Engineering a ROS program

Real-Time Embedded System - The F1tenth autonomous racing

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

High Performance Real Time Lab

Code&build: ROS Workspaces

A workspace is a directory containing ROS 2 packages. Before using ROS 2, it's necessary to source your ROS 2 installation workspace in the terminal you plan to work in. This makes ROS 2's packages available for you to use in that terminal.

You also have the option of sourcing an "overlay" – a secondary workspace where you can add new packages without interfering with the existing ROS 2 workspace that you're extending, or "underlay".

Your underlay must contain the dependencies of all the packages in your overlay. Packages in your overlay will override packages in the underlay. It's also possible to have several layers of underlays and overlays, with each successive overlay using the packages of its parent underlays.

Code&build: ROS Workspaces

- Defines context for the current workspace
- Creating a new workspace
  - `$ mkdir -p <your_workspace>/src`
  - Then you can put your desired ROS2 packages inside src
- Resolving dependencies
  - `$ cd <your_workspace>`
  - `$ rosdep install -i --from-path src --rosdistro foxy -y`

Code&build: Overlays and Underlays

We refer to the main ROS 2 environment as the **underlay**. This has all the necessary setup to run ROS 2. Your workspaces are referred to as **overlays**. By sourcing your overlays you get access to your packages on top of the base ROS 2 environment.

After building, in a new terminal, source the underlay by:

- `$ source /opt/ros/foxy/setup.bash`

And in the root of your desired workspace:

- `$ cd <your_workspace>`
- `$ source install/local_setup.bash`
- Note that there's also a `install/setup.bash` in your workspace. You can also source this instead. This is equivalent to sourcing both your workspace overlay and the underlay your workspace was created/built in.

You might be tempted to leave these sourcing commands in your bashrc. I highly recommend **against** it. I've seen too many times that a teammate working on a package wondering why their code isn't working. And it ended up being `bashrc` sourcing the wrong overlay.

Colcon: the ROS build system

Build the workspace with `colcon`:

- From the root of your workspace: `$ colcon build`
- Useful arguments when building:
  - `--packages-up-to`: builds the package you want, plus all its dependencies, but not the whole workspace. This will save some time if you don't need all the packages in the workspace.
  - `--symlink-install`: saves you from having to rebuild every time you tweak Python scripts
  - `--event-handlers console_direct+`: shows console output while building (can otherwise be found in the `log` directory)

Once build finishes, you'll see `build` `install` `log` `src` directories in your workspace. The `install` directory is where your workspace's setup files are.

Release: ROS 2 Packages

- A package can be considered a container for your ROS 2 code. If you want to be able to install your code or share it with others, then you'll need it organized in a package. With packages, you can release your ROS 2 work and allow others to build and use it easily.

- Package creation in ROS 2 uses ament as its build system and colcon as its build tool. You can create a package using either CMake or Python, which are officially supported, though other build types do exist.

ROS 2 Packages

- A single workspace can contain as many packages as you want, each in their own folder. You can also have packages of different build types in one workspace (CMake, Python, etc.). You cannot have nested packages.

- Best practice is to have a `src/` folder within your workspace, and to create your packages in there. This keeps the top level of the workspace “clean”.

Cmake and Python Packages

CMake packages:

- `package.xml`: file containing meta info about the package
- `CMakeLists.txt`: file describing how to build the code within the package

Python packages:

- `package.xml`: file containing meta info about the package
- `setup.py`: contains instructions for how to install the package
- `setup.cfg`: required when a package has executables, so ros2 run can find them
- `/<package_name>`: a directory with the same name as your package, used by ROS 2 tools to find your package, contains `__init__.py`

ROS 2 Packages

The simplest package may have a file structure that looks like:

- CMake:

```
cmake_package/
    CMakeLists.txt
    package.xml
```

- Python:

```
py_package/
    setup.py
    package.xml
    resource/my_package
```

ROS provides you with tools to create empty packages

```
$ cd <your_workspace>/src
$ ros2 pkg create --build-type ament_python <package_name>
$ ros2 pkg create --build-type ament_cmake <package_name>
```

ROS 2 Packages

A workspace with multiple packages might look like this:

```
workspace_folder/
    src/
      package_1/
          CMakeLists.txt
          package.xml

      package_2/
          setup.py
          package.xml
          resource/package_2
      ...
      package_n/
          CMakeLists.txt
          package.xml
```

Package contents

- CMake packages
  - `CMakeLists.txt, include, package.xml, src`
  - Node source files (.cpp) are in src, and headers files (.h) in include

- Python packages
  - `my_package, package.xml, resource, setup.cfg, setup.py, test`
  - Node source files (.py) are inside the `my_package` directory

Customizing package.xml

- Fill in name and email on `maintainer` line, edit description to `summarize` the package, update the `license` line.
- Fill in your dependencies under the `_depend` tags. For documentation on what types of depend tags, see: https://www.ros.org/reps/rep-0149.html#build-depend-multiple

Customizing setup.py

- For Python packages, you'll also need to fill in `setup.py`
- Fill in the same description, maintainer, and license fields as in `package.xml`. You'll need to match these exactly. You'll also need to match the `package_name` and `version`.

Adding Dependencies and Entrypoint

- Filling in the dependencies in package.xml for our example node:

```
<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

  - This declares the package needs `rclpy` and `std_msgs` when code is executed.

- [Python] Add an entrypoint in setup.py:

```
entry_points={
        'console_scripts': [
                'talker = py_pubsub.publisher_member_function:main',
        ],
},
```

Rosdep: installing dependencies

ROS 2 uses `rosdep` to install package dependencies.

- https://index.ros.org/ maintains repos and packages you can use as dependencies and has recipes for installation for `rosdep`.

To install dependencies for a workspace, in the workspace directory:

- `$ rosdep install -i --from-path src --rosdistro foxy -y`
- This will install dependencies declared in `package.xml` from all packages in the `src` directory for ROS 2 foxy

Ros run

Before, always check the deps!!!

› From the root of your workspace, run

```
$ rosdep install -i --from-path src --rosdistro foxy -y
```

...and source the installation files (1 node < -- > 1 terminal!)

```
$ colcon build --packages-select cpp_pubsub
```

You can run single nodes with

```
$ ros2 run <package name> <node name>
```

Ex: `$ ros2 run cpp_pubsub talker`

Launch files

- To start up and configure a number of executables containing ROS 2 nodes simultaneously.

- Running a single launch file with the `ros2 launch` command will start up your entire system - all nodes and their configurations - at once.

Code structure

- In your package, create a new `launch` directory for launch files
- Create your `<my launch>.py` file
- Written in Python now in ROS 2 instead of xml in ROS 1. This allows for access to Python libraries.

Example Launch file

```python
from launch import LaunchDescription
from launch_ros.actions import Node
from launch.substitutions import Command
from ament_index_python.packages import get_package_share_directory
import os
import yaml

def generate_launch_description():
    ld = LaunchDescription()
    config = os.path.join(
        get_package_share_directory('f1tenth_gym_ros'),
        'config',
        'sim.yaml'
        )
    config_dict = yaml.safe_load(open(config, 'r'))

    bridge_node = Node(
        package='f1tenth_gym_ros',
        executable='gym_bridge',
        name='bridge',
        parameters=[config]
    )
    rviz_node = Node(
        package='rviz2',
        executable='rviz2',
        name='rviz',
        arguments=['-d', os.path.join(get_package_share_directory('f1tenth_gym_ros'), 'launch', 'gym_bridge.rviz')]
    )
```

Example Launch file

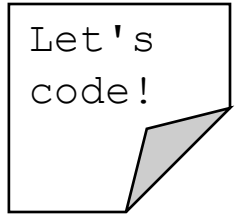
```
    map_server_node = Node(
        package='nav2_map_server',
        executable='map_server',
        parameters=[{'yaml_filename': config_dict['bridge']['ros__parameters']['map_path'] + '.yaml'},
                    {'topic': 'map'},
                    {'frame_id': 'map'},
                    {'output': 'screen'},
                    {'use_sim_time': True}]
    )
    nav_lifecycle_node = Node(
        package='nav2_lifecycle_manager',
        executable='lifecycle_manager',
        name='lifecycle_manager_localization',
        output='screen',
        parameters=[{'use_sim_time': True},
                    {'autostart': True},
                    {'node_names': ['map_server']}]
    )
    ego_robot_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='ego_robot_state_publisher',
        parameters=[{'robot_description': Command(['xacro ', os.path.join(get_package_share_directory('f1tenth_gym_ros'), 'launch',
'ego_racecar.xacro')])}],
        remappings=[('/robot_description', 'ego_robot_description')]
    )
```

Example Launch file

```
# finalize
    ld.add_action(rviz_node)
    ld.add_action(bridge_node)
    ld.add_action(nav_lifecycle_node)
    ld.add_action(map_server_node)
    ld.add_action(ego_robot_publisher)

    return ld
```

Exercise



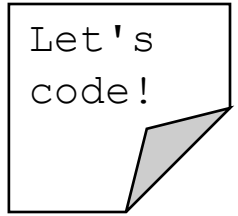
Build and run the pub/sub examples that come with in ROS2

- I already set up a workspace in the `Code/ROS2` folder of the course, including those nodes
- Both C++ and Python version (guess which one I prefer? ☺)
- `py_pubsub`, `cpp_pubsub`

More in details, you should

- Set up the ROS2 environment/workspace
- Resolve ROS2 dependencies (only once)
- Build the full workspace (or dedicated packages with `colcon`)
- Run a `listener` and a `talker` node

Exercise



Now, profile the turnaround time for a single message

- Implement a “ping” functionality in a Server node
- A Publisher node that tracks the time taken to send and receive a message
- There are several timestamp functionalities…
- You can modify `py_pubsub`, `cpp_pubsub`

Variants

- Increase the payload size, and plot how the time (Y axis) varies increasing it (X axis)
- Change the underlying comm layers: use different machines, and different networks (wired, wireless)
- Run additional nodes, and play with QoS

Timestamp in ROS2 – C++

- Rclpp offers a `Clock` class

```
#include "rclcpp/clock.hpp"


rclcpp::Clock *clk = new rclcpp::Clock();
rclcpp::Time time = clk->now();
rcl_time_point_value_t ns = time.nanoseconds(); // Uint64_t in ref impl.

RCLCPP_INFO(this->get_logger(), "Time is: %lld", ns); // Print timestamp
```

Useful links

- https://docs.ros2.org/bouncy/api/rclcpp/classrclcpp_1_1_clock.html
- https://docs.ros2.org/bouncy/api/rclcpp/classrclcpp_1_1_time.html
- https://docs.ros2.org/latest/api/rcutils/time_8h.html

Timestamp in ROS2 – Python

- Rclpp offers a `Clock` class

```
from rclpy.clock import Clock

now = Clock().now()
self.get_logger().info('Current time %i' % now.nanoseconds) # Print it
```

(Recap) Timestamp in UNIX – C++

› Use the `chrono` lib (in turn, will require `ctime`)

```
#include <chrono>
#include <ctime>

auto time2 = std::chrono::system_clock::now().time_since_epoch();
RCLCPP_INFO(this->get_logger(), "Time2 is: %lld", time2.count()); // Print it
```

Useful link

› https://en.cppreference.com/w/cpp/chrono/time_point/time_since_epoch

(Recap) Timestamp in UNIX – Python

- Use `datetime` module/object

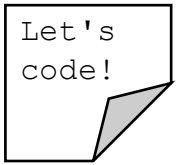
```
from datetime import datetime

now = datetime.now()
self.get_logger().info('Current time %i' % now.time().microsecond) # Print it
```

Useful links

- https://stackoverflow.com/questions/415511/how-do-i-get-the-current-time

How to run the examples



› Find them in `Code/ros2` folder from the course website

For building ROS2 workspaces

```
$ colcon build
```

Run

› In a new terminal!!!

```
$ ros2 run <PACKAGE> <NODE>
```

› Something wrong? Check nodes and topics with

```
$ ros2 node list
$ ros2 topic list
```

References

Course website

- http://personale.unimore.it/rubrica/contenutiad/markober/2023/71846/N0/N0/10005
- https://github.com/HiPeRT/F1tenth-RTES
  - Online resources/preview

My contacts

- paolo.burgio@unimore.it
- http://hipert.mat.unimore.it/people/paolob/

Resources

- https://docs.ros.org/en/foxy/Tutorials.htmlhttps://github.com/pburgio/CPP_For_MMR (for CMake)
- A "small blog»
  - http://www.google.com