Introduction to ROS2 Basics (ROS2 Intro, Packages, Nodes, Communication)

Introduction to Robotics Lab

Konstantinos Asimakopoulos *k_asimakopoulos@ac.upatras.gr*
Lampros Printzios *printzios_lampros@ac.upatras.gr*

Department of Electrical and Computer Engineering
University of Patras

October 14, 2025

Table of Contents

Introduction and First Steps - What is ROS?

1. **ROS (Robot Operating System)** is an open-source, flexible **development** framework, **visualization** software, and **packaging** system, that simplifies the development of complex and distributed robotic systems.
2. ROS is not a traditional operating system; rather, it acts as **middleware** that enables communication between hardware (motors, sensors, cameras), algorithms, and applications, across multiple platforms and programming languages (e.g., Python, C++).
3. ROS is released in distributions (**distros**), like the ones below:
   - ROS 1 **Noetic** → Ubuntu 20.04
   - ROS 2 **Foxy**, **Galactic** → Ubuntu 20.04
   - ROS 2 **Humble** → Ubuntu 22.04
   - ROS 2 **Jazzy** → Ubuntu 24.04

We will use ROS2 only, and mainly Python, for the code we run in the lab.

Introduction and First Steps - Virtual Environment & bashrc file

It is a good practice to work on our ROS2 projects inside a **Virtual Environment** (venv), to keep ROS and Python dependencies isolated and avoid conflicts. There are two common ways to do this:

- **Conda**: `conda create -n <env_name> python=<py_version> && conda activate <env_name>`
- **Python Venv**: `python3 -m venv <env_name> && source ~/<env_name>/bin/activate`

Also, to simplify our ROS2 setup process, we can write at the end of our bashrc file, using **`nano ~/.bashrc`** (and assuming the ROS2 <distro> installation is already done and we work in <ros2_ws> folder):

```
alias ros_2="source ~/<env_name>/bin/activate
&& source /opt/ros/<distro>/setup.bash"
```

Now we can just type our alias command every time we open a terminal!

Introduction and First Steps - Installation Instructions

You need to always check your Ubuntu version to install the proper ROS2 <distro>. Type the following commands in your terminal:

```
source ~/<env_name>/bin/activate
sudo apt update && sudo apt upgrade -y
sudo apt install -y software-properties-common curl terminator git
sudo add-apt-repository universe
export ROS_APT_SOURCE_VERSION=$(curl -s https://api.github.com/repos/ros-
    infrastructure/ros-apt-source/releases/latest | grep -F "tag_name" | awk -F
    \" '{print $4}')
curl -L -o /tmp/ros2-apt-source.deb "https://github.com/ros-infrastructure/ros-
    apt-source/releases/download/${ROS_APT_SOURCE_VERSION}/ros2-apt-source_${
    ROS_APT_SOURCE_VERSION}.$(. /etc/os-release && echo $VERSION_CODENAME)_all.
    deb" # If using Ubuntu derivatives use $UBUNTU_CODENAME
sudo dpkg -i /tmp/ros2-apt-source.deb
sudo apt update && sudo apt upgrade -y
sudo apt install -y ros-<distro>-desktop ros-dev-tools
```

You should setup ROS2 with **`source /opt/ros/<distro>/setup.bash`**. If you configure the bashrc file as shown previously, you don't have to type this command in every terminal, just the defined alias.

Packages - What is a Package?

1. A **Package** is the basic **unit of organization** in ROS2.
2. It groups together everything needed to share, build, and run a piece of functionality, making the software modular, reusable, and easy to maintain.
3. It usually contains:
   - **Source code** → executable nodes, libraries, or interface definitions (`msg`, `srv`, `action`)
   - **Configuration files** → parameters, launch files, and resource files
   - **Build system files** → metadata for building, packaging, installing, and resolving dependencies (e.g. `package.xml`, `setup.py`, `setup.cfg`, `CMakeLists.txt`)

ROS2 Package Functionalities

```
ros2 pkg -h
```

Packages - Basic Package Structures

```
py_pkg_using_messages
├── package.xml
├── py_pkg_using_messages
│   ├── amazing_quote_publisher_node.py
│   ├── amazing_quote_subscriber_node.py
│   └── __init__.py
├── resource
│   └── py_pkg_using_messages
├── setup.cfg
├── setup.py
└── test
    ├── test_copyright.py
    ├── test_flake8.py
    └── test_pep257.py
```

Package with Publisher and Subscriber Nodes

```
py_pkg_library/
├── package.xml
├── py_pkg_library
│   ├── __init__.py
│   └── sample_py_library
│       ├── __init__.py
│       ├── _sample_class.py
│       └── _sample_function.py
├── resource
│   └── py_pkg_library
├── setup.cfg
├── setup.py
└── test
    ├── test_copyright.py
    ├── test_flake8.py
    └── test_pep257.py
```

Library Package

```
pkg_with_interfaces/
├── CMakeLists.txt
├── include
│   └── pkg_with_interfaces
├── msg
│   ├── AmazingQuote.msg
│   └── AmazingQuoteStamped.msg
├── package.xml
├── src
└── srv
    └── AddPoints.srv
```

Interfaces Package

Each Python package directory contains an inner folder that defines the main *Python Module* (with the same name) where the source code is located, and some other build and packaging metadata, like `package.xml`, `setup.py` and `setup.cfg`. For the Interfaces Package, we are mainly interested in `msg` and `srv` (optionally `action`) folders, while the build metadata are `package.xml` and `CMakeLists.txt`.

Packages - Creating & building a Package

Initially, we create a source folder called `src/` for our `<ros2_ws>` workspace and we enter the source folder by typing a command like this:

```
mkdir ~/<ros2_ws>/src && cd ~/<ros2_ws>/src
```

To create a package we use a command like this (we can also use `ament_cmake` for C++, instead of `ament_python`):

```
ros2 pkg create <pkg_name> --build-type ament_python
--dependencies rclpy <other_pkg>
```

We can build the package by entering the `<ros2_ws>` folder and using the command **`colcon build`**. The folders **`build/`**, **`install/`** and **`log/`** are created. After building, always type **`source install/setup.bash`**, to make the changes visible to your current terminal.

ROS2 Package Creation Functionalities

```
ros2 pkg create -h
```

Packages - Creating & building a Package package.xml

The **package.xml** file exists in both ament_python and ament_cmake packages. Below you can see the two versions, where we have marked with red color some lines you may need to add before building the package (mainly dependencies).

```xml
<?xml version="1.0"?> <?xml-model
href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>py_pkg_using_messages</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="TODO">lampros</maintainer>
  <license>TODO: License declaration</license>

  <depend>rclpy</depend>
  <depend>pkg_with_interfaces/</depend>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

ament_python package.xml

```xml
<?xml version="1.0"?> <?xml-model
href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>pkg_with_interfaces</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="TODO">lampros</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>geometry_msgs</depend>

  <buildtool_depend>rosidl_default_generators</buildtool_depend>
  <exec_depend>rosidl_default_runtime</exec_depend>
  <member_of_group>rosidl_interface_packages</member_of_group>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

ament_cmake package.xml

Packages - Creating & building a Package setup.py

The **setup.py** file exists in ament_python packages, but not in ament_cmake ones (similarly with setup.cfg file).

```python
from setuptools import setup

package_name = 'py_pkg_node'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='lampros',
    maintainer_email='[redacted]',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'sample_py_node = py_pkg_node.sample_py_node:main',
            'print_forever_node = py_pkg_node.print_forever_node:main'
        ],
    },
)
```

simple setup.py file

Important!

We should always add the executable nodes in the console_scripts list (marked with red color).

Packages - Creating & building a Package CMakeLists.txt

The `ament_cmake` packages do not have `setup.py` and `setup.cfg` files. Everything is handled by a **`CMakeLists.txt`** file.

```
# find dependencies
find_package(ament_cmake REQUIRED)
find_package(geometry_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)

### ROS2 Interface Directives ###
set(interface_files
  # Messages
  "msg/AmazingQuote.msg"
  "msg/AmazingQuoteStamped.msg"

  # Services
  "srv/AddPoints.srv"
)

rosidl_generate_interfaces(${PROJECT_NAME}
  ${interface_files}
  DEPENDENCIES
  geometry_msgs
)

ament_export_dependencies(
  rosidl_default_runtime
)
### ROS2 Interface Directives [END] ###
```

lines to add in `CMakeLists.txt` file

Important!

- ensure required packages are declared with `find_package` for proper builds
- define interfaces
- generate message/service code
- export runtime dependencies

Nodes - What is a Node?

1. **Nodes** are **executable units** / processes, doing some computation.
2. **Each device or algorithm** in our distributed robotic system can be thought as a **distinct node**.
3. They are **decoupled**, meaning there is no shared memory between them, but they can **talk to each other by using interfaces**, like messages and services.
4. They communicate via **publish-subscribe** or **request-response** messaging patterns.

ROS2 Node Functionalities

```
ros2 node -h
```

Nodes - Creating & Running a Node

```python
import rclpy
from rclpy.node import Node

...
class PrintForever(Node):
    def __init__(self):
        super().__init__("print_forever")
        timer_period: float = 1.0
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.print_count: int = 0
    def timer_callback(self):
        self.get_logger().info(f"Printed {self.print_count} times.")
        self.print_count += 1

def main(args = None):
    try:
        rclpy.init(args = args)
        print_forever_node = PrintForever()
        rclpy.spin(print_forever_node)
    except KeyboardInterrupt:
        pass
    except Exception as e:
        print(e)

if __name__ == "__main__":
    main()
```

Simple ROS2 Python Node

create_timer

`create_timer( timer_period_sec, callback)`

Nodes - Creating & Running a Node

Let's assume that the node `print_forever_node.py` belongs to the package `py_pkg_node`. To make the node executable, we need to add it to the `console_scripts` list in the `setup.py` file as it is shown right below (`<print_forever_node>` is chosen as the name of our node).

At the end of setup.py file

```
entry_points = {'console_scripts': ['print_forever_node =
py_pkg_node.print_forever_node:main']}
```

Now, after we build the modified package, we can type the command `ros2 run py_pkg_node print_forever_node` to run the node, or, in the general case:

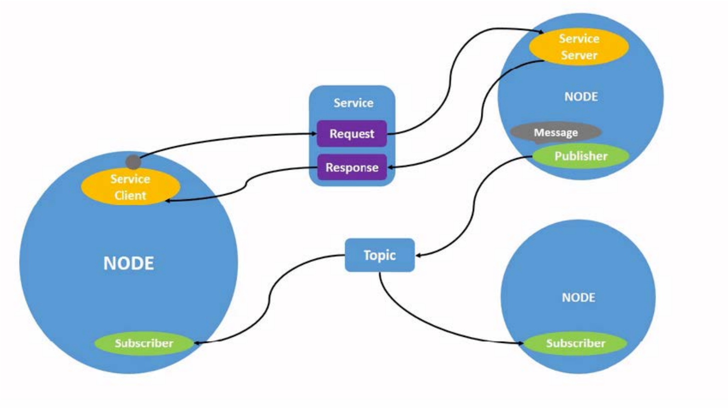
Command to run a node

```
ros2 run <pkg_name> <node_name>
```

To see the currently available (running) nodes, type `ros2 node list`.

ROS2 Communication System - General Communication Scheme

ROS2 General Communication Scheme

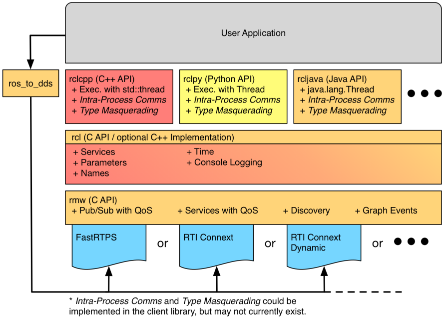


Here we see a basic ROS 2 communication scheme consisting of three nodes. Each node can act as a **publisher**, **subscriber**, **service server**, or **service client**. They communicate using simple messages over topics, and request–response message pairs through services.

ROS2 Communication System - ROS Client Library (RCL)

1. **RCL** is a **library layer** that helps client libraries use the underlying ROS2 MiddleWare.
2. It offers us an **API for writing nodes in different languages**, making the development and debugging of robot applications easier and more flexible.
3. Library implementations:
   - **rclcpp** (C++)
   - **rclpy** (Python)
   - **rcljava** (Java)

   rclcpp supports all rcl operations, while rclpy and rcljava support a subset of them.



ROS2 Internal Interfaces

ROS2 Communication System - Interfaces

Interfaces are files written in the ROS2 Interface Description Language (IDL). We can use them to communicate between nodes, that can even be written in different languages (Python, C++, ...). They are of 3 types:

- **Messages** → `.msg` files (simplest interface form)
- **Services** → `.srv` files (request & response messages)
- **Actions** → `.action` files (mixture of messages and services)

Some very well-known interface packages are **`geometry_msgs`** and **`sensor_msgs`**. Of course, we can write our own interface packages.

ROS2 Interface Functionalities

```
ros2 interface -h
```

ros2 interface show geometry_msgs/msg/Point

```
# This contains the position
of a point in free space
float64 x
float64 y
float64 z
```

ROS2 Communication System - Interfaces Messages

1. **Messages** (`.msg`) are data structures, functioning as the building blocks of communication over **topics**, **services**, and **actions**.
2. A message consists of **typed fields** (e.g. `int32`, `uint64`, `float64`, `bool`, `string`, `time`), that describe the exchanged data.
3. **Topics** are named communication channels (**buses**) that use **exclusively messages** as their payload.

ros2 interface show geometry_msgs/msg/Pose

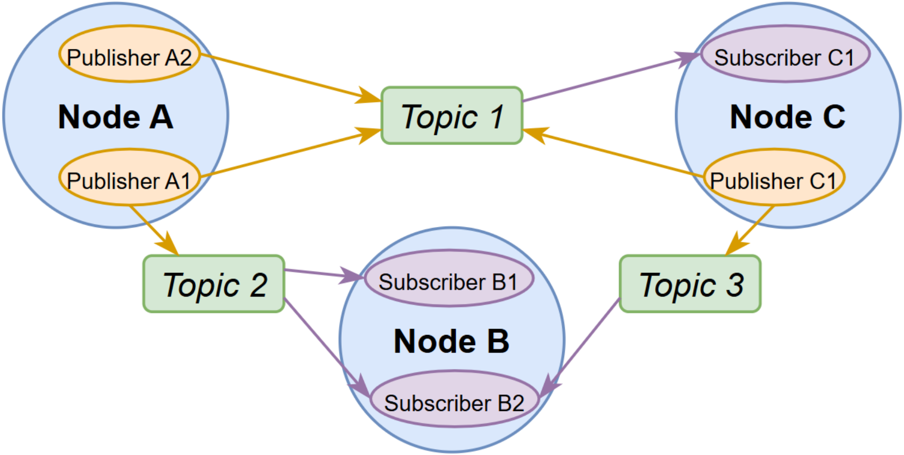
```
# A representation of pose
in free space, composed of
position and orientation

Point position
Quaternion orientation
```

To see the currently available topics, type `ros2 topic list`.

ROS2 Topic Functionalities

```
ros2 topic -h
```

ROS2 Communication System - Interfaces Messages



Topics, Publishers & Subscribers: Example communication scheme

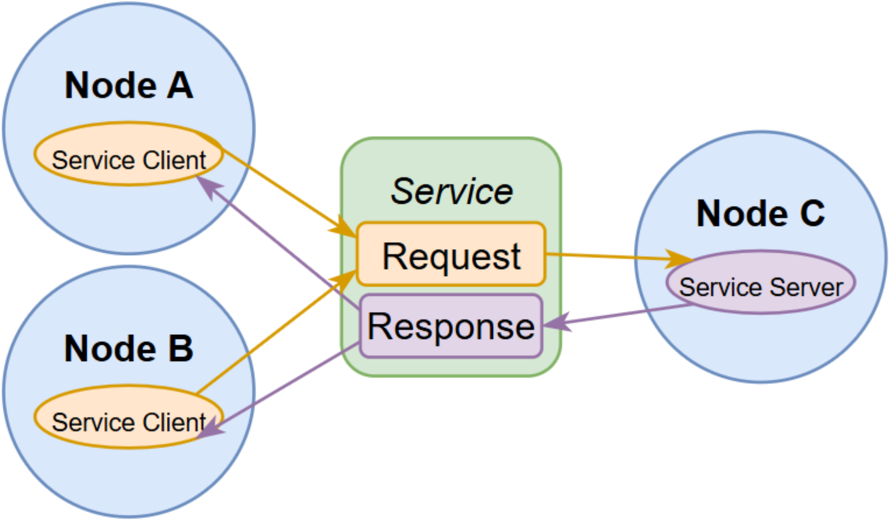
ROS2 Communication System - Interfaces Services

1. A **Service** is a **communication mechanism** similar to a topic, as both enable data exchange between nodes.
2. It differs from a topic in the sense that a topic is *one-way*, and *continuous* (publish - subscribe), while a service is ***two-way*** (bidirectional), and ***on-demand*** (request - response).
3. Each service should communicate with a ***single*** **Service Server**, that receives and processes a **request** message, and sends back to the Service Client a **response** message. There can be ***multiple*** **Service Clients** connected to a service.

pkg_with_interfaces/srv/AddPoints.srv (example, not native)

```
# Adds the values of points 'a' and 'b' to give the 'result'

geometry_msgs/Point a                                    # Request
geometry_msgs/Point b
---
geometry_msgs/Point result                              # Response
```

ROS2 Communication System - Interfaces Services



Services, Servers & Clients: Example communication scheme

ROS2 Communication System - Interfaces Actions

1. An **Action** is a **communication mechanism** similar to a service, as both provide *two-way, on-demand* interaction.
2. It differs from a service in the sense that it is designed for ***long-running tasks***, where the client may need **feedback** during execution, and the option to **cancel** the task.
3. Each action is defined by three message structures:
    - **Goal** (sent by the client to the server)
    - **Result** (sent by the server after completion)
    - **Feedback** (sent by the server while executing)

ros2 interface show nav2_msgs/action/FollowWaypoints

```
geometry_msgs/PoseStamped[] poses                    # Goal
---
int32[] missed_waypoints                             # Result
---
uint32 current_waypoint                              # Feedback
```

ROS2 Communication System - Creating a Publisher

create_publisher

`create_publisher(msg_type, topic, qos_profile)`

pkg_with_interfaces/msg/AmazingQuote.msg

```
# Quote Message Structure

int32 id
string quote
string philosopher_name
```

Important!

package `pkg_with_interfaces` is a dependency, so we need to add it to the `package.xml` file.

```python
import rclpy
from rclpy.node import Node
from pkg_with_interfaces.msg import AmazingQuote

...
class AmazingQuotePublisherNode(Node):
    def __init__(self):
        super().__init__("amazing_quote_publisher_node")
        self.amazing_quote_publisher = self.create_publisher(
            msg_type = AmazingQuote,
            topic = "/amazing_quote",
            qos_profile = 1)
        timer_period: float = 1.0
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.incremental_id: int = 0
    def timer_callback(self):
        amazing_quote = AmazingQuote()
        amazing_quote.id = self.incremental_id
        amazing_quote.quote = "Use the force, Pikachu!"
        amazing_quote.philosopher_name = "Uncle Ben"
        self.amazing_quote_publisher.publish(amazing_quote)
        self.incremental_id += 1

def main(args = None):
    # Written in the standard try-except scheme for Nodes
    ...

if __name__ == "__main__":
    main()
```

Simple ROS2 Python Publisher

ROS2 Communication System - Creating a Subscriber

```python
import rclpy
from rclpy.node import Node
from pkg_with_interfaces.msg import AmazingQuote

...
class AmazingQuoteSubscriberNode(Node):
    def __init__(self):
        super().__init__("amazing_quote_subscriber_node")
        self.amazing_quote_subscriber = self.create_subscription(
            msg_type = AmazingQuote,
            topic = "/amazing_quote",
            callback = self.amazing_quote_subscriber_callback,
            qos_profile = 1)
    def amazing_quote_subscriber_callback(self, msg: AmazingQuote):
        self.get_logger().info(f"\" I have received the most amazing of quotes.
        It says '{msg.quote}'. It was thought by the genious -- {msg.
        philosopher_name}. This latest quote had the id = {msg.id}. \"")

def main(args = None):
    # Written in the standard try-except scheme for Nodes
    ...

if __name__ == "__main__":
    main()
```

Simple ROS2 Python Subscriber

create_subscription

```
create_subscription(
msg_type,
topic,
callback,
qos_profile)
```

ROS2 Communication System - Creating a Service Server

create_service

```
create_service(srv_type,
srv_name, callback)
```

pkg_with_interfaces/srv/AddPoints.srv

```
# Adds the values of
points 'a' and 'b' to give
the 'result'

geometry_msgs/Point a
geometry_msgs/Point b
---
geometry_msgs/Point result
```

```python
import rclpy
from rclpy.node import Node
from pkg_with_interfaces.srv import AddPoints

class AddPointsServiceServerNode(Node):
    def __init__(self):
        super().__init__("add_points_service_server")
        self.service_server = self.create_service(
            srv_type = AddPoints,
            srv_name = "/add_points",
            callback = self.add_points_service_callback
        )
        self.service_server_call_count: int = 0
    def add_points_service_callback(self,
                                    request: AddPoints.Request,
                                    response: AddPoints.Response
                                    ) -> AddPoints.Response:
        response.result.x = request.a.x + request.b.x
        response.result.y = request.a.y + request.b.y
        response.result.z = request.a.z + request.b.z
        return response

def main(args = None):
    # Written in the standard try-except scheme for Nodes
    ...

if __name__ == "__main__":
    main()
```

Simple ROS2 Python Service Server

ROS2 Communication System - Creating a Service Client

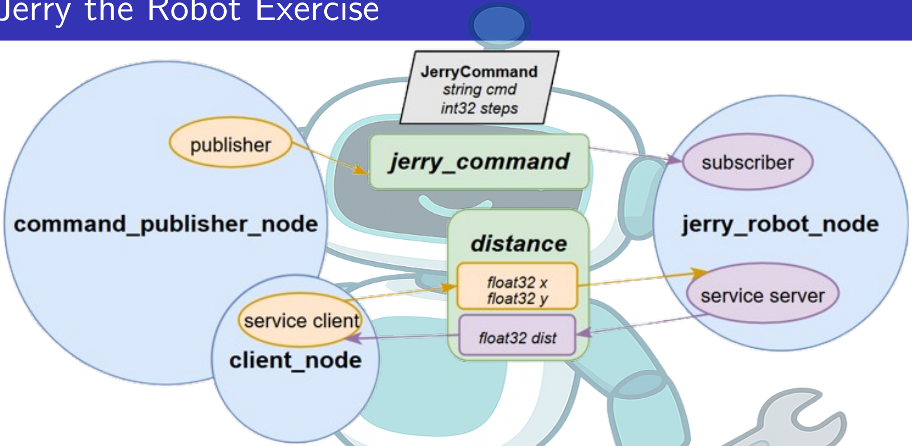
```python
import random
import rclpy
from rclpy.node import Node
from rclpy.task import Future
from pkg_with_interfaces.srv import AddPoints

class AddPointsServiceClientNode(Node):
    def __init__(self):
        super().__init__("add_points_service_client")
        self.service_client = self.create_client(
            srv_type = AddPoints,
            srv_name = "/add_points"
        )
        while not self.service_client.wait_for_service(timeout_sec = 1.0):
            self.get_logger().info(f"Service {self.service_client.srv_name} not available,
            waiting ...")
        self.future: Future = None
        self.timer = self.create_timer(0.5, self.timer_callback)
    def timer_callback(self):
        request = AddPoints.Request()
        request.a.x = random.uniform(0, 100)
        request.a.y = random.uniform(0, 100)
        request.a.z = random.uniform(0, 100)
        request.b.x = random.uniform(0, 100)
        request.b.y = random.uniform(0, 100)
        request.b.z = random.uniform(0, 100)
        self.get_logger().info(f"Trying to add ({request.a.x}, {request.a.y}, {request.a.z}) and
        ({request.b.x}, {request.b.y}, {request.b.z}) ...")
        if self.future is not None and not self.future.done():
            self.future.cancel()
            self.get_logger().warn("Service Future cancelled. The Node took too long to process the
            service call. Is the Service Server still alive?")
        self.future = self.service_client.call_async(request)
        self.future.add_done_callback(self.process_response)
    def process_response(self, future: Future):
        response = future.result()
        if response is not None:
            self.get_logger().info(f"The result was {(response.result.x, response.result.y,
            response.result.z)}")
        else:
            self.get_logger().info("The response was None.")

def main(args = None):
    # Written in the standard try-except scheme for Nodes
    ...

if __name__ == "__main__":
    main()
```

Simple ROS2 Python Service Client

create_client

```
create_client(
srv_type,
srv_name)
```

Jerry the Robot Exercise



Jerry is a simulated robot in a 2D world. The **`command_publisher_node`** sends random movement commands (*"forward"*, *"backward"*, *"left"*, *"right"*) with step counts as *JerryCommand* messages. The **`jerry_robot_node`** updates Jerry's position subscribing to these commands, and provides a */distance* service for computing Jerry's distance to randomly located obstacles. The **`client_node`** queries this service to obtain the computed distance.

References

- **ROS Home**
- **ROS2 Documentation for all distros, here for Foxy**
- **Murilo's ROS2 Tutorial**
- **Kevin Wood ROS2 Youtube Tutorials**
- **rclpy library documentation**