IA712: Mobile Robotics

Lecture 3: System Integration

Zhi Yan

ENSTA - Institut Polytechnique de Paris

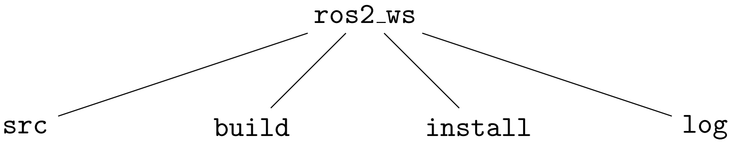
What is a Workspace?

A ROS 2 **workspace** is a directory containing the source code for your (custom) ROS 2 packages.
⟹ The place where you develop, build, and install your own robotics software!

A typical workspace (`ros2_ws`) has the following structure:

- `/src`: Source space. ▷ Place your ROS 2 packages here.
- `/build`: Build space. ▷ The build tool, `colcon`, uses this for intermediate files. You rarely interact with it directly.
- `/install`: Install space. ▷ After a successful build, the compiled programs, libraries, and launch files are placed here, ready to be used.
- `/log`: Log space. ▷ Contains logs from the build process for debugging.

What is a Workspace?



The Build Tool

`colcon` (**col**lective **con**struction): The standard command-line tool to build ROS 2 packages.

Core command:

To build all packages in your workspace, you navigate to the root of the workspace and run:

```
cd ~/ros2_ws/
colcon build
```

- `colcon` **automatically** discovers all the packages in the `src` directory.
- It resolves the **dependencies** between packages.
- It builds them in the correct **order**.
- It places the final **executables** and other files into the `install` directory.

Important: You must run `colcon build` from the **root** of the workspace, not from the `src` directory.

What is a Package?

A **package** is the fundamental unit of software organization in ROS 2.

It's a directory containing everything related to a specific piece of functionality.

A package can contain:

- ROS 2 nodes (C++ or Python source code)
- Launch files
- Configuration files
- Custom message/service/action definitions
- A `package.xml` file (metadata)
- A build file (`CMakeLists.txt` or `setup.py`)

What is a Package?

Packages are created using the `ros2 pkg create` command:

```
# Example for a Python package
ros2 pkg create --build-type ament_python my_package_name
```

- ament: A collection of build **rules and tools** for building, testing, and installing ROS 2 packages.
- colcon: Use the rules defined by ament to execute the build process.

The Manifest

Every ROS 2 package **must** contain a `package.xml` file, which defines the package's properties.

$\Longrightarrow$ Package's "ID card"!

Key Information in `package.xml`:

- `<name>`: The unique name of the package.
- `<version>`: The version number of the package.
- `<description>`: A brief summary of what the package does.
- `<maintainer>`: The person responsible for the package.
- `<license>`: The software license (e.g., Apache 2.0, MIT).
- `<build_depend>`: Dependencies needed to *build* the package.
- `<exec_depend>`: Dependencies needed to *run* the package.

Important: Correctly defining dependencies is crucial for `colcon` to build the workspace successfully.

Why Version Control?

How do you manage code, especially in a team?

Without Version Control

- `nav_node.py`
- `nav_node_working.py`
- `nav_node_final.py`
- `nav_node_final_v2.py`
- Zipping files and sending them via email
- Who changed what, and when?
- How to revert a change that broke everything?

With Version Control

- A complete history of every change.
- The ability to revert to any previous state.
- **Branching**: Work on new features without breaking the main code.
- **Collaboration**: Merge changes from multiple people reliably.
- It's the professional standard for all software development!

The Basic Git Workflow

Git is a distributed version control system.

Your packages in `ros2_ws/src` should be Git repositories.

Typical solo workflow:

1. **git add <files>**: Stage your changes for commit.
2. **git commit -m "Message"**: Save a snapshot of your changes to the local history.
   ⟹ A good message explains *why* you made the change.
3. **git push**: Upload your commits to a remote server like GitHub.
4. **git pull**: Download changes from the remote server.

The Basic Git Workflow

The most important file: `.gitignore`

- Your repository should only track **source code**.
- The `build`, `install`, and `log` directories are generated and should **not** be tracked.

```
# Add this to a .gitignore file in your workspace root
build/
install/
log/
```

Collaboration with GitHub

GitHub hosts your remote Git repositories and provides powerful collaboration tools.

The Pull Request (PR) workflow:

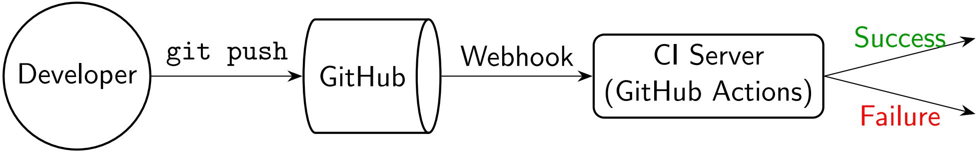
1. **Create a Branch**: Make a new branch for your feature (e.g., `feature/add-lidar-filter`).
2. **Push the Branch**: Push your commits to this new branch on GitHub.
3. **Open a Pull Request**: From GitHub, request to merge your new branch into the main branch (`main` or `master`).
4. **Review and Discuss**: Team members can review your code, suggest changes, and discuss the implementation directly on the PR.
5. **Merge**: Once approved, the changes are merged into the main branch.

We will use this workflow for the final course project.

ENSTA
IP PARIS

What is Continuous Integration?

Continuous Integration (CI) is the practice of automating the build and testing of code every time a team member commits changes to version control.



Core Idea: Catch integration bugs early and automatically.

$\Longrightarrow$ Your `main` branch should **always** build successfully.

A CI Workflow for ROS 2 with GitHub Actions

GitHub Actions is a CI/CD platform integrated into GitHub.

Define workflows in YAML files inside a `.github/workflows` directory in your repository.

A typical ROS 2 CI workflow when a developer opens a Pull Request:

1. A virtual machine running Ubuntu is automatically started.
2. ROS 2 Humble is installed.
3. Your repository's source code is checked out.
4. Dependencies from your `package.xml` files are installed using `rosdep`.
5. The entire workspace is built with `colcon build`.
6. Automated tests are run with `colcon test`.
7. A ✓ (success) or x (failure) is reported on the Pull Request.

This prevents merging code that breaks the system.

The Development Workflow

The standard workflow for developing with your own packages is:

1. Create a workspace directory (e.g., `ros2_ws/src`).
2. Create or clone your package(s) inside the `src` directory.
3. Navigate to the workspace root (`ros2_ws`).
4. Run `colcon build` to compile your packages.
5. **Source the local setup file**: `source install/setup.bash`.
6. Run your nodes or launch files.

Sourcing is very essential: Sourcing your local `install/setup.bash` file tells the ROS 2 environment where to find your newly built executables and packages.

Overlays and Underlays

Sourcing files creates a layered environment:

- **Underlay:** The base ROS 2 installation you sourced first (e.g., `/opt/ros/humble/setup.bash`).
  $\Longrightarrow$ It provides all the standard packages.
- **Overlay:** Your local workspace (`/ros2_ws/install/setup.bash`).
  $\Longrightarrow$ When you source this *after* the underlay, it adds your custom packages on top.

Overlays and Underlays

Overlay

Your Workspace (`ros2_ws`)

Underlay

Base ROS 2 Install (`/opt/ros/humble`)

The system will prioritize executables from the Overlay. $\Longrightarrow$ This allows you to use your own modified version of a package over the default one.

The Problem with `ros2 run`

For a real robot, you might need to start 10, 20, or even more nodes at once:

- Camera driver
- LiDAR driver
- Motor controller
- Localization node
- Path planner node
- ...and many more.

The Problem with `ros2 run`

For a real robot, you might need to start 10, 20, or even more nodes at once:

- Camera driver
- LiDAR driver
- Motor controller
- Localization node
- Path planner node
- ...and many more.

Challenge:

Starting each one manually in a separate terminal with `ros2 run` is inefficient, error-prone, and doesn't scale.

The Problem with `ros2 run`

For a real robot, you might need to start 10, 20, or even more nodes at once:

- Camera driver
- LiDAR driver
- Motor controller
- Localization node
- Path planner node
- ...and many more.

Challenge:

Starting each one manually in a separate terminal with `ros2 run` is inefficient, error-prone, and doesn't scale.

Solution:

ROS 2 Launch System. $\Longrightarrow$ Launch files allow you to start and configure an entire system of multiple nodes with a single command.

ENSTA
IP PARIS

Example: Python Launch File

Launch files are typically written in Python.

⟹ Allow for programmatic and flexible system startups.

File: `my_package/launch/turtlesim.launch.py`

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
return LaunchDescription([
  Node(
  package='turtlesim',
  executable='turtlesim_node',
  name='sim'
  ),
  Node(
  package='turtlesim',
  executable='turtle_teleop_key',
  name='teleop'
  )
])
```

This file describes a system with two nodes. To run it, you use the command:

```
ros2 launch my_package turtlesim.launch.py
```

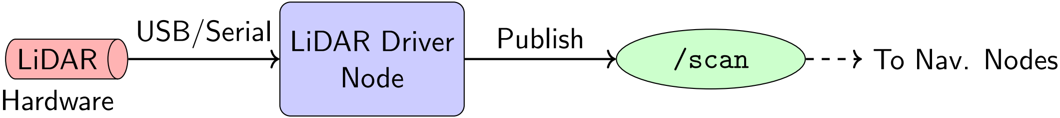
ENSTA IP PARIS

The Role of a Hardware Driver

How does ROS 2 talk to a real sensor or actuator?

The Role of a Hardware Driver

How does ROS 2 talk to a real sensor or actuator? $\Longrightarrow$ A driver is just a ROS 2 node.

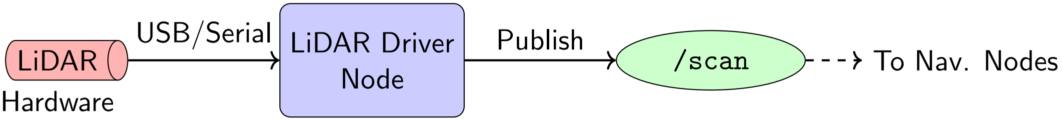


- The driver node reads raw data from the hardware (e.g., via a USB serial port).
- It converts this raw data into a standard ROS 2 message type (e.g., `sensor_msgs/msg/LaserScan`).
- It publishes this standardized message onto a topic for the rest of the system to use.

Benefit?

The Role of a Hardware Driver

How does ROS 2 talk to a real sensor or actuator? $\implies$ A driver is just a ROS 2 node.



- The driver node reads raw data from the hardware (e.g., via a USB serial port).
- It converts this raw data into a standard ROS 2 message type (e.g., `sensor_msgs/msg/LaserScan`).
- It publishes this standardized message onto a topic for the rest of the system to use.

Benefit? $\implies$ *Your navigation code doesn't need to know which brand of LiDAR you are using.*

Questions?

Next: Practical Work 3 - ROS 2 Intermediate Level