Robotics: Robot Operating System

Minor in AI - IIT ROPAR

5th June, 2025

1 Robot Operating System (ROS)

1.1 Definition

The **Robot Operating System (ROS)** is a flexible framework for writing robot software. It is not an operating system in the traditional sense but provides a collection of software libraries, tools, and conventions aimed at simplifying the task of creating complex and modular robot applications.

ROS helps integrate sensors, control algorithms, and actuators in a unified environment and enables seamless communication between different components of a robot.

1.2 Key Features

- **Modular design with nodes:** Software is organized into independent processes called *nodes*, each performing a specific task.
- **Communication through topics and services:**
  - *Topics* enable asynchronous publish/subscribe messaging.
  - *Services* provide synchronous request/response communication.
- **Integration of sensor data, control algorithms, and actuators:** Facilitates the flow of data across the entire robot system.

1.3 ROS Architecture

Core Concepts:

Master: The ROS master coordinates communication between nodes by tracking active nodes, topics, and services.

Nodes: Individual processes performing computation tasks.

Topics: Named communication buses for asynchronous publish/subscribe messaging.

Services: Mechanism for synchronous communication using request/response calls.

Messages: Data structures defining the format of information exchanged through topics and services.

1.4 Use Cases

ROS is commonly used in:

- Autonomous robots such as drones and self-driving vehicles.
- Service robots for healthcare, household tasks, and delivery.
- Industrial automation involving robotic arms and sensor integration.

1.5 ROS Filesystem (Brief Overview)

ROS organizes software into *packages* which contain nodes, libraries, configuration files, and message/service definitions. This organization aids in code reuse and distribution.

2 Introduction to ROS Filesystem

When working with ROS (Robot Operating System), managing your robot software projects systematically is essential. ROS uses a specific filesystem structure to organize source code, build files, and configuration. This structure helps keep your project organized, manageable, and scalable.

Two main organizational concepts in ROS are:

- **ROS Workspace**
- **ROS Package**

Understanding these is fundamental to working effectively with ROS.

2.1 What is a ROS Workspace?

A **ROS Workspace** is like a dedicated folder (directory) on your computer where you develop and build ROS projects. It acts as a container holding one or more ROS packages — the basic units of ROS software.

The most common type of workspace is called a **Catkin workspace** (named after the Catkin build system ROS uses).

Typical Structure of a Catkin Workspace:

- `~/catkin_ws/` — The root folder of your workspace (you can name it anything, but `catkin_ws` is a convention).
- `src/` — The most important folder inside the workspace; it contains the source code for your ROS packages.
- `build/` — This folder is automatically created when you build your workspace; it stores temporary files needed for compiling your packages.
- `devel/` — Also created during build; it contains the compiled binaries, libraries, and setup files used to run your code in the workspace.

2.1.1 Why these folders?

- `src/` is where you write your code (source code).
- `build/` and `devel/` are automatically created by the build system (Catkin) to organize the compilation process and the built products.
- You generally do **not** manually change `build/` or `devel/` — these are managed by ROS tools.

2.2 What is a ROS Package?

A **ROS Package** is the smallest unit of a ROS project. It contains all the files necessary for a particular functionality or feature. Think of a package as a self-contained module or app that does something specific, like controlling a robot arm or reading data from a sensor.

A package may contain:

- Source code files (`.cpp`, `.py`, etc.)
- Configuration files (`.xml`, `.yaml`)
- Launch files to start nodes
- Message and service definitions
- Dependency information

Packages are stored inside the `src/` folder of your workspace.

2.3 Structure inside a ROS Package

When you create a package, certain files and folders help organize its contents:

- `CMakeLists.txt`
  This is a build configuration file. It tells the Catkin build system how to compile your package's source code, where to find dependencies, and what targets to create.
- `package.xml`
  This file contains metadata about your package, such as the package name, version, maintainer information, license, and dependencies on other packages or system libraries.
- `scripts/`
  This folder typically contains Python scripts that serve as ROS nodes (processes).
- `src/`
  Contains C++ source code files. This is where your compiled code lives.
- `launch/`
  Contains XML files that define how to start one or more nodes at once, configure parameters, and set up the runtime environment. These are called launch files.

2.4 How to Create a ROS Package — Step by Step

Let's say you want to create a new ROS package named `robot_basics`. Here's how you do it from zero knowledge:

Step 1: Open a Terminal All ROS commands are run inside a terminal window in Linux (ROS typically runs on Ubuntu).

Step 2: Navigate to the Workspace Source Folder Change directory to the `src` folder inside your workspace:

```
cd ~/catkin_ws/src
```

(If `catkin_ws` does not exist yet, you need to create a workspace first — this can be explained separately.)

Step 3: Run the Package Creation Command ROS provides a command to create a new package skeleton with standard files. The command is:

```
catkin_create_pkg robot_basics std_msgs rospy roscpp
```

What does this mean?

- `catkin_create_pkg` — the command to create a package.
- `robot_basics` — the name of your new package.
- `std_msgs rospy roscpp` — these are dependencies your package will rely on.
  - `std_msgs` provides standard message definitions.
  - `rospy` allows you to write ROS nodes in Python.
  - `roscpp` allows writing ROS nodes in C++.

You can add more dependencies if needed later.

Step 4: Check What Files Are Created After running the command, ROS creates a folder named `robot_basics` inside `src/`, with the following essential files:

- `package.xml` — describing your package and its dependencies.
- `CMakeLists.txt` — build instructions.
- Folders like `src/`, `scripts/`, and `launch/` may not be created automatically; you add them manually when you start writing code.

2.5 Next Steps After Creating a Package

- Add your source code inside `src/` (for C++) or `scripts/` (for Python).
- Create launch files in the `launch/` folder to start nodes easily.
- Build your workspace from the root directory (`~/catkin_ws`) using:

```
catkin_make
```

- Source your workspace setup script to add it to your environment:

```
source devel/setup.bash
```

This makes ROS aware of your new package and lets you run your nodes.

2.6 Summary (Key Points)

Concept	Description
ROS Workspace	A directory containing ROS packages (`src/` folder) and build files (`build/`, `devel/`).
`src/`	Folder where all ROS packages' source code lives.
`build/`	Intermediate files created during build (auto-generated).
`devel/`	Contains built executables and environment scripts (auto-generated).
ROS Package	A self-contained module with source code and config files.
`package.xml`	Metadata and dependencies file inside a package.
`CMakeLists.txt`	Build instructions file inside a package.
`scripts/`	Folder for Python nodes.
`src/`	Folder for C++ source files.
`launch/`	Folder for XML launch files to start nodes.

3 Core Concepts of ROS (Robot Operating System)

In this section, we will explore the fundamental components that make up the Robot Operating System (ROS). Understanding these core elements is essential for effectively developing robotic applications using ROS. Each concept is described in detail, with examples and explanations to clarify their role and functionality within the ROS ecosystem.

3.1 Node

Definition: A **Node** in ROS is a single executable program that uses ROS communication infrastructure to interact with other nodes. Each node is designed to perform a specific task, such as controlling a motor, reading sensor data, or processing information.

Detailed Explanation: Nodes are the fundamental building blocks of a ROS-based system. By decomposing robot functionality into multiple nodes, ROS enables a modular design where different components can operate independently and concurrently. This modularity improves code organization, maintainability, and scalability.

For example:

- One node might be responsible for controlling the wheels of a robot, handling motor speed and direction commands.
- Another node might read data from a laser scanner sensor and publish this data to other nodes.
- A third node could process the sensor data to perform obstacle detection and planning.

Because nodes are separate processes, they can run in parallel on the same computer or distributed across multiple machines.

Key Points:

- Each node is an independent executable.
- Nodes communicate via ROS communication mechanisms such as topics and services.
- They enable parallelism and separation of concerns.

3.2 Master

Definition: The **ROS Master** is a central coordination service that enables nodes to find each other and establish communication channels.

Detailed Explanation: The Master acts like a directory or registry for all active nodes. When a node starts, it registers with the Master, providing information such as its name, published topics, and subscribed topics. Other nodes query the Master to discover the network locations of publishers or services they want to communicate with.

The Master provides:

- **Name registration:** keeps track of all node names and topic names.
- **Name lookup:** helps nodes find the publishers or services they want to connect to.

Important Note: The ROS Master must be running before any other nodes can start, as nodes depend on it to establish communication.

3.3 Topic

Definition: A **Topic** is a named bus over which nodes exchange messages asynchronously in a unidirectional manner.

Detailed Explanation: Topics are the primary communication method in ROS for streaming data. Nodes communicate by publishing messages to a topic or subscribing to receive messages from a topic.

The communication is **asynchronous**:

- **Publishers** send data to a topic whenever they want.
- **Subscribers** listen to the topic and receive data as it is published.

This decouples the producers and consumers of data so they do not need to be aware of each other's existence directly—only the topic name is required.

Example: A laser scanner node might publish sensor readings on a topic named `/scan`. Another node that performs mapping might subscribe to `/scan` to receive this data for processing.

3.4 Message

Definition: A **Message** in ROS is a strictly defined data structure used for communication over topics or services.

Detailed Explanation: Messages define the format and data types of the information exchanged. They are described in `.msg` files which specify fields like integers, floats, arrays, or nested messages.

ROS provides many standard messages, for example:

- `std_msgs/String` — a message containing a string.
- `sensor_msgs/LaserScan` — a message containing laser scan data.

Nodes use messages to structure the data they publish or receive, ensuring interoperability.

3.5 Service

Definition: A **Service** provides synchronous communication between nodes, based on a request/response model.

Detailed Explanation: Unlike topics, which are asynchronous and unidirectional, services require one node to send a request and wait for a response from another node.

Services are useful when:

- A node needs to query another for information (e.g., get the current robot state).
- An operation needs confirmation or a return value.

Services are defined in `.srv` files which specify the structure of the request and response messages.

3.6 Parameter Server

Definition: The **Parameter Server** is a shared, centralized storage system for configuration parameters accessible to all nodes at runtime.

Detailed Explanation: The Parameter Server allows nodes to store and retrieve configuration values such as PID controller gains, robot dimensions, or sensor frame names.

It acts like a global dictionary:

- Parameters can be set or changed at runtime.
- Nodes query parameters to adjust their behavior dynamically.

This helps to centralize configuration and avoid hardcoding values inside node code.

3.7 ROS Bags

Definition: **ROS Bag Files** are used to record and replay messages that flow through topics.

Detailed Explanation: Bag files are essential tools for debugging, testing, and data analysis. They allow developers to capture data streams from running robots and play them back later to reproduce scenarios without needing physical hardware.

Common commands:

- `rosbag record` — to record selected topics into a bag file.
- `rosbag play` — to replay the recorded data.

This feature enables offline analysis and consistent testing of algorithms.

3.8 ROS Launch Files

Definition: A **Launch File** is an XML file used to start multiple ROS nodes and set parameters in one command.

Detailed Explanation: Launch files simplify managing complex ROS systems by grouping node startup commands and configurations into a single file. They can:

- Launch several nodes simultaneously.
- Set parameters on the Parameter Server before nodes start.
- Include or group other launch files.

Basic syntax example:

```
<launch>
    <node pkg="robot_driver" type="driver_node" name="driver" output="screen"/>
    <param name="max_speed" value="1.5"/>
</launch>
```

3.9 ROS Tools Overview

ROS provides many command-line and GUI tools for development, debugging, and monitoring:

- `rqt_graph` — visualize the graph of nodes and topics.
- `rosnode` — manage and inspect running nodes.
- `rostopic` — view and publish topic messages.
- `rosbag` — record and replay topic messages.
- `rosparam` — set and get parameters on the Parameter Server.

These tools assist developers in understanding system behavior and troubleshooting.

3.10 RViz: Visualization Tool

Definition: **RViz** is a 3D visualization tool for displaying sensor data and robot states.

Detailed Explanation: RViz allows developers to visualize point clouds, camera images, robot models, and coordinate frames in a graphical interface. It is invaluable for debugging sensor inputs and robot localization or mapping.

Features include:

- Support for a wide range of sensor data types.
- Interactive markers for robot state visualization.
- Customizable views and layouts.

3.11 Gazebo: Simulation Environment

Definition: **Gazebo** is a powerful 3D robot simulator integrated with ROS.

Detailed Explanation: Gazebo simulates physics-based environments allowing developers to test navigation, manipulation, and perception algorithms without physical robots. It supports:

- Realistic physics including gravity, friction, and collisions.
- Sensor simulation (e.g., cameras, lidars).
- Multiple robots and complex environments.

This accelerates development and enables testing in scenarios difficult to replicate in reality.

3.12 ROS1 vs ROS2

Comparison:

- **ROS1** (e.g., Noetic):
  - Mature and widely adopted.
  - Uses custom middleware and communication protocols.
  - Strong community and rich ecosystem.
- **ROS2** (e.g., Foxy, Humble):
  - Designed for improved security, real-time performance.
  - Uses DDS (Data Distribution Service) as middleware.
  - Better suited for production-grade applications, embedded systems, and multi-robot setups.

Transition Considerations:

- Syntax and structural differences exist between ROS1 and ROS2.
- Many libraries and tools now support both versions.
- Choosing between ROS1 and ROS2 depends on project requirements such as real-time constraints and deployment environment.

Understanding these core concepts deeply will allow you to harness the full power of ROS for developing sophisticated and robust robotic applications.