Introduction to ROS2 Programming with Python

Ugo Albanese
Egidio Falotico

{e.falotico, u.albanese}@santannapisa.it

Outline

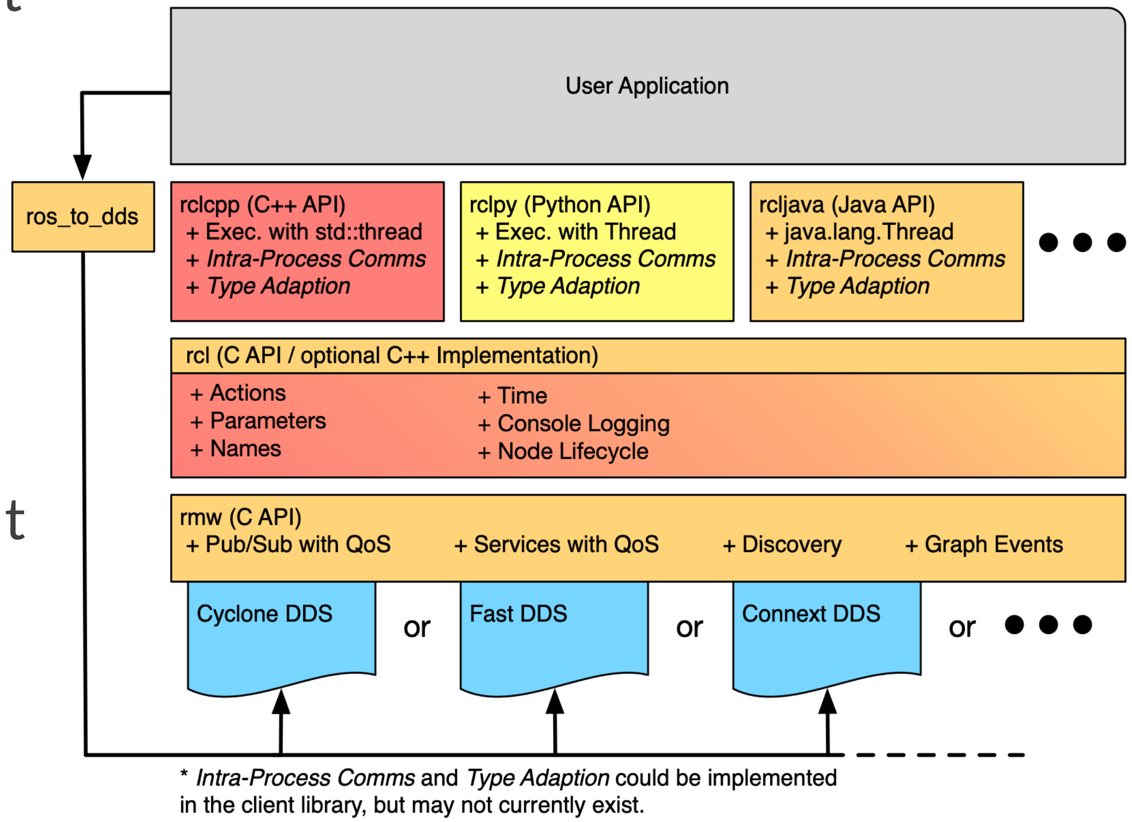
- Client Libraries
- Workspaces and packages
- Topics
- Services
- Custom msg and srv
- Parameters
- Actions
- Logging

ROS Client Libraries

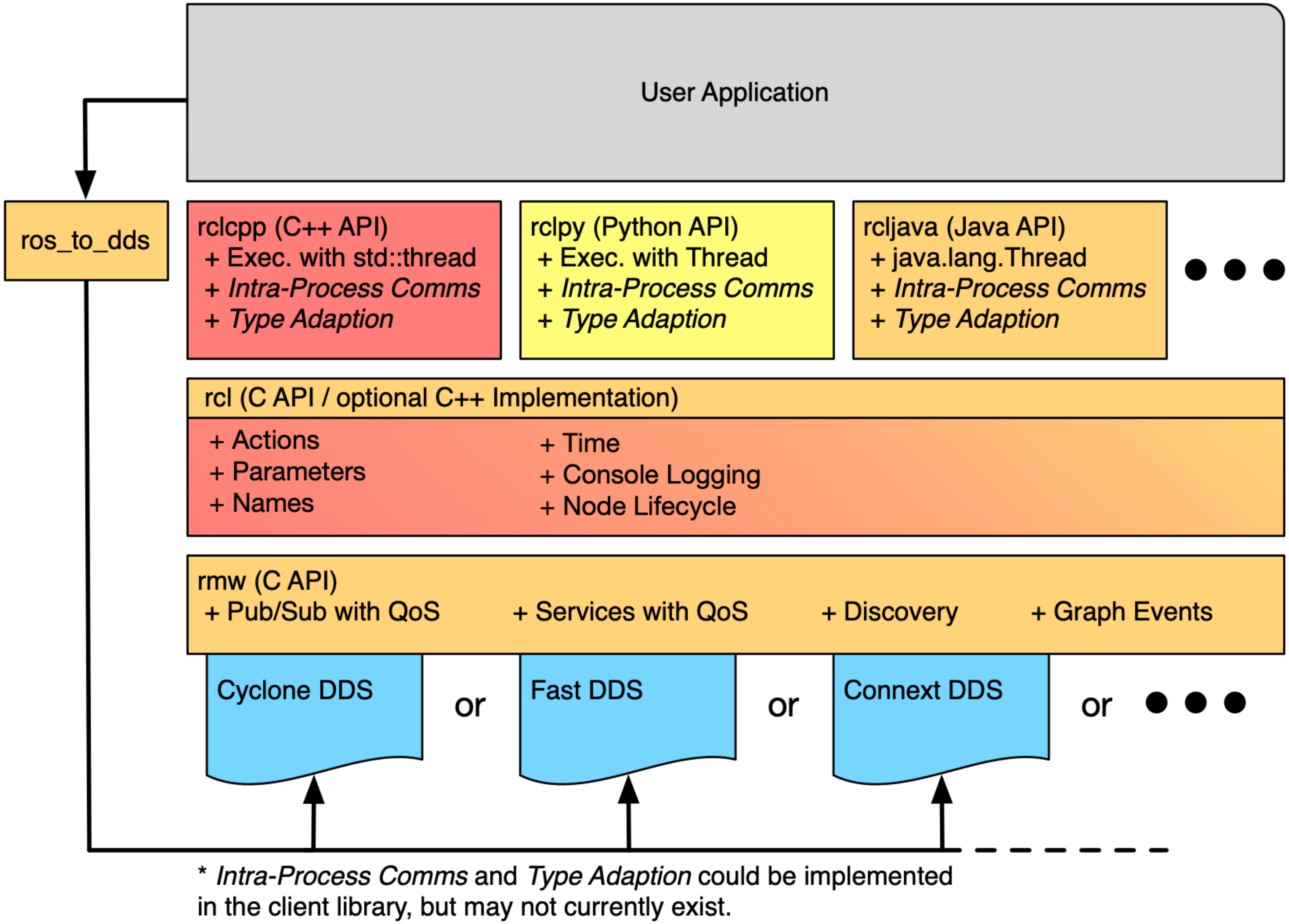
- *Client libraries* are the APIs that allow users to implement their ROS 2 code.
- Users gain access to ROS 2 concepts such as **nodes**, **topics**, **services**, etc.
- Come in a variety of programming languages so that users may write ROS 2 code in the language that is best-suited for their application.
- Nodes written using *different client libraries* are able to share messages with each other because all client libraries implement code generators that provide users with the capability to interact with ROS 2 interface files in the respective language.

reference

https://docs.ros.org/en/humble/Concepts/Basic/About-Client-Libraries.html



ROS Client Libraries – API Stack



S. Macenski, T. Foote, B. Gerkey, C. Lalancette, W. Woodall,
“Robot Operating System 2: Design, architecture, and uses in the wild”
Science Robotics vol. 7, May 2022.

ROS Client Libraries - `rclcpp`

- The ROS Client Library for C++ (`rclcpp`) is the user facing, C++ idiomatic interface which provides all of the ROS client functionality like creating nodes, publishers, and subscriptions.

- Builds on top of `rcl` and the `rosidl` API, and it is designed to be used with the C++ messages generated by rosidl_generator_cpp.

- Makes use of all the features of C++ and C++17 to make the interface as easy to use as possible

- Maintains a consistent behavior with the other client libraries that use the `rcl` API.

API docs

https://docs.ros.org/en/humble/p/rclcpp/

GitHub repo

https://github.com/ros2/rclcpp

ROS Client Libraries - `rclpy`

- The ROS Client Library for Python (`rclpy`) is the Python counterpart to the C++ client library.
- Also builds on top of the `rcl` C API for its implementation.
  - consistent with the other client libraries in terms of feature parity and behavior.
- Idiomatic Python experience that uses native Python types and patterns.

API docs

https://docs.ros.org/en/humble/p/rclpy/

GitHub repo

https://github.com/ros2/rclpy

ROS Client Libraries - `rclpy`

- It generates custom Python code for each ROS message that the user interacts with
  - eventually converts the native Python message object into the C version of the message.

- All operations happen on the Python version of the messages until they need to be passed into the `rcl` layer, at which point they are converted into the plain C version of the message so it can be passed into the `rcl` C API.

- This is avoided if possible when communicating between pubs and subs in the same process to cut down on the conversion into and out of Python.

API docs

https://docs.ros.org/en/humble/p/rclpy/

GitHub repo

https://github.com/ros2/rclpy

ROS Client Libraries
Community mantained

- Ada This is a set of packages (binding to `rcl`, message generator, binding to `tf2`, examples and tutorials) that allows the writing of Ada applications for ROS 2.

- C `rclc` does not put a layer on top of `rcl` but complements `rcl` to make `rcl+rclc` a feature-complete client library in C. See micro.ros.org for tutorials.

- JVM and Android Java and Android bindings for ROS 2.

- .NET Core, UWP and C# This is a collection of projects (bindings, code generator, examples and more) for writing ROS 2 applications for .NET Core and .NET Standard.

- Node.js `rclnodejs` is a Node.js client for ROS 2. It provides a simple and easy JavaScript API for ROS 2 programming.

- Rust This is a set of projects (the `rclrs` client library, code generator, examples and more) that enables developers to write ROS 2 applications in Rust.

reference

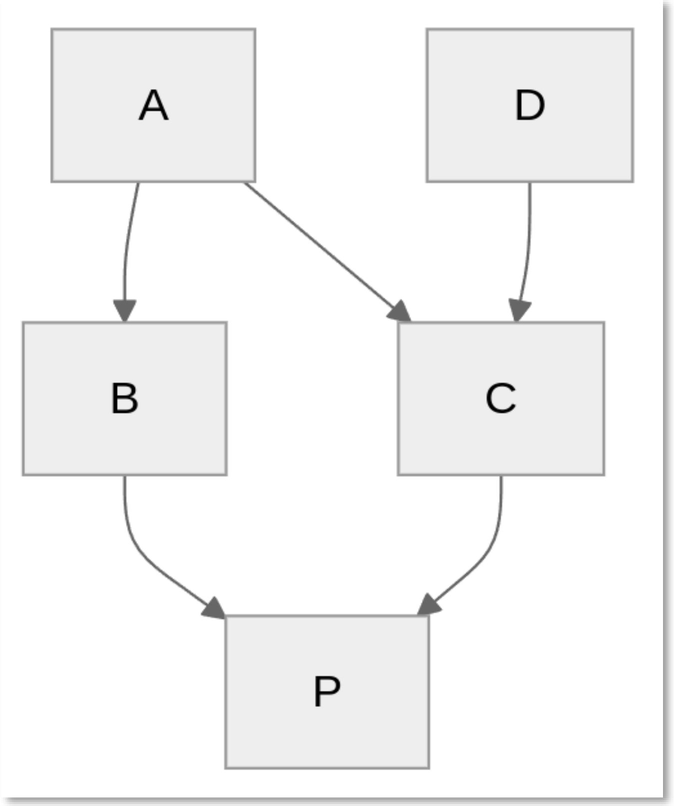
https://docs.ros.org/en/humble/Concepts/Basic/About-Client-Libraries.html#community-maintained

ROS2 build tool

- In the ROS ecosystem the software is separated into numerous packages.
- It is very common that a developer is working on multiple packages at the same time.
- The “manual” approach to build a set of packages consists of building all packages in their [topological order](https://en.wikipedia.org/wiki/Topological_sorting) one by one. Such a workflow is impracticable at scale without a tool that automates that process.
- A build *tool* performs the task of building a set of packages with a single invocation.
- A build *tool* determines the dependency graph and invokes the specific build *system* for each package in topological order.

Topological Order

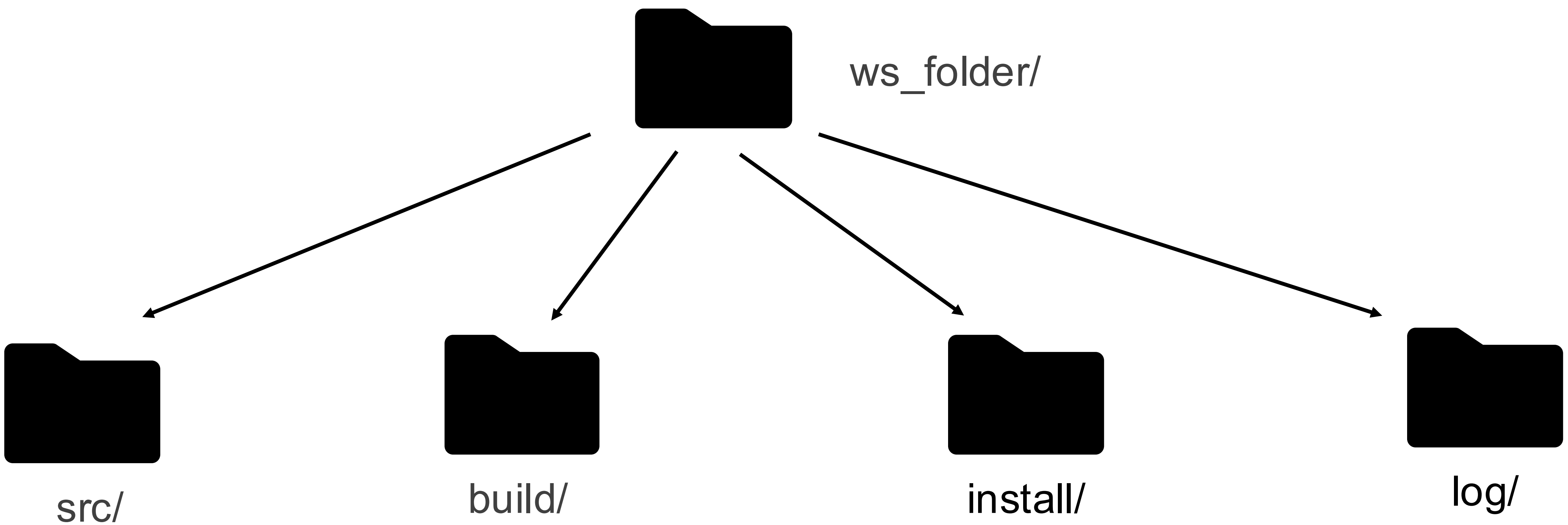
https://en.wikipedia.org/wiki/Topological_sorting



ROS2 Workspace

A ROS workspace is a directory with a stardard structure.

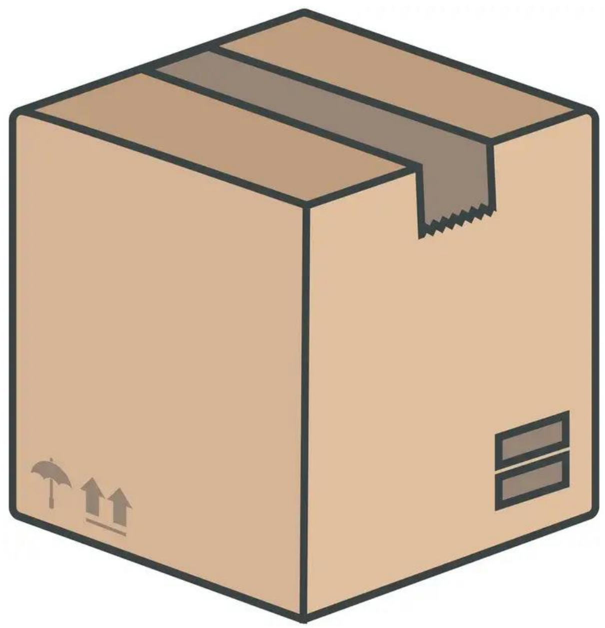
Where to create and build ROS packages.



Where the source code of ROS packages is located.	Where intermediate files are stored. For each package a subfolder will be created in which e.g. CMake is being invoked.	Where each package will be installed to. By default each package will be installed into a separate subdirectory.	Contains various logging information about each colcon invocation

ROS2 packages

- A package is an organizational unit for your ROS 2 code.
- If you want to be able to install your code or share it with others, then it has to be organized in a package.
- With packages, you can release your ROS 2 work and allow others to build and use it easily.
- Package creation in ROS 2 uses `ament` as its build system and `colcon` as its build tool.



ROS2 packages

- ROS 2 **Python** packages have the following required contents:
  - **`package.xml`** file containing meta information about the package
  - **`resource/<package_name>`** marker file for the package
  - **`setup.cfg`** is required when a package has executables, so `ros2 run` can find them
  - **`setup.py`** containing instructions for how to install the package
  - **`<package_name>`** a directory with the same name as your package, used by ROS 2 tools to find your package, contains `__init__.py`

```
my_package/
      package.xml
      resource/my_package
      setup.cfg
      setup.py
      my_package/
```

`package.xml`
Format specs v3

https://ros.org/reps/rep-0149.html

Setup a workspace and create a package

- From terminal:

```
# install colcon (one time only)
$ sudo apt install python3-colcon-common-extensions
# create directories
$ mkdir -p ~/ros2_ws/src
$ cd ~/ros2_ws
$ colcon build # init ws, call from the ws root!
```

- Create the package:

```
$ cd ~/ros2_ws/src
$ ros2 pkg create --build-type ament_python --
license Apache-2.0 --node-name my_node my_package
```

- Build the workspace:

```
$ cd ~/ros2_ws
$ colcon build
```

- Enable the workspace in the current terminal with:

```
$ source ~/ros2_ws/install/setup.bash
```

`APT - Advanced package tool`

https://wiki.debian.org/AptCLI

`source`

A bash shell built-in command that executes the content of the file passed as an argument, ***in the current shell***.
It has a synonym in . (period).

Setup a workspace and create a package

- Before using ROS 2, it's necessary to **source** your ROS 2 installation workspace in the terminal you plan to work in. This makes ROS 2's packages available for you to use in that terminal.

- You also have the option of sourcing an "***overlay***"
  - a secondary workspace where you can add new packages without interfering with the existing ROS 2 workspace that you're extending, i.e. the "***underlay***".

- Your *underlay* must contain the dependencies of **all** the packages in your overlay. Packages in your overlay will **override** packages in the underlay.

- It's also possible to have *several layers* of underlays and overlays, with each successive overlay using the packages of its parent underlays.

NOTE (reference)

- Sourcing the `local_setup` of the overlay will only add the packages available in the overlay to your environment. `setup` sources the overlay *as well as the underlay it was created in*, allowing you to utilize both workspaces.

- So, sourcing your main ROS 2 installation's setup and then the `ros2_ws` overlay's local_setup, like you just did, is the same as just sourcing `ros2_ws`'s `setup`, because that includes the environment of its underlay.

Setup a workspace and create a package

- Create the package:

```
$ ros2 pkg create --build-type ament_python --license Apache-2.0 --node-name my_node my_package
going to create a new package
package name: my_package
destination directory: /home/brairlab/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['brairlab <brairlab@todo.todo>']
licenses: ['Apache-2.0']
build type: ament_python
dependencies: []
node_name: my_node
creating folder ./my_package
creating ./my_package/package.xml
creating source folder
creating folder ./my_package/my_package
creating ./my_package/setup.py
creating ./my_package/setup.cfg
creating folder ./my_package/resource
creating ./my_package/resource/my_package
creating ./my_package/my_package/__init__.py
creating folder ./my_package/test
creating ./my_package/test/test_copyright.py
creating ./my_package/test/test_flake8.py
creating ./my_package/test/test_pep257.py
creating ./my_package/my_package/my_node.py
```

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>my_package</name>
 <version>0.0.0</version>
 <description>TODO: Package description</description>
 <maintainer email="user@todo.todo">user</maintainer>
 <license>TODO: License declaration</license>

 <test_depend>ament_copyright</test_depenad>
 <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>
```

setup.py

```
from setuptools import find_packages, setup

package_name = 'my_package'

setup(
 name=package_name,
 version='0.0.0',
 packages=find_packages(exclude=['test']),
 data_files=[
  ('share/ament_index/resource_index/packages',
   [f'resource/{package_name}']),
  (f'share/{package_name}', ['package.xml']),
 ],
 install_requires=['setuptools'],
 zip_safe=True,
 maintainer='brairlab',
 maintainer_email='brairlab@todo.todo',
 description='TODO: Package description',
 license='Apache-2.0',
 tests_require=['pytest'],
 entry_points={
  'console_scripts': [
  'my_node = my_package.my_node:main'
  ],
 },
)
```

- Contains the same description, maintainer and license fields as `package.xml`
- They need to match exactly in both files.
- The version and name (`package_name`) also need to match exactly and should be (automatically) populated in both files.

Python Packaging

https://packaging.python.org/

Managing Dependencies
rosdep

- A dependency management utility that can work with packages and external libraries.
- It is a command-line utility for identifying and installing dependencies to build or install a package.
- A meta-package manager that uses its own knowledge of the system and the dependencies to find the appropriate package to install on a particular platform.
- The actual installation is done using the system package manager (e.g. `apt` on Debian/Ubuntu, `dnf` on Fedora/RHEL, etc).
- It is most often invoked before building a workspace, where it is used to install the dependencies of the packages within that workspace.

```
# one time installation and init
$ sudo apt-get install python3-rosdep
$ sudo rosdep init
$ rosdep update # update locally cached rosdistro index
```

APT - Advanced package tool

https://wiki.debian.org/AptCLI

DNF

https://dnf.readthedocs.io/

Managing Dependencies
declaring dependencies in `package.xml`

- `rosdep` finds the set of dependencies listed in `package.xml`.

- It is important that the list of dependencies in the `package.xml` is complete and correct, which allows all of the tooling to determine the packages dependencies.

- Missing or incorrect dependencies can lead to users not being able to use your package, to packages in a workspace being built out-of-order, and to packages not being able to be released.

- Dependencies in the `package.xml` file are generally referred to as "**`rosdep keys`**".

- These dependencies are *manually* populated in the `package.xml` file by the package's creators (yes, YOU) and should be an exhaustive list of any non-builtin libraries and packages it requires.

- These are represented in the tags specified by REP-149.

`package.xml`
Dependecy tags

https://ros.org/reps/rep-0149.html#dependency-tags

Managing Dependencies
dependency tags in `package.xml`

- `<depend>`
  - These are dependencies that should be provided at both build time and run time for your package. For C++ packages, if in doubt, use this tag. Pure Python packages *generally* don't have a build phase, so should never use this and should use `<exec_depend>` instead.

- `<build_export_depend>`
  - If you export a header that includes a header from a dependency, it will be needed by other packages that `<build_depend>` on yours. This mainly applies to headers and CMake configuration files. Library packages referenced by libraries you export should normally specify `<depend>`, because they are also needed at execution time.

- `<exec_depend>`
  - This tag declares dependencies for shared libraries, executables, Python modules, launch scripts and other files required when running your package.

- `<test_depend>`
  - This tag declares dependencies needed only by tests. Dependencies here should *not* be duplicated with keys specified by `<build_depend>`, `<exec_depend>`, or `<depend>`.

Managing Dependencies
how rosdep works

- rosdep will check for package.xml files in its path or for a specific package and find the rosdep keys stored within.
- These keys are then cross-referenced against a central index to find the appropriate ROS package or software library in various package managers.
- Finally, once the packages are found, they are installed.

- rosdep works by retrieving the central index on to your local machine so that it doesn't have to access the network every time it runs (on Debian/Ubuntu the configuration for it is stored in /etc/ros/rosdep/sources.list.d/20-default.list).

- The central index is known as rosdistro, which may be found online.

Rosdistro central index

https://github.com/ros/rosdistro

Managing Dependencies
Where to find packages keys and usage

- If the package you want to depend in your package is ROS-based, AND has been released into the ROS ecosystem (i.e. the package is listed in one or more directories in the rosdistro database) e.g. `nav2_bt_navigator`, you may simply use the name of the package.
  - You can find a list of all released ROS packages in rosdistro at `<distro>/distribution.yaml` (e.g. `humble/distribution.yaml`) for your given ROS distribution.

- If you want to depend on a non-ROS package, something often called "*system dependencies*", you will need to find the keys for a particular library.

- In general, there are two files of interest:
  - rosdep/base.yaml contains the `apt` system dependencies
  - rosdep/python.yaml contains the **Python dependencies**
  - To find a key, search for your library in these files and find the name. This is the key to put in a `package.xml` file.

```
# Install dependencies of packages in workspace ros2_ws
$ cd ros2_ws/
$ rosdep install
```

Topic publisher Node
Create package

Let's create a package for ROS node that publishes a String to a Topic at a fixed rate.

```
$ cd ~/ros2_ws/src
$ ros2 pkg create --build-type ament_python --license Apache-2.0
py_pubsub
```

Your terminal will return a message verifying the creation of your package `py_pubsub` and all its necessary files and folders.

Topic publisher Node

Let's create a ROS node that publishes a String to a Topic at a fixed rate.

Create and edit publisher_member_function.py

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import String

...

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

    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)

    # Destroy the node explicitly
    # (optional, otherwise garbage collected)
    minimal_publisher.destroy_node()
    rclpy.shutdown()

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

CAUTION!

Copy pasting the code may not work

Topic publisher Node

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

The first lines import `rclpy` so its `Node` class can be used.

```
from std_msgs.msg import String
```

The next statement imports the built-in string message type that the node uses to structure the data that it passes on the topic.
These lines represent the node's dependencies. Recall that dependencies have to be added to *package.xml*.

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

    # Destroy the node explicitly (optional, otherwise garbage collected)
    minimal_publisher.destroy_node()
    rclpy.shutdown()

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

Node Initialization, Shutdown, and Spinning

https://docs.ros2.org/latest/api/rclpy/api/init_shutdown.html

First the `rclpy` library is initialized, then the node is created, and then it "spins" the node so its callbacks are called.

Topic publisher Node

```python
class MinimalPublisher(Node):

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

        self.publisher_ = self.create_publisher(String, 'topic', 10)

        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)

        self.i = 0

    def timer_callback(self):
        msg = String() # or String(data="some_string")
        msg.data = f'Hello World: {self.i}'

        self.publisher_.publish(msg)

        self.get_logger().info(f'Publishing: "{msg.data}"')

        self.i += 1
```

Topic publisher Node

```
class MinimalPublisher(Node):
```

The MinimalPublisher class is created, which inherits from (or is a subclass of) Node

```
super().__init__('minimal_publisher')
```

The definition of the class's constructor. super().__init__ calls the Node class's constructor and gives it your node name, in this case minimal_publisher.

```
self.publisher_ = self.create_publisher(String, 'topic', 10)
```

create_publisher declares that the node publishes messages of type String (imported from the std_msgs.msg module), over a topic named topic, and that the "queue size" is 10.
Queue size is a required QoS (quality of service) setting that limits the amount of queued messages if a subscriber is not receiving them fast enough.

Node API

https://docs.ros2.org/latest/api/rclpy/api/node.html

Topic publisher Node

```
self.timer = self.create_timer(timer_period, self.timer_callback)
```

A timer is created with a callback to execute every 0.5 seconds. `self.i` is a counter used in the callback.

```
def timer_callback(self):
    msg = String()
    msg.data = f'Hello World: {self.i}'

    self.publisher_.publish(msg)

    self.get_logger().info(f'Publishing: "{msg.data}"')

    self.i += 1
```

`timer_callback` creates a message with the counter value appended, and publishes it to the console with `get_logger().info`.

Logger API

https://docs.ros2.org/latest/api/rclpy/api/logging.html

Topic publisher Node
Add dependencies

```xml
<description>Examples of minimal publisher/subscriber using rclpy</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache-2.0</license>
```

Edit package.xml adding the following dependencies corresponding to your node's import statements

```xml
<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

This declares the package needs `rclpy` and `std_msgs` when its code is executed (`<exec_depend>`).

Topic publisher Node
Add entry point in `setup.py`

```
maintainer='YourName',
maintainer_email='you@email.com',
description='Examples of minimal publisher/subscriber using rclpy',
license='Apache-2.0',
```

Add the following line within the `console_scripts` brackets of the `entry_points` field:

```
entry_points={
        'console_scripts': [
                'talker = py_pubsub.publisher_member_function:main',
        ],
},
```

This declares the package needs `rclpy` and `std_msgs` when its code is executed (**`<exec_depend>`**).

Topic publisher Node
Check `setup.cfg` and build

```
[develop]
script_dir=$base/lib/py_pubsub
[install]
install_scripts=$base/lib/py_pubsub
```

This is simply telling setuptools to put your executables in `lib`, because `ros2 run` will look for them there.

```
$ cd ~/ros2_ws
$ colcon build

$ source ~/ros2_ws/install/setup.bash
```

Build your package, source the local setup files

Topic subscriber Node

Let's create a ROS node that subscribes to a Topic

Create and edit `subscriber_member_function.py`

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import String


...

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

    minimal_subscriber = MinimalSubscriber()

    rclpy.spin(minimal_subscriber)

    # Destroy the node explicitly
    minimal_subscriber.destroy_node()
    rclpy.shutdown()

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

Topic subscriber Node

```python
class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            'topic',
            self.listener_callback,
            10)

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

Topic subscriber Node

```
self.subscription = self.create_subscription(
    String,
    'topic',
    self.listener_callback,
    10) # 10 messages-deep history
```

Create a subscriber with the same arguments as the publisher.

Recall that the topic name and message type used by the publisher and subscriber must match to allow them to communicate.
The subscriber’s constructor and callback don’t include any timer definition, because it doesn’t need one. **Its callback gets called as soon as it receives a message.**

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

The callback definition simply prints an info message to the console.

Node API

https://docs.ros2.org/latest/api/rclpy/api/node.html

Topic subscriber Node
Add entry point in `setup.py` and build

```
entry_points={
        'console_scripts': [
                'talker = py_pubsub.publisher_member_function:main',
                'listener = py_pubsub.subscriber_member_function:main',
        ],
},
```

```
$ cd ~/ros2_ws
$ colcon build

$ source ~/ros2_ws/install/setup.bash
```

- Build your package and source the setup file

There are several ways you could write a publisher and subscriber in Python; check out the `minimal_publisher` and `minimal_subscriber` packages in the ros2/examples repo.

Run the example (`ros2 launch`)

1. Create `pubsub_launch.xml` (`_launch` suffix and xml extension is customary)

```
<launch>
    <node name="listener_node" pkg="py_pubsub" exec="listener" output="screen"/>
    <node name="talker_node" pkg="py_pubsub" exec="talker" output="screen"/>
</launch>
```

2. Edit `setup.py` to enable colcon to locate and utilize launch files

```
import os
from glob import glob

# Other imports ...

package_name = 'py_pubsub'

setup(
    # Other parameters ...
    data_files=[
        # ... Other data files
        # Include all launch files.
        (os.path.join('share', package_name, 'launch'),
         glob(os.path.join('launch', '*launch.[pxy][yma]*'))
        ) # assumes _launch suffix
    ]
)
```

Run the example (`ros2 launch`)

3. Run `ros2 launch py_pubsub pubsub.xml`

```
brairlab@brairlab-vm:~$ ros2 launch py_pubsub pubsub_launch.xml
[INFO] [launch]: All log files can be found below /home/brairlab/.ros/log/2024-10-08-
16-15-29-916156-brairlab-vm-10499
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [listener-1]: process started with pid [10500]
[INFO] [talker-2]: process started with pid [10502]
[talker-2] [INFO] [1728396930.709366932] [talker_node]: Publishing: "Hello World: 0"
[listener-1] [INFO] [1728396930.721843194] [listener_node]: I heard: "Hello World: 0"
[talker-2] [INFO] [1728396931.195176739] [talker_node]: Publishing: "Hello World: 1"
[listener-1] [INFO] [1728396931.196069809] [listener_node]: I heard: "Hello World: 1"
[talker-2] [INFO] [1728396931.850646196] [talker_node]: Publishing: "Hello World: 2"
[listener-1] [INFO] [1728396931.851639796] [listener_node]: I heard: "Hello World: 2"
[talker-2] [INFO] [1728396932.194022204] [talker_node]: Publishing: "Hello World: 3"
[listener-1] [INFO] [1728396932.195033405] [listener_node]: I heard: "Hello World: 3"
[talker-2] [INFO] [1728396932.697243304] [talker_node]: Publishing: "Hello World: 4"
[listener-1] [INFO] [1728396932.697645178] [listener_node]: I heard: "Hello World: 4"
[talker-2] [INFO] [1728396933.194441203] [talker_node]: Publishing: "Hello World: 5"
[listener-1] [INFO] [1728396933.194886941] [listener_node]: I heard: "Hello World: 5"
```

Service server Node
Create package

Let's create a package for ROS service server node implementing a service to add two integers.

```
$ cd ~/ros2_ws/src
$ ros2 pkg create --build-type ament_python --license Apache-2.0
py_srvcli --dependencies rclpy example_interfaces
```

The `--dependencies` argument will automatically add the necessary dependency lines to `package.xml`. `example_interfaces` is the package that includes the .srv file you will need to structure your requests and responses:

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

Service server Node
Update package.xml and setup.py

```
<description>Python client server tutorial</description>
<maintainer email="brairlab@todo.todo">brairlab</maintainer>
<license>Apache-2.0</license>
```

Because you used the `--dependencies` option during package creation, you don't have to manually add dependencies

```
<exec_depend>rclpy</exec_depend>
<exec_depend>example_interfaces</exec_depend>
```

Update `setup.py` matching the info contained in `package.xml`

```
maintainer='Your Name',
maintainer_email='brairlab@todo.todo',
description='Python client server tutorial',
license='Apache License 2.0',
```

Service server Node

Create and edit `service_member_function.py` in the `ros2_ws/src/py_srvcli/py_srvcli` directory

```
from example_interfaces.srv import AddTwoInts

import rclpy
from rclpy.node import Node

...

def main():
    rclpy.init()

    minimal_service = MinimalService()

    rclpy.spin(minimal_service)

    rclpy.shutdown()


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

CAUTION!

Copy pasting the code may not work

Service server Node

```python
class MinimalService(Node):

    def __init__(self):
        super().__init__('minimal_service')
        self.srv = self.create_service(AddTwoInts, 'add_two_ints', self.add_two_ints_clbck)

    def add_two_ints_clbck(self, request, response):
        response.sum = request.a + request.b
        self.get_logger().info(f'Incoming request\na: {request.a} b: {request.b}')

        return response
```

- The `MinimalService` class constructor initializes the node with the name `minimal_service`. Then, it creates a service and defines the **type**, **name**, and **callback**.

- The definition of the service callback receives the request data, sums it, and returns the sum as a response.

Service server Node

```python
from example_interfaces.srv import AddTwoInts

import rclpy
from rclpy.node import Node
```

The first import statement imports the AddTwoInts service type from the example_interfaces package. The following import statement imports the ROS 2 Python client library, and specifically the Node class.


```python
from example_interfaces.srv import AddTwoInts

import rclpy
from rclpy.node import Node
```

Service server Node
Add entry point in `setup.py`

To allow the `ros2 run` command to run your node, you must add the entry point to `setup.py` (located in the `ros2_ws/src/py_srvcli` directory).

```
entry_points={
        'console_scripts': [
                'service = py_srvcli.service_member_function:main',
        ],
},
```

Service client Node

Create and edit `client_member_function.py` in the `ros2_ws/src/py_srvcli/py_srvcli` directory

```
import sys

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

...

def main():
    rclpy.init()

    minimal_client = MinimalClientAsync()

    future = minimal_client.send_request(int(sys.argv[1]), int(sys.argv[2]))
    rclpy.spin_until_future_complete(minimal_client, future)
    response = future.result()

    minimal_client.get_logger().info(
        'Result of add_two_ints: for {int(sys.argv[1])} + {int(sys.argv[2])} = {response.sum}')

    minimal_client.destroy_node()
    rclpy.shutdown()

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

Security risk!

Input data must be validated before use

CAUTION!

Copy pasting the code may not work

Service client Node

```python
class MinimalClientAsync(Node):

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

        self.cli = self.create_client(AddTwoInts, 'add_two_ints')

        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service not available, waiting again...')

        self.req = AddTwoInts.Request()

    def send_request(self, a, b):
        self.req.a = a
        self.req.b = b
        return self.cli.call_async(self.req)
```

Service client Node

```python
def __init__(self):
    super().__init__('minimal_client_async')
    self.cli = self.create_client(AddTwoInts, 'add_two_ints')

    while not self.cli.wait_for_service(timeout_sec=1.0):
        self.get_logger().info('service not available, waiting again...')

    self.req = AddTwoInts.Request()
```

- The `MinimalClientAsync` class constructor initializes the node with the name `minimal_client_async`.
- The constructor definition creates a client with the same type and name as the service node.
- The **type** and **name** must **match** for the client and service to be able to communicate.
- The `while` loop in the constructor checks if a service matching the type and name of the client is available once a second. Finally it creates a new `AddTwoInts` request object.

Service client Node

```python
def send_request(self, a, b):
    self.req.a = a
    self.req.b = b
    return self.cli.call_async(self.req)
```

- The `send_request` method, will send the request (**asynchronously**) and return a future that can be passed to `spin_until_future_complete`

```python
def main():
    rclpy.init()
    minimal_client = MinimalClientAsync()

    future = minimal_client.send_request(int(sys.argv[1]), int(sys.argv[2]))
    rclpy.spin_until_future_complete(minimal_client, future)

    response = future.result()
```

- The `main` method, which constructs a `MinimalClientAsync` object, sends the request using the passed-in command-line arguments, calls `spin_until_future_complete`, and logs the results.

Service client Node
Add entry point in `setup.py` and build

To allow the `ros2 run` command to run your node, you must add the entry point to `setup.py` (located in the `ros2_ws/src/py_srvcli` directory).

```
entry_points={
        'console_scripts': [
                'service = py_srvcli.publisher_member_function:main',
                'client = py_srvcli.client_member_function:main',
        ],
},
```

```
$ cd ~/ros2_ws
$ colcon build --packages-select py_srvcli

$ source ~/ros2_ws/install/setup.bash
```

- Build (only) your package and source the setup file

Run the service

1. Run `ros2 run py_pysrvcli client`

```
brairlab@brairlab-vm:~$ ros2 run py_srvcli client 2 3
[INFO] [1728469999.404540525] [minimal_client_async]: service not available, waiting again...
[INFO] [1728470000.248217108] [minimal_client_async]: Result of add_two_ints: for 2 + 3 = 5
```

2. In an **different** terminal, source the setup file
3. Run `ros2 run py_pysrvcli service`

```
brairlab@brairlab-vm:~$ ros2 run py_srvcli service
[INFO] [1728470000.246944085] [minimal_service]: Incoming request
a: 2 b: 3
```

There are several ways you could write a service and client in Python; check out the `minimal_client` and `minimal_service` packages in the ros2/examples repo.

Service clients
Synchronous vs. Asynchronous

- *ROS1* Services are *synchronous*, i.e. the thread of a client node issuing a request to a service server blocks until a response is (eventually) received.
- This behavior can be replicated in *ROS2* using `rclpy.client.Client` method `call()`, its usage is discouraged because it can cause a deadlock if not properly used.

- Async calls in `rclpy` are entirely safe and the recommended method of calling services. They can be made from anywhere without running the risk of blocking other ROS and non-ROS processes, unlike sync calls.
- An asynchronous client will immediately return `future`, a value that indicates whether the call and response is finished (not the value of the response itself), after sending a request to a service. The returned `future` may be queried for a response at any time.

- Since sending a request doesn't block anything, a loop can be used to both spin `rclpy` and check `future` in the same thread, for example:

```
while rclpy.ok():
    rclpy.spin_once(node) # Execute one item of work (callbacks)

    if future.done():
        # Get response
```

reference

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

Custom messages and services
Create package

Let's create custom `.msg` and `.srv` files in their own package (`tutorial_interfaces`), and then utilizing them in a separate package. Both packages should be in the same workspace.

```
$ cd ~/ros2_ws/src
$ ros2 pkg create --build-type ament_cmake --license Apache-2.0 tutorial_interfaces
```

A *CMake* package it's required (`ament_cmake` build type), but this doesn't restrict in which type of packages you can use your messages and services. You can create your own custom interfaces in a *CMake* package, and then use it in a C++ or Python node

The `.msg` and `.srv` files are required to be placed in directories called `msg` and `srv` respectively.

Create the directories in `ros2_ws/src/tutorial_interfaces`:

```
$ mkdir msg srv
```

Custom messages and services
msg definition

In the tutorial_interfaces/msg directory you just created, make a new file called Num.msg with one line of code declaring its data structure:

```
int64 num
```

This is a custom message that transfers a single 64-bit integer called num.

Also in the tutorial_interfaces/msg directory you just created, make a new file called Sphere.msg with the following content:

```
geometry_msgs/Point center
float64 radius
```

This custom message uses a message from another message package (geometry_msgs/Point in this case).

Custom messages and services
srv definition

In the tutorial_interfaces/srv directory, make a new file called AddThreeInts.srv with the following request and response structure:

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

This is your custom service that requests three integers named a, b, and c, and responds with an integer called sum.

ROS built-in types supported

Type name	C++	Python	DDS type
bool	bool	builtins.bool	boolean
byte	uint8_t	builtins.bytes*	octet
char	char	builtins.int*	char
float32	float	builtins.float*	float
float64	double	builtins.float*	double
int8	int8_t	builtins.int*	octet
uint8	uint8_t	builtins.int*	octet
int16	int16_t	builtins.int*	short
uint16	uint16_t	builtins.int*	unsigned short
int32	int32_t	builtins.int*	long
uint32	uint32_t	builtins.int*	unsigned long
int64	int64_t	builtins.int*	long long
uint64	uint64_t	builtins.int*	unsigned long long
string	std::string	builtins.str	string
wstring	std::u16string	builtins.str	wstring

reference

https://design.ros2.org/articles/idl_interface_definition.html

ROS built-in types supported - arrays

Type name	C++	Python	DDS type
static array	std::array<T, N>	builtins.list*	T[N]
unbounded dynamic array	std::vector	builtins.list	sequence
bounded dynamic array	custom_class<T, N>	builtins.list*	sequence<T, N>
bounded string	std::string	builtins.str*	string

reference

https://design.ros2.org/articles/idl_interface_definition.html

Custom messages and services
`CMakeLists.txt`

To convert the interfaces you defined into language-specific code (like C++ and Python) so that they can be used in those languages, add the following lines to `CMakeLists.txt`

```
find_package(geometry_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/Num.msg"
  "msg/Sphere.msg"
  "srv/AddThreeInts.srv"
  DEPENDENCIES geometry_msgs # Add packages that above messages depend on, in this case geometry_msgs for Sphere.msg
)
```

NOTE: The first argument (library name) in the `rosidl_generate_interfaces` is `${PROJECT_NAME}`, it will be evaluated to the name of the project; i.e. the string parameter passed to the `project()` directive.

Custom messages and services
`package.xml`

- Because the interfaces rely on `rosidl_default_generators` for generating language-specific code, you need to declare a build tool dependency on it.
- `rosidl_default_runtime` is a runtime or execution-stage dependency, needed to be able to use the interfaces later.
- The `rosidl_interface_packages` is the name of the dependency group that your package, `tutorial_interfaces`, should be associated with, declared using the `<member_of_group>` tag.

- Add the following lines within the `<package>` element of `package.xml`:

```
<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>
```

Custom messages and services build package and test

```
$ cd ~/ros2_ws
$ colcon build --packages-select tutorial_interfaces

$ source ~/ros2_ws/install/setup.bash
```

- Build (only) your package and source `setup.bash`
- Now you can confirm that your interface creation worked by using the `ros2 interface show` command:

```
$ ros2 interface show tutorial_interfaces/msg/Num

$ ros2 interface show tutorial_interfaces/msg/Sphere

$ ros2 interface show tutorial_interfaces/srv/AddThreeInts
```

Custom messages and services
Exercise: use custom `msg` and `srv`

- Rework the previous pub/sub example so that it will use our custom `Num.msg` message
- Do the same for the custom `AddThreeInts.srv` and service nodes
- NOTE: do not forget to edit `package.xml` files to add the new dependencies

Using Parameters
Create package

When making your own nodes you will sometimes need to add parameters that can be set from the launch file.

```
$ cd ~/ros2_ws/src
$ ros2 pkg create --build-type ament_python --license Apache-2.0 python_parameters
--dependencies rclpy
```

- The `--dependencies` argument will automatically add the necessary dependency lines to `package.xml`
- Make sure to add the description, maintainer email and name, and license information to `package.xml`.

```
<description>Python parameter tutorial</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

Using Parameters
Python node

Inside the ros2_ws/src/python_parameters/python_parameters directory, create a new file called python_parameters_node.py

```
import rclpy
import rclpy.node

...

def main():
    rclpy.init()

    node = MinimalParam()

    rclpy.spin(node)

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

CAUTION!

Copy pasting the code may not work

Using Parameters
Python node

CAUTION!

Parameters need to be declared before calling `get_parameter`

```python
class MinimalParam(rclpy.node.Node):
    def __init__(self):
        super().__init__('minimal_param_node')

        # declaration is required unless the node is initialized with
        # allow_undeclared_parameters=True
        self.declare_parameter(name='my_parameter', value='world')

        self.timer = self.create_timer(1, self.timer_callback)

    def timer_callback(self):
        my_param = self.get_parameter('my_parameter').get_parameter_value().string_value

        self.get_logger().info(f'Hello {my_param}!')

        my_new_param = rclpy.parameter.Parameter(
            name='my_parameter',
            type_=rclpy.Parameter.Type.STRING, # optional inferred from value
            value='world'
        )
        all_new_parameters = [my_new_param]
        self.set_parameters(parameter_list=all_new_parameters)
```

Using Parameters
Python node – examine the code

```python
class MinimalParam(rclpy.node.Node):
    def __init__(self):
        super().__init__('minimal_param_node')

        self.declare_parameter('my_parameter', 'world')

        self.timer = self.create_timer(1, self.timer_callback)
```

- Create the class and the constructor.
- Creates a parameter with the name `my_parameter` and a default value of `world`.
  - The parameter type is inferred from the default value. (i.e. string type in this case).
- Next the `timer` is initialized with a period of 1, which causes the `timer_callback` function to be executed once a second.

Using Parameters
Python node – examine the code

```python
def timer_callback(self):
    my_param = self.get_parameter('my_parameter').get_parameter_value().string_value

    self.get_logger().info(f'Hello {my_param}!')

    my_new_param = rclpy.parameter.Parameter(
        name='my_parameter',
        type_=rclpy.Parameter.Type.STRING,
        value='world'
    )
    all_new_parameters = [my_new_param]
    self.set_parameters(parameter_list=all_new_parameters)
```

- Gets the parameter `my_parameter` from the node, and stores it in `my_param`.
- The `get_logger` function ensures the event is logged.
- The `set_parameters` function then sets the parameter `my_parameter` back to the default string value `world`. In the case that the user changed the parameter externally, this ensures it is always reset back to the original.

Using Parameters
Python node – ParameterDescriptor

```
from rcl_interfaces.msg import ParameterDescriptor

class MinimalParam(rclpy.node.Node):
    def __init__(self):
        super().__init__('minimal_param_node')

        my_parameter_descriptor = ParameterDescriptor(description='This parameter is mine!')

        self.declare_parameter(name='my_parameter',
                               value='world',
                               descriptor=my_parameter_descriptor)

        self.timer = self.create_timer(1, self.timer_callback)
```

- Optionally, you can set a descriptor for the parameter to specify a text description of the parameter and its constraints, like making it read-only, specifying a range, etc.

```
$ ros2 param describe /minimal_param_node my_parameter
```

Using Parameters
Add entry point in `setup.py`

Match the `maintainer`, `maintainer_email`, `description` and `license` fields in `setup.py` file to your `package.xml`

```
maintainer='YourName',
maintainer_email='you@email.com',
description='Examples of minimal publisher/subscriber using rclpy',
license='Apache-2.0',
```

Add the following line within the `console_scripts` brackets of the `entry_points` field

```
entry_points={
    'console_scripts': [
        'minimal_param_node = python_parameters.python_parameters_node:main',
    ],
},
```

Using Parameters
build, run and `ros2 param set`

- Build (only) your package and source the setup file and run the node

```
$ cd ~/ros2_ws
$ colcon build --packages-select python_parameters

$ source ~/ros2_ws/install/setup.bash
```

- Run the node

```
$ ros2 run python_parameters minimal_param_node
[INFO] [parameter_node]: Hello world!
[INFO] [parameter_node]: Hello world!
[INFO] [parameter_node]: Hello world!
```

- In a new terminal (do not forget to source `setup.bash`)

```
$ ros2 param list
   my_parameter
   use_sim_time

$ ros2 param set /minimal_param_node my_parameter earth
Set parameter successful
```

Using Parameters launch file

- Parameters can be set in a launch file
- Create the `ros2_ws/src/python_parameters/launch` directory
- In there, create a new file called `param_launch.xml`

```
<launch>
  <node name="param_node" pkg="python_parameters" exec="minimal_param_node" output="screen">
    <param name="my_parameter" value="earth"/>
  </node>
</launch>
```

- Edit setup.py to enable colcon to locate and utilize launch files

```
setup(
    # Other parameters ...
    data_files=[
        # ... Other data files
        # Include all launch files, assumes _launch suffix
        (os.path.join('share', package_name, 'launch'),
         glob(os.path.join('launch', '*launch.[pxy][yma]*')))
    ]
)
```

Using Parameters launch file

- Build (only) your package and source the setup file

```
brairlab@brairlab-vm:~$ cd ~/ros2_ws
brairlab@brairlab-vm:~/ros2_ws$ colcon build --packages-select python_parameters
…
brairlab@brairlab-vm:~/ros2_ws$ source ~/ros2_ws/install/setup.bash
```

- Run the `launch` file

```
brairlab@brairlab-vm:~$ ros2 launch python_parameters param_launch
[INFO] [launch]: All log files can be found below ...
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [minimal_param_node-1]: process started with pid [21126]
[minimal_param_node-1] [INFO] [1728662667.340035974] [minimal_param_node]: Hello earth!
[minimal_param_node-1] [INFO] [1728662669.316700898] [minimal_param_node]: Hello earth!
```

Actions
Create package

Actions are like services that allow you to execute **long running tasks**, provide regular **feedback**, and are **cancelable**.

- E.g. a robot system would use actions for navigation.
  - An action goal could tell a robot to travel to a position.
  - While the robot navigates to the position, it can send updates along the way (i.e. feedback), and then a final result message once it's reached its destination.

```
$ cd ~/ros2_ws/src
$ ros2 pkg create action_tutorials_interfaces
```

Actions
`.action` file

Actions are defined in `.action` files of the form

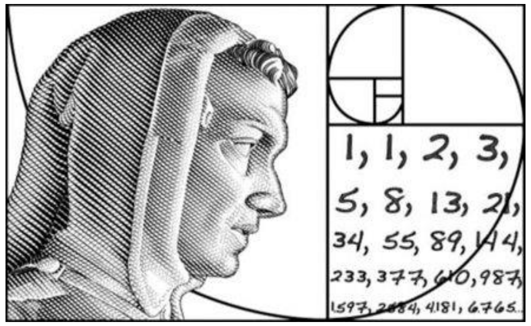
```
# Request
---
# Result
---
# Feedback
```

An action definition is made up of **three** message definitions separated by `---`.

- A `Request` message is sent from an action client to an action server initiating a new goal.
- A `Result` message is sent from an action server to an action client when a goal is done.
- `Feedback` messages are periodically sent from an action server to an action client with updates about a goal.

An instance of an action is typically referred to as a *goal*.

Actions
`Fibonacci.action` example



Liber Abbaci

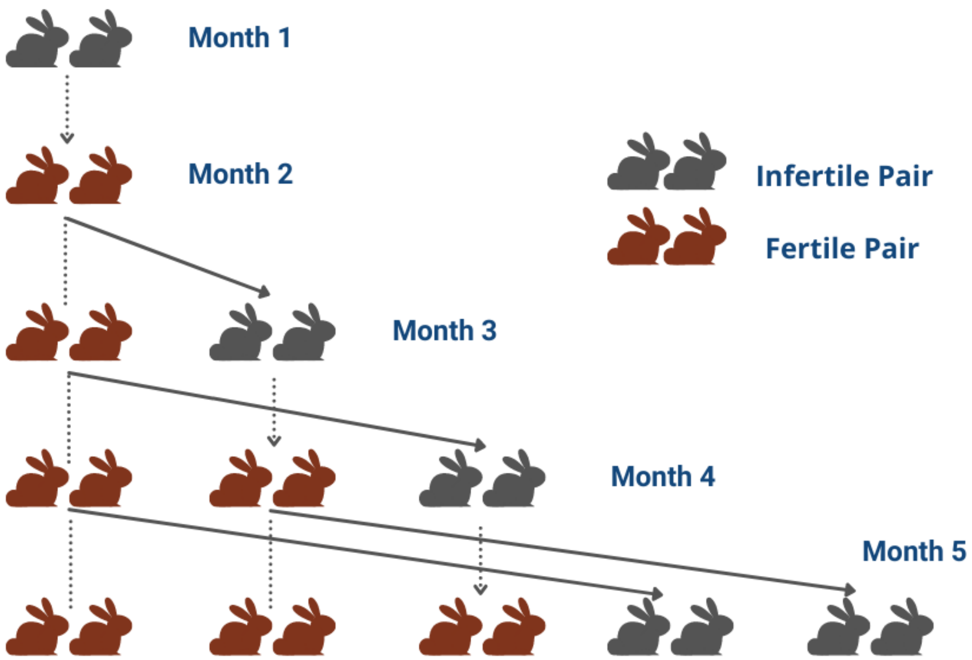
Chapter 12.7:

How Many Pairs of Rabbits Are Created by One Pair in One Year

A certain man had one pair of rabbits together in a certain enclosed place, and one wishes to know how many are created from the pair in one year when it is the nature of them in a single month to bear another pair, and in the second month those born to bear also.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …

$$Fib_n = Fib_{n-2} + Fib_{n-1}$$



Actions
`Fibonacci.action` example

Create an `action` directory in our ROS 2 package `action_tutorials_interfaces`

```
$ cd action_tutorials_interfaces
$ mkdir action
```

Within the `action` directory, create a file called `Fibonacci.action` with the

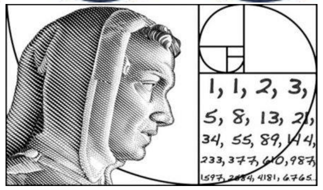
```
int32 order
---
int32[] sequence
---
int32[] partial_sequence
```

CAUTION!

`.action` files name must be capitalized!

- The goal request is the `order` of the Fibonacci sequence we want to compute,
- The result is the final `sequence`
- The feedback is the `partial_sequence` computed so far.

Actions
`Fibonacci.action` example - build



Before we can use the new Fibonacci action type in our code, we must pass the definition to the `rosidl` code generation pipeline.

This is accomplished by adding the following lines to our `CMakeLists.txt` before the `ament_package()` line, in the `action_tutorials_interfaces`:

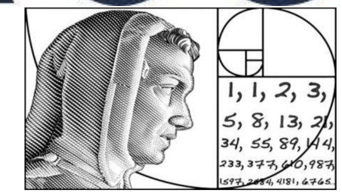
```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "action/Fibonacci.action"
)
```

We should also add the required dependencies to our `package.xml`:

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<depend>action_msgs</depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Note, we need to depend on `action_msgs` since action definitions include additional metadata (e.g. goal IDs).

Actions
`Fibonacci.action` example - build



We should now be able to build the package containing the `Fibonacci` action definition:

```
# Change to the root of the workspace
cd ~/ros2_ws
# Build
colcon build --packages-select action_tutorials_interfaces
```

By convention, action types will be prefixed by their package name and the word `action`.
So when we want to refer to our new action, it will have the full name `action_tutorials_interfaces/action/Fibonacci`.

```
# Source the workspace
. install/setup.bash
# Check that our action definition exists
ros2 interface show action_tutorials_interfaces/action/Fibonacci
```

Action server Node
Create package

Let's write an action server that computes the Fibonacci sequence using `Fibonacci.action`

```
$ cd ~/ros2_ws/src
$ ros2 pkg create --build-type ament_python --license Apache-
2.0 action_tutorials_py --dependencies rclpy action_tutorials_interfaces
```

The `--dependencies` argument will automatically add the necessary dependency lines to `package.xml.`
`action_tutorial_interfaces` is the package that includes the `.action` file defining the action

```
int32 order
---
int32[] sequence
---
int32[] partial_sequence
```

Action server Node
Update package.xml and setup.py

```
<description>Python action tutorial code</description>
<maintainer email="brairlab@todo.todo">brairlab</maintainer>
<license>Apache-2.0</license>
```

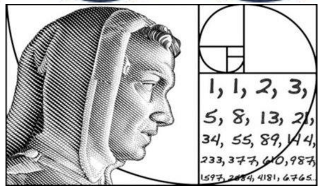
Because you used the `--dependencies` option during package creation, you don’t have to manually add dependencies

```
<exec_depend>rclpy</exec_depend>
<exec_depend>action_tutorials_interfaces</exec_depend>
```

Update `setup.py` matching the info contained in `package.xml`

```
maintainer='Your Name',
maintainer_email='brairlab@todo.todo',
description='Python action tutorial code',
license='Apache License 2.0',
```

Action server Node



Create and edit `fibonacci_action_server.py` in the `ros2_ws/src/action_tutorials_py/action_tutorials_py` directory

```
import rclpy
from rclpy.action import ActionServer
from rclpy.node import Node

from action_tutorials_interfaces.action import Fibonacci


...

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

    fibonacci_action_server = FibonacciActionServer()

    rclpy.spin(fibonacci_action_server)


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

CAUTION!

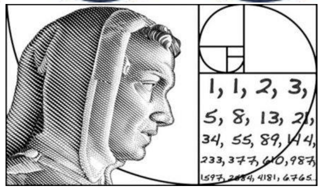
Copy pasting the code may not work

Action server Node

```
class FibonacciActionServer(Node):

    def __init__(self):
        super().__init__('fibonacci_action_server')
        self._action_server = ActionServer(
            node=self,
            action_type=Fibonacci,
            action_name='fibonacci',
            execute_callback=self.execute_callback)
```

- Definition of a class `FibonacciActionServer` subclassing `Node`.
- The class is initialized by calling the `Node` constructor, naming our node `fibonacci_action_server`

Action server Node




```
        self._action_server = ActionServer(
            node=self,
            action_type=Fibonacci,
            action_name='fibonacci',
            execute_callback=self.execute_callback)
```

An action server requires four arguments:

- A ROS 2 node to add the action client to: `self`.
- The type of the action: `Fibonacci` (imported earlier).
- The action name: `'fibonacci'`.
- A callback function for executing accepted goals: `self.execute_callback`. This callback **must** return a result message for the action type.

Action server Node

```python
def execute_callback(self, goal_handle):
    self.get_logger().info('Executing goal...')

    fbck_msg = Fibonacci.Feedback(partial_sequence=[0, 1])

    for i in range(1, goal_handle.request.order):
        fbck_msg.partial_sequence.append(
            fbck_msg.partial_sequence[i] + fbck_msg.partial_sequence[i-1])

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

        goal_handle.publish_feedback(fbck_msg)
        time.sleep(1)

    goal_handle.succeed()

    return Fibonacci.Result(sequence=fbck_msg.partial_sequence)
```

This is the method that will be called to execute a goal once it is accepted.
Actions have the the ability to provide feedback to an action client during goal execution.
We can make our action server publish feedback for action clients by calling the goal handle's publish_feedback() method

Action server Node
Add entry point in setup.py and build

To allow the ros2 run command to run your node, you must add the entry point to setup.py (in the ros2_ws/src/action_tutorials_py/action_tutorials_py directory).

```
entry_points={
        'console_scripts': [
            'fibonacci_action_server = action_tutorials_py.fibonacci_action_server:main',
        ],
},
```

Let's build the node and source the setup.bash file

```
$ cd ~/ros2_ws
$ colcon build --packages-select action_tutorials_py

$ source ~/ros2_ws/install/setup.bash
```

Action server Node
run the server

Run the server node

```
$ ros2 run action_tutorials_py fibonacci_action_server

# in a different terminal
$ ros2 action send_goal fibonacci action_tutorials_interfaces/action/Fibonacci "{order: 4}"
```

```
brairlab@brairlab-vm:~$ ros2 run action_tutorials_py fibonacci_action_server
[INFO] [1728991572.0] [fibonacci_action_server]: Executing goal...
[INFO] [1728991572.0] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1])
[INFO] [1728991573.1] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1, 2])
[INFO] [1728991574.1] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1, 2, 3])
[INFO] [1728991575.1] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1, 2, 3, 5])
```

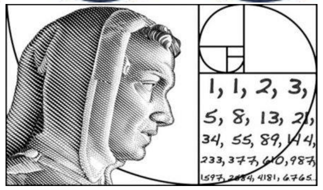
Action server Node
run the server

Send a goal

```
# in a different terminal
$ ros2 action send_goal fibonacci action_tutorials_interfaces/action/Fibonacci "{order: 4}"
```

```
brairlab@brairlab-vm:~$ ros2 action send_goal fibonacci
action_tutorials_interfaces/action/Fibonacci "{order: 4}

Waiting for an action server to become available...
Sending goal:
     order: 4

Goal accepted with ID: 4cfc72032977423690875272bfe1b1dd

Result:
    sequence:
- 0
- 1
- 1
- 2
- 3

Goal finished with status: SUCCEEDED, 1, 2, 3, 5])
```

Action client Node



Create and edit fibonacci_action_client.py in the

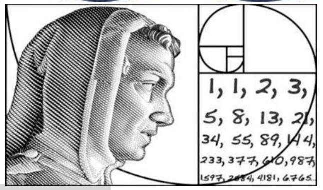
ros2_ws/src/action_tutorials_py/action_tutorials_py directory

```
import rclpy
from rclpy.action import ActionClient
from rclpy.node import Node

from action_tutorials_interfaces.action import Fibonacci

...

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

    action_client = FibonacciActionClient()

    future = action_client.send_goal(10)

    rclpy.spin_until_future_complete(action_client, future)


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

CAUTION!

Copy pasting the code may not work

Action client Node send goals




```python
class FibonacciActionClient(Node):

    def __init__(self):
        super().__init__('fibonacci_action_client')
        self._action_client = ActionClient(self, Fibonacci, 'fibonacci')

    def send_goal(self, order):
        goal_msg = Fibonacci.Goal(order=order)

        self._action_client.wait_for_server()

        return self._action_client.send_goal_async(goal_msg)
```

- Definition of a class `FibonacciActionClient` subclassing `Node`. The class is initialized by calling the `Node` constructor, naming our node `fibonacci_action_client`
- This method waits for the action server to be available, then sends a goal to the server. It returns a future that we can later wait on.

Action client Node

```
        self._action_client = ActionClient(node=self,
                                           action_type=Fibonacci,
                                           action_name='fibonacci')
```

An action client requires three arguments:

- A ROS 2 node to add the action client to: `self`.
- The type of the action: `Fibonacci` (imported earlier).
- The action name: `'fibonacci'`.

Let's add the entry_point in `setup.py`

```
entry_points={
        'console_scripts': [
            'fibonacci_action_server = action_tutorials_py.fibonacci_action_server:main',
            'fibonacci_action_client = action_tutorials_py.fibonacci_action_client:main',
        ],
},
```

Let's build the node and source the `setup.bash` file

```
$ cd ~/ros2_ws
$ colcon build --packages-select action_tutorials_py

$ source ~/ros2_ws/install/setup.bash
```

Action client Node

Run the server in a terminal

```
brairlab@brairlab-vm:~$ ros2 run action_tutorials_py fibonacci_action_server
[INFO] [1728991572.0] [fibonacci_action_server]: Executing goal...
[INFO] [1728991572.0] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1])
[INFO] [1728991573.1] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1, 2])
[INFO] [1728991574.1] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1, 2, 3])
[INFO] [1728991575.1] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1, 2, 3, 5])
```

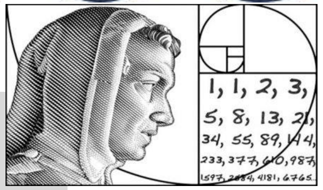
and the client in another

```
brairlab@brairlab-vm:~$ ros2 run action_tutorials_py fibonacci_action_client
brairlab@brairlab-vm:~$
```

We have sent a **goal** but we haven't got neither a **result** nor any intermediate **feedback.**

Let's start with getting **results**.

Action client Node handle results



```python
def send_goal(self, order):
    goal_msg = Fibonacci.Goal(order=order)
    self._action_client.wait_for_server()
    self._send_goal_future = self._action_client.send_goal_async(goal_msg)

    self._send_goal_future.add_done_callback(self.goal_response_callback)

def goal_response_callback(self, future):
    goal_handle = future.result()
    if not goal_handle.accepted:
        self.get_logger().info('Goal rejected :(')
        return

    self.get_logger().info('Goal accepted :)')

    self._get_result_future = goal_handle.get_result_async()
    self._get_result_future.add_done_callback(self.get_result_callback)

def get_result_callback(self, future):
    result = future.result().result
    self.get_logger().info(f'Result: {result.sequence}')
    rclpy.shutdown()
```

CAUTION!

Copy pasting the code may not work

Action client Node handle results

The ActionClient.send_goal_async() method returns a future to a goal handle.
We register a callback for when the future is complete

```
self._send_goal_future.add_done_callback(self.goal_response_callback)
```

Note that the future is completed when an action server accepts or rejects the goal request.

We can check to see if the goal was rejected and return early since we know there will be no result:

```
    def goal_response_callback(self, future):
        goal_handle = future.result()
        if not goal_handle.accepted:
            self.get_logger().info('Goal rejected :(')
            return

        self.get_logger().info('Goal accepted :)')
```

Action client Node handle results

Now that we’ve got a goal handle, we can use it to request the result with the method get_result_async().
Similar to sending the goal, we will get a future that will complete when the result is ready.
Let’s register a callback just like we did for the goal response:

```
        self._get_result_future = goal_handle.get_result_async()
        self._get_result_future.add_done_callback(self.get_result_callback)
```

In the callback, we log the result sequence and shutdown ROS 2 for a clean exit:

```
    def get_result_callback(self, future):
         result = future.result().result
         self.get_logger().info('Result: {result.sequence}')
         rclpy.shutdown()
```

Let's build the node and source the `setup.bash` file

```
$ cd ~/ros2_ws
$ colcon build --packages-select action_tutorials_py

$ source ~/ros2_ws/install/setup.bash
```

Action client Node
handle results

Run the server in a terminal:

```
brairlab@brairlab-vm:~$ ros2 run action_tutorials_py fibonacci_action_server
[INFO] [1728991572.0] [fibonacci_action_server]: Executing goal...
[INFO] [1728991572.0] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1])
[INFO] [1728991573.1] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1, 2])
[INFO] [1728991574.1] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1, 2, 3])
[INFO] [1728991575.1] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1, 2, 3, 5])
```

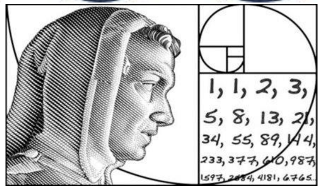
and the client in another:

```
brairlab@brairlab-vm:~$ ros2 run action_tutorials_py fibonacci_action_client
[INFO] [1729002908.7] [fibonacci_action_client]: Goal accepted :)
[INFO] [1729002911.8] [fibonacci_action_client]: Result: array('i', [0, 1, 1, 2, 3])
```

We have sent a **goal** and got the **result** (like a Service).

Let's try to get intermediate **feedback** too.

Action client Node handle feedbacks



CAUTION!

Copy pasting the code may not work

```
def send_goal(self, order):
     goal_msg = Fibonacci.Goal(order=order)
     self._action_client.wait_for_server()
     self._send_goal_future = \
         self._action_client.send_goal_async(goal_msg,
                                             feedback_callback=self.feedback_callback)

     self._send_goal_future.add_done_callback(self.goal_response_callback)

...

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

Action client Node handle feedbacks

Here's the callback function for feedback messages:

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

In the callback we get the feedback portion of the message and print the `partial_sequence` field to the screen.

We need to register the callback with the action client. This is achieved by additionally passing the callback to the action client when we send a goal:

```
self._send_goal_future = \
    self._action_client.send_goal_async(goal_msg,
                                        feedback_callback=self.feedback_callback)
```

Action client Node handle results

Run the client

```
brairlab@brairlab-vm:~$ ros2 run action_tutorials_py fibonacci_action_client
[INFO] [1729004162.3] [fibonacci_action_client]: Goal accepted :)
[INFO] [1729004162.3] [fibonacci_action_client]: Received feedback: array('i', [0, 1, 1])
[INFO] [1729004163.3] [fibonacci_action_client]: Received feedback: array('i', [0, 1, 1, 2])
[INFO] [1729004164.3] [fibonacci_action_client]: Received feedback: array('i', [0, 1, 1, 2, 3])
[INFO] [1729004165.3] [fibonacci_action_client]: Result: array('i', [0, 1, 1, 2, 3])
```

We have sent a **goal** and got the intermediate **feedbacks** and the **result**

- There are several ways you could write an action server and client in Python; check out the `minimal_action_server` and `minimal_action_client` packages in the ros2/examples repo.
- For more detailed information about ROS actions, please refer to the design article.

Logging

To log your nodes activities do not use *naked* print statement, be smart and leverage instead ROS 2 logging subsystem.

It delivers logging messages to a variety of targets, including:

- To the console (if one is attached)
- To log files on disk (if local storage is available)
- To the `/rosout` topic on the ROS 2 network

By default, log messages in ROS 2 nodes will go out to the console (on *stderr*), to log files on disk, and to the `/rosout` topic on the ROS 2 network.

All of the targets can be individually enabled or disabled on a per-node basis.

reference

https://docs.ros.org/en/humble/Concepts/Intermediate/About-Logging.html

Standard streams

https://en.wikipedia.org/wiki/Standard_streams

Logging

- Log messages have a severity level associated with them: `DEBUG`, `INFO`, `WARN`, `ERROR` or `FATAL`, in ascending order.

- A logger will only process log messages with severity at or higher than a specified level chosen for the logger.

- Each node has a logger associated with it that automatically includes the node's name and namespace.
  - If the node's name is externally remapped to something other than what is defined in the source code, it will be reflected in the logger name.
  - Non-node loggers can also be created that use a specific name.

reference

https://docs.ros.org/en/humble/Concepts/Intermediate/About-Logging.html

Logging

- `logger.{debug,info,warning,error,fatal}` - output the given Python string to the logging infrastructure. The calls accept the following keyword args to control behavior:
  - `throttle_duration_sec` - if not `None`, the duration of the throttle interval in floating-point seconds
  - `skip_first` - if `True`, output the message all but the first time this line is hit
  - `once` - if `True`, only output the message the first time this line is hit

- `rclpy.logging.set_logger_level` - Set the logging level for a particular logger name to the given severity level

- `rclpy.logging.get_logger_effective_level` - Given a logger name, return the logger level (which may be unset)

reference

https://docs.ros.org/en/humble/Concepts/Intermediate/About-Logging.html

Logging

- **`node.get_logger().debug()`**
  - Information that you never need to see if the system is working properly
- **`node.get_logger().info()`**
  - Small amounts of information that may be useful to a user
- **`node.get_logger().warn()`**
  - Information that the user may find alarming, and may affect the output of the application, but is part of the expected working of the system
- **`node.get_logger().error()`**
  - Something serious (but recoverable) has gone wrong
- **`node.get_logger().fatal()`**
  - Something unrecoverable has happened

reference

https://docs.ros.org/en/humble/Tutorials/Demos/Logging-and-logger-configuration.html

Executors (Advanced Topic)

- Execution management in ROS2 is handled by **Executors**
- An Executor uses one or more ***threads*** of the *underlying operating system* to invoke the callbacks of subscriptions, timers, service servers, action servers, etc. on incoming messages and events.

- In the simplest case, the main thread is used for processing the incoming messages and events of a Node by calling `rclpy.spin(node)`

```
...
def main():
    rclpy.init()

    node = SomeNode()

    rclpy.spin(node)
    rclpy.shutdown()

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

reference

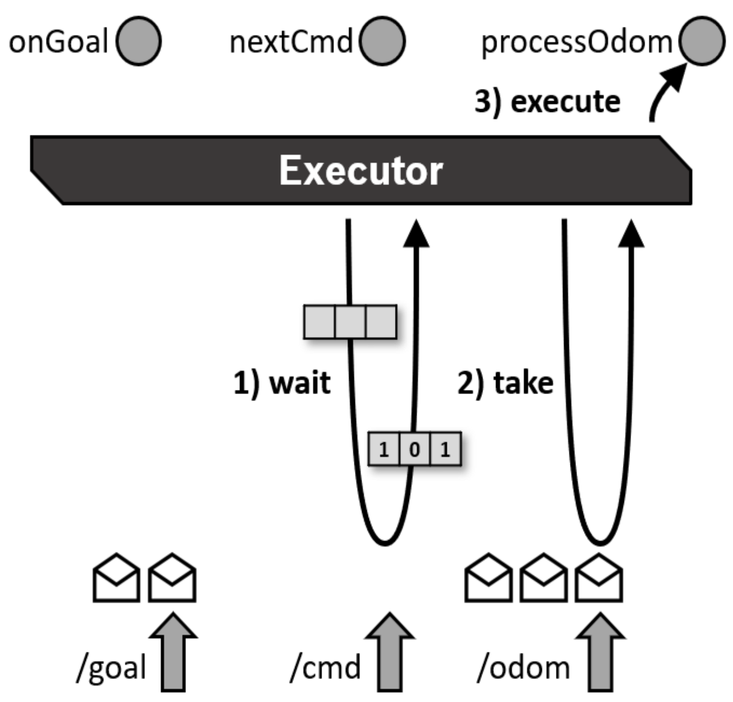
https://docs.ros.org/en/humble/Concepts/Intermediate/About-Executors.html

Executors

- The call to `rclpy.spin(node)` basically expands to an instantiation and invocation of the Single-Threaded Executor, the simplest Executor:

```
from rclpy.executors import SingleThreadedExecutor

Node = SomeNode()

Executor = SingleThreadedExecutor();
executor.add_node(node);

executor.spin();
```

- By invoking `spin()` of the Executor instance, the current thread starts querying the `rcl` and middleware layers for incoming messages and other events and calls the corresponding callback functions until the node shuts down.
- A *wait set* is used to inform the Executor about available messages on the middleware layer, with one binary flag per queue. The *wait set* is also used to detect when timers expire.



Executors

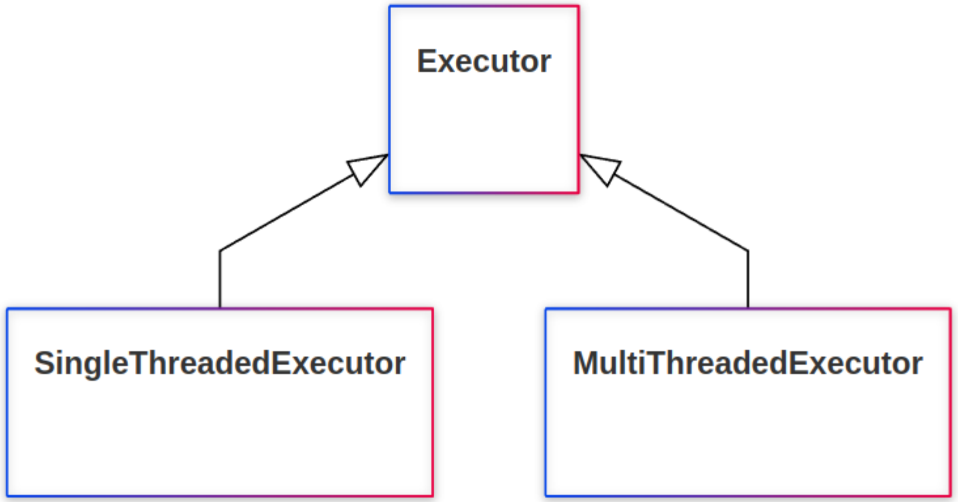
`rclpy.executors` defines two types of Executors:

- `SingleThreadedExetutor`
- `MultiThreadedExecutor`

- `SingleThreadedExetutor`
- `MultiThreadedExecutor` creates a configurable number of threads to allow for processing multiple messages or events in parallel
  - The actual parallelism depends on the ***callback groups***.

reference

https://docs.ros.org/en/humble/Concepts/Intermediate/About-Executors.html



Executors - Callback Groups

- ROS 2 allows organizing the callbacks of a node in groups.
- In `rclpy`, such a *callback group* can be created by calling the constructor of the specific callback group type.
- The callback group must be stored throughout execution of the node (eg. as a class member), otherwise the executor won't be able to trigger the callbacks. Then, this callback group can be specified when creating a subscription, timer, etc.

```
my_callback_group = MutuallyExclusiveCallbackGroup()

my_subscription = self.create_subscription(Int32, "/topic", self.callback, 1,
callback_group=my_callback_group)
```

- All subscriptions, timers, etc. that are created without the indication of a callback group are assigned to the *default callback group*.
  - It can be queried with `Node.default_callback_group` in `rclpy`.

reference

Callback Groups

Executors - Callback Groups

- There are two types of callback groups, where the type has to be specified at instantiation time:
  - *Mutually exclusive:* Callbacks of this group must not be executed in parallel.
  - *Reentrant:* Callbacks of this group may be executed in parallel.

- These callback groups restrict the execution of their callbacks in different ways. In short:
  - *Mutually Exclusive Callback Group* prevents its callbacks from being executed in parallel - essentially making it as if the callbacks in the group were executed by a `SingleThreadedExecutor`.
  - *Reentrant Callback Group* allows the executor to schedule and execute the group's callbacks in any way it sees fit, without restrictions. This means that, in addition to different callbacks being run parallel to each other, different instances of the same callback may also be executed concurrently.
  - Callbacks belonging to different callback groups (of any type) can always be executed parallel to each other.

- Keep in mind that different ROS 2 entities relay their callback group to all callbacks they spawn.
  - e.g. if one assigns a callback group to an action client, all callbacks created by the client will be assigned to that callback group.

reference

[Using Callback Groups]

Executors - Callback Groups

In the context of ROS 2 and executors, a callback means a function whose scheduling and execution is handled by an executor.

Examples of callbacks in this context are:

- *subscription callbacks* (receiving and handling data from a topic),
- *timer callbacks*,
- *service callbacks* (for executing service requests in a server),
- *different callbacks* in action servers and clients,
- *done-callbacks* of Futures.

Keep the following in mind when working with callback groups:

- Almost *everything* in ROS 2 is a callback! Every function that is run by an executor is, by definition, a callback.
- Sometimes the callbacks are hidden and their presence may not be obvious from the user/developer API. This is the case especially with any kind of "synchronous" call to a service or an action (in `rclpy`).
  - For example, the synchronous call `Client.call(request)` to a service adds a Future's done-callback that needs to be executed during the execution of the function call, but this callback is not directly visible to the user.

reference

[Using Callback Groups]

Executors - Callback Groups

In order to control execution with callback groups, one can consider the following guidelines.

- For the interaction of an individual callback with itself:
    - Register it to a *Reentrant Callback Group* if it should be executed in parallel to itself. An example case could be an action/service server that needs to be able to process several action calls in parallel to each other.
    - Register it to a *Mutually Exclusive Callback Group* if it should **never** be executed in parallel to itself. An example case could be a timer callback that runs a control loop that publishes control commands.
- For the interaction of different callbacks with each other:
    - Register them to the same *Mutually Exclusive Callback Group* if they should **never** be executed in parallel. An example case could be that the callbacks are accessing shared critical and non-thread-safe resources.

reference

Using Callback Groups

Executors - Callback Groups

If they should be *executed in **parallel***, you have two options, depending on whether the individual callbacks should be able to overlap themselves or not:

- Register them to different *Mutually Exclusive Callback Groups* (no overlap of the individual callbacks)
- Register them to a *Reentrant Callback Group* (overlap of the individual callbacks)

An example case of running different callbacks in parallel is a Node that has a synchronous service client and a timer calling this service.

reference

Using Callback Groups

Executors - Callback Groups

- Setting up callback groups of a node incorrectly can lead to **deadlocks** (or other unwanted behavior), especially if one desires to use *synchronous* calls to services or actions.
- Indeed, even the API documentation of ROS 2 discourages *synchronous* calls to actions or services should not be done in callbacks, because it can lead to deadlocks.
  - While using *asynchronous* calls is indeed safer in this regard, *synchronous* calls can also be made to work.
  - On the other hand, *synchronous* calls also have their advantages, such as making the code simpler and easier to understand.
- Note that every node's *default callback* group is a *Mutually Exclusive Callback Group*.
  - If the user does not specify any other callback group when creating a timer, subscription, client etc., any callbacks created then or later by these entities will use the node's default callback group.
  - If everything in a node uses the same *Mutually Exclusive Callback Group*, that node essentially acts as if it was handled by a *Single-Threaded Executor*, even if a multi-threaded one is specified!
  - Whenever one decides to use a *Multi-Threaded Executor*, some callback group(s) should always be specified in order for the executor choice to make sense.

reference

Using Callback Groups

Executors - Callback Groups

Here are a couple guidelines to help avoid deadlocks:

- If you make a *synchronous call in any type of a callback*, this callback and the client making the call need to belong to
    - *different callback groups* (of any type), or
    - a *Reentrant Callback Group.*

If the above configuration is not possible due to other requirements - such as thread-safety and/or blocking of other callbacks while waiting for the result (or if you want to make absolutely sure that there is never a possibility of a deadlock), use asynchronous calls.

Failing the first point will always cause a deadlock.

- An example of such a case would be making a synchronous service call in a timer callback (example here), or in a topic subscriber callback.

reference

Using Callback Groups