Introduction to ROS1: Getting Started with Robot Operating System

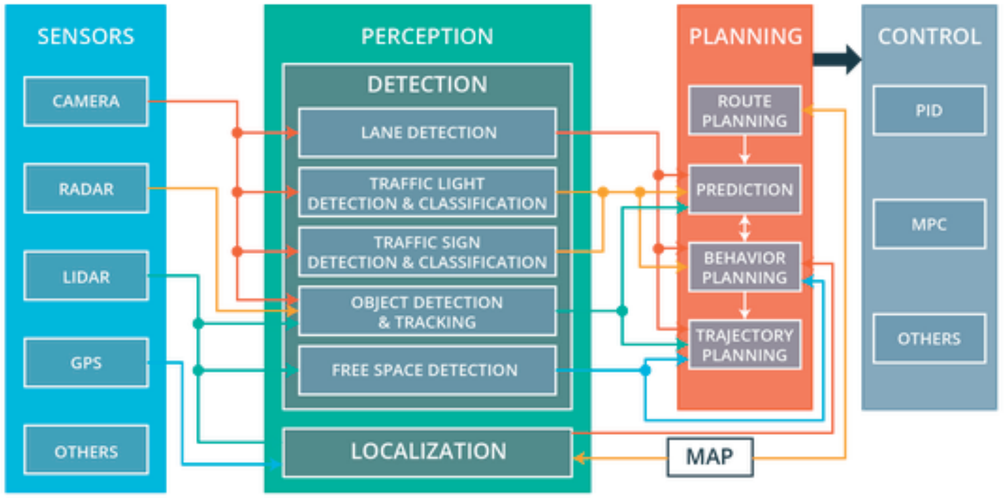
Jegathesan S

What is ROS1?

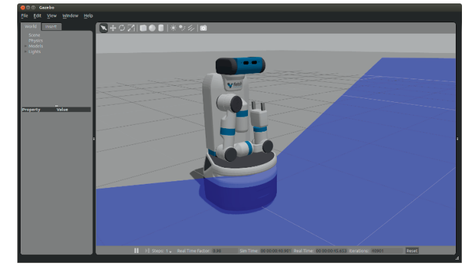
ROS1 (Robot Operating System) is a popular **open-source middleware platform** used for building and controlling robots.

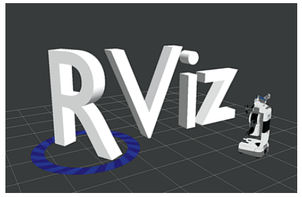
ROS1 provides a set of **software libraries** and **tools** to help developers create robot applications. It includes a wide range of features, such as **messaging**, **hardware abstraction**, **visualization**, and **navigation**.

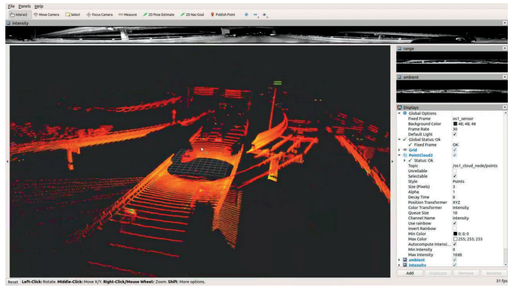
ROS1 is designed to be **modular** and **distributed**, allowing developers to create complex robot systems by combining different components.



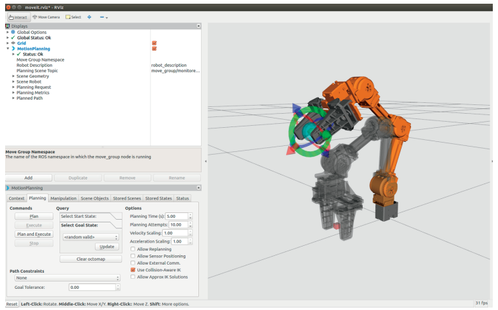
Self-driving Software Architecture





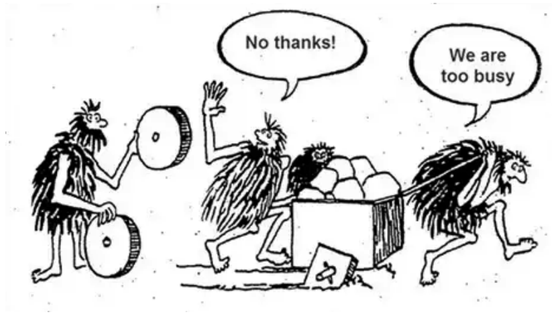


MoveIt



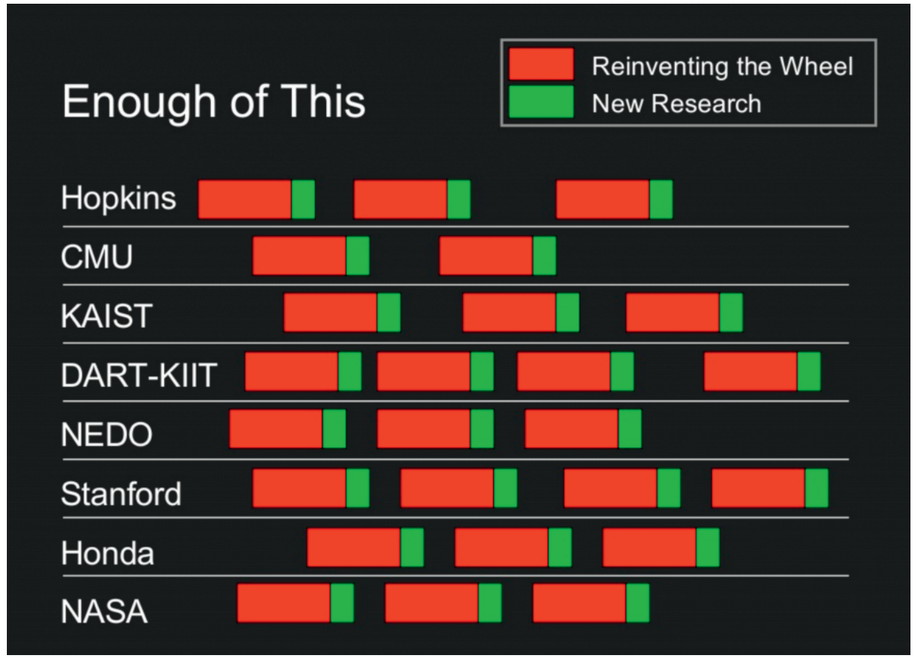
A Decade of Robotics Innovation: A History of ROS

> »Tesla has created a software platform with more lines of code than Windows, but with a fraction of Microsoft's software-development capabilities. How did Tesla achieve this without an army of engineers and three decades of experience? Through the extensive use of available and mature open-source software. Companies are now assembling capabilities through available libraries rather than writing code from scratch.«
>
> from "An executive's guide to software development" – McKinsey&Company, February 2017



ROS1 was first introduced in 2007 by researchers at **Stanford University** and is now maintained by the **Open Robotics** organization.

This personal project started to solve a common problem in robotics: the need to constantly **reinvent the software infrastructure** required to build complex robotic algorithms. They noticed that too much time was spent on building drivers for sensors and actuators, as well as on creating communication systems between different programmes inside the same robot. As a result, **too little time was left for actually building intelligent robotics programs.**



Most of the time spent in robotics was reinventing the wheel (slide from Eric and Keenan pitch deck)

Why ROS based software system?

Modularity:

ROS provides a modular architecture that allows developers to easily create, test, and reuse individual software components. This can help to improve the efficiency and maintainability of the development process.

Interoperability:

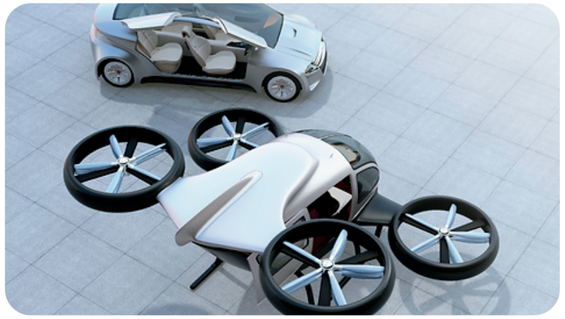
ROS provides a set of standard interfaces and message types that allow different software components to communicate and interact with each other, regardless of the specific hardware or operating system they are running on. This can help to ensure that the final solution is compatible with a wide range of devices and platforms.

Large Ecosystem:

ROS provides a modular architecture that allows developers to easily create, test, and reuse individual software components. This can help to improve the efficiency and maintainability of the development process.

ROS For Aerospace - AeroDef Manufacturing

NASA Valkyrie - Space ROS



ROS for Self Driving car and Air Taxi

Leading the Way: A Look at Who is Using ROS in Robotics Development

Over 500 companies have integrated ROS into their products

open robotics

ROS 2

ROS 2 TSC

ADLINK Apex.AI
BOSCH TOYOTA RESEARCH INSTITUTE
Microsoft amazon
ubuntu intel SAMSUNG
PICKNIK open robotics
eevcom LG Electronics
ROVER ROBOTICS eProsima
iRobot SONY
ROBOTIS WIND

ROS Industrial

78 Members

Government

- NASA
- NHSTA / USDOT
- DARPA
- Army / Navy / AF
- NIST
- Dozens of Universities
- Singapore Hospital System

BOEING Boeing

Website: http://www.boeing.com/
Type: MNC
ROS Repositories: https://github.com/boeing

Notes: Boeing designs, manufactures, and sells airplanes, rotorcraft, rockets, and satellites worldwide
source: website in the above link.

BOSCH Bosch

Website: https://www.bosch.com/
Type: OEM
ROS Wiki page: http://wiki.ros.org/bosch-ros-pkg
ROS Repositories: https://github.com/bosch-ros-pkg
https://github.com/BoschSensortec
https://github.com/bsinno

Notes: Robert Bosch LLC provides technology and services for mobility solutions, industrial technology, consumer goods, and energy and building technology sectors.
source: website in the above link.

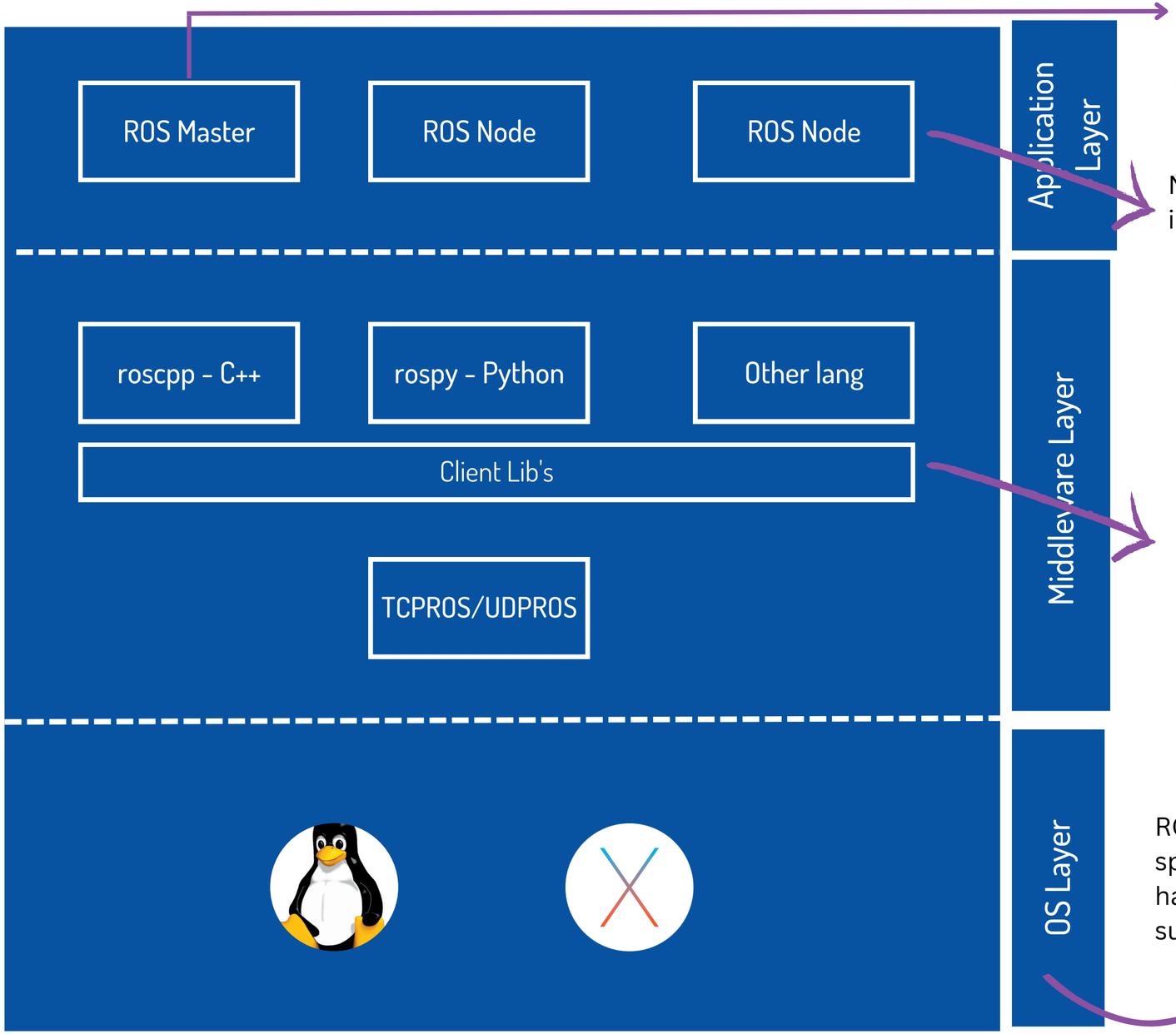
BMW

Website: https://www.bmw.com/en/index.html
Type: OEM

Notes: Manufactuer that currently produces automobiles and motorcycles.
source: website in the above link.

ROS Industrial Members

ROS Architecture



The ROS Master in ROS1 is implemented using XMLRPC, which is a remote procedure call protocol that uses XML to encode its requests and responses. It is a stateless, HTTP-based protocol.

Nodes are the fundamental unit of computation in ROS1. They are independent processes that can perform various tasks,

Roscpp client lib is used for C++, and Rospy client lib is used for Python. Each library is built from scratch and is completely independent. In other words, the API is not necessarily the same for roscpp and rospy, and some features are developed for one but not the other.

ROS1 was originally developed to run on Unix-based platforms, specifically Linux, and it has been optimized for this platform. ROS1 has also been tested and used on other Unix-based operating systems, such as macOS and some versions of Unix-like operating systems.

ROS packages - Overview

ROS1 uses package to organise its programs. ROS1 packages are typically organized within a workspace, which is a directory that contains all of the resources and code needed for a particular project.

ROS1 packages are typically created using the **Catkin** build tool, which is used to manage the dependencies and build process for packages within a workspace. Catkin is a build system that was specifically designed for ROS1, and it is based on the CMake build system.

Catkin uses a workspace concept, where each workspace contains one or more packages. Packages can depend on other packages within the same workspace or on external packages installed on the system. Catkin also provides tools for building and testing packages, such as **catkin_make** and **catkin_tools**.

```
ws                                  # Root Workspace
└── package_structure               # Package Name
    ├── action                      # custom ROS Action definitions
    ├── CHANGELOG.rst               # Compliant Changelog
    ├── CMakeLists.txt              # Compilation and Installation steps for C++ Code
    ├── config                      # Configuration files
    ├── CONTRIBUTING                # Contribution Guidelines
    ├── doc                         # Design or other documentation
    ├── include                     # C++ Header files .hpp
    │   └── package_structure       # Python Module. Keep the same name of package name
    ├── launch                      # Launch files
    ├── LICENSE
    ├── models                      # 3D Models (SDF)
    ├── msg                         # Custom .msg files
    ├── package_structure           # Python modules that can be imported to other ROS packages
    ├── package.xml                 # Define the properties of packages and dep
    ├── README.md                   # Package purpose, installation stpe, and Usage
    ├── rviz                        # RVIZ Visualizer files
    ├── scripts                     # Bash scripts
    ├── setup.py                    # Python Module installation steps
    ├── src                         # C++ Source file .cpp
    ├── srv                         # Custome service file
    ├── test                        # Unit test
    ├── urdf                        # URDF Files
    └── worlds                      # Gazebo world files
```

ROS packages - Create a First Package

```
# Create a Package
mkdir ~/catkin_ws/src/  ## Create a directory
cd ~/catkin_ws/src/     ## Move to src directory
catkin_create_pkg pub_sub rospy std_msgs # Create a package
## rospy is python clinet lib for ROS
## std_msgs contains common message types
## representing primitive data types

# Create a Nodes
cd pub_sub && touch src/publisher.py ## Create a file
chmod a+x src/publisher.py ## Executable permission
touch src/subscriber.py
chmod a+x src/subscriber.py
```

Create a Package:

Use **catkin_create_pkg** to create a new package: This command will create a new package directory with the necessary files and directories for a ROS1 package. You can specify the package dependencies and other details during the creation process.

Create a new node:

Once you have a package, you can create a new Python node file in the src directory of the package. The node should include the necessary ROS1 imports and setup code to create a new node.

ROS packages - Create a First Package

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

import rospy
from std_msgs.msg import String

def publisher():
    # Initialize the node
    rospy.init_node('string_publisher', anonymous=True)

    # Create a publisher object with topic name, message type and queue size
    pub = rospy.Publisher('string_topic', String, queue_size=10)

    # Set the loop rate
    rate = rospy.Rate(10)

    # Publish messages until the node is shut down
    while not rospy.is_shutdown():
        # Create a message
        message = String()
        message.data = "Hello, World!"

        # Publish the message
        pub.publish(message)

        # Sleep to maintain the loop rate
        rate.sleep()

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

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

import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo("Received message: %s", data.data)

def subscriber():
    # Initialize the node
    rospy.init_node('string_subscriber', anonymous=True)

    # Create a subscriber object with topic name, message
    # type and callback function
    rospy.Subscriber('string_topic', String, callback)

    # Spin until the node is shut down
    rospy.spin()

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

ROS packages - Create a First Package

```
# Launch File
mkdir launch && touch launch/pub_sub.launch # Create a empty lauch file

<launch>
    <!-- Start the publisher node -->
    <node pkg="pub_sub" type="publisher.py" name="publisher"  output="screen">
    </node>

    <!-- Start the subscriber node -->
    <node pkg="pub_sub" type="subscriber.py" name="subscriber"  output="screen">
    </node>
</launch>

# Compile
cd ~/catkin_ws/ # go to root directory of the packages
catkin_make # Compile a ROS package
source devel/setup.bash # Source the ROS env variable

# Run
roslaunch pub_sub pub_sub.launch
```

Create a launch file:

A launch file is a configuration file that can be used to launch one or more ROS nodes with a single command. You can create a launch file in the launch directory of your package. The launch file should specify which nodes to run, and any necessary parameters or arguments.

Compile the package:

To compile the package, you need to run catkin_make from the root directory of your catkin workspace. This will build all packages in the workspace, including your new package.

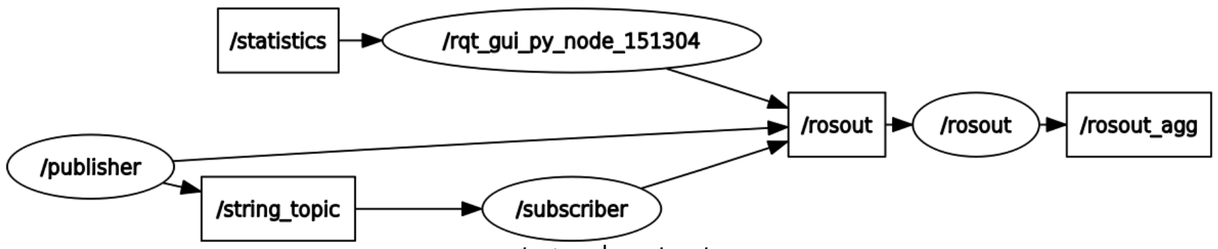
Run the package:

Once the package is compiled, you can run it by executing the launch file using the roslaunch command. This will start the nodes specified in the launch file, and you should see output from the nodes in the console.

ROS packages - Create a First Package

```
*[master][~/Desktop/mygit/ROS1-Workshop/catkin_ws]$ roslaunch pub_sub pub_sub.launch
... logging to /home/nullbyte/.ros/log/ef6f4eb0-bb24-11ed-84ef-f3e7d2a6ff0f/roslaunch-edgeai-150371.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
WARNING: disk usage in log directory [/home/nullbyte/.ros/log] is over 1GB.
It's recommended that you use the 'rosclean' command.

started roslaunch server http://edgeai:40645/

SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.15.15

NODES
  /
    publisher (pub_sub/publisher.py)
    subscriber (pub_sub/subscriber.py)

auto-starting new master
process[master]: started with pid [150417]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to ef6f4eb0-bb24-11ed-84ef-f3e7d2a6ff0f
process[rosout-1]: started with pid [150467]
started core service [/rosout]
process[publisher-2]: started with pid [150470]
process[subscriber-3]: started with pid [150472]
[INFO] [1678000283.839303]: Received message: Hello, World!
```

roslaunch output



rqt_graph output

What is **roslaunch server**?
what is **PARAMETERS**?
what is **NODES**?
What is **ROS_MASTER_URI**?

What is this graph?

ROS1 Core Concepts

Computational Graph

The ROS graph is the most **important part of any ROS system**. The computational graph is a **network of nodes** that work together to **solve problems** and **talk to each other**. Each "**node**" in the computational graph is a process that does **some kind of computation**, and the "**edges**" between the nodes are **the ways that the nodes share data with each other**.

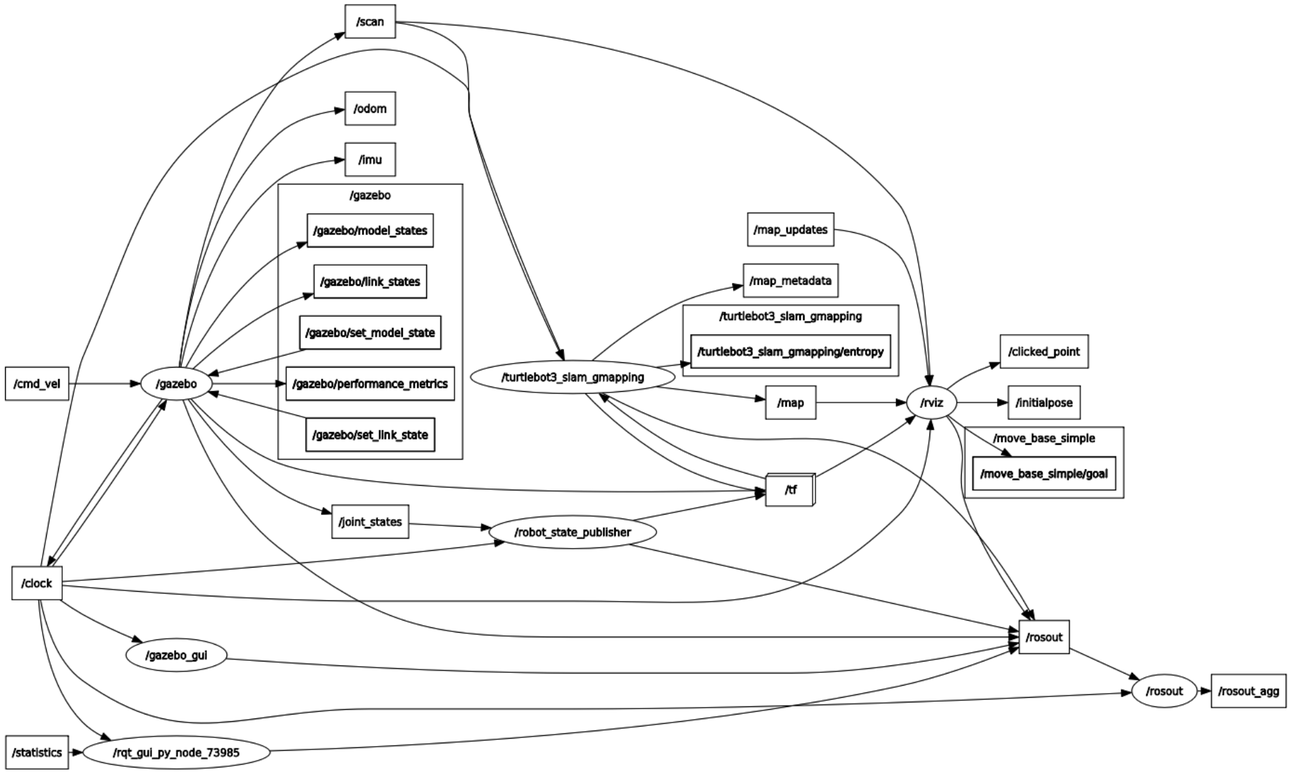
The computational graph in ROS1 provides a **flexible** and **modular architecture** for building complex robotics systems. Each node in the graph can perform a specific function, such as sensing, actuation, or computation, and nodes can communicate with each other through topics to exchange data and coordinate their behavior.

The computational graph consists of several components,

1. ROS Master
2. Parameter Server
3. Nodes
4. Messages
5. Topics
6. Service
7. Action

Nodes can be **added** or **removed** from the graph as needed, and different nodes can be combined together to create new functionalities or behaviors.

This allows ROS1 systems to be easily adapted to **new tasks or environments**, without requiring major changes to the underlying software architecture.



Turtlenot3 SLAM rqt_graph

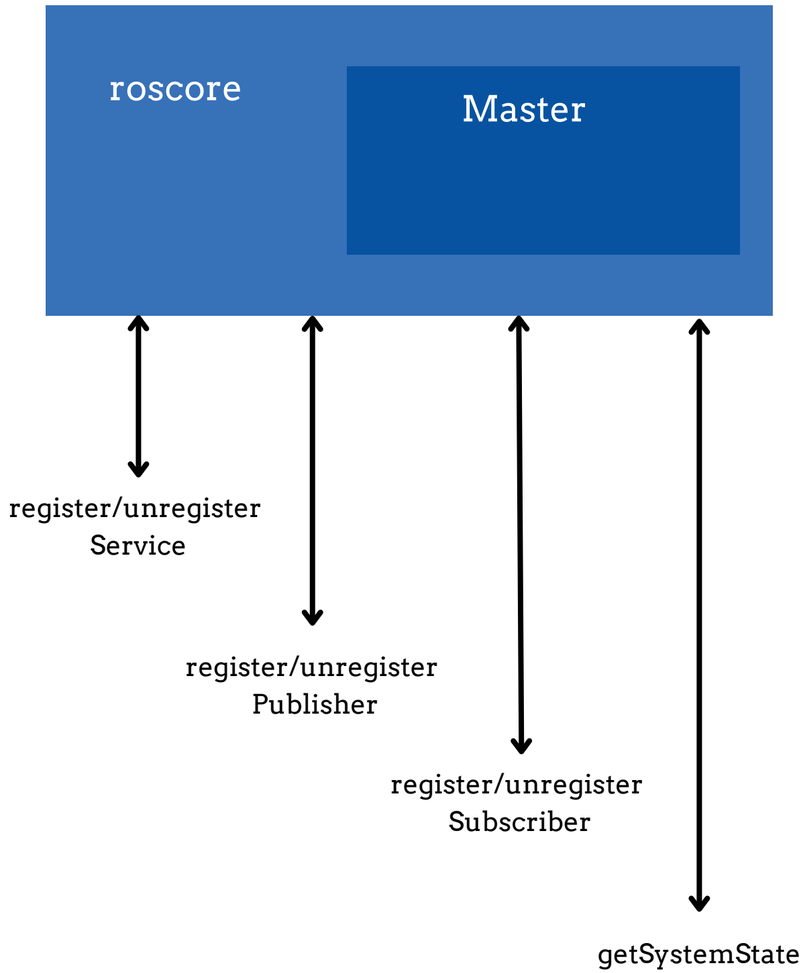
ROS Master

The ROS Master is another crucial component in the ROS1 architecture, as it provides a centralised registry of all available **nodes**, **topics**, and **services** in the system. It acts as a middleman between nodes, allowing them to discover and communicate with each other in a distributed system.

When a node starts up, **it registers with the Master** by providing its name, the topics it publishes to and subscribes from, and the services it provides or uses. The Master then keeps track of this information and provides it to other nodes that need to communicate with the registered node.

The Master also handles the **creation and destruction of topics and services**, as well as the **negotiation of communication protocols between nodes**. For example, when a node subscribes to a topic, it sends a request to the Master, which returns the IP address and port number of the publisher node. The subscribing node can then connect to the publisher node and begin receiving messages.

The Master uses the **XML-RPC protocol**, which is a stateless, HTTP-based protocol, to communicate with nodes. This means that the Master is lightweight and doesn't require a lot of computational resources.



ROS Master

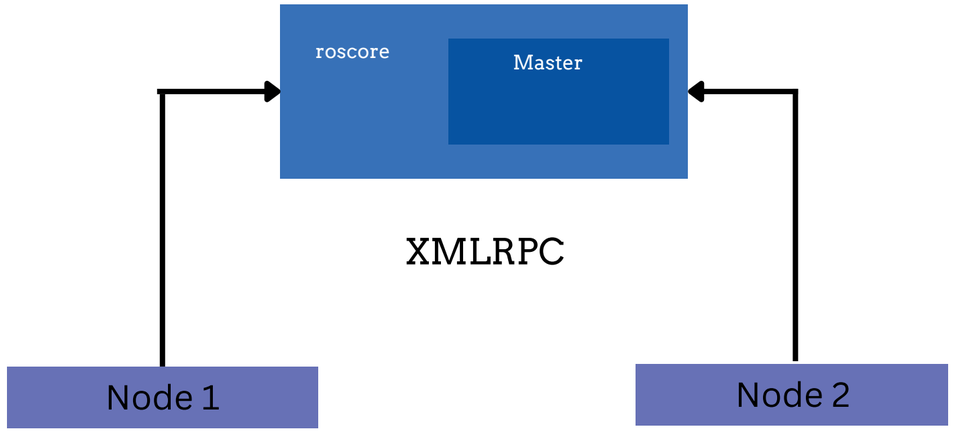
```
*[master][~/Desktop/mygit/ROS1-Workshop/catkin_ws/pub_sub]$ python2.7
Python 2.7.18 (default, Jul  1 2022, 12:27:04)
[GCC 9.4.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from xmlrpclib import ServerProxy
>>> import os
>>> master = ServerProxy(os.environ['ROS_MASTER_URI'])
>>> master.getSystemState('/')
[1, 'current system state', [[['/rosout_agg', ['/rosout']], ['/rosout', ['/publisher', '/subscriber']],
['/string_topic', ['/publisher']]], [['/rosout', ['/rosout']], ['/string_topic', ['/subscriber']]],
[['/rosout/get_loggers', ['/rosout']], ['/rosout/set_logger_level', ['/rosout']], ['/publisher/get_loggers',
['/publisher']], ['/publisher/set_logger_level', ['/publisher']], ['/subscriber/get_loggers', ['/subscriber']],
['/subscriber/set_logger_level', ['/subscriber']]]]]
>>> caller_id='/string_topic'
>>> master.getTopicTypes(caller_id)
[1, 'current system state', [['/rosout_agg', 'rosgraph_msgs/Log'], ['/rosout', 'rosgraph_msgs/Log'], ['/string_topic',
'std_msgs/String']]]
>>> master.getUri(caller_id)
[1, '', 'http://edgeai:11311/']
```

ServerProxy:
Instance is an object that manages communication with a remote XML-RPC server.
getSystemState:
Retrieve list representation of system state (i.e. publishers, subscribers, and services).
getTopicTypes:
Retrieve list topic names and their types.
getUri:
Get the URI of the master.

ROS Node

Node is a process that **performs a specific function** within the system. It communicates with other nodes using a **publish/subscribe** messaging model, which allows nodes to **send and receive data** without knowing the specifics of other nodes. A node can also provide and use **services**, which are **request/response** communication mechanisms that allow nodes to ask for specific actions or data from other nodes.

To interact with the system, a ROS1 node has several APIs, including a **slave API** that **receives callbacks from the Master** and negotiates connections with other nodes using an XMLRPC protocol.




```python
#!/usr/bin/env python
import rospy # Python client lib for ROS

rospy.init_node("hello_world")     # Initiate a node called hello_world
rate = rospy.Rate(1)               # We create a Rate object of 1Hz
while not rospy.is_shutdown():     # Continous loop
   print("My first ROS package Hello world ")
   rate.sleep()                    # We sleep the needed time to
                                   # maintain the above Rate
```

simple hello_world

```
# Commands to remember
*[master][~/Desktop/mygit/ROS1-Workshop/catkin_ws/src/pub_sub]$ rosnode list
/hello_world
/rosout

*[master][~/Desktop/mygit/ROS1-Workshop/catkin_ws/src/pub_sub]$ rosnode info /hello_world
--------------------------------------------------------------------------------
Node [/hello_world]
Publications: None

Subscriptions: None

Services: None

cannot contact [/hello_world]: unknown node
```

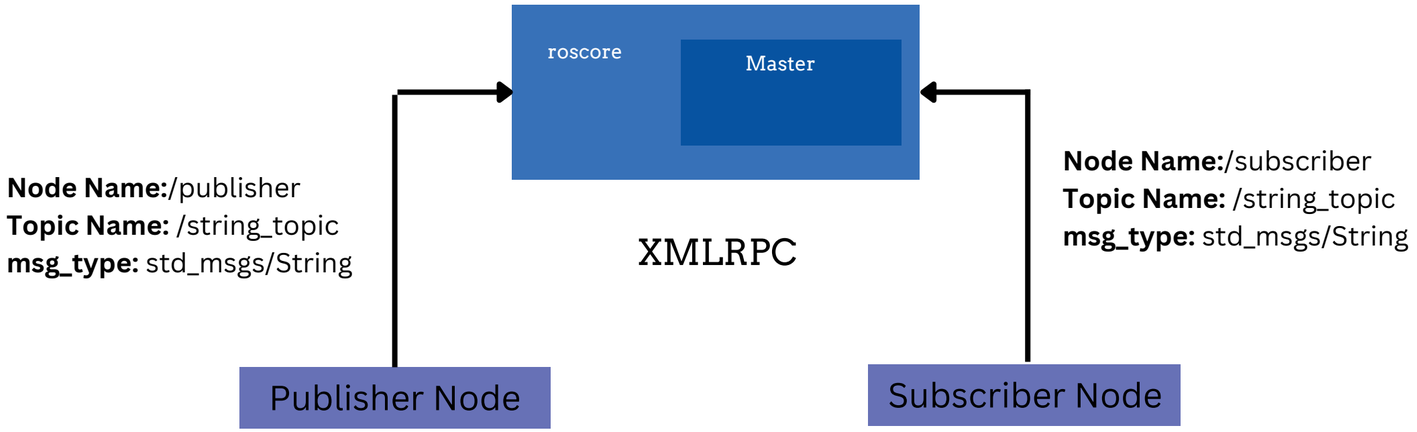
rosnode commandline

ROS Topics: Publisher and Subscriber

Publisher/Subscriber messaging mechanism that enables bi-directional communication between nodes in a ROS system. The Publisher/Subscriber model allows nodes to **send and receive data** without knowing anything about the other nodes in the system.

Publishers are nodes that create and send messages to a **specific topic**, while **subscribers are nodes that receive those messages** from the same topic. Multiple nodes can publish to the same topic, and multiple nodes can subscribe to the same topic.

In order for a subscriber node to receive messages from a publisher node, the subscriber must first register with the **publisher topic name** by subscribing to the same topic. Once the publisher and subscriber nodes are connected, the publisher can send messages to the topic, which will be received by all subscribed nodes.



ROS Topics: Publisher and Subscriber

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

import rospy
from std_msgs.msg import String

def publisher():
    # Initialize the node
    rospy.init_node('string_publisher', anonymous=True)

    # Create a publisher object with topic name, message type and queue size
    pub = rospy.Publisher('string_topic', String, queue_size=10)

    # Set the loop rate
    rate = rospy.Rate(10)

    # Publish messages until the node is shut down
    while not rospy.is_shutdown():
        # Create a message
        message = String()
        message.data = "Hello, World!"

        # Publish the message
        pub.publish(message)

        # Sleep to maintain the loop rate
        rate.sleep()

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

This line is called a "shebang" and is used to specify the interpreter.

These lines import the necessary ROS and message types. rospy is the Python client library for ROS, and String is a message type from the std_msgs package that represents a string.

This line initializes the ROS node with the name "string_publisher". The anonymous parameter is set to True, which appends a unique number to the end of the node name to ensure that each node has a unique name.

This line creates a ROS publisher object called pub. The publisher will send messages of type String on the string_topic topic, which has a queue size of 10.

This line creates a Rate object that will be used to regulate the loop rate of the publisher. In this case, the loop rate is set to 10 Hz.

This line begins a loop that will continue until the node is shut down.

These lines create a String message object and set its data field to "Hello, World!".

This line publishes the message object on the string_topic topic.

This line waits for the remainder of the loop cycle time to maintain the loop rate.

These lines define the main entry point of the script. The if statement checks if the script is being run as the main module, and if so, calls the publisher() function. The try-except block catches any ROSInterruptException exceptions that may be raised when the node is shut down.

ROS Topics: Publisher and Subscriber

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

import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo("Received message: %s", data.data)

def subscriber():
    # Initialize the node
    rospy.init_node('string_subscriber', anonymous=True)

    # Create a subscriber object with topic name, message
    # type and callback function
    rospy.Subscriber('string_topic', String, callback)

    # Spin until the node is shut down
    rospy.spin()

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

This is a callback function that will be executed whenever a new message is received on the subscribed topic. The data argument contains the message data.

This line initializes the ROS node with the given name, 'string_subscriber', and the anonymous parameter set to True. The anonymous parameter appends a unique identifier to the node name, making the node name unique.

This line creates a subscriber object for the topic 'string_topic' with message type String, and specifies the callback function callback to be executed whenever a message is received on the topic.

This line starts the subscriber node and blocks until the node is shutdown. It continuously processes incoming messages and executes the callback function for each message.

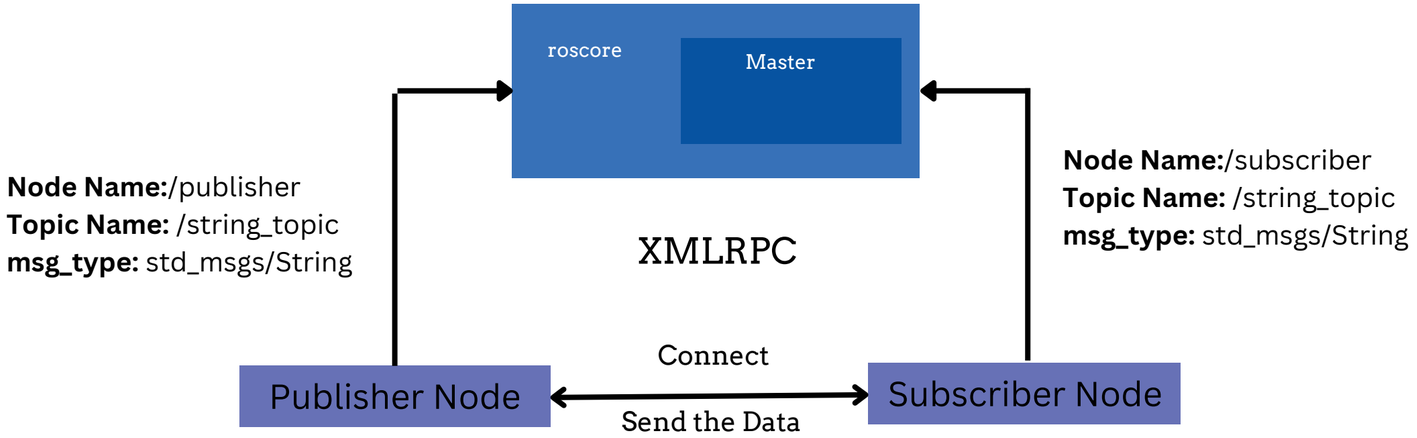
ROS Transports:

In ROS1, **TCPROS** and **UDPROS** are two common transportation mechanisms used for sending data between nodes.

TCPROS uses a persistent, stateful TCP/IP socket connection and ensures that packets always arrive in order. When a publisher wants to establish a topic connection with a subscriber, **it gives the subscriber its IP address and port number**. The subscriber then creates a TCP/IP socket to the specified address and port, and the nodes exchange a Connection Header that includes information such as the MD5 sum of the message type and the name of the topic. Once the connection is established, **the publisher begins sending serialized message data directly over the socket to the subscriber.**

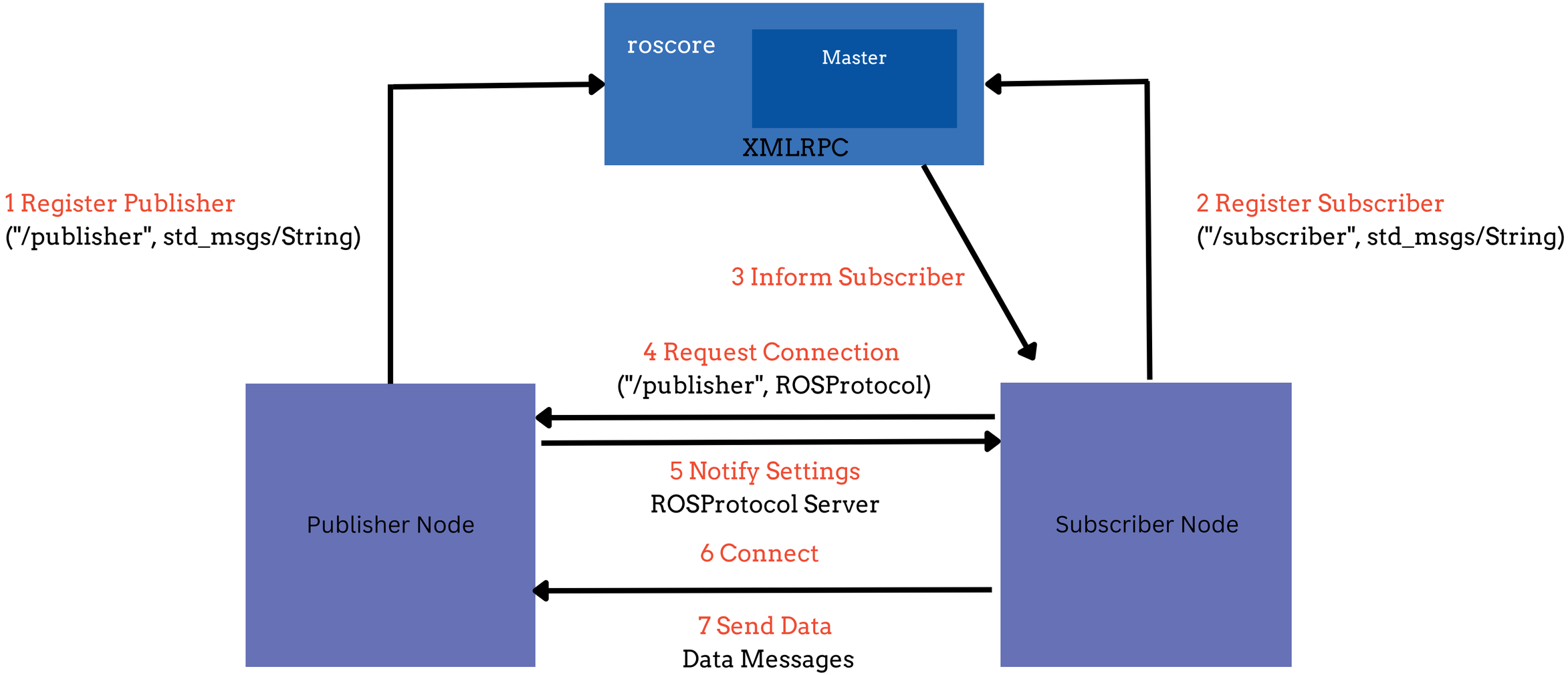
UDPROS, on the other hand, uses a stateless, unreliable UDP/IP transport. With UDPROS, the publisher broadcasts its message to all subscribers on the topic, and subscribers can choose which messages to process based on their individual requirements. However, because **UDPROS is a stateless protocol, it does not guarantee that packets will arrive in order or that they will arrive at all.**

Note: The XMLRPC system is used by the ROS master to manage the communication between nodes. So, while the master plays an important role in managing the ROS network and connecting nodes, it is not involved in the actual data exchange between nodes.



Lets connect the dots...

Establishing a Topic Connection and Publish the Data



ROS Launch

To run our previous pub_sub example,

```
# Terminal 1
$ source /opt/ros/noetic/setup.zsh
$ roscore # Run ros core/ ros master

# Terminal 2
$ source /opt/ros/noetic/setup.zsh
$ rosrun pub_sub subscriber.py # Run the publisher node using rosrun

# Terminal 3
$ source /opt/ros/noetic/setup.zsh
$ rosrun pub_sub publisher.py # Run the subscriber node using rosrun
```

```
/clicked_point
/clock
/cmd_vel
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/performance_metrics
/gazebo/set_link_state
/gazebo/set_model_state
/imu
/initialpose
```

```
/joint_states
/map
/map_metadata
/map_updates
/move_base_simple/goal
/odom
/rosout
/rosout_agg
/scan
/tf
/tf_static
/turtlebot3_slam_gmapping/entropy
```

Turtlebot3 SLAM Simulation

When dealing with a complex robotics system that involves **multiple nodes**, it becomes **impractical to manually launch each node using the rosrun command**. In such cases, it is more convenient to use a launch file that can simultaneously start multiple nodes with their respective parameters and configurations.

a launch file is **an XML file** that allows you to **start multiple ROS nodes with a single command**. It simplifies the process of starting and managing multiple nodes that need to work together in a specific way.

A launch file typically defines **one or more nodes**, along with their **parameters**, **topics**, and **services**. It can also include **arguments that can be passed to the nodes at runtime**, allowing for greater flexibility and reusability.

Launch files can be used for a wide range of tasks, from starting a single node with specific settings to launching complex robot control systems with multiple nodes and sensors.

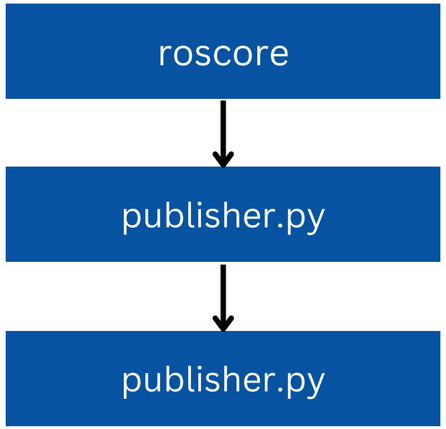
ROS Launch

```
<launch>
    <!-- Start the publisher node -->
    <node pkg="pub_sub" type="publisher.py" name="publisher"  output="screen">
    </node>

    <!-- Start the subscriber node -->
    <node pkg="pub_sub" type="subscriber.py" name="subscriber"  output="screen">
    </node>
</launch>
```

pkg: specifies the name of the package that contains the node
type: specifies the name of the Python script that should be executed to start the node
name: specifies the name of the node
output: specifies how the output from the node should be displayed. In this case, it is set to "screen", which means that output will be printed to the terminal.

Note:
It is important to note that when using roslaunch, there is no need to manually start a roscore instance as roslaunch takes care of starting it automatically in the background.

roslaunch pub_sub pub_sub.launch



ROS Launch

Here's an example launch file that includes various options such as argument passing and setting parameters:

```
<launch>

  <!-- Define an argument for the robot name -->
  <arg name="robot_name" default="turtlebot3" />

  <!-- Load the robot description parameter file -->
  <param name="robot_description" textfile="$(find my_robot)/urdf/$(arg robot_name).urdf" />

  <!-- Start the robot state publisher -->
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" />

  <!-- Start the joint state publisher -->
  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher">
    <!-- Set the robot name as an argument -->
    <param name="source_list" value="$(arg robot_name)_joint_states" />
  </node>

  <!-- Start the Gazebo simulator -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <!-- Set the robot name as an argument -->
    <arg name="world_name" value="$(find my_robot)/worlds/$(arg robot_name).world" />
  </include>

  <!-- Start the RViz visualization tool -->
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find my_robot)/rviz/$(arg robot_name).rviz" />

</launch>
```

Indicates the start of the launch file

Defines an argument named robot_name with a default value of turtlebot3.

Sets a parameter named robot_description to the contents of the URDF file for the specified robot name.

Starts the robot_state_publisher node from the robot_state_publisher package.

Starts the joint_state_publisher node from the joint_state_publisher package.

Sets a parameter for the joint_state_publisher node, using the value of the robot_name argument to construct the topic name.

Includes another launch file for starting the Gazebo simulator.

Sets an argument for the included launch file, using the value of the robot_name argument to construct the world file path.

Starts the rviz node from the rviz package, with a specified configuration file based on the value of the robot_name argument.

ROS Message

Topics are called “buses,” and nodes send messages to each other over them.

Messages are **the bits of data that are sent from one node to another over a topic**. They are written in the **ROS Message Description Language (.msg)** and are used to describe how the data being sent is structured. Messages can contain many different types of data, from simple ones like **integers** and floats to more complicated ones like **arrays** and **custom message types**.

ROS provides many built-in message types for various data types.

GridCells
MapMetaData
OccupancyGrid
Odometry
Path

GetMap
GetPlan
LoadMap
SetMap

GetMap

nav_msgs

Bool
Byte
ByteMultiArray
Char
ColorRGBA
Duration
Empty
Float32
Float32MultiArray
Float64
Float64MultiArray
Header
Int16
Int16MultiArray
Int32
Int32MultiArray
Int64
Int64MultiArray
Int8
Int8MultiArray
MultiArrayDimension
MultiArrayLayout
String
Time
UInt16
UInt16MultiArray
UInt32
UInt32MultiArray
UInt64
UInt64MultiArray
UInt8
UInt8MultiArray

std_msgs

BatteryState
CameraInfo
ChannelFloat32
CompressedImage
FluidPressure
Illuminance
Image
Imu
JointState
Joy
JoyFeedback
JoyFeedbackArray
LaserEcho
LaserScan
MagneticField
MultiDOFJointState
MultiEchoLaserScan
NavSatFix
NavSatStatus
PointCloud
PointCloud2
PointField
Range
RegionOfInterest
RelativeHumidity
Temperature
TimeReference

SetCameraInfo

sensor_msgs

ROS Custom Messages

```
mkdir ~/catkin_ws/src/  # Create a directory
cd ~/catkin_ws/src/     # Move to src directory
catkin_create_pkg custom_msg rospy
cd custom_msg

touch msg/my_robot_msgs.msg # Create a msg file

# CMakeLists.txt changes
find_package(catkin REQUIRED COMPONENTS
  rospy
  dynamic_reconfigure
  message_generation
  geometry_msgs
)

add_message_files(
  FILES
  my_robot_msgs.msg
)

generate_messages(
  DEPENDENCIES
  geometry_msgs
)

catkin_package(
  CATKIN_DEPENDS dynamic_reconfigure message_runtime geometry_msgs
)

# package.xml changes
  <buildtool_depend>catkin</buildtool_depend>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>message_generation</build_depend>
  <exec_depend>message_runtime</exec_depend>
  <depend>dynamic_reconfigure</depend>
  <build_depend>geometry_msgs</build_depend>
  <build_depend>rospy</build_depend>
  <exec_depend>geometry_msgs</exec_depend>
  <build_export_depend>rospy</build_export_depend>
  <exec_depend>rospy</exec_depend>
# Build the package
cd ~/catkin_ws/ # go to root directory of the packages
catkin_make # Compile a ROS package
source devel/setup.bash # Source the ROS env variable

$ rosmsg list |grep custom
$ rosmsg show custom_msg/my_robot_msgs
```

Steps to create a custom message

```
# my_robot_msgs.msg
# Definition of the Pose message
# It contains the position and orientation of an object in 3D space

geometry_msgs/Point position  # 3D position of the object
geometry_msgs/Quaternion orientation  # Orientation of the object in 3D space
```

my_robot_msgs.msg

```
message = my_robot_msgs()
message.position.x = 1.0
message.position.y = 2.0
message.position.z = 0.5
message.orientation.x = 0.0
message.orientation.y = 0.0
message.orientation.z = 0.0
message.orientation.w = 1.0
```

Msg definition

Note:
The generated msg serialised file you can find here,
./devel/include/custom_msg/my_robot_msgs.h

ROS Custom Messages - Publisher & Subscriber

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

import rospy
from custom_msg.msg import my_robot_msgs

def publisher():
    # Initialize the nodes
    rospy.init_node('pose_publisher', anonymous=True)

    # Create a publisher object with topic name, message type and queue
size
    pub = rospy.Publisher('pose_topic', my_robot_msgs, queue_size=10)

    # Set the loop rate
    rate = rospy.Rate(10)

    # Publish messages until the node is shut down
    while not rospy.is_shutdown():
        # Create a message
        message = my_robot_msgs()
        message.position.x = 1.0
        message.position.y = 2.0
        message.position.z = 0.5
        message.orientation.x = 0.0
        message.orientation.y = 0.0
        message.orientation.z = 0.0
        message.orientation.w = 1.0

        # Publish the message
        pub.publish(message)

        # Sleep to maintain the loop rate
        rate.sleep()

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

publisher.py

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

import rospy
from custom_msg.msg import my_robot_msgs

def callback(data):
    rospy.loginfo("Received pose: x=%f, y=%f, z=%f, qx=%f, qy=%f, qz=%f,
qw=%f", data.position.x, data.position.y, data.position.z,
data.orientation.x, data.orientation.y, data.orientation.z,
data.orientation.w)

def subscriber():
    # Initialize the node
    rospy.init_node('pose_subscriber', anonymous=True)

    # Create a subscriber object with topic name, message type and callback
function
    rospy.Subscriber('pose_topic', my_robot_msgs, callback)

    # Spin until the node is shut down
    rospy.spin()

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

subscriber.py

ROS Parameter Server

The ROS Parameter Server is a **key-value store** where nodes can store and retrieve parameters at **runtime**. It allows nodes to share configuration data without the need for direct communication between them.

The Parameter Server can be used to store various types of data, such as integers, floats, strings, and arrays. Nodes can retrieve parameter values from the server during runtime to adjust their behaviour accordingly. The parameter values can be set manually using the command-line tool "**rosparam**" or **programmatically** using ROS libraries.

The Parameter Server is useful for **managing system-wide configuration settings**, such as calibration values or default behavior settings. It can also be used to pass parameters between nodes, allowing for more dynamic and flexible system behavior.

```
mkdir config/ && touch config/param.yaml

# param.yaml
Pose:
  position:
    x: 100
    y: 50
    z: 25
  orientation:
    x: 1.0
    y: 2.0
    z: 3.0
    w: 10.0

# custom_msg.launch
<launch>
    <rosparam file="$(find custom_msg)/config/param.yaml" />

    <!-- Start the publisher node -->
    <node pkg="custom_msg" type="publisher.py" name="publisher"
 output="screen">
    </node>

    <!-- Start the subscriber node -->
    <node pkg="custom_msg" type="subscriber.py" name="subscriber"
 output="screen">
    </node>
</launch>
```

```
message = my_robot_msgs()
message.position.x = rospy.get_param("/Pose/position/x")
message.position.y = rospy.get_param("/Pose/position/y")
message.position.z = rospy.get_param("/Pose/position/z")
message.orientation.x = rospy.get_param("/Pose/orientation/x")
message.orientation.y = rospy.get_param("/Pose/orientation/y")
message.orientation.z = rospy.get_param("/Pose/orientation/z")
message.orientation.w = rospy.get_param("/Pose/orientation/w")
```

publisher.py changes

ROS Topics with ROS Perception Packages

```
# Create a package
catkin_create_pkg image_pipeline rospy cv_bridge image_transport sensor_msgs

# Create a Publisher and subscriber
cd image_pipeline/ && touch src/publisher.py
chmod a+x src/publisher.py
touch src/subscriber.py
chmod a+x src/subscriber.py

# Create a param file
mkdir config/ && touch config/param.yaml

# Create a Launch file
mkdir launch && touch launch/image_pipeline.launch

# Compile
catkin_make
source devel/setup.zsh

# Run
roslaunch image_pipeline image_pipeline.launch
```

cv_bridge is a package in ROS that provides functionality to convert between ROS messages and OpenCV images. It is used to bridge the gap between the image data in a ROS message and OpenCV data structures. cv_bridge is used extensively in **ROS applications that deal with image processing, such as object detection, tracking, and segmentation.**

image_transport is a package that provides a set of plugins for efficient transport of images over ROS. **It provides a publish-subscribe mechanism for sending and receiving images between nodes.** image_transport uses a transport layer that optimizes the transmission of image data, reducing latency and bandwidth usage. It also provides features such as compression and encryption.

sensor_msgs is a package that defines messages for commonly used sensor data, including images, point clouds, and IMU data. It provides a standard interface for sensor data in ROS applications, allowing different sensor types to communicate with each other seamlessly. sensor_msgs/Image message is used for the transmission of images in ROS applications. It contains fields such as image height, width, encoding, and pixel data.

ROS Topics with ROS Perception Packages

```python
#!/usr/bin/env python3
import sys
import rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
import cv2

class Publisher:
    def __init__(self):
        self.image_pub = rospy.Publisher("publisher", Image, queue_size=100)
        self.bridge = CvBridge()
        self.image_publisher()

    def image_publisher(self,):

        frequency = rospy.get_param("/image_acquisition/frequency")
        rate = rospy.Rate(frequency)

        input_type = rospy.get_param("/image_acquisition/input_type")

        video_capture = None
        if input_type == "camera_usb":
            video_capture = cv2.VideoCapture(0)
        else:
            video_path =
rospy.get_param("/image_acquisition/video/video_path_0")
            video_capture = cv2.VideoCapture(video_path)

        while True:
            ret, frame = video_capture.read()
            if frame is not None:
                msg = self.bridge.cv2_to_imgmsg(frame, "bgr8")
                self.image_pub.publish(msg)
                rate.sleep()

def main(args):
    rospy.init_node('publisher')
    Publisher()
if __name__ == '__main__':
    main(sys.argv)
```

publisher.py

```python
#!/usr/bin/env python3
import sys
import rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
import cv2

class Subscriber:
    def __init__(self):
        self.image_sub = rospy.Subscriber("publisher", Image, self.callback)
        self.bridge = CvBridge()

    def callback(self,data):
        try:
            cv_image = self.bridge.imgmsg_to_cv2(data, "bgr8")
            cv2.imshow("Display", cv_image)
            cv2.waitKey(1)

        except CvBridgeError as e:
            print(e)

def main(args):
    rospy.init_node('subscriber')
    Subscriber()
    try:
        rospy.spin()
    except KeyboardInterrupt:
        print("Shutting down")

if __name__ == '__main__':
    main(sys.argv)
```

subscriber.py

ROS Topics with ROS Perception Packages

```
<launch>
    <rosparam file="$(find image_pipeline)/config/param.yaml" />
    <!-- image_pipeline launch file -->
    <node pkg="image_pipeline" type="publisher.py" name="publisher"
 output="screen">
    </node>
    <node pkg="image_pipeline" type="subscriber.py" name="subscriber"
 output="screen">
    </node>
</launch>
```

image_pipeline.launch

```
image_acquisition:
  input_type: "video" #camera_usb, video

  camera_usb:
      camera_node: 1

  video:
      video_node: 1
      video_path_0: "/home/nullbyte/Desktop/mygit/ROS1-
Workshop/catkin_ws/test/test_1.mp4"

  frequency: 10
```

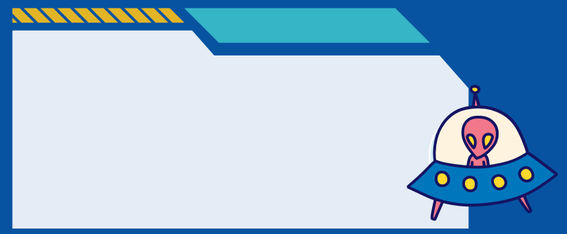
param.yaml

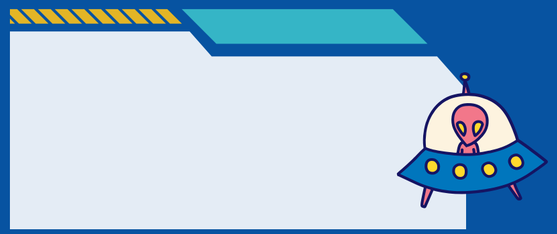
ROS Services Server & Client

The ROS service provides a s**ynchronous request/response communication mechanism** between nodes. Unlike topics, services are not intended for continuous data streams but for occasional requests that require a response.

A service is defined by **two messages**: one for the **request** and one for the **response**. When a node calls a service, it sends a request message to the service server and waits for the server to send a response message. The server processes the request and sends back a response message to the client.

A common use case for services is to perform a one-time action or query. For example, a service could be used to turn on or off an actuator, request sensor data, or change a configuration parameter.





THANK YOU

FOLLOW ME