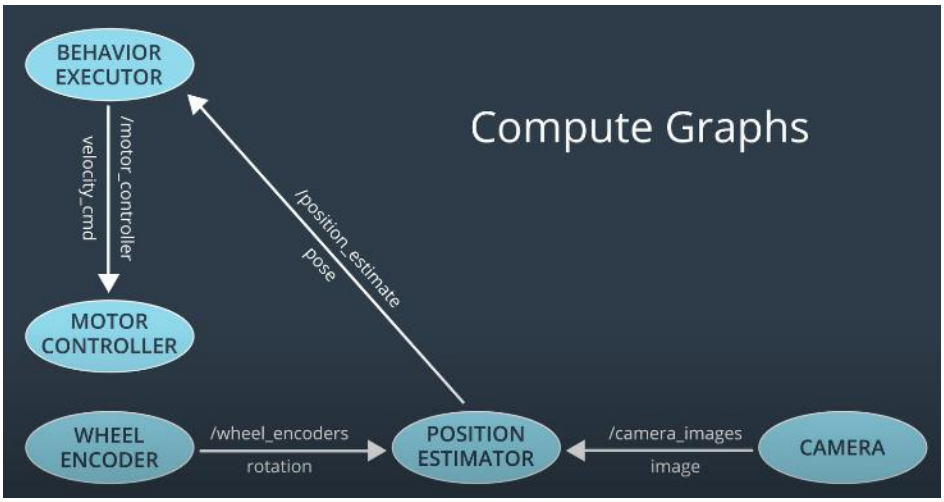
- [ ] Messages are used to pass only text based content between nodes.
- [ ] Your ROS distribution will typically have just one predefined message type.
- [x] Messages come in hundreds of different types and may contain many different types of data.
- [x] In addition to default message types, you can define your own custom message types.

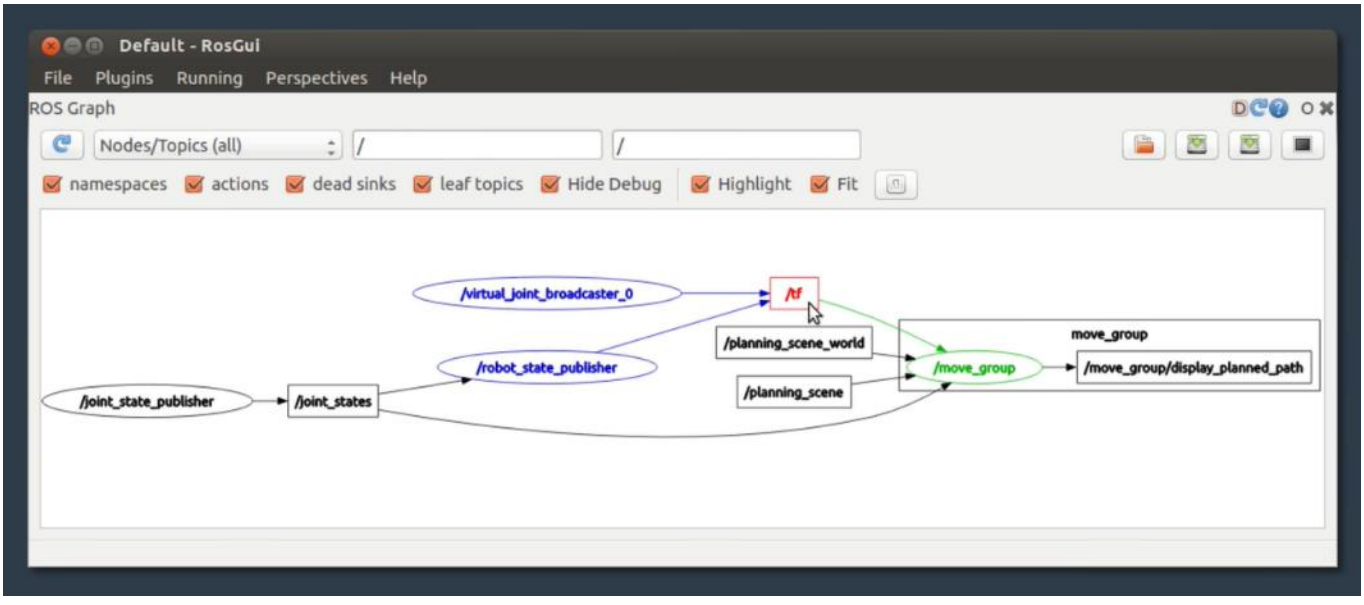


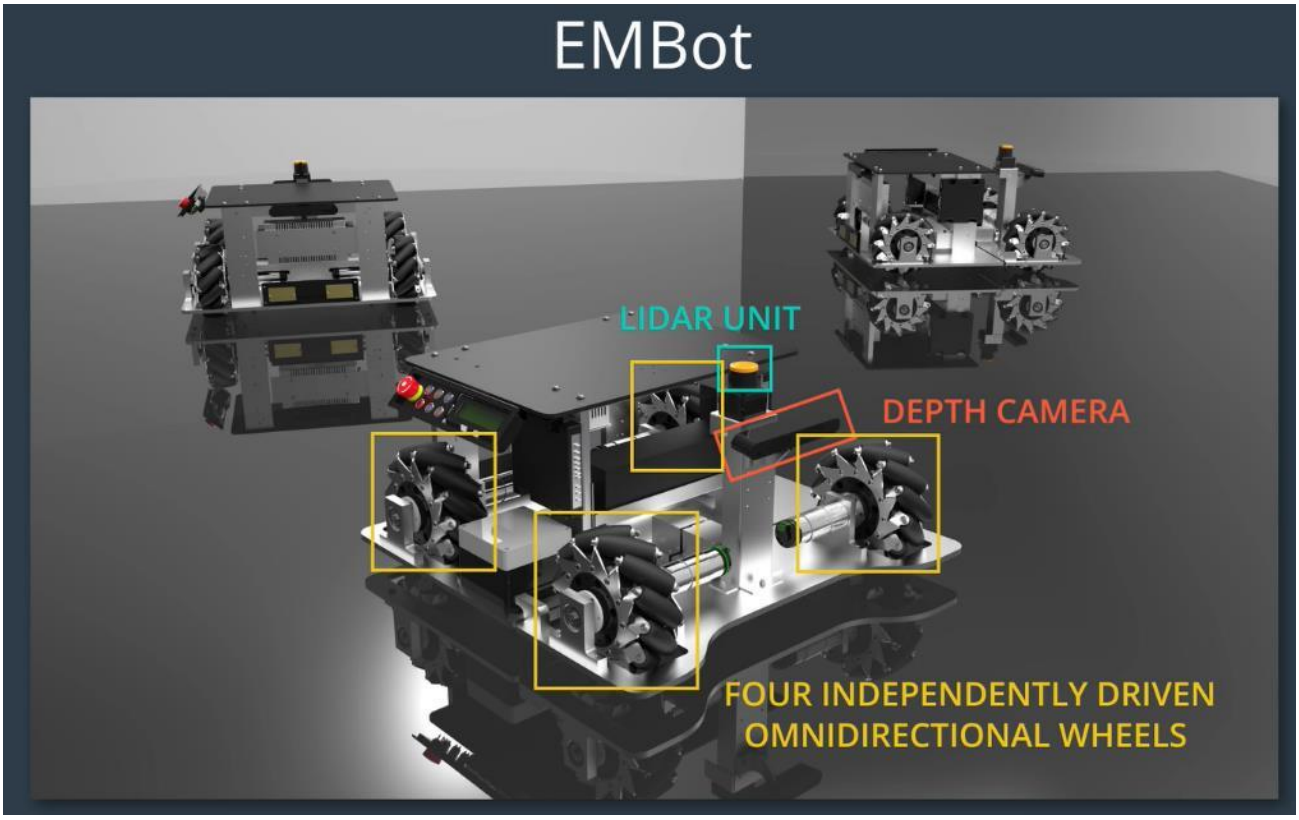
QUIZ QUESTION

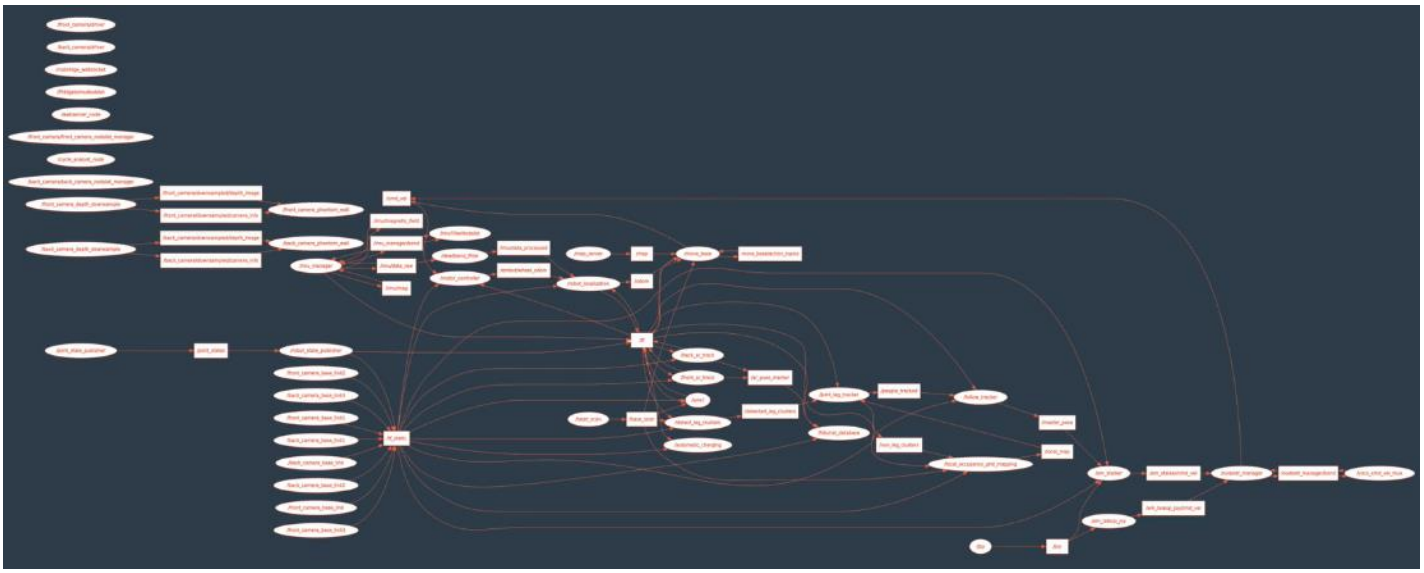
Which of the following statements about services are correct?

- [ ] In ROS terminology, "service" is just another name for "topic"
- [x] Services are similar to topics in that they facilitate the passing of messages between nodes.
- [x] Services use a request-response message passing scheme, rather than the pub-sub method used with topics.
- [x] A request message on a service might actually trigger a new sensor measurement, like a new camera image, with settings like exposure time specified in the message.









/.bashrc is a bash shell which is automatically run everytime a bash is opened

rosrun → execute a node

rosnode list:

`/rosout` This node is launched by roscore. It subscribes to the standard /rosout topic, the topic to which all nodes send log messages.

`$rostopic` list

```
$ rostopic echo /turtle1/cmd_vel
```

`$rostopic` info

rosmsg show geometrymsgs/Twist

rosed geometry_msgs Twist.msg → to get more info like dev comments

Catkin packages

ROS software is organized and distributed into packages, which are directories that might contain source code for ROS nodes, libraries, datasets, and more. Each package also contains a file with build instructions - the CMakeLists.txt file - and a package.xml file with information about the package. Packages enable ROS users to organize useful functionality in a convenient and reusable format.

Catkin workspaces

A catkin workspace is a top-level directory where you build, install, and modify catkin packages. The workspace contains all of the packages for your project, along with several other directories for the catkin system to use when building executables and other targets from your source code.

Create a Catkin Workspace

Step 1: mkdir -p ~/catkin_ws/src

All of the ROS related code you develop throughout this course will reside in your catkin workspace. You only need to create and initialize the workspace once.

First, create the top level catkin workspace directory and a sub-directory named `src` (pronounced source). The top level directory's name is arbitrary, but is often called `catkin_ws` (an abbreviation of catkin_workspace), so we will follow this convention. You can create these two directories with a single command:

```
$ mkdir -p ~/catkin_ws/src
```

Step 2: cd ~/catkin_ws/src

Next, navigate to the `src` directory with the cd command:

```
$ cd ~/catkin_ws/src
```

Step 3: catkin_init_workspace

Now you can initialize the catkin workspace:

```
$ catkin_init_workspace
```

```
robo@robo-virtual-machine:~/catkin_ws/src$ catkin_init_workspace
Creating symlink "/home/robo/catkin_ws/src/CMakeLists.txt" pointing to "/opt/ros/kinetic/share/ca
tkin/cmake/toplevel.cmake"
robo@robo-virtual-machine:~/catkin_ws/src$
```

Let's list the contents of the current directory to see what changed.

```
$ ls -l
```

Notice that a symbolic link (`CMakeLists.txt`) has been created to `/opt/ros/kinetic/share/catkin/cmake/toplevel.cmake`

Step 4: cd ~/catkin_ws

Return to the top level directory,

```
$ cd ~/catkin_ws
```

Step 5: catkin_make

and build the workspace.

Note: you must issue this command from within the top level directory (i.e., within `catkin_ws` NOT `catkin_ws/src`)

```
$ catkin_make
```

Installing Missing Packages Using apt-get

I happen to know that `controller_manager` refers to a ROS package from ROS Control. We can fix this by installing the associated Debian package. If I didn't already know this, I would probably have to rely on a Google search to figure out the exact name of the package required.

```
$ sudo apt-get install ros-kinetic-controller-manager
```

`roslaunch` allows you to do the following

- Launch ROS Master and multiple nodes with one simple command
- Set default parameters on the parameter server
- Automatically re-spawn processes that have died

To use `roslaunch`, you must first make sure that your workspace has been built, and sourced:

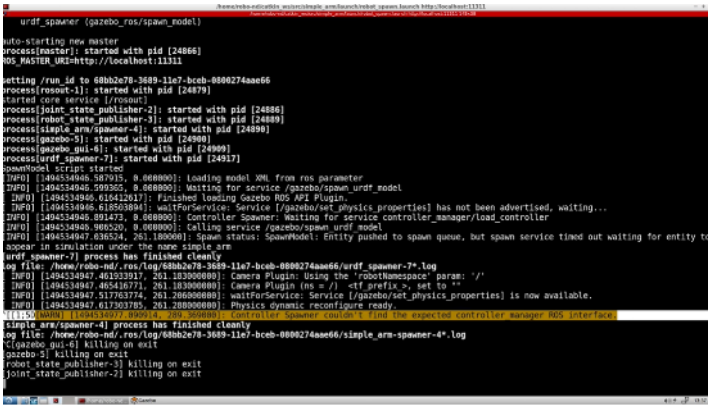
```
$ cd ~/catkin_ws
$ catkin_make
```

Once the workspace has been built, you can source it's setup script:

```
$ source devel/setup.bash
```

With your workspace sourced you can now launch `simple_arm`:

```
$ roslaunch simple_arm robot_spawn.launch
```



The VM has been updated since producing this video. You should expect your build to be error-free. and absent the build errors the speaker addresses in the video.

After the last exercise, you might have noticed the following warning line:

```
The controller spawner couldn't find the expected controller_manager ROS interface.
```

ROS packages have two different types of dependencies: build dependencies, and run dependencies. This error message was due to a missing runtime dependency.

The `rosdep` tool will check for a package's missing dependencies, download them, and install them.

To check for missing dependencies in the `simple_arm` package:

```
$ rosdep check simple_arm
```

Note: In order for the command to work, the workspace must be sourced.

This gives you a list of the system dependencies that are missing, and where to get them.

To have `rosdep` install packages, invoke the following command from the root of the catkin workspace

```
$ rosdep install -i simple_arm
```

Dive Deeper into Packages

Here you'll begin your dive into ROS packages by creating one of your own. All ROS packages should reside under the `src` directory.

Assuming you have already sourced your ROS environment and your catkin workspace (or return to *ROS Workspace* in the "Introduction to ROS" lesson if you forgot), navigate to the src directory:

```
$ cd ~/catkin_ws/src
```

The syntax for creating a catkin package is simply,

```
$ catkin_create_pkg <your_package_name> [dependency1 dependency2 …]
```

```
$ catkin_create_pkg first_package
```

Voilà. You just created your first catkin package! Navigating inside our newly created package reveals that it contains just two files, `CMakeLists.txt` and `package.xml`. This is a minimum working catkin package. It is not very interesting because it doesn't do anything, but it meets all the requirements for a catkin package. One of the main functions of these two files is to describe dependencies and how catkin should interact with them. We won't pay much attention to them right now but in future lessons you will see how to modify them.

I mentioned earlier that ROS packages have a conventional directory structure. Let's take a look at a more typical package.

- scripts (python executables)
- src (C++ source files)
- msg (for custom message definitions)
- srv (for service message definitions)
- include -> headers/libraries that are needed as dependencies
- config -> configuration files
- launch -> provide a more automated way of starting nodes

Other folders may include

- urdf (Universal Robot Description Files)
- meshes (CAD files in .dae (Collada) or .stl (STereoLithography) format)
- worlds (XML like files that are used for Gazebo simulation environments)

Folder Name	**Function**
Executables and Source Files	
scripts	Python executables
src	Source files
Message File Types	
msg	Custom message type definitions. Message file have a .msg file extension.
srv	For call/response type messages. Service messages have a .srv file extension.
launch	Launch files provide a convenient way of running multiple nodes at once. Launch files have a .launch file extension.
config	Configuration files.

ROS Publishers

Before you see the code for `simple_mover`, it may be helpful to see how ROS Publishers work in Python.

Publishers allow a node to send messages to a topic, so that data from the node can be used in other parts of the ROS system. In Python, ROS publishers typically have the following definition format, although other parameters and arguments are possible:

```
pub1 = rospy.Publisher("/topic_name", message_type, queue_size=size)
```

The `"/topic_name"` indicates which topic the publisher will be publishing to. The message_type is the type of message being published on "/topic_name".

ROS publishing can be either synchronous or asynchronous:

- Synchronous publishing means that a publisher will attempt to publish to a topic but may be blocked if that topic is being published to by a different publisher. In this situation, the second publisher is blocked until the first publisher has serialized all messages to a buffer and the buffer has written the messages to each of the topic's subscribers. This is the default behavior of a `rospy.Publisher` if the `queue_size` parameter is not used or set to `None`.
- Asynchronous publishing means that a publisher can store messages in a queue until the messages can be sent. If the number of messages published exceeds the size of the queue, the oldest messages are dropped. The queue size can be set using the `queue_size` parameter.

Once the publisher has been created as above, a `message` with the specified data type can be published as follows:

```
pub1.publish(message)
```

QUIZ QUESTION

Assume that a queued message is typically picked up in an average time of 1/10th of a second with a standard deviation of 1/20th of a second, and your publisher is publishing at a frequency of 10Hz. Of the options below, which would be the best setting for `queue_size`?

- `queue_size=None`
- ✓ `queue_size=2`
- `queue_size=10`
- `queue_size=0`

Choosing a good `queue_size` is somewhat subjective, but since messages are picked up at roughly the same rate they are published, a `queue_size` of 2 provides a little room for messages to queue without being too large.

Adding the scripts directory

In order to create a new node in python, you must first create the `scripts` directory within the `simple_arm` package, as it does not yet exist.

```
$ cd ~/catkin_ws/src/simple_arm/
$ mkdir scripts
```

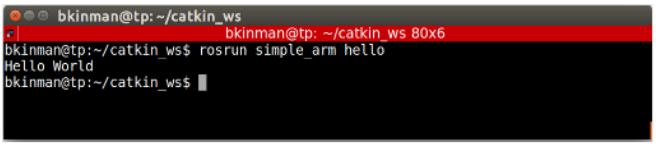
Creating a new script

Once the scripts directory has been created, executable scripts can be added to the package. However, in order for `rosrun` to find them, their permissions must be changed to allow execution. Let's add a simple bash script that prints "Hello World" to the console.

```
$ cd scripts
$ echo '#!/bin/bash' >> hello
$ echo 'echo Hello World' >> hello
```

After setting the appropriate execution permissions on the file, rebuilding the workspace, and sourcing the newly created environment, you will be able to run the script.

```
$ chmod u+x hello
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
$ rosrun simple_arm hello
```



And there you have it! You have now added a script

Creating the empty simple_mover node script

To create the simple_mover node script, you will must simply follow the same basic routine introduced a moment ago.

```
$ cd ~/catkin_ws/src/simple_arm
$ cd scripts
$ touch simple_mover
$ chmod u+x simple_mover
```

ROS Services

Defining services

A ROS service allows request/response communication to exist between nodes. Within the node providing the service, request messages are handled by functions or methods. Once the requests have been handled successfully, the node providing the service sends a message back to the requester node. In Python, a ROS service can be created using the following definition format:

```
service = rospy.Service('service_name', serviceClassName, handler)
```

Here, the `service_name` is the name given to the service. Other nodes will use this name to specify which service they are sending requests to.

The `serviceClassName` comes from the file name where the service definition exists. You will see more about this in the next classroom concept, but each service has a definition provided in an `.srv` file; this is a text file that provides the proper message type for both requests and responses.

The `handler` is the name of the function or method that handles the incoming service message. This function is called each time the service is called, and the message from the service call is passed to the `handler` as an argument. The `handler` should return an appropriate service response message.

Using Services

Services can be called directly from the command line, and you will see an example of this in the upcoming `arm_mover` classroom concepts.

On the other hand, to use a ROS service from within another node, you will define a `ServiceProxy`, which provides the interface for sending messages to the service:

```
service_proxy = rospy.ServiceProxy('service_name', serviceClassName)
```

One way the `ServiceProxy` can then be used to send requests is as follows:

```
msg = serviceClassNameRequest()
#update msg attributes here to have correct data
response = service_proxy(msg)
```

In the code above, a new service message is created by calling the `serviceClassNameRequest()` method. This method is provided by rospy, and its name is given by appending `Request()` to the name used for `serviceClassName`. Since the message is new, the message attributes should be updated to have the appropriate data. Next, the `service_proxy` can be called with the message, and the response stored.

QUIZ QUESTION

Which of the following ROS nodes might best be implemented using a service?

- [ ] A node for an autonomous vehicle that provides lidar data for other nodes to use in localization.
- [x] A node for a lunar rover that shuts down a robotic arm by folding the arm and killing all related processes.
- [x] A node that sets a given parameter on request. For example, a node in turtlesim that sets the pen color in the turtlesim window.
- [x] A node which executes movement for a robotic arm, checking that the arm joints are within specified bounds.

Each of these nodes would likely benefit from the request/response format of a ROS service.

Practical case:

The first node that you will be writing is called `simple_mover`. `simple_mover` does nothing more than publish joint angle commands to `simple_arm`.

After you've developed a basic understanding of the general structure of a ROS Node written in Python, you will be writing another node called `arm_mover`. `arm_mover` provides a service called `safe_move`, which allows the arm to be moved to any position within its workspace which has been deemed to be "safe". The safe zone is bounded by minimum and maximum joint angles, and is configurable via the ROS' parameter server.

The last node you'll write in this lesson is the `look_away` node. This node subscribes to a topic where camera data is being published. When the camera detects an image with uniform color, meaning it's looking at the sky, the node will call the `safe_move` service to move the arm to a new position.

Simple Mover: The Code

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

import math
import rospy
from std_msgs.msg import Float64

def mover():
    pub_j1 = rospy.Publisher('/simple_arm/joint_1_position_controller/command',
                             Float64, queue_size=10)
    pub_j2 = rospy.Publisher('/simple_arm/joint_2_position_controller/command',
                             Float64, queue_size=10)
    rospy.init_node('arm_mover')
    rate = rospy.Rate(10)
    start_time = 0

    while not start_time:
        start_time = rospy.Time.now().to_sec()

    while not rospy.is_shutdown():
        elapsed = rospy.Time.now().to_sec() - start_time
        pub_j1.publish(math.sin(2*math.pi*0.1*elapsed)*(math.pi/2))
        pub_j2.publish(math.sin(2*math.pi*0.1*elapsed)*(math.pi/2))
        rate.sleep()

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

Arm Mover

Description of Arm Mover

In many respects, `arm_mover` is quite similar to `simple_mover`. Like `simple_mover`, it is responsible for commanding the arm to move. However, instead of simply commanding the arm to follow a predetermined trajectory, the `arm_mover` node provides the service `move_arm`, which allows other nodes in the system to send `movement_commands`.

In addition to allowing movements via a service interface, `arm_mover` also allows for configurable minimum and maximum joint angles, by using parameters.

Creating a new service definition

As you learned earlier, an interaction with a service consists of two messages being passed. A request passed to the service, and a response received from the service. The definitions of the request and response message type are contained within .srv files living in the `srv` directory under the package's root.

Let's define a new service for simple_arm. We shall call it `GoToPosition`.

```
$ cd ~/catkin_ws/src/simple_arm/
$ mkdir srv
$ cd srv
$ touch GoToPosition.srv
```

You should now edit `GoToPosition.srv`, so it contains the following:

```
float64 joint_1
float64 joint_2
---
duration time_elapsed
```

Service definitions always contain two sections, separated by a '---' line. The first section is the definition of the request message. Here, a request consists of two float64 fields, one for each of `simple_arm`'s joints. The second section contains is the service response. The response contains only a single field, time_elapsed. The `time_elapsed` field is of type duration, and is responsible for indicating how long it took the arm to perform the movement.

Note: Defining a custom message type is very similar, with the only differences being that message definitions live within the `msg` directory of the package root, have a ".msg" extension, rather than `.srv`, and do not contain the "---" section divider. You can find more detailed information on creating messages and services here, and here, respectively.

Modifying CMakeLists.txt

In order for catkin to generate the python modules or C++ libraries which allow you to utilize messages in your code you must first modify `simple_arm`'s `CMakeLists.txt` (`~/catkin_ws/src/simple_arm/CMakeLists.txt`).

CMake is the build tool underlying catkin, and `CMakeLists.txt` is nothing more than a CMake script used by catkin. If you're familiar with GNU make, and the concept of makefiles, this is a similar concept.

First, ensure that the `find_package()` macro lists `std_msgs` and `message_generation` as required packages. The `find_package()` macro should look as follows:

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

As the names might imply, the `std_msgs` package contains all of the basic message types, and `message_generation` is required to generate message libraries for all the supported languages (cpp, lisp, python, javascript).

Note: In your `CMakeLists.txt`, you may also see `controller_manager` listed as a required package. In actuality this package is not required. It was simply added as a means to demonstrate a build failure in the previous lesson. You may remove it from the list of REQUIRED COMPONENTS if you choose.

Next, uncomment the commented-out `add_service_files()` macro so it looks like this:

```
## Generate services in the 'srv' folder
add_service_files(
   FILES
   GoToPosition.srv
)
```

This tells catkin which files to generate code for.

Lastly, make sure that the `generate_messages()` macro is uncommented, as follows:

```
generate_messages(
   DEPENDENCIES
   std_msgs  # Or other packages containing msgs
)
```

It is this macro that is actually responsible for generating the code. For more information about `CMakeLists.txt` check out this page on the ROS wiki.

Modifying package.xml

Now that the `CMakeLists.txt` file has been covered, you should technically be able to build the project. However, there's one more file which needs to be modified, `package.xml`.

`package.xml` is responsible for defining many of the package's properties, such as the name of the package, version numbers, authors, maintainers, and dependencies.

Right now, we're worried about the dependencies. In the previous lesson you learned about build-time dependencies and run-time package dependencies. When `rosdep` is searching for these dependencies, it's the `package.xml` file that is being parsed. Let's add the `message_generation` and `message_runtime` dependencies.

```
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>message_generation</build_depend>

  <run_depend>controller_manager</run_depend>
  <run_depend>effort_controllers</run_depend>
  <run_depend>gazebo_plugins</run_depend>
  <run_depend>gazebo_ros</run_depend>
  <run_depend>gazebo_ros_control</run_depend>
  <run_depend>joint_state_controller</run_depend>
  <run_depend>joint_state_publisher</run_depend>
  <run_depend>robot_state_publisher</run_depend>
  <run_depend>message_runtime</run_depend>
  <run_depend>xacro</run_depend>
```

Building the package

If you build the workspace successfully, you should now find that a python package containing a module for the new service `GoToPosition` has been created deep down in the `devel` directory.

```
$ cd ~/catkin_ws
$ catkin_make
$ cd devel/lib/python2.7/dist-packages
$ ls
```

After sourcing the newly created `setup.bash`, the new `simple_arm` package has now become part of your `PYTHONPATH` environment variable, and is ready for use!

```
$ env | grep PYTHONPATH
```

Creating the empty arm_mover node script

The steps you take to create the arm_mover node are exactly the same as the steps you took to create the simple_mover script, excepting the actual name of the script itself.

```
$ cd ~/catkin_ws
$ cd src/simple_arm/scripts
$ touch arm_mover
$ chmod u+x arm_mover
```

You can now edit the empty arm_mover script with your favorite text editor.

Let's move onto the code for `arm_mover`.

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

import math
import rospy
from std_msgs.msg import Float64
from sensor_msgs.msg import JointState
from simple_arm.srv import *

def at_goal(pos_j1, goal_j1, pos_j2, goal_j2):
    tolerance = .05
    result = abs(pos_j1 - goal_j1) <= abs(tolerance)
    result = result and abs(pos_j2 - goal_j2) <= abs(tolerance)
    return result

def clamp_at_boundaries(requested_j1, requested_j2):
    clamped_j1 = requested_j1
    clamped_j2 = requested_j2

    min_j1 = rospy.get_param('~min_joint_1_angle', 0)
    max_j1 = rospy.get_param('~max_joint_1_angle', 2*math.pi)
    min_j2 = rospy.get_param('~min_joint_2_angle', 0)
    max_j2 = rospy.get_param('~max_joint_2_angle', 2*math.pi)

    if not min_j1 <= requested_j1 <= max_j1:
        clamped_j1 = min(max(requested_j1, min_j1), max_j1)
        rospy.logwarn('j1 is out of bounds, valid range (%s,%s), clamping to: %s',
                      min_j1, max_j1, clamped_j1)

    if not min_j2 <= requested_j2 <= max_j2:
        clamped_j2 = min(max(requested_j2, min_j2), max_j2)
        rospy.logwarn('j2 is out of bounds, valid range (%s,%s), clamping to: %s',
                      min_j2, max_j2, clamped_j2)

    return clamped_j1, clamped_j2
```

```
def move_arm(pos_j1, pos_j2):
    time_elapsed = rospy.Time.now()
    j1_publisher.publish(pos_j1)
    j2_publisher.publish(pos_j2)

    while True:
        joint_state = rospy.wait_for_message('/simple_arm/joint_states', JointState)
        if at_goal(joint_state.position[0], pos_j1, joint_state.position[1], pos_j2):
            time_elapsed = joint_state.header.stamp - time_elapsed
            break

    return time_elapsed

def handle_safe_move_request(req):
    rospy.loginfo('GoToPositionRequest Received - j1:%s, j2:%s',
                  req.joint_1, req.joint_2)
    clamp_j1, clamp_j2 = clamp_at_boundaries(req.joint_1, req.joint_2)
    time_elapsed = move_arm(clamp_j1, clamp_j2)

    return GoToPositionResponse(time_elapsed)

def mover_service():
    rospy.init_node('arm_mover')
    service = rospy.Service('~safe_move', GoToPosition, handle_safe_move_request)
    rospy.spin()

if __name__ == '__main__':
    j1_publisher = rospy.Publisher('/simple_arm/joint_1_position_controller/command',
                                   Float64, queue_size=10)
    j2_publisher = rospy.Publisher('/simple_arm/joint_2_position_controller/command',
                                   Float64, queue_size=10)

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

Arm Mover: Launch and Interact

Launching the project with the new service

To get the `arm_mover` node, and accompanying `safe_move` service to launch along with all of the other nodes, you will modify `robot_spawn.launch`.

Launch files, when they exist, are located within the `launch` directory in the root of a catkin package. `simple_arm`'s launch file is located in `~/catkin_ws/src/simple_arm/launch`

To get the arm_mover node to launch, simply add the following:

```
<!-- The arm mover node -->
<node name="arm_mover" type="arm_mover" pkg="simple_arm">
  <rosparam>
    min_joint_1_angle: 0
    max_joint_1_angle: 1.57
    min_joint_2_angle: 0
    max_joint_2_angle: 1.0
  </rosparam>
</node>
```

Then, in a new terminal, verify that the node and service have indeed launched.

```
$ rosnode list
$ rosservice list
```

Assuming that both the service (`/arm_mover/safe_move`) and the node (`/arm_mover`) show up as expected (If they've not, check the logs in the `roscore` console), you can now interact with the service using `rosservice`.

To view the camera image stream, you can use the command `rqt_image_view` (you can learn more about rqt and the associated tools here):

```
$ rqt_image_view /rgb_camera/image_raw
```

The camera is displaying a gray image. This is as to be expected, given that it is straight up, towards the gray sky of our gazebo world.

To point the camera towards the numbered blocks on the counter top, we would need to rotate both joint 1 and joint 2 by approximately pi/2 radians. Let's give it a try:

```
$ cd ~/catkin_ws/
$ source devel/setup.bash
$ rosservice call /arm_mover/safe_move "joint_1: 1.57
joint_2: 1.57"
```

Note: `rosservice call` can tab-complete the request message, so that you don't have to worry about writing it out by hand. Also, be sure to include a line break between the two joint parameters.

Upon entering the command, you should be able to see the arm move, and eventually stop, reporting the amount of time it took to move the arm to the console. This is as expected.

What was not expected is the resulting position of the arm. Looking at the `roscore` console, we can very clearly see what the problem was. The requested angle for joint 2 was out of the safe bounds. We requested 1.57 radians, but the maximum joint angle was set to 1.0 radians.

By setting the `max_joint_2_angle` on the parameter server, we should be able to bring the blocks into view the next time a service call is made. To increase joint 2's maximum angle, you can use the command `rosparam`

```
$ rosparam set /arm_mover/max_joint_2_angle 1.57
```

Now we should be able to move the arm such that all of the blocks are within the field of view of the camera:

```
rosservice call /arm_mover/safe_move "joint_1: 1.57
joint_2: 1.57"
```

ROS Subscribers

A Subscriber enables your node to read messages from a topic, allowing useful data to be streamed into the node. In Python, ROS subscribers frequently have the following format, although other parameters and arguments are possible:

```
sub1 = rospy.Subscriber("/topic_name", message_type, callback_function)
```

The `"/topic_name"` indicates which topic the Subscriber should listen to.

The `message_type` is the type of message being published on `"/topic_name"`.

The `callback_function` is the name of the function that should be called with each incoming message. Each time a message is received, it is passed as an argument to `callback_function`. Typically, this function is defined in your node to perform a useful action with the incoming data. Note that unlike service handler functions, the `callback_function` is not required to return anything.

QUIZ QUESTION

Which of the following ROS nodes would likely need a Subscriber?

- [x] A node for an autonomous vehicle that implements pedestrian detection using camera data.
- [ ] A node for a robotic arm that implements a service for moving the arm.
- [ ] A node for a robot which implements a random number generator and publishes new random numbers at a frequency of 50Hz.
- [x] A controller node for a lunar rover which implements the actuation of the throttle and brake given target velocities as input.

Any node that will require a steady stream of input data to accomplish a task will need a subscriber to get the data into the node.

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

import math
import rospy
from sensor_msgs.msg import Image, JointState
from simple_arm.srv import *


class LookAway(object):
    def __init__(self):
        rospy.init_node('look_away')

        self.sub1 = rospy.Subscriber('/simple_arm/joint_states',
                                     JointState, self.joint_states_callback)
        self.sub2 = rospy.Subscriber("rgb_camera/image_raw",
                                     Image, self.look_away_callback)
        self.safe_move = rospy.ServiceProxy('/arm_mover/safe_move',
                                     GoToPosition)

        self.last_position = None
        self.arm_moving = False

        rospy.spin()

    def uniform_image(self, image):
        return all(value == image[0] for value in image)

    def coord_equal(self, coord_1, coord_2):
        if coord_1 is None or coord_2 is None:
            return False
        tolerance = .0005
        result = abs(coord_1[0] - coord_2[0]) <= abs(tolerance)
        result = result and abs(coord_1[1] - coord_2[1]) <= abs(tolerance)
        return result
```

```
    def joint_states_callback(self, data):
        if self.coord_equal(data.position, self.last_position):
            self.arm_moving = False
        else:
            self.last_position = data.position
            self.arm_moving = True

    def look_away_callback(self, data):
        if not self.arm_moving and self.uniform_image(data.data):
            try:
                rospy.wait_for_service('/arm_mover/safe_move')
                msg = GoToPositionRequest()
                msg.joint_1 = 1.57
                msg.joint_2 = 1.57
                response = self.safe_move(msg)

                rospy.logwarn("Camera detecting uniform image. \
                               Elapsed time to look at something nicer:\n%s",
                               response)

            except rospy.ServiceException, e:
                rospy.logwarn("Service call failed: %s", e)



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

Logging overview

In the code for the `simple_mover`, `arm_mover`, and `look_away` nodes, you may have noticed logging statements such as:

```
rospy.logwarn('j1 is out of bounds, valid range (%s,%s), clamping to: %s',
              min_j1, max_j1, clamped_j1)
```

and

```
rospy.loginfo('GoToPositionRequest Received - j1:%s, j2:%s',
              req.joint_1, req.joint_2)
```

Logging statements allow ROS nodes to send messages to a log file or the console. This allows errors and warnings to be surfaced to the user, or log data to be used later for debugging.

By default all logging messages for a node are written to the node's log file which can be found in `~/.ros/log` or `ROS_ROOT/log` . If `roscore` is running, you can use `roscd` to find log file directory by opening a new terminal window and typing:

```
roscd log
```

In this directory, you should see directories from runs of your ROS code, along with a `latest` directory with log files from the most recent run.

Below, we'll show some of the options available for logging different types of messages, filtering messages, and changing how messages are surfaced to a user.

Logging levels and outputs

Rospy has several message levels and provides a variety of options for how to display or store these messages:

```
rospy.logdebug(...)
rospy.loginfo(...)
rospy.logwarn(...)
rospy.logerr(...)
rospy.logfatal(...)
```

All levels of logging messages are recorded in ROS log files, but some message levels may also be sent to Python `stdout`, Python `stderr`, or the ROS topic `/rosout`.

The `loginfo` messages are written to Python's `stdout`, while `logwarn`, `logerr`, and `logfatal` are written to Python's `stderr` by default. Additionally, `loginfo`, `logwarn`, `logerr`, and `logfatal` are written to `/rosout`.

The following table summarizes the default locations log messages are written to (source here):

	Debug	Info	Warn	Error	Fatal
stdout		X			
stderr			X	X	X
log file	X	X	X	X	X
/rosout		X	X	X	X

Modifying message level sent to /rosout

Although `logdebug` messages are not written to `/rosout` by default, it is possible to modify the level of logging messages written to `/rosout` to display them there, or change the level of logging messages written to `/rosout` to be more restrictive. To do this you must set the `log_level` attribute within the `rospy.init_node` code. For example, if you'd like to allow lodebug messages to be written to `/rosout`, that can be done as follows:

```
rospy.init_node('my_node', log_level=rospy.DEBUG)
```

Other possible `rospy` options for `log_level` are `INFO`, `WARN`, `ERROR`, and `FATAL`.

Modifying display of messages sent to `stdout` and `stderr`

It is also possible to change how messages to `stdout` and `stderr` are displayed or logged. Within a package's `.launch` file, the `output` attribute for a node tag can be set to `"screen"` or `"log"`. The following table summarizes how the different output options change the display of the node's `stdout` and `stderr` messages:

	`stdout`	`stderr`
`"screen"`	screen	screen
`"log"`	log	screen and log

For example, setting `output="screen"` for the `look_away` node in `robot_spawn.launch` will display both `stdout` and `stderr` messages in the screen:

```
<!-- The look away node -->
<node name="look_away" type="look_away" pkg="simple_arm" output="screen"/>
```

If the `output` attribute is left empty, the default is `"log"`.