Masters in Critical Computing Systems

LAB 2 – ROS2 Introduction and CLI

Ricardo Severino
Luis Miguel Pinho

Goals

Understanding ROS2 workspaces, underlays and overlays.

Understanding ROS2 packages and the tree structure.

Compiling ROS2 code with colcon.

Understanding ROS2 nodes and services. Creating custom nodes and services.

Intro to ROS2 concepts

In this lab you will be exploring and consolidating ROS2 concepts such as nodes, topics, and services.

Jump in your ade environment and let us begin by running some of the examples you went through in the previous lecture.

```
$ ade start --enter
ade $ source /opt/ros/foxy/setup.bash
ade $ mkdir ros2_iacos_ws
ade $ cd ros2_iacos_ws
ade $ git clone https://github.com/ros2/examples.git src/examples
ade $ cd src/examples/
ade $ git checkout foxy
ade $ cd ../..
ade $ colcon build --symlink-install
```

You have now compiled all the examples. List the ros2_examples_ws contents and analyse the results.

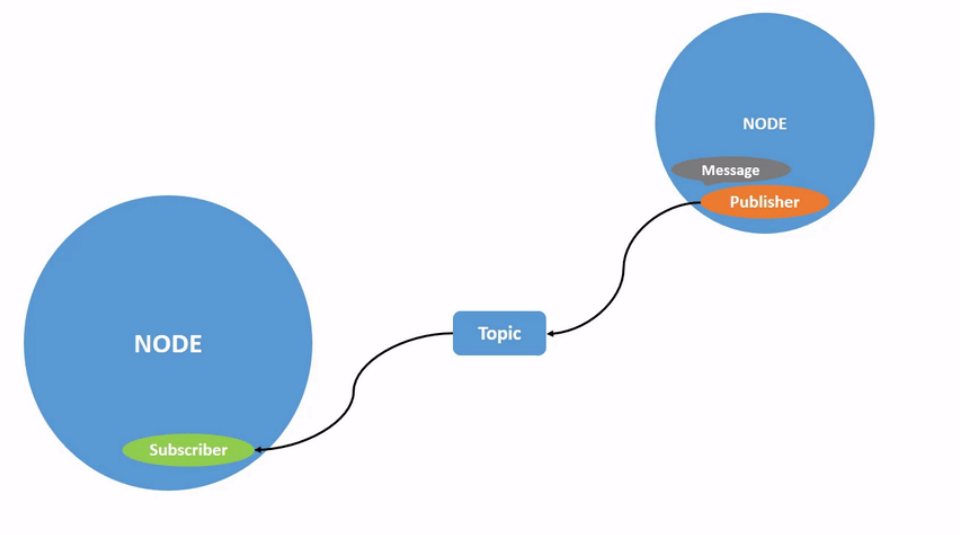
```
ade $ ls
```

Notice you now have a **build**, **install**, **log** and **src** folders. By invoking **colcon**, you have created a new **workspace**. A ROS workspace is a directory with a particular structure. By default it will create the following directories as peers of the src directory:

- The build directory will be where intermediate files are stored. For each package a subfolder will be created in which e.g. CMake is being invoked.
- The install directory is where each package will be installed to. By default each package will be installed into a separate subdirectory.
- The log directory contains various logging information about each colcon invocation.

Lets begin by looking into nodes and publishers.

ROS 2 breaks complex systems down into many modular nodes. Topics are a vital element of the ROS graph that act as a bus for nodes to exchange messages.



A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics.

First, let's **source your overlay**.

```
ade $ cd ros2_iacos_ws
ade $  source install/setup.bash
```

Let's first explore the CLI

```
ade $ ros2 --help
usage: ros2 [-h] Call `ros2 <command> -h` for more detailed usage. ...

ros2 is an extensible command-line tool for ROS 2.

optional arguments:
  -h, --help            show this help message and exit

Commands:
  action     Various action related sub-commands
  bag        Various rosbag related sub-commands
  component  Various component related sub-commands
  daemon     Various daemon related sub-commands
  doctor     Check ROS setup and other potential issues
  interface  Show information about ROS interfaces
  launch     Run a launch file
  lifecycle  Various lifecycle related sub-commands
  multicast  Various multicast related sub-commands
  node       Various node related sub-commands
  param      Various param related sub-commands
  pkg        Various package related sub-commands
  run        Run a package specific executable
  security   Various security related sub-commands
  service    Various service related sub-commands
  test       Run a ROS2 launch test
  topic      Various topic related sub-commands
  wtf        Use `wtf` as alias to `doctor`

  Call `ros2 <command> -h` for more detailed usage.
```

Take a while to understand the implemented functionalities. Lookup further information about the run command:

```
ade $ ros2 run --help
usage: ros2 run [-h] [--prefix PREFIX] package_name executable_name ...

Run a package specific executable

positional arguments:
  package_name     Name of the ROS package
  executable_name  Name of the executable
  argv             Pass arbitrary arguments to the executable

optional arguments:
  -h, --help       show this help message and exit
  --prefix PREFIX  Prefix command, which should go before the executable. Command
                   must be wrapped in quotes if it contains spaces (e.g. --prefix
                   'gdb -ex run --args').
```

Run a Publisher Node

We can now go ahead and run a simple publisher node. (TAB is your friend for auto-completion). Run the publisher_member_function executable. You should see a Hello World message being published.

In a new terminal window, in your ade environment, try to **echo** the topic that in being published.

Use **ros2 topic --help** for more information about the available tools.

Identify the topic publishing rate.

By relying on Simple Publisher and Subscriber analyse the publisher code.

Change the message being printed and the publishing rate to 10 Hz, and topic name to /hello.

Compile it. Confirm the topic name has changed by using your ROS2 CLI.

Now, let us look at the subscriber. Take a while to look to the **ros2_iacos_ws/src/examples/rclcpp/topics/minimal_subscriber/member_function.cpp**

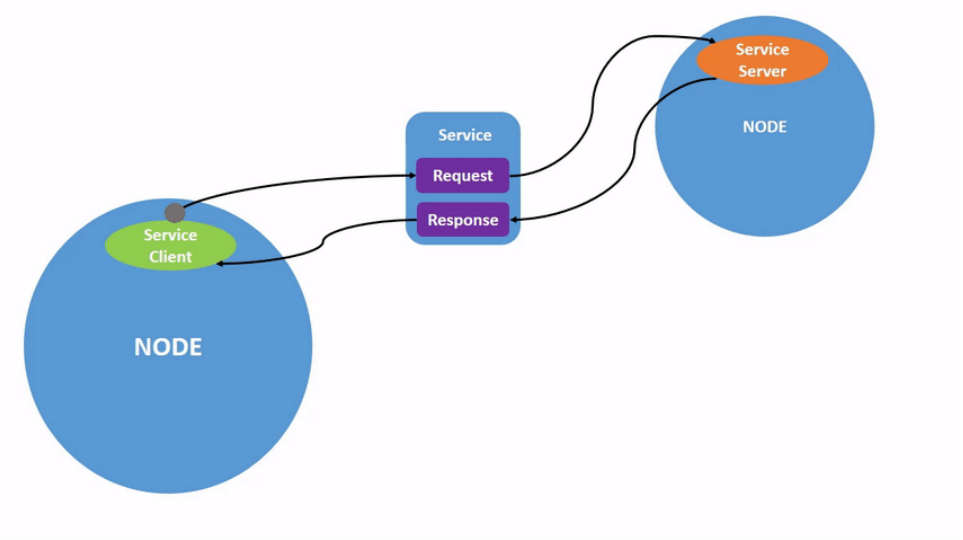
Change the topic accordingly and re-compile the code invoking colcon build.

TIP: you can rely on the **--packages-select** option to compile a single package.

Run the subscriber and check the result.

Run a Service

Services are another method of communication for nodes in the ROS graph. Services are based on a call-and-response model, versus topics' publisher-subscriber model. While topics allow nodes to subscribe to data streams and get continual updates, services only provide data when they are specifically called by a client.



When nodes communicate using services, the node that sends a request for data is called the client node, and the one that responds to the request is the service node. The structure of the request and response is determined by a .srv file.

In your IACOS T2 Lecture you explored the code for a simple integer addition system; one node requests the sum of two integers, and the other responds with the result. Let's quickly go through the lecture example in your development environment. This time you will be setting up the package and code.

Start by creating a new ROS2 package

Open a new ade terminal and source your ROS 2 installation so that ros2 commands will work.

Recall that packages should be created in the **/src** directory, not the root of the workspace. Navigate into the **ros2_iacos_ws/src** and and create a new package:

```
ade $ ros2 pkg create --build-type ament_cmake cpp_srvcli --dependencies rclcpp
example_interfaces
```

Your terminal will return a message verifying the creation of your package **cpp_srvcli** and all its necessary files and folders.

The --dependencies argument will automatically add the necessary dependency lines to **package.xml** and **CMakeLists.txt**.

example_interfaces is the package that includes the .srv file you will need to structure your requests and responses (example_interfaces/srv/add_two_ints.hpp).

To check this service interface attributes, use your CLI:

List available interfaces, check that **example_interfaces/srv/AddTwoInts** is in the list.

```
ade $ ros2 interface list -s
Services:
    action_msgs/srv/CancelGoal
    composition_interfaces/srv/ListNodes
    composition_interfaces/srv/LoadNode
    composition_interfaces/srv/UnloadNode
    diagnostic_msgs/srv/AddDiagnostics
    diagnostic_msgs/srv/SelfTest
    dwb_msgs/srv/DebugLocalPlan
    dwb_msgs/srv/GenerateTrajectory
    dwb_msgs/srv/GenerateTwists
    dwb_msgs/srv/GetCriticScore
    dwb_msgs/srv/ScoreTrajectory
    example_interfaces/srv/AddTwoInts
    example_interfaces/srv/SetBool
    example_interfaces/srv/Trigger
    geographic_msgs/srv/GetGeoPath
```

Now check the interface. The first two lines are the parameters of the request, and below the dashes is the response.

```
ade $ ros2 interface show example_interfaces/srv/AddTwoInts
int64 a
int64 b
---
int64 sum
```

Update package.xml

Because you used the --dependencies option during package creation, you don't have to manually add dependencies to package.xml or CMakeLists.txt. As always, though, make sure to add the description, maintainer email and name, and license information to package.xml.

```
<description>C++ client server tutorial</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

Write the service nodes (server)

Inside the **ros2_ws/src/cpp_srvcli/src** directory, create a new file called **add_two_ints_server.cpp**.

With your favourite code editor, start by placing your includes which are your dependencies: **rclcpp** and **add_two_ints.hpp**. Also include **<memory>** for the Smart Pointer.

Go for the main function, pass the traditional **(int argc, char *argv)** arguments and write the code to initialize ROS 2 C++ client library.

rclcpp::init(argc, argv);

Create a node named **add_two_ints_server** using the rclcpp::Node class and a smart pointer.

```
std::shared_ptr <rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_server");
```

At the bottom **spin the node**, making the service available and **shutdown** for a clean exit.

```
rclcpp::spin(node);
rclcpp::shutdown();
node = nullptr;
```

Now, we want to create a **service named add_two_ints** for that node and automatically advertises it over the networks with the **&add** method. Use the **rclcpp::create_service** function. The type of service is example_interfaces::srv::AddTwoInts.

```
rclcpp::Service<example_interfaces::srv::AddTwoInts>::SharedPtr service =
node->create_service<example_interfaces::srv::AddTwoInts>("add_two_ints", &add);
```

Print a log message when it's ready:

```
RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to add two ints.");
```

This part must be placed before the spin function. Now, let's create the add service processing function you just mentioned. Above the main(), create:

```
void add(const std::shared_ptr<example_interfaces::srv::AddTwoInts::Request> request,
         std::shared_ptr<example_interfaces::srv::AddTwoInts::Response>      response)
{
    response->sum = request->a + request->b;
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Incoming request\na: %ld" " b: %ld",
        request->a, request->b);
       RCLCPP_INFO(rclcpp::get_logger("rclcpp"),  "sending  back  response:  [%ld]",  (long
int)response->sum);
}
```

Notice how we fetch the request arguments and write the response.

Add executable

To generate an executable out of this you must use the **add_executable** macro. Add the following code block to **CMakeLists.txt** just bellow the last find_package() to create an executable named server:

```
add_executable(server src/add_two_ints_server.cpp)
ament_target_dependencies(server
rclcpp example_interfaces)
```

So ros2 run can find the executable, add the following lines to the end of the file, right before ament_package():

```
install(TARGETS
  server
  DESTINATION lib/${PROJECT_NAME})
```

Build the package. And run the service. Stop with CTRL+C.

If there was an error, find the package that is missing and add it to CMakeList.. tip: something to do with example_interfaces.

Write the client node

Inside the **ros2_ws/src/cpp_srvcli/src directory**, create a new file called **add_two_ints_client.cpp**.

You can go ahead and add the client to your **CMakeLists.txt** already.

Add this to the top of the **add_two_ints_client.cpp**

```
#include "rclcpp/rclcpp.hpp"
#include "example_interfaces/srv/add_two_ints.hpp"
#include <chrono>
#include <cstdlib>
#include <memory>
using namespace std::chrono_literals;
```

Create your main() and inside:

Initialize ROS 2 C++ client library and process its arguments making sure the correct number is passed.

```
rclcpp::init(argc, argv);
if (argc != 3) {
RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "usage: add_two_ints_client X Y");
return 1;
}
```

Setup the new client node and create the client:

```
std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_client");
rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedPtr client =
    node->create_client<example_interfaces::srv::AddTwoInts>("add_two_ints");
```

Create the request. Its structure is defined by the .srv file mentioned earlier.

```
auto request = std::make_shared<example_interfaces::srv::AddTwoInts::Request>();
request->a = atoll(argv[1]);
request->b = atoll(argv[2]);
```

Then wait for the service every 1 second.

```
while (!client->wait_for_service(1s)) {
if (!rclcpp::ok()) {
RCLCPP_ERROR(rclcpp::get_logger("rclcpp"),  "Interrupted  while  waiting  for  the  service.
Exiting.");
return 0;
}
RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting again...");
}
```

If found, it sends the request and the node spins until it receives its response, or fails.

```
auto result = client->async_send_request(request);
// Wait for the result.
if (rclcpp::spin_until_future_complete(node, result) ==
rclcpp::FutureReturnCode::SUCCESS)
{
RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Sum: %ld", result.get()->sum);
} else {
```

```
RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Failed to call service add_two_ints");
}
```

Finally, shutdown the node.

Build the package and run the server and the client in separate terminal windows.

Your own Service - Rectangle Area Computation

Now, it is your turn to build a new service with a custom interface. The objective is to create a service that can compute the area of a rectangle. For that, let's create a new package **tutorial_interfaces** in which you will define the new interface.

Go to **ros2_iacos_ws/src**

```
ade $ ros2 pkg create --build-type ament_cmake tutorial_interfaces
```

Create a new service interface file.

```
ade $ cd tutorial_interfaces
ade $ mkdir srv
ade $ touch srv/RectangleArea.srv

Complete it with the needed information.
```

To convert the interfaces you defined into language-specific code (like C++) so that they can be used in those languages, add the following lines to CMakeLists.txt:

```
find_package(rosidl_default_generators REQUIRED)
rosidl_generate_interfaces(${PROJECT_NAME}
"srv/RectangleArea.srv"
)
```

Add to your **package.xml** these dependencies.

```
<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Compile and check if the interface pops up in the list with

ade $ ros2 interface list

Now add to the **cpp_srvcli** package two new executables (a server and a client) that will use this interface. Rely on the previous AddTwoInts service example for this.

Build everything back and test the new service!

Congratulations!

Credits
Version 1.0, Ricardo Severino, based on:
Autoware Training
ROS2 Tutorials