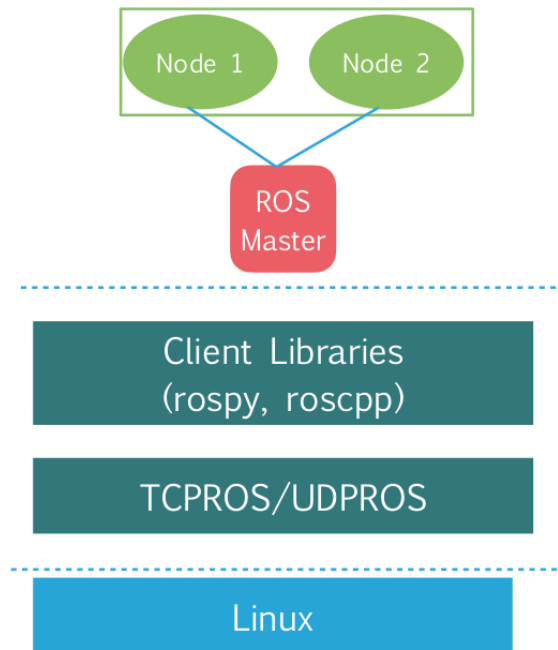
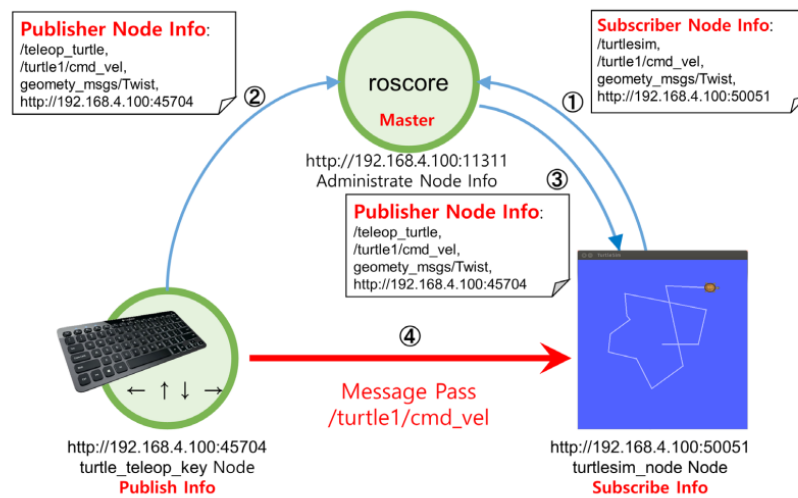


# Introduction

- **Topics:**
  1. ROS Basic and Foundation
  2. Motion in ROS
  3. Perception
  4. Interfacing with Arduino
- **Robot process cycle:**  
Sense → Think → Act
- **High-level API Vs Low-level API:**  
High: For making decisions, Thinking  
Low: Interfacing with actuators and sensors
- **ROS Architecture:**



- ❖ Run 1st ROS Master which will be a common communication point between all proceeding nodes
- ❖ Thus if any ROS Master crashes, all other dependent processes will crash.

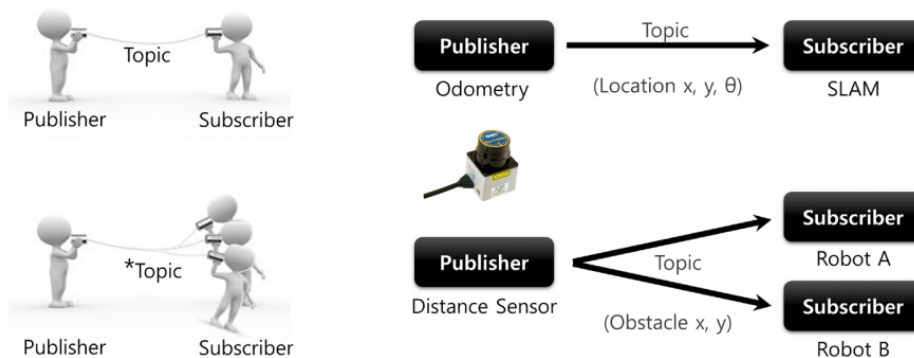


- **ROS Communication:**

Different nodes communicate with each other in 3 different ways:

1. **Publisher/ Subscriber**

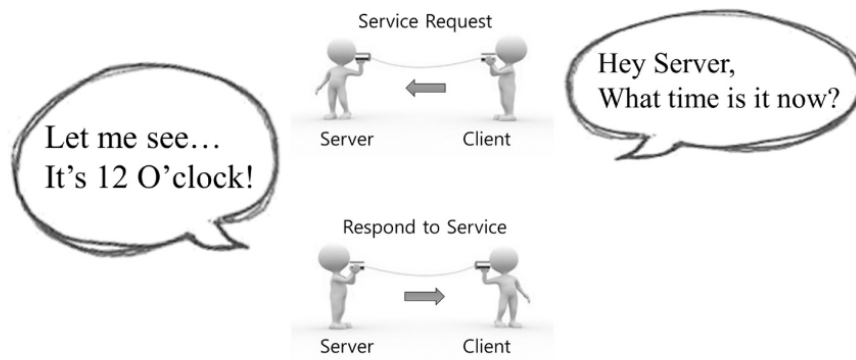
1:1, 1:N, N:N ways of publisher-subscriber communication.



2. **ROS Services**

It is a request-response type method of communication. It is a **synchronous** type of communication.

ROS service blocks the next lines of code until the request gets complete and the response receives from a node.

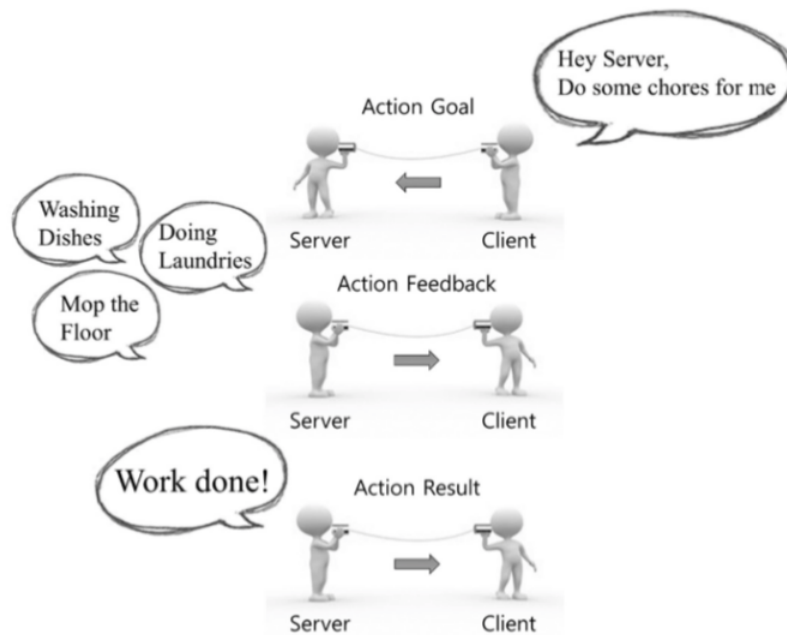


### 3. ROS Action lib

It is similar to ROS Service communication, just it is an **Asynchronous** type of communication.

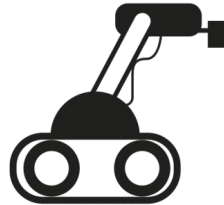
ROS Action will not block the program until the action gets complete.

Action Goal, Action Feedback, and Action Result



- **Limitation of ROS1:**

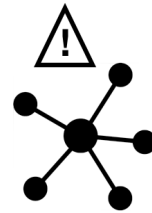
1. Designed for Single Robot case.
2. Not real-time. Need to set up the priority of each process during execution.
3. Need a reliable network of communication.
4. Single point of failure. (ROS Master)



Single Robot

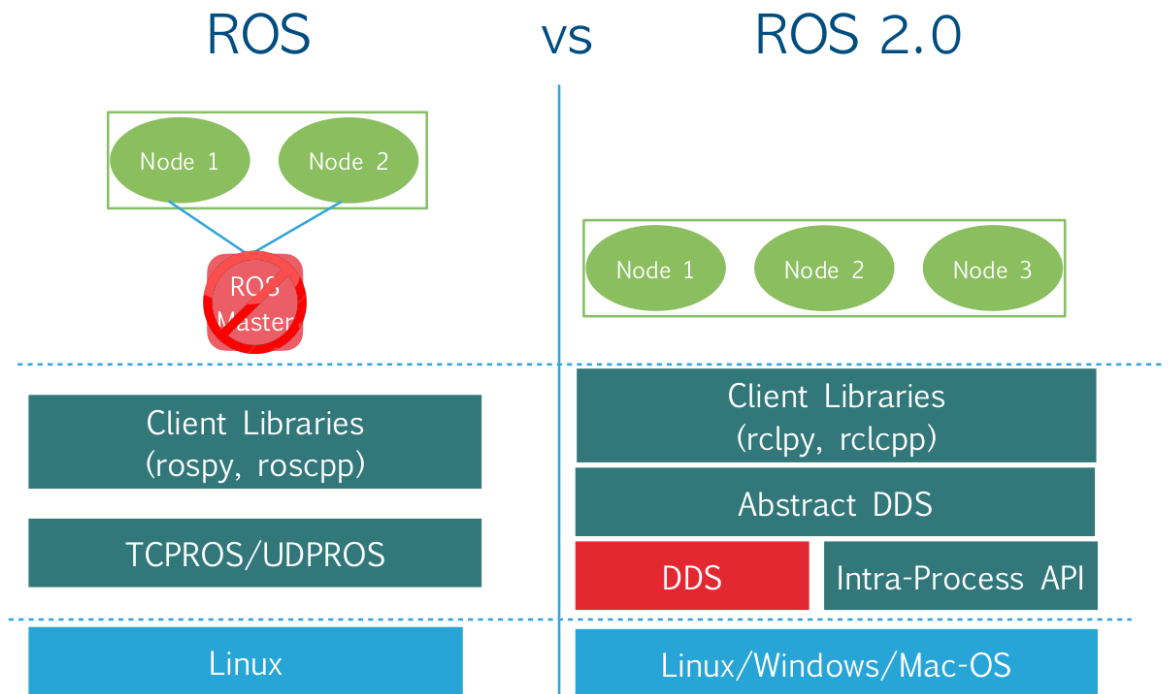


Not Real-Time



Need Reliable Network  
Single Point of Failure

- ROS1 Vs ROS2:



1. No central node.
2. Standard data distribution (DDS)
3. Cross-platform.
4. Multi-robot application.

5. Real-time services
  6. Middleware communication protocols.  
ZeroMQ, Protocol buffers, ZeroConf, DDS.
  7. DDS (Data Distribution Service)  
Data-centric Publish-Subscribe system (DCPS)  
Real-time M2M communication.
- Some more topics need to see:
    1. ROSLink Bridge
    2. ROSLink Proxy
    3. ROSLink Client
    4. ROSLink Cloud
  - Hardware Abstraction  
Low-level device control  
Message Passing  
Package Management

# ROS Concepts

- 3 main levels of ROS are:
  1. Filesystem level
  2. Computation graph level
  3. Community-level

## 1. Filesystem level:

- It is related to ROS resources encountered with storage on disk.
- **Package:** It is an organized ROS software that contains nodes, libraries, datasets, and configuration files.
- **Metapackage:** Group of related other packages.
- **Package manifest:** .xml file contains name, version, description, dependencies, etc.
- **Repositories:** Collection of packages.
- **Message (msg) file:** A message description file that defines message data structure.  
my\_package/msg/MyMsgType.msg
- **Service (srv) type:** A service description file. It contains a request/response data structure.

## 2. Computation Graph Level:

- **Nodes:** Processes that perform computation written using either ROSCPP or ROSPY.
- **Master:** Middleman.
- **Parameter server:** Stores data. It is a part of the master.
- **Message:**
- **Topics:**
- **Service:** Request/reply communication type.
- **Bags:** A way to store and playback ROS Message data. It stores sensor data. Useful during simulation.

# ROS Computation Graph

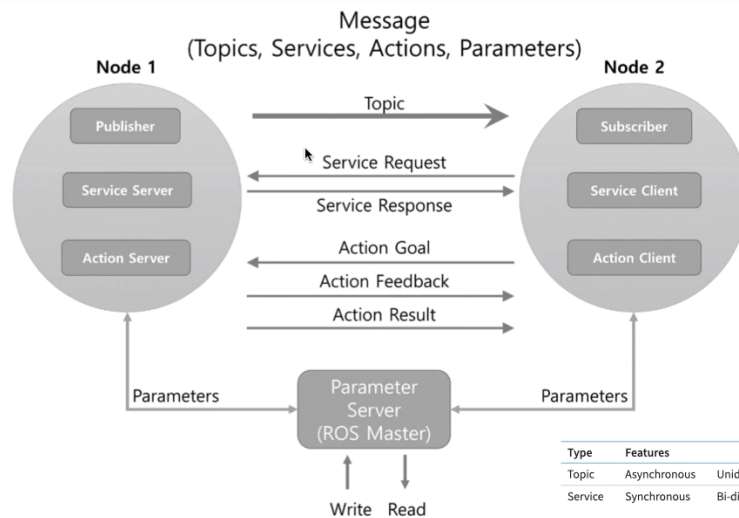


FIGURE 4-1 Message Communication between Nodes

Type	Features	Description
Topic	Asynchronous Unidirectional	Used when exchanging data continuously
Service	Synchronous Bi-directional	Used when request processing requests and responds current states
Action	Asynchronous Bi-directional	Used when it is difficult to use the service due to long response times after the request or when an intermediate feedback value is needed

TABLE 4-1 Comparison of the Topic, Server, and Action

## 3. Community-level:

- **Distributions:**
- **Repositories:**
- **ROS Wiki:**

# **ROS Installation**

1. Install ROS as per instructions given on the website.

<http://wiki.ros.org/ROS/Installation>

2. Set up a catkin environment/workspace.
  - A. Create a workspace folder (project folder)

***mkdir -p catkin\_ws/src***

- B. Inside catkin\_ws folder, run catkin\_make to create a catkin environment.

***catkin\_make***

- C. Source the setup file in devel folder.

***source /devel/setup.bash***

- D. Add path of project to ROS PACKAGE PATH

***echo \$ROS\_PACKAGE\_PATH  
/home/sanket/....catkin\_ws/src:/opt/ros/noetic/share***

- E. Add source file to .bash so no need to run everytime. First open .bashrc file.

***nano ~/.bashrc***

Then add the following commands in the .bashrc file.

***source /home/sanket/....catkin\_ws/devel/setup.bash***

3. Notes to be remembered.
  - The recent source command in .bashrc will be executed and considered as a current project file.
  - Every time any changes are made in the project, need to run the ***catkin\_make*** command so it will get execute the project in ROS.
- ROS is installed in /opt/ros/noetic folder. It consists of different files which act as a ROS workspace and a working directory for ROS.



- This workspace consists of different types of setup files for different types of terminal environments. These files need to be run every time you want to use ROS. Thus it was added to the .bashrc folder.
- It is not recommended to use the default ROS directory for the project. Instead, we need to create a separate folder for each project as used as an environment. For this follow step 2.

# Create Subscriber & Publisher with ROS

- Now catkin environment is created, the next step is to create a package inside the **src** folder. The package folder is like a project in an environment that contains all programs in python, cpp, and scripts.
- Command to create a package. It needs to be executed inside the src folder.

*catkin\_create\_pkg my\_package\_name dependency1 dependency2 .....*

Eg. *catkin\_create\_pkg ros\_basics\_tutorial roscpp std\_msgs rospy*

- After package creation also we can add dependencies to it.
- Again compile the library with the *catkin\_make* command.
- Once this command executes we can see the package folder created inside the src folder.
- Any package folder contains
  1. **CMakeLists.txt**: It contains information about messages, services, and actions.
  2. **Package.xml**: It contains information about the dependencies of a package, its version, name, and description. We can customize our package by making changes to package.xml.
  3. **src** folder: It contains all scripts of programs that we developed. .py, .cpp, etc.
- Let's create the first listener and talker node inside the **src** folder of the package.
- For creating python executables for ROS then store them inside the **script** folder inside the **src** folder. *src/script*
- After writing the python script we need to make them executables by the following commands:

*chmod +x listener.py*
- Again run **catkin\_make**. To build the package.
- Talker.py:

```

src > ros_basics_tutorial > src > scripts > talker.py > talker
1  #!/usr/bin/env python3
2
3  import rospy
4  from std_msgs.msg import String
5
6  def talker():
7      pub = rospy.Publisher('chatter', String, queue_size=10)
8      rospy.init_node('talker_py', anonymous=True)
9      rate = rospy.Rate(10)
10
11     count = 0
12     while not rospy.is_shutdown():
13         hello_str = "hello world {}".format(count)
14         count+=1
15         rospy.loginfo(hello_str)
16         pub.publish(hello_str)
17         rate.sleep()
18
19 if __name__ == '__main__':
20     try:
21         talker()
22     except rospy.ROSInterruptException:
23         pass

```

- listener.py

```

src > ros_basics_tutorial > src > scripts > listener.py > ...
1  #!/usr/bin/env python3
2
3  import rospy
4  from std_msgs.msg import String
5
6  def callback(data):
7      rospy.loginfo(rospy.get_caller_id() + "I heard {}".format(data.data))
8
9  def listener():
10     rospy.init_node('listener_py', anonymous=True)
11     rospy.Subscriber('chatter', String, callback)
12     rospy.spin()
13
14 if __name__ == '__main__':
15     listener()
16

```

- Listener.cpp:

```
src > ros_basics_tutorial > src > G+ listener.cpp > main(int, char **)
1  #include "ros/ros.h"
2  #include "std_msgs/String.h"
3
4  void chatterCallback(const std_msgs::String::ConstPtr& msg)
5  {
6      ROS_INFO("[Listener] I heard: [%s] \n", msg->data.c_str());
7  }
8
9  int main(int argc, char **argv)
10 {
11     ros::init(argc, argv, "listener_node");
12     ros::NodeHandle node;
13
14     ros::Subscriber sub = node.subscribe("chatter", 1000, chatterCallback);
15
16     ros::spin();
17
18     return 0;
19 }
```

- Talker.cpp:

```
src > ros_basics_tutorial > src > G+ talker.cpp > ...
1  #include "ros/ros.h"
2  #include "std_msgs/String.h"
3  #include <sstream>
4
5  int main(int argc, char **argv)
6  {
7      ros::init(argc, argv, "talker_node");
8
9      ros::NodeHandle n;
10
11     ros::Publisher chatter_publisher = n.advertise<std_msgs::String>("chatter", 1000);
12
13     ros::Rate loop_rate(0.5);
14
15     int count = 0;
16     while (ros::ok())
17     {
18         std_msgs::String msg;
19
20         std::stringstream ss;
21         ss<< "Hello World" << count;
22
23         msg.data = ss.str();
24
25         ROS_INFO("[Talker] I published %s\n", msg.data.c_str());
26
27         chatter_publisher.publish(msg);
28
29         ros::spinOnce();
30         loop_rate.sleep();
31         count++;
32     }
33
34     return 0;
35 }
```

- In terms of cpp nodes, we need to proceed few more steps. We need to edit the **CMakeLists.txt** file from the same package folder.

- **#talker**

```
add_executable(talker_ros_node src/talker.cpp)
target_link_libraries(talker_ros_node ${catkin_LIBRARIES})
#add_dependencies(talker gaitech_doc_generate_message_cpp)
```

- **#listener**

```
add_executable(listener_ros_node src/listener.cpp)
target_link_libraries(listener_ros_node ${catkin_LIBRARIES})
```

- Add the above two statements into the txt file and save it. Again run *catkin\_make*.

- <http://wiki.ros.org/rospy/Overview>

- Now let's analyze:

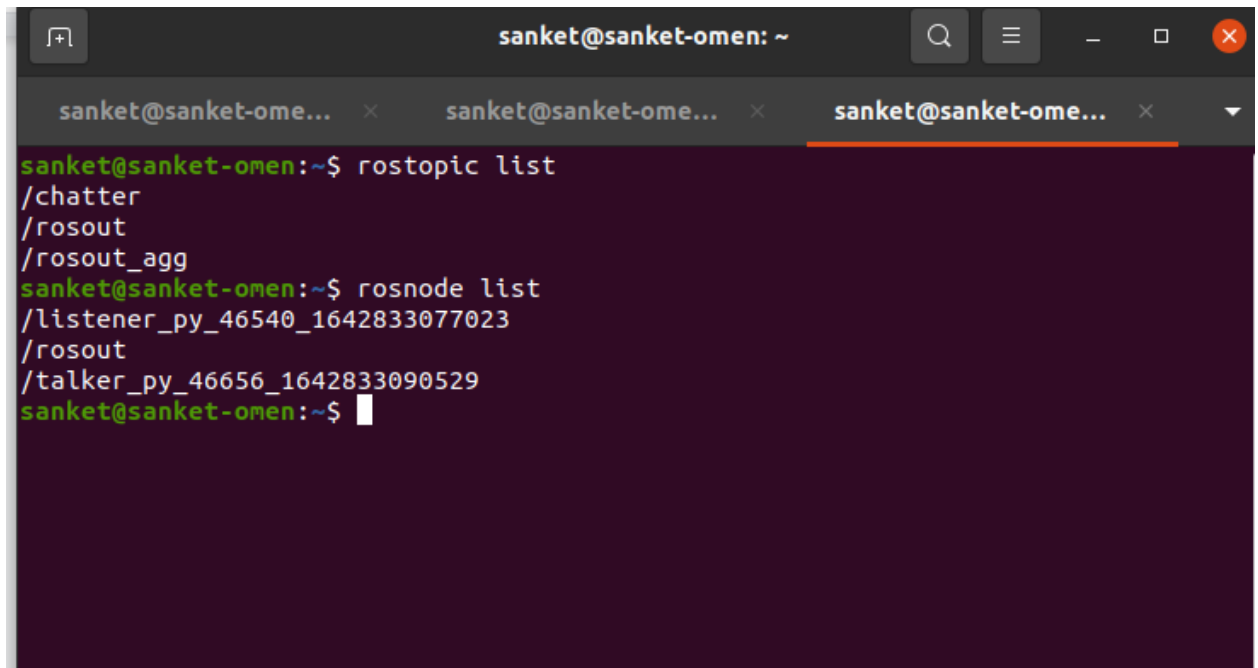
- Every time we run any node we can see live nodes via command:

*rostopic list*

- Also similarly we can get a list of all active topics:

*rostopic list*

- 

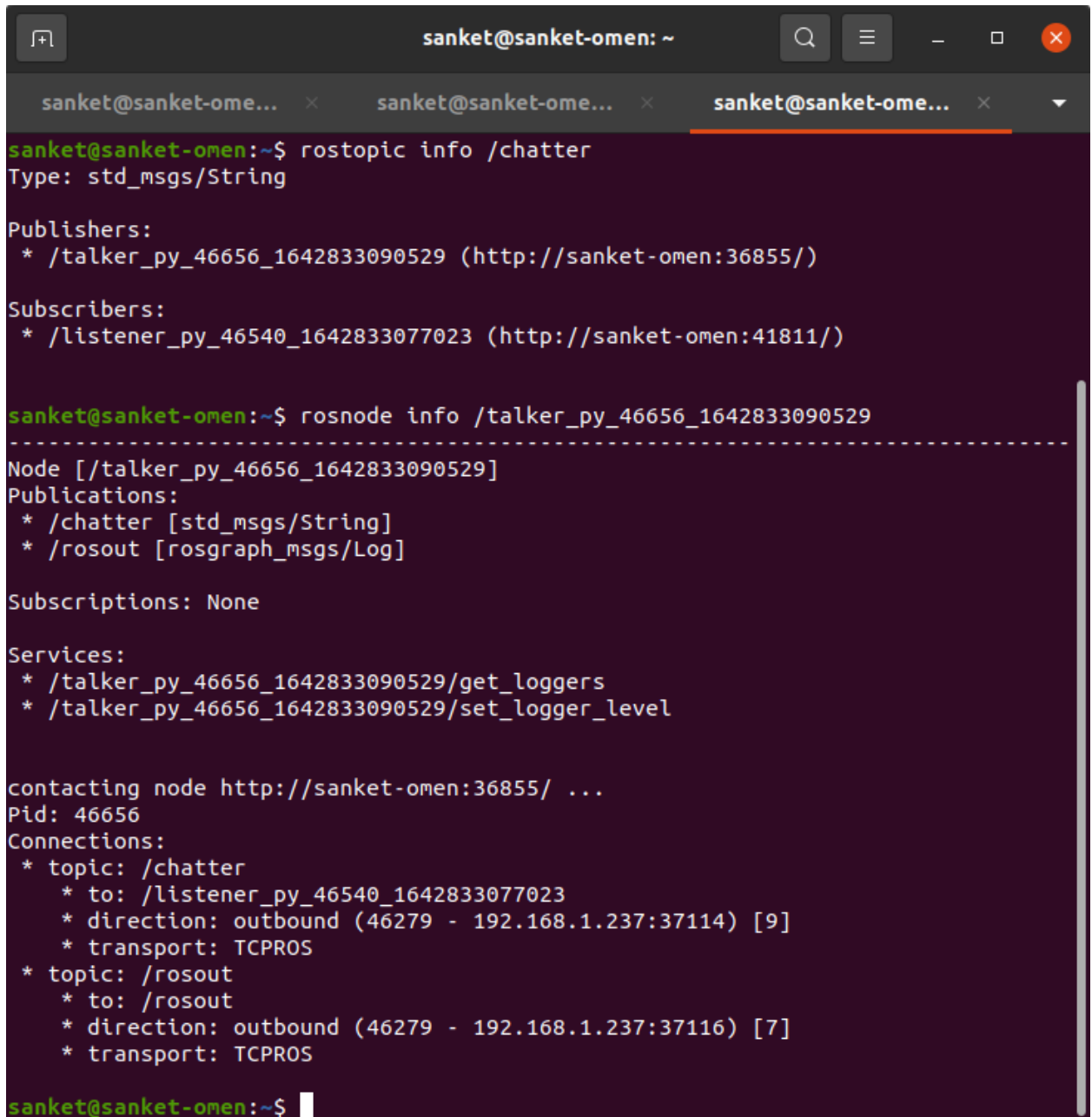


```
sanket@sanket-omen: ~
sanket@sanket-ome... x sanket@sanket-ome... x sanket@sanket-ome... x
sanket@sanket-omen:~$ rostopic list
/chatter
/rosout
/rosout_agg
sanket@sanket-omen:~$ rosnodetopics list
/listener_py_46540_1642833077023
/rosout
/talker_py_46656_1642833090529
sanket@sanket-omen:~$
```

- We can see info on each topic and node by following commands:

*rostopic info /topic\_name*

*rostopic info /node\_name*

A terminal window with a dark purple background and white text. The window title is 'sanket@sanket-omen: ~'. There are three tabs open, all with the same title. The terminal shows two commands and their outputs. The first command is 'rostopic info /chatter', which outputs the message type 'std\_msgs/String', a list of publishers, and a list of subscribers. The second command is 'rostopic info /talker\_py\_46656\_1642833090529', which outputs detailed information about a specific node, including its publications, subscriptions, services, and network connections.

```
sanket@sanket-omen:~$ rostopic info /chatter
Type: std_msgs/String

Publishers:
* /talker_py_46656_1642833090529 (http://sanket-omen:36855/)

Subscribers:
* /listener_py_46540_1642833077023 (http://sanket-omen:41811/)

sanket@sanket-omen:~$ rostopic info /talker_py_46656_1642833090529
-----
Node [/talker_py_46656_1642833090529]
Publications:
* /chatter [std_msgs/String]
* /rosout [roscpp_msgs/Log]

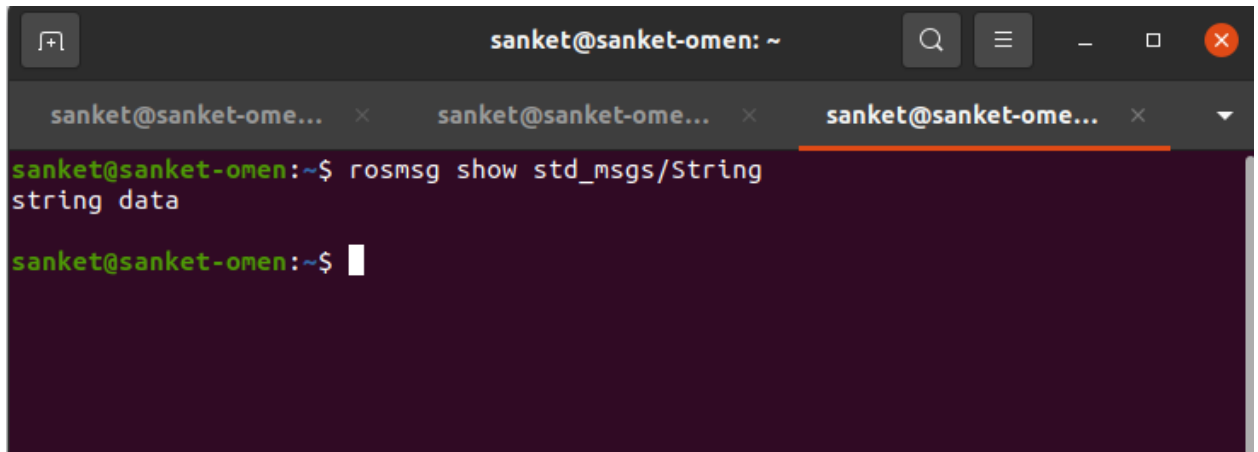
Subscriptions: None

Services:
* /talker_py_46656_1642833090529/get_loggers
* /talker_py_46656_1642833090529/set_logger_level

contacting node http://sanket-omen:36855/ ...
Pid: 46656
Connections:
* topic: /chatter
  * to: /listener_py_46540_1642833077023
  * direction: outbound (46279 - 192.168.1.237:37114) [9]
  * transport: TCPROS
* topic: /rosout
  * to: /rosout
  * direction: outbound (46279 - 192.168.1.237:37116) [7]
  * transport: TCPROS

sanket@sanket-omen:~$
```

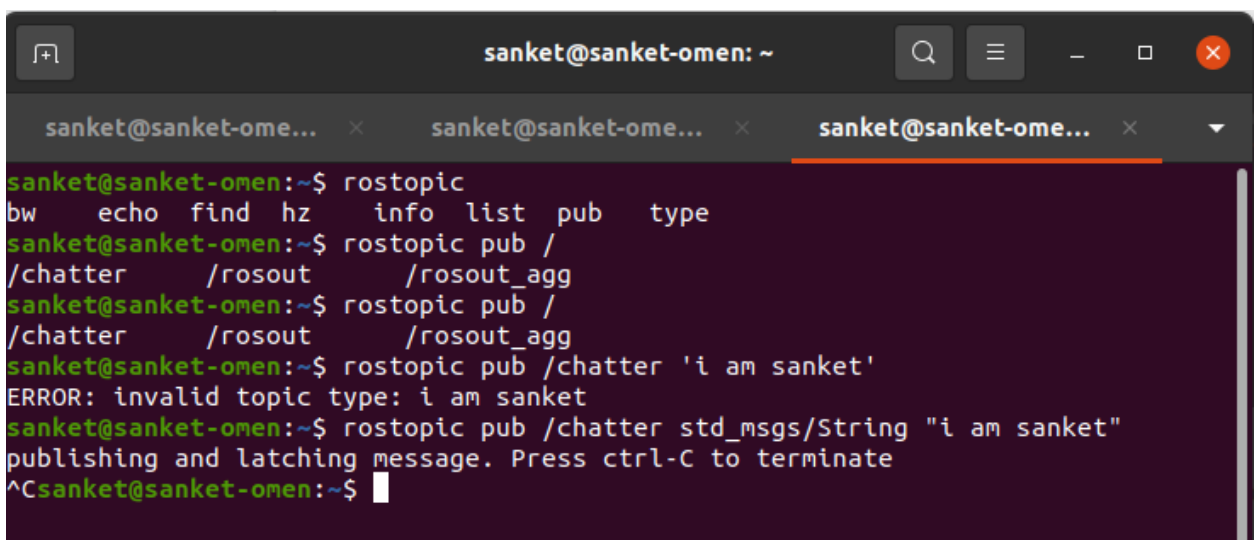
- Here let's focus on the message type. Info rostopic info we can see the topic is sending a message of type *std\_msgs/String*.  
std\_msgs is the Python package for message and String is the ROS message type.
- To see the content of ROS Msg we can use the following command:  
*rostopic show std\_msgs/String*



```
sanket@sanket-omen: ~  
sanket@sanket-omen:~$ rosmmsg show std_msgs/String  
string data  
sanket@sanket-omen:~$
```

- Until now we used Python scripts to publish messages to a given topic in repetition. But we can publish using CMD. We need to use the following command:

*rostopic pub /topic\_name msg\_type "Your message"*  
*rostopic pub /chatter std\_msgs/String "I am Sanket"*



```
sanket@sanket-omen: ~  
sanket@sanket-omen:~$ rostopic  
bw echo find hz info list pub type  
sanket@sanket-omen:~$ rostopic pub /  
/chatter /rosout /rosout_agg  
sanket@sanket-omen:~$ rostopic pub /  
/chatter /rosout /rosout_agg  
sanket@sanket-omen:~$ rostopic pub /chatter 'i am sanket'  
ERROR: invalid topic type: i am sanket  
sanket@sanket-omen:~$ rostopic pub /chatter std_msgs/String "i am sanket"  
publishing and latching message. Press ctrl-C to terminate  
^Csanket@sanket-omen:~$
```

- Notes:
  1. During the process, if ROSMaster crashes, then currently running programs will continue, but it is not possible to start a new program. Also, if we stop any service, then we cant restart it.
- Tips for creating a Publisher:
  1. Step 1: Determine the **name of the Topic** to publish.
  2. Step 2: Determine the **type of message**.
  3. Step 3: Determine the **frequency of topic** publication.
  4. Step 4: **Create a publisher** object with the parameter chosen.

5. Step 5: **Keep publishing messages** on a given topic.
- Tips for creating a Subscriber:
    1. Step 1: Identify the **name of the topic** to listen to.
    2. Step 2: Identify the **type of message**.
    3. Step 3: **Define a callback function** that will automatically execute when any message will receive.
    4. Step 4: **Start listening** to the topic message.
    5. Step 5: **Spin** to listen forever.
  -



# Messages

- msg files contain information about messages that we send and receive between different nodes.
- Until now we just used pre-defined messages like std\_msgs and geometry\_msgs.
- Different variable **field** types for msgs:

*int32, int16, int8, int64*

*float32, float64*

*string*

*time, duration*

*variable length array [ ], fixed length array [ ]*

*Other msg files*

- The message name is composed of two parts:

*package\_name/message\_type*

Eg.

std\_msgs/String

geometry\_msgs/Twist

turtlesim/Pose

- Each message\_type has a **field** in its description.

**package\_name/message\_type**

**type1 field1**

**type2 field2**

Eg.

std\_msgs/String

string data

geometry\_msgs/Twist

Vector3 linear

Vector3 angular

geometry\_msgs/Vector3

float64 x

float64 y

float64 z

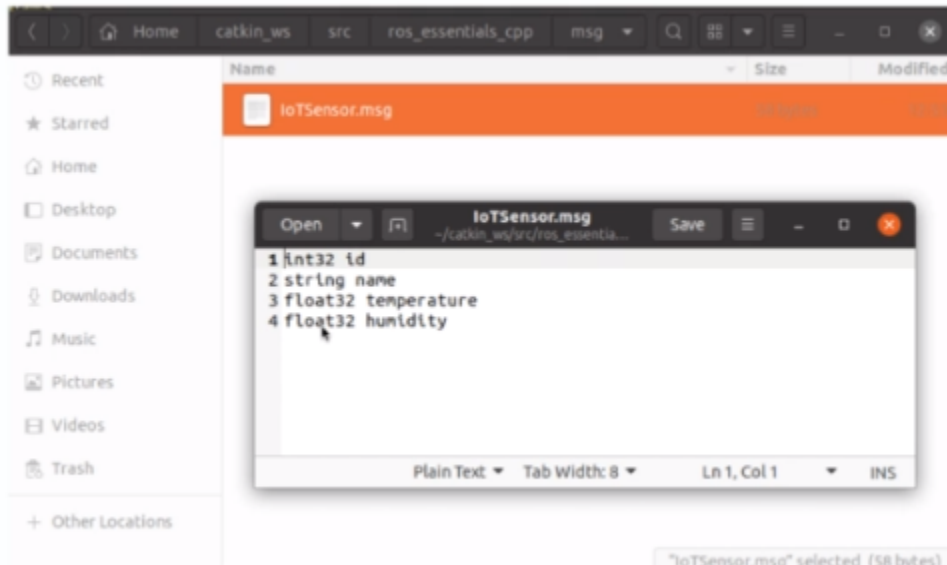
- As a field type, we can use the default variable field as mentioned above or we can use the custom build message type also.

As we see above, std\_msgs/String used the default msg-type **string** while

geometry\_msgs/Twist used a custom **Vector3** message type that contains 3 **floating** numbers as its field.

- These custom message files are stored in a **package\_name/msg** folder. For this, we need to create another new folder.

Eg. **ros\_basic\_tutorial/msg**



- Once we create our own custom message we need to edit the **package CMakeLists.txt** file so ROS will identify the message.
- Update the dependencies in CMakeLists.txt. We just need to uncomment these sections and add some texts.

```
# Do not just add this to your CMakeLists.txt, modify the existing text to add message_generation before the closing parenthesis
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
```

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

- Add message name in CMakeLists.txt file. All msg file names must be included.

```
add_message_files(
  FILES
  Num.msg
)
```

```
add_message_files(
  FILES
  Num.msg
)
```

- Also, we need to uncomment the **generate\_messages()** function in **CMakeLists.txt** so ROS will find this during execution. Make sure to add all dependencies in the function which we used while creating the message file.

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

- The last update need to be done for **catkin\_package()** function in **CMakeLists.txt** file. Add a message\_runtime command in the function so ROS will run the newly generated message during execution.

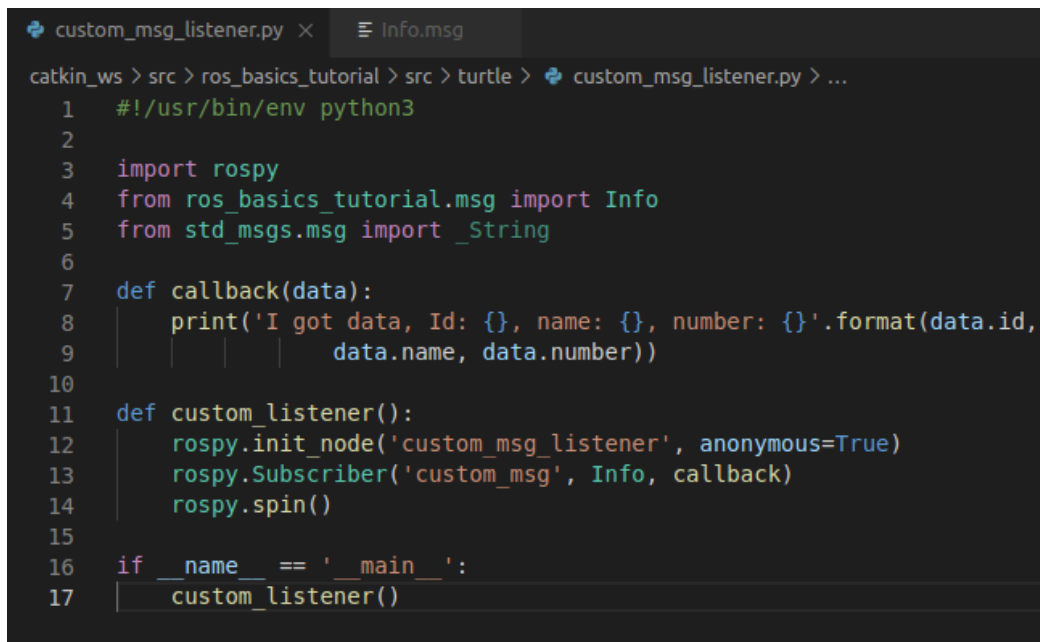
```
catkin_package(
  INCLUDE_DIRS include
  LIBRARIES ros_basics_tutorial
  CATKIN_DEPENDS roscpp rospy std_msgs message_runtime
  DEPENDS system_lib
)
```

```
catkin_package(
  INCLUDE_DIRS include
  LIBRARIES ros_basics_tutorial
  CATKIN_DEPENDS roscpp rospy std_msgs message_runtime
```

```
DEPENDS system_lib
)
```

- Other than a change in CMakeLists.txt we need to edit the **package.xml** file to turn the msg file into source code.
- Add the following lines into package.xml:

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```
- We need message\_generation during build time and message\_runtime during runtime.
- Steps to follow:
  1. Create a *message\_name.msg* file into *package\_name/msg* folder.
  2. Update *find\_package()* script into package CMakeLists.txt file.
  3. Update *add\_message\_files()* script into package CMakeLists.txt file.
  4. Update *generate\_messages()* script into package CMakeLists.txt file.
  5. Update *catkin\_package()* script into package CMakeLists.txt file.
  6. Add *<build\_depend>* and *<exec\_depend>* into package **package.xml** file.
  7. Run *catkin\_make*
  8. Check msg into ROS using: *rosmmsg show MessageName*.
- Use of custom message into the file.



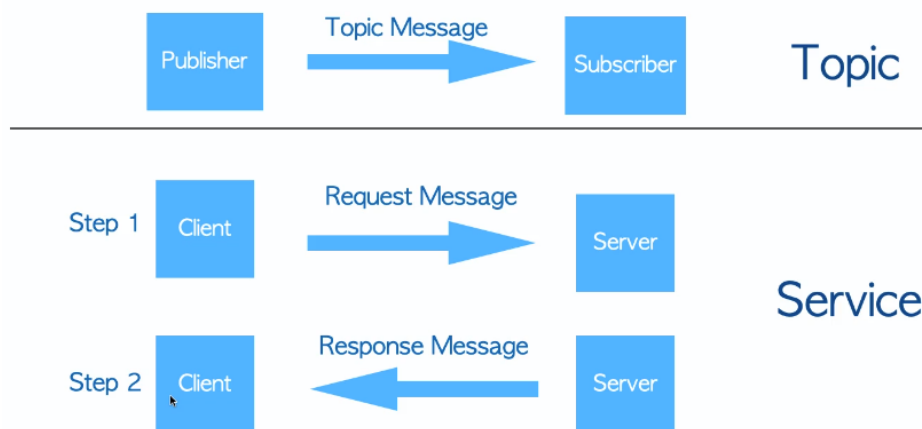
```
custom_msg_listener.py x Info.msg
catkin_ws > src > ros_basics_tutorial > src > turtle > custom_msg_listener.py > ...
1  #!/usr/bin/env python3
2
3  import rospy
4  from ros_basics_tutorial.msg import Info
5  from std_msgs.msg import _String
6
7  def callback(data):
8      print('I got data, Id: {}, name: {}, number: {}'.format(data.id,
9          data.name, data.number))
10
11 def custom_listener():
12     rospy.init_node('custom_msg_listener', anonymous=True)
13     rospy.Subscriber('custom_msg', Info, callback)
14     rospy.spin()
15
16 if __name__ == '__main__':
17     custom_listener()
```

```
talker.py listener.py custom_msg_talker.py x CMakeLists.txt
catkin_ws > src > ros_basics_tutorial > src > turtle > custom_msg_talker.py > custom_talker
1  #!/usr/bin/env python3
2
3  import rospy
4  from std_msgs.msg import String
5  from ros_basics_tutorial.msg import Info
6
7  def custom_talker():
8      custom_msg_pub = rospy.Publisher('custom_msg', Info, queue_size=10)
9      rospy.init_node('custom_msg_talker', anonymous=True)
10     rate = rospy.Rate(5)
11
12     id = 1001
13     name = 'Sanket'
14     number = 3.14
15
16
17     while not rospy.is_shutdown():
18         info_msg = Info()
19         info_msg.id = id
20         info_msg.name = name
21         info_msg.number = number
22         custom_msg_pub.publish(info_msg)
23         id += 1
24         number += 3.14
25         rate.sleep()
26
27
28
29 if __name__ == '__main__':
30     try:
31         custom_talker()
32     except rospy.ROSInterruptException:
33         pass
```

# Services

- ROS Service is a request-response type of communication. It contains one **ROS Server** node and one **ROS Client** node.
- A ROS Client will send a request and ROS Server will respond back.  
Client: Sending a request  
Server: Sending a response
- This is a **synchronous** type of communication. This Means the Client will wait for a response from Server and its execution will be paused for that duration.
- In the case of Topic, it is unidirectional communication. Only Publisher will send information. But in the case of Service, it is a **bidirectional communication**. This means both the client and server can send information.

## What is ROS Service?



- Application:
  1. When we want a robot to perform some action at that time we can send a service request to the robot server to perform an action.
- **rosservice list:** It gives all available services which are currently available.
- **rosservice info /service\_name** will give you information about the particular service. It will give information like Node name, URI, Type of the message, and argument for that message. This service message is different from the topic message that we saw earlier. It is a request-response type of message.
- **rossrv info message\_type:** This will give you information about the service message.

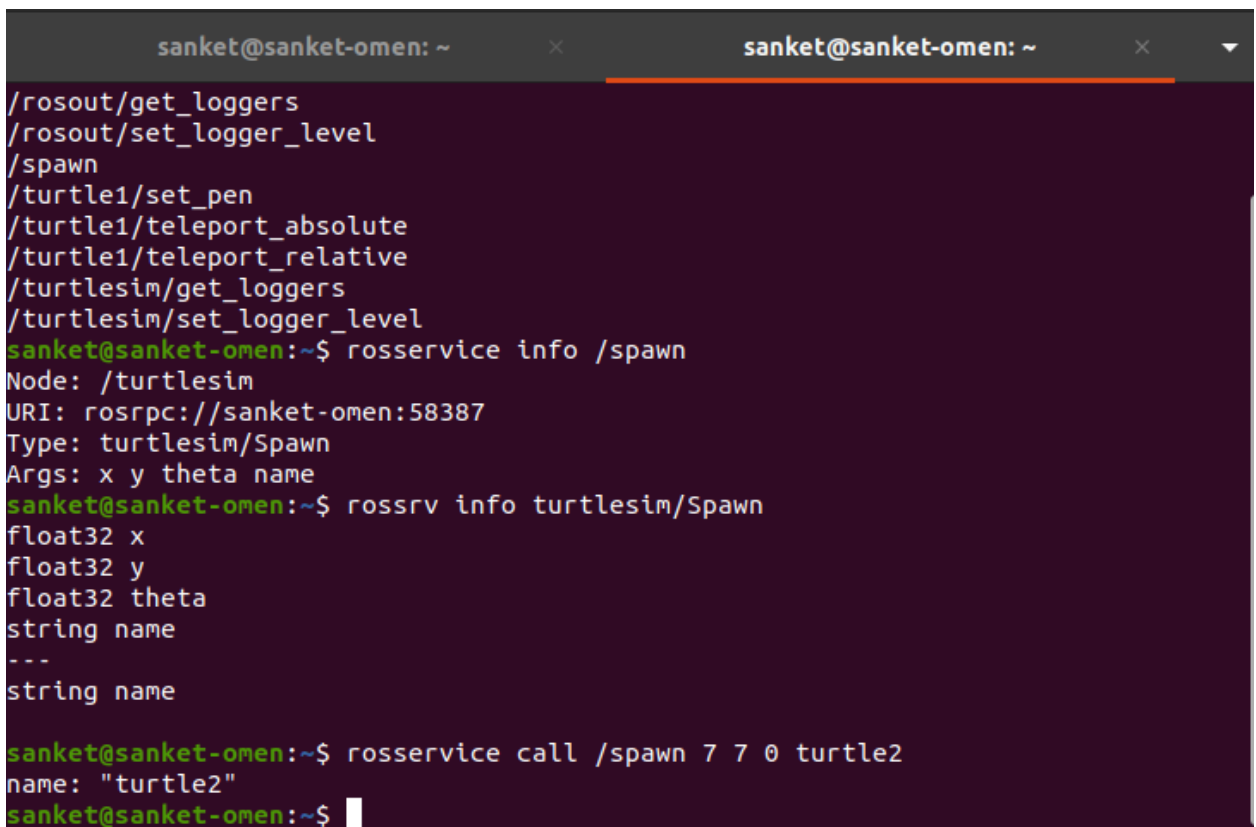
```
sanket@sanket-omen:~$ rossrv info turtlesim/Spawn
float32 x
float32 y
float32 theta
string name
---
string name
```

- This is one of the Service messages and its content. *First, 3 variables are arguments that the client will send to the service as a request* (request message) while *the last variable is a response from the server* (response message).

This service message is different from the topic message. In the topic message, there is only one section of the message i.e. from Publisher to Subscriber. But in service messages, there are two sections of messages as seen above.

- We can execute any service from the terminal using the following command:

***rosservice call /service\_name args***



```
sanket@sanket-omen: ~
sanket@sanket-omen: ~
/roscout/get_loggers
/roscout/set_logger_level
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
sanket@sanket-omen:~$ rosservice info /spawn
Node: /turtlesim
URI: rosrpc://sanket-omen:58387
Type: turtlesim/Spawn
Args: x y theta name
sanket@sanket-omen:~$ rossrv info turtlesim/Spawn
float32 x
float32 y
float32 theta
string name
---
string name

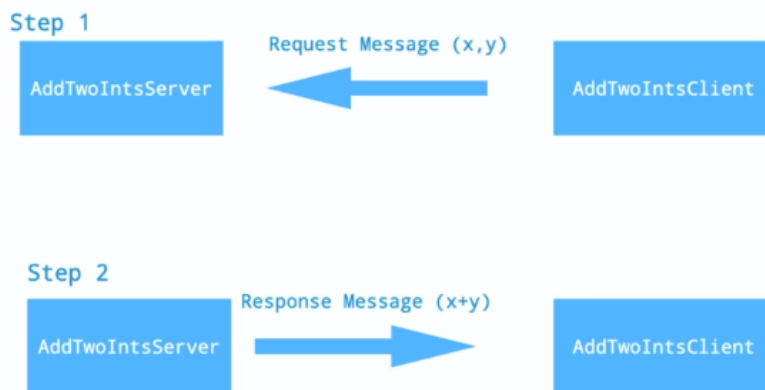
sanket@sanket-omen:~$ rosservice call /spawn 7 7 0 turtle2
name: "turtle2"
sanket@sanket-omen:~$
```

- As seen here in the above rosservice info of /spawn, there are 3 arguments i.e x y theta and name. These arguments will transfer from client to server to perform an action.

# Creating custom Service

- Steps in creating Client/Server services:
  1. Define the service message (service file). One for a request and another for a response.
  2. Create ROS Server node
  3. Create ROS Client node
  4. Execute Service
  5. Consume service by the client

## ROS Service: Adding Two Ints

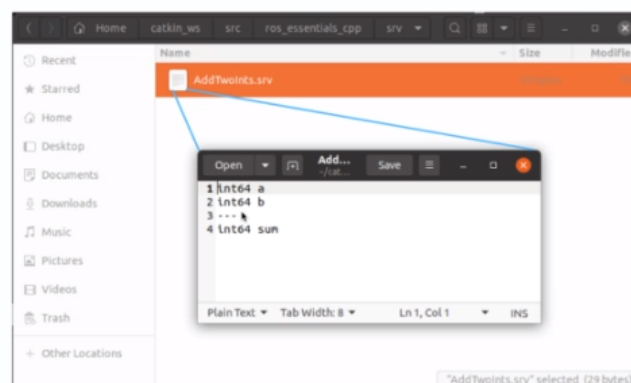


- 1. Define the Service message:

Similar to message creation, for a custom service file first **create an srv folder** inside the package location.

**Name the file with extension .srv** and save it. It should contain two message structures. One for a request and another for a response.

## Define the Service Message





Now need to update dependencies.

Package.xml:

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

If a custom message is already built, then these dependencies will be already included in the file.

CMakeLists.txt:

Add **message\_generation** in **find\_package()**.

```
# Do not just add this line to your CMakeLists.txt, modify the existing line
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
```

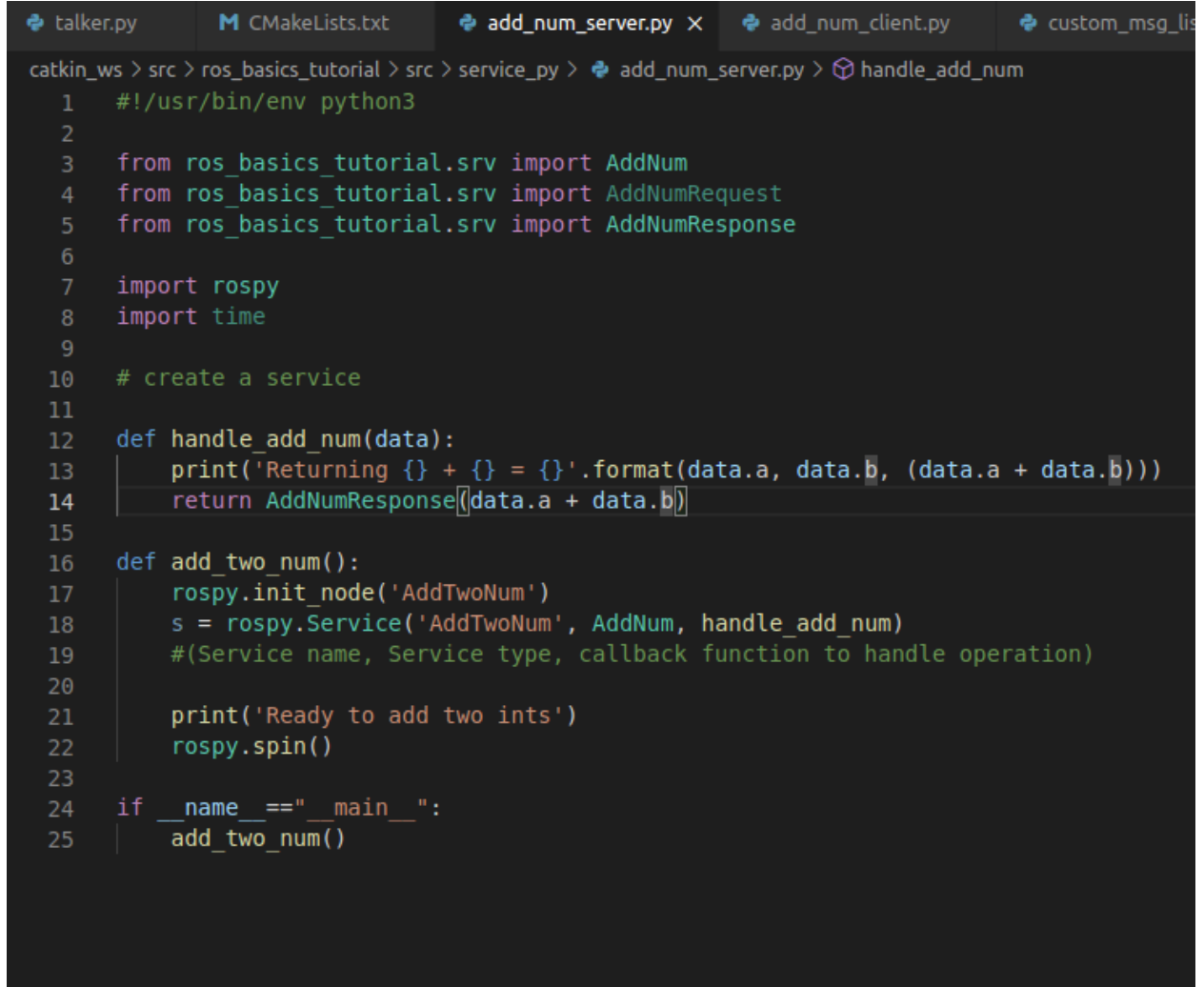
Uncomment/add **add\_service\_files()** section in CMakeLists. Also add custom service name.

```
add_service_files(
  FILES
  AddTwoInts.srv
)
```

- Steps to follow:
  1. Create a **ServiceName.srv** file into **package\_name/srv** folder.
  2. Update **find\_package()** script into package **CMakeLists.txt** file.
  3. Update **add\_service\_files()** script into package **CMakeLists.txt** file.
  4. Add **<build\_depend>** and **<exec\_depend>** into package **package.xml** file.
  5. Run **catkin\_make**
  6. Check srv into ROS using: **rosservice find package\_name/ServiceName.srv**

We can check all service and message files in the **catkin\_ws/devel/include/package\_name** folder. Here all service and message header files will be generated.

## 2. Creating a ROS Server node:



```
catkin_ws > src > ros_basics_tutorial > src > service_py > add_num_server.py > handle_add_num
1  #!/usr/bin/env python3
2
3  from ros_basics_tutorial.srv import AddNum
4  from ros_basics_tutorial.srv import AddNumRequest
5  from ros_basics_tutorial.srv import AddNumResponse
6
7  import rospy
8  import time
9
10 # create a service
11
12 def handle_add_num(data):
13     print('Returning {} + {} = {}'.format(data.a, data.b, (data.a + data.b)))
14     return AddNumResponse(data.a + data.b)
15
16 def add_two_num():
17     rospy.init_node('AddTwoNum')
18     s = rospy.Service('AddTwoNum', AddNum, handle_add_num)
19     #(Service name, Service type, callback function to handle operation)
20
21     print('Ready to add two ints')
22     rospy.spin()
23
24 if __name__ == "__main__":
25     add_two_num()
```

Import service library file i.e. service, service\_request, service\_response

Initiate the server node.

Create a server service object with the service name, service type, and handling function.

In function return the response with the required value i.e. sum.

Run continuously until the manual stop.

## 3. Creating ROS client node:

```
catkin_ws > src > ros_basics_tutorial > src > service_py > add_num_client.py > add_two_nums > y
1  #!/usr/bin/env python3
2
3  from ros_basics_tutorial.srv import AddNum
4  from ros_basics_tutorial.srv import AddNumRequest
5  from ros_basics_tutorial.srv import AddNumResponse
6
7  import rospy
8
9  import sys
10
11 def add_two_nums(x, y):
12     rospy.wait_for_service('AddTwoNum')
13     # it will wait client untill service is alive
14
15     try:
16         AddTwoNum = rospy.ServiceProxy('AddTwoNum', AddNum)
17         # ServiceProxy is an alternative name for client
18         # AddTwoNum is the service name which should match with name on Server side
19         # AddNum is the server type
20
21         resp = AddTwoNum(x,y)
22         return resp.sum
23     except rospy.ServiceException as e:
24         print('Service Call Failed %s'%e)
25
26
27 if __name__ == "__main__":
28     if len(sys.argv) == 3:
29         x = int(sys.argv[1])
30         y = int(sys.argv[2])
31     else:
32         print("%s [x y]"%sys.argv[0])
33         sys.exit(1)
34     print('Requesting {}+{}'.format(x, y))
35     s = add_two_nums(x, y)
36     print("{} + {} = {}".format(x, y, s))
```

Import service header files i.e. service, service\_request, service\_response

Wait till the service server starts.

Create a service client object i.e. service proxy with the same service name and service type.

Request server to execute an action with required arguments.

It will give back a response from the server.

4.



# ROS Launch

- To launch multiple ROS nodes in a single command.
- **roslaunch** will automatically start a **roscore** if it is not already started.
- This roslaunch file needs to be saved in a separate launch folder inside the package.

*package\_name/launch/file\_name.xml*

*ros\_basic\_tutorial/launch/talker\_listner.xml*

- It is an **XML** file with **.xml** extension.  
*roslaunch package\_name launch\_file\_name.launch*
- ROS Launch XML file contains a list of nodes to launch, parameters to be set with nodes, and attributes.
- Simple minimal launch file

```
<launch>
  <node name="talker" pkg="rospy_tutorials" type="talker" />
</launch>
```

- Tags to use in ROS launch file:
  1. **<launch>**: Root element of launch file. It is a container for other elements. All other tags are included inside this tag.
  2. **<node>**: It launches a ROS Node. It can start and stop any node. ROSLaunch does not guarantee the order in which nodes can be started. There is no feedback to know node is launched or not.

**<node name="listener1" pkg="rospy\_tut" type="listner.py" args="--test" respawn="true" />**

name = "node\_name":

Node name. It should be unique for each node.

pkg = "my\_package\_name":

Name of the package where a particular node is situated.

type = "node\_type":

Name of the executable in the package.

args = "arg1 arg2 arg3":

Arguments need to pass for a particular executable.

respawn = "true" : (default: false)

Restart the node if it quits.

machine = "machine\_name":

Launch node on a particular machine. Default is the local machine.

respawn\_delay = "30"

required = "true"

It has following tags can be included inside.

<env> <remap> <rosparam> <param>

3. **<machine>**: It declare the machine in which ros node can be run. No need if running nodes in locally.

Attributes:

name = "machine\_name":

Unique name for each machine. It should match with the machine attribute of <node>

address = "ip\_address":

The network address of the machine.

env-loader= "/opt/ros/fuerte/env.sh"

default= "true/false/never"

user= "ssh\_username"

password= "ssh\_password"

timeout= "10.0"

Following tags can be used: <env>

4. **<include>**: It enables to import another roslaunch XML file into current file.

Attribute:

file= "/package\_name/launch/filename.xml"

ns= "foo"

clear\_params= "true/false"

pass\_all\_args= "true/false"

Following tags can be used inside: <env> <args>

5. **<remap>**: It allows to trick ROS node so that when it thinks it is subscribing/publishing to /some\_topic it is actually subscribing/publishing to /some\_other\_topic.

Sometimes a message on a specific ROS Topic which goes to one set of nodes also needs to be received by another node. Remapping node ends up subscribing to /needed\_topic when node thinks it is subscribing to /different\_topic.

Remapping applies to the lines following the remap. Nodes that are launched before any remap lines are not affected.

Remapping affects both which topics a node subscribes to or publishes to. However, usually remapping is done on the subscribing node.

```
<remap from="/different_topic" to="/needed_topic"/>
```

Attributes:

from= "original\_topic\_name"

to= "new\_topic\_name"

6. **<env>**: It allows to set environment variables on nodes that are launched. It can be only used within scope of <launch> <include> <node> <machine>.

Attributes:

name= "environment\_variable\_name"

value= "environment\_variable\_value"

7. **<param>**: It defines a parameter to be set on Parameter server.

Attributes:

name= "namespace/name"

value= "value"

type= "str/int/double/bool/yaml"

textfile= "package\_name/path/file.txt"

binfile= "package\_name/path/file"

command= "package\_name/exe"

```
<param name="publish_frequency" type="double" value="10.0" />
```

8. **<rosparam>**: It enables the use of rosparam YAML files for loading and dumping parameters from the ROS Parameter Server. It can also be used to remove parameters.

Attribute:

command= "load/dump/delete"

file= "package\_name/path/foo.yaml"

param= "param-name"

ns= "namespace"

subst\_value= "true/false"

```
<rosparam command="load" file="$(find rosparam)/example.yaml" />
```

```
<rosparam command="delete" param="my/param" />
```

9. **<group>**: It apply settings to a group of nodes.

Attributes:

ns= "namespace"

clear\_params= "true/false"

It can include the following tag elements:

<node> <param> <remap> <rosparam> <machine> <include> <env> <test>  
<arg>

10. **<test>**: It is syntactically similar to the <node> tag. They both specify a ROS node to run, but the <test> tag indicates that the node is actually a test node to run.

It has same attributes as <node>

***<test test-name="test\_1\_2" pkg="mypkg" type="test\_1\_2.py" time-limit="10.0" args="--test1 --test2" />***

11. **<arg>**: It allows to create more re-usable and configurable launch files by specifying values that are passed via the command-line, passing in via an <include>, or declared for higher-level files. Args are not global. An arg declaration is specific to a single launch file, much like a local parameter in a method.

Attributes:

name= "arg\_name"

default= "default\_value"

value= "value" (cannot be combined with default)

doc= "description for this arg"

***<include file="included.launch">***

***<!-- all vars that included.launch requires must be set -->***

***<arg name="hoge" value="fuga" />***

***</include>***

***roslaunch my\_file.launch hoge:=my\_value***



```

<launch>
  <!-- local machine already has a definition by default.
       This tag overrides the default definition with
       specific ROS_ROOT and ROS_PACKAGE_PATH values -->
  <machine name="local_alt" address="localhost" default="true" ros-root="/u/user/ros/ros/" ros-p
ackage-path="/u/user/ros/ros-pkg" />
  <!-- a basic listener node -->
  <node name="listener-1" pkg="rospy_tutorials" type="listener" />
  <!-- pass args to the listener node -->
  <node name="listener-2" pkg="rospy_tutorials" type="listener" args="-foo arg2" />
  <!-- a respawn-able listener node -->
  <node name="listener-3" pkg="rospy_tutorials" type="listener" respawn="true" />
  <!-- start listener node in the 'wg1' namespace -->
  <node ns="wg1" name="listener-wg1" pkg="rospy_tutorials" type="listener" respawn="true" />
  <!-- start a group of nodes in the 'wg2' namespace -->
  <group ns="wg2">
    <!-- remap applies to all future statements in this scope. -->
    <remap from="chatter" to="hello"/>
    <node pkg="rospy_tutorials" type="listener" name="listener" args="--test" respawn="true" />
    <node pkg="rospy_tutorials" type="talker" name="talker">
      <!-- set a private parameter for the node -->
      <param name="talker_1_param" value="a value" />
      <!-- nodes can have their own remap args -->
      <remap from="chatter" to="hello-1"/>
      <!-- you can set environment variables for a node -->
      <env name="ENV_EXAMPLE" value="some value" />
    </node>
  </group>
</launch>

```

# Gazebo

- **Dynamic Simulations:** Create robot dynamics with a physics engine.  
**Advanced 3D graphics:** Rendering, lighting, shadow, texture, etc.  
**Sensors:** Add sensors to a robot for data generation, and noise simulation.  
**Plugins:** To interact with the world, robot, and sensor.  
**Model Database:** Available default robots, and environments or create our own.  
**Socket-based communication:** Interact with a remote server through socket communication.  
**Cloud simulator:** Using cloud with browser.  
**Command-line tools:** Control using command-line tools.

## Gazebo components:

- **Gazebo Server:** The first component to run. It parses the description files for the simulation and object. It simulates the scene using these files, physics, and sensor engine.  
*gzserver*

- **Gazebo client:** The second component to run. It provides a graphical client to render simulations with tools.

*gzclient*

It is not required to run these two commands separately, as the following command launches both files.

*gazebo*

- **World Files:** It contains the elements for the simulated environment. It contains a robot model, environment, lighting, sensor, and other objects. It has a .world extension.

*gazebo file\_name.world*

The world file is written in an SDF format (Simulation Description Format).

```

<?xml version="1.0" ?>
<sdf version="1.5">
  <world name="default">
    <physics type="ode">
      ...
    </physics>

    <scene>
      ...
    </scene>

    <model name="box">
      ...
    </model>

    <model name="sphere">
      ...
    </model>

    <light name="spotlight">
      ...
    </light>

  </world>
</sdf>

```

- **Model File:** Along with a world file, a model file is also required which represents a single robot model. It can be imported into a world file. This model file is in either SDF or URDF format.
- **Environment variables:** Environment variables are used to locate files, set of communication between server and client, etc. Mostly these variables should remain default.
- **Plugins:** It is used to interact with a world, model, and sensor in a gazebo. These plugin are written in a cmd line or added into an SDF world file.

# URDF Model

- Create a package with the name *robot\_description* for any robot model want to use with *urdf* dependencies.

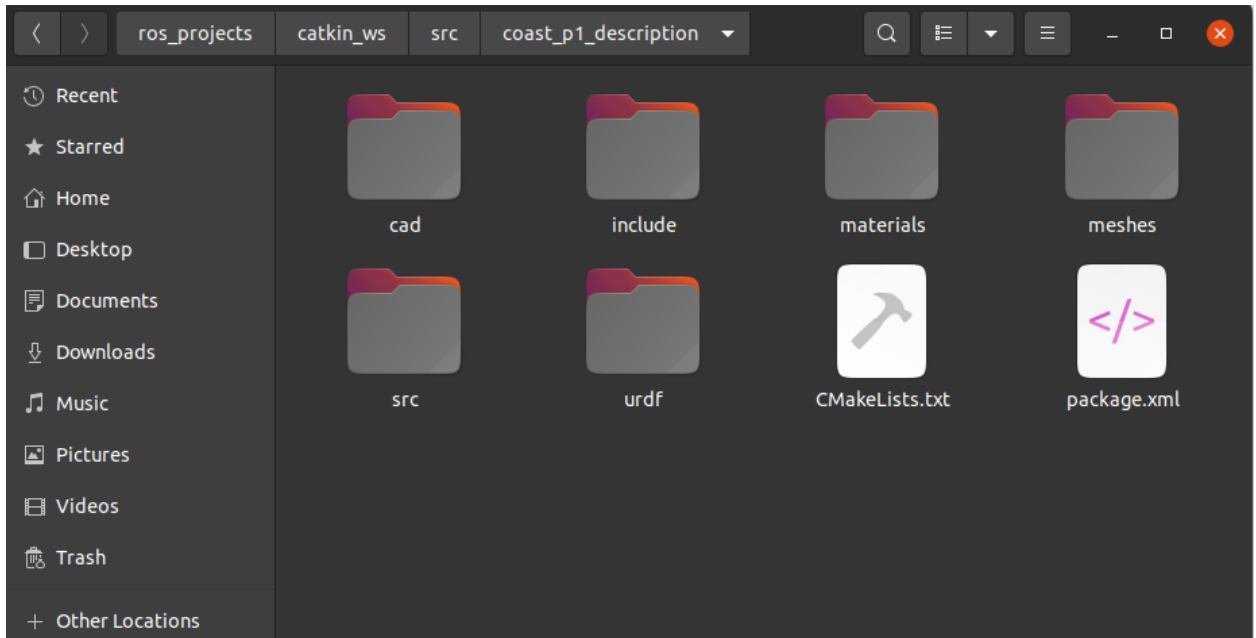
```
sanket@sanket-omen:~/Projects/ros_projects/catkin_ws/src$ catkin_create_pkg coast_p1
_description urdf roscpp rospy std_msgs
Created file coast_p1_description/package.xml
Created file coast_p1_description/CMakeLists.txt
Created folder coast_p1_description/include/coast_p1_description
Created folder coast_p1_description/src
Successfully created files in /home/sanket/Projects/ros_projects/catkin_ws/src/coast
_p1_description. Please adjust the values in package.xml.
sanket@sanket-omen:~/Projects/ros_projects/catkin_ws/src$
```

- Run the *catkin\_make* command at the *catkin\_ws* location.

```
sanket@sanket-omen:~/Projects/ros_projects/catkin_ws/src$ catkin_create_pkg coast_p1
_description urdf roscpp rospy std_msgs
Created file coast_p1_description/package.xml
Created file coast_p1_description/CMakeLists.txt
Created folder coast_p1_description/include/coast_p1_description
Created folder coast_p1_description/src
Successfully created files in /home/sanket/Projects/ros_projects/catkin_ws/src/coast
_p1_description. Please adjust the values in package.xml.
sanket@sanket-omen:~/Projects/ros_projects/catkin_ws/src$ cd ..
sanket@sanket-omen:~/Projects/ros_projects/catkin_ws$ catkin_make
Base path: /home/sanket/Projects/ros_projects/catkin_ws
Source space: /home/sanket/Projects/ros_projects/catkin_ws/src
Build space: /home/sanket/Projects/ros_projects/catkin_ws/build
Devel space: /home/sanket/Projects/ros_projects/catkin_ws/devel
Install space: /home/sanket/Projects/ros_projects/catkin_ws/install
####
#### Running command: "cmake /home/sanket/Projects/ros_projects/catkin_ws/src -DCATK
IN_DEVEL_PREFIX=/home/sanket/Projects/ros_projects/catkin_ws/devel -DCMAKE_INSTALL_P
REFIX=/home/sanket/Projects/ros_projects/catkin_ws/install -G Unix Makefiles" in "/h
ome/sanket/Projects/ros_projects/catkin_ws/build"
####
```

- Create other subfolders i.e. /urdf /meshes /materials /cad etc

```
/MYROBOT_description
package.xml
CMakeLists.txt
/urdf
/meshes
/materials
/cad
```



## Simple One Link Robot:

- Write a simple **robot.urdf** file for a robot in the **/urdf** folder. For easy implementation use the ros extension in visual code.

```
catkin_ws > src > coast_p1_description > urdf > p1.urdf
1  <?xml version="1.0"?>
2  <<robot name="p1">
3    <link name="base_link">
4      <visual>
5        <geometry>
6          <cylinder radius="0.6" length="0.2"/>
7        </geometry>
8      </visual>
9    </link>
10
11 </robot>
```

- Create a **launch** file that will launch **rviz** with a robot model saved in URDF. Create a new **/launch** folder that will store **robot\_display.launch** file.

```
talker.py p1.urdf p1_display.launch X
catkin_ws > src > coast_p1_description > launch > p1_display.launch
1 <launch>
2   <arg name="model" default="$(find coast_p1_description)/urdf/p1.urdf"/>
3   <param name="robot_description" textfile="$(find coast_p1_description)/urdf/p1.urdf" />
4   <param name="use_gui" value="true"/>
5
6   <node name="joint_state_publisher" pkg="joint_state_publisher_gui" type="joint_state_publisher_gui" />
7   <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" />
8   <node name="rviz" pkg="rviz" type="rviz" args="-d $(find coast_p1_description)/rviz/urdf.rviz" required="true" />
9 </launch>
```

- Launch file description:
  - It loads the model into the parameter server.
  - Run the node to publish Joint\_state.
  - Start rviz with a configuration file.
- The rviz configuration file is located at */rviz/urdf.rviz*.
- Run *catkin\_make* to update the package.
- To run a robot model in rviz run the launch file.  
*roslaunch package\_name robot\_display.launch*  
*roslaunch coast\_p1\_description p1\_display.launch*

## Simple Robot model with STL mesh file:

- To create a robot model with a mesh STL file, the above procedure is the same. Instead of adding predefined shapes like box, the cylinder adds a **mesh file with <mesh>** and filename.
- Store the STL file in the */mesh* folder.

```
talker.py p1.urdf p1_display.launch
catkin_ws > src > coast_p1_description > urdf > p1.urdf
1 <?xml version="1.0"?>
2
3 <robot name="p1_first_trial">
4
5   <!--1st link-->
6   <link name="base_link">
7     <visual>
8       <origin xyz="0 0 0" rpy="0 0 0"/>
9       <geometry>
10        <!-- <cylinder radius="0.2" length="0.6"/> -->
11        <mesh filename="package://coast_p1_description/meshes/P1_gazebo.stl" />
12      </geometry>
13    </visual>
14  </link>
15 </robot>
```

- Run the model using the roslaunch command.

*roslaunch coast\_p1\_description p1\_display.launch*

## Robot model with multiple links and joints:

- With multiple link joints `<joint>` must be added in the urdf file.
- The overall structure remains the same as above.

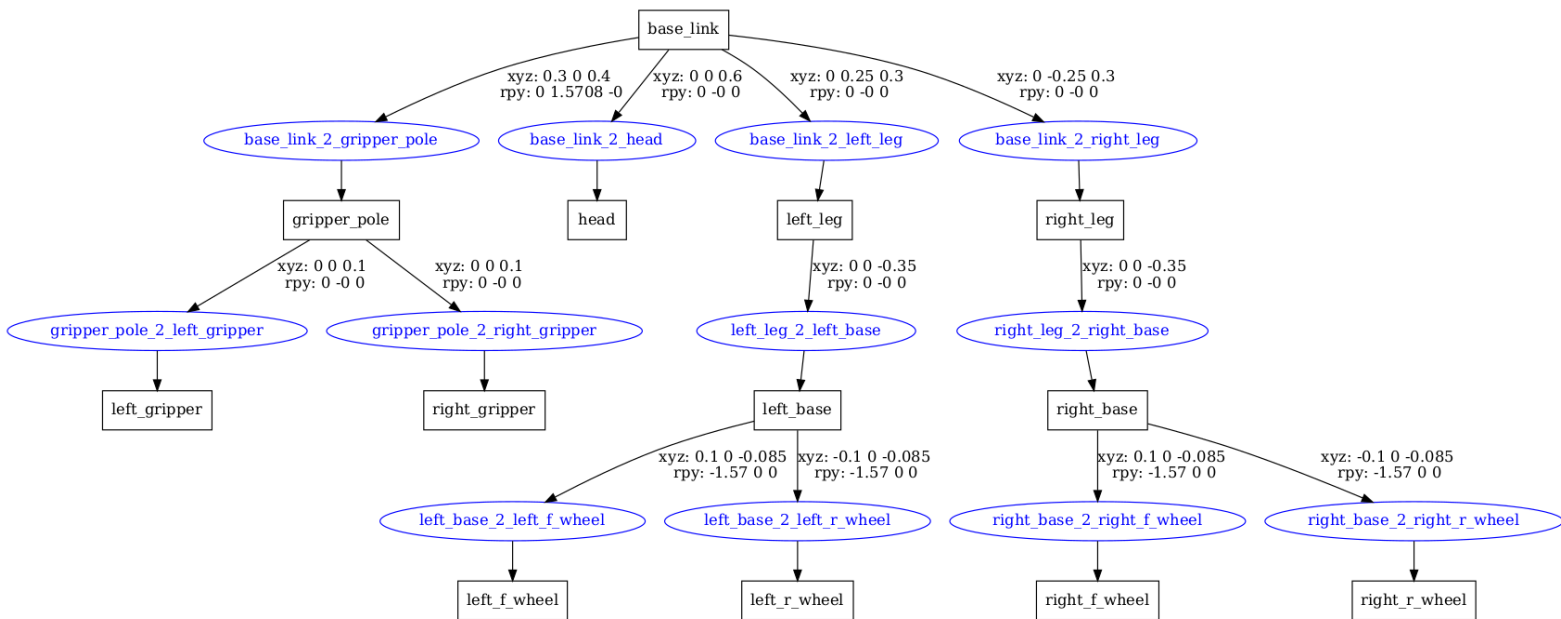
```

3  <robot name="p1_first_trial">
4
5      <!-- Material color definition -->
6      <material name="blue">
7          <color rgba="0.0 0.0 0.8 1.0"/>
8      </material>
9
10     <material name="white">
11         <color rgba="1.0 1.0 1.0 1.0"/>
12     </material>
13
14
15     <!-- 1st link -->
16     <link name="base_link">
17         <visual>
18             <origin xyz="0 0 0.3" rpy="0 0 0"/>
19             <geometry>
20                 <cylinder radius="0.2" length="0.6"/>
21                 <!-- <mesh filename="package://coast_p1_description/meshes/P1_gazebo.stl" /> -->
22             </geometry>
23             <material name="blue" />
24         </visual>
25     </link>
26
27     <!-- 2nd Link -->
28     <link name="right_leg">
29         <visual>
30             <origin xyz="0 0 0" rpy="0 0 0"/>
31             <geometry>
32                 <box size="0.2 0.1 0.6"/>
33             </geometry>
34             <material name="white"/>
35         </visual>
36     </link>
37
38     <!-- Joint bet 1st and 2nd link -->
39     <joint name="base_link_2_right_leg" type="fixed">
40         <parent link="base_link"/>
41         <child link="right_leg"/>
42         <origin xyz="0 -0.25 0.3"/>
43     </joint>
44
45 </robot>

```

- Run the launch file.

*roslaunch coast\_p1\_description p1\_display.launch*



## Robot with motion:

- Until now we built an urdf model of the robot with multiple links and joints, where we used only fixed joints.
- By adding different joint types we can give motion to each link.
- These joint types are explained below.

## Robot with physical, collision properties:

- Similar to <visual> we can add collision and physical properties like mass, inertia, and dynamics into the robot model. These properties help during simulation in Gazebo.
- **Collision:** Similar to <visual>, <collision> also has origin, geometry tags. To detect collisions in Gazebo with other models this property needs to be assigned. Computing collisions is heavy, thus unless and until it is required do not use it. Also, try to use simple geometries.
- **Physical properties:** To simulate a model in the gazebo, several physics engine properties need to be defined. Inertia and mass (kg) need to be defined. Also, the origin will define the center of gravity wrt links reference frame. We can also define contact coefficients i.e. friction coefficient, stiffness coefficient, and damping coefficient.



Toggle line numbers

```
1 <link name="base_link">
2   <visual>
3     <geometry>
4       <cylinder length="0.6" radius="0.2"/>
5     </geometry>
6     <material name="blue">
7       <color rgba="0 0 .8 1"/>
8     </material>
9   </visual>
10  <collision>
11    <geometry>
12      <cylinder length="0.6" radius="0.2"/>
13    </geometry>
14  </collision>
15  <inertial>
16    <mass value="10"/>
17    <inertia ixx="0.4" ixy="0.0" ixz="0.0" iyy="0.4" iyz="0.0" izz="0.2"/>
18  </inertial>
19 </link>
```

## Xacro:

- Xacro is a micro language. Using this xacro we can generate urdf in the launch file automatically.
- Xacros allows specifying properties as a constant.

```
1 <xacro:property name="width" value="0.2" />
2 <xacro:property name="bodylen" value="0.6" />
3 <link name="base_link">
4   <visual>
5     <geometry>
6       <cylinder radius="${width}" length="${bodylen}" />
7     </geometry>
8     <material name="blue"/>
9   </visual>
10  <collision>
11    <geometry>
12      <cylinder radius="${width}" length="${bodylen}" />
13    </geometry>
14  </collision>
15 </link>
```

## URDF with Gazebo:

- 
- **XML tags in URDF:**
  1. <robot>: Represent a robot description.
  2. <link>
  3. <joint>
  4. <sensor>
  5. <transmission>
  6. <gazebo>
  7. <model\_state>
  8. <model>

### 1. <robot name= " ">:

Root element of urdf. All other elements are encapsulated within it.

### 2. <link name= " ">:

It describes a rigid body with its inertia, visual, and collision properties.

Elements inside <link>:

#### I. <inertial>: Link's mass, the position of center of mass, and central inertia properties.

<origin xyz= " " rpy= " " />: Describe the position and orientation of the **link's center of mass** wrt reference **frame of link**.

<mass value= " " />: Mass of the link.

<inertia ixx= " " ixy= " " ixz= " " iyy= " " iyz= " " izz= " " />: Links moment of inertia.

#### II. <visual>: Visual properties of link like the shape of the object. The union of multiple <visual> define forms of visual representation.

<origin xyz= " " rpy= " " />: Ref frame of visual wrt reference **frame of link**.

<geometry>: Shape of a visual object.

<box size= " " />: Cube with size x y z.

<cylinder radius= " " length= " " />

<sphere radius= " " />

<mesh filename= " " scale= " " />: With any geometry file. Best format .dae

<material>: Material with color and texture file.

<color rgba= " " />: Color in red blue green and alpha.

<texture />

#### III. <collision>: Collision properties of the link. The union of multiple <collision> defines forms of visual representation.



***<origin xyz= “ ” rpy= “ ” />***  
***<geometry>***

```

Toggle line numbers
1  <link name="my_link">
2    <inertial>
3      <origin xyz="0 0 0.5" rpy="0 0 0"/>
4      <mass value="1"/>
5      <inertia ixx="100" ixy="0" ixz="0" iyy="100" iyz="0" izz="100" />
6    </inertial>
7
8    <visual>
9      <origin xyz="0 0 0" rpy="0 0 0" />
10     <geometry>
11       <box size="1 1 1" />
12     </geometry>
13     <material name="Cyan">
14       <color rgba="0 1.0 1.0 1.0"/>
15     </material>
16   </visual>
17
18   <collision>
19     <origin xyz="0 0 0" rpy="0 0 0"/>
20     <geometry>
21       <cylinder radius="1" length="0.5"/>
22     </geometry>
23   </collision>
24 </link>

```

#### 4. ***<joint name= “ ” joint= “ ”>:***

It describes the kinematics and dynamics of the joint between two links. It has two attributes.

***name:*** Unique name for Joint.

- type:***
1. ***revolute:*** Rotating joint with the upper and lower limit.
  2. ***continuous:*** Rotation joint without any limit.
  3. ***prismatic:*** Sliding joint along axis. Range limited by the upper and lower limit
  4. ***fixed:*** Joint can not move
  5. ***floating:*** Motion in all 6DOF
  6. ***planar:*** Motion in a plane perpendicular to the axis.

Elements of <joint>:

***I. <origin xyz= “ ” rpy= “ ” />:*** Transform from **parent link** to **child link**. The Joint is

located at the origin of the child link. In meter and radian.

**II.** `<parent link= “name” />`: Parent link name. Compulsory.

**III.** `<child link= “name” />`: Child link name. Compulsory.

**IV.** `<axis xyz= “ ” />`: Joint axis in joint frame. Axis of rotation for revolute joint, and axis of translation for a prismatic joint. Default is 1 0 0.

**V.** `<calibration rising= “ ” falling= “ ” />`: To calibrate the absolute position of the joint.

**Rising:** when a joint moves in a **+ve direction**, the reference position will trigger a **rising edge**.

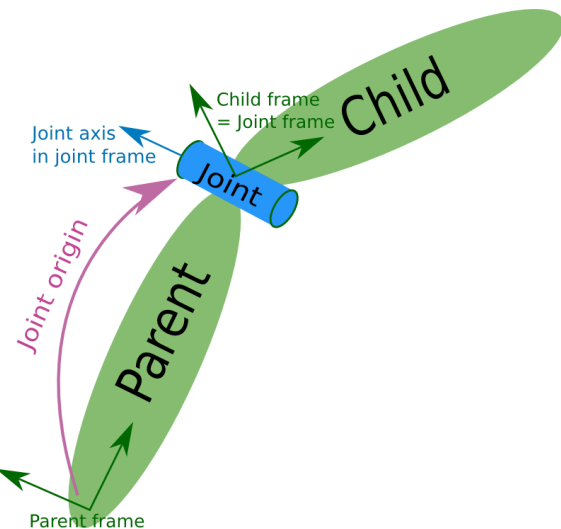
**Falling:** when the joint moves in a **+ve direction**, the reference position will trigger a **falling edge**.

**VI.** `<dynamics damping= “ ” friction= “ ” />`: Specify the physical properties of the point.

**VII.** `<limit lower= “ ” upper= “ ” effort= “ ” velocity= “ ” />`: To specify limits of sensor as per type. Max velocity, max joint effort, max upper limit, and min lower limit.

**VIII.** `<mimic joint= “name” multiplier= “ ” offset= “ ” />`: To specify the defined joint mimics another existing joint.

**IX.** `<safety_controller soft_lower_limit=“ ” soft_upper_limit=“ ” k_position=“ ” k_velocity=“ ” />`



Toggle line numbers

```
1 <joint name="my_joint" type="floating">
2   <origin xyz="0 0 1" rpy="0 0 3.1416"/>
3   <parent link="link1"/>
4   <child link="link2"/>
5
6   <calibration rising="0.0"/>
7   <dynamics damping="0.0" friction="0.0"/>
8   <limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
9   <safety_controller k_velocity="10" k_position="15" soft_lower_limit="-2.0" soft_upper_
limit="0.5" />
10 </joint>
```

5. **<sensor name= “ ” type= “ ” id= “ ” update\_rate= “ ”>**: It used to add sensor in URDF model.

**name:** unique name of the sensor.

**type:** Type of sensor like camera/ray/imu/etc.

**id:** Id of a sensor to describe the order of sensor

**update\_rate:** Frequency at which sensor data is generated.

Elements inside <sensor>:

- I. **<parent link= “name” />** : Name of the link to which sensor is connected.
- II. **<origin xyz= “ ” rpy= “ ” />**: Pose and orientation of sensor frame wrt parents reference frame. The sensor's optical frame usually has z-forward, x-right, and y-down.

Now different types of sensors which can be embedded inside <sensor>

III. **<camera>**

**<image width= “ ” height= “ ” format= “ ” hfov= “ ” near= “ ” far= “ ” />**

**<ray>**

**<horizontal samples= “ ” resolution= “ ” min\_angle= “ ” max\_angle= “ ” />**

**<vertical samples= “ ” resolution= “ ” min\_angle= “ ” max\_angle= “ ” />**

**<imu>**

**<gyro> <noise>**

**<acceleration> <noise>**

**<magnetometer>**

**<gps>**

*<position\_sensing> <noise>*  
*<velocity\_sensing> <noise>*  
*<force\_torque>*  
*<contact>*  
*<sonar>*  
*<rfidtag>*  
*<rfid>*

6.

# Moveit

-

# Commands

- **roscore** : Start a ROS
- **catkin\_make** : Build a catkin environment.
- **roscd** : Change directory directly to ROS project defined in bash file.
- **catkin\_create\_pkg** *my\_package\_name dependency1 dependency2 ....* : To create catkin package inside src folder of workspace.
- **rospack depends1** *my\_package\_name* : It gives information about the dependencies of packages.
  
- **rostopic list** : It gives a list of all online nodes in ROS.
- **rostopic info** */node\_name* : It gives information about that particular node like a list of the publisher, subscribers, services, network, etc.
- **rostopic kill** */node\_name* : To shut down a specific node.
  
- **rostopic list** : It gives a list of all online topics in ROS.
- **rostopic info** */topic\_name* : It gives all information about a particular topic name.
- **rostopic pub** */topic\_name msg\_type "Your Message"* : To publish a message to a particular topic from CMD.
- **rostopic echo** */topic\_name* : To see what message is going to publish on a given topic.
  
- **rosmmsg list**: It gives a list of all ROS Msg available in the environment.
- **rosmmsg show** *std\_msgs/String* : It gives the content of each specific msg.
  
- **rosservice list**: It gives all available services which are currently available.
- **rosservice info** */service\_name*: It will give you the information about the service.
- **rossrv info** *message\_type*: This will give you information about the service message.
- **rosservice call** */service\_name args*: To send service request by giving an argument in it. This info can get from the rossrv info command.
- 
  
- **roslaunch** *my\_package\_name node\_name* : Run a particular node from the created package.
- **chmod +x** *listener.py* : To make python files executables.



- *roslaunch rqt\_graph rqt\_graph*                      **OR**  
*rqt\_graph*                      **OR**  
*rqt* : To visualize all computational graphs. (If not worked  
uninstall pip3 PyQt5 and PyQt5-sip)
-