ROS 2
Workflow and basic communication

Roberto Masocco

`roberto.masocco@uniroma2.it`

University of Rome Tor Vergata
Department of Civil Engineering and Computer Science Engineering

May 10, 2024

TOR VERGATA
UNIVERSITY OF ROME

School of Engineering

The **robotics industry** is evolving rapidly towards the best practices adopted in the **software industry** in the last 30 years.

ROS 2 is an open-source **middleware** for the development of robotics software and **distributed** control architectures.

ROS 2 is currently based on the **DDS** middleware to implement the communication layer, but this may change in the near future because of **scalability issues** and **lossy networks**.

Today, it is the **de facto** standard for the development of robotic applications, and it is supported by a **vast community** of developers and researchers.

This lecture is [here](#).

Roadmap

1 ROS 2 development workflow

2 C++ bootstrap

3 Message topics

Roadmap

1 ROS 2 development workflow

2 C++ bootstrap

3 Message topics

On **Ubuntu** systems, the easiest way to install ROS 2 Humble is through Debian packages.

The installation steps can be summarized as follows:

1. ensure that **locales** are properly configured;
2. **upgrade** your system (required because of some potential issues with udev that might break your installation ☺);
3. add and configure **apt repositories**;
4. **install** packages.

We have a script for this: `bin/ros2_humble_install.sh`.

After the installation is complete, in order to use **CLI tools** and have libraries available to **build packages**, you need to **source the installation**:

```
source /opt/ros/humble/setup.bash (there are also a .zsh and a .sh)
```

so that your shell, and all its child processes from then on, will know the **paths** of all the **executables**, **shared objects** (libraries), and **include directories** installed by ROS 2, plus many **environment variables**.

Additional commands are required to set up **command line completion** and other useful environment variables.
We have a script for this too: config/ros2_cmds.sh.
Source it, then enter ros2humble and you're good to go!

Currently, ROS 2 officially supports **two programming languages**:

- **C++** (C++17 in Humble)

- **Python** ($\geq$3.5, 3.10 works with Humble)

You can develop software packages using **only one** of them, or **both** at the same time (unofficially).

The two languages are both fully supported since they are **complementary**:

- **C++** allows to build complex software using **modern paradigms**, but also to easily access the **hardware**, **libraries**, and **operating system** APIs when required, and to **optimize** the code for **performance**.

- **Python** allows to **rapidly prototype** software, especially high-level modules, and to easily **interact** with the user and **visualize data**.

Note how one prioritizes other features with respect to the other, and vice versa.

> **This course will focus on C++, because of its better performance, major functionalities, and widespread use in the industry and robotics development community.**

The entire <u>ros2cli</u> suite is written in Python, and is fully **expandable**.
Python examples will still be provided and discussed whenever possible.

Language support
The build system

The ROS 2 build system supports both **C++** and **Python** packages through a common **package manager**: colcon (collective construction).

Spawned as a child project of the ROS community, its main features are:

- organization of the **build workspace** in a set of standard directories;

- **isolated** builds of packages, with **no pollution** of the system;

- automatic **dependency resolution** and **parallel** builds;

- support for **C**/**C++** packages through CMake;

- support for **Python** packages through setuptools.

Its configuration for a package can be found in the package.xml **manifest file**.

CMake is a **cross-platform** build configuration generator, which allows to build software using a **single**, **unified syntax** on all supported platforms.
Remember **Makefiles**? CMake is a compiler-agnostic **Makefile generator**.

We will write `CMakeLists.txt` files, which are essentially **scripts** that tell CMake how to build our software.

ROS 2 extends CMake with a set of **macros** and **functions**: the <u>ament</u> library.
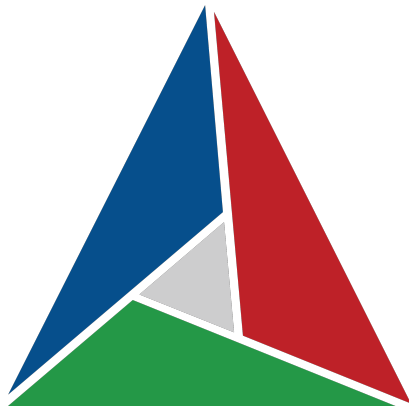

Figure 1: CMake logo.

`setuptools` is a **Python library** designed to ease the setup of Python projects, namely Python software packages.

Remember nightmarish `import` issues? `setuptools` is a **dependency resolver**.

We will write:

- `setup.py` files, in which the `setup(...)` function specifies the package's metadata and dependencies;

- `setup.cfg` files, which specify the location of all **executable scripts** in the package.

Figure 2: setuptools logo.

The `colcon` command you will use the most is

`colcon build`

which builds all packages in the current workspace (*i.e.*, directory and child directories).
It has many options, but the most useful ones are:

- `--event-handlers` to specify which **build events** to log (*e.g.*, `console_direct+`);

- `--packages-select` to specify which packages to build;

- `--symlink-install` to **symlink** the executables into the `install/` directory, instead of copying them (useful for Python packages);

- `--packages-up-to` to build a package and all its dependencies;

- `--packages-ignore` to ignore a package and all its dependencies.

> **Beware!**
>
> **During development, a good 85% of all issues happens during integration and build.**

We will see later on tools that help with this...

The workspace
Anatomy of a ROS 2 development directory

The organization of directories in a ROS 2 workspace is **standardized** because of `colcon`. We have, at least:

- `build/` (autogenerated), which contains the **build artifacts** of all packages;

- `install/` (autogenerated), which contains the **build products**;

- `log/` (autogenerated), which contains the **build logs**;

- `src/`, which contains the **source code** of all packages.

If you use Git, remember to add `build/`, `install/`, and `log/` to your `.gitignore` file!
Similarily, `colcon` ignores them when recursively looking for packages to build.

The workspace
Package creation

In the beginning was

```
ros2 pkg create <package_name>
```

which has way too many options. The main ones are:

- `--destination-directory src/`

- `--build-type <build_type>` (`ament_cmake` or `ament_python`)

- `--dependencies <package_name> ...`

Most of this stuff can be specified afterwards, eventually modifying the package.xml file.

Roadmap

Code examples

Find all course materials on **GitHub** at ros2-examples (`humble` branch).

The repository is organized as a **ROS 2 workspace** ready to be built, and intended to support Visual Studio Code as **IDE**. Find all information in `README.md`.

It is also organized with **Docker containers** in mind, and supports the automated build of development containers in VS Code (more on this later).
Such containers are based on our **Distributed Unified Architecture** project, which is part of Roberto Masocco's PhD thesis.
Their inner workings are totally transparent, but if you're curious see `dua_template.md`.

> Suggestion: clone it and checkout your own branch locally, to be still able to get and `merge` updates from remote.

C++ fundamentals

Back to basics

C++ has been developed from C, and is a **compiled**, **strongly-typed** (mostly) language.
Its main features began as extensions of C to support modern **object-oriented programming**
and **generic programming** paradigms, but it has evolved into much more.

In order to get started with ROS 2, a minimal subset of its features is required.
Dust off your C programming skills, then add:

- **Object-oriented programming**

- **Namespaces**

- **Templates**

- **Smart pointers**

C++ fundamentals
Object-oriented programming

```cpp
1 class MyClass : public ParentClass
2 {
3 public:
4    MyClass();
5    // ...
6 protected:
7    // ...
8 private:
9    // ...
10 };
```

Listing 1: Example of definition of a C++ class.

Pay attention to <u>inheritance</u> rules.

C++ was designed to allow for the developmnent of large codebases, which integrated libraries and code from potentially different sources.

Subdivision of the **global namespace** is necessary to avoid naming collisions between multiple libraries, resolved with the **:: operator**.
It works like the dot in web URLs (*e.g.*, ing.uniroma2.it).

Names may become very long, so usually they are hidden with typedef.

C++ fundamentals

Namespaces

```cpp
namespace MyLib {
  void foo() { /* Does something */ }
} // This is typically done for libraries

class MyClass
{
public:
  void foo() { /* Does something as well */ }
} my_obj; // Watch out for the ';'!

MyLib::foo(); // This is calling foo from MyLib!
my_obj.foo(); // This is calling foo from MyClass!
```

Listing 2: Example of namespaces usage.

Classes or functions whose **implementation depends on some data type**.
When instantiated or called with a specific type, **the corresponding code is generated by the compiler**.

```cpp
1 std::vector<int> int_vector;
2 std::vector<double> double_vector;
```

Listing 3: Example of objects of the template class std::vector.

It is possible to write **specialized code** for a specific data type in the template definition.
These too make names very long, so are usually typedef'd.

A kind of **smart pointer** (there are also `unique` and `weak`) that also holds an **usage counter**, incremented by every function or object that is handling the pointer.
When the `shared_ptr` is destroyed, if the counter is zero the pointed object is also destroyed and its memory deallocated.

```
1 {
2   // A new scope starts here
3   std::shared_ptr<rclcpp::Node> node =
4     std::make_shared<rclcpp::Node>("my_node");
5 }
6 // Here the node and its pointer have been destroyed!
```

Listing 4: Example of shared pointer creation.

Obviously `std::shared_ptr` is a template class.
ROS 2 heavily relies on them, and the `SharedPtr` alias is frequently defined.

Roadmap

1. ROS 2 development workflow

2. C++ bootstrap

3. **Message topics**

ROS 2 messages

A message is a **single data packet** sent over a **topic**, from **publisher nodes** to **subscriber nodes**, with a specific **QoS policy**.
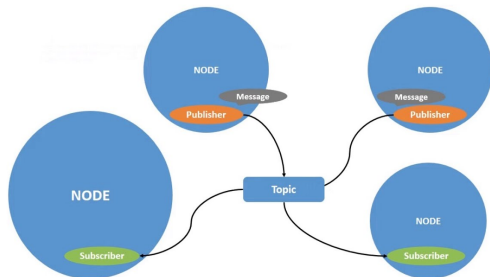

Figure 3: Example of a topic with multiple publisher and subscriber nodes.

Figure 3 is also an example of a simple **node graph**, a pivotal concept in a distributed context.

Interface files format is **derived from the DDS standard**, with data types resolved to machine types according to the platform being used[1].

Message file names end with `.msg`.

Things start very simply...

```
1 int64 data
```

Listing 5: Definition of the std_msgs/msg/Int64 message.

```
1 string data
```

Listing 6: Definition of the std_msgs/msg/String message.

---

[1] About ROS 2 interfaces (ROS 2 Humble docs)

... then escalate quickly!

```
 1 std_msgs/Header header
 2 # ^ This includes another message type as a sub-structure
 3
 4 uint32 height
 5 uint32 width
 6
 7 string encoding
 8
 9 uint8 is_bigendian
10 uint32 step
11 uint8[] data # This specifies a variable-length uint8 array
```

Listing 7: Definition of the sensor_msgs/msg/Image composite message.

Special values (*i.e.*, **constants**) or **default values** may be specified.

```
1 int64 MYNUM=1 # Must be of compatible type
2
3 int64 number
4 int64 number_with_default 2
```

Listing 8: Definition of an example message with a constant and a default value.

They are not bound to any field and will appear as **special selectable values** in the generated C++/Python libraries code.

ROS 2 adds its own **guidelines**[2], and installed interfaces can be inspected with

```
ros2 interface show
```

---

[2]ros2-examples/interfaces.md

Message topics
Quality of Service

A **QoS policy** for publishers/subscribers is a data structure with the following attributes:

- **History** (*keep last N* or *all*)

- **Depth** (queue size *N*)

- **Reliability** (*best-effort* or *reliable*, default: reliable)

- **Durability** (publishers resend all messages to "late-joiners")

- Deadline

- **Lifespan** (message expiration date)

- Liveliness

- Lease duration

Default **profiles** are available (e.g. **Sensor data**, **Service**...), see the <u>docs</u>.

The command line tool `ros2 topic` can be used to **inspect topics** and related entities.
It has a lot of **verbs**, the most important ones are:

- `list` (list all topics)

- `echo` (print messages to the console)

- `pub` (publish messages from the console)

- `hz` (print publishing rate and statistics)

- `info` (print information about a topic)

- `type` (print the message type)

each with many useful options.
Nodes can be inspected with the `ros2 node` command, and its many verbs.

Now go have a look at the first two example packages:

ros2-examples/src/cpp/topic_pubsub_cpp

ros2-examples/src/python/topic_pubsub_py

Exercises

- Install ROS 2 on a platform of your choice.

- Run the demo nodes.

- Inspect the demo topics.

- Interact with the demo nodes from the command line.

- Clone `ros2-examples` as suggested and rebuild the first example packages.

- Listen to the `/rosout` topic from the command line while other nodes run.