Robot Operating System (ROS)

Getting started

Claudia Álvarez Aparicio
calvaa@unileon.es

GRUPO DE ROBÓTICA

Seguridad en Sistemas Ciberfísicos

Curso 2023–2024

Contenedorización

1.1. Contenedores versus VMs

Contenedorización

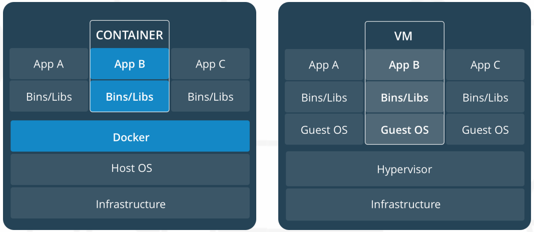
- El uso de contenedores Linux para implementar aplicaciones se denomina **contenedorización**.
- Los contenedores no son nuevos, pero su uso para implementar aplicaciones fácilmente sí lo es.
- **Contenedor** $\longrightarrow$ ejemplar en tiempo de ejecución de una imagen (cargada en memoria).
- **Imagen** $\longrightarrow$ paquete ejecutable que incluye todo lo necesario para ejecutar una aplicación: código, entorno de ejecución, bibliotecas, variables de entorno y archivos de configuración.

Docker

Plataforma para desarrolladores y administradores de sistemas para desarrollar, implementar y ejecutar aplicaciones con contenedores. Permite separar las aplicaciones de la infraestructura para desplegar software rápidamente.

Contenedores versus VMs

- Un contenedor se ejecuta nativamente en Linux y comparte el kernel de la máquina host con otros contenedores.
- Un contenedor se ejecuta como cualquier otro proceso, no teniendo más memoria que otro ejecutable (lightweight).
- Una máquina virtual (VM) ejecuta un sistema operativo “invitado” completo con acceso virtual a recursos de host a través de un hipervisor.
- En general, las VM proporcionan un entorno con más recursos de los que la mayoría de las aplicaciones necesitan.



1.2. Get started

Docker installation I

1. Set up the repository:

```
$ sudo apt-get update # Update the package index
$ sudo apt-get install ca-certificates curl gnupg lsb-release # Install packages to allow apt to use
     a repository over HTTP
$  curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/
     keyrings/docker-archive-keyring.gpg  # Add Docker's official GPG key
$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.
     gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/
     sources.list.d/docker.list > /dev/null # Use the following command to set up the stable
     repository
```

Docker installation II

2 Install Docker CE:

```
$ sudo apt-get update # Update the package index
$ sudo apt-get install apt-get install docker-ce docker-ce-cli containerd.io # Install the latest
     version of Docker CE
$ sudo docker run hello-world # Verify that Docker CE is installed correctly by running the hello-
     world image.
...
Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```

3 Post-installation

- Manage Docker as a non-root user:

```
$ sudo groupadd docker # Create the docker group
$ sudo usermod -aG docker $USER # Add your user to the docker group. Log out and log back in so
     that your group membership is re-evaluated or run 'newgrp docker' to activate the changes
     to groups
$ docker run hello-world # Verify that you can run docker commands without sudo
```

Basic commands

- List Docker CLI commands:

```
$ docker
$ docker container --help
```

- Display Docker version and info:

```
$ docker --version
$ docker version
$ docker info
```

- Execute Docker images:

```
$ docker run hello-world
$ docker run -it <image> sh # -it attaches us to an interactive tty in the container
```

- List Docker images/networks/containers:

```
$ docker image ls
$ docker network ls
$ docker container ls        # containers running
$ docker container ls --all  # all
$ docker container ls -a -q  # all in quiet mode
$ docker ps
```

1.3. Define a container

Sample application

- Create a simple Python application using the Flask framework:

```
$ cd /path/to/app
$ pip3 install Flask
$ pip3 freeze | grep Flask >> requirements.txt
$ touch app.py
```

- Enter the following code into the `app.py` file:

```python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, Docker!'
```

- Test the application:

```
$ pwd
/path/to/app
$ python3 -m flask run
```

- Open a browser and navigate to `http://localhost:5000`. Switch back to the terminal where the server is running and you should see the requests in the server logs.

Create a Dockerfile

- *Dockerfile* defines what goes on in the environment inside your container:

```
FROM python:3.8-slim-buster # Docker images inherit from other base images

WORKDIR /app # default location for all subsequent commands

COPY requirements.txt requirements.txt # copy files into the image
RUN pip3 install -r requirements.txt # execute the command
#RUN pip3 install Flask # Uncomment on Windows and Mac and comment the above lines

COPY . . # Takes all the files located in the current directory and copies them into the
     image

CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"] # Command to run when the image
      is executed inside a container
```

- The Python application directory (`/path/to/app`) structure would now look like:

```
app
|____ app.py
|____ requirements.txt
|____ Dockerfile
```

Build and run the app

- Build the app:

```
$ pwd
/path/to/app
$ docker build --tag sample-app .
...
$ docker images
...
```

- Run the app:

```
$ docker run --publish 5000:5000 sample-app
```

- In a new terminal:

```
$ curl localhost:5000
Hello, Docker!
```

- Run the app in the background, in detached mode:

```
$ docker run -d -p 5000:5000 sample-app
$ docker ps
$ docker stop CONTAINER_ID
$ docker ps
```

1.4. Rocker

Rocker

Rocker

Herramienta que permite ejecutar imágenes de Docker con soporte local, por ejemplo soporte de nvidia. Dando la posibilidad de abrir interfaces gráficas en el equipo host.
`https://github.com/osrf/rocker`

- Installation:

```
$ pip install rocker
```

- Extensions:
  - x11 – Enable the use of X11 inside the container via the host X instance.
  - nvidia – Enable NVIDIA graphics cards for rendering
  - cuda – Enable NVIDIA CUDA in the container
  - user – Create a user inside the container with the same settings as the host and run commands inside the container as that user.
  - home – Mount the user's home directory into the container
  - pulse – Mount pulse audio into the container
  - ssh – Pass through ssh access to the container.

1.5. Docker Hub

Docker Hub

Docker Hub

Repositorio de imágenes docker.
`https://hub.docker.com/`

- Download an image:

```
$ docker pull <name of the image>
```

Robot Operating System 2 (ROS 2)

References

ROS 2 Documentation

ROS 2 Official Website

ROS 2 Tutorials

Introduction to ROS 2

What is ROS?

Definition

ROS (Robot Operating System) is an open-source, flexible framework for writing robot software.

- Not an actual operating system, but a **middleware**
- Collection of tools, libraries, and conventions
- Facilitates communication between processes
- Provides hardware abstraction and low-level device control

Key Features

- **Distributed computing**: Processes can run on multiple machines
- **Reusable code**: Extensive package ecosystem
- **Language independence**: C++, Python, and more
- **Tool ecosystem**: Visualization, simulation, debugging

History and Motivation I

The Problem Before ROS

- Robot development required **reinventing the wheel**
- No standardized way to share code between robots
- Difficult to integrate different hardware and sensors
- Limited collaboration across research groups
- Each project started from scratch

History and Motivation II

ROS History

- **2007**: Started at Stanford University (STAIR project)
- **2008**: Development continued at Willow Garage
- **2010**: ROS 1.0 released
- **2013**: Open Source Robotics Foundation (OSRF) created
- **2017**: ROS 2 development begins in earnest
- **2020**: ROS 2 reaches production maturity
- **Today**: Used worldwide in research, industry, and education

History and Motivation III

Why ROS Succeeded

- **Open source**: Free to use and modify
- **Community-driven**: Thousands of contributors
- **Rich ecosystem**: Thousands of packages available
- **Industry adoption**: Used in commercial robots
- **Educational value**: Widely taught in universities
- **Active development**: Continuous improvements

ROS Versions

ROS 1: Mature, widely used, but with limitations
ROS 2: Modern redesign addressing ROS 1 limitations

Why ROS 2? I

Limitations of ROS 1

- **Single point of failure**: roscore required for all communications
- **No real-time support**: not suitable for safety-critical systems
- **Limited multi-robot support**: challenging to run multiple robots
- **Network dependency**: relies heavily on stable network connections
- **Security**: minimal built-in security features
- **Platform support**: primarily Linux-focused

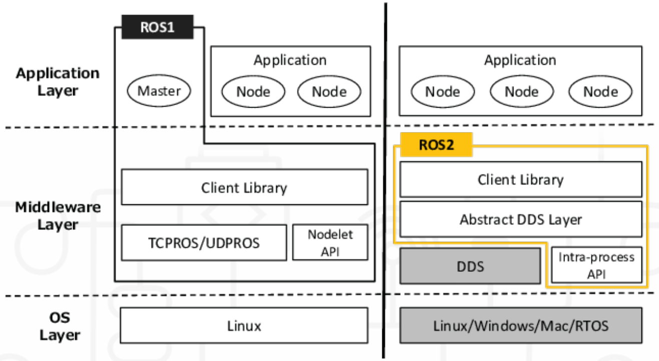
Why ROS 2? II

ROS 2 Improvements

- **No roscore**: fully distributed peer-to-peer architecture
- **Real-time capable**: support for real-time systems with DDS
- **Multi-robot native**: designed for multiple robot systems
- **Better reliability**: no single point of failure
- **Security**: built-in security features (SROS2)
- **Cross-platform**: Windows, Linux, and macOS support
- **Production ready**: suitable for commercial products

ROS 2 Architecture

- Built on top of **DDS** (Data Distribution Service)
- DDS provides **discovery**, **serialization**, and **transportation**
- Multiple DDS implementations supported (FastDDS, CycloneDDS, etc.)



ROS 2 Architecture based on DDS

What is DDS? I

Definition

DDS (Data Distribution Service) is an industry-standard protocol for real-time, scalable, and high-performance data exchange.

- Standardized by **Object Management Group (OMG)**
- Publish-subscribe middleware for distributed systems
- Used in mission-critical applications (aerospace, defense, healthcare)

What is DDS? II

Key Features of DDS

- **Decentralized**: No central broker required (peer-to-peer)
- **Quality of Service (QoS)**: Fine-grained control over reliability, latency, bandwidth
- **Data-centric**: Focus on data rather than connections
- **Discovery**: Automatic discovery of publishers and subscribers
- **Real-time capable**: Deterministic and low-latency communication
- **Scalable**: Supports large-scale distributed systems

What is DDS? III

DDS Implementations in ROS 2

ROS 2 supports multiple DDS vendors:

- **Fast DDS** (eProsima): Default in ROS 2 Humble
- **Cyclone DDS** (Eclipse Foundation): Lightweight and fast

Why DDS for ROS 2?

DDS solves ROS 1 limitations: no single point of failure, built-in QoS, real-time support, and production-ready reliability.

ROS 2 Distributions I

LTS (Long Term Support) Distributions

- **Humble Hawksbill** (May 2022): Ubuntu 22.04, supported until May 2027
- **Foxy Fitzroy** (June 2020): Ubuntu 20.04, supported until May 2023

Current Distributions (as of 2024)

- **Iron Irwini** (May 2023): Ubuntu 22.04
- **Jazzy Jalisco** (May 2024): Ubuntu 24.04
- Rolling Ridley: continuously updated development distribution

Recommendation

For production and learning, use **Humble Hawksbill** (LTS version)

ROS 2 Concepts

3 ROS 2 Concepts

ROS 2 Core Concepts I

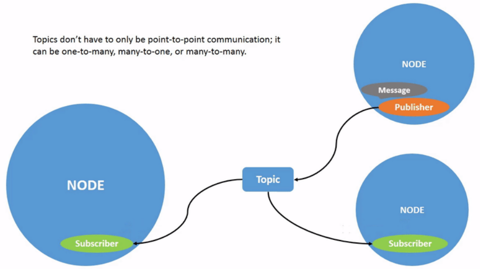
Nodes	Processes that perform computation. Same as ROS 1 but with improved lifecycle management.
Topics	Named buses for asynchronous communication using publish/subscribe pattern. Similar to ROS 1.
Services	Synchronous request-reply communication. Enhanced in ROS 2 with better timeout handling.
Actions	Asynchronous goal-oriented tasks with feedback. Improved implementation compared to ROS 1.
Parameters	Runtime configuration values for nodes. Enhanced with parameter events and better type safety.
Quality of Service (QoS)	NEW in ROS 2: fine-grained control over communication behavior (reliability, durability, etc.).

ROS 2 Communication: Topics

Definition

Buses **many-to-many**, **asynchronous** communication using the publish-subscribe pattern.

- Multiple publishers/subscribers can send/receive messages to/from a topic
- Fire-and-forget: publishers don't wait for acknowledgment
- Best for: sensor data, continuous streams (e.g., camera images, laser scans)


Topics don't have to only be point-to-point communication; it can be one-to-many, many-to-one, or many-to-many.
NODE
Message
Publisher
Topic
NODE
Subscriber
NODE
Subscriber

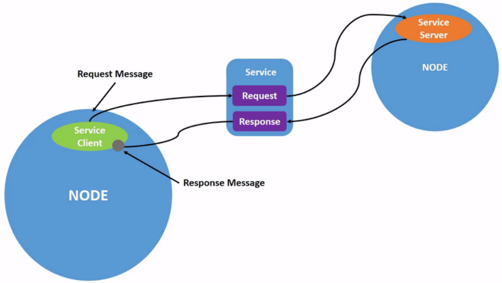
Topic communication pattern (click for tutorial)

ROS 2 Communication: Services

Definition

Provide **one-to-one**, **synchronous** request-reply communication.

- Client sends a request and waits for response
- Server processes the request and sends back a reply
- Blocking operation: client waits for the response
- Best for: occasional tasks, computations (e.g., inverse kinematics, spawn entities)

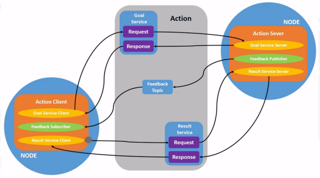


ROS 2 Communication: Actions

Definition

Designed for **asynchronous**, **long-running** tasks with feedback and the ability to cancel.

- Client sends a goal and receives periodic feedback
- Server can send progress updates while executing
- Goals can be cancelled by the client
- Result is sent when the action completes
- Best for: navigation, manipulation, time-consuming operations



Action communication pattern (click for tutorial)

Installing ROS 2

4 Installing ROS 2

Ubuntu install of ROS 2 Humble I

ROS 2 Humble supports Ubuntu 22.04 (Jammy Jellyfish).
Installation instructions

1. **Set locale**:

```
$ locale  # check for UTF-8
$ sudo apt update && sudo apt install locales
$ sudo locale-gen en_US en_US.UTF-8
$ sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
$ export LANG=en_US.UTF-8
```

2. **Setup Sources**:

```
$ sudo apt install software-properties-common
$ sudo add-apt-repository universe
$ sudo apt update && sudo apt install curl -y
$ sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/
    keyrings/ros-archive-keyring.gpg
```

Ubuntu install of ROS 2 Humble II

```
$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg]
      http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo $UBUNTU_CODENAME) main" | sudo
     tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

3 **Install ROS 2 packages**:

```
$ sudo apt update
$ sudo apt upgrade
$ sudo apt install ros-humble-desktop
```

4 **Environment setup**:

```
$ source /opt/ros/humble/setup.bash
$ echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
```

5 **Install development tools**:

```
$ sudo apt install ros-dev-tools
```

Setting up Docker image for ROS 2 I

1. **Download the image from Docker Hub**:

```
$ docker pull osrf/ros:humble-desktop-full
```

2. **Launch the container using Rocker**:

```
$ rocker --nvidia --x11 --network host --name ros2-humble --volume <path_to_shared_folder>:/root/
     ros2_ws -- docker.io/osrf/ros:humble-desktop-full
```

- -nvidia option only if the computer has a graphic card
- -volume for sharing folder between host and container

3. **To open a new terminal inside the container**:

```
$ docker exec -it ros2-humble /bin/bash
```

Setting up Docker image for ROS 2 II

4. **To source the ROS 2 environment**:

```
$ source /opt/ros/humble/setup.bash
```

5. **To source the workspace** (after building):

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

Verify Installation

Test that ROS 2 is installed correctly:

```
$ ros2 --help
```

You should see a list of available commands:

```
usage: ros2 [-h] Call 'ros2 <command> -h' for more detailed usage. ...

ros2 is an extensible command-line tool for ROS 2.
  ...
```

Main commands:

```
ros2 node      # Node introspection tools
ros2 topic     # Topic introspection tools
ros2 service   # Service introspection tools
ros2 action    # Action introspection tools
ros2 param     # Parameter introspection tools
ros2 bag       # Recording and playback tools
ros2 run       # Run a node
ros2 launch    # Launch files
ros2 pkg       # Package tools
ros2 interface # Interface introspection
```

ROS 2 Workspace

Creating a ROS 2 Workspace I

A ROS 2 workspace is organized differently from ROS 1:

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

Build the workspace using colcon (Inside the ros2_ws directory):

```
$ colcon build
```

After building, you'll have these directories:

```
ros2_ws/
  build/     # Build artifacts
  install/   # Installation files
  log/       # Build logs
  src/       # Source code
```

Creating a ROS 2 Workspace II

Source the workspace:

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

To automatically source on every new terminal:

```
$ echo "source ~/ros2_ws/install/setup.bash" >> ~/.bashrc
```

Build specific packages

```
$ colcon build --packages-select <package_name>
```

Installing Turtlesim

Turtlesim is a lightweight simulator for learning ROS 2 concepts.
Learn more about Turtlesim.

Install on Ubuntu

```
$ sudo apt update
$ sudo apt install ros-humble-turtlesim
```

Verify installation

Check if turtlesim package is available:

```
$ ros2 pkg executables turtlesim
turtlesim draw_square
turtlesim mimic
turtlesim turtle_teleop_key
turtlesim turtlesim_node
```

Running Turtlesim

Start the turtlesim node:

```
$ ros2 run turtlesim turtlesim_node
```

A window will appear with a turtle in the center.

In a new terminal, start the teleop node:

```
$ ros2 run turtlesim turtle_teleop_key
```

Use arrow keys to control the turtle!

Tip

Make sure the terminal with `turtle_teleop_key` is active (in focus) when pressing arrow keys.

Turtlesim in Docker

If running ROS 2 in Docker, you need X11 forwarding for GUI applications.

Using Rocker (recommended)

```
$ rocker --x11 --nvidia osrf/ros:humble-desktop-full
```

Inside the container

```
$ source /opt/ros/humble/setup.bash
$ apt update && apt install -y ros-humble-turtlesim
$ ros2 run turtlesim turtlesim_node
```

Note

The `--x11` flag enables X11 forwarding for displaying GUI windows.

Working with Nodes I

List running nodes:

```
$ ros2 node list
```

Get node information:

```
$ ros2 node info /node_name
```

Run a node:

```
$ ros2 run <package_name> <executable_name>
```

Example with turtlesim:

```
$ sudo apt install ros-humble-turtlesim
$ ros2 run turtlesim turtlesim_node
```

Working with Nodes II

Remap node name:

```
$ ros2 run turtlesim turtlesim_node --ros-args --remap __node:=my_turtle
```

Get node info:

```
$ ros2 node info /turtlesim
Node [/turtlesim]
  Subscribers:
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Publishers:
    /turtle1/color_sensor: turtlesim/msg/Color
    /turtle1/pose: turtlesim/msg/Pose
  Service Servers:
    /clear: std_srvs/srv/Empty
    /kill: turtlesim/srv/Kill
    ...
```

Working with Topics I

List topics:

```
$ ros2 topic list
$ ros2 topic list -t  # Show message types
```

Echo topic messages:

```
$ ros2 topic echo /turtle1/cmd_vel
```

Get topic info:

```
$ ros2 topic info /turtle1/cmd_vel
Type: geometry_msgs/msg/Twist
Publisher count: 1
Subscription count: 1
```

Working with Topics II

Publish to a topic:

```
$ ros2 topic pub /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x:
     0.0, y: 0.0, z: 1.8}}"
```

Add `--once` to publish once or `--rate 1` for continuous publishing at 1 Hz.

Get publishing frequency:

```
$ ros2 topic hz /turtle1/pose
average rate: 62.506
  min: 0.014s max: 0.018s std dev: 0.00070s window: 64
```

Get bandwidth:

```
$ ros2 topic bw /turtle1/pose
```

Working with Services I

List services:

```
$ ros2 service list
$ ros2 service list -t  # Show service types
```

Get service type:

```
$ ros2 service type /clear
std_srvs/srv/Empty
```

Find services by type:

```
$ ros2 service find std_srvs/srv/Empty
/clear
/reset
```

Working with Services II

Call a service:

```
$ ros2 service call /clear std_srvs/srv/Empty
```

Spawn a new turtle:

```
$ ros2 service call /spawn turtlesim/srv/Spawn "{x: 2.0, y: 2.0, theta: 0.2, name: ’turtle2’}"
```

Show service interface:

```
$ ros2 interface show turtlesim/srv/Spawn
float32 x
float32 y
float32 theta
string name
---
string name
```

Working with Parameters I

List parameters:

```
$ ros2 param list
```

Get parameter value:

```
$ ros2 param get /turtlesim background_r
Integer value is: 69
```

Set parameter value:

```
$ ros2 param set /turtlesim background_r 150
Set parameter successful
```

Working with Parameters II

Dump parameters to file:

```
$ ros2 param dump /turtlesim
Saving to:  ./turtlesim.yaml
```

Load parameters from file:

```
$ ros2 param load /turtlesim turtlesim.yaml
```

Start node with parameters:

```
$ ros2 run turtlesim turtlesim_node --ros-args --params-file turtlesim.yaml
```

Working with Actions I

Actions are for long-running tasks with feedback.
List actions:

```
$ ros2 action list
$ ros2 action list -t  # Show action types
```

Get action info:

```
$ ros2 action info /turtle1/rotate_absolute
Action: /turtle1/rotate_absolute
Action clients: 0
Action servers: 1
    /turtlesim
```

Working with Actions II

Show action interface:

```
$ ros2 interface show turtlesim/action/RotateAbsolute
# Goal
float32 theta
---
# Result
float32 delta
---
# Feedback
float32 remaining
```

Send action goal:

```
$ ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: 1.57}" --feedback
```

Creating ROS 2 Packages

Creating a Package I

ROS 2 supports two build types: **ament_cmake** and **ament_python**.
Create a Python package:

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

Create a C++ package:

```
$ ros2 pkg create --build-type ament_cmake --node-name my_node my_package --dependencies rclcpp std_msgs
```

Creating a Package II

Package structure (Python):

```
my_package/
  my_package/
    __init__.py
    my_node.py
  resource/
    my_package
  test/
  package.xml
  setup.py
  setup.cfg
```

Creating a Package III

Package structure (C++):

```
my_package/
  include/
    my_package/
  src/
    my_node.cpp
  CMakeLists.txt
  package.xml
```

Building and Running

Build the package:

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

Source the workspace:

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

Run the node:

```
$ ros2 run my_package my_node
```

Tip

Use `--symlink-install` flag for Python packages to avoid rebuilding after code changes:

```
$ colcon build --symlink-install
```

Writing Nodes in Python

Simple Publisher (Python) I

Create file publisher_member_function.py:

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

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()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
```

Simple Publisher (Python) II

```python
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()
```

Simple Publisher (Python) III

Key concepts:

- `rclpy.init()`: Initialize ROS 2 Python client library
- `Node`: Base class for ROS 2 nodes
- `create_publisher()`: Create a publisher with topic name, message type, and QoS
- `create_timer()`: Create a timer for periodic callbacks
- `rclpy.spin()`: Keep the node running
- `get_logger()`: Built-in logging functionality

Simple Subscriber (Python) I

Create file `subscriber_member_function.py`:

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalSubscriber(Node):
    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            'topic',
            self.listener_callback,
            10)
        self.subscription  # prevent unused variable warning

    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s"' % msg.data)

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

Simple Subscriber (Python) II

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

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

Simple Subscriber (Python) III

Add to setup.py:

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

Build and run:

```
$ cd ~/ros2_ws
$ colcon build --packages-select my_package
$ source install/setup.bash
$ ros2 run my_package talker
```

In another terminal:

```
$ ros2 run my_package listener
```

Service Server (Python) I

Create file `service_member_function.py`:

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

class MinimalService(Node):
    def __init__(self):
        super().__init__('minimal_service')
        self.srv = self.create_service(
            AddTwoInts,
            'add_two_ints',
            self.add_two_ints_callback)

    def add_two_ints_callback(self, request, response):
        response.sum = request.a + request.b
        self.get_logger().info(
            'Incoming request\na: %d b: %d' % (request.a, request.b))
        return response

def main(args=None):
```

Service Server (Python) II

```
    rclpy.init(args=args)
    minimal_service = MinimalService()
    rclpy.spin(minimal_service)
    rclpy.shutdown()

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

Service Client (Python) I

Create file `client_member_function.py`:

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

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

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

Service Client (Python) II

```python
def main(args=None):
    rclpy.init(args=args)
    minimal_client = MinimalClientAsync()
    response = minimal_client.send_request(int(sys.argv[1]), int(sys.argv[2]))
    minimal_client.get_logger().info(
        'Result of add_two_ints: %d + %d = %d' %
        (int(sys.argv[1]), int(sys.argv[2]), response.sum))
    minimal_client.destroy_node()
    rclpy.shutdown()

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

Launch Files

ROS 2 Launch Files I

Launch files in ROS 2 can be written in **Python**, XML, or YAML.
Python launch file (`my_launch.py`):

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

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            executable='turtle_teleop_key',
            name='teleop',
            prefix='xterm -e',
            output='screen'
        ),
    ])
```

ROS 2 Launch Files II

Create launch directory:

```
$ mkdir -p ~/ros2_ws/src/my_package/launch
```

Update setup.py to include launch files:

```
import os
from glob import glob
# ...

data_files=[
    # ...
    (os.path.join('share', package_name, 'launch'),
     glob('launch/*.py')),
],
```

ROS 2 Launch Files III

Run launch file:

```
$ ros2 launch my_package my_launch.py
```

Launch with arguments:

```
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration

def generate_launch_description():
    return LaunchDescription([
        DeclareLaunchArgument('use_sim_time', default_value='false'),
        Node(
            package='my_package',
            executable='my_node',
            parameters=[{'use_sim_time': LaunchConfiguration('use_sim_time')}]
        ),
    ])
```

```
$ ros2 launch my_package my_launch.py use_sim_time:=true
```

Custom Interfaces

Creating Custom Messages I

Create interface package:

```
$ ros2 pkg create --build-type ament_cmake tutorial_interfaces
```

Create directory structure:

```
$ mkdir -p tutorial_interfaces/msg
$ mkdir -p tutorial_interfaces/srv
$ mkdir -p tutorial_interfaces/action
```

Create custom message (`msg/Num.msg`):

```
int64 num
```

Create custom message with nested types (`msg/Sphere.msg`):

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

Creating Custom Messages II

Update CMakeLists.txt:

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

rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/Num.msg"
  "msg/Sphere.msg"
  DEPENDENCIES geometry_msgs
)
```

Update 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>
```

Creating Custom Messages III

Build the interface package:

```
$ cd ~/ros2_ws
$ colcon build --packages-select tutorial_interfaces
$ source install/setup.bash
```

Verify the interface:

```
$ ros2 interface show tutorial_interfaces/msg/Num
int64 num
```

Use in your node:

```
from tutorial_interfaces.msg import Num

# In your node:
self.publisher_ = self.create_publisher(Num, 'topic', 10)
msg = Num()
msg.num = 42
```

Creating Custom Services I

Create service definition (`srv/AddThreeInts.srv`):

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

Update CMakeLists.txt:

```
rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/Num.msg"
  "srv/AddThreeInts.srv"
)
```

Build and verify:

```
$ colcon build --packages-select tutorial_interfaces
$ ros2 interface show tutorial_interfaces/srv/AddThreeInts
```

Quality of Service (QoS) I

What is QoS?

Quality of Service policies allow you to configure the behavior of communication in ROS 2:

- **Reliability**: Reliable (guaranteed delivery) vs Best Effort
- **Durability**: Transient Local (late joiners get last message) vs Volatile
- **History**: Keep Last N messages vs Keep All
- **Deadline**: Maximum time between messages
- **Lifespan**: Maximum age of a message

Quality of Service (QoS) II

QoS Profiles

ROS 2 provides predefined QoS profiles:

- **Sensor Data**: Best effort, volatile (e.g., camera images)
- **Parameters**: Reliable, volatile (e.g., configuration)
- **Services**: Reliable, volatile
- **System Default**: Reliable, volatile, keep last 10

Important

Publishers and subscribers must have **compatible** QoS policies to communicate!

ROS 2 Tools and Visualization

ROS 2 Bag I

rosbag2 is used for recording and playing back topic data.
Record topics:

```
$ ros2 bag record /turtle1/cmd_vel /turtle1/pose
```

Record all topics:

```
$ ros2 bag record -a
```

Record with output name:

```
$ ros2 bag record -o my_bag /topic1 /topic2
```

ROS 2 Bag II

Get bag info:

```
$ ros2 bag info my_bag
```

Play bag:

```
$ ros2 bag play my_bag
```

Play at different rate:

```
$ ros2 bag play my_bag --rate 2.0  # 2x speed
```

Play in loop:

```
$ ros2 bag play my_bag --loop
```

RQT and Visualization Tools

rqt is a Qt-based framework for GUI development in ROS 2.
Launch rqt:

```
$ rqt
```

Useful rqt plugins:

- `rqt_graph`: Visualize node graph
- `rqt_console`: View log messages
- `rqt_plot`: Plot topic data
- `rqt_publisher`: Publish messages
- `rqt_service_caller`: Call services
- `rqt_reconfigure`: Dynamic reconfigure (limited in ROS 2)

```
$ ros2 run rqt_graph rqt_graph
$ ros2 run rqt_console rqt_console
$ ros2 run rqt_plot rqt_plot
```

Best Practices

ROS 2 Best Practices

1. **Use appropriate QoS profiles** for your application
   - Sensor data: Best effort
   - Commands: Reliable
   - State: Transient local
2. **Namespace your nodes** to avoid conflicts
3. **Use parameters** for configuration, not hardcoded values
4. **Log appropriately**: DEBUG, INFO, WARN, ERROR, FATAL
5. **Handle shutdown gracefully**: cleanup resources
6. **Use lifecycle nodes** for critical systems
7. **Write launch files** instead of manual node starting
8. **Use composition** for performance-critical applications

Package Organization

Good package structure:

```
my_robot/
  my_robot_bringup/        # Launch files
  my_robot_description/    # URDF/meshes
  my_robot_control/        # Control nodes
  my_robot_navigation/     # Navigation configuration
  my_robot_interfaces/     # Custom messages/services
  my_robot_gazebo/        # Simulation
```

Principles:

- One package = one clear purpose
- Separate interfaces from implementation
- Keep launch files in dedicated packages
- Use package dependencies appropriately

Migration from ROS 1 to ROS 2

Key Differences

- No `roscore` required
- `colcon` instead of `catkin_make`
- Different build types: `ament_cmake`, `ament_python`
- `ros2` CLI instead of separate tools
- Different client libraries APIs
- QoS policies
- Launch files in Python (preferred)
- Different parameter handling

Tools

`ros1_bridge` allows ROS 1 and ROS 2 nodes to communicate

Practical Exercise

Exercise: Publisher-Subscriber System

Create a temperature monitoring system:

1. Create a package called `temperature_monitor`
2. Create a publisher node that:
   - Publishes random temperature values (15-35℃) at 1 Hz
   - Uses a custom message with: temperature (float), timestamp (time), sensor_id (string)
3. Create a subscriber node that:
   - Subscribes to temperature data
   - Logs a warning if temperature > 30℃
   - Logs an error if temperature > 32℃
4. Create a service that returns:
   - Average temperature
   - Min and max temperatures
5. Create a launch file to start all nodes

Exercise: Solution Structure

```
temperature_monitor/
  temperature_monitor/
    __init__.py
    temperature_publisher.py
    temperature_subscriber.py
    temperature_service.py
  launch/
    temperature_system.launch.py
  package.xml
  setup.py
  setup.cfg

temperature_interfaces/
  msg/
    Temperature.msg
  srv/
    GetStats.srv
  CMakeLists.txt
  package.xml
```

Time: 30-45 minutes

Resources and Next Steps

Learning Resources

- **Official Documentation**: `https://docs.ros.org/en/humble/`
- **ROS 2 Design**: `https://design.ros2.org/`
- **ROS Discourse**: `https://discourse.ros.org/`
- **ROS Answers**: `https://answers.ros.org/`
- **GitHub**: `https://github.com/ros2`
- **ROS Index**: `https://index.ros.org/`

Books

- "A Concise Introduction to Robot Programming with ROS2" by F. Martín Rico
- "ROS 2 for Beginners" (Online courses)

Next Steps

1. **Practice with turtlesim** - master the basics
2. **Learn URDF** - robot description format
3. **Study TF2** - coordinate frame transformations
4. **Navigation stack** - autonomous navigation
5. **MoveIt 2** - motion planning
6. **Gazebo** - robot simulation
7. **RViz2** - 3D visualization
8. **Real robots** - apply to physical systems
9. **Contribute** - give back to the community

Common ROS 2 Packages

geometry2	TF2 and related tools
image_transport	Image topic infrastructure
laser_geometry	Convert laser scans to point clouds
navigation2	Navigation stack
moveit2	Motion planning framework
gazebo_ros_pkgs	Gazebo simulation
ros2_control	Hardware control framework
rqt	Qt-based GUI tools
rviz2	3D visualization
rosbag2	Data recording and playback

Summary

What we covered

- ROS 2 architecture and improvements over ROS 1
- Installation and workspace setup
- Core concepts: nodes, topics, services, actions, parameters
- Quality of Service (QoS) policies
- CLI tools for introspection
- Creating packages and custom interfaces
- Writing publishers, subscribers, services in Python
- Launch files
- ROS 2 bags and visualization tools
- Advanced topics: lifecycle, executors, composition
- Best practices

Questions?

¿Preguntas?

Thank you for your attention!

Contact
Claudia Álvarez Aparicio
`calvaa@unileon.es`

Robot Operating System (ROS)

Getting started

Claudia Álvarez Aparicio
calvaa@unileon.es

Seguridad en Sistemas Ciberfísicos

Curso 2023–2024