THE BIOROBOTICS INSTITUTE

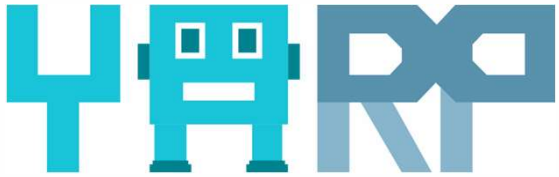
Sant'Anna
School of Advanced Studies – Pisa

ROS and YARP
Robot Software Platforms

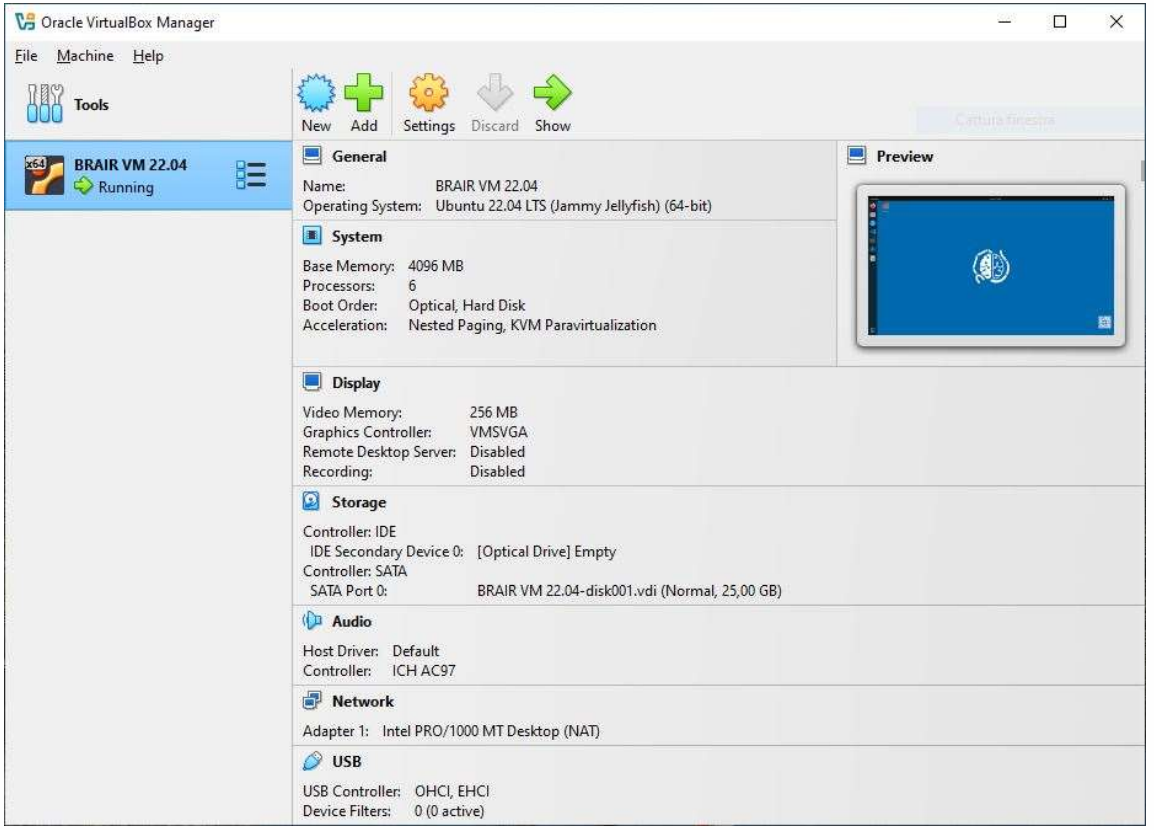
Egidio Falotico - e.falotico@santannapisa.it
Ugo Albanese - u.albanese@santannapisa.it

Outline

- Introduction to Robot Software Platforms
  - ROS
    - Intro to Communication mechanisms
  - YARP
    - Intro to Communication mechanisms
  - Programming
    - with ROS
    - with YARP
    - ROS for robot control

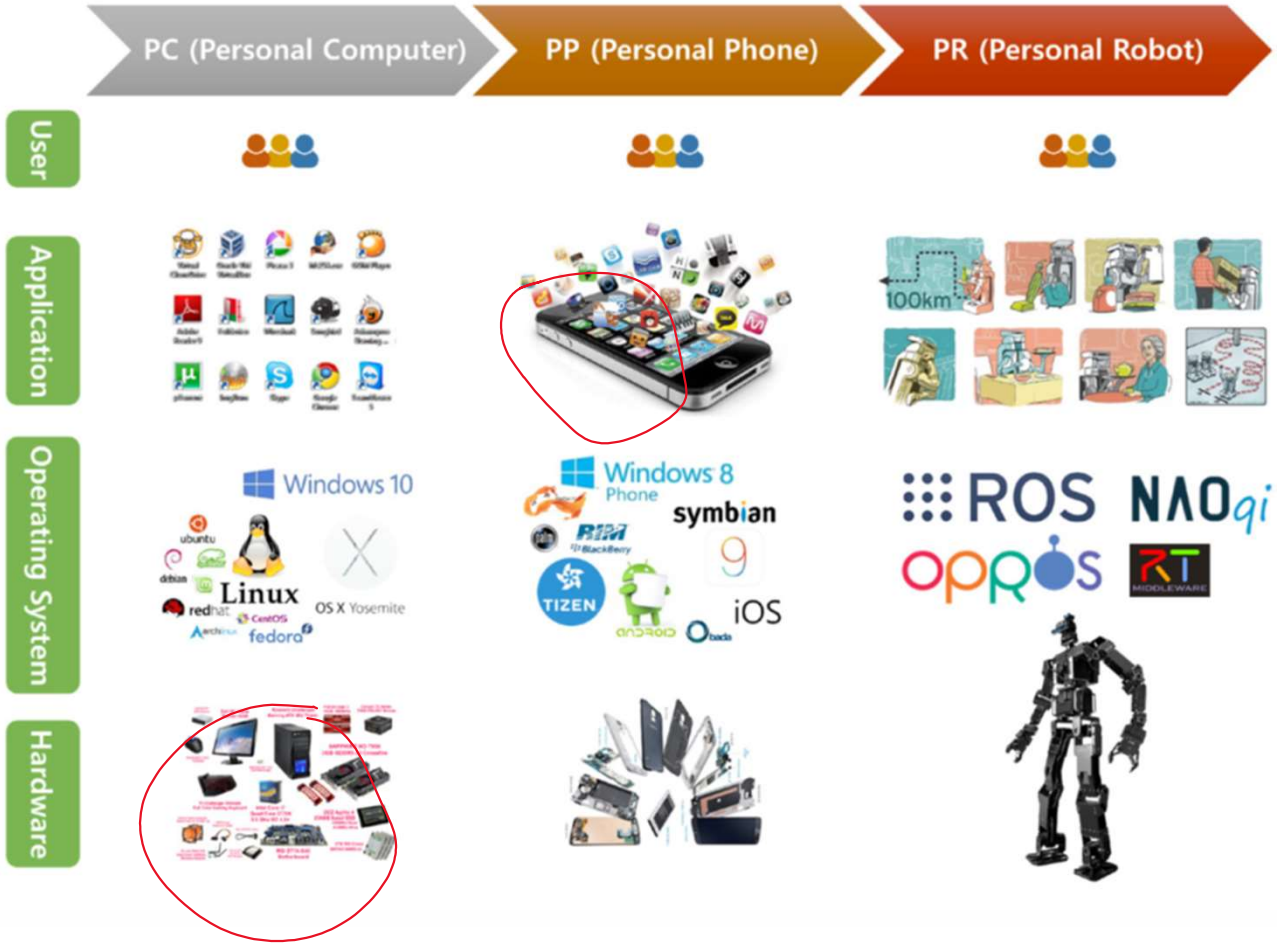


Tools



- Download VirtualBox
- Import the `.ova` file
- Follow the instructions in the `Readme.md` file

Platforms Components



Robot Software Platform

A platform is divided into *software* and *hardware.*

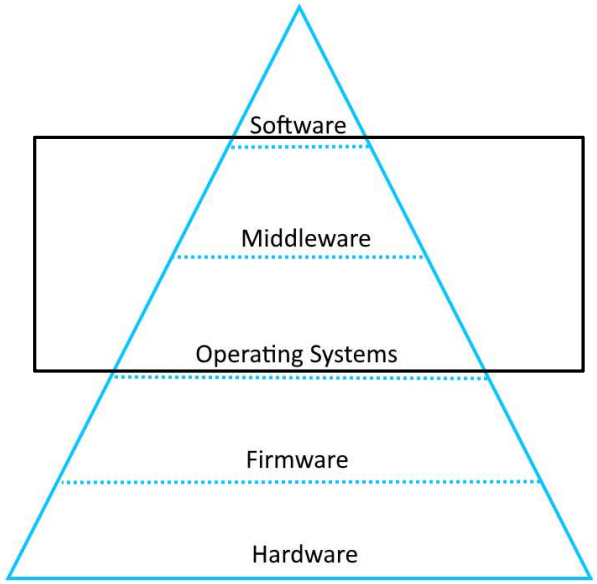
A robot *software* platform includes tools that are used to develop robot application programs such as:

- hardware abstraction
- low-level device
- control
- sensing, recognition
- SLAM (Simultaneous Localization And Mapping)
- navigation
- manipulation and package management
- libraries
- debugging and development tools

Why a Robot Software Platform

- What is noteworthy is that this hardware abstraction is occurring in conjunction with the aforementioned software platforms, making it possible to develop application programs using a software platform even without having expertise in hardware.

- This is the same with how we can develop mobile apps without knowing the hardware composition or specifications of the latest smartphone.



Why should we use a Robot Software Platform

- **Reusability of the program**
  - Focus only on features of interest
- **Communication-based program**
  - Allows modularization
- **Availability of support and development tools**
  - Debugging, visualization
- **Active community**
  - Sharing and collaborating to improve and speed up the development

ROS

Robot Operating System

ROS is a *meta* operating system

- An Operating System (OS) performs processes such as scheduling, loading, monitoring and error handling by utilizing *virtualization* layer between applications and distributed computing resources

- ROS runs on the existing OS acting as a **middleware** (plumbing)

Objectives of ROS

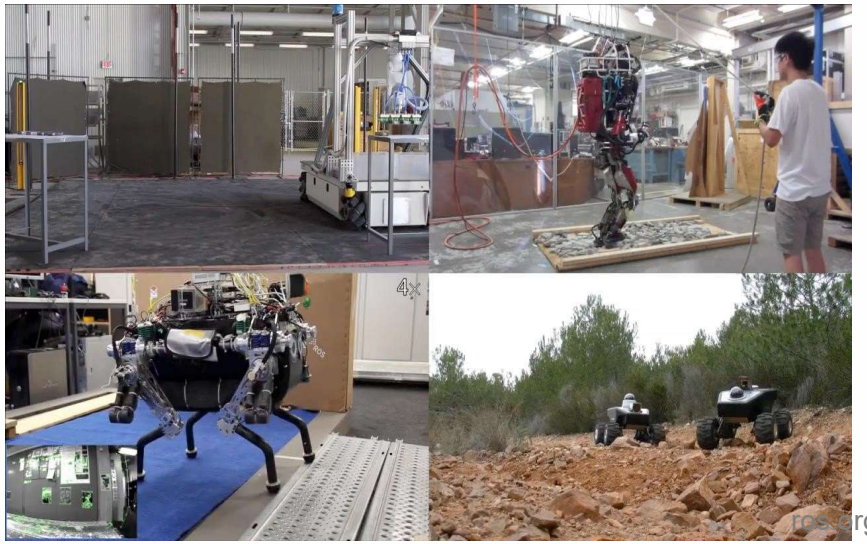
- ROS is focused on **maximizing code reuse** in the robotics research and development.
- To support this, ROS has the following characteristics:
  - Distributed process
  - Package management
  - Public repository + API
  - Supporting different programming languages: Python, C++, Java, Matlab

History of ROS

- Originally developed in 2007 at the Stanford Artificial Intelligence Laboratory
  - ROS1 popularized by the robotics incubator, Willow Garage
- Since 2013 managed by OSRF that became Open Robotics in May 2017
- Today used by many robots, universities and companies
  - De-facto standard for open-source robot programming (e.g. research)
- ROS2 is its latest incarnation

ROS2

- ROS1 design focused on having a quality and performant system, but *security*, *network topology*, and *system up-time were not prioritized*.
- As commercial opportunities transitioned into products, ROS's foundation as a research platform began to show its limitations
- *Security*, *reliability* in non-traditional environments, and support for large *scale embedded systems* became essential



ROS2

- Redesigned from the ground up to address these challenges while building on the success of ROS1.

- Based on the *Data Distribution Service* (**DDS**), an open standard for communications that is used in critical infrastructure such as military, spacecraft, and financial systems
  - DDS enables ROS2 to obtain best-in-class *security*, embedded and *realtime* support, multi-robot communication, and operations in non-ideal networking environments

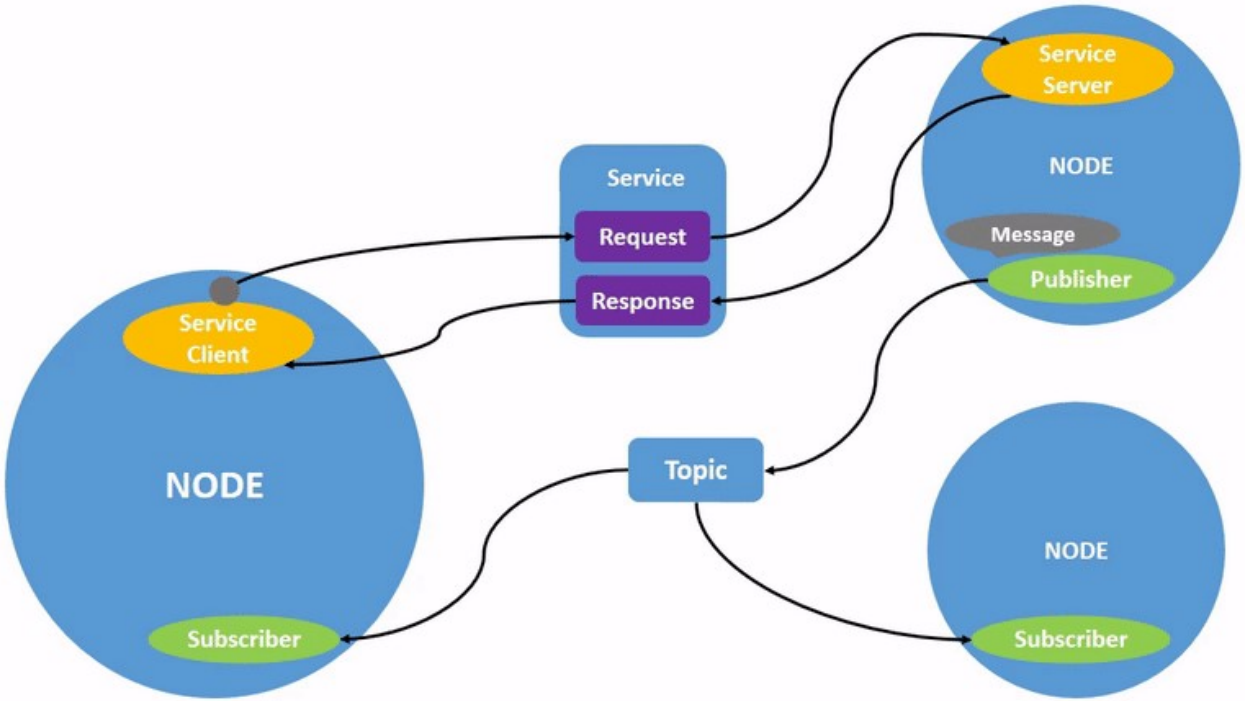
ROS

DDS

ros.org

ROS Graph

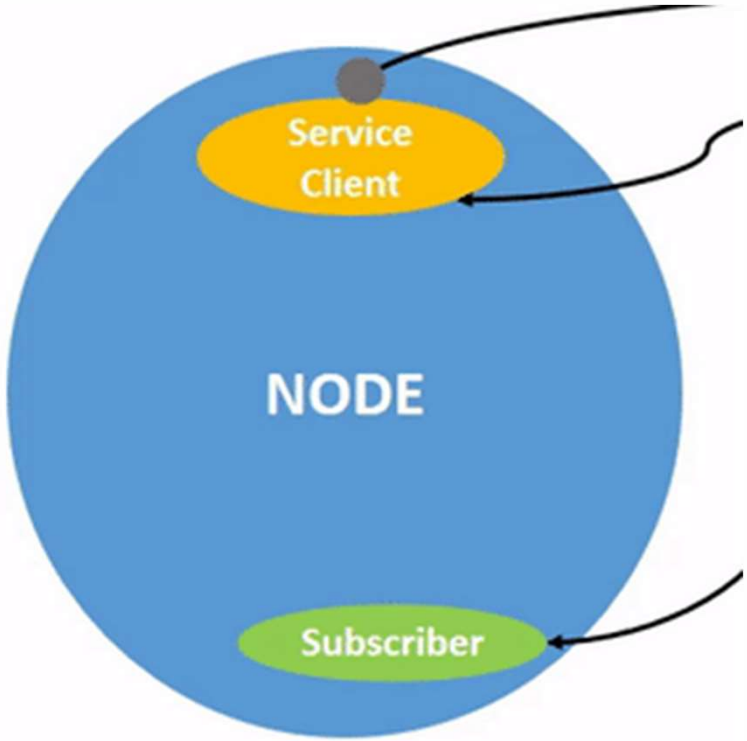
- A network of ROS 2 elements processing data together at the same time.
- It encompasses all executables and the connections between them if you were to map them all out and visualize them.



ROS Node

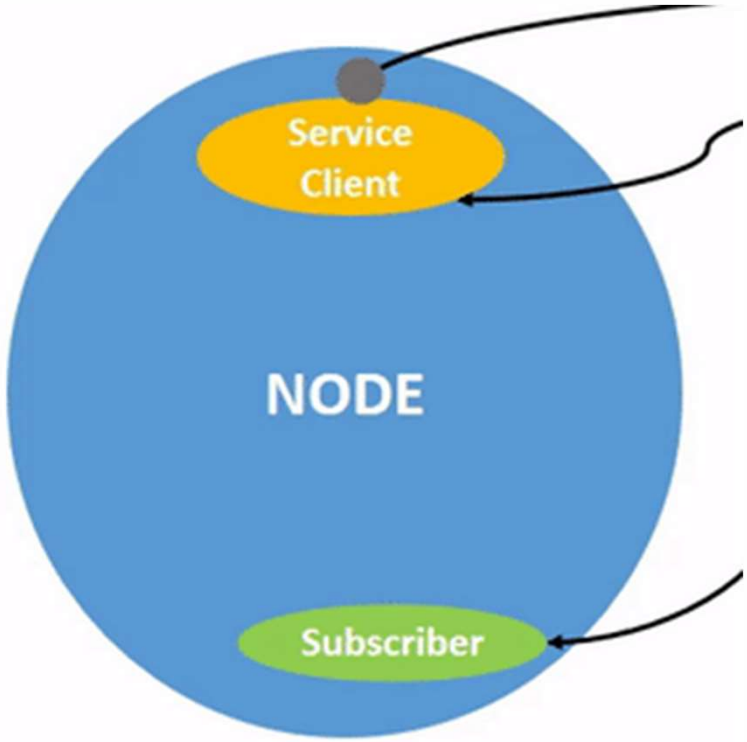
- Each **node** in ROS should be responsible for a ***single***, ***modular purpose***, e.g. controlling the wheel motors or publishing the sensor data from a laser range-finder. Each node can *send* and *receive* data from other nodes via **topics**, **services**, **actions**, or **parameters**.

- A full robotic system is comprised of many nodes working in concert. In ROS 2, a single executable (C++ program, Python program, etc.) can contain one or more nodes.



ROS Node

- A **node** is a participant in the ROS 2 graph, communicating with other nodes within the same process, in a different process, or on a different machine.
- **Nodes** are typically the unit of computation in a ROS graph; each node should do one logical thing.
- Connections between **nodes** are established through a *distributed* discovery process.

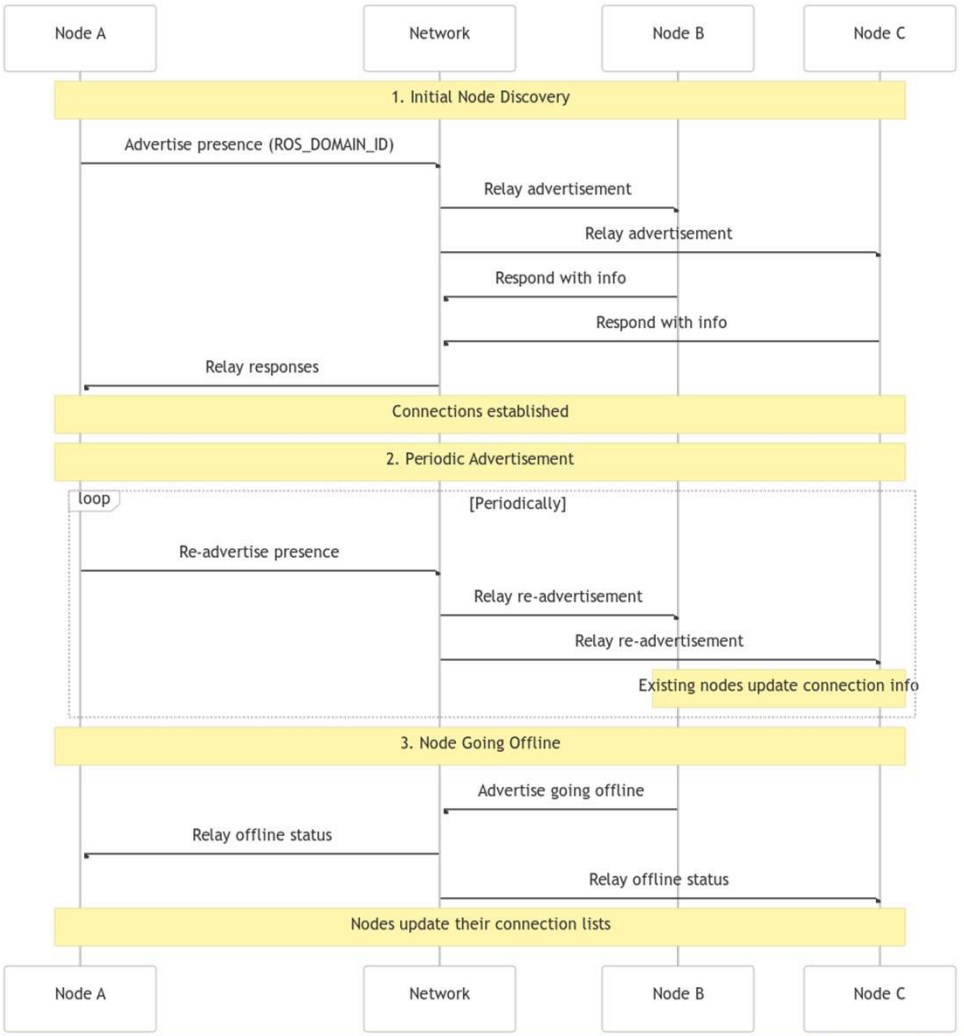


ROS Node Discovery

1. When a node is started, it advertises its presence to other nodes on the network with the same ROS domain (set with the `ROS_DOMAIN_ID` environment variable). Nodes respond to this advertisement with information about themselves so that the appropriate connections can be made and the nodes can communicate.

2. Nodes periodically advertise their presence so that connections can be made with new-found entities, even after the initial discovery period.

3. Nodes advertise to other nodes when they go offline.



ROS Nodes

Execution of a node with

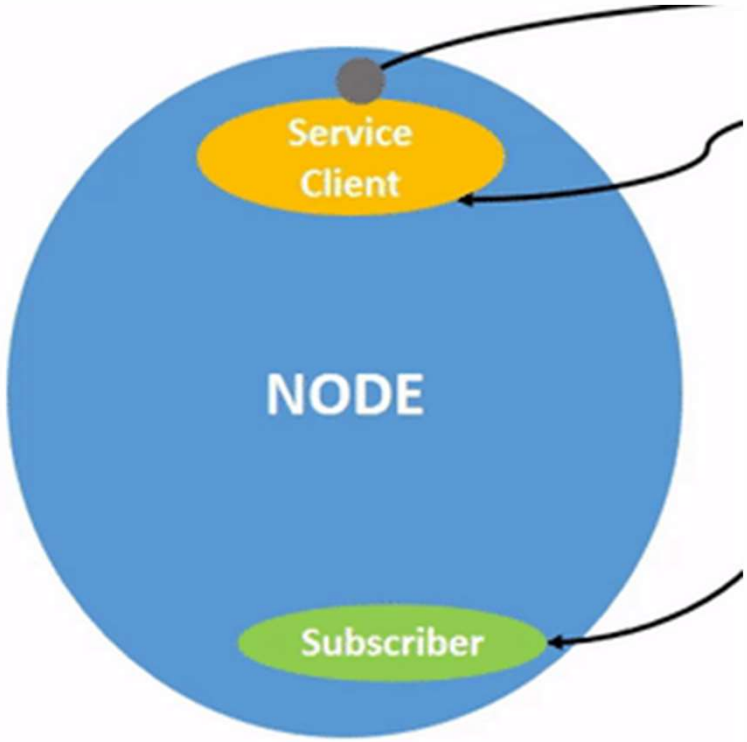
```
> ros2 run package_name executable_name
```

Check active nodes

```
> ros2 node list
```

Get information about a node with

```
> ros2 node info node_name
```

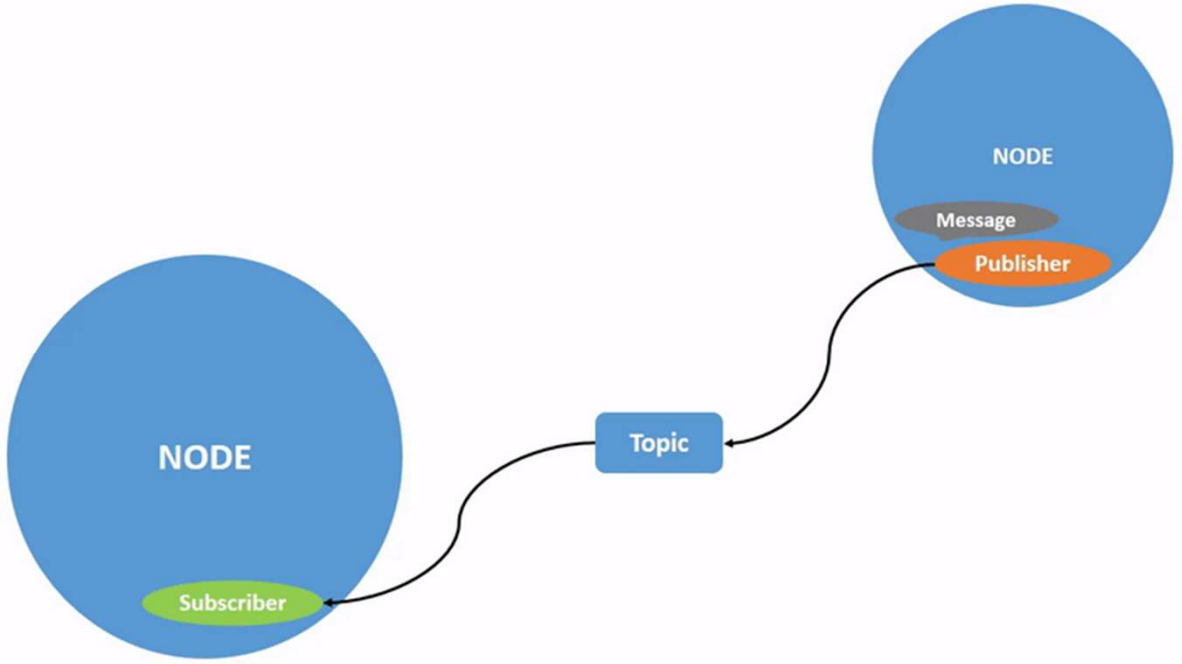


Package

- A package is the basic unit of distribution of ROS software.
- The ROS application is developed on a package basis, and the package contains a configuration file to launch other packages or nodes.
- The package also contains all the files necessary for running the package, including ROS dependency libraries for running various processes, datasets, and configuration file.

ROS Topics

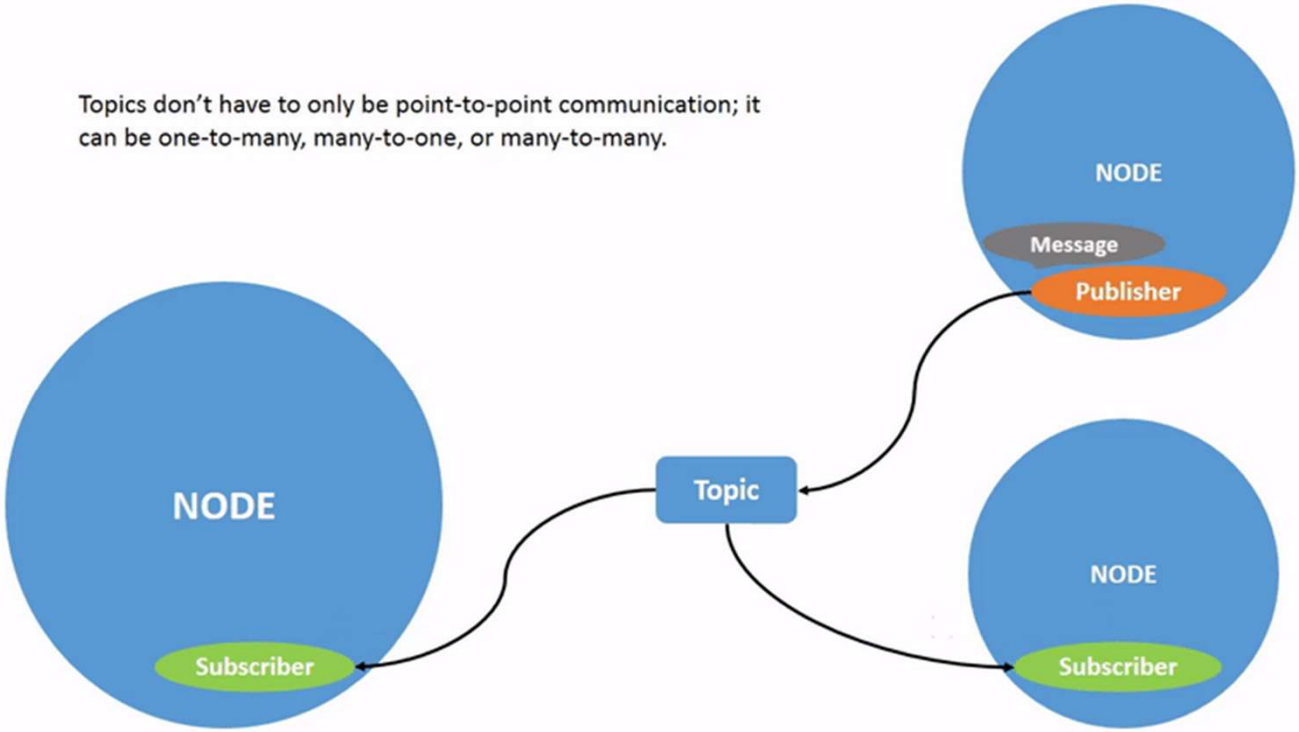
- Topics are a vital element of the ROS graph that act as a *bus* for nodes to exchange messages.
- One of the three primary styles of interfaces provided by ROS 2.
- Should be used for continuous data streams, like sensor data, robot state, etc.



ROS Topics

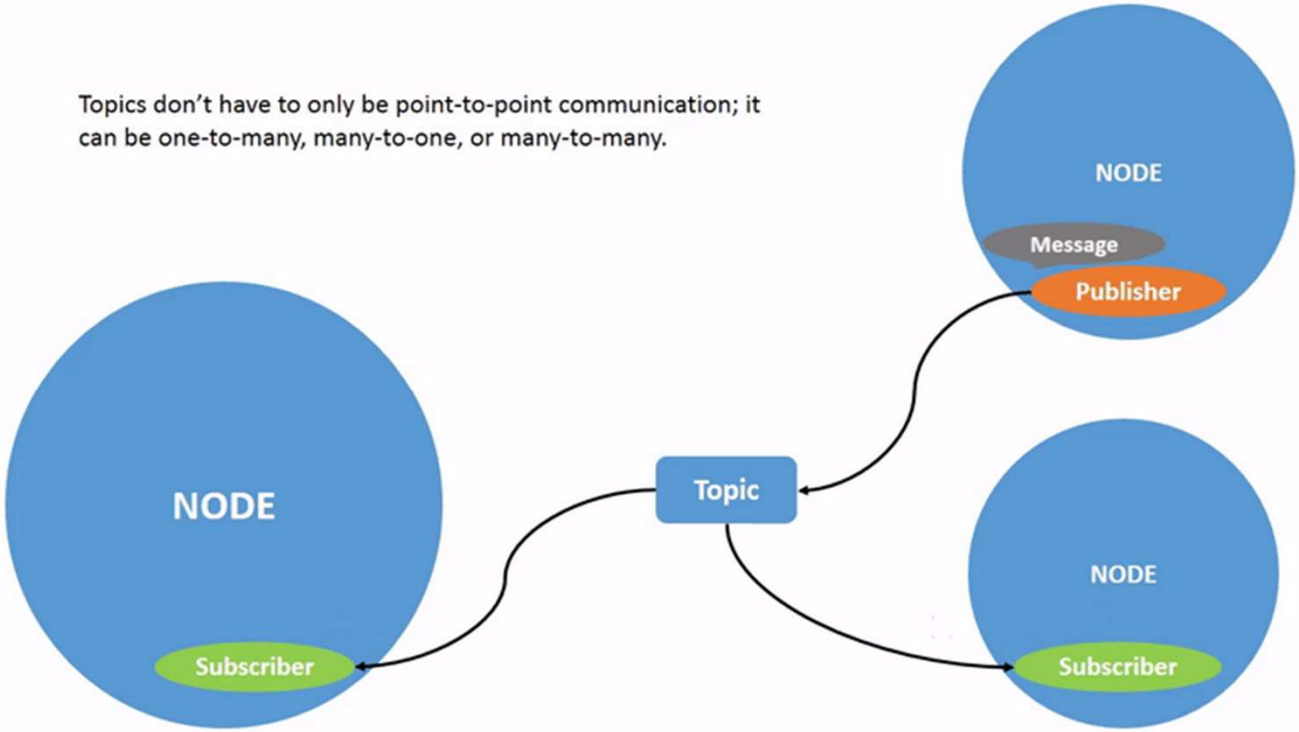
- A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics.

- A communication system:
  - **strongly-typed**
  - **anonymous**
  - **publish/subscribe**



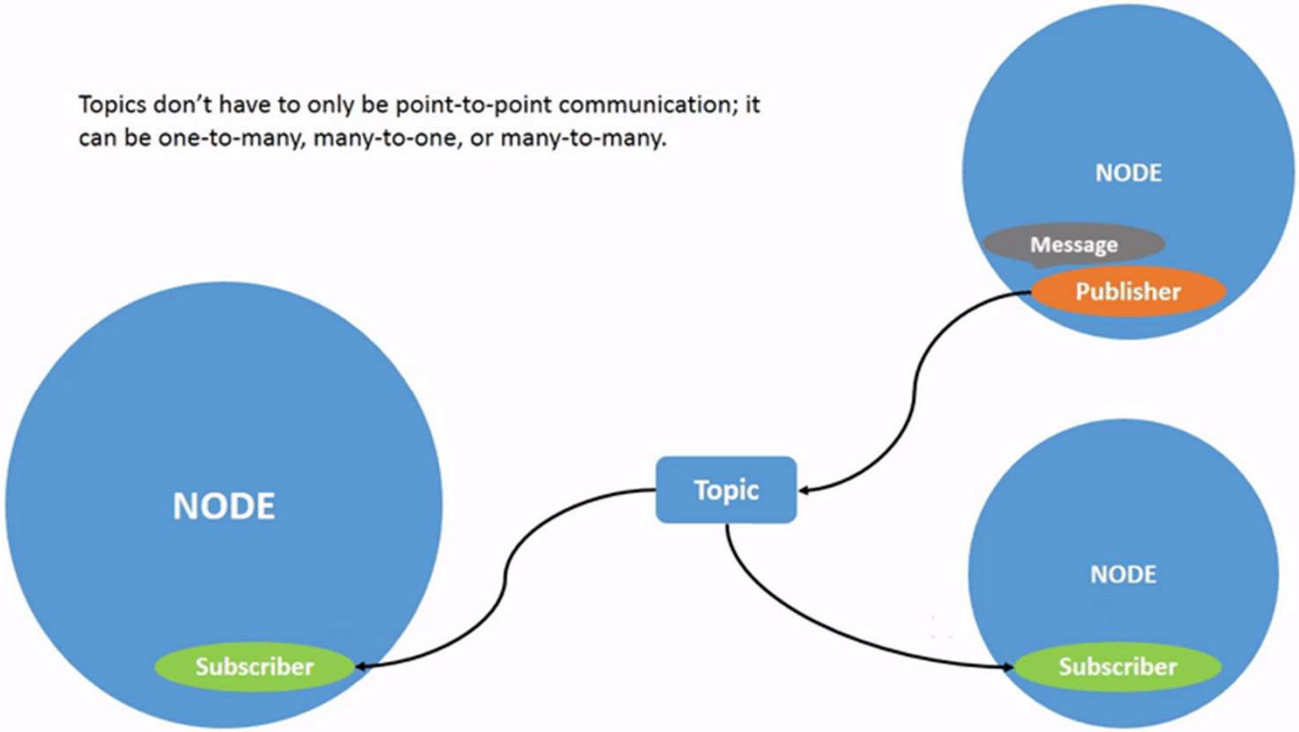
ROS Topics – Publish/Subscribe

- A publish/subscribe system consists of data *producers* (publishers) and *consumers* (subscribers).
- Publishers and subscribers connect through named channels, i.e. "***topics***"
- Multiple publishers and subscribers can exist for a single topic
- When data is **published** to a topic, ***all*** **subscribers** receive it.



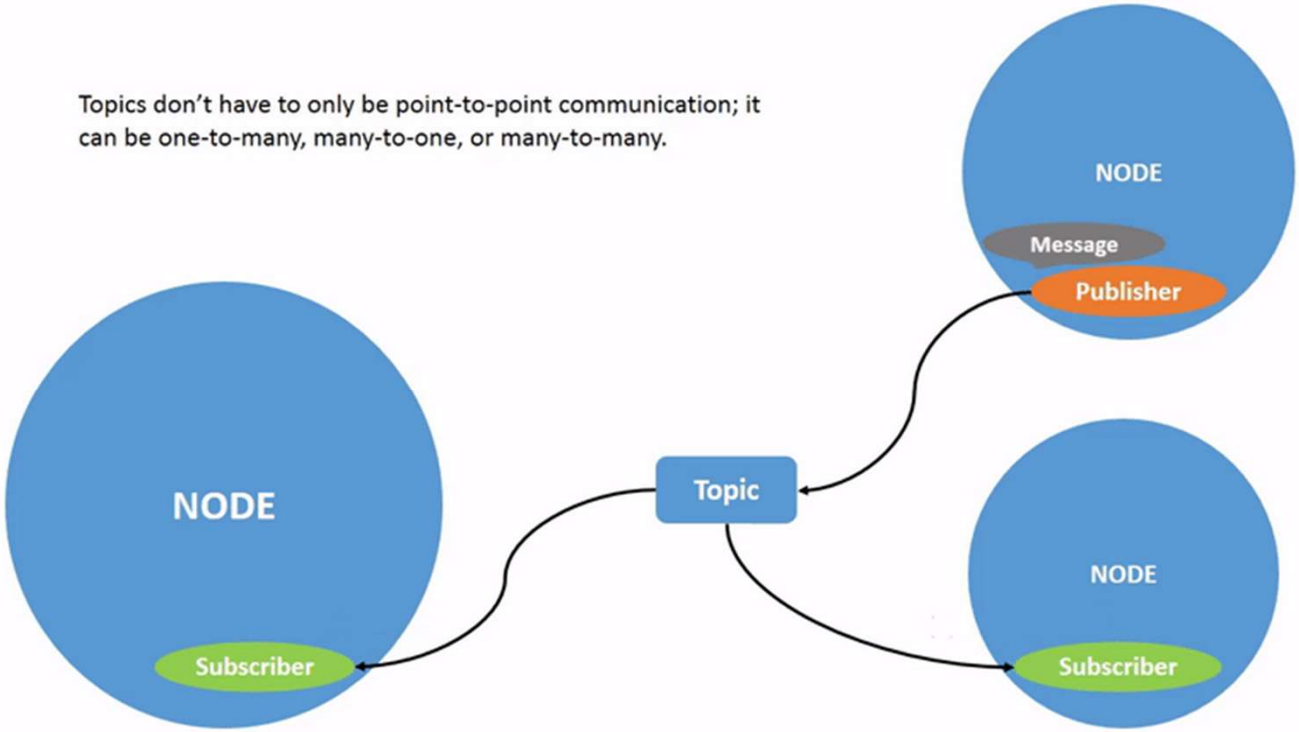
ROS Topics – Publish/Subscribe

- This system is also referred to as a "bus," similar to electrical engineering concepts.
- The bus-like structure contributes to ROS2's power and flexibility.
- Publishers and subscribers can be added or removed dynamically.
- This dynamic nature facilitates debugging and system introspection.
- Data recording with `ros2 bag`



ROS Topics – Anonymity

- When a subscriber gets a piece of data (i.e. a message), it doesn't generally know or care which publisher originally sent it (though it can find out if it wants).
- The benefit to this architecture is that publishers and subscribers can be swapped out at will without affecting the rest of the system.



ROS Topics – Strongly typed

- The types of each field in a ROS message are typed, and that type is enforced at various levels.

- The semantics of each field are well-defined. There is *no automated* mechanism to ensure this, but all of the core ROS types have strong semantics associated with them. e.g *IMU* message contains a 3D vector for the measured angular velocity, and each of the dimensions is specified to be in rad/s.

```
uint32 field
string field2
```

- The code will ensure that `field1` is always an unsigned integer and that `field2` is always a string.

ROS Topics

List available topics

```
> ros2 topic list -t
```

Subscribe and print the content of a topic:

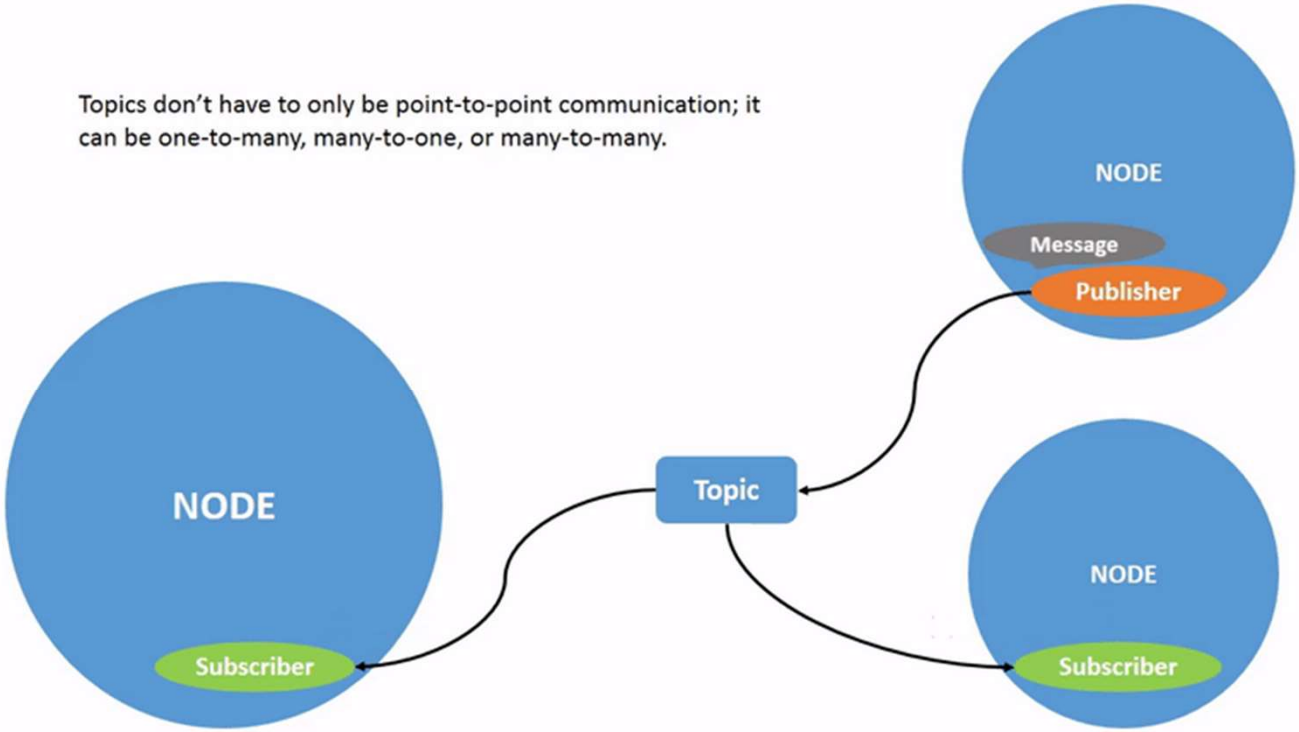
```
> ros2 topic echo /topic
```

Retrieve information about a topic

```
> ros2 topic info /topic
```

Publish a message to a topic

```
> ros2 topic pub /topic type data
```

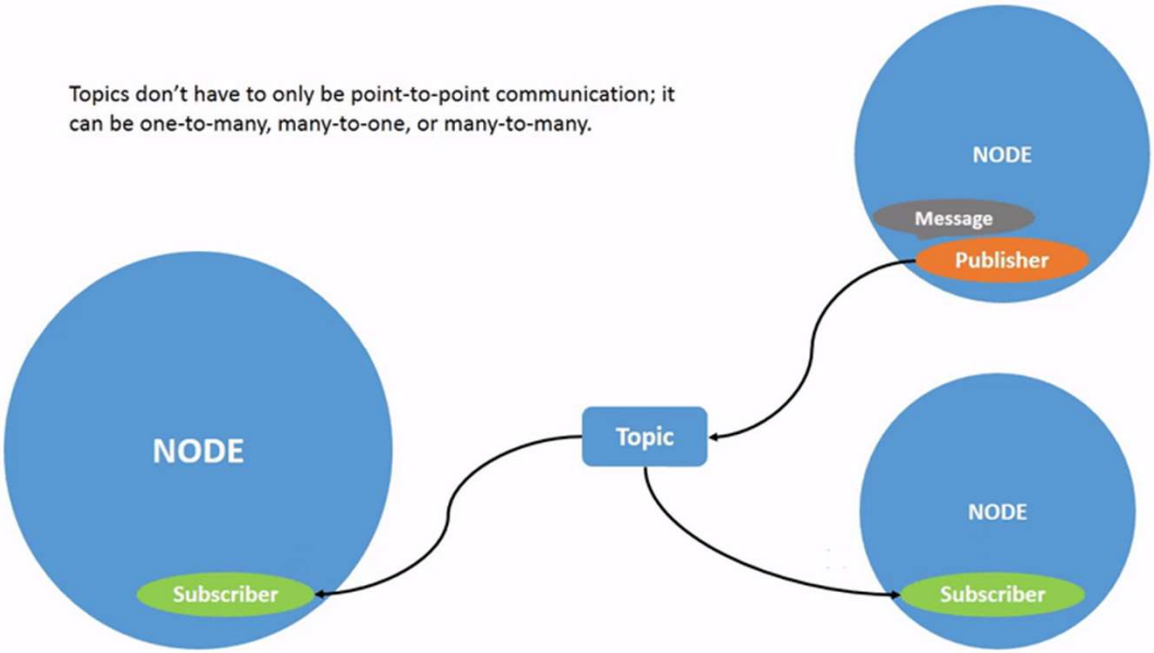


ROS Messages

- Data structure defining the *type* of a topic
- Comprised of a nested structure of basic types(floats, bool, double, etc.), messages and arrays of them.
- Defined in **.msg* files used to generate source code in different languages.

Print message definition

```
> ros2 interface show message
```



ROS Messages


```
> ros2 interface show geometry_msgs/msg/Twist
```

```
# This expresses velocity in free space broken into its linear and angular parts.

Vector3  linear
  float64 x
  float64 y
  float64 z
Vector3  angular
  float64 x
  float64 y
  float64 z
```

ROS Messages

geometry_msgs/Point.msg

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

sensor_msgs/Image.msg

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

geometry_msgs/PoseStamped.msg

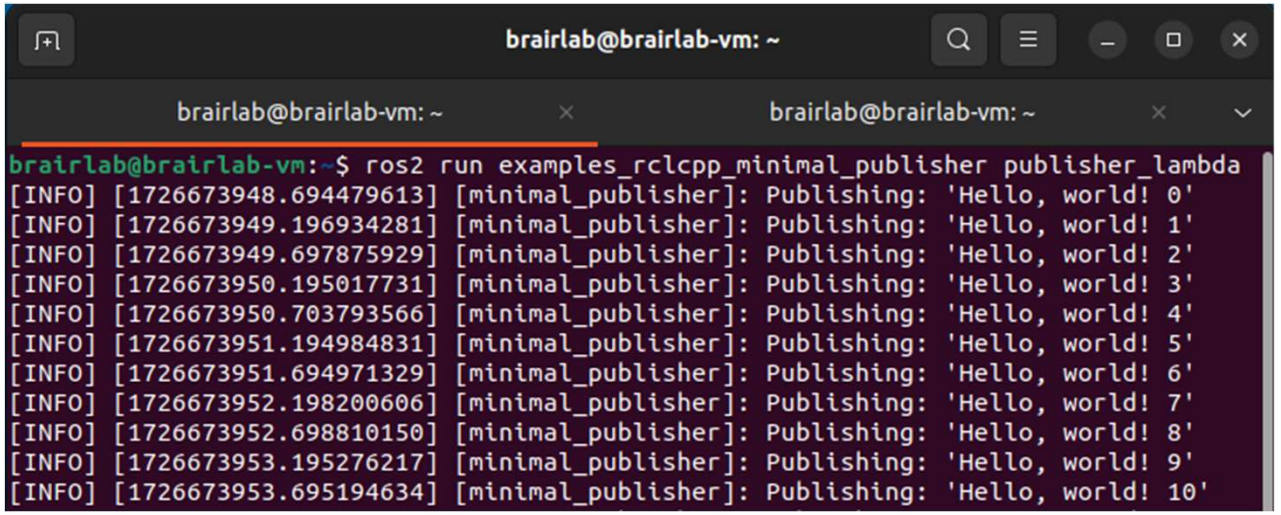
```
std_msgs/Header header
 uint32 seq
 time stamp
 string frame_id
geometry_msgs/Pose pose
  geometry_msgs/Point position
    float64 x
    float64 y
    float64 z
  geometry_msgs/Quaternion orientation
    float64 x
    float64 y
    float64 z
    float64 w
```

Example - Starting a *talker* node

Run a talker demo node

```
> ros2 run examples_rclcpp_minimal_publisher publisher_lambda
```

PACKAGE NAME



Example - Talker node

List of active nodes

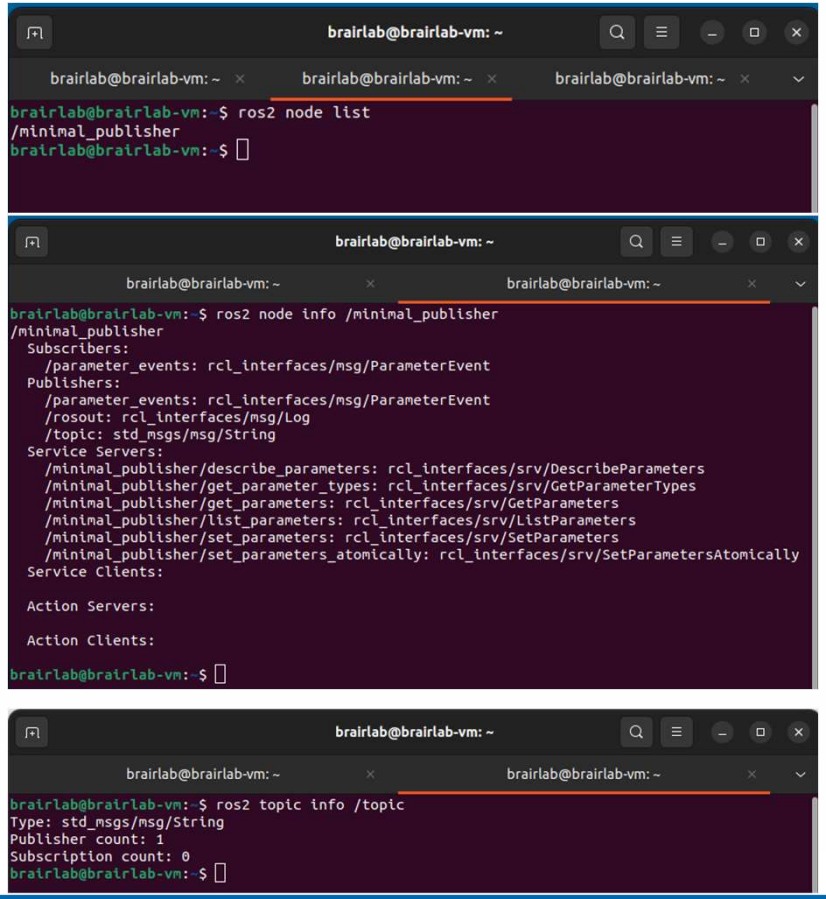
```
> ros2 node list
```

Information about the *talker* node

```
> ros2 node info /minimal_publisher
```

Information about the *chatter* topic

```
> ros2 topic info /topic
```



Example – The «*Topic*» topic

Check the type of *topic*

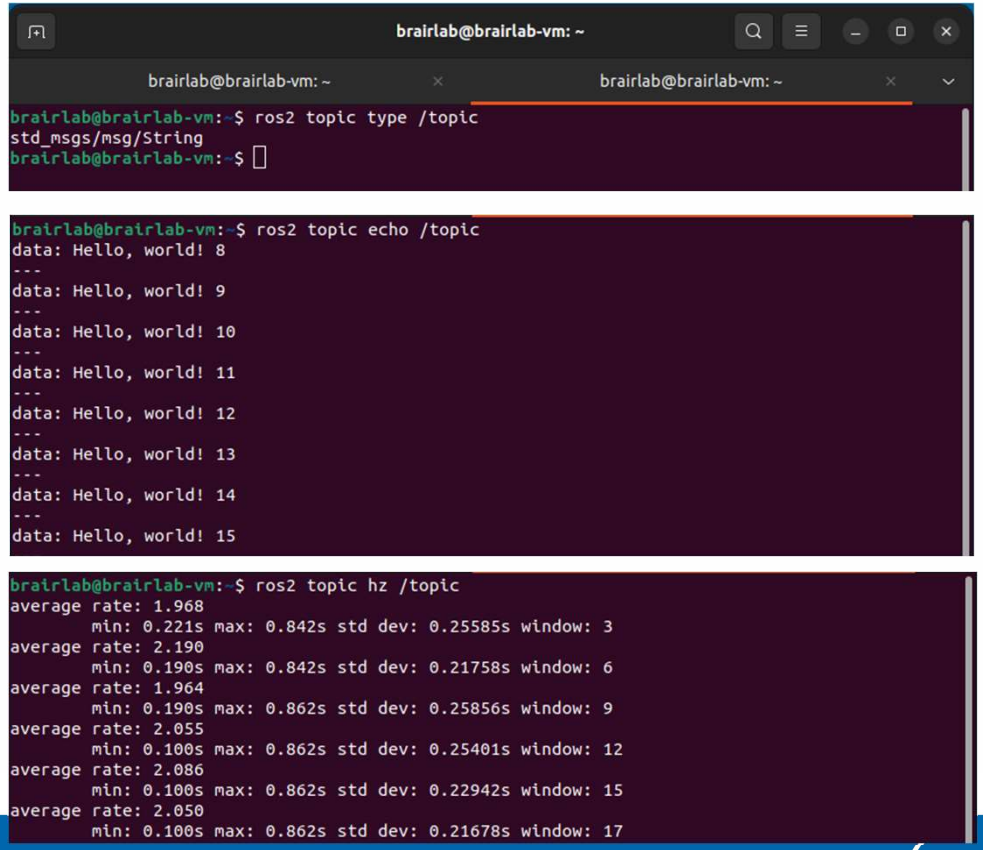
```
> ros2 topic type /topic
```

Show the message contents of the topic

```
> ros2 topic echo /topic
```

Analyze the frequency

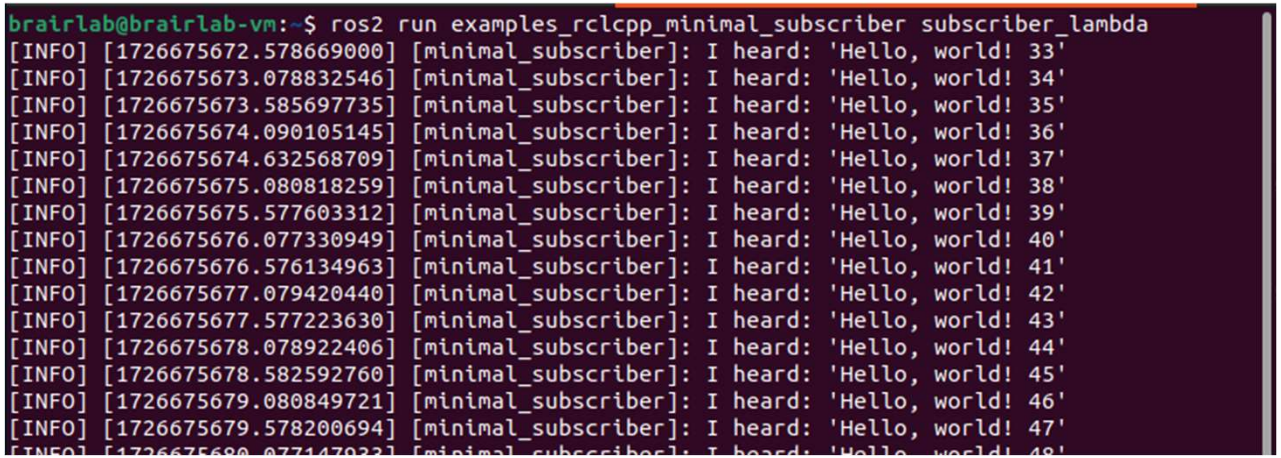
```
> ros2 topic hz /topic
```



Example – *Listener* node

listener demo node

```
> ros2 run examples_rclcpp_minimal_subscriber subscriber_lambda
```



Example - Analyze nodes and topic

See the new *listener* node

```
> ros2 node list
```

```
brairlab@brairlab-vm:~$ ros2 node list
/minimal_publisher
/minimal_subscriber
brairlab@brairlab-vm:~$
```

Show the connection of the nodes over the chatter topic

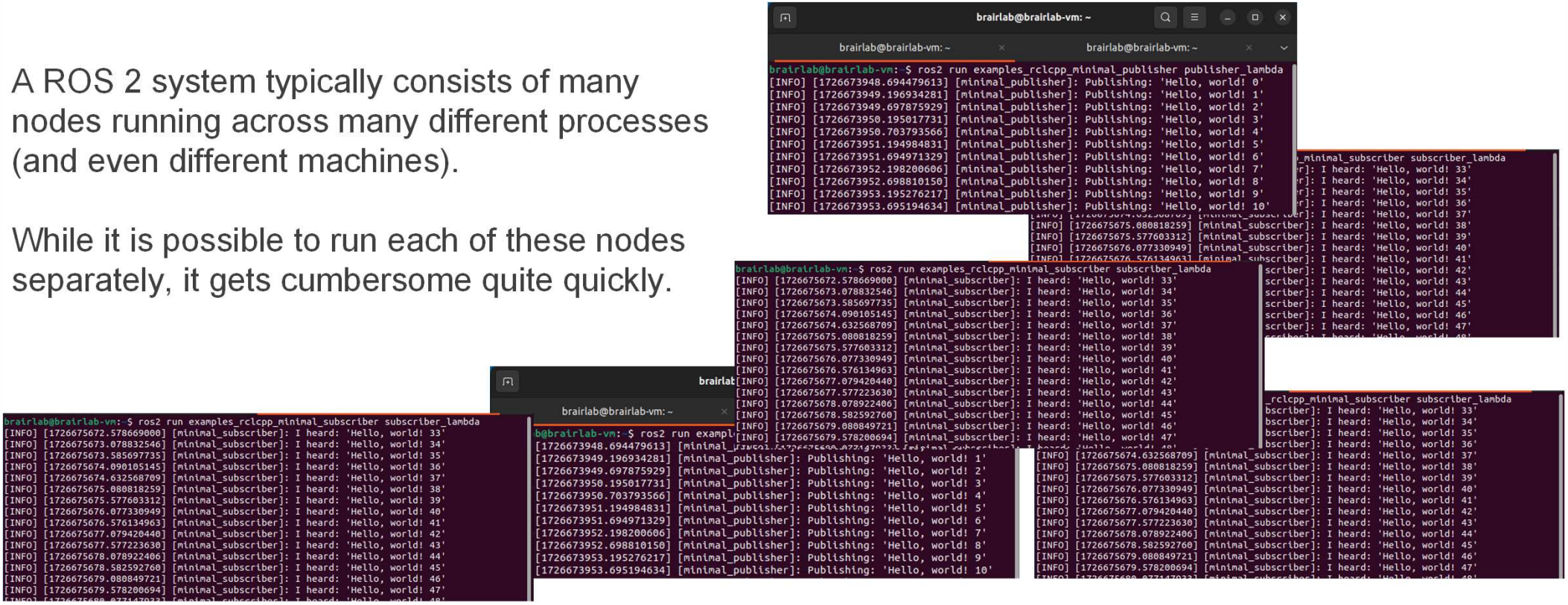
```
> ros2 topic info /topic
```

```
brairlab@brairlab-vm:~$ ros2 topic info /topic
Type: std_msgs/msg/String
Publisher count: 1
Subscription count: 1
```

ROS Launch

A ROS 2 system typically consists of many nodes running across many different processes (and even different machines).

While it is possible to run each of these nodes separately, it gets cumbersome quite quickly.



ROS Launch

- *launch* is a tool for launching multiple nodes (as well as setting parameters)
- Written in Python/XML/YAML

Browse to the folder and start a launch file with

```
> ros2 launch file_name.launch
```

Start a launch file from a package with

```
> ros2 launch package_name file_name.launch
```

```
brairlab@brairlab-vm:~$ ros2 launch topics_example.launch
[INFO] [launch]: All log files can be found below /home/brairlab/.ros/log/2024-09-20-16-48-14-667312-brairla
b-vm-10628
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [subscriber_lambda-1]: process started with pid [10629]
[INFO] [publisher_lambda-2]: process started with pid [10631]
[publisher_lambda-2] [INFO] [1726843695.707269462] [talker]: Publishing: 'Hello, world! 0'
[subscriber_lambda-1] [INFO] [1726843695.714224147] [listener]: I heard: 'Hello, world! 0'
[publisher_lambda-2] [INFO] [1726836454.673239062] [talker]: Publishing: 'Hello, world! 1'
[subscriber_lambda-1] [INFO] [1726836454.674846258] [listener]: I heard: 'Hello, world! 1'
[publisher_lambda-2] [INFO] [1726843696.320115686] [talker]: Publishing: 'Hello, world! 2'
[subscriber_lambda-1] [INFO] [1726843696.320714717] [listener]: I heard: 'Hello, world! 2'
[publisher_lambda-2] [INFO] [1726843696.818288790] [talker]: Publishing: 'Hello, world! 3'
[subscriber_lambda-1] [INFO] [1726843696.818738957] [listener]: I heard: 'Hello, world! 3'
[publisher_lambda-2] [INFO] [1726843697.318523509] [talker]: Publishing: 'Hello, world! 4'
[subscriber_lambda-1] [INFO] [1726843697.319112578] [listener]: I heard: 'Hello, world! 4'
[publisher_lambda-2] [INFO] [1726843697.818191315] [talker]: Publishing: 'Hello, world! 5'
[subscriber_lambda-1] [INFO] [1726843697.818714375] [listener]: I heard: 'Hello, world! 5'
```

ROS Launch File Structure

talker_listener.launch

```
<launch>
    <node name="listener" pkg="examples_rclcpp_minimal_subscriber" exec="subscriber_lambda" output="screen"/>
    <node name="talker" pkg="examples_rclcpp_minimal_publisher" exec="publisher_lambda" output="screen"/>
</launch>
```

- **`launch`**: Root element of the launch file
- **`node`**: Each *<node>* tag is a node to be launched
- **`name`**: Node name
- **`pkg`**: Package containing the node
- **`exec`**: the executable (with the same name)
- **`output`**: Specifies where to output log messages (`screen`: console, `log`: log file)

More info

http://wiki.ros.org/roslaunch/XML

https://docs.ros.org/en/humble/How-To-Guides/Migrating-from-ROS1/Migrating-Launch-Files.html

http://wiki.ros.org/roslaunch/Tutorials/Roslaunch%20tips%20for%20larger%20projects

ROS Launch – Python and YAML

ROS

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='examples_rclcpp_minimal_subscriber',
            executable='subscriber_lambda',
            name='listener',
            output='screen'
        ),
        Node(
            package='examples_rclcpp_minimal_publisher',
            executable='publisher_lambda',
            name='talker',
            output='screen'
        )
    ])
```

```yaml
launch:
  - node:
      pkg: "examples_rclcpp_minimal_subscriber"
      exec: "subscriber_lambda"
      name: "listener"
      output: "screen"

  - node:
      pkg: "examples_rclcpp_minimal_publisher"
      exec: "publisher_lambda"
      name: "talker"
      output: "screen"
```

ROS Launch Arguments

- Create re-usable launch files with <arg> tag, which works like a parameter (default optional)

```
<arg name="arg_name" default="default_value"/>
```

- Use arguments in launch file with

```
$(arg arg_name)
```

- Arguments can be set with

```
> ros2 launch launch_file.launch arg_name:=value
```

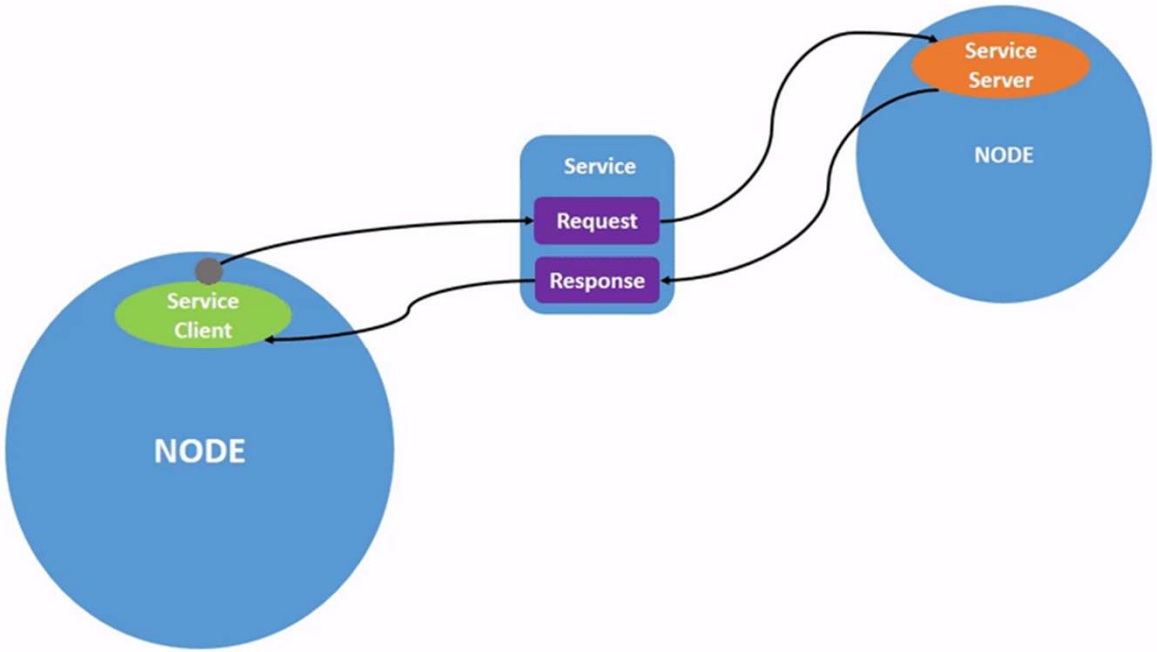
range_world.launch (simplified)

```
<?xml version="1.0"?>
<launch>
  <arg name="use_sim_time" default="true"/>
  <arg name="world" default="gazebo_ros_range"/>
  <arg name="debug" default="false"/>
  <arg name="physics" default="ode"/>

  <group if="$(arg use_sim_time)">
    <param name="/use_sim_time" value="true" />
  </group>

  <include file="$(find gazebo_ros)
                                /launch/empty_world.launch">
    <let name="world_name" value="$(find gazebo_plugins)/
                    test/test_worlds/$(arg world).world"/>
    <let name="debug" value="$(arg debug)"/>
    <let name="physics" value="$(arg physics)"/>
  </include>
</launch>
```

More info
http://wiki.ros.org/roslaunch/XML/arg

ROS Services

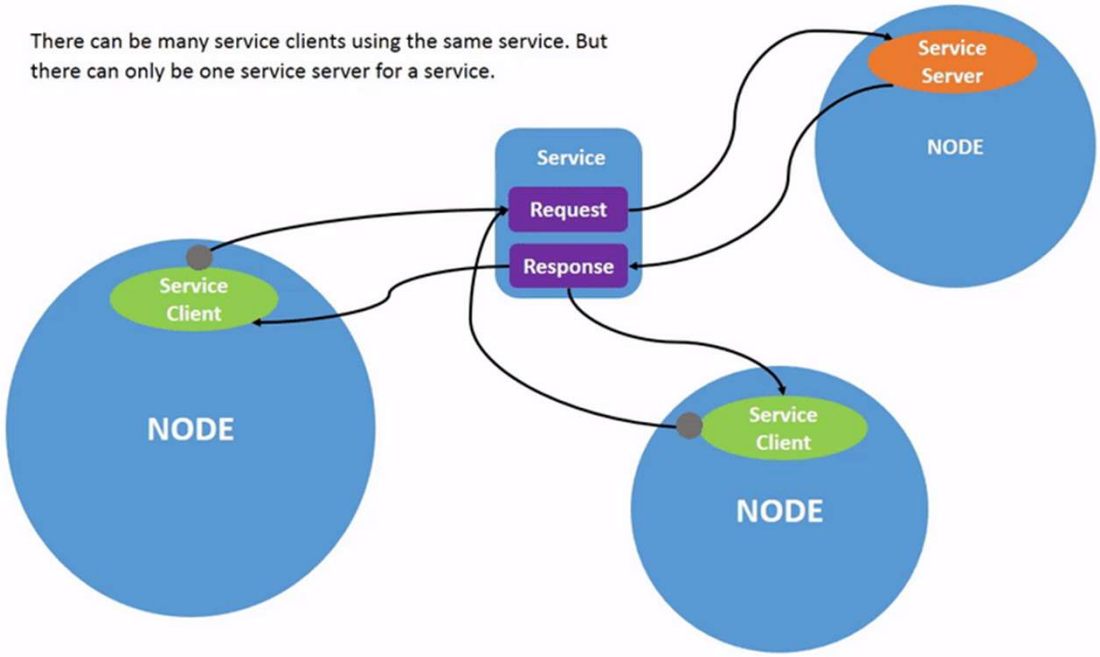
- Another method of communication for nodes in the ROS graph.
- **Call-and-response** model
  - versus topics **publisher-subscriber**
- While *topics* allow nodes to subscribe to data streams and get continual updates, services *only* provide data *when* they are specifically called by a client.



ROS Services

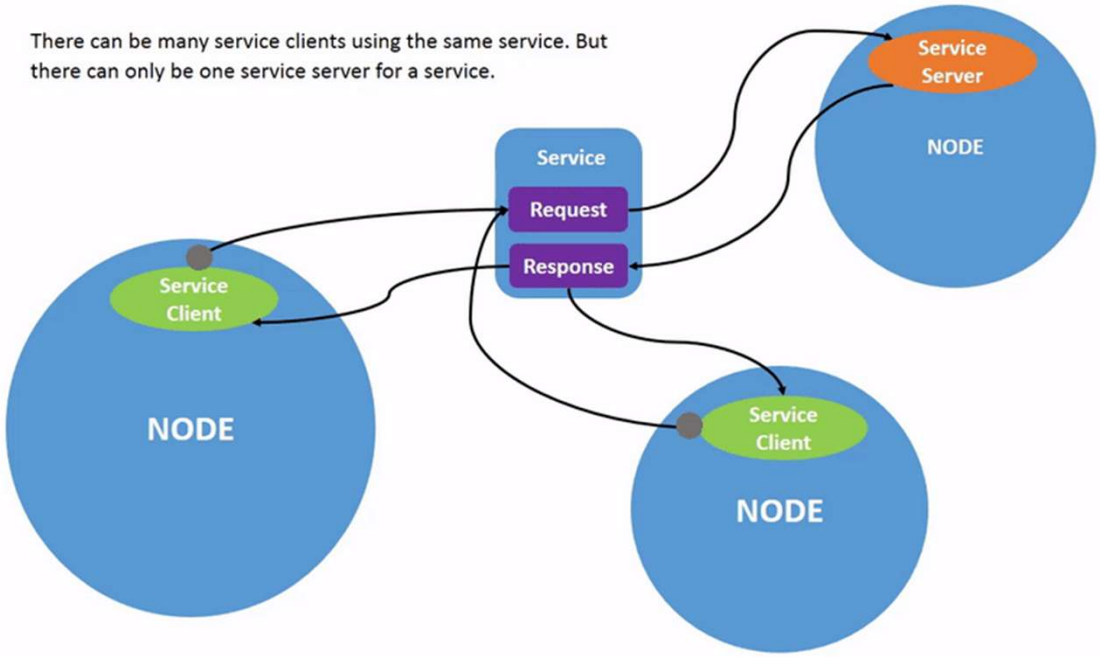
- This *communication pattern* is also known as **Remote Procedure Call** (RPC), in fact one ore more nodes can make a remote procedure call to another node which will do a computation and return a result.

- Services are identified by a (*unique*) service name, which looks much like a topic name (but is in a different namespace).



ROS Services

- Are expected to return **quickly**, as the client is *generally* waiting on the result.
- Services **should never be used for longer running processes**
- If you have a service that will be doing a long-running computation, consider using an **action** instead.



More info

https://docs.ros.org/en/humble/How-To-Guides/Sync-Vs-Async.html

uint32 a uint32 b --- uint32

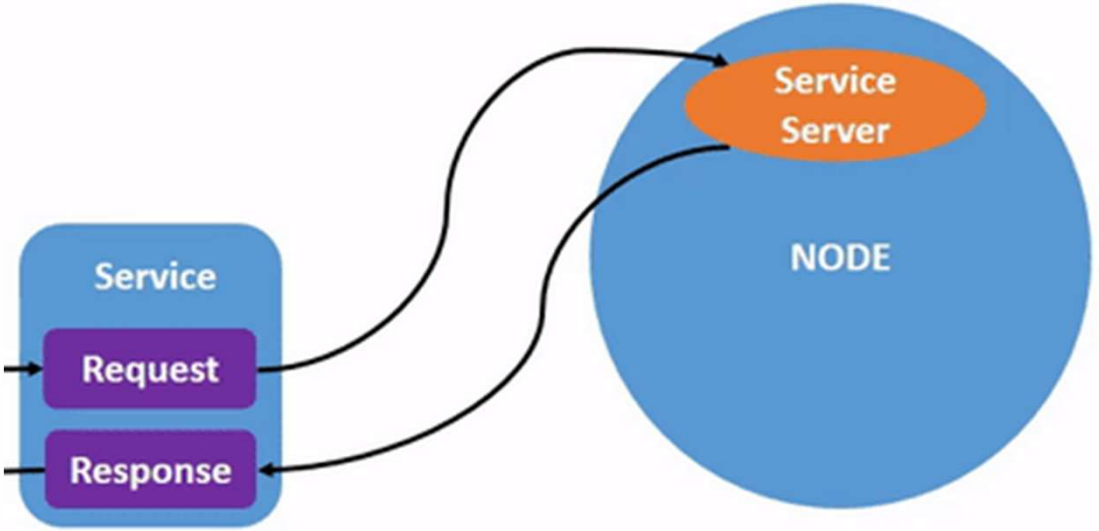
ROS Services – Server

- A service **server** is the entity that will accept a remote procedure request, and perform some computation on it.

- The interface of the server is defined in a `.srv` file

```
uint32 a
uint32 b
---
uint32 sum
```

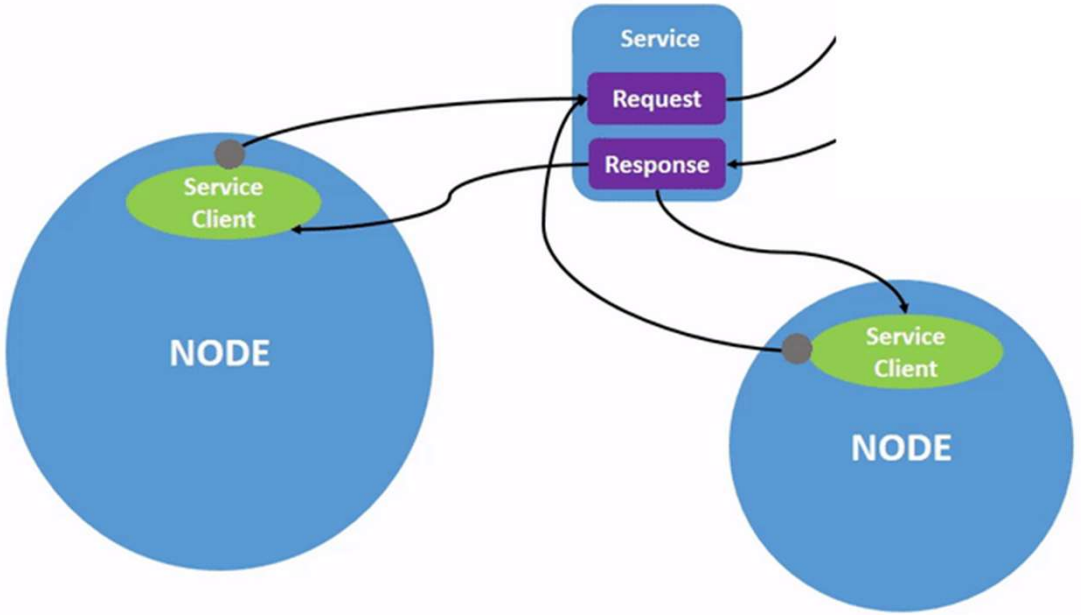
- E.g. defines a service that accepts 2-field *requests* (named `a` and `b`) and return a *response* having a `sum` field



uint32 a uint32 b --- uint32

ROS Services – Client

- A service *client* is an entity that will **request** a **remote service server** to **perform** a **computation** on its behalf.
- The service client is the entity that creates the initial message containing a and b, and *waits* for the service server to compute the `sum` and return the result.
- Unlike the service server, there can be *arbitrary* numbers of service clients using the *same* service name.



ROS Services - Example

std_srvs/Trigger.srv

```
---
bool success
string message
```

Request

Response

nav_msgs/GetPlan.srv

```
geometry_msgs/PoseStamped start
geometry_msgs/PoseStamped goal
float32 tolerance
---
nav_msgs/Path plan
```

More info

https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html

ROS Services - Example
Analyze and call a service

Start `add_two_ints_server`

```
>  ros2 run examples_rclcpp_minimal_service service_main
```

Available services

```
> ros2 service list
```

See the type of the service

```
> ros2 service type /add_two_ints
```

Show the service definition

```
> ros2 interface show
  example_interfaces/srv/AddTwoInts
```

Call the service

```
> ros2 service call /add_two_ints
  example_interfaces/srv/AddTwoInts
  "{a: 10, b: 20}"
```

```
brairlab@brairlab-vm:~$ ros2 service list
/add_two_ints
/minimal_service/describe_parameters
/minimal_service/get_parameter_types
/minimal_service/get_parameters
/minimal_service/list_parameters
/minimal_service/set_parameters
/minimal_service/set_parameters_atomically
brairlab@brairlab-vm:~$
```

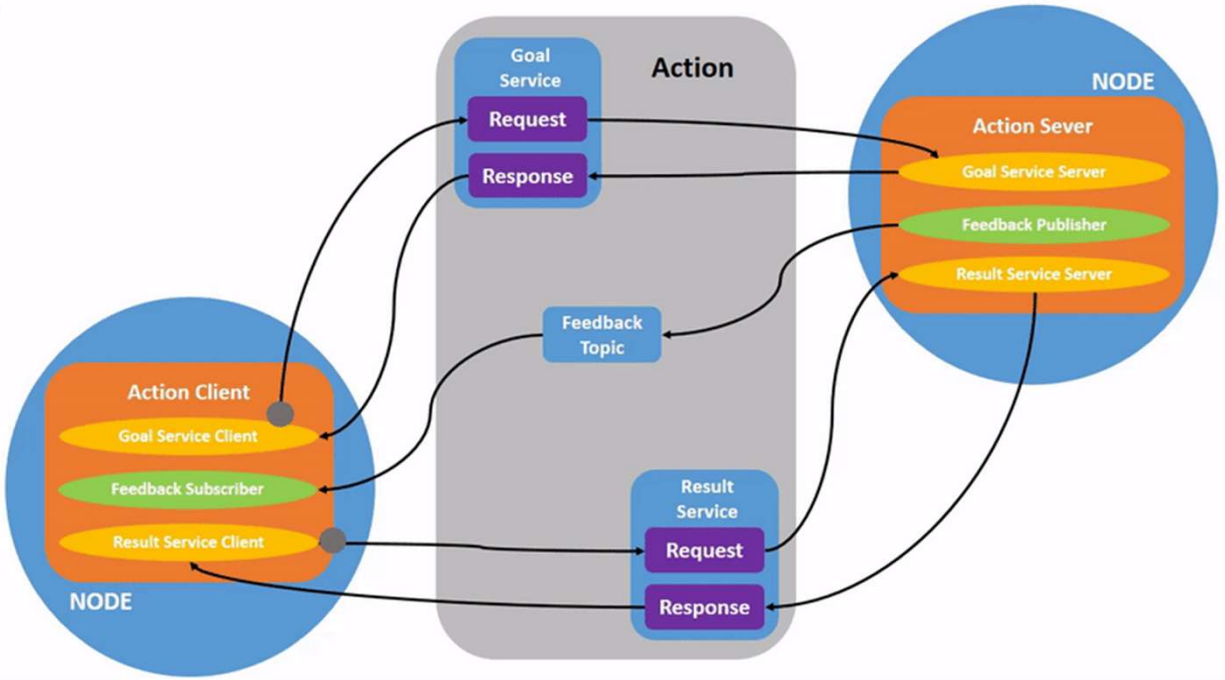
```
brairlab@brairlab-vm:~$ ros2 service type /add_two_ints
example_interfaces/srv/AddTwoInts
brairlab@brairlab-vm:~$
```

```
brairlab@brairlab-vm:~$ ros2 interface show example_interfaces/srv/AddTwoInts
int64 a
int64 b
---
int64 sum
brairlab@brairlab-vm:~$
```

```
brairlab@brairlab-vm:~$ ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts "{a: 10, b: 20}"
requester: making request: example_interfaces.srv.AddTwoInts_Request(a=10, b=20)

response:
example_interfaces.srv.AddTwoInts_Response(sum=30)

brairlab@brairlab-vm:~$
```

ROS Actions

- Actions are one of the communication types in ROS 2 and are intended for long running tasks. They consist of three parts: a goal, feedback, and a result.
- Actions are built on topics and services. Their functionality is similar to services, except actions can be canceled. They also provide steady feedback, as opposed to services which return a single response.
- Actions use a client-server model, similar to the publisher-subscriber model. **Asynchronous bidirectional communication.**
- An "**action client**" node sends a goal to an "**action server**" node that acknowledges the goal and returns a stream of feedback and a result.



More info

https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html

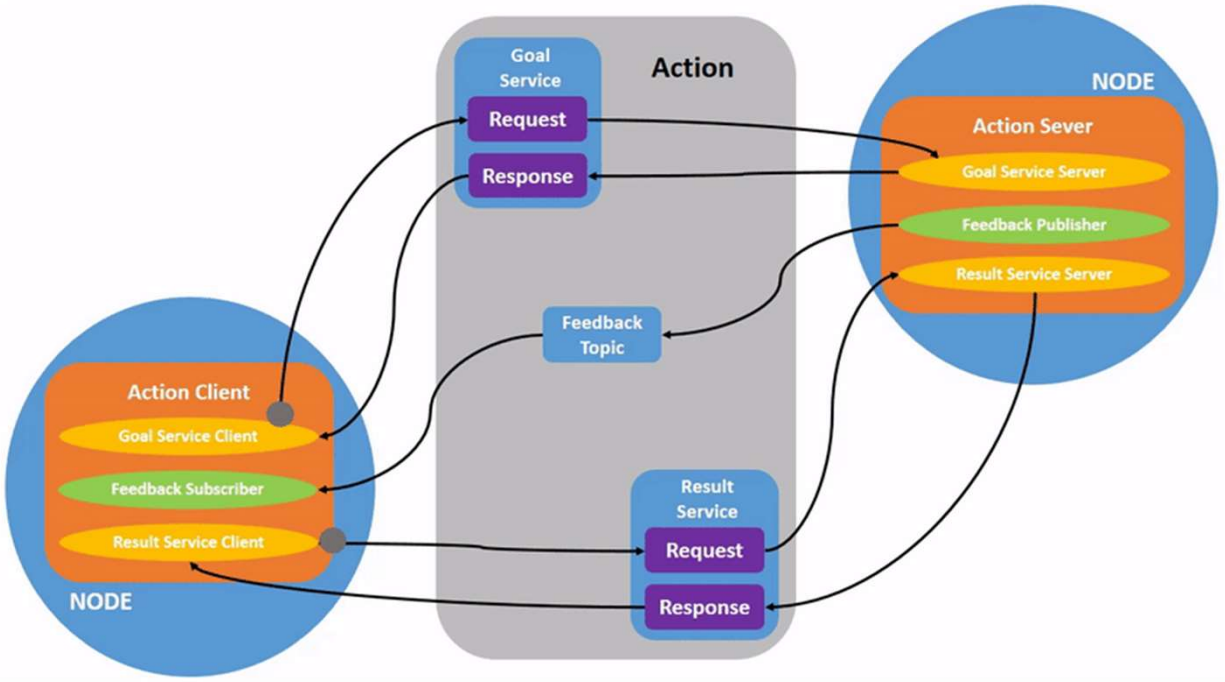
ROS Actions

This structure is reflected in how an action message definition looks (`.action` file):

```
int32 request
---
int32 response
---
int32 feedback
```

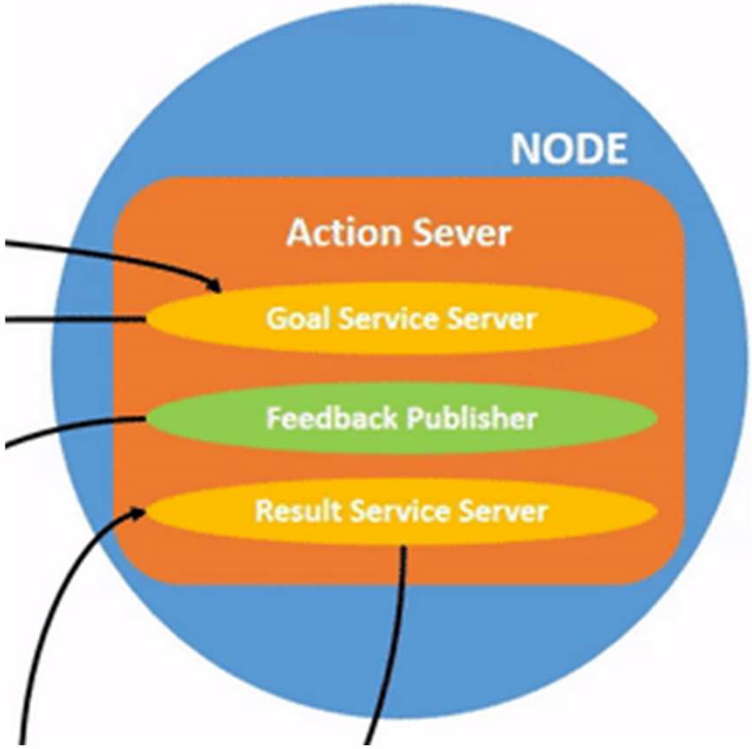
In ROS 2, **actions** are expected to be **long running procedures**, as there is overhead in setting up and monitoring the connection. If you need a *short running remote* procedure call, consider using a *service* instead.

Actions are *identified* by an **action name**, which looks much like a topic.



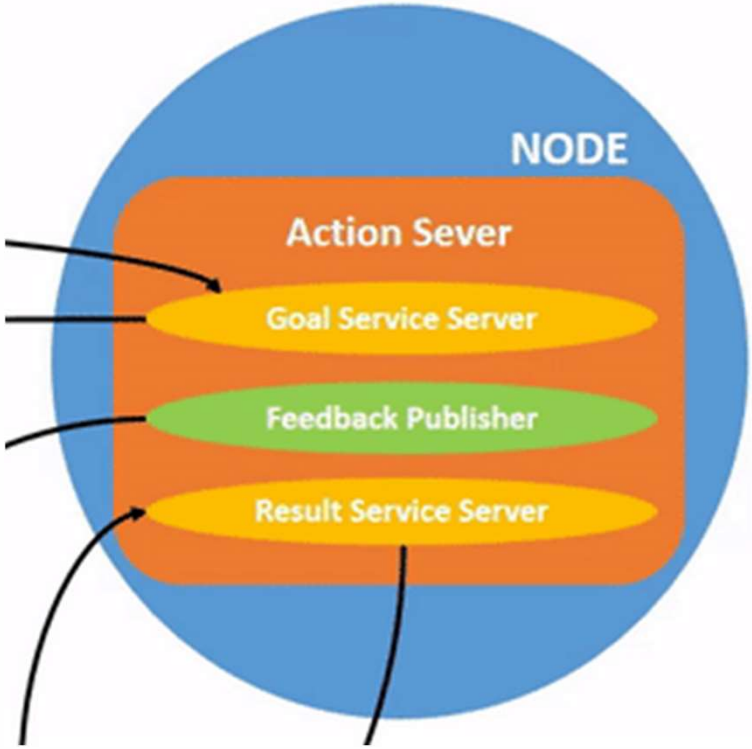
ROS Action - Server

- The action server is the entity that will accept the remote procedure request and perform some procedure on it.
- It is also responsible for sending out feedback as the action progresses and should react to cancellation/preemption requests.
- NOTE: There should only ever be one action server per action name.



ROS Action - Client

- An action client is an entity that will request a remote action server to perform a procedure on its behalf.
- The action client is the entity that creates the initial message containing the order, and waits for the action server to compute the sequence and return it (with feedback along the way).
- Unlike the action server, there can be arbitrary numbers of action clients using the same action name.



ROS communication methods - review

Type	Features	Description
Topic	Asynchronous, Unidirectional	Loosely coupled continuous data exchange
Service	(A)synchronous, Bi-directional	Short-running remote call, request/response
Action	Asynchronous, Bi-directional	Long-running preemptable request with intermediate feedback

ROS Bag

- The data from the ROS messages can be recorded.
- The file format used is called **bag**, and `*.bag` is used as the file extension.

- **bag** can be used to record messages and play them back when necessary to reproduce the environment when messages are recorded.

- For example, when performing a robot experiment using a sensor, sensor values are stored in the message form using the bag.

- This recorded message can be repeatedly loaded without performing the same test by playing the saved bag file.

- Record and play functions of rosbag are especially useful when developing an algorithm with frequent program modifications.

ROS Bag

- A *bag* is used to store message data
- Binary format with file extension `*.bag`
- Suited for logging and recording datasets for visualization and analysis

Record all topics in the bag

```
> ros2 bag record --all
```

Record from given topics

```
> ros2 bag record topic_1 topic_2 topic_3
```

Stop bag recording with `Ctrl + C`
Bags are saved with start date and time as file name in the current folder (e.g. `2024-04-23-10-27-13.bag`)

Information about a bag

```
> ros2 bag info bag_name.bag
```

Playback options can be defined as:

```
> ros2 bag play --rate=0.5 bag_name.bag
```

define the rate