Lab Instructions

For the first lab you need to complete **6 tasks** explained below. As you go along the document you have explanations and examples to help you accomplish them. You will need to provide your **source code** and a **report (pdf or markdown)** with **screenshots** showing relevant result for each part alongside necessary comments. Try to keep the naming for each task as comprehensible as possible.

For each task please also include the commands you need to run to make each one work for easy checkup and grading.

Example python scripts are provided alonside this instructions.

Prerequisites and Workspace

Ubuntu22 + Ros 2 Humble

This code has been tested with that setup. However, it is possible that it can work with other configurations.

DON'T FORGET TO ADD YOUR ROS_DOMAIN_ID CORRESPONDING TO YOU GROUP NUMBER

```
export ROS_DOMAIN_ID=[Group #]
```

You will need to do this in **every** terminal you launch. However you can add it to your .bashrc file

```
echo "export ROS_DOMAIN_ID=[Group #]" >> ~/.bashrc
source ~/.bashrc
```

Grading

The grading of this labs is the following:

- Grade 0: Nothing is delivered
- Grade 1: Task 1
- Grade 2: Task 2
- Grade 3: Task 3 + Task 4
- Grade 4: Task 5
- Grade 5: Task 6 + 2 Optional tasks

Logging

For ROS nodes the correct practice is to **use loggings** in diferent levels. Please try to use this instead of prints statements, depending on how you launch the nodes print **won't work**. The logging inside the clase can be in

different levels and can be done:

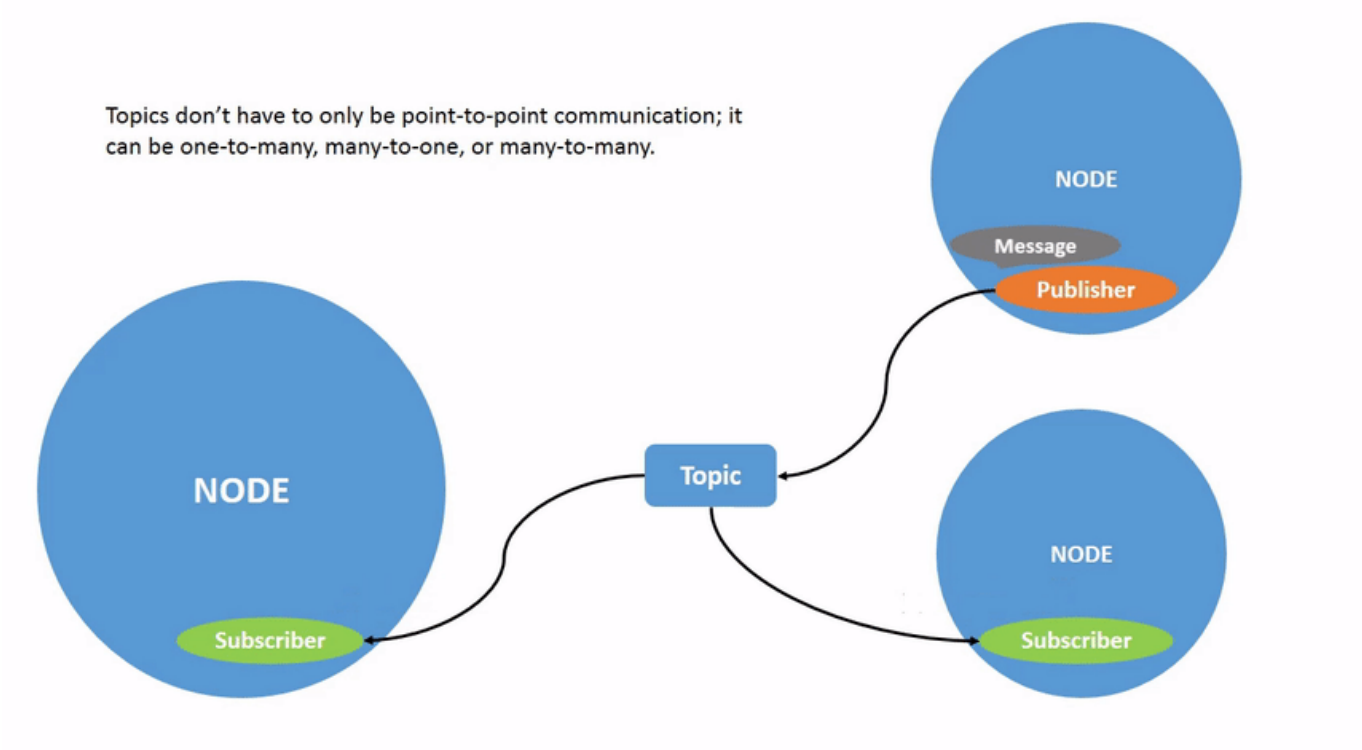
```
self.get_logger().debug(f'')
self.get_logger().info(f'')
self.get_logger().warn(f'')
self.get_logger().error(f'')
```

1.- Topics

First, we will discuss the basics of ROS2 and ROS2 topics. A topic is a named channel over which nodes publish and subscribe to data messages.

- **Publisher:** sends messages.
- **Subscriber:** receives messages.
- **Message type:** defines the structure of the data (e.g., sensor_msgs/msg/Image). (we will create our own messages later)

Topics are asynchronous. Publishers and subscribers don't know about each other, they just agree on the **topic name, message type and quality of service**. Think of a topic as a radio frequency. A publisher is the radio station, and subscribers are the radios tuned in. They don't talk to each other, just transmit and receive.



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

Useful commands

Now let's see some useful ROS2 commands for working with topics, afterwards, we will create our first package!

```
ros2 topic list
```

Shows you all currently active topics. If you don't see your topic, your node might not be running or publishing yet. For example:

```
/parameter_events
/rosout
/test
```

/parameter_events and */rosout* are two fundamental, automatically created topics in any ROS2 system that provide essential system-wide communication.

- ***/parameter_events:*** system-wide notifications about parameter changes within nodes.
- ***/rosout:*** central logging topic for all nodes in the system.

```
ros2 topic echo </topic_name>
```

Prints the data published to a topic in real-time. Super useful for checking if your publisher is working. Example:

```
data: Message 159
---
data: Message 160
---
```

```
ros2 topic info </topic_name>
```

Displays detailed info about a topic, including message type, number of publishers and number of subscribers, and quality of service settings. Example:

```
Type: std_msgs/msg/String
Publisher count: 1
Subscription count: 0
```

```
ros2 topic pub </topic_name> std_msgs/msg/String "{data: 'Hello from CLI'}"
```

Manually publishes a message into the topic. Great for testing subscribers. You can add arguments like *--once* to only publish one message or *--rate #* to define the frequency of the message.

```
ros2 topic hz </topic_name>
```

Tells you how fast messages are being published which helps when tuning sensors or checking timers.

Now that we have seen how topics work, let's create a minimal ROS2 package and make a publisher and subscriber pair to send and receive messages. We will use pyhton for simplicity.

Create a Package

To create a package in the Python environment using ROS2, we use *ros2 pkg create* command. First, create a folder called *ros2_ws*. This folder will be your **workspace**. Next, we have to create a source or *src* folder inside your workspace, it is where your packages will be.

```
mkdir -p ros2_ws/src
cd ros2_ws/src
```

Go to the *src* folder inside your *workspace* and enter the following command:

```
ros2 pkg create --build-type ament_python <package_name> --license Apache-2.0 --
dependencies rclpy std_msgs geometry_msgs
```

Let's break down this command:

- **ros2 pkg create**
    - ROS2 command to create a new package.
- **--build-type ament_python**
    - Tells ROS2 that this package will be written in Python and built using the ament_python build system.
- **--license Apache-2.0**
    - Sets the license type in the generated *package.xml*. You can replace this with MIT, BSD, etc. (optional)
- **--dependencies**

- A flag in ROS2 command that automatically adds these dependencies as both <**build_depend**> and <**exec_depend**> tags in your *package.xml* file.

After creating the package, we will have a folder inside the *src* with the package name. You will have something like this:

```
ros2_ws/
└── src/
    └── [package_name]/
        ├── package.xml
        ├── setup.py
        ├── resource/
        ├── <package_name>/
        └── test/
```

Feel free to explore the *package.xml* file. Some other dependencies that might be needed later are below and can be added manually:

```
<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>
<exec_depend>example_interfaces</exec_depend> <!-- for Int32 -->
<exec_depend>rcl_interfaces</exec_depend>   <!-- for parameter events -->
<exec_depend>action_msgs</exec_depend>
<exec_depend>rclpy_action</exec_depend>
```

Publisher

Navigate to *ros2_ws/src/<package_name>/<package_name>* and create two files named *publisher.py* and *subscriber.py* inside the folder.

Python script: *publisher.py*

Move and open the file *publihser.py* or copy the following code in your script:

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy,
HistoryPolicy

class MinimalPublisher(Node):

    def __init__(self):
        super().__init__('publisher')
        topic_name = '/----' # TODO: fill with the name of the topic
```

```
        self.qos_profile = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,         # Ensure delivery of
messages
            durability=DurabilityPolicy.TRANSIENT_LOCAL,    # Keep messages even
if the publisher dies
            history=HistoryPolicy.KEEP_LAST,                # Only store the last
N messages
            depth=10                                        # The number of
messages to store (N=10)
        )

        self.publisher_ = self.create_publisher(
            String,
            topic_name,
            self.qos_profile
            )

        timer_period = # TODO:      # time in seconds
        self.timer = self.create_timer(timer_period,
                                       self.timer_callback)

        self.i = 0


    def timer_callback(self):
        msg = String()
        msg.data = f''# TODO: fill out this line so using the counter provided in
the constructor.
        self.publisher_.publish(msg)
        self.get_logger().info(f'Publishing: {msg.data}')
        self.i += 1


def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()


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

You can quickly verify that it is working by **running the python script**.

```
python publisher.py
```

Expected output:

```
[INFO] [1756208372.479133544] [publisher]: Publishing: Message 0
[INFO] [1756208372.971478204] [publisher]: Publishing: Message 1
[INFO] [1756208373.472033853] [publisher]: Publishing: Message 2
[INFO] [1756208373.972118382] [publisher]: Publishing: Message 3
```

Register executables

Now, we need to tell ROS2 where the entry points of our nodes are. We can do this in *setup.py* (in cmake version, we have to do this in *CMakeList.txt*). Open *setup.py* and update the *entry_points* section:

```
entry_points={
    'console_scripts': [
        'publisher = <package_name>.publisher:main',
    ],
},
```

Then, we can build our package: go to your ROS2 workspace (**ros2_ws**) and run the following command:

```
colcon build
```

After the build, source the bash script inside the install folder:

```
source install/setup.bash
```

Tip: create an alias in your *.bashrc* file to make it easier for you

```
alias sb="source install/setup.bash"
```

Now we can use ROS2 to launch our script. Let's use the *ros2 run* command:

```
ros2 run <package_name> publisher
```

You should see the output in your terminal and it should be the same as before. Moreover, open another terminal, source the install folder as shown before and run the following:

```
ros2 topic list
```

You should be able to see your topic name among the other topics. You can also check the content of the topic with *ros2 topic echo <*topic_name*>.

Now that we have created a simple publisher, we need to see how subscribers work.

Subscriber

Python script: *subscriber.py*

Inside the *subscriber.py*, copy and paste the following, (or open the given file):

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy,
HistoryPolicy

class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('subscriber')
        topic_name = '/---' # TODO: # subscribe to the publisher’s topic.

        self.qos_profile = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,         # Ensure delivery of
messages
            durability=DurabilityPolicy.TRANSIENT_LOCAL,    # Keep messages even
if the publisher dies
            history=HistoryPolicy.KEEP_LAST,                # Only store the last
N messages
            depth=10                                        # The number of
messages to store (N=10)
        )

        self.subscription = self.create_subscription(
            String,
            topic_name,
            self.listener_callback,
            self.qos_profile)

        self.subscription


    def listener_callback(self, msg):
        self.get_logger().info(f'I heard: {msg.data}')


def main(args=None):
    rclpy.init(args=args)
```

```
    minimal_subscriber = MinimalSubscriber()
    rclpy.spin(minimal_subscriber)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()


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

Follow the same steps as before to setup the package again. First, we need to tell ROS2 that this is also an entry point, open *setup.py* and add the subscriber to it, your *entry_point* should look like this:

```
entry_points={
    'console_scripts': [
        'publisher = <package_name>.publisher:main',
        'subscriber = <package_name>.subscriber:main',
    ],
},
```

Afterwards, navigate to your *ros2_ws* and:

```
colcon build
source install/setup.bash
```

This time we will use *ros2 run* to run the subscriber node:

```
ros2 run <package_name> subscriber
```

As you have noticed, nothing happens, this is because the node has subscribed to the topic, but there's nothing published yet, so we need to run publisher too.

In your *ros_ws* open another terminal, source the installation in both terminals you are using.

Terminal 1:

```
source install/setup.bash
ros2 run <package_name> publisher
```

Terminal 2:

```
source install/setup.bash
ros2 run <package_name> subscriber
```

You should see the publisher printing messages and the subscriber logging what it receives.

```
[INFO] [1756208624.003176882] [subscriber]: I heard: Message 43
[INFO] [1756208624.003519465] [subscriber]: I heard: Message 44
[INFO] [1756208624.003799365] [subscriber]: I heard: Message 45
```

Analyzing the code

Going through the code to understand what is happening.

```
class MinimalPublisher(Node):

    def __init__(self):
        super().__init__('publisher')
```

Up to this point, we have created a class called *MinimalPublisher*, which inherits from the Node class. We pass the name of the node to the constructor of the parent class using *"super"*. When you run this node, ROS2 will know it as */publisher* in the node graph.

```
self.publisher_ = self.create_publisher(
        String,
        topic_name,
        self.qos_profile
        )
```

Here is what this line does:

- It tells ROS2 to publish messages of type *String*
- On the topic named "*topic*"
- With a Quality of Service of type ***qos_profile***

What is Quality of Service (QoS)?

It controls how messages are handled between the publisher and subscriber. In ROS2, when you create a publisher or subscriber, you pass in a QoS profile to define rules like:

- Should messages be reliable?
- Should late subscribers get old data?
- How many messages should be stored in the queue?

Right now the qos_profile is:

```
self.qos_profile = QoSProfile(
    reliability=ReliabilityPolicy.RELIABLE,         # Ensure delivery of messages
    durability=DurabilityPolicy.TRANSIENT_LOCAL,    # Keep messages even if the
publisher dies
    history=HistoryPolicy.KEEP_LAST,                # Only store the last N
messages
    depth=10                                        # The number of messages to
store (N=10)
)
```

Reliability Policy

The Reliability Policy tell us whether messages are guaranteed to be delivered.

- **Reliable:** ensures that every message published will be received by the subscriber, even if it requires retransmissions.
- **Best effort:** messages are sent without a guarantee of delivery.

Durability Policy

The Durability Policy determines how messages are handled for late-joining subscribers.

- **Transient local:** means that when a new subscriber connects, it will receive a history of messages that the publisher has stored.
- **Volatile:** the publisher does not store any messages for late-joining subscribers.

History Policy

The HistoryPolicy specifies how the communication layer should manage its message buffer.

- **Keep Last:** This is the most common policy. The system stores only the last N messages, where N is defined by the *depth* parameter.
- **Keep All:** tells the system to store all messages up to the limit of the available memory.

Depth

The depth parameter is an integer that works with *HistoryPolicy.KEEP_LAST*. It defines the maximum number of messages that will be stored in the publisher's buffer.

However, remember that **your subscriber and publisher must have the same QoS!! No warnings, no errors, they just won’t connect.**

```
timer_period = # TODO:      # time in seconds
self.timer = self.create_timer(timer_period,
                                    self.timer_callback)
```

This sets up a timer that fires every # seconds. Each time it triggers, it will call *self.timer_callback()*. This is how your publisher sends messages periodically. Let's head to the callback function:

```
msg = String()
msg.data = f''# TODO: fill out this line so using the counter provided in the
constructor.
self.publisher_.publish(msg)
self.get_logger().info(f'Publishing: {msg.data}')
self.i += 1
```

Basically, we are telling Python to create a new message of type String, set its *.data* field, publish it, log it, and increment the counter.

Now, as the last part, let's head to the main function and see how it all comes together:

```
rclpy.init(args=args)
minimal_publisher = MinimalPublisher()
rclpy.spin(minimal_publisher)
minimal_publisher.destroy_node()
rclpy.shutdown()
```

rclpy.init(args=args) initializes the ROS2 Python client library (rclpy).

- It sets up the whole ROS2 communication system such as topics, nodes, logging, etc.
- It must be called before you create or use any nodes.
- The *args=args* part allows the node to accept command-line arguments (e.g. *--ros-args*), even though you're not using any right now.

Next, we create an instance of the publisher class that we created, and we pass it to *rcply.spin*. This is the loop that **keeps the node alive.**

- It waits for callbacks to happen and in this case the timer.
- If you had a subscriber, it would also wait for incoming messages and trigger *listener_callback()*.
- **The program will sit here until you manually stop it (Ctrl+C). This is important to understand!!**
- Next two lines make sure that when the node is done (usually after you press Ctrl+C), the program is cleaned up and is fully shutted down the ROS2 communication system for your process.

TASK 1

Create a publisher/subscriber pair for a *PoseStamped* ROS2 message at a given frequency. The *PoseStamped* message needs to simulate the movement of a robot in a straight line. For this taks you need to do the following:

- Check on the ROS2 documentation the format of the message you need.
- Create a new package for the publisher and the subscriber nodes.

- Create a publisher node for the *PoseStamp* message, simulating a straight line movement in any direction.
- Create a node that subscribes to the data and prints *x* and *y* positions.
- Use the corresponding commands to test in the terminal
    - The existence of the topic
    - The value of the message
    - The frequency of the topic
    - The number of subscribers and publisher for the topic
- OPTIONAL 1:
    - Explain what would be the QoS parameters for critical data in the system?
    - What about live data?
    - What type of data would it make sense to use Transient Local with?

Take screenshots of the outputs to include in your report!

2.- Custom Messaging

ROS2 provides many useful interfaces, but sometimes we might need to create our own.

We will create a new package for this, since currently there is no way to generate a custom interface in a pure python package. However, you can create a *custom_interface* in a CMake package and then use it in a Python node.

```
ros2 pkg create --build-type ament_cmake --license Apache-2.0 custom_interface --
dependencies rclcpp std_msgs
```

Notice that we have added the *--dependencies* flag for *rclcpp* and *std_msgs*. This will add a few dependencies to our package. We could also add these dependencies manually to *package.xml*

Now, go inside the created package folder and make new a folder called *msg*. Inside it, we will create a file called *Person.ms*g

```
cd custom_interface
mkdir msg
touch msg/Person.msg
```

Inside *Person.msg*

```
string name
int32 age
bool is_student
```

This is the general format of a message file: [type] [name]. You can also use ROS2 stablished messages for a more complex message type.

Modify CMakeList

We have to make some changes to our *cmakelist* file. We need to add two functions:

find_package()

Add the following:

```
find_package(rosidl_default_generators REQUIRED)
```

Immediately below the function

rosidl_generate_interfaces()

Add the following:

```
rosidl_generate_interfaces(${PROJECT_NAME} "msg/Person.msg" )
```

Modify package.xml

Now we must modify *package.xml* to add the necessary dependencies:

```
<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Go back to the root of your workspace, build and source the packages:

```
colcon build --packages-select custom_interfaces
source install/setup.bash
```

To check if you have built it successfully, run the following command:

```
ros2 interface show custom_interface/msg/Person
```

You should see:

```
string name
int32 age
bool is_student
```

TASK 2

For this task you need to incorporate a new message type with a publisher and subscriber node. You can either add a *setup.py* to the current *custome_interface* package and start writing your nodes OR the **(prefere method)** you can create another python package as before. For the new package you need to add the custom message as dependency.

Message format needs to be:

```
string name
int32 age
bool is_student
```

Tip for the setup.py

```
install_requires=['setuptools', 'rclpy', 'custom_interface'],
```

You have to:

- Create a new package for the cusotm messge of this task.
- Create a new message with the provided format.
- Create or a adapt a package to work with both a custom message and python scripts.
- Create a publisher of the new message type, where it will publish a person's age. The person needs to ba a studen between the ages of 10 and 18.
- Create a subscriber to get the information of the topic and log the information.
- **OPTIONAL 2:** add a *Header message* to the *<Message_name>.msg* so you can have the timestamp of the message. **Tip:** you will need to add dependencies.
    - Modify the subscriber and check that the frequency of your publisher is indeed working using the data obtained from the timestamp in the Header.

3.- Services

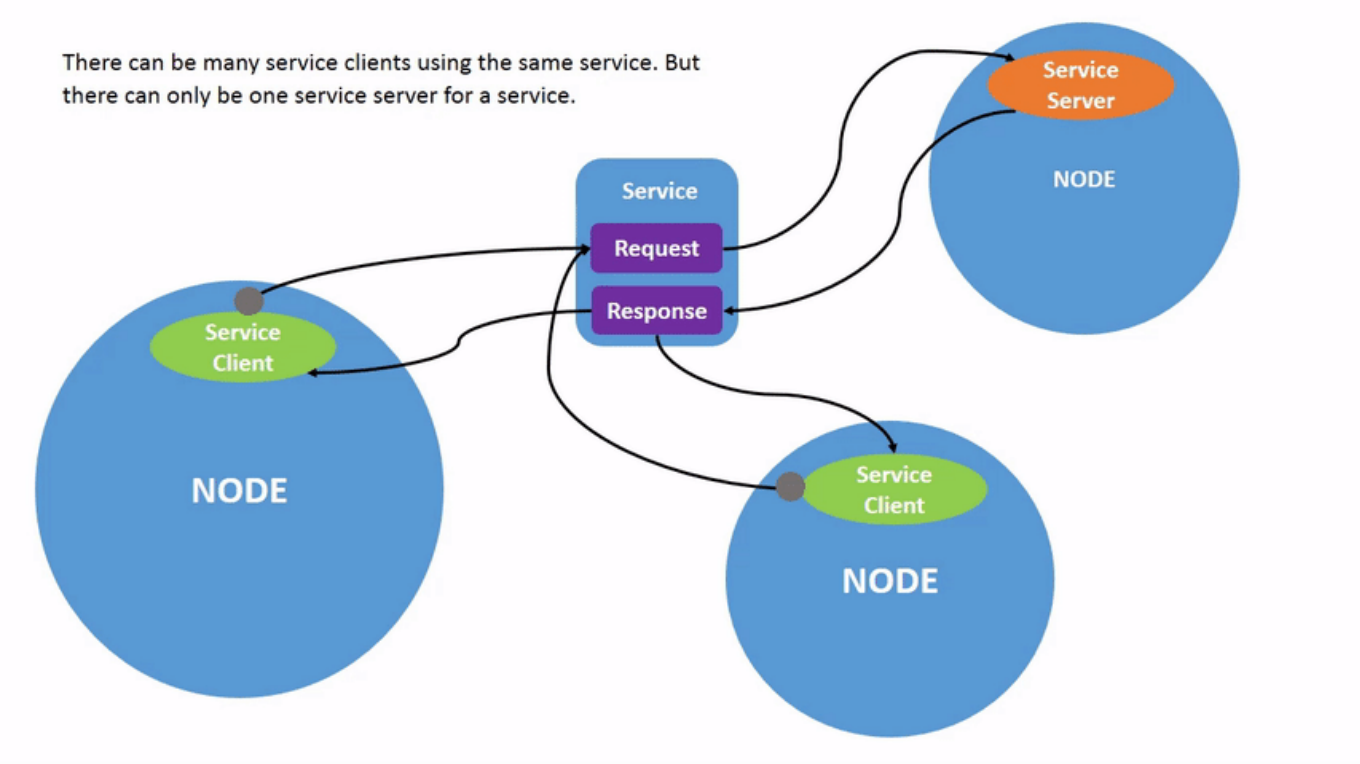
A Service in ROS2 is a way for nodes to do synchronous communication, meaning one node asks for something, and another node responds immediately.

It works like a function call across processes or even computers.

- **Client node:** makes a request.

- **Service Server node:** receives the request, does something, then returns a result.
- This is **not continuous** like topics; it only runs when requested.

In other words, you can see how services work as how client and server work in web development; unlike services, nothing is being continuously published or listened to, something happens only when a node requests it from another node. Let's dive deeper into how they work.



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

Structures of the Services

ROS2 services use a *.srv* file to define two parts:

```
<request fields>
---
<response fields>
```

For example, imagine we want to create a service that receives two inputs and returns the sum of those two inputs. The *.srv* file would be like:

```
int64 a
int64 b
---
int64 sum
```

Keep in mind that the **3 dashes need to be there** after the request fields, no more no less.

This means:

- The client sends *a* and *b*.
- The server receives them, adds them, and sends back *sum*.

CLI Tools for the services

Before heading into creating a package to test the services, let's go through some tools to work with the services through the CLI.

```
ros2 service list
```

Shows all services currently running in the system.

```
ros2 service type /add_two_ints
```

Returns the full message type of the service (e.g., example_interfaces/srv/AddTwoInts).

```
ros2 service info /add_two_ints
```

Tells you which node is offering the service and its type.

```
ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts "{a: 5, b: 9}"
```

Sends a request with *a=5*, *b=9*, and prints the result. Try this when a server is running and you want a quick test of it.

Server and Client Example

Now let's dive into creating a simple service and client. Go ahead and create another package using the following command:

```
ros2 pkg create --build-type ament_python --license Apache-2.0 <service_example> -
-dependencies rclpy example_interfaces
```

Again, notice that we have added *--dependencies* flag for *rclpy* and *example_interfaces*. We are going to use *example_interfaces* to use the *.srv* file that we creating.

example_interfaces is and exiting ROS2 service

In case that we didn't use *--dependecies* flag, we have to add the needed dependencies on *package.xml* and *setup.py* manually. In **package.xml** we need to add *example_interfaces* and in **setup.py** we need to update *install_require* line: *install_requires=['setuptools', 'example_interfaces']*. **Same for rclpy**.

Server

Navigate to the folder <*service_example*> folder and create a file called *add_two_ints_server.py*.

Python script: *add_two_ints_server.py*

Copy and paste the following code inside the file:

```
import rclpy
from rclpy.node import Node
from example_interfaces.srv import AddTwoInts

class SumServerNode(Node):

    def __init__(self):
        super().__init__("sum_server")
        self.server_ = self.create_service(AddTwoInts, "----",
self.sum_service_callback) # TODO: give your sevice a name
        self.get_logger().info("Service server Python node has been created")


    def sum_service_callback(self, request, response):
        response.sum = TODO: # Add the integers
        self.get_logger().info(f'Processed request: {request.a} + {request.b} ->
{response.sum}')
        return response


def main(args = None):
    rclpy.init(args=args)
    node = SumServerNode()
    rclpy.spin(node)
    rclpy.shutdown()


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

Client

Python script: *add_two_ints_client.py*

Create another file for the client called *add_two_ints_client.py* and paste the following:

```
import rclpy
from rclpy.node import Node
from example_interfaces.srv import AddTwoInts
from functools import partial

class SumClientNode(Node):

    def __init__(self):
        super().__init__('sum_client_node')
        self.get_logger().info('Sum Client Python node has been created')

        # declare parameters for AddTwoInts
        a_ = # TODO: fill with a value
        b_ = # TODO: fill with a value

        self.call_sum_server(a_, b_)


    def call_sum_server(self, a, b):
        client = self.create_client(AddTwoInts, '----') # TODO: fill with your
service name
        while not client.wait_for_service(1.0):
            self.get_logger().info('Waiting for the Server...')

        # create request
        request = AddTwoInts.Request()
        request.a = # TODO: asign the correct variable
        request.b = # TODO: asign the correct variable

        #send request asynchronously
        future = client.call_async(request)
        future.add_done_callback(partial(self.sum_service_callback, a=a, b=b))


    def sum_service_callback(self, future, a, b):
        try:
            response = future.result()
            self.get_logger().info(f'-----') # TODO: print the values and the
result
        except Exception as e:
            self.get_logger().info(f'Service call failed {e}')


def main(args=None):
    rclpy.init(args=args)
    node = SumClientNode()
    rclpy.spin(node)
    rclpy.shutdown()
```

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

Don’t forget to add the entry points.

Build and source the package, and finally:

```
ros2 run py_services add_two_ints_server
```

Open another terminal, source your installs, and:

```
ros2 run py_services add_two_ints_client
```

This should give you:

```
[INFO] [1756211740.343215393] [sum_client_node]: Sum Client Python node has been
created
[INFO] [1756211740.597404341] [sum_client_node]: 4 + 7 = 11
```

Custom Services

Just as the custom messages in the previous section, we will use the *custom_interface* package that we created to store the new *.srv* files. Let’s go back to the *custom_interface* package folder. Inside that create a folder called *srv*:

```
mkdir srv
touch srv/AddThreeInts.srv
```

Inside this new service file, write:

```
int64 a
int64 b
int64 c
---
int64 sum
```

As you have guessed, we are going to use this file to add three integers this time. Go to the *CMakeList* file and add "*srv/AddThreeInts.srv*" to the *rosidl_generate_interface* function.

Now go back to the root, build the package and source it afterwards.

Confirm the package build by running:

```
ros2 interface show custom_interface/srv/AddThreeInts
```

It should show you the interface you created.

You could now modify your code to work with three numbers and then test it.

TASK 3

For this task you need to create a service that performs basic robot control calculations. You need to implement a service */Calculate_distance* that takes two 2D points (*geometry_msgs/Point*) and returns the Euclidean distance. It needs to also return a bool for success and a strig message.

You need to:

- Create a new package for the task.
- Create a new custom service interface with two 2D points and three return variables (distance, success and message). This can be created either in you package or in the *custom_interface* package from before.
- Create server node that receives the two points, computes the distance and sends the return variables.
- Create a client node that to use this service
- Show your service interface using

  ```
  ros2 interface show [your_package]/srv/[your_service]
  ```

4.- Parameters

ROS2 parameters are configurable values that can be dynamically set and accessed. It allows for changing values without having to modify the code directly. They are commonly used for tunning algorithms, setting hardware configuration, specifying modes of operation and configure behavior. Parameters are also very important when implementing launch files.

We will be using the *AddTwoInts* from the service portion of this lab. You can create a copy of the files to add parameters.

Declare and get parameters

We use two simple methods from the class rclpy.node.Node.

Declare

```
self.declare_parameter( <parameter_name>, <default_value>)
```

Get

```
parameter = self.get_parameter(<parameter_name>).value
```

To run the code now you need to send the values of the parameters using *--ros-args:*

```
ros2 run <pkg_name> <node_name> --ros-args -p <param1_name>:=<value> -p
<param2_name>:=<value>
```

For this example, for the modified file let's try:

```
ros2 run service_example param_client --ros-args -p a:=5 -p b:=13
```

Useful Tools

```
ros2 param get <node_name> <param_name>
```

Ros2 param get will list the current param values that are set in memory.

```
ros2 param set <node_name> <param_name> <value>
```

Ros2 param set will change the current param value at runtime.

```
ros2 param describe <node_name> <param_name>
```

Ros2 param describe will give information about param values.

YAML Files

YAML files are commonly used to specify parameter configurations for nodes and other components within a ROS2 system. It is a human readable data serailization format, well suited for representing structured data.

YAML files are ideal for defining parameter configurations.

An example of a parameter declaration in yaml file is:

```
arg:
    name: "param1"
    default: "value1"
arg:
    name: "param2"
    default: "value2"
```

Let's create a basic one for the previous example in a folder named *config/* and name it *params.yaml*:

```
sum_client_node:  # Name of your node!
    ros__parameters:
        a: 5
        b: 23
```

OR

```
/**:
    ros__parameters:
    a: 5
    b: 23
```

Note:* "/**:"* is wildcard where the parameter applies to any node that runs the file.

Now run:

```
ros2 run service_example param_client --ros-args --params-file
[path_to_file]/params.yaml
```

TASK 4

For this task you will modify your service from Task 3 (*/calculate_distance*).

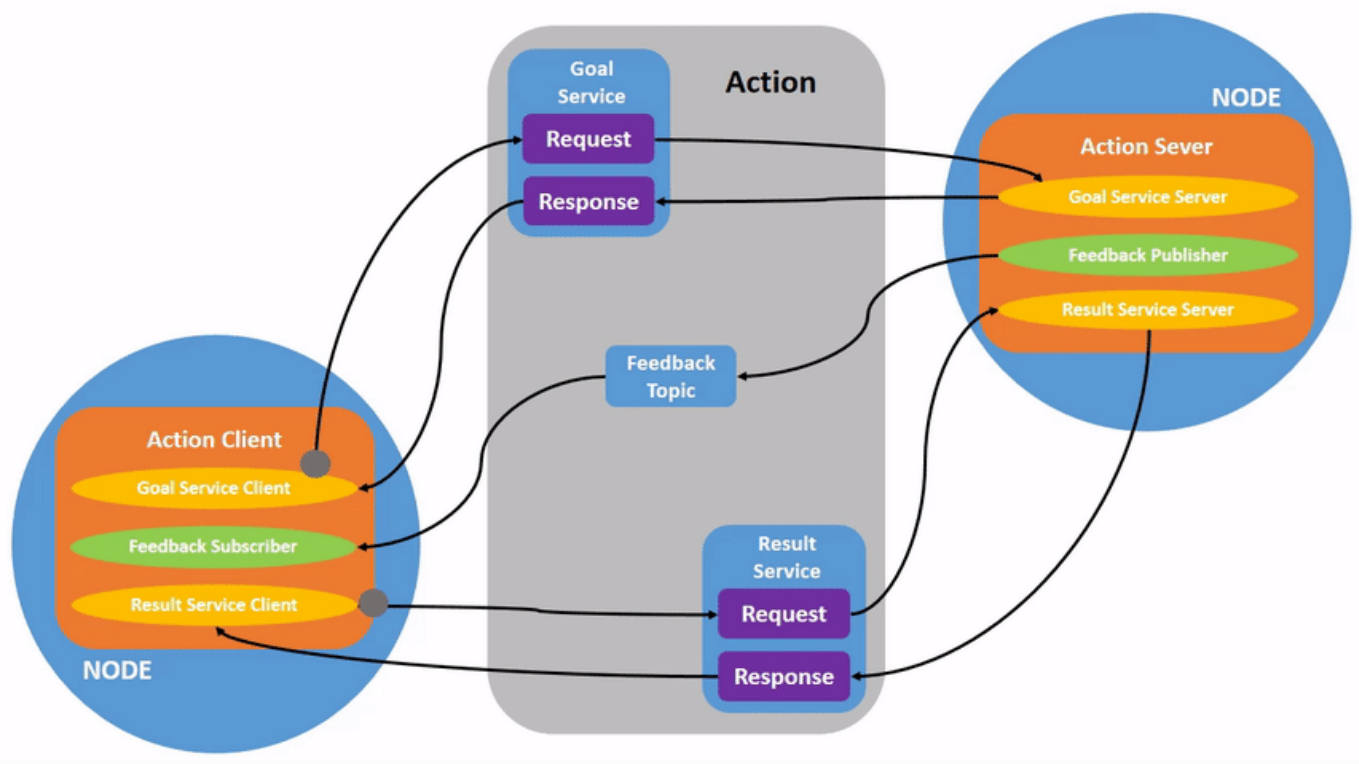
- Set as parameters the values for the two given points.
- Create a yaml file to launch the client with it.
- Launch the the client and server from command line while reading from the file.

5.- Actions

Resource: https://www.roboticsunveiled.com/ros2-actions-in-python-and-cpp/

In ROS2 actions are a long running remote asynchronous call procedure that has feedback and the ability to cancel or preempt the goal.

For example, a robot planner that makes a robot go from A to B. The movement of the robot can take several minutes and may result in a failure during execution. The robot planner could also decide to change the goal or abort the movement all together. The feedback given could be done to verify the robot's position to check on the action.



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

As with the service, and action has at least a server and one client, though it could have multiple ones.

The *.action* interface contains the following three fields:

- Goal
- Feedback
- Result

Actions vs Services

Actions and services both provide a protocol for request-response communication. However, actions can deal with more complex tasks since they can cancel the request and have feedback and result. Actions can also manage multiple clients and can be implemented fully asynchronously.

- **Actions**
    - Manage long running task
    - Can cancel a goal
    - Gives feedback

    - Multiple robots can request the task, and an interface will autonomously mange the requests

- **Services**

    - Short live server-client task
    - Client node will be stuck during the request server execution.

Action Interface

Like for the messages and services we will use the *custom_interface* package to create our action. It is common practice to put all custom interface definitions like **messages, services and action** in a single package. This keeps everything organized and accessible for other packages in the workspace, making the structure cleaner and more manageable. It also helps, since for other packages you just need to declare a single dependency.

In the package *custom_inteface* create a folder named *action*, with the file *Fibonacci.action* with the following structure:

```
# Goal
int64 order
---
# Result
int64[] result_sequence
---
# Feedback
int64[] partial_sequence
```

CMake

In the *cmake file* adapt the following to fit your case:

```
# Add to your find_package call
find_package(ament_cmake REQUIRED)
find_package(rosidl_default_generators REQUIRED)
find_package(action_msgs REQUIRED)

# This command will process all .msg, .srv, and .action files
rosidl_generate_interfaces(${PROJECT_NAME}
    "msg/MyCustomMsg.msg"
    "srv/MyCustomSrv.srv"
    "action/MyCustomAction.action"
    DEPENDENCIES action_msgs
)
```

Package.xml

For the *packaging.xml* file make sure you have the following:

```
<build_depend>action_msgs</build_depend>
<exec_depend>action_msgs</exec_depend>
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
```

Finally build the package and source your workspace.

Server

Lets create a new package for the server and the client.

```
ros2 pkg create --build-type ament_python fibonacci --dependencies rclpy
custom_interface
```

In the package *custom_inteface* create a folder named *action*, with the file *Fibonacci.action* with the following structure:

```
int64 order
---
int64[] result_sequence
---
int64[] partial_sequence
```

CMake

In the cmake file adapt the following to fit your case:

```
# Add to your find_package call
find_package(ament_cmake REQUIRED)
find_package(rosidl_default_generators REQUIRED)
find_package(action_msgs REQUIRED)

# This command will process all .msg, .srv, and .action files
rosidl_generate_interfaces(${PROJECT_NAME}
    "msg/MyCustomMsg.msg"
    "srv/MyCustomSrv.srv"
    "action/MyCustomAction.action"
    DEPENDENCIES action_msgs
)
```

Package.xml

For the packaging.xml file make sure you have the following:

```
<build_depend>action_msgs</build_depend>
<exec_depend>action_msgs</exec_depend>
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
```

Finally build the package and source your workspace.

Server and Client

Create a *fibonacci_sever.py* and *fibonacci_client.py* in the corresponding folder. Don't forget to add the entry points to the *setup.py* and the dependencies to the *package.xml*

```
entry_points={
        'console_scripts': [
            'fibonacci_server = [package].fibonacci_server:main',
            'fibonacci_client = [package].fibonacci_client:main',
        ],
    },
```

And this:

```
<depend>rclpy</depend>
<depend>custome_interface</depend>
```

Server

Python script: *fibonacci_server.py*

In the server file copy and fill the following:

```
import rclpy
from rclpy.node import Node
from custom_interface.action import Fibonacci
from rclpy.action import ActionServer
import time

class FibonacciServerNode(Node):

    def __init__(self):
        super().__init__('fibonacci_server_node')

        self.action_server_ = ActionServer(
            self,
            Fibonacci,
```

```
            '----', # TODO: fill with the name of the action
            execute_callback=self.execute_callback
        )

        self.get_logger().info('fibonacci action server has been started')


    def execute_callback(self, goal_handle):
        """execute the action and return the result"""

        self.get_logger().info('executing the goal')

        # get goal request value
        order = goal_handle.request.order

        # initialize feedback
        feedback_msg = Fibonacci.Feedback()
        feedback_msg.partial_sequence = [0, 1]

        # process Fibonacci
        for i in range(1, order):

            feedback_msg.partial_sequence.append(feedback_msg.partial_sequence[i]
+ feedback_msg.partial_sequence[i-1])

            self.get_logger().info(f'Feedback: {feedback_msg.partial_sequence}')

            goal_handle.publish_feedback(feedback_msg)

            time.sleep(--) # TODO: fill with the time delay in seconds, don't make
it to fast or slow

        # set the goal state as succeeded
        goal_handle.succeed()

        # return the result
        result = Fibonacci.Result()
        result.result_sequence = feedback_msg.partial_sequence

        return result


def main(args=None):
    rclpy.init(args=args)
    node = FibonacciServerNode()
    rclpy.spin(node)
    rclpy.shutdown()


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

Analysis the code

```
self.action_server_ = ActionServer(
        self,
        Fibonacci,
        'fibonacci_action',
        execute_callback=self.execute_callback
    )
```

The action is instantiated, the name of the action defined, and an asynchronous callback used. Action servers can provide different callback arguments for each action functionality. But if no other action is present the goal with always be accepted.

```
# get goal request value
order = goal_handle.request.order
```

In the callback, we then get the goal message provided and wrap it in the goal handle

```
# initialize feedback
feedback_msg = Fibonacci.Feedback()
feedback_msg.partial_sequence = [0, 1]

# process Fibonacci
for i in range(1, order):

    feedback_msg.partial_sequence.append(feedback_msg.partial_sequence[i] +
feedback_msg.partial_sequence[i-1])

    self.get_logger().info(f'Feedback: {feedback_msg.partial_sequence}')

    goal_handle.publish_feedback(feedback_msg)

    time.sleep(1.0)

# set the goal state as succeeded
goal_handle.succeed()

# return the result
result = Fibonacci.Result()
result.result_sequence = feedback_msg.partial_sequence

return result
```

The custom function (fibonacci sequence), computes the fibonacci sequence and sets the goal state to *SUCCEDED* and returns the result message. The feedback message is also handled here.

Note: If we don't set the goal state to succeeded, it will be automatically considered ABORTED

Now we can test it.

Terminal 1

```
ros2 run fibonacci fibonacci_server
```

Terminal 2

We can emulate a client by sending a goal to the server.

```
ros2 action send_goal fibonacci_action custom_interface/action/Fibonacci "{
'order' : 5}"
```

And the expected results are:

Terminal 1

```
[INFO] [1756212574.826961373] [fibonacci_server_node]: executing the goal
[INFO] [1756212574.827832685] [fibonacci_server_node]: Feedback: array('q', [0, 1,
1])
[INFO] [1756212575.829941326] [fibonacci_server_node]: Feedback: array('q', [0, 1,
1, 2])
```

Terminal 2

```
[INFO] [1756212574.815798549] [fibonacci_client_node]: Fibonacci action client has
been started
[INFO] [1756212574.816422659] [fibonacci_client_node]: sending the goal: 3
[INFO] [1756212574.826942448] [fibonacci_client_node]: Goal accepted
[INFO] [1756212574.828570925] [fibonacci_client_node]: Received feedback:
array('q', [0, 1, 1])
[INFO] [1756212575.831563852] [fibonacci_client_node]: Received feedback:
array('q', [0, 1, 1, 2])
[INFO] [1756212578.839076475] [fibonacci_client_node]: result: array('q', [0, 1,
1, 2])
```

Client

Python script: *fibonacci_client.py*

In the *fibonacci_client.py* copy the following code.

```
import rclpy
from rclpy.node import Node
from custom_interface.action import Fibonacci
from rclpy.action import ActionClient
from rclpy.action.client import ClientGoalHandle

import time

class FibonacciClientNode(Node):

    def __init__(self):
        super().__init__('fibonacci_client_node')

        self.action_client_ = ActionClient(
            self,
            Fibonacci,
            '----' # TODO: fill with action name
        )

        # get goal parameter value
        # TODO: declare and set ROS parameter varlue for fibonacci order
        self.order == # TODO:


        self.declare_parameter('order', 5) # default 5
        self.order = self.get_parameter('order').value

        self.get_logger().info('Fibonacci action client has been started')


    def send_goal(self, order):
        """the action client sends the goal"""

        # wait for the server
        timeout_ = 1.0
        self.action_client_.wait_for_server(timeout_sec=timeout_)

        # create a goal
        goal = Fibonacci.Goal()
        goal.order = order

        # send the goal
        self.get_logger().info(f'sending the goal: {order}')

        # send_goal_async with feedback_callback. It returns future. when future
is returned, we can add a callback
        self.action_client_.send_goal_async(goal,
feedback_callback=self.goal_feedback_callback).add_done_callback(self.goal_respons
```

```
e_callback)


    def goal_feedback_callback(self, feedback_msg):
        """gets feedback message from feedback_callback"""

        feedback = feedback_msg.feedback
        self.get_logger().info(f'Received feedback: {feedback.partial_sequence}')


    def goal_response_callback(self, future):
        """checks if the goal request was accepted"""

        self.goal_handle_ : ClientGoalHandle = future.result()

        # checks if the goal has been accepted
        if not self.goal_handle_.accepted:
            self.get_logger().info('Goal rejected ')
            return

        # if the goal has been accepted we can request for the result
        self.get_logger().info('Goal accepted')


self.goal_handle_.get_result_async().add_done_callback(self.goal_result_callback)


    def goal_result_callback(self, future):
        """gets the result """
        result = future.result().result
        fibonacci_sequence = result.result_sequence

        self.get_logger().info(f'result: {fibonacci_sequence}')


def main(args=None):
    rclpy.init(args=args)
    node = FibonacciClientNode()

    # send goal
    node.send_goal(node.order)

    rclpy.spin(node)

    try:
        rclpy.shutdown()
    except:
        pass
```

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

Analysis the code

```
def __init__(self):
    super().__init__('fibonacci_client_node')

    self.action_client_ = ActionClient(
        self,
        Fibonacci,
        'fibonacci_action'
    )

    # get goal parameter value
    self.declare_parameter('order', 5) # default 5
    self.order = self.get_parameter('order').value

    self.get_logger().info('Fibonacci action client has been started')
```

It is the constructor of the class, where we instantiate the object *ActionClient*. We also declare and get the parameter order, which in this case is the length of the fibonacci sequence.

```
def send_goal(self, order):
    """the action client sends the goal"""

    # wait for the server
    timeout_ = 1.0
    self.action_client_.wait_for_server(timeout_sec=timeout_)

    # create a goal
    goal = Fibonacci.Goal()
    goal.order = order

    # send the goal
    self.get_logger().info(f'sending the goal: {order}')

    # send_goal_async with feedback_callback. It returns future. when future is
returned, we can add a callback
    self.action_client_.send_goal_async(goal,
feedback_callback=self.goal_feedback_callback).add_done_callback(self.goal_respons
e_callback)
```

Sends goal to the server after waiting for the server to be available. We create the goal object and perform the calling for both the feedback and the response.

Note: always use *send_goal_async* and not *send_goal* to ensure asynchronous communication between server and client.

```
def goal_response_callback(self, future):
    """checks if the goal request was accepted"""

    self.goal_handle_ : ClientGoalHandle = future.result()

    # checks if the goal has been accepted
    if not self.goal_handle_.accepted:
        self.get_logger().info('Goal rejected ')
        return

    # if the goal has been accepted we can request for the result
    self.get_logger().info('Goal accepted')


self.goal_handle_.get_result_async().add_done_callback(self.goal_result_callback)
```

This will check if the goal sent has been successfully received and accepted by the server. It will asynchronously call the *get_result* service. It introduces the add_done_callback to get the result.

```
def goal_result_callback(self, future):
    """gets the result """
    result = future.result().result
    fibonacci_sequence = result.result_sequence

    self.get_logger().info(f'result: {fibonacci_sequence}')
```

Gets the result of the action server, in this case the fibonacci sequence.

```
def goal_feedback_callback(self, feedback_msg):
    """gets feedback message from feedback_callback"""

    feedback = feedback_msg.feedback
    self.get_logger().info(f'Received feedback: {feedback.partial_sequence}')
```

This method is to extract the feedback message from the callback and log onto the screen.

```
def main(args=None):
    rclpy.init(args=args)
    node = FibonacciClientNode()

    # send goal
```

```
    node.send_goal(node.order)

    rclpy.spin(node)

    try:
        rclpy.shutdown()
    except:
        pass
```

The structure is as usual with the addition of calling the *send_node* method.

Now you should be able to run the server and client

Terminal 1

```
ros2 run fibonacci fibonacci_server
```

Terminal 2

```
ros2 run fibonacci fibonacci_client
```

Cancel the Goal

In actions it is possible to cancel the goal sent. To be able to do so create a copy of both the server and the client nodes so we can modify them.

Server

Python script: *fibonacci_server_cancel.py*

```
import rclpy
from rclpy.node import Node
from custom_interface.action import Fibonacci
from rclpy.action import ActionServer
from rclpy.action import CancelResponse
import time

from rclpy.executors import MultiThreadedExecutor, SingleThreadedExecutor

class FibonacciServerNode(Node):
    def __init__(self):
        super().__init__('fibonacci_server_node')

        self.action_server_ = ActionServer(self, Fibonacci, '----', # TODO: fill
with the action name
```

```
                                        execute_callback=self.execute_callback,
                                        cancel_callback=self.cancel_callback)

        self.get_logger().info('fibonacci action server has been started')


    def cancel_callback(self, cancel_request):
        """accepts or rejects a client request to cancel"""

        self.get_logger().info('received cancel request')
        return CancelResponse.ACCEPT


    def execute_callback(self, goal_handle):
        """execute the action and return the result"""

        self.get_logger().info(f'executing the goal ') #
{goal_handle.goal_id.uuid}')

        # get goal request value
        order = goal_handle.request.order

        # initialize feedback
        feedback_msg = Fibonacci.Feedback()
        feedback_msg.partial_sequence = [0, 1]

        # process Fibonacci or cancel if a cancel request is received
        for i in range(1, order):

            self.get_logger().info(str(goal_handle.is_cancel_requested))

            if goal_handle.is_cancel_requested:
                goal_handle.canceled()
                self.get_logger().info('-----') # TODO: Filled with a mesage for
the cancellation

                return Fibonacci.Result()

            feedback_msg.partial_sequence.append(feedback_msg.partial_sequence[i]
+ feedback_msg.partial_sequence[i-1])

            self.get_logger().info(f'Feedback: {feedback_msg.partial_sequence}')

            goal_handle.publish_feedback(feedback_msg)

            time.sleep(1.0)

        # set the goal state as succeeded
        goal_handle.succeed()

        # return the result
```

```
        result = Fibonacci.Result()
        result.result_sequence = feedback_msg.partial_sequence
        return result


def main(args=None):
    rclpy.init(args=args)
    node = FibonacciServerNode()

    executor = MultiThreadedExecutor()
    executor.add_node(node)

    try:
        executor.spin()
    except KeyboardInterrupt:
        print('KeyBoardInterrupt')


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

- In the main we use a *MultiThreadedExecutor* to spin the node. *MultiThreadedExecutor* is designed to execute callback functions concurrently in multiple threads.
- In the class constructor now, we added a *cancel_callback* to handle the cancel request
- The *cancel_callback* changes the *CancelRequest* status to accepted
- In the *execute_callback* while looping we check if the cancel request has been received and cancel accordingly.

Client

Python script: *fibonacci_client_cancel.py*

```
import rclpy
from rclpy.node import Node
from custom_interface.action import Fibonacci
from rclpy.action import ActionClient
from rclpy.action.client import ClientGoalHandle

import time

class FibonacciCancelClientNode(Node):
    def __init__(self):
        super().__init__('fibonacci_cancel_client_node')

        self.action_client_ = ActionClient(self, Fibonacci, '----') # TODO: fill
with action name

        # get goal parameter value
```

```
        # TODO: declare and set ROS parameter varlue for fibonacci order
        self.order == # TODO:

        self.get_logger().info('Fibonacci action client has been started')

    def send_goal(self):
        """ sends goal request """

        self.get_logger().info('Waiting for action server...')
        self.action_client_.wait_for_server()


        # create a goal
        goal = Fibonacci.Goal()
        goal.order = self.order_

        self.get_logger().info('Sending goal request...')


self.action_client_.send_goal_async(goal,feedback_callback=self.feedback_callback)
.add_done_callback(self.goal_response_callback)


    def feedback_callback(self, feedback):
        """ logs feedback response """
        self.get_logger().info('Received feedback:
{0}'.format(feedback.feedback.partial_sequence))


    def goal_response_callback(self, future):
        """ if the goal is accepted, starts a timer. After timeout, timer_callback
gets called"""

        goal_handle = future.result()

        if not goal_handle.accepted:
            self.get_logger().info('Goal rejected :(')
            return

        self._goal_handle = goal_handle
        self.get_logger().info('Goal accepted :)')

        # Start a 2 second timer
        self._timer = self.create_timer(--, self.timer_cancel_goal_callback) #
TODO: add time for when to cancel, consider the timing from the server


    def timer_cancel_goal_callback(self):
        """ sends cancel goal async request"""

        self.get_logger().info('----') # TODO: Assing a meessage for the
```

```
cancelation
        # Cancel the goal
        future =
self._goal_handle.cancel_goal_async().add_done_callback(self.cancel_done)

        # Cancel the timer
        self._timer.cancel()


    def cancel_done(self, future):
        """ receives cancel goal response and checks if it was successfull"""

        cancel_response = future.result()

        if len(cancel_response.goals_canceling) > 0:
            self.get_logger().info('----') # TODO: message for the goal
sucessfully canceled
        else:
            self.get_logger().info('----') # TODO: message for the goal failed to
cancel

        rclpy.shutdown()


def main(args=None):
    rclpy.init(args=args)

    node = FibonacciCancelClientNode()
    node.send_goal()

    rclpy.spin(node)
    rclpy.try_shutdown()

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

- We added time
- We create a timer in the *goal_response_callback* to do the cancel request
- In the *timer_cancel_goal_callback* we call the goal handler to send the cancel request.
- In *cancel_done* we check the response

Run both new server and client, you should expect the following output

Terminal 1

```
[INFO] [1756212765.063898553] [fibonacci_server_node]: fibonacci action server has
been started
[INFO] [1756212782.844181913] [fibonacci_server_node]: executing the goal
[INFO] [1756212782.844699108] [fibonacci_server_node]: False
```

```
[INFO] [1756212782.845130126] [fibonacci_server_node]: Feedback: array('q', [0, 1,
1])
[INFO] [1756212783.847238615] [fibonacci_server_node]: False
[INFO] [1756212783.847555474] [fibonacci_server_node]: Feedback: array('q', [0, 1,
1, 2])
[INFO] [1756212784.846249710] [fibonacci_server_node]: received cancel request
[INFO] [1756212784.849358943] [fibonacci_server_node]: True
[INFO] [1756212784.850261944] [fibonacci_server_node]: Goal canceled
```

Terminal 2

```
[INFO] [1756212782.841280329] [fibonacci_cancel_client_node]: Fibonacci action
client has been started
[INFO] [1756212782.841506282] [fibonacci_cancel_client_node]: Waiting for action
server...
[INFO] [1756212782.841726004] [fibonacci_cancel_client_node]: Sending goal
request...
[INFO] [1756212782.843326990] [fibonacci_cancel_client_node]: Goal accepted :
[INFO] [1756212782.845558399] [fibonacci_cancel_client_node]: Received feedback:
array('q', [0, 1, 1])
[INFO] [1756212783.848892077] [fibonacci_cancel_client_node]: Received feedback:
array('q', [0, 1, 1, 2])
[INFO] [1756212784.844174487] [fibonacci_cancel_client_node]: Canceling goal
[INFO] [1756212784.846925616] [fibonacci_cancel_client_node]: Goal successfully
canceled
```

TASK 5

For this task you need to write an Action Protocol in python to do a Countdown sequence with the format.

```
int32 start_from
---
string result_text
---
int32 current
```

- The action needs to receive a number to count down to zero from it while using parameters.
- Write a server-client pair to send a # number and count down from it.
- Write a seconde server-client pair that handles the cancellation after an specified amount of time.
- The client needs to send the goal, wait for a specified amount of time, set by a parameters and, then send a cancel request.
- Create a new package for this task and update as needed.

- Show the output of

```
ros2 interface show custom_interface/action/[new_action].
```

- Use parameters for sending the goal value and the time for the cancellation.

6.- Launch Files

A launch file is used to start multiple ROS2 nodes (or other processes) with specific configurations, in a single command. Instead of opening many terminals and typing *ros2 run* multiple times, you can write a Python launch file that does it all as well as set the parameters.

You can think of it as a component that runs multiple different nodes with different configurations on a single command.

You can write a launch file in three possible formats (Python, Xml, Yaml), for this lab we will be doing so in Python. This format allows for flexibility, modularity, error handling, and integration with ROS2 Python API. However, it is complex and can potentially add overhead in runtime interpretation.

Creation

It's good practice to create a new package in your workspace to contain all the Launch Files. The standard name for the package should indicate the project name and "*bringup*" as:

```
<workspace_name>_bringup
```

So let's create a basic package with that name.

```
ros2 pkg create --build-type ament_python <workspace>_bringup --license Apache-2.0
--dependencies rclpy
```

Note: The following instructions follow best practice and project organization, where the package only contains launch files. You can technically add the launch folder and dependencies in an existing packge.

Remove the */include* and */src* directories and create a new */launch* and *config/* directory inside your package for the structure:

```
ros2_ws/
└── src/
    └── [package_name]/
        ├── launch/
        ├── config/
        ├── CMakeLists.txt or setup.py
```

```
        ├── LICENSE
        ├── package.xml
```

CMakeLists or setup.py

Let's configure the package modifying the *CMakeLists.txt* file, so that the launch folder will be installed.

Add the following after *find_package*:

```
# install the launch directory --------------
install(DIRECTORY
launch
DESTINATION share/${PROJECT_NAME})
# ----------------------------------------

# install the config directory --------------
install(DIRECTORY
config
DESTINATION share/${PROJECT_NAME})
# ----------------------------------------
```

In case of a python package change the *setup.py*

```
data_files=[
    ('share/ament_index/resource_index/packages',
    ['resource/' + package_name]),
    ('share/' + package_name, ['package.xml']),
    # Include launch files
    (os.path.join('share', package_name, 'launch'), glob('launch/*.py')),
    # Include config files
    (os.path.join('share', package_name, 'config'), glob('config/*.yaml')),
],
```

And also:

```
install_requires=['setuptools', 'rclpy', 'launch', 'launch_ros', ….]
```

Package

We also need to modify the *package.xml* file, adding a standard syntax and the workspace packages where the nodes that will be launched are defined.

After <*buildtool_depend*>*ament_cmake*</*buildtool_depend*> add:

```
<!-- add packages dependencies-->
<exec_depend>py_pkg</exec_depend>
<exec_depend>cpp_pkg</exec_depend>
<exec_depend>launch</exec_depend>
<exec_depend>launch_ros</exec_depend>
<!-- -->
```

Launch file blueprint

Python script: *blueprint.launch.py*

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

# exactly this name!
def generate_launch_description():

    #instantiate a LaunchDescription object
    ld = LaunchDescription()

    topic_remap = ['', '']      # topic name, new topic name
    service_remap = ['', '']    # service name, new service name

    first_node = Node(

        package = '',     # package name

        executable = '',  # node's executable name

        name = '',        # if we want to remap the node name

        remappings = [topic_remap,
                      service_remap],     # if we want to remap topics or services
names

        output = '', # log = redirects the output to a log file based on log
level,
                     # screen = Explicitly prints subscriber output to the terminal

        # parameters=[{'log_level': 'WARN'}] # Set the log level as a parameter
        # OR
        # arguments=['--ros-args', '--log-level', 'simple_publisher:=WARN']

        parameters = [{'parameter1': '',
                      'parameter2': ''}]  # declare parameters of the node
<parameter_name> <default value>
                                          # (if the value is not string, omit quotes
    )
```

```
    second_node = Node(
        ## same structure
    )

    # add as much nodes needed for your application

    # launch description
    ld.add_action(first_node)
    ld.add_action(second_node)

    return ld
```

Tip: keep the blueprint and modify as needed for your projects

In specific for the talker and subscriber we created at the beginning:

```
first_node = Node(
        package='lab1',
        executable='publisher',
        name='simple_publisher',
        output = 'screen'
     )

 second_node = Node(
        package='lab1',
        executable='subscriber',
        name='simple_subscriber',
        output = 'screen'

    )
```

Build and source your workspace and run the launch file:

```
colcon build --packages-select [workspace]_bringup
source install/setup.bash
ros2 launch [workspace]_bringup [launch_file]
```

Printing (Optional)

You may have noticed that both nodes are showing their outputs in the same terminal, there are several ways to address this, one way is to replace Node with *ExecuteProcess*.

So out publisher command can look like this:

```
ExecuteProcess(
       condition=IfCondition(use_terminals),
       cmd=['gnome-terminal', '--', 'ros2', 'run', '[package], 'publisher'],
       name='publisher_terminal',
       output='screen'
   ),
```

You will also need to add:

```
from launch.actions import DeclareLaunchArgument, ExecuteProcess
```

So far so good, we are using *gnome-terminal* to launch our nodes directly. What if we wanted to decide when to use launch commands, to have them in separate terminals or one terminal? That's where *LaunchArguments* come in handy.

Let's change our imports to this:

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, ExecuteProcess
from launch_ros.actions import Node
from launch.substitutions import LaunchConfiguration
from launch.conditions import IfCondition, UnlessCondition
```

These let you parameterize your launch file. You define a launch argument called use_terminals for example and then use its value later in the script.

- *DeclareLaunchArgument* makes the argument available.
- *LaunchConfiguration('use_terminals')* fetches its value at runtime.

Python script: *printing.launch.py*

So, the code would look something like this:

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, ExecuteProcess
from launch_ros.actions import Node
from launch.substitutions import LaunchConfiguration
from launch.conditions import IfCondition, UnlessCondition


def generate_launch_description():
    use_terminals = LaunchConfiguration('use_terminals')
```

```
    return LaunchDescription([
        DeclareLaunchArgument(
            'use_terminals',
            default_value='0',
            description='Launch each node in a separate terminal (1=yes, 0=no)'
        ),


        ExecuteProcess(
            condition=IfCondition(use_terminals),
            cmd=['gnome-terminal', '--', 'ros2', 'run', 'lab1', 'publisher'],
            name='publisher_terminal',
            output='screen'
        ),


        Node(
            condition=UnlessCondition(use_terminals),
            package='lab1',
            executable='publisher',
            name='simple_publisher',
            output='screen'
        ),


        ExecuteProcess(
            condition=IfCondition(use_terminals),
            cmd=['gnome-terminal', '--', 'ros2', 'run', 'lab1', 'subscriber'],
            name='subscriber_terminal',
            output='screen'
        ),


        Node(
            condition=UnlessCondition(use_terminals),
            package='lab1',
            executable='subscriber',
            name='simple_subscriber',
            output='screen'
        )
    ])
```

Go ahead and build your package again, try to launch the launch file just like before, add this to the end of your launch command and see what happens: *use_terminals:=1*

```
ros2 launch lab1_bringup printing_launch.py use_terminals:=1
```

Two new terminals should open, one with the subscriber and one with the publisher.

Yaml in Launch File

You can use yaml file alongside launch files to configure it even more.

Python script: *params.launch.py*

An example with the *AddTwoInst* sever we used before is below:

```
import os
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory

def generate_launch_description():
    # Get the launch directory
    bringup_dir = get_package_share_directory('[]') # TODO: your package name for
launchs

    # Path to the parameter file
    param_file = LaunchConfiguration('params_file')

    # Declare the launch argument for parameter file
    declare_params_file_cmd = DeclareLaunchArgument(
        'params_file',
        default_value=os.path.join(bringup_dir, 'config', '[name].yaml'), # TODO:
your yaml file name goes here
        description='Full path to the ROS2 parameters file to use'
    )

    # Service server node
    service_server_node = Node(
        package='----', # TODO: your package name
        executable='----',  # TODO: Make sure this matches your setup.py entry
point
        name='sum_server',  # TODO:
        output='screen'
        # No parameters needed for server
    )

    # Service client node
    service_client_node = Node(
        package='----', # TODO: your package name
        executable='----',  # TODO: Make sure this matches your setup.py entry
point
        name='sum_client_node',  # TODO:
        output='screen',
        parameters=[param_file]
    )
```

```
    # Create the launch description and populate
    ld = LaunchDescription()

    # Add the commands to the launch description
    ld.add_action(declare_params_file_cmd)
    ld.add_action(service_server_node)
    ld.add_action(service_client_node)

    return ld
```

TASK 6

For this task you need to:

- Create a new package for the launch files
- Configure it to use launch files and configuration yaml files
- Create the launch file to use the server you created in Task 3 and set the parameters from a yaml file
- OPTIONAL 3: make the launch file launch two separate terminal for the server and the client.