1 ROS Essentials

Welcome to the very first lesson of this ROS MOOC! Before we dive into ROS, it is helpful to know something about how ROS works internally. In this video, we will outline the ROS essentials, which are: Nodes, Topics, and how they interact.

There are two fundamental concepts in ROS, which are: Nodes and Topics.

## Nodes:

These are software processes that do 'stuff' (e.g. process data, command hardware, execute algorithms). Nodes provide modularity to robotic projects that use ROS. They are often written in C++ or Python. In this course, we will use Python to write them. ROS is only compatible with Python 2.7.x.

## Topics:

Transport information between nodes, in the form of messages.

In a real robot application you will often have to deal with a large number of nodes and topics. It is important to know which nodes are talking to each other, and what topics are being used to pass the information (messages) between nodes.

# 1.1 ROS Nodes

In this lesson we will find out what happens with ROS nodes. The focus of the video will be on three different ways to inspect ROS nodes, and see what they're doing.

Remainder: Don't forget to source your workspace if you want to use ROS commands!

If you've followed the procedure under the Course Setup section correctly, you only need to execute the following in the shell:

$ source $HOME/hrwros\_ws/devel/setup.bash

## Summary:

There are three main ways to inspect a ROS node:

Simply list all currently running nodes in the terminal:

rosnode list

Make a visual graph of all running nodes and their connections:

rqt\_graph

List information about a specific node in the terminal:

rosnode info <node\_name>

In the following lectures, we will learn much more about Nodes. For example, we will see that they can publish (write) to a topic and subscribe (read) from a topic.

## Question 1

1/1 point (ungraded)

In robotics, we often have to interact with the real-world. One type of device we can use to interact with the environment is a sensor: we can sense a desired quantity from the environment and turn it into data.

Are sensors an example of a ROS node, or ROS topic?

Think of the block diagram from the lecture video!

Node

Topic

correct

**Answer**

Correct:

This is correct, because a sensor processes information and then provides data. It does not transport it between nodes.

# 1.2 ROS Topics

After learning about nodes, it's important to know how topics work. Topics are the information highway of ROS. In this lesson, we will learn more about how they are structured and how we can inspect them using the same approach as with nodes.

## ****Summary:****

ROS topics transport information between nodes. This information is organized as a data structure and can have different data types. Topics can be identified by their name and their type. You can think of topics as strongly typed message buses. Any node can connect to it to send or receive data, as long as they are the right type.

As with nodes, there are three ways to inspect topics:

Don't forget to source your workspace if you want to use ROS commands! If you've followed the procedure under the Course Standards section correctly, you only need to execute the following in the shell:  
$ source $HOME/hrwros\_ws/devel/setup.bash

* Display a list of all topics that are currently being exchanged between active nodes  
  rostopic list
* Display information about the data structure of a specific topic  
  rostopic info <topic\_name>
* Print the current content of a topic in the terminal  
  rostopic echo <topic\_name>

Important detail: Do not forget the "/" before node and topic names!

## Question 1

1/1 point (ungraded)

Nodes and topics together form the backbone of any ROS application. It is important to know how they work together!

This question is deliberately content that is NOT literally addressed in the course video. This is intended to encourage you to explore [answers.ros.org](https://answers.ros.org/) where you can actually find the correct answer, or ask a question if you can’t find one. This way, you can also participate with the worldwide ROS learning community.

How many nodes can publish to a single topic?

Only one at a time.

The amount is defined by the topic.

Any number of nodes can publish, as long as the message has the right type.

Any number of nodes can publish any kind of message to any node.

correct

**Answer**

Correct:

This is true!

# 1.3 ROS application introduction

**Summary**

It's time to build your own ROS application structure! There are four fundamental types of ROS nodes that can be used to build a ROS application:

* Publishers
* Subscribers
* Services
* Actions

Please keep in mind, that here we refer to Services and Actions as fundamental types of nodes. That is, we are referring to nodes that use services and actions for communication. Publishers, Subscribers, Services and Actions are used by ROS nodes to communicate between each other.

First, we will focus on Publisher and Subscriber nodes.

## 1.3.1/1.3.2 Build your own ROS Application - Publisher and subscriber nodes

This time, we will take a look at two different classes of ROS nodes: publishers and subscribers.

**Publisher Nodes**

A ROS node that generates information is called a publisher. A publisher sends information to nodes via topics. With robotics often these publishers are connected with sensors like cameras, encoders, etc.

If you use the rosnode info command you can see to which topics a node is connected, and if these are outbound or inbound connections.

**Subscriber Nodes**

A ROS node that receives information is called a subscriber. It's subscribed to information in a topic and uses topic callback functions to process the received information. With robotics, subscribers typically monitor system states such as triggering an alert when the robot reaches joint limits.

You can again use the rosnode info command to find information about a subscriber node. Don't forget to source your workspace!

Publisher node - Example code

## Node to publish a string topic.

import rospy

from std\_msgs.msg import String

def simplePublisher():

simple\_publisher = rospy.Publisher('topic\_1', String, queue\_size = 10)

rospy.init\_node('node\_1', anonymous = False)

rate = rospy.Rate(10)

# The string to be published on the topic.

topic1\_content = "my first ROS topic"

while not rospy.is\_shutdown():

simple\_publisher.publish(topic1\_content)

rate.sleep()

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

try:

simplePublisher()

except rospy.ROSInterruptException:

pass

Subscriber node - example code

## Node to subscribe to a string and print the string on terminal.

import rospy

from std\_msgs.msg import String

# Topic callback function.

def stringListenerCallback(data):

rospy.loginfo(' The contents of topic1: %s', data.data)

def stringListener():

rospy.init\_node('node\_2' , anonymous = False)

rospy.Subscriber('topic\_1' , String, stringListenerCallback)

# spin() simply keeps python from exiting until this node is stopped

rospy.spin()

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

stringListener()

### **Question 1**

1/1 point (ungraded)

Is the following statement True or False: A ROS Topic can be published without initializing a ROS Node.

How do you publish to a topic?

True

False

correct

**Answer**

Correct:

A ROS Topic can only be published from a ROS Node.

Show answer

Submit

Some problems have options such as save, reset, hints, or show answer. These options follow the Submit button.

### **Question 2**

1/1 point (ungraded)

Is the following statement True or False: A subscriber callback function is executed continuously, that is, it is processing all the time.

When does data processing happen?

True

False

correct

**Answer**

Correct:

It's not always processing, it will only process when new data is published.

# 1.3.3 ROS File system - part 1

This lesson consists of two videos about the ROS File system.  In this first part you will mainly focus on setting up your workspace with catkin.

# **Important note**

The video mentions working with the older ROS version Kinetic. The course currently uses Melodic, therefore you should source the appropriate file: opt/ros/melodic/setup.bash.

# **ROS file system - nomenclature**

ROS workspace (catkin workspace) consists of different subspaces. A workspace is a folder to organize ROS project files. ROS uses catkin, which is a build tool to compile source files into binary files. Your code goes into the src workspace folder and catkin manages the other ones. A catkin ROS workspace contains three main spaces:

**src** space: contains source code, this will be your main work folder  
**devel** space: contains setup files for the project ROS environment  
**build** space: contains the compiled binary files  
If you'd like to read some more about these spaces, you can do so [here](http://wiki.ros.org/catkin/workspaces" \l "Catkin_Workspaces).

We will now guide you through the creation of our own catkin workspace in ROS. This workspace will just be used as an example to show you the process, and won't be used anymore after this.

First, create a new folder for your workspace:  
$ mkdir -p new\_ros\_ws/src

You can use the ls command to verify the new folder that was created.   
Next move to your newly created workspace using the command cd new\_ros\_ws.  
Thereafter, setup the correct ROS environment using the command source /opt/ros/melodic/setup.bash. Finally, we initialize catkin:  
$ catkin init  
$ catkin build

##### Warning!

If you independently follow the tutorials on ROS Wiki, you might come across the command catkin\_make. That is different to what we use in our course which is catkin. Don't try and mix the two! They are not compatible. You can read up on catkin [here](https://catkin-tools.readthedocs.io/en/latest/index.html).

## Question 1

0/1 point (ungraded)

We will provide you with four statements. Please select the following statements that are TRUE.

There are two correct statements.

1) A ROS workspace can be located anywhere in your home folder.

2) A ROS workspace which uses catkin as its build tool is called a catkin workspace.

3) Everything you need for your ROS application (Ex: drivers for sensors, cameras etc) goes into your catkin workspace.

4) The command mkdir -p new\_ros\_ws/src creates a catkin workspace.

incorrect

**Answer**

Incorrect:

A ROS workspace is just a folder on your Linux installation. The concept of a workspace is to only organize your ROS application development in one location. But, the workspace itself can reside anywhere in your home folder.

When you use catkin as the build tool for your ROS workspace, it is called a catkin workspace. So, in essence a catkin workspace and ROS workspace are pretty much the same, just that catkin workspace is a bit explicit in saying what build tool you are using for the ROS workspace.

ot everything you need for your ROS application goes into a catkin workspace. You can simply install the binaries using sudo apt-get install desired-binaries. This command is very similar to running a .exe file on Windows or .dmg file on MacOSX, whenever you want to install a new device or software on your system.

# 1.3.3 ROS File system - part 2

In this second part you will learn more about the src folder in your workspace and ROS packages.

ROS packages reside in the 'src' space. In ROS, software is organised in ROS packages. A ROS **package** typically contains the following things:  
- CMakeList.txt  
- package.xml (These two files indicate that the folder is a ROS package file. More on these two later in the course)  
- scripts/ (This folder contains all Python scripts. We will only use Python in this course)  
- src/ (This folder contains all C++ source files. We will not use these in this course)

To create a new ROS package, we will use catkin:  
$ cd <path\_to\_ros\_ws>/src  
$ catkin\_create\_pkg hrwros\_week2 std\_msgs

Please keep in mind that you don't need to place the source files of any dependencies, such as std\_msgs in the line above. Instead, you can simply install their binaries with the rosdep command:  
$ rosdep install <package\_name>

You can also install all ROS package dependencies in one command:  
$ cd <path\_to\_ros\_ws>/src  
$ rosdep install --from-paths . --ignore-src -y

src is not the only space in your workspace: there is also the 'devel' space. This contains all binary executables from your src spaces.

If you want more information on the folder structure of a catkin workspace, you can do so [here](http://wiki.ros.org/catkin/workspaces).

### **Question 1**

1/1 point (ungraded)

It is not mandatory to create a folder inside the "src" space of your catkin workspace which in turn contains all your ROS packages. For example, new\_ros\_ws/src/hrwros/hrwros\_week1 where, only hrwros\_week1 is the ROS package

Is the above statement True of False?

True

False

correct

**Answer**

Correct:

Having a separate folder inside the src space is only for file organization sanity. It is not mandatory to create a "top-level" folder inside your catkin workspace that contains all your ROS packages. But, it is recommended practice.

## 1.3.4 Custom message types

In previous lectures, we spoke of ROS topics having a topic type. In this lecture, ROS message and topic types will be further explained.

**Summary**

Nodes can process and share information through topics. The information they pass through these topics needs to be structured in some way, to make it actually usable. Such a structure is known as a data structure. In ROS, we can abstract these data structures as ROS message types. Common data structures in ROS are for example floats, integers, and strings.

In ROS, we can easily combine multiple data structures using derived message types. For example, to represent a position in 3D space we will need 3 floating point values: X, Y and Z. The derived message type will then be a struct of three floating point numbers: {float x, float y, float z}.

These (derived) message types are defined in message files. These message files are typically located in <ros\_package\_name>/msg, with the filename <NewMessageType>.msg. For example, look in the $HOME/hrwros\_ws/src/hrwros/hrwros\_msgs/msg folder for some examples.

#### Example: Ultrasound distance sensor.

We want to construct a new message type called SensorInformation. It should contain:

* A ROS message type for interfacing with distance sensors
* A string containing the manufacturer name
* An unsigned integer containing the sensor part number

We would create the following file: $HOME/hrwros\_ws/src/hrwros/hrwros\_msgs/msg/SensorInformation.msg.

It will contain the following:  
sensor\_msgs/Range sensor\_data  
string maker\_name  
uint32 part\_number

We can see something really interesting here: We use an already derived message, sensor\_msgs/Range, and simply add a string and an integer to it. So we can create new message types using already existing derived message types! This idea of stacking is really useful in ROS, since you can easily re-use what already exists. So in the above example, the sensor\_msgs/Range already contains everything we need from the range sensor itself, and we only add the maker name and the part number.

### **Question 1**

1/1 point (ungraded)

It is important to note that ROS message types are associated with topics.

This question is deliberately a content that is NOT literally addressed in the course video. This is intended to encourage you to explore [answers.ros.org](https://answers.ros.org/) where you can actually find the correct answer, or ask a question if you can’t find one. This way, you can also participate with the worldwide ROS learning community.

How many message types can be published to a topic?

0.

1.

2.

At least ten.

An arbitrary amount.

correct

**Answer**

Correct:

Topics are strongly associated with only one message type.

## 1.3.5 Define and build custom message types

Bookmark this page

In this video tutorial, we will show you how to define your own message type in a message file, and add the file to your CMakeLists.txt to create a new ROS message type.

Don't forget to source your workspace! If you've followed the procedure under the Course Setup section correctly, you only need to execute the following in the Course Command Shell (CCS):  
$ source $HOME/hrwros\_ws/devel/setup.bash

First, navigate to the folder where ROS message types are defined:  
$ roscd hrwros\_msgs  
$ cd msg

Then, create the messagefile:  
$ touch SensorInformation.msg

You can see all created message types by typing  
$ rosmsg show sensor\_msgs/  
and hitting the Tab key.

Since we will need the Range type, enter  
$ rosmsg show sensor\_msgs/Range  
and you will see what the Range message type consists of.

In the following example, we will create our own message type.

* Create the file and contents as shown below:
  + Detail the data types the message type will contain
  + And then some comments detailing the data entries
* Make sure to add the name to the add\_message\_files section in the CMakeLists.txt file
* Finally, run the catkin build command

SensorInformation.msg

# Example message file

sensor\_msgs/Range sensor\_data # placeholder for the ROS message type for interfacing with distance sensors

string maker\_name # placeholder for manufacturer name

uint32 part\_number # placeholder for part number of the sensor

### **Question 1**

1 point possible (ungraded)

In our custom message definitions, we can have different fields, including those defined in other ROS packages.

Is the above statement True or False?

True

False

unanswered

Submit

Some problems have options such as save, reset, hints, or show answer. These options follow the Submit button.

### **Question 2**

1 point possible (ungraded)

You will always have to update the CMakeLists.txt file with your new message definition and run catkin build, to make sure the python objects corresponding to your new message definitions are available in your python scripts.

Is this statement True or False?

True

False

unanswered

Submit

Some problems have options such as save, reset, hints, or show answer. These options follow the Submit button.

## 1.4 ROS Services

Bookmark this page

**Summary:**

So far, communications between nodes have been handled by topics. Topics can connect many nodes to many other other nodes. There are no acknowledgements between the nodes: they are simply pumping around information. This is very useful when monitoring certain values, but can become problematic with some types of information such as video data. In cases like those, it would be nice to only receive information when requested.

ROS services implement these request-response type of communications. They consist of two message types: One for requesting data, and one for the response. These services are the gateway to event-based ROS executions. Services are defined in the same package as messages, in their own /srv folder.  
The nomenclature for ROS services: One ROS node is the **service server**. It advertises a service, and makes it available to other nodes. Another node is the **service client**. It sends a request message to the service server once the service is available.

#### Example

A useful example of a ROS service could be a service that converts measurements in metres to measurements in imperial feet. The request message would contain the measurement in metres, and the response would contain the measurement in imperial feet plus a boolean value indicating the conversion was successful.  We would create the file hrwros\_msgs/srv/ConvertMetresToFeet.srv:

hrwros\_msgs/srv/ConvertMetresToFeet.srv

float64 measurement\_metres # Request field

--- # Demarcation between request and response

float64 measurement\_feet # Response field

bool succes

One important quality of services is that they are blocking: They stop the execution of program flow until the response has been received. This is useful for sequential behaviours. It also makes it desirable to have quickly executable computations for the service callback, so that the program is blocked for the least amount of time. There is also no going back on a service call: Once a request has been made, it will be processed until a response is sent. It can't be interrupted.

### **Question 1**

1/1 point (ungraded)

One place we can find client-server interactions is in the kitchen of a busy restaurant. The waiter will place the order with the kitchen, and after some time, the kitchen will provide the server with the requested dish.

Which of the following statements are True?

There are three correct answers.

A: The waiter is an example of a service client.

B: The waiter is an example of a service provider.

C: The order the waiter gives to the kitchen is a request.

D: The order the waiter gives to the kitchen is a reponse.

E: The kitchen is an example of a response.

F: The kitchen is an example of a service server.

correct

**Answer**

Correct:

A: This is true, because the waiter asks the kitchen to give him the correct dish.

C: This is true, because the waiter requests the correct dish with his order paper.

F: This is true, because the kitchen provides the waiter with the correct dish after the order is placed.

HintShow answer

Submit

Some problems have options such as save, reset, hints, or show answer. These options follow the Submit button.

## 1.5.1 ROS Actions action message and goal processing

The action generator script displays 7 actions:

* Specified actions:

1) CounterWithDelayFeedback  
2) CounterWithDelayGoal  
3) CounterWithDelayResult

* Additional actions which are used internally by the actionlib package:

4) CounterWithDelayActionFeedback  
5) CounterWithDelayActionGoal  
6) CounterWithDelayAction  
7) CounterWithDelayActionResult

CounterWithDelayAction.msg is an aggregation of the other action messages. This is the action type of our action client

The action client code is something that we as users develop to leverage the facilities of the actionlib ROS package.

Sent goals are registered by the action server and the actionlib package. The package requires additional information which is CounterWithDelayAction.msg.

Goal contains:

1. Timestamp and header info
2. Unique goal identifier
3. Goal message

Action Result contains:

1. Timestamp and header info
2. Different states the goal can be in

Feedback contains:

1. Timestamp and header info
2. Different states the goal can be in
3. Feedback message

Action server topics

* Feedback topic is published by the action server an has action feedback message type
* Action server subscribers to the cancel topic and is of the GoalID type

**Attention!**

* Simple Action Server/Client.
* Non-blocking execution of ONE goal at a time.
* New goal to the same action server will pre-empt an active goal. Meaning that the active goal is completed first before a new goal will be processed.

|  |  |
| --- | --- |
| **Ros node and ros topic** | |
| , you only need to execute the following in the shell: | $ source $HOME/hrwros\_ws/devel/setup.bash |
| First, create a new folder for your workspace: | $ mkdir -p new\_ros\_ws/src |
| Simply list all currently running nodes in the terminal: | rosnode list |
| Make a visual graph of all running nodes and their connections: | rqt\_graph |
| List information about a specific node in the terminal: | rosnode info <node\_name> |
| * Display a list of all topics that are currently being exchanged between active nodes | rostopic list |
| * Display information about the data structure of a specific topic | rostopic info <topic\_name> |
| * Print the current content of a topic in the terminal | * rostopic echo <topic\_name> |

|  |  |
| --- | --- |
| First, create a new folder for your workspace:  You can use the ls command to verify the new folder that was created.  Next move to your newly created workspace using the command  Thereafter, setup the correct ROS environment using the command  . Finally, we initialize catkin: | * $ mkdir -p new\_ros\_ws/src   cd new\_ros\_ws.  source /opt/ros/melodic/setup.bash  $ catkin init $ catkin build |
| * To create a new ROS package, we will use catkin: * Please keep in mind that you don't need to place the source files of any dependencies, such as std\_msgs in the line above. Instead, you can simply install their binaries with the rosdep command:   find gazebo\_plugin  You can also install all ROS package dependencies in one command: | * $ cd <path\_to\_ros\_ws>/src * $ catkin\_create\_pkg hrwros\_week2 std\_msgs   $ rosdep install <package\_name>  grep gazebo\_plugin -R catkin\_ws/src/crane\_simulation  $ cd <path\_to\_ros\_ws>/src $ rosdep install --from-paths . --ignore-src -y |

for i in {1..10}; do echo -n "This is a test in loop $i "; rosservice call /live\_scan/execute\_live\_scan "file\_name: 'test'

speed: 10" ; sleep 1 ; done

rosrun tf static\_transform\_publisher 0 0 0 0 0 0 base\_footprint base\_scan 100

|  |  |
| --- | --- |
| Define and build custom message types | |
| *Don't forget to source your workspace! If you've followed the procedure under the Course Setup section correctly, you only need to execute the following in the Course Command Shell (CCS):* | $ source $HOME/hrwros\_ws/devel/setup.bash |
| First, navigate to the folder where ROS message types are defined: | $ roscd hrwros\_msgs $ cd msg |
| Then, create the messagefile: | $ touch SensorInformation.msg |
| You can see all created message types by typing | $ rosmsg show sensor\_msgs/ and hitting the Tab key. |
| Since we will need the Range type, enter  and you will see what the Range message type consists of. | $ rosmsg show sensor\_msgs/Range |
| In the following example, we will create our own message type. | |
| * Create the file and contents as shown below: | # Example message file  sensor\_msgs/Range sensor\_data # placeholder for the ROS message type for interfacing with distance sensors  string maker\_name # placeholder for manufacturer name  uint32 part\_number # placeholder for part number of the sensor |
| * + Detail the data types the message type will contain And then some comments detailing the data entries | |
| * Make sure to add the name to the add\_message\_files section in the CMakeLists.txt file | |
| * Finally, run the catkin build command | |

|  |  |
| --- | --- |
| **Start first application** | |
| Set up ROS sources and workspace | |
| You will first setup your ROS sources inside the CCS (The "Course Command Shell" - see the previous page) by running the following commands: | $ source /opt/ros/melodic/setup.bash  $ rosdep update |
| 2. Create your workspace folder, also inside the CCS (The "Course Command Shell"), with this commands: | $ mkdir -p $HOME/hrwros\_ws/src/hrwros  $ cd $HOME/hrwros\_ws |
| Test your first ROS application | |
| Go back to the CCS and run the following commands: | $ cd $HOME/hrwros\_ws  $ catkin build |
| If all went well, run the following command in the CCS: | $ source $HOME/hrwros\_ws/devel/setup.bash |
| At this point, you are all setup. To verify your course setup, run the following command in the CCS: | $ roslaunch hrwros\_week1 hrwros\_welcome.launch |

|  |  |
| --- | --- |
| **Publisher node with custom message type** | |
| following in the shell: | source $HOME/hrwros\_ws/devel/setup.bash |
| Let's first navigate to our Week 1 workspace folder  Enter the scripts folder | $ roscd hrwros\_week1  $ cd scripts |
| Then, create our node python script: | $ touch sensor\_info\_publisher.py |
| We can now copy the contents of the template file template\_publisher\_script.py to our new python script: | $ cp template\_publisher\_script.py sensor\_info\_publisher.py |
| Starting your new ROS node | |
| First run roscore in a CCS after sourcing the setup files |  |
| Then, in another CCS,  the ROS node with the following command, after sourcing the setup files: | $rosrun hrwros\_week1 sensor\_info\_publisher.py |
| This will lead to an executable error. This error can be resolved by a simple command which makes the node executable. | $ roscd hrwros\_week1/scripts $ chmod +x sensor\_info\_publisher.py |
| **Inspecting our node** | |
| This node is a publisher, publishing to a topic. So let's see if we can find the topic. | $ rostopic list   (we look for /sensor\_info topic) |
| Let's look at its contents. | $ rostopic echo /sensor\_info |
| It's always nice to receive some acknowledgement that a ROS node is working. To do that we open our node in the gedit editor. | $ gedit sensor\_info\_publisher.py |
| After publishing our topic we print a log message by adding the following linen of code to the file: | rospy.loginfo('All went well. Publishing topic') |

|  |  |
| --- | --- |
| 1.3.7 Subscriber ROS node with a custom message type Bookmark this page | |
| Use the template and start editing it. We will first create a new file, and copy the contents of the example file into it. | $ roscd hrwros\_week1/scripts $ touch sensor\_info\_subscriber.py $ cp template\_subscriber\_script.py sensor\_info\_subscriber.py |
| Make the subscriber script executable | chmod +x sensor\_info\_subscriber.py |
| To run our new subscriber node: Run roscore in a new course command shell. Remember you need to source your workspace in every new shell! | $ source $HOME/hrwros\_ws/devel/setup.bash $ roscore |
| Then, in a separate CCS: | $ source $HOME/hrwros\_ws/devel/setup.bash $ rosrun hrwros\_week1 sensor\_info\_subscriber.py |
| Check with | rostopic list: $ rostopic info /sensor\_info |
| To actually see the contents of the topic, you will need to publish it from a new CCS. | $ rosrun hwros\_week1 sensor\_info\_publisher.py |
|  |  |

|  |  |
| --- | --- |
| 1.4.1 ROS Services code illustration: Service server part 1 | |
| Let's begin by sourcing our setup files, and navigate to our workspace msgs folder: | $ source $HOME/hrwros\_ws/devel/setup.bash $ roscd hrwros\_msgs/ |
| We can see the msg/ folder, but we just learned services are defined in the srv/ folder. So let's create that, and create our service file: | $ mkdir srv $ cd srv $ touch ConvertMetresToFeet.srv |
| Now our service file is created, we can start editing it. Remember, we first want the request message type, then a demarcation of three dashes, and then a response message type: | float64 distance\_metres     # Request message: Distance in (m) to be converted to (ft) ---                         # Demarcation float64 distance\_feet       # Response message: Distance in (ft) after conversion bool success                # Response message: Success or failure of conversion |
| add our new service to the CMakeLists.txt file | |
| Now, just like we did for messages, we add our new service to the CMakeLists.txt file in the hrwros\_msgs/ folder. Note that services have their own section, after messages. Add the name of our new service definition file after the FILES entry. Don't forget the .srv file extension. |  |
| Finish by building our service by running the catkin b command. We can check if it worked correctly by using the rossrv command:  The terminal should show the request message field, and the response message field. | catkin b  $ rossrv show hrwros\_msgs/ConvertMetresToFeet $ rossrv show hrwros\_msgs/ConvertMetresToFeet -r |
| Inspecting our service server code Now we've created a new service, we can use it in our code. Let's go to the week 1 script folder, where we already have two template implementations. Let's begin by inspecting the server: metres\_to\_feet\_server.py.  We are importing the newly created service definitions from hrwros\_msgs.srv. Note that we not only have to import the service definition, but also explicitly the request and response message definitions (#L6!  The service request is processed in the service callback function (#L13).   * We first create an object for the response message type, and in the example, do a check in the distance input to check if it is a positive real number. * If the sanity check fails, we return a default error value in the response. * Otherwise, we do the conversion and return both the converted value and a flag that indicates the conversion was successful. * In either case, we return the response.   Then, we have the next function for the administrative settings of the service server node (#L38).   * Here, we create a ROS node for our service (#L32). * We create the actual service by giving it a name, a type, and a callback function for processing a service request. (#L35) * Finally, in the main function (#L41) we start the service server node. | |
| 1.4.1 ROS Services code illustration: Service client part 2BookmarkInspecting our service client code We will now be looking at the service client example: metres\_to\_feet\_client.py.  Lines 6 through 8 contain all the required imports, and the client definition on #L10. We first wait for the required service to become available. Then, we create a service proxy to be able to call the service (#L16). Then, we actually call the service on #L19, and return the service response from the service call (#L22).  In the main function, we have the ROS node for the service client (#L30). We send a test value to the server via the client (#L38). Once we have a response, we check if the conversion was successful, and report the return value accordingly. We are using the logerr function for error messages (#L42). | |
| Using the service server & client | |
| First, make sure you are in the hrwros\_msgs folder. Start roscore in a separate terminal. Then, source our setup files and run our server, and: | $ $HOME/hrwros\_ws/devel/setup.bash $ rosrun hrwros\_week1 metres\_to\_feet\_server.py |
| The service is now available. Time to also start the ROS client in a separate terminal:  The terminal will both display the request and response messages. | $ $HOME/hrwros\_ws/devel/setup.bash $ rosrun hrwros\_week1 metres\_to\_feet\_client.py |
| We learned, that services will block execution. Let's look at that as well. Since the processing, which is blocking, is happening in the server, let's go to the server file and add a delaying test loop. In metres\_to\_feet\_server.py, navigate to #L27 and add the following block of code: | for test\_idx in range(0,10):     rospy.sleep(1) |
| On the client side, let's write a log message after the response has been received. In metres\_to\_feet\_client.py, navigate to #L21 and add the following block of code: print('I only got here AFTER the service call was completed!')  First, terminate the server in the terminal (CTRL+C), and restart it. Restart the client the same way. You will now see the client will only print out every 10 seconds, because the processing is not complete yet on the server side. | |

|  |  |
| --- | --- |
| **1.5 ROS Actions client server communication**  In this lecture, we will look into the desirability of (non)-blocking executions, which leads to ROS Actions. We will show you how the terminology works and give an example of implementation of a counter. | |
| We don't always want blocking execution. ROS Actions allow non-blocking execution. This way, multiple things can happen at the same time. They are a generalized request-response system (as for services): a **client-server** infrastructure. Actions are defined by three message types: | 1. A **goal** (request), 2. The **result** (response), 3. And **feedback**. |
| There are a few ROS commands to interact with actions:  Generate action messages manually:   Show the contents of an action message: | $ rosrun actionlib\_msgs genaction.py <path\_to\_action\_file>  $ rosmsg show <stack\_name>\_msgs/<ActionMessage> |
| ROS actions have a few functions available to process requests: | * The goalCallback function processes a goal request. * Goal statuses: ACTIVE, SUCCEEDED, ABORTED. |
| Nomenclature: | * Action server: A ROS node that advertises an action, so that other nodes can request action goals to be processed. * Action client: A ROS node that sends goal requests to the action server. |
| Code example Here, we will show the general structure of an action file.  Requirement: Counter with a 1s delay between each increment.  Goal message: Number to count up to (uint32) Result message: A status message (string) Feedback message: The number of counts completed (uint32) | hrwros\_msgs/action/CounterWithDelay.action    uint32 num\_counts # Goal field  --- # Separator  string result\_message # Result field  --- # Separator  uint32 counts\_elapsed # Feedback field |

|  |  |
| --- | --- |
| 1.5.2 ROS Actions code illustration: Part 1 Bookmark this page | |
| Creating an action file | |
| Let's first create our action folder and our new action file. | $ mkdir action $ cd action $ nano CounterWithDelay.action |
| We first define the goal message as the number of counts to count up to. Then a result message which is a string message for the result and a feedback message to indicate the numbers of counts elapsed. | CounterWithDelay.action    uint32 num\_counts  ---  string result\_message  ---  uint32 counts\_elapsed |
| Now we update the CMakeLists.txt file with the vim editor in the action files subsection. (#L25) | CounterWithDelay.action (#L27)  We need to add the actionlib package that provides all the ROS action functionalities. We need to add it at all dependancy locations. (#L4,37,45) actionlib actionlib\_mgs  Update the package.xml file also with the dependency. (#L21) |
| Now run catkin build or in short catkin b. |  |
| Action message | |
| Don't forget to source your ROS workspace first! | $ source $HOME  /hrwros\_ws/devel/setup.bash |
| If we type the complete action name  you will see the complete message definition.  We need to pay attention to where our message definitions are located as a lot of the message is for the actionlib package. | $ rosmsg show hrwros\_msgs  /CounterWithDelayAction |
| If you want to see the message definition, you can go to the hrwros\_msgs package folder. | $ roscd hrwros\_msgs |
| And  run the following script. | $ rosrun actionlib\_msgs genaction.py -o msg action/CounterWithDelay.action |
| There we find all the action definitions. | $ rosmsg show hrwros\_msgs/ |

|  |  |
| --- | --- |
| **1.5.2 ROS Actions code illustration: Part 2**  Bookmark this page | |
| Similar to the services in the previous module we have the action server and the action client. This example script has been modified from the actionlib tutorial on the ROS wiki to suit this course context.  The relevant imports and the python class implementation of the action server. Followed by the defined feedback and result messages. Then we initialise the action server. After that, we start the action server with the start function. Now the execute callback function needs to be configured. In the final part, we first create the ROS node for the action server. Then we instantiate the action server implementation by initialising it with the name of the node.  Now to the action client. First, you have the imports followed by a simple action client. After that, you wait for the server to become available. Then we create a goal message. We can include a blocking functionality or perform other actions while the action is being processed. Lastly, we return the result. | |
| Don't forget to source your workspace in every shell first! | $ source $HOME/hrwros\_ws/devel/setup.bash |
| First start roscore in a different terminal. |  |
| Then we start the action server and client with the following commands. | $ rosrun hrwros\_week1 counter\_with\_delay\_**as**.py $ rosrun hrwros\_week1 counter\_with\_delay\_**ac**.py |
| You can see while the goal is being progressed you can do other things. We can verify this with the following code. | $ rostopic echo /counter\_with\_delay/feedback |

|  |
| --- |
| **1.6 ROS Launch Files**  Launching lots of ROS nodes often takes a lot of time, and a lot of terminals. Is there a better way to go about this? This lecture, we will take a look at ROS launch files. |
| Up until now, whenever we wanted to test our ROS nodes, we had to open up many new terminal files to run all the components. For larger packages, this gets unwieldy really fast. This is where ROS launch files come in. This way, we can start all our ROS nodes together in one terminal.  The idea is to group ROS nodes in one file. The code for the nodes themselves is still in separate files, but we link them in one launch file, which we can launch with a single command. It also allows for passing arguments, node specific parameters, and namespaces. They allow for much more, but for now, we will focus our attention on the arguments and parameters.  Launch files are located in the launch folder within a ROS package. It is common practice to follow a naming convention of <package\_name>\_<file\_name>.launch, e.g. hrwros\_week1\_servers.launch.  Let's look at a small example. You can see how the different nodes are started, and how the action server uses a parameter to launch. |
| hrwros\_week1\_servers.launch    <?xml version="1.0"?>  <launch>  <!--Argument to the launch file-->  <arg name="counter\_delay\_parameter" default="1.0"/>  <!--Start the metres\_to\_feet service server ROS node-->  <node name-"metres\_to\_feet" pkg="hrwros\_week1" type-"metres\_to\_feet\_server.py" output="screen"/>  <!--Start the action server ROS node-->  <node name="counter\_with\_delay" pkg-"hrwros\_week1" type="counter\_with\_delay\_as.py" output="screen">  <param name="counter\_delay" type="double" value="$(arg counter\_delay\_parameter)"/>  </node>  </launch> |