Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

b-it Bonn-Aachen International Center for Information Technology

ROS Nodes, Topics, and Messages

Foundation Course

September 6, 2019

Hassan Umari

Recap

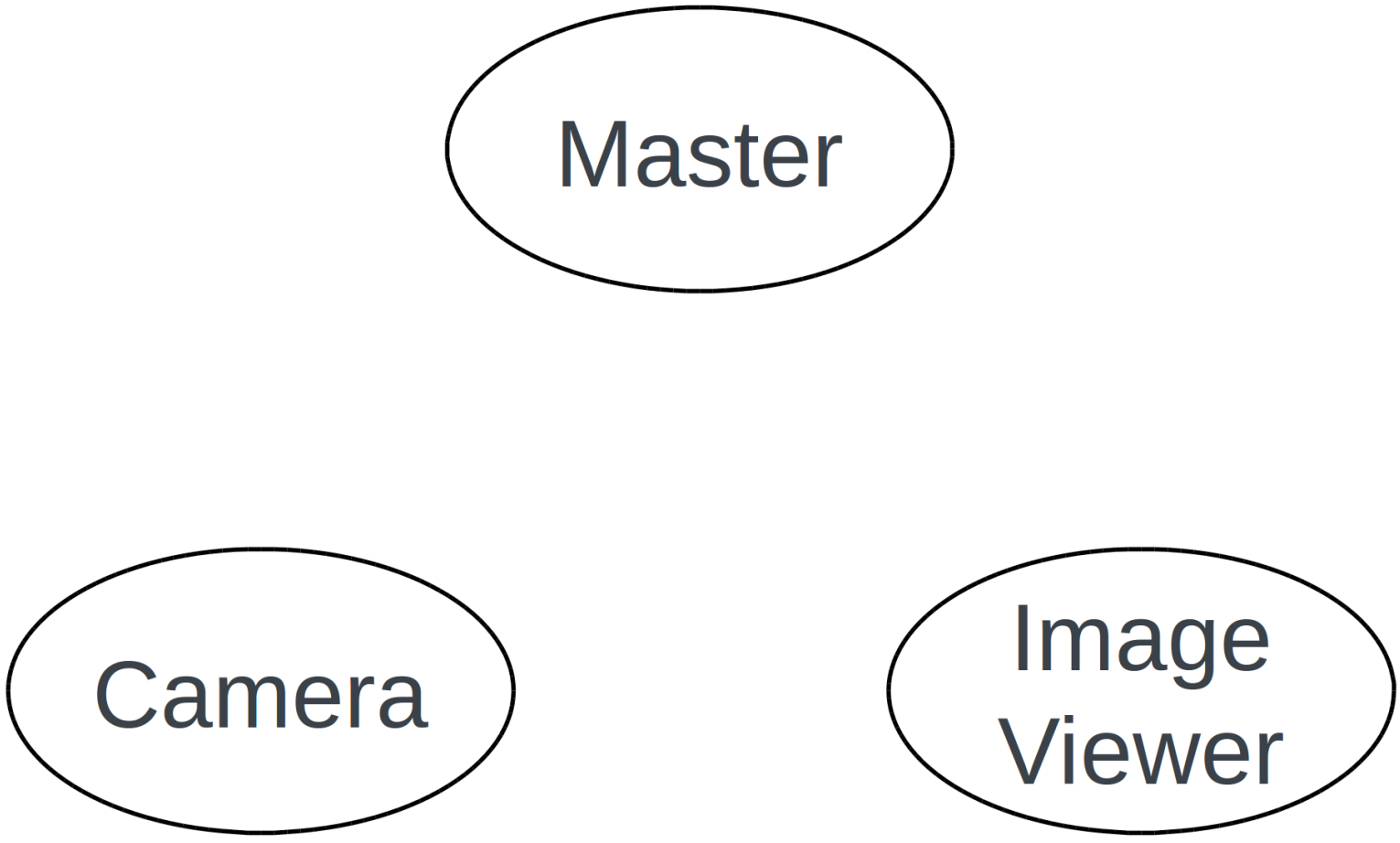
Summary of yesterday's session

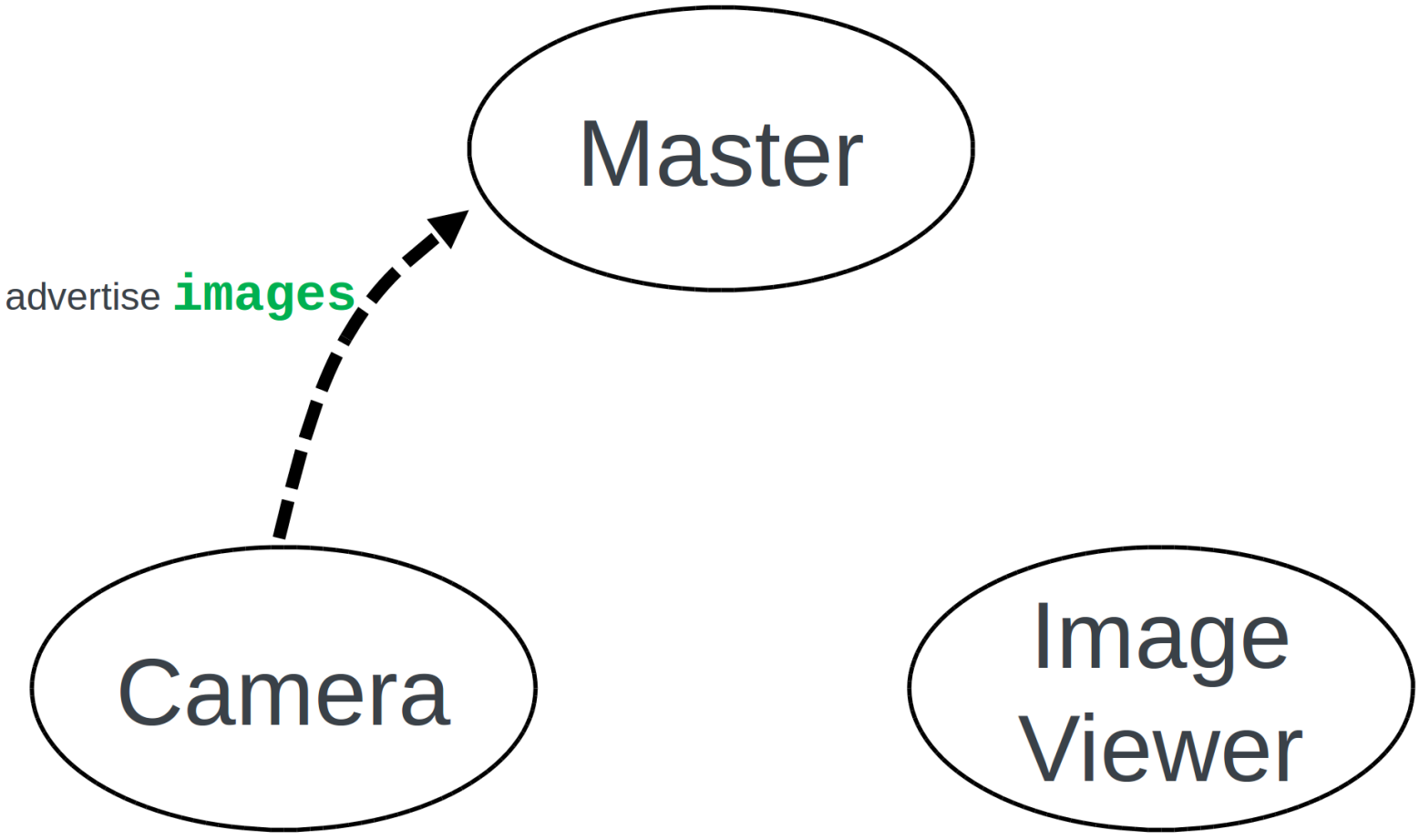
- ROS is a collection of libraries and tools that helps you when you develop software for robots.
- ROS provides several ways to transfer data between nodes:
  1. ROS topics and messages (**publish/subscribe**).
  2. ROS services (**request/reply**).
  3. ROS actions (**request/reply**).
  4. Parameter server.

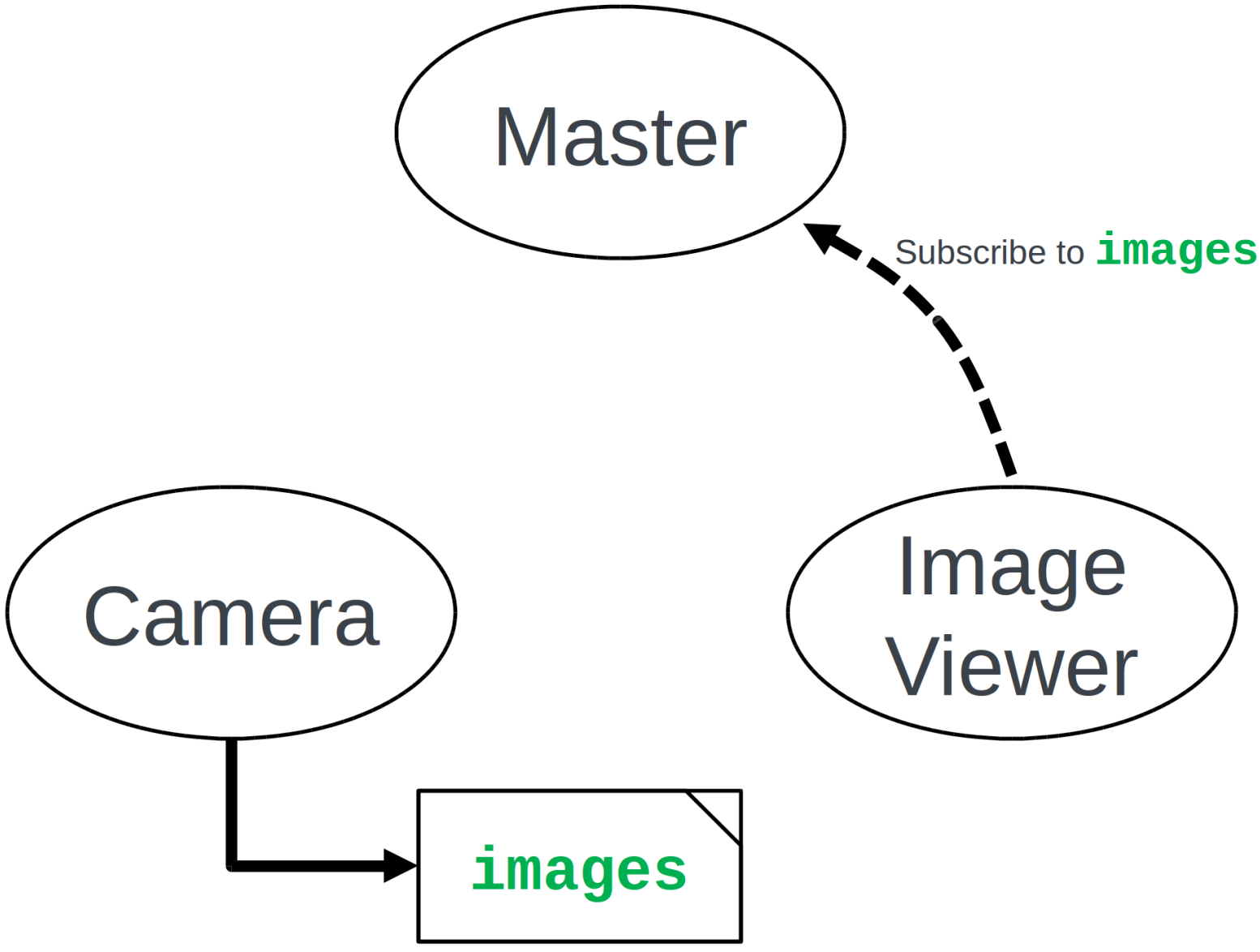
Recap

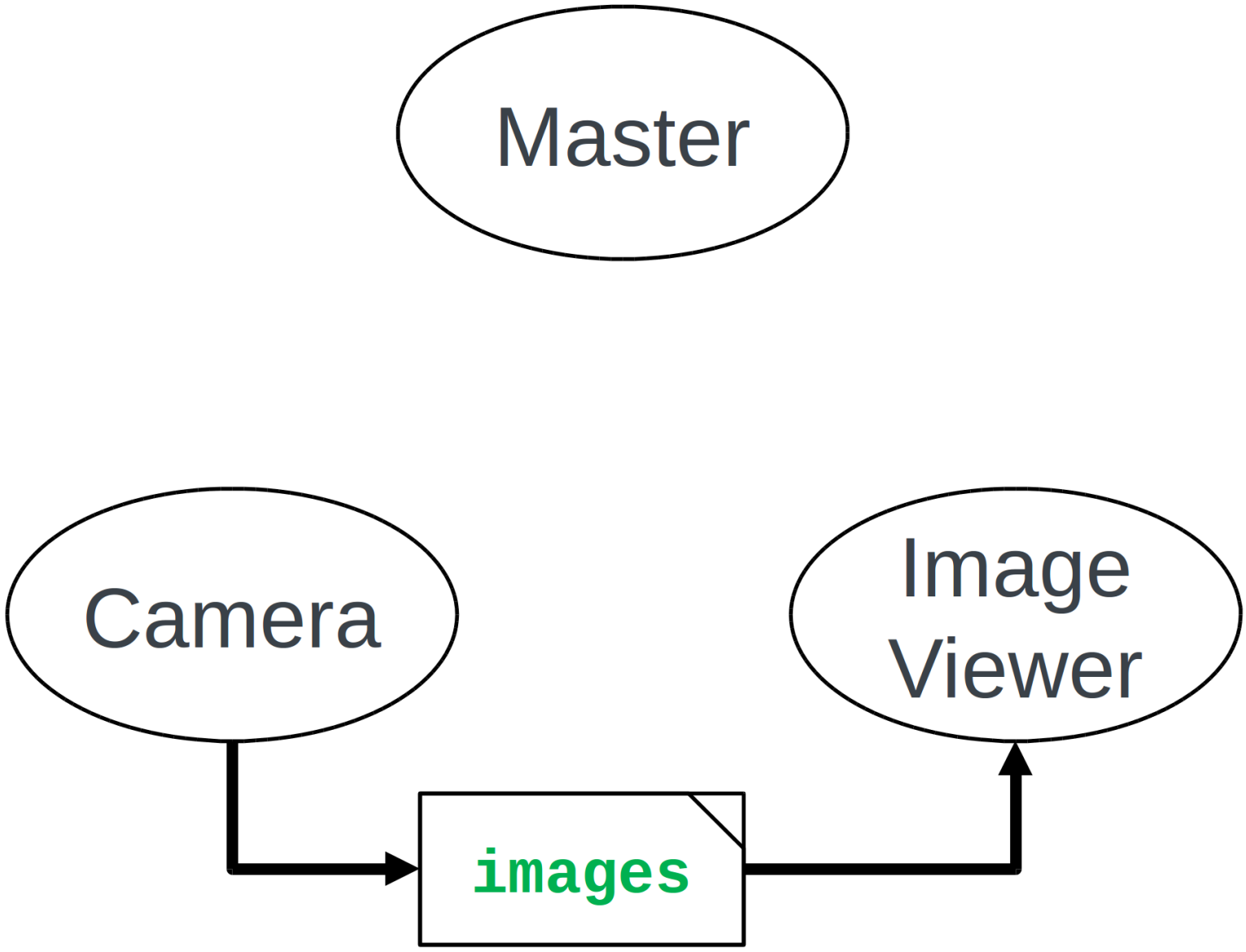
Summary of yesterday's session

- Today, we will focus on ROS topics and messages.


Master
Camera
Image
Viewer


Master
advertise images
Camera
Image
Viewer


Master
Subscribe to images
Camera
Image Viewer
images


Master
Camera
Image
Viewer
images

A simple ROS node

../scripts/00_simple_node.py

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


import rospy
from time import sleep

rospy.init_node("print_text")


while True:
    print "Hello world!"
    sleep(1)
```

A simple ROS node

../scripts/01_simple_node.py

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

import rospy


rospy.init_node("print_text")
rate = rospy.Rate(1)

while not rospy.is_shutdown():
    print "Hello world!"
    rate.sleep()
```

Writing a publisher node in Python

../scripts/02_simple_publisher.py

```
rospy.init_node('node name')
```

- nodes name must be unique. If you want to make sure the name of the node is unique:

```
rospy.init_node('node name', anonymous= True)
```

- node name will look like this:
  `/print_text_19637_1567065017476`

Three ways to run a node

ROS Nodes

There are 3 ways to run a node:

1. Like you normally do (not recommended). Example (in case of python node):

```
python <file name>
```

2. using rosrun command:

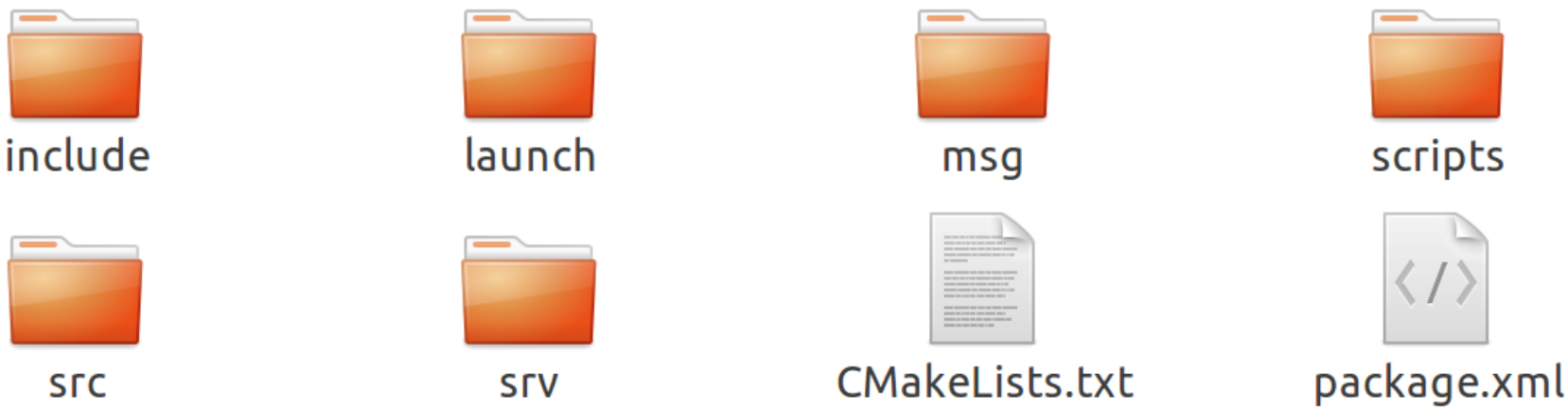
```
rosrun <package name> <node name>
```

3. Using launch files. (we'll see it later)

Let's create a package first!

ROS Nodes

- ROS commands find your files (python scripts, cpp files, launch files, message definitions) if they are located in a package inside the workspace.
- Normally, a package looks like this:



Let's create a package first!

ROS Nodes

- go to the README and do the steps for **creating a package**.

ROS commands

ROS Nodes

- Navigate to a ROS package directly:

```
roscd <package name>
```

- run a node without navigating to it's directory:

```
rosrun <package name> <executable>
```

Let's create a package first!

ROS Nodes

- go to the README and do the steps for **running a node**.

More ROS commands

ROS Nodes, Topics, and Messages

- List all the running nodes:

```
rosnode list
```

- Get more info about a certain node:

```
rosnode info <node name>
```

Writing a publisher node in Python

ROS Nodes, Topics, and Messages

- Let's extend our previous node and make it publish a String ROS message.

Writing a publisher node in Python

../scripts/02_simple_publisher.py

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

import rospy

from std_msgs.msg import String

rospy.init_node('talker')

pub = rospy.Publisher('myFirstTopic', String, queue_size=10)

rate = rospy.Rate(1)

my_message = String()
my_message.data = "Hello there! How are you?"

while not rospy.is_shutdown():
    pub.publish(my_message)
    rate.sleep()
```

Writing a publisher node in Python

../scripts/02_simple_publisher.py

```
rospy.Publisher(name, data_class, queue_size)
```

- `name`: Name of the topic to publish on.
- `data_class`: The type of message. It is a ROS message class.
- `queue_size`: The size of the outgoing message queue.

Writing a publisher node in Python

../scripts/02_simple_publisher.py

```
rospy.Publisher(
name,
data_class,
subscriber_listener=None,
tcp_nodelay=False,
latch=False,
headers=None,
queue_size=None
)
```

Writing a publisher node in Python

Things to note..

- ROS messages are implemented as classes.
- To publish a message you also need to define a **Publisher** class.
- Most ROS functionalities are implemented as classes.

Writing a publisher node in Python

ROS Nodes, Topics, and Messages

- Go to the README file and do the instructions of section: **some of ROS commands**.

More ROS commands

ROS Nodes, Topics, and Messages

- Get the current list of topics:

```
rostopic list
```

- Print published messages:

```
rostopic echo <topic name>
```

More ROS commands

ROS Nodes, Topics, and Messages

- Publish a message from terminal:

```
rostopic pub <topic name> <msg type> <msg>
```

- Get message type of a topic:

```
rostopic type <topic name>
```

How to use ROS messages

ROS Nodes, Topics, and Messages

- ROS messages are just classes with attributes you can fill.
- ROS messages are defined in separate files and have to be placed in a package. (will be covered today).
- The following command can be used to see the class attributes, or the description, of a ROS message:

```
rosmsg show <package/msg>
```

How to use ROS messages

Example

```
rosmsg show std_msgs/String
```

The output is:

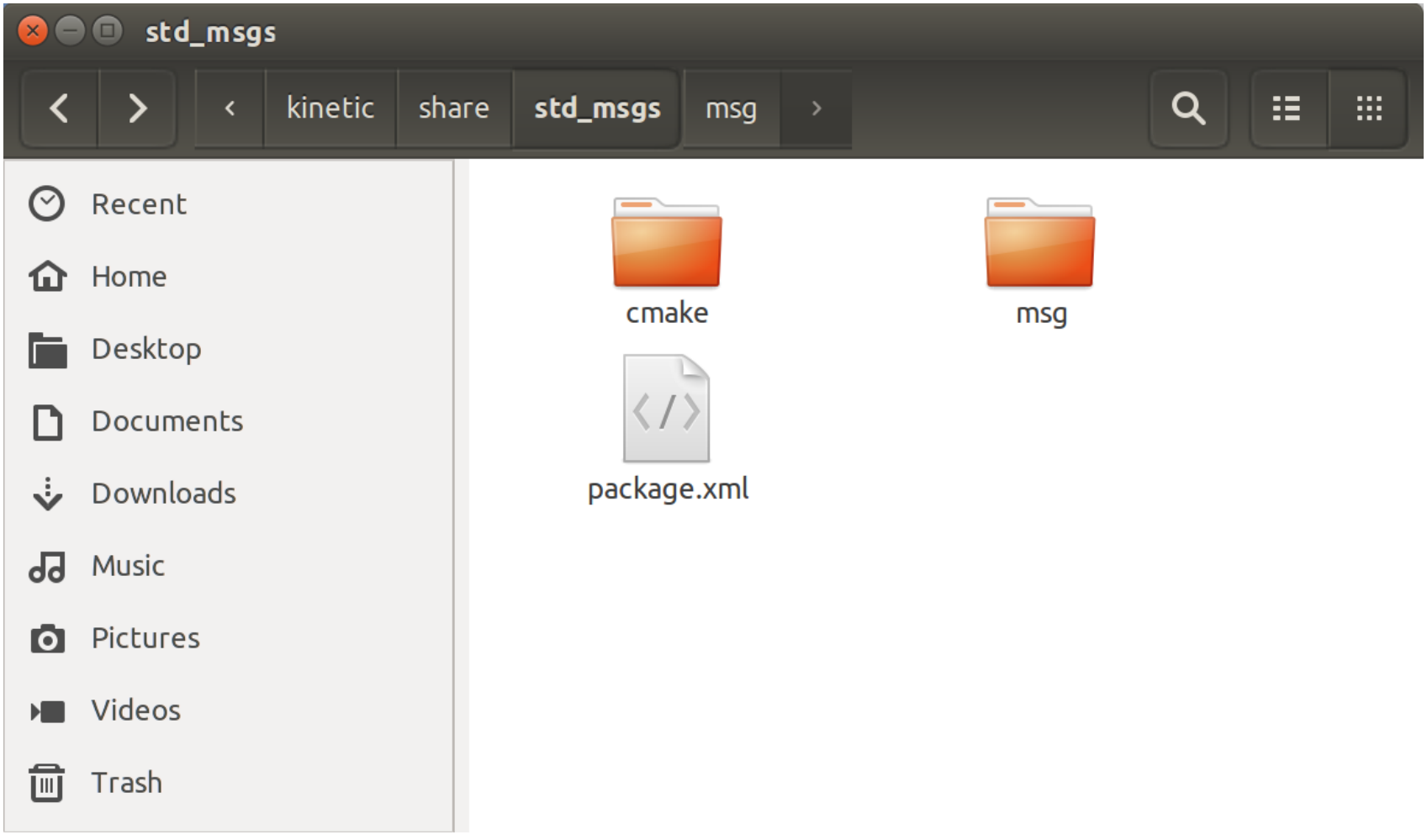
```
string data
```

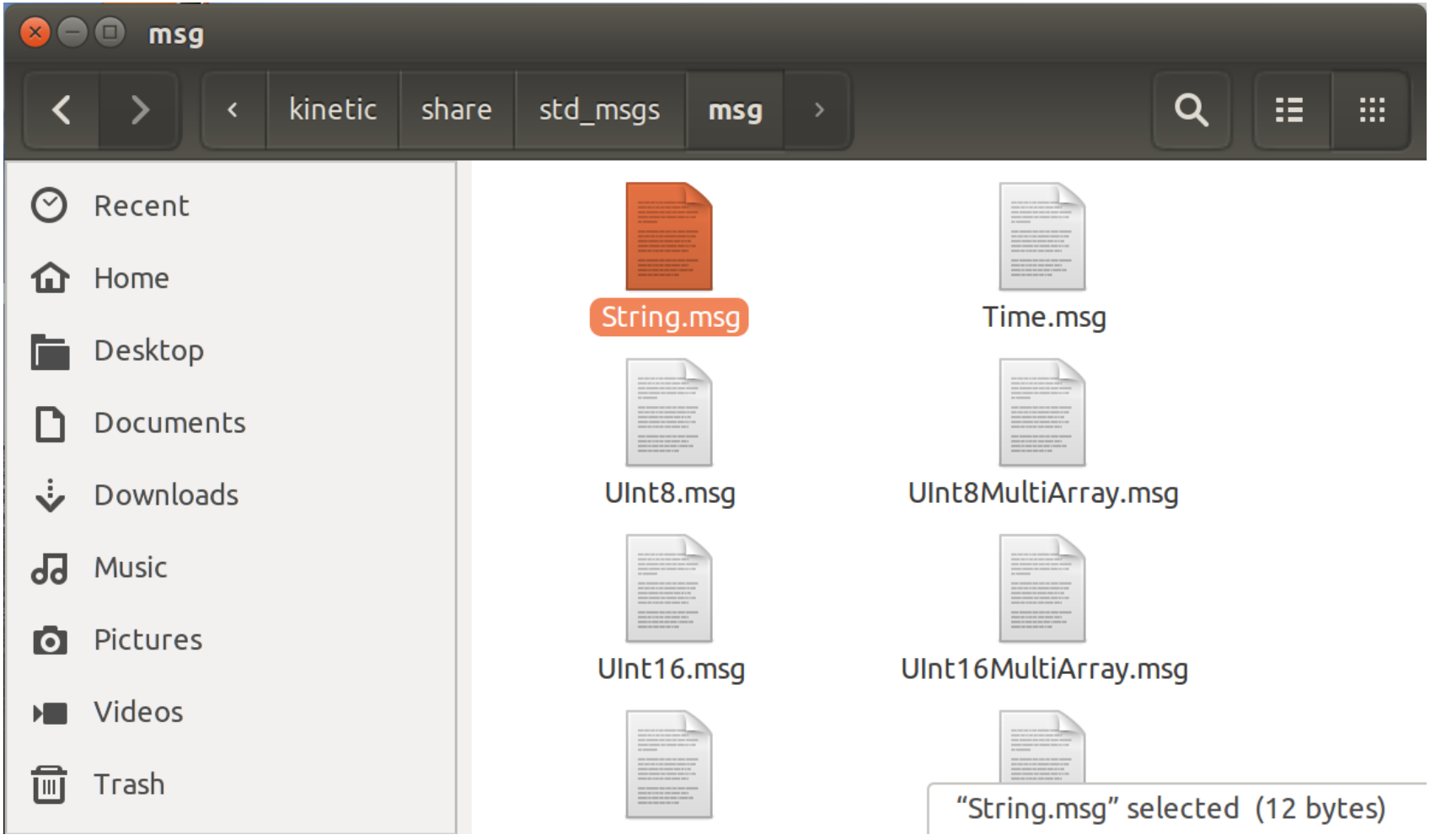
- It means ROS `String` message is a class with an attribute named `data` of type string (Python string).

How to use ROS messages

Importing a ROS message

- `String` message is located in the `std_msgs` package.


std_msgs
kinetic
share
std_msgs
msg
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
cmake
msg
package.xml



Note: this is not the file that is imported in our script though! this file is not even Python nor C++. We will see later what this file is.

How to use ROS messages

Importing a ROS message

Python

```python
from std_msgs.msg import String
```

C++

```cpp
#include "std_msgs/String.h"
```

Exercise 1

Writing a subscriber node in Python

../scripts/03_simple_subscriber.py

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

import rospy
from std_msgs.msg import String

def my_callback_function(msg):
    print msg

rospy.init_node('listener')
rate = rospy.Rate(100)
sub = rospy.Subscriber('myFirstTopic', String,
                       callback=my_callback_function)

while not rospy.is_shutdown():
    pass
```

Writing a subscriber node in Python

../scripts/04_simple_subscriber.py

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

import rospy
from std_msgs.msg import String


def my_callback_function(msg):
    print msg

rospy.init_node('listener')
rate = rospy.Rate(100)

rospy.Subscriber('myFirstTopic', String, callback=my_callback_function)

rospy.spin()
```

Writing a subscriber node in Python

Subscriber class

- go to the README, and let's do section:
  Simple subscriber

Writing a subscriber node in Python

Subscriber class

```
rospy.Subscriber(
name,
data_class,
callback=None,
callback_args=None,
queue_size=None,
buff_size=65536,
tcp_nodelay=False
)
```

Writing a subscriber node in Python

Subscriber class

- go to the README, and let's do section:
  Simple subscriber

General notes

more on ROS nodes

- You can define multiple publishers and subscribers in a node.
- You can call `init_node` function once only! (If you are spawning multiple processes in your script, then you can call it once for each process).
- To make your code look neat, you can wrap ROS stuff in a class and hide them!

Let's see an example with multiple publishers/subscribers and let's define them in a class...

Let’s see script
05_node_example.py

ROS Launch Files

- If you are working on a project that is comprised of multiple nodes, it's not practical to run each node manually everytime!
- ROS Launch files provide a way to run/launch multiple nodes at once.

ROS Launch Files

../launch/00_simple.launch

```
<!-- Example launch file to run the turtlesim node-->

<launch>

    <node pkg="turtlesim" type="turtlesim_node" name="turtle"/>

</launch>
```

ROS Launch Files

- To run a launch file:

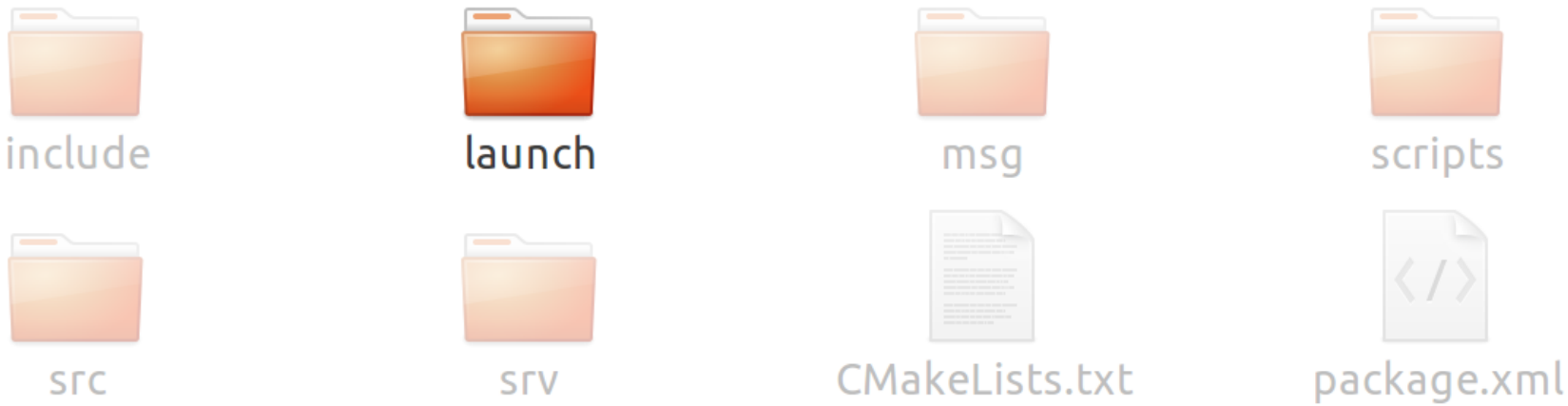
```
roslaunch <package name> <launch file>
```

- Example

```
roslaunch my_first_package 00_simple.launch
```

ROS Launch Files

- Ros launch files are XML files that consists for tags.
- They have a `.launch` file extension.
- You would ideally place them inside a `launch` folder inside your package.



ROS Launch Files

- With launch files, you can do the following:
  - Define nodes to be run.
  - Set ROS parameters on the parameter server.
  - Name remapping (will be covered later).
  - Define arguments for the launch file itself.
  - Include other launch files, and pass arguments to them.
  - Bring up the master (no need to call `roscore`).
  - Push a group of nodes into a separate namespace (will be covered later).
  - The list goes on..

ROS Launch Files

../launch/01_simple.launch

```
<!-- Example launch file to run
the turtlesim node and our script 06-->

<launch>

    <node pkg="turtlesim" type="turtlesim_node" name="turtle"/>

    <node pkg="my_first_package" type="06_parameters_example.py"
     name="commander"/>

</launch>
```

ROS Launch Files

- Go to the README, and do section
  Using launch files

Exercise 2

Names in ROS

- Resources in ROS computation graph (nodes, topics, parameters, etc..) must all have unique names.

Names in ROS

- Suppose we have a `camera` node that publishes on a topic named `image`
- what if you added a camera to your robot, and want to run two `camera` nodes?
- You will get name collision if you run the node twice!

Names in ROS

- ROS provides two mechanisms to avoid such situation:
  - Namespaces.
  - Remappings.

Namespaces

Names in ROS

- A node can be run under a namespace. For example, if we have two cameras, left camera, and right camera, the two `camera` nodes can have different namespaces:
- Node names:

```
/left/camera
/right/camera
```

- Topic names:

```
/left/image
/right/image
```

Namespaces

How to? Using rosrun

- From the terminal using `rosrun` command:

```
rosrun <package name> <node name> __ns:=<namespace>
```

Example:

```
rosrun turtlesim turtlesim_node __ns:=first
```

Node name becomes:

```
/first/turtlesim
```

- Notice the forward slash (/) is used as a separator.

Namespaces

How to? from a launch file

- Using the `group` tag, you can push down resource names under a chosen namespace.

Namespaces

How to? from a launch file

../launch/02_group_tag.launch

```
<!-- Example launch file using the group tag-->

<launch>

    <group ns="/first">
        <node pkg="turtlesim" type="turtlesim_node" name="turtle"/>
    </group>

</launch>
```

Name Remappings

- The second way ROS provides to change resource names is **remapping**.
- With name remapping, you can rename any resource. Examples:
  - rename a node.
  - Change the name of the topic a node publishes/subscribes to.
- Again, this is done without having to edit source code of the node. (node is reusable without altering it).

Name Remappings

How to? Using rosrun

- Example, from the terminal using `rosrun` command:

```
rosrun turtlesim turtlesim_node /turtle1/cmd_vel:=/vel
```

Changes topic name to:

/vel

- Example, from the terminal changing node name:

```
rosrun turtlesim turtlesim_node /turtlesim:=/second_turtle
```

Changes node name to:

/second_turtle

Name Remappings

How to? from a launch file

../launch/03_remap_tag.launch

```
<!-- Example launch file using remap tag-->

<launch>
    <node pkg="turtlesim" type="turtlesim_node" name="turtle">
    <remap from="/turtle1/cmd_vel" to="/vel"/>
    </node>
</launch>
```

Name resolution

- When you define a resource name (ex: node name, topic name, etc..), you can specify how names should be resolved.

Name resolution

- Example, suppose the snippet below is for a node named `talker` and has a namespace of `group1` (so the resolved node name is `/group1/talker`):

```
rospy.Publisher(myTopic, String, queue_size=10)
```

- The resolved topic name (what appears when you do `rostopic list`) will be:

```
/group1/myTopic
```

Name resolution

- The previous example shows **relative** name resolution, which is the default way names are resolved in ROS.
- The following table shows all types of name resolutions

Figures

Tables, graphs, and images

node name	syntax	type	resolved name
/group1/talker	`myTopic`	relative	/group1/myTopic
/group1/talker	`/myTopic`	global	/myTopic
/group1/talker	`~myTopic`	private	/group1/talker/myTopic

Parameter Server

- A network-shared dictionary accessible to all nodes.
- All nodes can access and modify those values.
- Parameter server is a part of ROS Master.

Parameter Server

Command line tools

- The `rosparam` command can do the following:
  - Get a list of parameter names currently held by the master.

```
rosparam list
```

  - Set/change the value of a parameter:

```
rosparam set <parameter name>
```

  - Get/fetch the value of a parameter:

```
rosparam get <parameter name>
```

Parameter Server

rospy interface to the parameter server

- A node can fetch/retrieve a parameter as follows:

```
rospy.get_param(param_name)
```

- It can optionally define a default value in case the parameter is not set in the parameter server:

```
rospy.get_param(param_name,default=value)
```

- A node can set a parameter on the parameter server:

```
rospy.set_param(param_name,param_value)
```

Let’s see script

06_parameters_example.py

Parameter Server

Using launch files

../launch/04_param_tag.launch

```
<!-- Example launch file using param tag-->

<launch>
<param name="rate" value="100"/>
<param name="radius" value="0.5"/>
<param name="angular_speed" value="1"/>

<node pkg="my_first_package" type="06_parameters_example.py" name="command
<node pkg="turtlesim" type="turtlesim_node" name="turtle"/>
</launch>
```

Parameter Server

Name resolution

- Name resolution applies also to parameter names (as we discussed earlier).
- The examples that follows, assume the node name is `move` and has a namespace of `/robot1`, so the resolved node name is `/robot1/move`.

Parameter Server

- Global naming:

```
var = rospy.get_param("/speed")
resolves to ⇒ /speed
```

- Relative naming:

```
var = rospy.get_param("speed")
resolves to ⇒ /robot1/speed
```

- Private naming:

```
var = rospy.get_param("~speed")
resolves to ⇒ /robot1/move/speed
```

Let’s use private parameter names instead of relative

07_parameters_example.py

Parameter Server

Using launch files

../launch/05_private_params.launch

```
<!-- Example launch file, setting private parameters-->

<launch>
<node pkg="my_first_package" type="07_parameters_example.py" name="command
<param name="rate" value="100"/>
<param name="radius" value="0.5"/>
<param name="angular_speed" value="1"/>
</node>
<node pkg="turtlesim" type="turtlesim_node" name="turtle"/>
</launch>
```

Exercise 3

Catkin

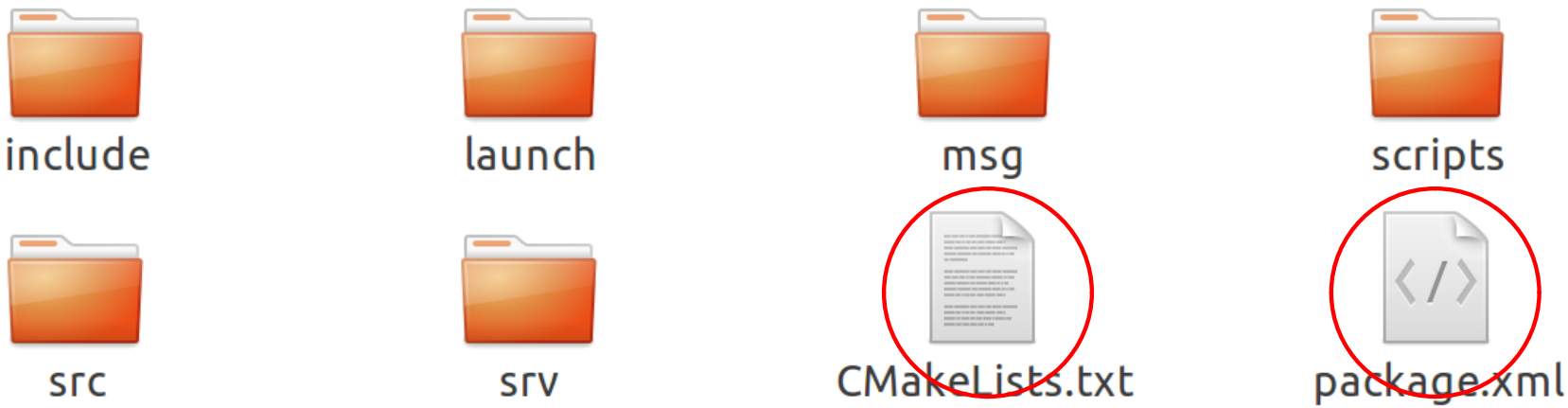
- In ROS, packages are compiled and built using Catkin.
- The build process is not only to compile C++ nodes, it's also used to generate source files (ex. Python and C++ files that define ROS messages as classes which you can import/include).
- Today, we will look at ROS message generation, and how to define custom ROS messages.

Catkin

- Catkin is not a compiler. It is a tool that handles the compilation process starting from source files up to generating executables.
- It invokes system's compiler (ex. g++, gcc, etc..) and handles your package dependencies.
- to do that, Catkin needs information on what to compile, where to find, etc...

A ROS package

- The `CMakeList.txt` file is what tells Catkin these things.
- `package.xml` file is where you add information about the package (author, maintainer, dependencies, etc..).



Catkin

- These files are auto-generated by `catkin_create_pkg` command.

Package Manifest

package.xml file

- Go to the package we have created earlier (`my_first_package`) and open the `package.xml` file there.

Package Manifest

package.xml file

- It's an XML file that consists of tags you need to fill.
- It defines properties of the package which include:
  - Name of the package: <name>.
  - Version number: <version>.
  - Description: <description>.
  - Package maintainer: <maintainer>.
  - License: <license>.
  - Package URL: <url>.
  - Author of package <author>.

Package Manifest

package.xml file

- It also defines package dependencies:
  - Build Dependencies: <build_depend>.
  - Build Export Dependencies: <build_export_depend>.
  - Execution Dependencies: <exec_depend>.
  - Test Dependencies: <test_depend>.
  - Build Tool Dependencies: <buildtool_depend>. (catkin)
  - Documentation Tool Dependencies: <doc_depend>
  - All of the above: <depend>.

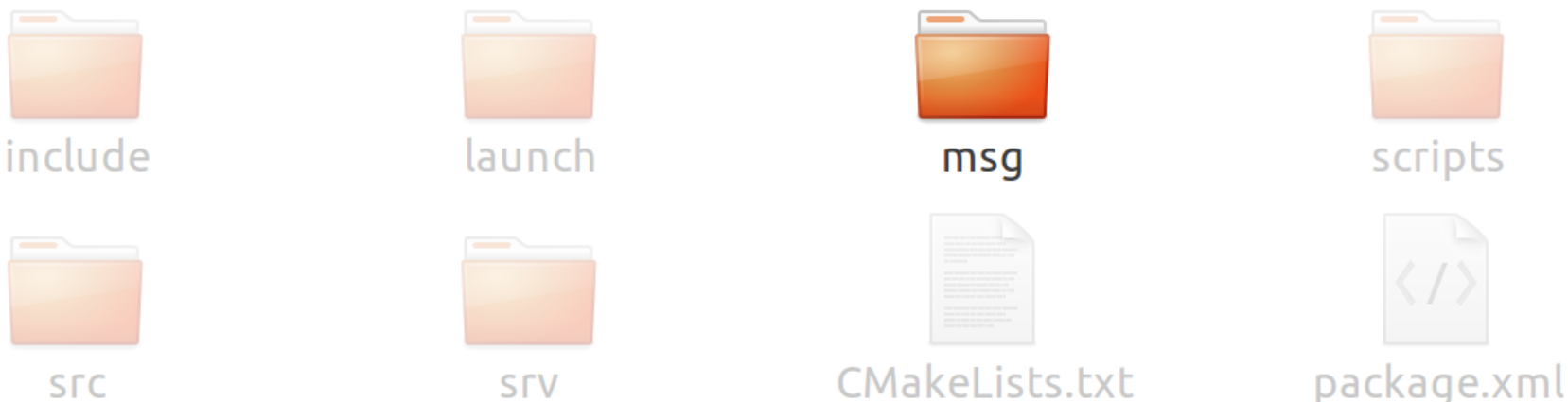
CMakeLists.txt

- The second file is (`CMakeLists.txt`). This defines how to build the package, what to compile, etc..
- Go to the package we have created earlier (`my_first_package`) and open the(`CMakeLists.txt`).

Custom ROS message

How to define them

- Custom messages are defined in separate text files.
- These files have `.msg` extension and have to be placed in `msg` folder inside the package.



Custom ROS message

msg file format

- The format is as follows:

```
fieldtype1    fieldname1
fieldtype2    fieldname2
fieldtype3    fieldname3
```

Custom ROS message

Examples

- `String.msg`:

```
string data
```

- The `String` message has one field of built-in type `string`.

Custom ROS message

Built-in types

- Some of the built-in field types:

Built-in type	C++	Python
bool	uint8_t	bool
int8	int8_t	int
int32	int32_t	int
string	std::string	string (str)
float32	float	float

Custom ROS message

Examples

- `Twist.msg`:

```
Vector3 linear
Vector3 angular
```

- It's possible to embed other message descriptions. In this example `Vector3` is another message.

Custom ROS message

Examples

- `Twist.msg`:

```
Vector3 linear
Vector3 angular
```

- `Vector3.msg`:

```
float64 x
float64 y
float64 z
```

Custom ROS message

- Let's define a custom message and see how to build the package!

- Go to the README and see section **Custom ROS messages**.

Custom ROS message

Behind the scenes

- Catkin uses these message description files to generate Python and C++ source files.
- The generated source files actually define the messages as classes, with public attributes matching the fields defined in the message description file.
- The generated source files are what you actually import or include in your nodes.

Custom ROS message

Behind the scenes

- This is why you need to build your package after writing a new `.msg` file. Otherwise you won't be able to import it.
- You need to explicitly tell Catkin about your `.msg` files and the dependencies needed to generate message classes.
- You do that in both the `CMakeLists.txt` and `package.xml` files.

References

- rospy full documentation (all the classes, all the functions ..etc):

  `http://docs.ros.org/kinetic/api/rospy/html/`
- ROS wiki
- On ROS names: “Programming Robots with ROS: a practical introduction to the Robot Operating System" Book, 2015, by Quigley and others.
- More on ROS names: `https://wiki.ros.org/Remapping%20Arguments`

Thank you

Any questions?