Interplanetar Workshop 2024: Software

Lecture-4: ROS Packages

In this lecture, we will take a look at ROS packages. We will be discussing on what ROS packages are, the structure of ROS packages, the build tools utilized to create a package, and how to use them. We will go through examples of creating a ROS package from scratch, as well as installing existing ROS packages.

Table of Content

Introduction

Software in ROS is organized in packages. A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module. The goal of these packages it to provide this useful functionality in an easy-to-consume manner so that software can be easily reused. In general, ROS packages follow a "Goldilocks" principle: enough functionality to be useful, but not too much that the package is heavyweight and difficult to use from other software.

For more information: https://wiki.ros.org/Packages

ROS Package Structure

A ROS package folder has directories in it based on type and functionality:

- `CMakeLists.txt`: Contains build information for CMake.
- `package.xml`: Contains metadata for the package (e.g., package name, maintainer's information, dependencies).
- `src`: Contains the C++ files for nodes designed in C++.
- `include`: Contains header files required for the C++ nodes.
- `launch`: Contains the launch files for that package.
- `msg`: Contains custom ROS message types for that package (if created).

- `srv`: Contains files for ROS services in that package.
- `action`: Contains ROS action files.
- `config`: Contains necessary config(.yaml) files for the package (if needed).
- `urdf`: Contains robot description files that contain information for constructing a 3D visualization of a robot using STL files (if created).
- `meshes`: Contains the necessary STL files needed to construct a 3D visualization of a robot (if created).
- `scripts`: Contains executable python files for the nodes designed in Python.
- `README.md`: Contains description about the package.

A typical ROS package may have a structure like this:

```
my_package/
├── CMakeLists.txt
├── package.xml
├── src/
│   └── my_package_node.cpp
├── include/
│   └── my_package/
│       └── my_package_node.h
├── launch/
│   └── my_package.launch
├── msg/
│   └── MyMessage.msg
├── srv/
│   └── MyService.srv
├── action/
│   └── MyAction.action
├── config/
│   └── my_package.yaml
├── urdf/
│   └── my_robot.urdf
├── meshes/
│   └── my_robot_part.stl
├── scripts/
│   └── my_script.py
└── README.md
```

Introduction to Build Tool: Catkin

A *build system* is responsible for generating 'targets' from raw source code that can be used by an end user. These targets may be in the form of libraries, executable programs, generated scripts, exported interfaces (e.g. C++ header files) or anything else that is not static code. In ROS terminology, source code is organized into 'packages' where each package typically consists of one or more targets when built.

CMake is a popular build system that is widely used for compiling C++ code. However, ROS supports multiple language support, mainly C++ and Python. To compile executable code from multiple files written in multiple languages and seamlessly use them together, ROS uses a custom build tool called **Catkin**.

Catkin combines CMake macros and Python scripts to provide some functionality on top of CMake's normal workflow. Catkin was designed for better distribution of packages, better cross-compiling support, and better portability.

Creating ROS Packages

In order to create and use a ROS package, we need to have a ROS workspace set up with catkin and build the packages in it. For creating a workspace, we need to create a folder and run `catkin_make`.

```
$ mkdir ros_ws
$ cd ros_ws
$ mkdir src
$ catkin_make
```

> **Note:** You have to run `catkin_make` at the **workspace** folder every time you add or create a new package. To streamline this process and better manage the packages, we can use *catkin_tools* commands instead of running `catkin_make`. To install `catkin_tools`, follow the instructions here. Then, you can run the following commands:
>
> ```
> $ mkdir ros_ws
> $ cd ros_ws
> $ mkdir src
> $ catkin build
> ```
>
> **Note:** You should use *either* `catkin_make` *or* `catkin build` for building packages, not *both*.

Creating From Scratch

To create a ROS package from scratch, run the `catkin_create_pkg` command on the terminal at the `ros_ws/src` directory. The syntax for the command is as follows:

```
catkin_create_pkg package_name dependency1 dependency2 dependency3
```

Here, the `package_name` is the name you want to give to your package, and `dependency1`,... are the packages that will be used in your package as dependencies.

Here's an example of creating a package:

```
# assuming the workspace is created in ~/ros_ws directory
$ cd ~/ros_ws/src
```

```
$ catkin_create_pkg ros_tutorial_pkg roscpp rospy std_msgs geometry_msgs
```

Now you should see your package being built, and the package directory should look something like this:

```
ros_ws
|--- build
|--- devel
|--- src
    |--- ros_tutorial_pkg
        |--- CMakeLists.txt
        |--- package.xml
        |--- src
```

If you decide to add/remove dependencies to your package, you will need to update your `package.xml` file. Add the dependencies as `<depend>` and `<exec_depend>` in the `package.xml` file. For Python nodes, you will need to add that dependency as `<exec_depend>`. For example:

```
<exec_depend> dependency4 </exec_depend>
```

For C++ nodes, you will need to add dependencies as `<build_depend>` and/or `<exec_depend>` based on what type of dependency that package is.

```
<build_depend> dependency4 </build_depend>

<exec_depend> dependency4 </exec_depend>
```

To learn more about different types of dependencies, check this out.

Installing Pre-existing Packages

There exists many useful ROS packages that can be installed into a workspace. We will be covering how you can install an external ROS package using GitHub and Apt.

From GitHub

All you have to do is clone the package repository into the `src` forder of your workspace, and run `catkin_make` as usual to build the package.

```
# installing package from GitHub
$ cd ~/ros_ws/src
$ git clone -b <branch> <address>
$ cd ..
$ catkin_make
```

> **Note:** Generally, the ROS packages are organized in Git branches according to the ROS versions. If you're installing from the official ROS Github repo (https://github.com/ros), make sure to check out the branch that contains the ROS version your device is running. If you're checking out other repositories, make sure to download the package folder and paste the folder in the `src` folder of your workspace and build the package.

From Apt

Open a terminal and run the apt command to download a package.

```
# installing package using apt-get
$ sudo apt-get install <package-name>
```

For the purpose of this lecture, we are going to install the **Turtlesim** package by running the command:

```
$ sudo apt-get install ros-$(rosversion -d)-turtlesim
```

Creating Launch Files

Launch files are very handy for running multiple ROS nodes simultaneously. These are XML files that contain information on which nodes to run. The structure of a launch file is as the following:

```
<launch>
  <!-- Set global parameters -->
  <param name="global_param1" type="string" value="value1" />
  <param name="global_param2" type="int" value="10" />

  <!-- Node 1 (Python) -->
  <node name="node1" pkg="package1" type="node1_script.py" output="screen">
  </node>

  <!-- Node 2 (C++) -->
  <node name="node2" pkg="package2" type="node2_script" output="screen">
  </node>

</launch>
```

Save the launch file as `<filename>.launch`. Launch files must be put in the `launch` folder in your package, and the Python nodes must be made executable before using.

```
$ cd ~/ros_ws/src/ros_tutorial_pkg/scripts
$ sudo chmod +x node1_script.py
```

After doing that, you have to source your terminal, and use `roslaunch` command to run a launch file.

```
$ cd ~/ros_ws
$ source devel/setup.bash
$ roslaunch ros_tutorial_pkg <filename>.launch
```

Using ROS Packages

In order to use any Python node in a ROS package, we first need to make the .py files executable. We can do this with command lines by using the `chmod +x` command.

```
# making python nodes executable
$ sudo chmod +x /dir/to/python_node.py
```

For C++ nodes, we do not need to do this step.

Once that's done, we need to source the `setup.bash` file of the workspace in order to use the nodes in the package. We need to repeat this process every time we switch to a new terminal.

```
$ cd ~/ros_ws
$ source devel/setup.bash
```

We also need to source the `setup.bash` file for ROS every time we switch to a new terminal.

```
$ source /opt/ros/noetic/setup.bash # if the ROS version is 'noetic'
```

> **Note:** Sourcing ROS every time for a new terminal is a cumbersome process. To avoid doing that, we can write this command in the `~/.bashrc` file, so that every time a new terminal is created, the sourcing is done automatically.
>
> Open the `~/.bashrc` file with a text editor and write down the command in the file.
>
> ```
> $ source /opt/ros/noetic/setup.bash # if the ROS version is 'noetic'
> ```
>
> Save the file, and now every time a terminal opens, the `setup.bash` file for ROS will already be sourced.

Once all the sourcing is done, we can use `rosrun` command to run a node from the sourced packages.

```
$ rosrun ros_tutorial_pkg python_node.py
```

For C++ nodes, we do not need to mention the extension of file name when running a node.

```
$ rosrun ros_tutorial_pkg cpp_node
```

We can also check if a certain package exists that can be used using the `rospack find` command.

```
$ rospack find <package-name>
```

If the package exists and is usable, you should see the directory of the package. If the package is not found, the package either does not exist, or it is not sourced.

Example: Using Turtlesim Package

In the previous topics, we have installed the `turtlesim` package and the `turtlesim_gui` package. The `turtlesim_gui` package contains a python node named `gui.py` that can be used to control the turtle in the `turtlesim_node`. We will run the `turtlesim_node` and `gui.py` nodes, that we can use to play around with the turtlesim.

First, we need to source the `setup.bash` files for ROS and ros_ws.

```
$ cd ~/ros_ws
$ source devel/setup.bash
$ source /opt/ros/noetic/setup.bash
```

Before using the `gui.py` node, we need to make the file executable. We will use the `chmod +x` command to make it executable.

```
$ sudo chmod +x src/ros_tutorial_pkg/scripts/gui.py
```

Then, we can run the `rosrun` commands to run the nodes we want to run.

```
$ rosrun turtlesim turtlesim_node
$ rosrun ros_tutorial_pkg gui.py
```

Now, we should see a turtlesim simulation window and a gui window. We can use the buttons in the gui to control the turtle.

What is happening under the hood, is that the gui is publishing data into the `/cmd_vel` topic for controlling the turtle. The turtlesim node takes that data and moves the turtle accordingly.

We can run the `rostopic list` command to see all the available topics.

```
$ rostopic list
```

It should give a list of all the topics currently in use. We can use the `rostopic echo` command to check the data being published in `/cmd_vel` by the gui.

```
$ rostopic echo /cmd_vel
```