Complete Guide to ROS 2 Development and CR5 Robot Control

This guide serves as a manual for navigating and modifying the ROS environment, and also as an instruction set for executing and testing the various Dobot scripts.

1. Creating a New ROS 2 Package

Creating a package is the essential first step for organizing your code in ROS 2. Packages can be written in Python or C++.

1.1. Creating a Python Package

To create a Python package, use the following command in your ROS 2 workspace folder (e.g., `~/ros2_ws/src`):

```
ros2 pkg create --build-type ament_python <package_name>
# Example: ros2 pkg create --build-type ament_python cr5_moveit_script
```

The `-build-type ament_python` flag indicates that the package will use Python code. This command will generate a basic structure with essential files such as `setup.py` and `package.xml`.

Note on license warning: During creation, you might receive a warning about license specification. It is good practice to add one (e.g., Apache 2.0, MIT) in the `package.xml` file for compliance and transparency.

1.2. Creating a C++ Package

If you intend to write your node in C++, the command is similar, but with a different `-build-type`:

```
ros2 pkg create --build-type ament_cmake <package_name>
# Example: ros2 pkg create --build-type ament_cmake cr5_moveit_cpp_demo
```

The `-build-type ament_cmake` flag indicates that the package will use CMake for C++ code compilation. This will also create a `CMakeLists.txt` file where you will configure the compilation of your C++ code.

Main Differences Between Python and C++ Packages

- **File Extension**: For C++ packages, source files will have the `.cpp` extension and headers `.hpp`. For Python, files are `.py`.
- `CMakeLists.txt` vs `setup.py`:
  - **C++ (`CMakeLists.txt`)**: This file is the core of the compilation process. Here you define which `.cpp` files should be compiled into executables (nodes), which libraries to include, and which dependencies to resolve. This is where you use `add_executable` to create your nodes.
  - **Python (`setup.py`)**: This file configures how the Python package is installed and which scripts become executable (nodes) via the `entry_points` section.
- **Compilation**: C++ packages require explicit compilation with `colcon build`. Python packages are generally "installed" by `colcon` without a traditional compilation phase, but code changes are immediately visible after environment setup.

2. Node Code Development

2.1. Creating the Node Script/Source File

For both package types, create the file where you will write your node.

- **For Python**: ROS 2 expects Python node files to be inside a subfolder with the same name as the package (e.g., `cr5_moveit_script/cr5_moveit_script/`).

```
touch <package_name>/<package_name>/<node_name>.py
# Example: touch cr5_moveit_script/cr5_moveit_script/move_cr5.py
```

- **For C++**: C++ source files are typically located in an `src/` folder within the package.

```
touch <package_name>/src/<node_name>.cpp
# Example: touch cr5_moveit_cpp_demo/src/move_cr5_node.cpp
```

After creating the file, write your code inside it.

2.2. Making the Python File Executable (Python Packages Only)

If you are developing a Python node, you must make it executable:

```
chmod +x <package_name>/<package_name>/<node_name>.py
# Example: chmod +x cr5_moveit_script/cr5_moveit_script/move_cr5.py
```

Furthermore, at the top of every Python file that acts as a node, add the "shebang" to tell the system which interpreter to use:

```
#!/usr/bin/env python3
```

2.3. Configuring the `setup.py` File (Python Packages Only)

To allow `ros2 run` to find your Python node, you must modify the `setup.py` file in the main package folder. Add your node to the `entry_points` section under `console_scripts`. An example might be:

```
from setuptools import setup

package_name = 'cr5_moveit_script' # Replace with your package name

setup(
name=package_name,
version='0.0.0',
packages=[package_name],
data_files=[
('share/ament_index/resource_index/packages',
['resource/' + package_name]),
('share/' + package_name, ['package.xml']),
],
install_requires=['setuptools'],
zip_safe=True,
maintainer='Your Name',
maintainer_email='your.email@example.com',
description='TODO: Package description',
license='TODO: License declaration',
tests_require=['pytest'],
entry_points={
  'console_scripts': [
  'move_cr5_node = cr5_moveit_script.move_cr5:main', # Format: <
      executable_name> = <package_name>.<python_file_name_without_ext>:
      main_function
  ],
},
)
```

Make sure that `main` is the name of the main function of your Python node.

2.4. Configuring the `CMakeLists.txt` File (C++ Packages Only)

For C++ nodes, you will need to add your executable to the package's `CMakeLists.txt` file. Look for the section where executables are added and add your node. Simplified example:

```
# ... other CMake configurations ...

# Add an executable for your C++ node
add_executable(<node_name> src/<node_name>.cpp)

# Link the necessary dependencies for your node
ament_target_dependencies(<node_name> rclcpp moveit_ros_planning_interface
    # Add other dependencies here)

# Install the executable in the package's installation directory
install(TARGETS <node_name> DESTINATION lib/${PROJECT_NAME})

# ... other CMake configurations ...
```

Make sure to link all the libraries needed by your node. If you do not want a specific `.cpp` file (e.g., `follow_target.cpp`) to be compiled into an executable, you can comment out or remove the corresponding `add_executable` line in `CMakeLists.txt`.

3. Dependency Management

3.1. Declaring Dependencies in `package.xml`

For every library or ROS package that your node uses (via `import` in Python or `#include` in C++), you must declare the dependencies in your package's `package.xml` file. Use tags such as `<build_depend>` (for build dependencies), `<exec_depend>` (for runtime dependencies), and `<depend>` (for both).

Note on `message_generation`: If you encounter errors related to `message_generation` (often a deprecation warning), make sure to use `rosidl_default_generators` for message generation, as in the example. This is particularly relevant for packages that define custom messages or services.

```
<build_depend>rosidl_default_generators</build_depend>
```

This line should be added instead of or in addition to `<build_depend>message_generation</build_depend>` in your `package.xml` (e.g., in `src/DOBOT_6Axis_ROS2_V4/dobot_bringup_v4/package.xml`, if relevant).

3.2. Installing System Dependencies

After updating `package.xml` with ROS dependencies, install the necessary system dependencies (non-ROS) using `rosdep`. Ensure you are in the root of your ROS 2 workspace (e.g., `~/ros2_ws`) or in your package directory:

```
sudo apt update
rosdep install --from-paths . --ignore-src -r -y
```

Note on "mongo" error: If `rosdep check -from-paths src -ignore-src` reports "mongo" as unsatisfied but "mongo" does not exist as an installable package (or is not required for your project), it is often a false positive due to non-essential transient dependencies. You can generally ignore it if you are not directly using MongoDB databases.

4. Compilation and Environment Setup

4.1. Package Compilation

After creating or modifying your nodes and updating dependencies, compile the package. This step is crucial for C++ packages but also necessary for Python packages to install their scripts. Execute the command from the root of your workspace (e.g., `~/ros2_ws`):

```
colcon build --packages-select <package_name>
# Example for Python: colcon build --packages-select cr5_moveit_script
# Example for C++: colcon build --packages-select cr5_moveit_cpp_demo
```

4.2. ROS 2 Environment Setup

To make your nodes executable and libraries available in the current terminal, you must source the ROS 2 environment of your workspace:

```
source install/setup.bash
```

This command must be executed every time you open a new terminal to work with ROS 2 in your workspace.

Automating Environment Setup

To avoid typing `source install/setup.bash` every time you open a terminal, you can add it to your `~/.bashrc` file. This file is executed automatically at the start of every terminal session.

```
# 1. Open the .bashrc file with a text editor
nano ~/.bashrc

# 2. Add the following line to the end of the file
# Make sure this line is PLACED AFTER any 'source /opt/ros/<distro>/setup.bash'
# and that the path '~/ros2_ws' is correct for your workspace.
source ~/ros2_ws/install/setup.bash

# 3. Save the file and exit the editor (Ctrl+O, Enter, Ctrl+X for nano).
# 4. Apply the changes in the current terminal without restarting it:
source ~/.bashrc
```

Now, every new terminal you open will have the ROS 2 environment of your workspace already configured.

5. Executing ROS 2 Nodes

5.1. Executing a Single Node

To execute a single node from your package:

```
ros2 run <package_name> <node_executable_name>
# Python Example: ros2 run cr5_moveit_script move_cr5_node
# C++ Example: ros2 run cr5_moveit_cpp_demo move_cr5_node
```

5.2. Execution via Launch File

Python launch files allow you to start multiple nodes and complex configurations (e.g., setting parameters, topic remapping) simultaneously.

```
ros2 launch <package_name> <launch_file_name>.launch.py
# Example: ros2 launch cr5_moveit_cpp_demo scripts_launch.launch.py
```

Your launch file, as set up previously, launches the `reworked_map_node` and `fake_zed_node` nodes.

6. Simulation (RViz)

6.1. Starting MoveIt! with RViz

To visualize the robot and motion planning in simulation (RViz) with MoveIt!, launch the appropriate file:

```
ros2 launch dobot_moveit dobot_moveit.launch.py
```

6.2. Starting the Fictitious Map Publisher (for testing) and the Map Reprocessor

If you do not have a real ZED camera or map sensor, you can launch this file which launches both the map reprocessor and the node that simulates map data publishing. This is useful for testing nodes that depend on this data without having the real hardware:

```
ros2 launch cr5_moveit_cpp_demo scripts_launch.launch.py
```

7. Real Robot Control

Configuration Automation

It is possible to automate all the subsequent configuration steps by executing the `dobot_startup.sh` script provided in the repository. To terminate the execution cleanly, execute the `dobot_stop.sh` script instead.

7.1. PC Network Configuration

Before connecting to the real Dobot CR5 robot, configure your PC's network interface to be on the same subnet as the robot (usually 192.168.5.x for Dobots). Replace `enp2s0` with the name of your network interface (you can find it with `ip a`):

```
sudo ip addr add 192.168.5.100/24 dev enp2s0
```

- **Connection Verification**: After setting the IP, check connectivity with the robot:

```
ping 192.168.5.1
```

- **Disable NetworkManager for the interface (Optional, if necessary)**: If you encounter problems with static IP configurations or conflicts with NetworkManager, you may need to prevent it from managing the specific interface:

```
sudo nmcli dev set enp2s0 managed no
sudo systemctl restart NetworkManager
```

7.2. Starting Real Robot Drivers

Launch the specific Dobot CR5 robot drivers via the launch file:

```
ros2 launch dobot_bringup_v3 dobot_bringup_ros2.launch.py
```

7.3. Starting All Nodes Required for Robot Operation

This command starts a complete set of nodes required for robot control and movement support nodes (including drivers, MoveIt! for planning, reworked_map, zed_listener, etc.):

```
ros2 launch cr5_moveit full_bringup.launch.py
```

7.4. Executing the Movement Script (Real and Simulated Robot)

To control the real robot with your movement script:

```
ros2 run cr5_moveit_cpp_demo move_cr5_node
```

8. Maintenance and Debugging

8.1. Checking and Updating Dependencies and CMakeLists.txt

Every time you modify the code, add new features, or use new libraries, it is essential to check and update the dependencies in the `package.xml` file and, for C++, in the `CMakeLists.txt`.

8.2. Frequent Recompilation and Setup

- **Recompile**: After any change to the source code, run the compilation again:

```
colcon build --packages-select <package_name>
# Example: colcon build --packages-select cr5_moveit_cpp_demo
```

- **Setup (if not automated)**: If you have not automated the `source install/setup.bash` in your `.bashrc`, you must re-execute it in every new terminal after compiling to make the changes available:

```
source install/setup.bash
```

8.3. Firewall Configuration

If you use a firewall (e.g., UFW), you may need to open specific ports that ROS 2 or other services use (e.g., port 9090 for `rosbridge_server`):

```
sudo ufw allow 9090
```

8.4. Useful Commands for the ROS 2 Environment

- `source /opt/ros/humble/setup.bash`: This command loads the base ROS 2 environment for the "Humble" distribution (or the one installed on your system). It is useful if you are not in a specific ROS 2 workspace or if your local environment is not configured.