ROS2 Foundations
Day 1 – Basics, DDS & Tooling

From zero to two communicating nodes

Thomas Birchler

UEB

Agenda

Theory

- Why ROS2?
- DDS & ROS2 architecture
- Workspaces, packages, build system
- How to build Nodes, Topics, (Services, Actions)

Lab – 2h

- Create new workspace & package
- Create a listener and publisher Node
- (Create a service)
- (Create an action)

Learning outcome: build & run two ROS2 nodes that exchange custom messages.

Learning Objectives

- Articulate the need for distributed middleware in modern robotics.
- Sketch the ROS2 software stack from DDS up to application layer.
- Differentiate Topics, Services, and Actions and their QoS policies.
- Navigate a ROS2 workspace; create & build a package with `colcon`.

What is ROS 2?

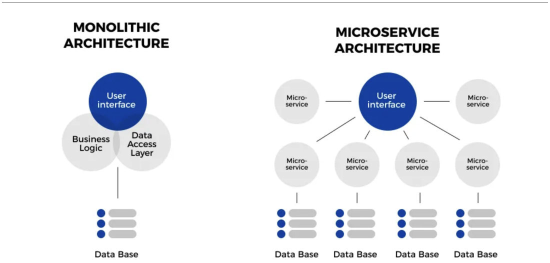
- **Robotics middleware framework** for developing modular, distributed robot software.
- **Provides infrastructure** for communication between processes: topics, services, actions.
- **Built on DDS (Data Distribution Service)** for real-time, scalable, peer-to-peer communication.
- **Language-agnostic**: supports nodes written in C++, Python, Rust, etc.

Core Idea

ROS 2 is the *glue* that lets independently developed software modules collaborate in a distributed robotic system.

Why shift from monolithic stacks? Why ROS2?

- **Scalability**: single-process control software quickly becomes unmaintainable.
- **Heterogeneity**: sensors, actuators, and AI modules often run on different OS/hardware.
- **Resilience**: process isolation prevents total failure.
- **Re-usability**: well-defined interfaces enable a vibrant ecosystem.



[1]

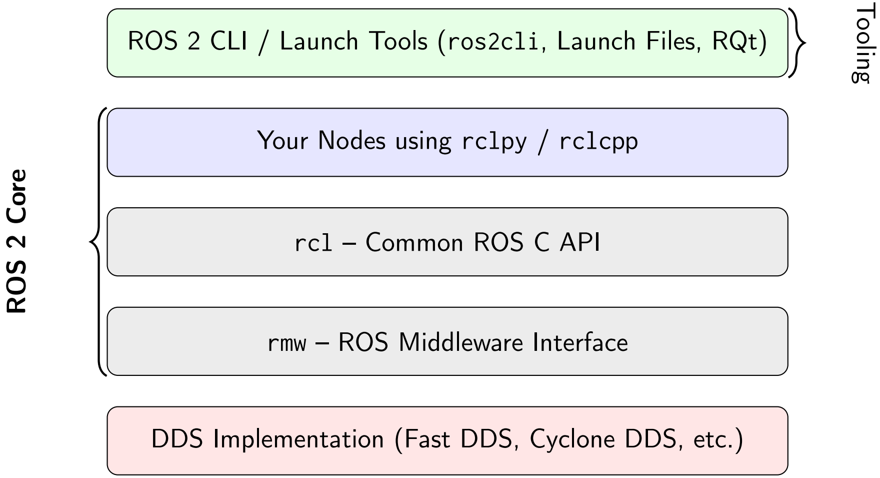
ROS 2 Software Stack

Layer	
Your Code (Python / C++ Scripts)	✓You write code here
Applications (Nodes)	✓You write code here
ROS Client Libraries `rclpy`, `rclcpp`, `rcljava`	✓You interact with these tools
ROS Middleware Interface (RMW)	? Rarely configure
DDS Middleware (Fast DDS, Cyclone DDS, etc.)	✗ Never touched
Operating System / Network Stack	✗ Never touched

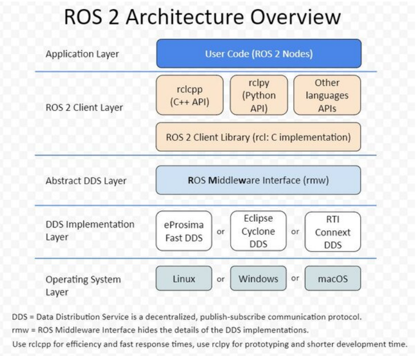
Key Idea

ROS 2 provides a layered abstraction enabling modular robotics applications with real-time, distributed communication.

ROS 2 Software Stack Overview



- **DDS**: OMG standard for publish–subscribe communication over LAN/WAN
- **Used in ROS 2** via pluggable middleware (RMW):
  - Fast DDS, Cyclone DDS, RTI Connext
- **Core concepts:**
  - **Domain**, **Participant**, **Topic**
  - **DataWriter**, **DataReader**
- **QoS Policies:**
  - Reliability, Durability, Deadline, Liveliness



Source: Lifecycle Nodes (ROS 2)

How We Use and Code with ROS 2

ROS 2 Programming Overview

- ROS 2 is organized into **packages**, each containing nodes, launch files, and configuration.
- You can write nodes in **Python (rclpy)** or **C++ (rclcpp)**.
- Communication between nodes uses **topics**, **services**, and **actions**.

ROS 2 abstracts most middleware details - focus is on message flow and system logic.

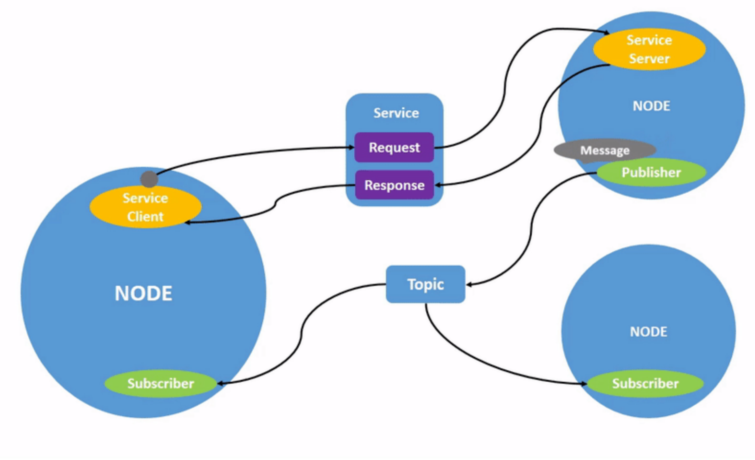
Nodes, Topics, Services, Actions

Nodes

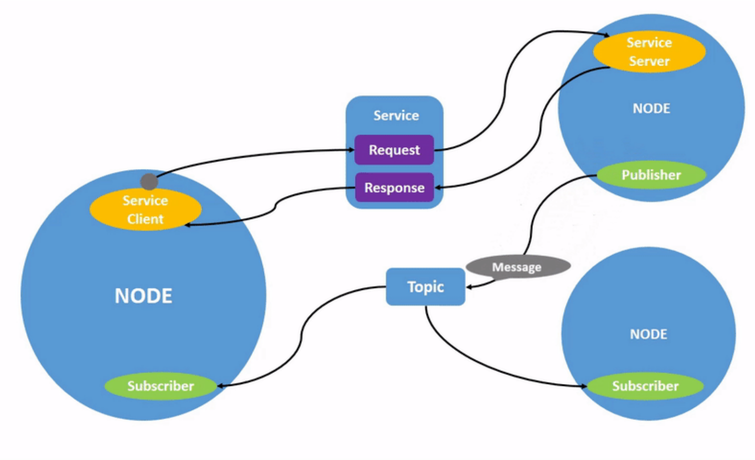
Independent processes that contain one or more executors.

Primitive	Pattern	Typical use-case
Topic	Pub-Sub (stream)	Sensor data, status updates
Service	Request-Reply	Synchronous parameter query, restart motors
Action	Goal-Feedback-Result	Long-running navigation goal with feedback

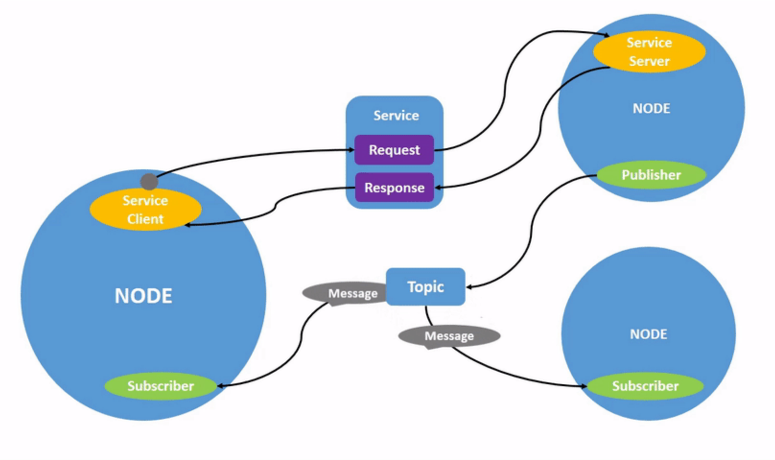
ROS 2 Node Communication Snapshot



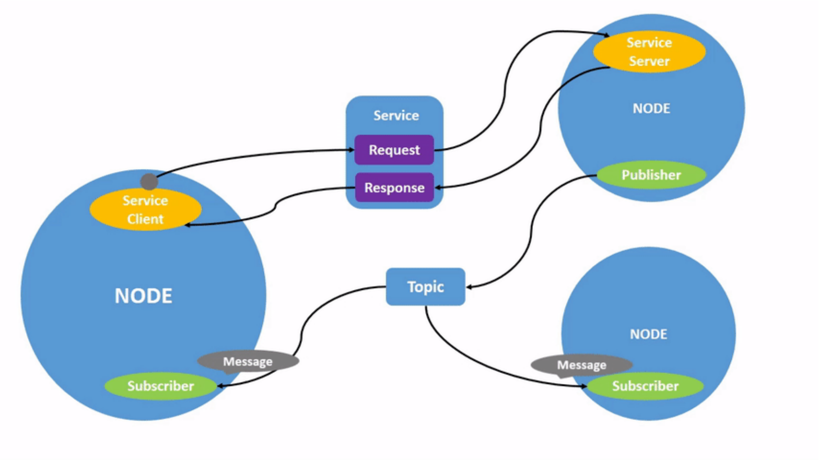
ROS 2 Node Communication Snapshot



ROS 2 Node Communication Snapshot



ROS 2 Node Communication Snapshot



ROS 2 Topics & Node Requirements

Topics: Core Pub-Sub Mechanism

- **Unidirectional**, streaming communication.
- Data is broadcast by publishers and received by any number of subscribers.
- Topics are strongly typed (each has a fixed `.msg` type).

Primitive	**Message Type?**	**Buffering?**	**Requires Name Match?**
Topic	✓(`.msg`)	✓(Queue)	✓(Topic name)
Service	✓(`.srv`)	✗	✓(Service name)
Action	✓(`.action`)	✓(Goals + Feedback)	✓(Action name)

To create a node that uses topics, services, or actions, you need:

- A unique **node name**
- The correct **interface type** (`.msg`, `.srv`, or `.action`)
- A matching **topic/service/action name**

Topic Naming Conventions & Message Types

Topic Naming

- By convention: `/namespace/robot/subsystem/data_type`
- Leading slash (/) $\rightarrow$ absolute topic; otherwise $\rightarrow$ relative to node's namespace.

Examples

- raw camera image: `/camera/image_raw`
- robot odometry: `/odom`
- velocity commands for robot1: `/robot1/cmd_vel`
- system health info: `/diagnostics`

Topic Naming Conventions & Message Types

Common Message Types

- std_msgs/String, std_msgs/Bool, std_msgs/Float32
- geometry_msgs/Twist – velocity commands
- sensor_msgs/Image – image stream
- nav_msgs/Odometry – position, velocity, orientation

sensor_msgs/Image Fields

Field	Type
header	std_msgs/msg/Header
height	uint32
width	uint32
encoding	string
is_bigendian	uint8
step	uint32
data	uint8[] (row-major image data)

Understanding QoS in ROS 2

Quality of Service (QoS) controls how ROS 2 nodes communicate over topics. It helps manage:

- Message reliability
- Message persistence
- Delivery timing

Common QoS Policies

Reliability	**RELIABLE** – ensure delivery **BEST_EFFORT** – drop if too slow
Durability	**VOLATILE** – forget messages immediately **TRANSIENT_LOCAL** – keep last messages for late subscribers
History/Depth	Buffer size (e.g., last 10 messages)
Deadline	Max interval between messages (used for monitoring timing)

Use Case Example:

IMU data at 100Hz should use `RELIABLE` to ensure no data loss.

The defaults best_effort and volatile usually work.

Recap

Up to now you learned:

- **Why ROS 2:** A modular, scalable, and real-time ready middleware.
- **Architecture:** Layered stack from your code to DDS via rcl and rmw.
- **Core Concepts:** Nodes, Topics, Services, Actions, and QoS policies.

Now we look into the foundation to build and debug ROS 2 systems:
Practical Skills:

- Setting up a workspace and creating packages
- Writing publishers and subscribers (Python)
- Building and running nodes with `colcon` and `ros2 run`

Goals of Communication Interfaces – The Radio Metaphor

Scenario

We interact with a virtual radio station broadcasting on topic `/radio/100_3fm`. We want to:

- **Listen** to the stream (pub/sub)
- **Ask** what song is playing (service)
- **Request** a specific song (action)

Goals of Communication Interfaces – The Radio Metaphor

Primitive	**Goal**	**Metaphor**
Topic	Continuous data stream	*Radio station broadcasts:* "Streaming since X seconds"
Service	Synchronous query	*Quest.: "What song is playing?"* *Reply: "Bohemian Rhapsody"*
Action	Long-running request with feedback	*You call to request a song:* → *Feedback: "20s played"* → *Result: "Your song played."*

Main Function for ROS 2 Nodes (Python)

- Every ROS 2 Python node starts with a ‘main()‘ function.
- This manages the node’s lifecycle: initialization, spinning, shutdown.

Typical Structure

```
def main(args=None):
    rclpy.init(args=args)        # Initialize ROS 2
    node = MinimalPublisher()    # Create the node instance
    rclpy.spin(node)             # Keep node alive and responsive
    node.destroy_node()          # Clean up node resources
    rclpy.shutdown()             # Shutdown ROS 2
```

Tip: This is common for both publishers and subscribers.

Publisher Node – Explained

Key Ideas:

- A **publisher** sends messages over a named topic.
- The topic has a **message type** (e.g., `std_msgs/String`).
- The node sets up a `timer` to publish at fixed intervals.

Key Methods:

- `create_publisher` defines topic + queue size.
- `create_timer` sets up a loop to trigger message sending.
- `publish(msg)` transmits the message on the topic.

Publisher Node – Explained

Core Structure

```
from std_msgs.msg import String

class Talker(Node):
    def __init__(self):
        super().__init__('talker')
        self.publisher_ =
        self.create_publisher(<data type>, '<topic>', <queue size>
       self.timer = self.create_timer(1.0, self.timer_callback)

    def timer_callback(self):
        msg = String()
        msg.data = 'Hello ROS 2'
        self.publisher_.publish(msg)
```

<data type> = String, <topic> = /radio/100_3fm, <queue size> = 10

Subscriber Node – Explained

Key Ideas:

- A **subscriber** listens to messages from a named topic.
- The topic must match the **name and message type** of the publisher.
- Incoming messages trigger a callback function.

Key Methods:

- `create_subscription` sets up the topic and callback.
- `msg` contains the received message.
- `self.get_logger().info()` prints the message.

Subscriber Node – Explained

Core Structure

```
class Listener(Node):
    def __init__(self):
        super().__init__('listener')
        self.subscription =
            self.create_subscription(
                String,
                'chatter',
                self.listener_callback,
                10)

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

Understanding ROS 2 Workspace Layout

Key Directories

- `src/` – Contains all source code and packages.
- `build/` – Stores temporary build files.
- `install/` – Holds final compiled outputs.
- `log/` – Includes logs from builds and tests.

1. Create a New Workspace

- A workspace is where you develop, build, and run ROS 2 packages.

Commands

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

- Initialize with a first build:

Build

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

2. Source the Environment

- To use ROS 2 and your own packages, your **shell environment** must be configured.
- This is done by `source`-ing setup scripts.
- This must be done in each new terminal.

Source Setup Files

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

Tip: Add to a script or your `.bashrc` *if using frequently.*

echo "source /opt/ros/humble/setup.bash" » /home/dev/.bashrc

echo "source /ros2_ws/install/setup.bash" » /.bashrc

3. Create a New ROS 2 Package

- Use the official ROS 2 CLI to scaffold a Python package:

Command for Python

```
cd ~/ros2_ws/src
ros2 pkg create --build-type ament_python <my_package>
```

use radio_communication for <>.

Command for C++

```
ros2 pkg create --build-type ament_cmake \\
        <my_cpp_package> --dependencies rclcpp std_msgs
```

- Creates folder structure + setup files
- Check ‘my_package/setup.py‘ and ‘package.xml‘

4. Add a Python Node

File: my_package/my_package/my_node.py

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

class MyNode(Node):
    def __init__(self):
        super().__init__('my_node')
        self.get_logger().info("Node started!")

def main():
    rclpy.init()
    node = MyNode()
    rclpy.spin(node)
    rclpy.shutdown()
```

5. Update setup.py Entry Points

- Register your Python node so ROS 2 can run it.

Inside setup.py, add:

```
entry_points={
    'console_scripts': [
        '<my_node> = my_package.my_node:main'
    ],
},
```

Also make sure `my_node.py` *is executable.*

```
chmod +x my_package/my_package/my_node.py
```

6. Build the Workspace

- Go back to the root of your workspace and build:

Build

```
cd ~/ros2_ws
colcon build --symlink-install
```

- Source the overlay after build:

Source Overlay

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

Build types: ament_cmake, ament_python, ament_cplusplus

7. Run the Node

- You can now start your Python node using the CLI:

Command

```
ros2 run my_package my_node
```

You should see the "Node started!" message.

Wrap-Up

1. ROS2 leverages DDS for deterministic pub-sub.
2. Nodes communicate via Topics, Services, Actions with tunable QoS.
3. Workspaces & packages isolate projects; `colcon` automates builds.

References I

Microservices vs. Monolithic. *https://medium.com/javanlabs/micro-services-versus-monolithic-architecture-what-are-they-e17ddc8d3910*

Lifecycle Nodes. `https://foxglove.dev/blog/how-to-use-ros2-lifecycle-nodes`