ROS Programming – Basics

Stefano Carpin

Department of Computer Science and Engineering
School of Engineering
University of California, Merced
https://sites.ucmerced.edu/scarpin
https//robotics.ucmerced.edu

Build tool and build system

- a ROS application consists of multiple nodes, possibly written in different languages and belonging to different packages
- build system: `ament_cmake` – an extension of CMake. Builds a single package
- build tool: `colcon` – uses the build system to build multiple packages, analyzes dependencies, etc.
- in ROS we use both

Creating and building a workspace

Before creating your packages, you should create your own workspace which will include your packages

```
$ mkdir -p CSE180/src
$ cd CSE180
$ colcon build
```

- you just built an empty workspace
- `colcon build` must be executed from the workspace folder
- packages go in the `src` folder
- `colcon` created various folders and files. To make your workspace visible to ROS as an *overlay* you must execute the following command from the worskspace folder

  ```
  $ . install/local_setup.bash
  ```

- official documentation suggests to do this from a shell different from where you run `colcon build`

Creating a package

- Must use `ros2 pkg` to create folders with the right structure

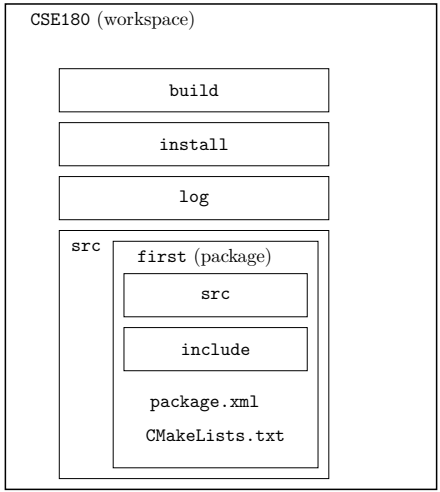
```
$ cd CSE180/src
$ ros2 pkg create first
```

- just created a package called `first`
- packages always go into the `src` folder of the workspace
- besides folders, created files `CMakeLists.txt` and `package.xml`
- erasing a package? just remove its folder
- can also indicate the build system

```
ros2 pkg create --build-type ament_cmake  first
```

- the package can now be built with `colcon`

Structure of a workspace and packages



package.xml

- an XML file included in every package
- includes information useful to build and distribute packages (e.g., dependencies)
- basic skeleton created by `ros2 pkg create`
- you typically have to update the list of package dependencies

`package.xml` example

`MRTP/MRTP/unsorted/package.xml`

CMakeLists.txt

- `CMake` file
- specifies sources, libraries, dependencies, etc.
- basic skeleton created by `ros2 pkg create`; most can be used “as is”
- typically must add some entries for each executable in the package (source, packages it depends on, and more)
- more later...

Creating ROS executables

- built on top of `rcl`: ROS client library
- `rclcpp` exposes `rcl` through a set of C++ classes and functions
- `rclpy` offers equivalent functionalities for Python
- `rclcpp` is part of a package called `rclcpp`; all code we will write depends on this package

ROS executable typical strucure

1. initialize ROS system
2. instantiate a node, and setup communication via topics (services, actions)
3. *do the job* – typically exchanging messages, calling/offering services, etc.
4. shut down

- spinning: the operation of taking care of incoming and outgoing messages through topics; almost all nodes must call one of the spinning functions

First ROS Node: publishing

Create a new package called `talklisten`

```
cd ~
mkdir -p MRTP/src
cd MRTP/src
ros2 pkg create --build-type ament_cmake talklisten
```

Create file

`MRTP/MRTP/src/talklisten/src/talker.cpp`

Take Home Messages

- always include `rclcpp.hpp` and create a node handler (pointer)
- familiarize yourself with the package `example_interfaces` (more later)
  - use `ros2 interface show` to find out the structure of its messages
- use fully qualified names for ROS classes (not a strict requirement, just common use)
- to keep things simple, in the beginning we will have just one node per executable
  - composition is an approach through which an executable generates multiple nodes (more later...)

Second ROS Node: subscribing

`MRTP/MRTP/src/talklisten/src/listener.cpp`

Take Home Messages

- subscribers must call `ros::spin` or `ros::spin_some` (see later) to retrieve messages
- messages are processed in *callback* functions
- use `RCLCPP_INFO` and associated macros to print to the screen
- callback functions are supposed to be quick to terminate because they may be called very frequently

`spin` **VS** `spin_some`

- `spin_some`: when called all messages pending are processed (sent or received); then the function returns
- `spin`: non returning function that keeps checking for messages
- think carefully about the logic of your node before deciding which one to call

Updating `package.xml`

`MRTP/MRTP/src/talklisten/package.xml`

Updating `CMakeLists.txt`

`MRTP/MRTP/src/talklisten/CMakeLists.txt`

Build and run

- move to the workspace folder
- run `colcon build`
- source the overlay (different shell)
- use `ros2 run` to run the executables

More ROS examples

- `MRTP/MRTP/src/multipletopics/src/multipublish.cpp`
- `MRTP/MRTP/src/multipletopics/src/multisub.cpp`
- Let's add our own launch file with multiple consoles
  `MRTP/MRTP/src/multipletopics/launch/multipletl.launch.xml`
- update `package.xml`
  `MRTP/MRTP/src/multipletopics/package.xml`
- and `CMakeLists.txt`
  `RTP/MRTP/src/multipletopics/CMakeLists.txt`

ROS STREAMS

RCLCPP_INFO : information to the user

RCLCPP_DEBUG : information for debugging

RCLCPP_WARN : communicate unusual conditions (warnings).

RCLCPP_ERROR : diagnostic about recoverable errors.

RCLCPP_FATAL : diagnostic about unrecoverable errors.

Standard messages

C++ data type	ROS message
`bool`	`example_interfaces::msg::Bool`
`char`	`example_interfaces::msg::Char`
`unsigned char`	`example_interfaces::msg::UInt8`
`int`	`example_interfaces::msg::Int32`
`unsigned int`	`example_interfaces::msg::UInt32`
`long int`	`example_interfaces::msg::Int64`
`unsigned long int`	`example_interfaces::msg::UInt64`
`float`	`example_interfaces::msg::Float32`
`double`	`example_interfaces::msg::Float64`

Table: Correspondences between C++ elementary data types and ROS standard messages.

- `std::string` can be sent using messages of type `example_interfaces::msg::string`

Sending/Receiving Multidimensional Arrays

- For arrays or multidimensional structures, we use `XXXMultiArray`, e.g., `Int32MultiArray`
- let us use `ros2 interface` to understand the structure of `Int32MultyArray`
- `MRTP/MRTP/src/examples/src/sendarray.cpp`
- `MRTP/MRTP/src/examples/src/subarray.cpp`
- `MRTP/MRTP/src/examples/src/sendmatrix.cpp`
- `MRTP/MRTP/src/examples/src/submatrix.cpp`

`MultiArray` and `MultiArrayLayout`

- data is stored linearly in `data`; `layout` maps multidimensional indices into a linear structure
- `stride` is used to store separation between successive elements
- store outer dimensions first
- If the multidimensional structure has n dimensions $A_1 \times A_2 \times A_3 \times \ldots A_n$, then the stride S_i of the i-th dimension is $S_i = A_i \times A_{i+1} \times \cdots \times A_n$. An element is identified by n indices $(i_1, i_2, \ldots, i_n)$. Its position in the linear data structure is (assuming 0 offset):

$$i_1 \cdot S_2 + i_2 \cdot S_3 + \cdots + i_{n-1} \cdot S_n + i_n$$

Subscribing and Publishing from the Same Node

- often nodes *both* publish and subscribe to topics
- two approaches:
  - publish from the handler function
  - use `spin_some`
- `MRTP/MRTP/src/examples/src/pubsub.cpp`
- publisher objects shall not be declared in the handler function