ROS 2 Fundamentals - Complete Recap & Cheat Sheet

Introduction to ROS 2

ROS 2 (Robot Operating System 2) is a framework for building robotics applications. Think of it as a "**nervous system**" for robots where different components (nodes) can communicate efficiently. Unlike ROS 1, ROS 2 is production-ready, supports multiple languages, and has better security.

Key Concept: ROS 2 uses a distributed system where independent programs (nodes) exchange information through various communication patterns.

Workspace & Package Management

Workspace Structure

```
text
```

```
ros2_ws/                    # Workspace ROOT
├── src/                    # Source code (YOUR PACKAGES)
│   ├── package_1/          # Your first package
│   ├── package_2/          # Your second package
│   └── ...
├── build/                  # Build files (AUTO-GENERATED)
├── install/                # Installed packages (AUTO-GENERATED)
└── log/                    # Build logs (AUTO-GENERATED)
```

Essential Commands

```
bash
```

```bash
# Create workspace
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws

# Create package
ros2 pkg create my_package --build-type ament_python --dependencies rclpy
```

```
# Build package
colcon build --packages-select my_package

# Source workspace (CRITICAL!)
source install/setup.bash

# List packages
ros2 pkg list
```

Remember: You must `source install/setup.bash` in every terminal and after every build!

Nodes - The Basic Building Blocks

What are Nodes?

Nodes are individual programs that perform specific tasks. A robot system typically has many nodes:

- Sensor nodes (camera, lidar)
- Processing nodes (object detection)
- Control nodes (motor control)
- Planning nodes (navigation)

Node Management Commands

```
bash
# List running nodes
ros2 node list

# Get node information
ros2 node info /node_name

# Run a node
ros2 run package_name node_name

# Remap node name
```

```
ros2 run package_name node_name --ros-args -r __node:=new_name
```

Example Node Lifecycle

```bash
# Terminal 1: Run node
ros2 run demo_nodes_cpp talker

# Terminal 2: See node info
ros2 node list
```

```
ros2 node info /talker
```

Topics - Asynchronous Communication

What are Topics?

Topics provide one-way, asynchronous communication between nodes. Think of them as radio stations:

- Publishers "broadcast" messages
- Subscribers "tune in" to receive messages
- Multiple subscribers can listen to one publisher

Topic Commands

```bash
# List all topics
ros2 topic list

# Show topic information
ros2 topic info /topic_name

# View message type
ros2 topic type /topic_name

# Monitor messages
ros2 topic echo /topic_name
```

```
# Publish message
ros2 topic pub /topic_name message_type "message_data"

# Show message structure
ros2 interface show message_type
```

Real Example with Turtlesim

```
bash
# Terminal 1: Start turtlesim
ros2 run turtlesim turtlesim_node

# Terminal 2: See available topics
ros2 topic list

# Terminal 3: Monitor turtle position
ros2 topic echo /turtle1/pose

# Terminal 4: Move turtle
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: 0.0}}"
```

Services - Synchronous Request/Response

What are Services?

Services provide synchronous request-response communication. Think of them as function calls between nodes:

- Client sends a request
- Server processes and returns a response
- Blocking operation (client waits for response)

Service Commands

```bash
# List all services
ros2 service list

# Show service type
ros2 service type /service_name

# Call a service
ros2 service call /service_name service_type "request_data"

# Show service structure
ros2 interface show service_type
```

Real Example with Turtlesim

```bash
# List available services
ros2 service list

# Spawn a new turtle
ros2 service call /spawn turtlesim/srv/Spawn "{x: 2.0, y: 2.0, theta: 0.0, name: 'turtle2'}"

# Change pen color
ros2 service call /turtle1/set_pen turtlesim/srv/SetPen "{r: 255, g: 0, b: 0, width: 5, off: 0}"

# Reset simulation
ros2 service call /reset std_srvs/srv/Empty
```

Actions - Long-running Tasks with Feedback

What are Actions?

Actions handle long-running tasks with progress feedback. Think of them as asynchronous operations:

- Client sends a goal
- Server provides periodic feedback
- Client can cancel the operation
- Server returns final result

Action Commands

```
bash
# List all actions
ros2 action list

# Show action info
ros2 action info /action_name

# Send action goal
ros2 action send_goal /action_name action_type "goal_data"

# Send goal with feedback
ros2 action send_goal /action_name action_type "goal_data" --feedback
```

Real Example

```
bash
# See available actions (turtlesim has one)
ros2 action list

# Send navigation goal with feedback
ros2 action send_goal /turtle1/rotate_absolute
turtlesim/action/RotateAbsolute "{theta: 3.14}" --feedback
```

Parameters - Runtime Configuration

What are Parameters?

Parameters are configuration values that can be changed at runtime. Think of them as settings for nodes:

- Key-value pairs
- Can be modified without restarting
- Support validation callbacks

Parameter Commands

```bash
# List all parameters
ros2 param list

# Get parameter value
ros2 param get /node_name param_name

# Set parameter value
ros2 param set /node_name param_name value

# Describe parameter
ros2 param describe /node_name param_name

# Dump parameters to file
ros2 param dump /node_name

# Load parameters from file
ros2 param load /node_name parameter_file.yaml
```

Message, Service, and Action Definitions

Creating Custom Interfaces:

```bash
# Interface package must use ament_cmake
ros2 pkg create my_interfaces --build-type ament_cmake



# Directory structure:
my_interfaces/
├── msg/
│   └── MyMessage.msg
├── srv/
│   └── MyService.srv
├── action/
│   └── MyAction.action
├── CMakeLists.txt
└── package.xml
```

Example Definitions:

```plaintext
# MyMessage.msg
string name
```

```
int32 id

float64[] data



# MyService.srv

string request_data

---

bool success

string response_message



# MyAction.action

# Goal

string target_name

---

# Result

bool success

string message

---

# Feedback

float32 progress

string status
```

Real Example with Turtlesim

```bash
# List turtlesim parameters
ros2 param list /turtlesim

# Change background color
ros2 param set /turtlesim background_r 150
ros2 param set /turtlesim background_g 150
ros2 param set /turtlesim background_b 150

# Save current settings
ros2 param dump /turtlesim > turtlesim_params.yaml
```

Communication Patterns Summary

Pattern	Use Case	Communication	Example
Topics	Continuous data	One-way, async	Sensor data, position
Services	Commands	Request-response, sync	Spawn turtle, reset
Actions	Long tasks	Goal-feedback-result, async	Navigation, manipulation
Parameters	Configuration	Key-value, sync	Colors, settings

Discovery & Debugging Commands

General Discovery

```bash
# See everything running
ros2 node list
ros2 topic list
ros2 service list
ros2 action list

# Get detailed info
ros2 node info /node_name
ros2 topic info /topic_name
ros2 service info /service_name
ros2 action info /action_name
```

Message Inspection

```bash
# See message structure
ros2 interface show geometry_msgs/msg/Twist
ros2 interface show turtlesim/srv/Spawn

# Monitor specific message field
ros2 topic echo /turtle1/pose --field x
```

Runtime Monitoring

```bash
# See node graph
rqt_graph

# Monitor all communications
ros2 topic hz /topic_name
ros2 topic bw /topic_name
```

Quick Start Workflow

1. Setup Workspace
2. bash

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

3. cd ~/ros2_ws
4. Create & Build Package
5. bash

```
ros2 pkg create my_robot --build-type ament_python --dependencies rclpy
colcon build --packages-select my_robot
```

6. source install/setup.bash
7. Run & Discover
8. bash

```
ros2 run my_robot my_node
ros2 node list
```

9. ros2 topic list
10. Monitor & Control
11. bash

```
ros2 topic echo /my_topic
ros2 service call /my_service service_type "data"
```

12. ros2 param set /my_node parameter value