In this lesson, you’ll be learning how to write nodes in Python.

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.

**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)

For more information about ROS publishers, see [**the documentation here**](http://docs.ros.org/kinetic/api/rospy/html/rospy.topics.Publisher-class.html).

You will now go through the process of implementing your first ROS node in python. This node is called simple\_mover. As it’s name implies, this node only has one responsibility, and that is to command joint movements for simple\_arm.

To do so, it must publish joint angle command messages to the following topics:

| **Topic Name** | **/simple\_arm/joint\_1\_position\_controller/command** |
| --- | --- |
| **Message Type** | std\_msgs/Float64 |
| **Description** | Commands joint 1 to move counter-clockwise, units in radians |
| **Topic Name** | **/simple\_arm/joint\_2\_position\_controller/command** |
| **Message Type** | std\_msgs/Float64 |
| **Description** | Commands joint 2 to move counter-clockwise, units in radians |

**Note**: If you no longer have the catkin workspace from the previous lesson, you can download a copy of it [**here**](https://github.com/udacity/simple_arm_01). Alternately, If you’d prefer to skip to the punch, you can download the entire, complete simple\_arm package from [**here**](https://github.com/udacity/simple_arm).

## 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

Text

Description automatically generated

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

You can now edit the empty simple\_mover script with your favorite text editor.

Let’s write the code!

# Simple Mover: The Code

Below is the complete code for the simple\_mover node, in it’s entirety, followed by a step-by-step explanation of what is happening. You can copy and paste this code into the simple\_mover script you created in the ~/catkin\_ws/src/simple\_arm/scripts/ directory like this:

First, open a new terminal, next:

$ cd ~/catkin\_ws/src/simple\_arm/scripts/

$ nano simple\_mover

You have opened the simple\_mover script with the **nano** editor, now copy and paste the code below into the script and use ctrl-x followed by y then enter to save the script.

## simple\_mover

*#!/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**

## The code: Explained

*#!/usr/bin/env python*

**import** math

**import** rospy

rospy is the official Python client library for ROS. It provides most of the fundamental functionality required for interfacing with ROS via Python. It has interfaces for creating Nodes, interfacing with Topics, Services, Parameters, and more. It will certainly be worth your time to check out the API documentation [**here**](http://docs.ros.org/kinetic/api/rospy/html/). General information about rospy, including other tutorials may be found on the [**ROS Wiki**](http://wiki.ros.org/rospy_tutorials/Tutorials/WritingPublisherSubscriber).

**from** std\_msgs.msg **import** Float64

From the std\_msgs package, we import Float64, which is one of the primitive message types in ROS. The [**std\_msgs**](http://wiki.ros.org/std_msgs) package also contains all of the other primitive types. Later on in this script, we will be publishing Float64 messages to the position command topics for each joint.

**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)

At the top of the mover function, two publishers are declared, one for joint 1 commands, and one for joint 2 commands. Here, the queue\_size parameter is used to determine the maximum number messages that may be stored in the publisher queue before messages are dropped. More information about this parameter can be found [**here**](http://wiki.ros.org/rospy/Overview/Publishers%20and%20Subscribers#queue_size:_publish.28.29_behavior_and_queuing).

rospy.init\_node('arm\_mover')

Initializes a client node and registers it with the master. Here “arm\_mover” is the name of the node. init\_node() must be called before any other rospy package functions are called. The argument anonymous=True makes sure that you always have a unique name for your node

rate = rospy.Rate(10)

The rate object is created here with a value of 10 Hertz. Rates are used to limit the frequency at which certain loops spin in ROS. Choosing a rate which is too high may result in unnecessarily high CPU usage, while choosing a value too low could result in high overall system latency. Choosing sensible values for all of the nodes in a ROS system is a bit of a fine-art.

start\_time = 0

**while** **not** start\_time:

start\_time = rospy.Time.now().to\_sec()

start\_time is used to determine how much time has elapsed. When using ROS with simulated time (as we are doing here), rospy.Time.now() will initially return 0, until the first message has been received on the /clock topic. This is why start\_time is set and polled continuously until a nonzero value is returned (more information [**here**](http://wiki.ros.org/rospy/Overview/Time)).

**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()

This the main loop. Due to the call to rate.sleep(), the loop is traversed at approximately 10 Hertz. Each trip through the body of the loop will result in two joint command messages being published. The joint angles are sampled from a sine wave with a period of 10 seconds, and in magnitude from [-\pi/2, +\pi/2−*π*/2,+*π*/2]. When the node receives the signal to shut down (either from the master, or via SIGINT signal in a console window), the loop will be exited.

**if** \_\_name\_\_ == '\_\_main\_\_':

**try**:

mover()

**except** rospy.ROSInterruptException:

**pass**

If the name variable is set to “**main**”, indicating that this script is being executed directly, the mover() function will be called. The try/except blocks here are significant as rospy uses exceptions extensively. The particular exception being caught here is the ROSInterruptException. This exception is raised when the node has been signaled for shutdown. If there was perhaps some sort of cleanup needing to be done before the node shuts down, it would be done here. More information about rospy exceptions can be found [**here**](http://wiki.ros.org/rospy/Overview/Exceptions).

## Running simple\_mover

Assuming that your workspace has recently been built, and it’s setup.bash has been sourced, you can launch simple\_arm as follows:

$ cd ~/catkin\_ws

$ roslaunch simple\_arm robot\_spawn.launch

Once ROS Master, Gazebo, and all of our relevant nodes are up and running, we can finally launch simple\_mover. To do so, open a new terminal and type the following commands:

$ cd ~/catkin\_ws

$ source devel/setup.bash

$ rosrun simple\_arm simple\_mover

Below is a video showing you what the expected movements should look like.

Congratulations! You’ve now written your first ROS node!

# ROS Services

Now that you've written your first ROS node, you've seen how publishing to a topic works, and you were able control the robotic arm by publishing to the /simple\_arm/joint\_2\_position\_controller/command topic. Next up, we'll see another node called arm\_mover which implements the safe\_move service to allow the arm to be controlled with service calls.

### 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.

For other ways to pass data to service\_proxy, see the ROS documentation [**here**](http://wiki.ros.org/rospy/Overview/Services).

You’ve written your first ROS node! This was no trivial task. You’ve had to learn quite a few things to get to this point. But before you will be prepared for the final project, we have some more ground to cover.

Namely, we still need to cover:

* Custom message generation
* Services
* Parameters
* Launch Files
* Subscribers
* Logging

In order to gain an understanding of some of the above, you will be writing another node called 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**](http://wiki.ros.org/msg), and [**here**](http://wiki.ros.org/srv), 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**](http://wiki.ros.org/catkin/CMakeLists.txt) 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>

You are now ready to build the package! For more information about package.xml, check out the [**ROS Wiki**](http://wiki.ros.org/catkin/package.xml).

## 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

# Arm Mover: The Code

# arm\_mover

*#!/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**

## The code: explained

*#!/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** \*

The imported modules for arm\_mover are the same as simple\_arm, with the exception of two new imports. Namely, the JointState message, and the simple\_arm.srv module.

JointState messages are published to the /simple\_arm/joint\_states topic, and are used for monitoring the position of the arm.

The simple\_arm package, and the srv module are automatically generated by catkin as part of the build process.

**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

This function returns True if the joint positions are close to the goals. When taking measurements from sensors in the real world, there will always be some amount of noise. The same is true of the joint positions reported by the gazebo simulator. If both joint positions are within .05 radians of the goal, True is returned.

**def** **clamp\_at\_boundaries**(requested\_j1, requested\_j2):

clamped\_j1 = requested\_j1

clamped\_j2 = requested\_j2

clamp\_at\_boundaries() is responsible for enforcing the minimum and maximum joint angles for each joint. If the joint angles passed in are outside of the operable range, they will be “clamped” to the nearest allowable value.

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)

The minimum and maximum joint angles are retrieved from the parameter server each time clamp\_at\_boundaries() is called. The “~” is the private namespace qualifier, and indicates that the parameter we wish to get is within this node’s [**private namespace**](http://wiki.ros.org/Names#Resolving) /arm\_mover/ (e.g. ~min\_joint\_1\_angle resolves to /arm\_mover/min\_joint\_1\_angle). The second parameter is the default value to be returned, in the case that rospy.get\_param() was unable to get the parameter from the param server.

**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

The rest of this function simply clamps the joint angle if necessary. Warning messages are logged if the requested joint angles are out of bounds.

**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

move\_arm() commands the arm to move, returning the amount of time that elapsed while the arm was moving.

Note: Within the function we are using the rospy.wait\_for\_message() call to receive JointState messages from the /simple\_arm/joint\_states topic. This is blocking function call, meaning that it will not return until a message has been received on the /simple\_arm/joint\_states topic.

In general, you should not use wait\_for\_message(). We simply use it here for the sake of clarity, and because move\_arm is being called from the handle\_safe\_move\_request() function, which demands that the response message is passed back as a return parameter. More discussion on this below.

**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)

This is the service handler function. When a service client sends a GoToPosition request message to the safe\_move service, this function is called. The function parameter req is of type GoToPositionRequest. The service response is of type GoToPositionResponse.

This is the service handler function, it is called whenever a new service request is received. The response to the service request is returned from the function.

Note: move\_arm() is blocking, and will not return until the arm has finished its movement. Incoming messages cannot be processed, and no other useful work can be done in the python script while the arm is performing it’s movement command. While this poses no real problem for this example, it is a practice that should generally be avoided. One great way to avoid blocking the thread of execution would be to use [**Action**](http://wiki.ros.org/actionlib). Here’s some [**informative documentation**](http://wiki.ros.org/ROS/Patterns/Communication#Communication_via_Topics_vs_Services_vs_X) describing when it’s best to use a Topic versus a Service, versus an Action.

**def** **mover\_service**():

rospy.init\_node('arm\_mover')

service = rospy.Service('~safe\_move', GoToPosition, handle\_safe\_move\_request)

rospy.spin()

Here the node is initialized with the name “arm\_mover”, and the GoToPosition service is created with the name “safe\_move”. As mentioned previously, the “~” qualifier identifies that safe\_move is meant to belong to this node’s private namespace. The resulting service name will be /arm\_mover/safe\_move . The third parameter to the rospy.Service() call is the function that should be called when a service request is received. Lastly, rospy.spin() simply blocks until a shutdown request is received by the node. Failure to include this line would result in mover\_service() returning, and the script completing execution.

**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**

This section of code is similar, to that of simple\_mover().

# Next steps

Now that you've written the arm\_mover node, the next step is to launch it, and then test it out by interacting with the service via the command line!

# 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>

More information on the format of the launch file can be found [**here**](http://wiki.ros.org/roslaunch/XML).

## Testing the new service

Now that you've modified the launch file, you are ready to test everything out.

To do so, launch the simple\_arm, verify that the arm\_mover node is running, and that the safe\_move service is listed:

**Note:** You will need to make sure that you've exited out of your previous roslaunch session before re-launching.

$ cd ~/catkin\_ws

$ catkin\_make

$ source devel/setup.bash

$ roslaunch simple\_arm robot\_spawn.launch

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**](http://wiki.ros.org/rqt)):

$ rqt\_image\_view /rgb\_camera/image\_raw

Graphical user interface, application

Description automatically generated

## Adjusting the view

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"

Graphical user interface, application

Description automatically generated

And there you have it. All of the blocks are within the field of view!

Now that you've written Arm Mover, you have gained an understanding of custom message generation, publishing to a topic, ROS services, parameters, and launch files. Before you are ready to write code, you'll still need to learn to use ROS Subscribers.

## 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.

For more information about subscribers, see [**the documentation here**](http://docs.ros.org/api/rospy/html/rospy.topics.Subscriber-class.html). Let's move on to the look\_away node so you can see subscribers in action!

# Look Away

To see a Subscriber in action, you'll write a node called look\_away. The look\_away node will subscribe to the /rgb\_camera/image\_raw topic, which has image data from the camera mounted on the end of the robotic arm. Whenever the camera is pointed towards an uninteresting image - in this case, an image with uniform color - the callback function will move the arm to something more interesting. There are a few extra pieces in the code to ensure that this procedure is executed smoothly, but you will learn more about them later.

## Creating the empty look\_away node script

Just as before when you created the arm\_mover and simple\_mover nodes, you can create the look\_away node as follows:

$ cd ~/catkin\_ws

$ cd src/simple\_arm/scripts

$ touch look\_away

$ chmod u+x look\_away

Let's have a look at the code for look\_away.

## Troubleshooting look\_away

In some cases look\_away is executing when running it manually but is not executing automatically with roslaunch. This is typically a timing issue. If look\_away starts before the system has fully initialized, then look\_away hangs in the call to safe\_move. Student jsteinbae offered a great solution to this issue:

My workaround was to add *wait\_for\_message* to the look\_away node before subscribing to the topics. This ensures that the callbacks are not called before the gazebo simulation (publishing these topics) is fully initialized.

**def** **\_\_init\_\_**(self):

rospy.init\_node('look\_away')

self.last\_position = **None**

self.arm\_moving = **False**

rospy.wait\_for\_message('/simple\_arm/joint\_states', JointState)

rospy.wait\_for\_message('/rgb\_camera/image\_raw', Image)

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)

rospy.spin()

## Updating the launch file

Just as you did with the arm\_mover node, to get look\_away to launch with the rest of the nodes, you will need to modify robot\_spawn.launch, which can be found in ~/catkin\_ws/src/simple\_arm/launch. You can add the following code there:

*<!-- The look away node -->*

<node name="look\_away" type="look\_away" pkg="simple\_arm"/>

While editing this file, it will be helpful to set max\_joint\_2\_angle: 1.57 in arm\_mover so that it isn't necessary to set it again from the command line:

*<!-- 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: 1.57

</rosparam>

</node>

# Look Away: The Code

Below is the complete code for look\_away, followed by a step-by-step explanation of what is happening. You can copy and paste this code into the look\_away script you created in the directory:

~/catkin\_ws/src/simple\_arm/scripts

## The look\_away code

*#!/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**

## The code: explained

### Import statements

*#!/usr/bin/env python*

**import** math

**import** rospy

**from** sensor\_msgs.msg **import** Image, JointState

**from** simple\_arm.srv **import** \*

The imported modules are similar to those in simple\_arm, except this time, we have the Image message type being imported so that the camera data can be used.

### The LookAway Class and \_\_init\_\_ method

**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()

We define a class for this node to better keep track of the robot arm's current movement state and position history. Just as in the node definitions before, the node is initialized using ropsy.init\_node, and at the end of the method rospy.spin() is used to block until a shutdown request is received by the node.

The first subscriber, self.sub1, subscribes to the /simple\_arm/joint\_states topic. The node is written to check the camera only when the arm is not moving, and by subscribing to /simple\_arm/joint\_states, changes in the position of the arm can be tracked. The message type for this topic is JointState, and with each message, the message data is passed to the joint\_states\_callback function.

The second subscriber, self.sub2, subscribes to the /rgb\_camera/image\_raw topic. The message type here is Image, and with each message, the look\_away\_callback function is called.

A ServiceProxy is how rospy enables calling a service from a node. The ServiceProxy here is created using the name of the service you wish to call along with the service class definition: in this case /arm\_mover/safe\_move and GoToPosition. The actual calls to the service will take place in the look\_away\_callback method below.

### The helper methods

**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

There are two helper methods defined in the code: uniform\_image and coord\_equal. The uniform\_image method takes an image as input and checks if all color values in the image are the same as the value of the first pixel. This essentially checks that all the color values in the image are the same.

The coord\_equal method returns True if the coordinates coord\_1 and coord\_2 have equal components up to the specified tolerance.

### The callback functions

**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)

When self.sub1 receives a message on /simple\_arm/joint\_states topic, the message is passed to the joint\_states\_callback in the variable data. The joint\_states\_callback uses the coord\_equal helper method to check if the current joint states provided in data are the same as the previous joint states, which are stored in self.last\_position. If the current and previous joint states are the same (up to the specified tolerance), then the the arm has stopped moving, so the self.arm\_moving flag is set to False. If the current and previous joint states are different, then the arm is still moving. In this case, the method updates self.last\_position with current position data and sets self.arm\_moving to True.

The look\_away\_callback is receiving data from the /rgb\_camera/image\_raw topic. The first line of this method verifies that the arm is not moving and also checks if the the image is uniform. If the arm isn't moving and the image is uniform, then a GoToPositionRequest() message is created and sent using the safe\_move service, moving both joint angles to 1.57. The method also logs a message warning you that the camera has detected a uniform image along with the elapsed time to return to a nicer image.

# Look Away: Launch and Interact

You can now launch and interact with simple\_arm just as before:

$ cd ~/catkin\_ws

$ catkin\_make

$ source devel/setup.bash

$ roslaunch simple\_arm robot\_spawn.launch

**Please note that if you are having trouble with roslaunch simple\_arm robot\_spawn.launch please try the safe\_spawner.sh script in the scripts folder. You can launch by using $ ./safe\_spawner.sh in a terminal of your choice.**

After launching, the arm should move away from the grey sky and look towards the blocks. To view the camera image stream, you can use the same command as before:

$ rqt\_image\_view /rgb\_camera/image\_raw

To check that everything is working as expected, open a new terminal, source devel/setup.bash, and send a service call to point the arm directly up towards the sky (note that the line break in the message is necessary):

rosservice call /arm\_mover/safe\_move "joint\_1: 0

joint\_2: 0"

What happens?

## 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**](http://wiki.ros.org/rospy/Overview/Logging)):

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

## Filtering and saving log messages from /rosout

Note that for messages written to /rosout, you can see the messages in real time as your program is running by echoing:

rostopic echo /rosout

Although it can be helpful to view messages this way, because of the volume of messages written to that topic, it can sometimes be helpful to filter messages by piping them to [**grep**](https://en.wikipedia.org/wiki/Grep). These grepped messages can also be saved to a file for debugging:

rostopic echo /rosout | grep insert\_search\_expression\_here

rostopic echo /rosout | grep insert\_search\_expression\_here > path\_to\_output/output.txt

## 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".

As you continue on with the project, the ROS systems that you write will become increasingly complex. Being able to use logging effectively may prove extremely useful for debugging!

## Congratulations!

You’ve now come the conclusion of the ROS Basics!

Armed with the basic understanding of ROS you’ve learned here you should now be prepared for the projects which follow!

## Additional Resources

While we’ve done our best to give you a solid understanding of the fundamental concepts of ROS, the lessons here are by no means comprehensive. Fortunately, there are a wealth of resources available online.

### [ROS Wiki](http://wiki.ros.org/)

This is the official source of documentation for all ROS packages. Additionally, there are many helpful tutorials here. Aside from a simple search on google, this is probably the first place you should go if you are having a ROS-related problem.

### [ROS Answers](http://answers.ros.org/)

If the wiki does not provide the answers you are looking for, ROS Answers will be the next best bet. With over 33,000 questions asked on ROS answers, there’s a good chance that somebody has already addressed the problem that you are dealing with.

### [ROS Cheat Sheet](https://github.com/ros/cheatsheet/releases/download/0.0.1/ROScheatsheet_catkin.pdf)

This is the official ROS cheat sheet. Even though the title indicates that it is for the indigo distribution, almost all of the commands still work in Kinetic, the distribution that we are using for this program.

### [A gentle Introduction to ROS](https://cse.sc.edu/~jokane/agitr/)

This is a great book, and is distributed not only in paperback form, but also for free as a PDF download.