Intro to ROS 2 Communication: Core Concepts Guide

This guide introduces the six foundational concepts of ROS 2 communication that students need to understand before diving into hands-on labs. It is designed for beginners working with ROS 2 Humble, Python, and Linux environments.

---

1. What is ROS 2 and Why It Exists

ROS 2 (Robot Operating System 2) is a middleware framework that enables modular, distributed robotics systems. It provides tools, libraries, and conventions for building robot applications. Unlike ROS 1, ROS 2 is built on **DDS (Data Distribution Service)**, which supports real-time communication, multi-platform compatibility, and improved security.

Key Features:

- Decentralized discovery (no single master node).
- Real-time capabilities.
- Cross-platform support (Linux, Windows, macOS).
- Enhanced security and QoS (Quality of Service) options.

---

2. ROS 2 Nodes

A **node** is the smallest executable unit in a ROS 2 system. Each node performs a specific function, such as reading sensor data, controlling motors, or processing images.

Characteristics:

- Nodes are independent processes.
- Nodes can run on the same machine or across multiple machines.
- Nodes communicate using topics, services, and actions.

Example:

- A camera_node publishes image data.
- A vision_node subscribes to image data and detects objects.

---

3. Topics and Messages

Topics are named communication channels that allow nodes to exchange data asynchronously using the **publish/subscribe** pattern.

- **Publisher**: Sends messages to a topic.

- **Subscriber**: Receives messages from a topic.
- **Message**: A strongly typed data structure (e.g., std_msgs/msg/String, geometry_msgs/msg/Twist).

Example:

- Topic: /sensor/temperature
- Publisher: Temperature sensor node.
- Subscriber: Logger node that records temperature data.

---

4. How ROS 2 Handles Message Passing Under the Hood

ROS 2 uses **DDS** as its communication backbone:

- **Discovery**: Nodes automatically find each other without a central master.
- **Serialization**: Messages are converted into a binary format for transport.
- **Transport**: DDS uses UDP, shared memory, or other mechanisms for efficient communication.

Why this matters:

- Nodes are decoupled: they only need to agree on topic name, message type, and QoS settings.
- ROS 2 adds naming conventions, remapping, and introspection tools on top of DDS.

---

5. Introspection Tools

ROS 2 provides the following basic CLI and GUI tools to observe and debug the system:

- **ros2 node list** : Lists active nodes.
- **ros2 node info <node_name>** : Displays details about a node (publishers, subscribers, services)
- **ros2 topic list** : Lists active topics.
- **ros2 topic echo <topic>** : Displays messages on a topic.
- **ros2 topic info <topic> --verbose** : Shows message type and QoS settings.
- **ros2 topic hz <topic>** : Measures publishing frequency.
- **rqt_graph** : Visualizes the ROS graph (nodes and topics).

Advanced Introspection tools:

- **ros2 run rqt_plot rqt_plot <topic name>/data** : Plots numeric topic data in real time
- **ros2 interface show <msg_type>** : Displays the structure of a message type.
- **ros2 bag record <topic_name>** : Records topic data and saves locally to a folder.

- **ros2 bag play <bag_folder>** : replays topic data that was captured during record.
- **ros2 doctor --report** : checks the health of your ROS2 installation and environment.

Why it matters:

- Helps diagnose missing connections, type mismatches, and QoS incompatibilities.
- Provides a visual understanding of the system architecture.

---

6. QoS (Quality of Service)

QoS defines **how** messages are delivered. DDS provides several QoS policies that ROS 2 exposes:

Key QoS Policies:

- **Reliability**:
  - reliable: Guarantees delivery (retries if needed).
  - best_effort: Drops messages if the network is congested.
- **Durability**:
  - volatile: Messages are not stored; late subscribers miss old data.
  - transient_local: Last message is stored and sent to late joiners (similar to ROS 1 latched topics).
- **History**:
  - KEEP_LAST: Store the last N messages.
  - KEEP_ALL: Store all messages (can use lots of memory).
- **Depth**:
  - Integer value (e.g., 10): Number of messages to store when using KEEP_LAST.

Why it matters:

- Incompatible QoS settings prevent nodes from connecting.
- QoS tuning is critical for real-time systems and unreliable networks.

Example:

- A camera publisher uses best_effort for high-speed image streaming.
- A control command subscriber uses reliable to ensure commands are never lost.

Summary Table

Concept	Key Idea
ROS 2	Middleware for modular robotics systems
Nodes	Independent processes performing tasks
Topics	Named channels for pub/sub communication
DDS Layer	Handles discovery, serialization, transport
Introspection	Tools for debugging and visualization
QoS	Policies for message delivery and reliability

Suggested Pre-Lab Exercise

1. Run the ROS 2 demo talker/listener in separate terminal windows:

```
ros2 run demo_nodes_cpp talker
ros2 run demo_nodes_cpp listener

```

2. Use introspection tools:

```
ros2 node list
ros2 node info /talker
ros2 topic list
ros2 topic echo /chatter

```

3. Open rqt_graph and observe the connection.

Next Step

With these concepts in mind, you are ready to dive into the Week 2 lab, where you will:

- Create publisher and subscriber nodes.
- Visualize the ROS graph.
- Experiment with topic names, message types, and QoS settings.