Robot Operating System

Cheat sheet for ROS 1 & 2

Goal of this document

This *cheat sheet* aims to recall the main steps and sequences to follow when working on a ROS 1 or ROS 2 (or hybrid) environment. It is separated into the following sections:

1. How and when to source workspaces
2. How and when to compile workspaces and packages
3. What do to if the system seems ill-configured

The goal is to be able to focus on the actual use or development of the packages and nodes, and let the ROS file architecture work as it should. You can also detect errors that do not come from your code but from an ill-functionning environment.

1 Sourcing workspaces

1.1 Official workflow

This workflow is the one assumed by most online tutorials on ROS 1 and ROS 2. The core architecture of ROS is to have packages placed in various *workspaces*. Some workspaces can *overlay* others, meaning that:

- They can depend on packages from lower-overlayed (underlayed) workspaces
- If a package exists in multiple workspaces, the overlayed one is prefered

The content of a workspace are a few standard directories:

```
<workspace>/src # package sources (the only folder we work in)
<workspace>/build # compilation artifacts
<workspace>/devel # runtime files if the workspace is not installed (ROS 1)
<workspace>/install # runtime files if the workspace is installed (ROS 1 / 2)
```

When opening a terminal, workspaces should be `sourced` in overlay ordering. Typically, if one uses ROS Noetic and their own workspace in `~/ros`, a terminal should be initialized as follows to work on ROS 1:

```
source /opt/ros/noetic/setup.bash # lowest-level workspace (base ROS installation)
source ~/ros/devel/setup.bash # or ~/ros/install/setup.bash if installed
```

Similarly, for ROS 2:

```
source /opt/ros/foxy/setup.bash
source ~/ros2/install/setup.bash # ROS 2 packages have to be installed anyway
```

1.2 Typical organization

The most classical organization is to use only two workspaces: the ROS installation and your own workspace, where you put all packages that you download (and are not available as Debian packages), and packages you work on. Another, classical organization is to use three workspaces, for example on ROS 2:

- The lowest-level workspace is the core one `/opt/ros/foxy`, that is installed from Debian packages;
- A median-level one for packages that you want to use but are not available as Debian packages. This workspace will be compiled once and rarely modified. It can be placed somewhere in your home directory, or (as in the VM) somewhere in `/opt` to remember that it should be treated almost like official packages;
- The high-level workspace is placed inside the user home directory, this is where we put the home-made packages (or the lab packages) we work on.

The `.bashrc` would in this case source 3 workspaces for ROS 1, and another 3 for ROS 2. Unfortunately the environment needs to be cleared to go from ROS 1 to 2 or back to 1. This is the goal of the `ros_management` tool.

1.3 The `ros_management` tool

This tool enables easily switching between ROS 1 and ROS 2. It is already setup on the computers in Centrale Nantes and on the Virtual Machines. It can also be obtained here if needed.
This tool needs only 3 lines in the `.bashrc` file:

- define ROS 1 workspaces in overlay ordering
- define ROS 2 workspaces in overlay ordering
- source the `ros_management.bash` file

For example, on the Virtual Machine with 3 workspaces:

```
ros1_workspaces="/opt/ros/noetic /opt/local_ws/ros1 ~/ros"
ros2_workspaces="/opt/ros/foxy /opt/local_ws/ros2 ~/ros2"
source ~/.ecn_install/ros_management.bash
```

With this, going to ROS 1 environment is just calling `ros1ws`, same with `ros2ws` for the ROS 2 environment. This can be changed in `.bashrc` if you mainly work on ROS 2.
For each workspace, the root folder is first checked for a `setup.bash` file, then the `install` folder, and finally `devel`. If none are found, the workspace is ignored.

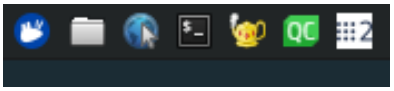
1.4 When to source ROS workspaces

Sourcing workspaces is the first thing to do when running a terminal. When sourcing workspaces, all currently available packages are detected and can be used. This is usually done in the

`.bashrc` file, either for ROS 1 or ROS 2.

It is also important to source the suitable environment when using a program that should interact with ROS. For example, using an IDE to program a package or a C++ node. The safest way to run, e.g QtCreator, is to do it from a terminal after calling either `ros1ws` or `ros2ws` (or sourcing your workspaces with the official way).

On ECN computers (and the VM), the top-left icons already enable to run QtCreator with:



- ROS 1 workspaces from the classical QtCreator icon
- ROS 2 workspaces from the ROS 2 icon

Finally, each time a package was compiled or installed, the workspace needs to be sourced again in order to enable its discovery. This means that if you install or re-compile a package in an IDE or in a terminal, other terminals will not be aware of these changes until the workspaces are sourced again (manually or just by closing / reopening another terminal).

This can lead to classical errors telling that a package is not known, or a node or a file inside a package are not found.

1.5 Some links

Here is the official documentation for ROS 1 and ROS 2 workspaces.

2 Compiling workspaces

In this section we detail how and when to use ROS-specific compilation tools, versus classical ones such as an IDE.
Both ROS 1 and ROS 2 propose shortcuts to compile all packages from a given workspace:

- In ROS 1, run `catkin build` from anywhere inside the workspace or its source tree
- In ROS 2, run `colcon build --symlink-install` from the root of the workspace

We can also require to compile only one package (it will also compile its dependencies from the same workspace, if any):

- In ROS 1, run `catkin build <package name>`
- In ROS 2, run `colcon build --symlink-install --package-select <package name>`

The `ros_management` tool gives the `colbuild` shortcut to run `colcon build --symlink-install`.

2.1 ROS manages the package-level compilation

`catkin` and `colcon` are high-level compilation tools that basically call `CMake` or a Python setup (for pure Python packages in ROS 2), but with a tailored configuration (use of the `build, devel, install` folders). As such, *they* are the tools that manage the compilation of the structure of a package. By structure we mean:

- Files to be compiled and potential options (like Debug mode in C++)
- Files to be installed during the compilation (launch files, maps, URDF, etc.)

A very important thing not to break a workspace is to let `catkin` and `colcon` do the work as soon as the structure was modified. This means:

- A new package was added (creation / download) to this workspace
- The structure of a C++ project was changed (any modification of `CMakeLists.txt` will do)
- New files are to be installed

2.2 IDEs manage the code-level compilation

Once a package was created or downloaded, and compiled a first time with the suitable ROS tool, it can be loaded into an IDE to ease the development. All IDEs that manage CMake are fine, but the `build` folder is imposed by ROS (so that ROS can find the executables and libraries that are compiled in this package).
For QtCreator, a tool is proposed to automatically configure the IDE with the correct paths.
As such, the procedure to load a package in an IDE is:

1. Get or create the package in the suitable workspace
2. Compile it, with `catkin` or `colcon`

3. Go to the root of the package and configure the IDE (QtCreator: call `gqt`)
4. Open the `CMakeLists.txt` file in the IDE (that should have been run with the correct sourced workspaces)

Working with the actual code can be done in a classical way (executables can be run and debugged from the IDE), with the only exception when you want to modify the `CMakeLists.txt` file. In this case:

1. Close the IDE (some of them run CMake automatically when the `CMakeLists.txt` file is motified)
2. Do the modification in `CMakeLists.txt`
3. Compile the package with `catkin` or `colcon`
4. Re-open the package inside the IDE and go code the `.h / .cpp` files

2.3 Running a node with parameters or remappings

An IDE will by default only run the raw executables, without particular parameters or remappings. In ROS though, nodes are classicaly run from a launch file that passes additional arguments.

If a need comes to debug / investigate an executable with particular options, the simplest thing is to temporarily run it as the launch file would. This is done by passing arguments such as:

- Remapping topics
  - ROS 1: `<node> <prev_topic>:=<new_topic>`
  - ROS 2: `<node> --ros-args -r <prev_topic>:=<new_topic>`
- Passing parameters
  - ROS 1: `<node> __<param>:=<value>`
  - ROS 2: `<node> --ros-args -p <param>:=<value>`
- Changing the namespace
  - ROS 1: `<node> __ns:=<namespace>`
  - ROS 2: `<node> --ros-args -r __ns:=<namespace>`

Another option is to (temporarily) give as default values (for parameters and topics) the one we want to use for debugging.

3 Dealing with ROS-specific errors

By *ROS-specific errors* we mean something else than a C++ code not compiling because you have forgotten a semi-colon somewhere. We also do not consider the *plumbing* errors where node cannot communicate simply because they do not use the same topics.
Such typical compilation or runtime errors are:

- A dependency (from this workspace or another) is suddenly not found at compilation
- CMake configuration fails in the IDE
- The executable that you run in the IDE is not the one that is run by calling `rosrun` or `ros2 run`

3.1 Usual error

Most of the time, we want to test our code too fast and forget to re-source the workspace properly in the terminal. Just run either `ros1ws` or `ros2ws` before calling things that use your latest modifications.

3.2 Ill-configured package, level 1

It may happen that CMake was called by the IDE but not by the ROS tool. In this case, some files would have been placed in standard `CMake` locations, that are not standard ROS ones. Quitting the IDE and compiling the package with the ROS tool might do the trick.
In particular, this is the case if you detect that the current executable is not the one run by ROS. If you have any doubt, just add a `std::cout` to be sure.

3.3 Ill-configured package, level 2

If a problem with a particular package is not resolved by recompiling it with `catkin` or `colcon`, the next step is to delete all compilation files for this particular package:

- ROS 1: call `catkin clean <package name>`
- ROS 2: delete the package folders in `build/` and `install/`

Re-source the workspace and re-compile the package to hope for any improvement.

3.4 Ill-configured workspace

If nothing is really working (and you are sure it is a compilation / runtime ROS thing) then you can just reset the whole workspace you were working on. Delete all folders except for `src` and recompile the whole workspace (or if you are in a hurry, only the packages you need do to the current work). Do not forget to source the workspaces after deletion and after re-compilation.

This is where using multiple workspaces comes handy, as you will have eventually to recompile all packages from this workspace.

4 Additional links

- Official documentation on ROS and IDEs
- Popular questions on which IDEs ROS developpers use
- Same with ROS 2

By experience, some developpers only compile their packages with the ROS tools and use basic text editors to code. This is a way to be sure the IDE will not do any extra, unwanted things during compilation or runtime. Yet, using an IDE is quite useful for auto-completion, structure exploration (node and message structures).

All IDEs require some kind of tweaking to work with ROS packages. The QtCreator tool is just one among others.

VS Code seems to be a widely-used IDE for ROS and ROS 2, as it provides plugins tailored for these specific file architectures.