Part 1 Getting Started with ROS2

- **Nodes**: These are the individual components, programs or modules in a distributed system. they can be
  - Instructions to print logs in terminal
  - Graphical windows (2D, 3D)
  - Hardware drivers, etc
- ROS system is made up of many nodes which communicate by sending/receiving messages, are connected by edges.
- **Edge** → Represents a stream of messages between two nodes (a ROS graph edge)
- **Nodes = POSIX processes**: This means each node usually runs as a separate operating system process that follows the POSIX (Portable Operating System Interface) standard — basically, a normal, independent UNIX/Linux process.
- **Edges = TCP connections**: The communication links between nodes are usually TCP network connections (like sockets), not shared memory or in-process function calls.

> This setup encourages **loose coupling**: each part of the system operates independently and communicates over standard protocols.

- Since each node is an independent process, if one crashes (e.g., due to a bug or exception), it **doesn't bring down the whole system**. Only that one process is affected.

> This increases reliability, because the rest of the system can keep working.

- The system is modeled as a **graph**, where:
  - **Nodes** = processes
  - **Edges** = message-passing communication
- If a single node fails, the **rest of the nodes keep communicating and functioning**, since they are decoupled and independent.
- If the system **logs incoming messages to a node**, you can **reproduce the exact sequence** of events that led to the crash.
- By **replaying these messages in a debugger**, you can step through what happened and diagnose the issue — a powerful way to **debug distributed systems**.

This architecture uses independent processes (nodes) and message-passing (TCP), allowing the system to survive individual component failures. Because processes are isolated, a crash affects only one node. By logging inputs, developers can replay and debug failures — making the system not only resilient but also easier to diagnose.

- **Loosely coupled**: Components (nodes) are independent and communicate via messages, not tightly integrated code.

- **Graph-based**: The system is organized as a network of nodes passing messages (data).
- **Rapid-prototype**: You can **quickly test and iterate** on different designs without writing a lot of extra "glue code" to make components work together.
- You don’t have to rewrite big chunks of the system just to test one part.
- For example, in a robot that needs to find and fetch an item, the **object recognition node** can be replaced easily.
- You can try a new object detection algorithm by just running a different process that:
  - Takes images as input
  - Outputs labeled objects
- As long as the new node **respects the same input/output interface**, the rest of the system doesn’t need to change.
- You’re not limited to replacing individual nodes — you can:
  - Replace **entire parts of the system**, such as navigation or perception subsystems.
  - Do this **at runtime**, without rebooting or restarting the whole system.
- Examples of flexible swapping:
  - Use a **simulator** instead of real hardware for testing.
  - Try different **navigation strategies**.
  - Modify and recompile an **algorithm**, then hot-swap it into the system.
- ROS automatically handles:
  - Network connections
  - Data routing
  - Message passing
- This means the developer doesn’t have to manually set up communication — it **just works**, making the system **interactive and modular**.
- Encourages **experimentation, iteration, and innovation** — which is ideal for robotics R&D.

ROS's graph-based, loosely coupled design allows developers to easily experiment by hot-swapping parts of a system — from single nodes to entire subsystems — without changing the rest of the architecture. It makes prototyping faster, debugging easier, and innovation smoother by automating the underlying communication layer.

log refers to a recorded message or set of messages that provide information about the operation of a system, node, or program. Logs are essential for **debugging, monitoring, and understanding the behavior** of your applications.

log is typically a text output that includes:

- **Timestamps**: When the event occurred.
- **Log level**: The severity or type of message (e.g., `INFO`, `WARN`, `ERROR`, `DEBUG`, `FATAL`).
- **Node name**: Which node generated the log.
- **Message**: The actual content or description.

Example:

```
[INFO] [1680199215.923849347] [turtlesim]: Starting turtlesim with turtle1
```

Common Uses of Logs:

- Tracking the startup and shutdown of nodes.
- Debugging unexpected behavior (e.g., errors or warnings).
- Verifying that published/subscribed topics and services are working.
- Recording runtime information for future analysis.

Log Levels

Level	Meaning
DEBUG	Detailed internal information (for devs)
INFO	General information about system operation
WARN	Something unexpected, but not fatal
ERROR	A serious issue that affects operation
FATAL	A critical issue causing termination

Viewing Logs in ROS 2:

1. Logs are automatically printed in the terminal when you run nodes.
2. ROS 2 also stores logs in a directory:

   ```
   ~/.ros/log/
   ```

roscore: A **service** that provides connection info to nodes so that the transmission of messages can take place.

- **Used only by nodes** to:
  - Locate their peers
  - Register themselves

> It does **not** pass messages.

To enable coordination between two nodes:

- Each node must know where the `roscore` is running
- Therefore, ROS nodes need to set `ROS_MASTER_URI`

The `ROS_MASTER_URI` tells nodes how to reach the `roscore` .

Example:

```
arduinohttp://hostname:11311/
```

- `roscore` is running on a computer named `hostname`
- It’s listening on port `11311` (a **prime** and **palindromic** number)

- `ROS_` → Indicates the variable is part of the ROS environment
- `MASTER` → Central coordinating process
  - Helps nodes discover and register with each other
  - Does **not** handle actual data transfer
- `URI` → Uniform Resource Identifier (standardized way to specify resource address)

`ROS_MASTER_URI` literally means the **location (URI)** where the ROS Master can be contacted.

- It is an environment variable used by every node to find the Master when starting up. Without it, nodes can’t register themselves or locate peers.

Starting a Node, Template: Nodes are organized into packages

- You must specify:
  - Package name (where the node is)
  - Executable file for that node

```
`ros2 run package <package> <executable>`
```

Example:

```
ros2 run demo_nodes_cpp
```

- This runs a program that prints a log in the terminal every second.

Basic Commands

cmd	desc
`ros2 run demo_nodes_cpp talker`	prints log in the terminal every second

cmd	desc
`ros2 run demo_nodes_cpp listener`	prints whatever is received from talker
`ctrl + c`	Terminate the currently running node
`ros2 -h`	Global help info
`ros2 run -h`	Help for the run sub-command
`ros2 rqt-graph`	shows visual representation of nodes

Turtlesim

`ros2 run turtlesim turtlesim_node`	**Start the turtlesim simulator**
`ros2 run turtlesim turtle_teleop_key`	Start the keyboard control for turtlesim

- keyboard teleoperation key node that lets you control turtle in turtlesim window using arrow keys.

Topic

Command	Description
`ros2 topic`	Access the ros2 topic command group
`ros2 topic -h`	Display help information for topic commands
`ros2 topic list`	List all active topics in the ROS2 system
`ros2 topic info <topic_name>`	Display information about a specific topic

Interface

Command	Description
`ros2 interface show <interface_name>`	Display the definition of a message, service, or action

Edge → Represents a stream of messages between two nodes (a ROS graph edge)

Command	Description
`ros2 node list`	List all active nodes
`ros2 topic echo <topic_name>`	Display messages published on a topic

Command	Description
`ros2 topic pub <topic_name> <msg_type> <values>`	Publish message to a topic
`ros2 service list`	List all active services
`ros2 param list`	List all parameters of ROS2 nodes
`ros2 launch <package> <launch_file>`	Run a launch file

Common Options

Option	Description
`-h, --help`	Display help information
`--ros-args`	Pass arguments to ROS 2 (e.g., parameters)
`-r, --remap`	Remap ROS resources (e.g., topics)
`-p, --parameter`	Set a parameter

Services

Communication mechanism used for **synchronous, two-way communication** between nodes. It allows a node (client) to send a **request** and wait for a **response** from another node (server). The server processes the request and returns a response to the client node.
They have been designed for quick execution, for example: a computation, or an immediate action, such as spawning a turtle on a screen.

ROS 2 provides a pre-built service example:

```
ros2 run demo_nodes_cpp add_two_ints_server
ros2 run demo_nodes_cpp add_two_ints_client
```

- The server offers a service `/add_two_ints`
- The client sends two integers (e.g., `a = 2`, `b = 3`) and receives the sum (`5`)

Use services when:

- You want a **request → reply** interaction
- The task is **short-lived** (e.g., one calculation, setting a parameter)
- You don’t need continuous data or progress updates

```
ros2 run demo_nodes_cpp add_two_ints_client
```

- **Starts a client node** that:

- Sends a request to `/add_two_ints` with two integers
- Waits for the response (the sum)
- Prints the result in the terminal

Usage

1. In one terminal:

   ```
   ros2 run demo_nodes_cpp add_two_ints_server
   ```

   This will start the service server.
2. In another terminal:

   ```
   ros2 run demo_nodes_cpp add_two_ints_client
   ```

   This will send a request (default values: `a=2`, `b=3`) and receive the sum (`5`).

Command	**Description**
`ros2 service list`	Lists all available services in the ROS 2 system
`ros2 service type <service_name>`	Displays the service type (e.g., `example_interfaces/srv/AddTwoInts`)
`ros2 interface show <interface_name>`	Shows the structure of the service type (fields and data types)
`ros2 service call <service_name> <type>`	Calls a service manually with input arguments
`ros2 service info <service_name>`	Displays information about a service, such as its node and type

Interfaces define the structure of data used in:

- **Messages** (`.msg`)
- **Services** (`.srv`)
- **Actions** (`.action`)

Actions

Basically the same thing as a service (client/server communication), but designed for longer tasks, and when you might want to also get some feedback during the execution, be able to cancel the execution, and so on.
Actions are used whenever a client/server communication might take more time and we want more control over it.
It is defined by a name and an interface. This time, the interface contains three parts: **goal,**

result, and feedback.
The goal and result are similar to the request and response for a service. The feedback is additional data that can be sent by the server to give some feedback during the goal execution.

Command	Description
`ros2 action list`	Lists all available action names on the ROS 2 system
`ros2 action info <action_name>`	Shows info about the action: number of clients/servers
`ros2 action info <action_name> -t`	Shows the action type (e.g., `turtlesim/action/RotateAbsolute`)
`ros2 interface show <action_type>`	Displays the structure of the action (goal, result, feedback)
`ros2 action send_goal <name> <type> "<goal>"`	Sends a goal to the action server (must match the action type format)
`ros2 action send_goal -f <name> <type> "<goal>"`	Sends a goal in the background (non-blocking)
`ros2 action list -t`	Lists actions with their types
`ros2 action cancel <goal_id>`	Cancels a goal in progress (requires the goal ID)
`ros2 action status <action_name>`	Shows active goals and their statuses for an action

Example

Command	Explanation
`ros2 action list`	Shows `/turtle1/rotate_absolute`
`ros2 action info /turtle1/rotate_absolute -t`	Confirms type is `turtlesim/action/RotateAbsolute`
`ros2 interface show turtlesim/action/RotateAbsolute`	Shows goal/result/feedback fields
`ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: 1.0}"`	Sends a goal to rotate the turtle to 1 radian

Usage:

- Use **Topics** when:
  - You need continuous data flow (e.g., images, sensor readings).
  - You don’t need acknowledgment or feedback.

- Use **Services** when:
  - You need a request/response pattern, such as querying data or triggering a quick task.
  - The task is **short** and doesn't need feedback or progress tracking.
- Use **Actions** when:
  - You're running a **long** or **interruptible** task.
  - You want **feedback**, **result**, and the **ability to cancel** the task.

Interface defines the structure of the data exchanged between nodes. It serves as a formal **contract** between nodes, specifying what kind of data they send or receive. Interfaces are the foundation of communication in ROS.

Interface Type	File Extension	Purpose	Example
Message (`msg`)	`.msg`	For continuous data via **topics**	`std_msgs/msg/String`, `geometry_msgs/msg/Twist`
Service (`srv`)	`.srv`	For **request-response** communication	`example_interfaces/srv/AddTwoInts`
Action (`action`)	`.action`	For **long-running tasks** with feedback	`nav2_msgs/action/NavigateToPose`

1. **Message Interface (`.msg`)**

- Defines a **single data structure**.
- Used with **publishers/subscribers** (topics).
- Example: `geometry_msgs/msg/Point`

```
float64 x
float64 y
float64 z
```

1. **Service Interface (`.srv`)**

- Defines a **request** and a **response** structure.
- Used with **service clients and servers**.
- Example: `AddTwoInts.srv`

```
int64 a
int64 b
```

```
---
int64 sum
```

3. **Action Interface (`.action`)**

- Defines a **goal**, **result**, and **feedback**.
- Used with **action clients and servers**.
- Example: `Fibonacci.action`

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

Interfaces are usually organized into **interface packages** like:

- `std_msgs` – Standard message types
- `geometry_msgs` – Geometric primitives
- `sensor_msgs` – Sensor-related messages
- `example_interfaces` – Example messages, services, and actions

You can inspect an interface using:

```
ros2 interface show <interface_name>
```

For example:

```
ros2 interface show example_interfaces/srv/AddTwoInts
```

Parameters
Exist within the node, If you stop a node, then the parameters would also be destroyed. They are basically named values that nodes can use to **configure their behavior at runtime**. They serve as a flexible way to modify how a node operates without changing code.
Parameters are settings that can be provided at runtime (which means when we run the node). They allow us to easily configure the different nodes that we start, and thus, they make ROS 2 applications more dynamic.
A parameter exists within a node. You can find all parameters for a node and get the value for each
one. When starting the node, you can give a custom value for the parameters you want to modify.

```
ros2 param list
```

lists all the parameters of the node

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

Retrieves the **current value** of a specific parameter from a given node.
Get here means retrieve or **read** the current value of that parameter from the specified node.

Command	What It Does
`ros2 param list`	Lists all parameters for all nodes
`ros2 param get <node_name> <param_name>`	Gets the value of a specific parameter
`ros2 param get /turtlesim background_r`	Gets the red component of the turtlesim background
`ros2 param get /turtlesim background_g`	Gets the green component of the turtlesim background
`ros2 param get /turtlesim background_b`	Gets the blue component of the turtlesim background

Launch Files allow you to start several nodes and parameters from just one file, which means that you can start your entire application with just one command line.

For launch files, we will use the template.

```
ros2 launch <package_name> <launch_file>
```

Command	Purpose	Launches
`ros2 launch demo_nodes_cpp talker_listener_launch.py`	Runs a talker (publisher) and listener (subscriber)	Two C++ nodes communicating via a topic
`ros2 launch turtlesim multisim.launch.py`	Runs multiple turtlesim simulators	Two or more turtles with separate namespaces