Masters in Critical Computing Systems

LAB 3 – Advancing on ROS2

Ricardo Severino
Luis Miguel Pinho

Goals

Learn some important CLI commands to work with topics and services

Build an Action Server and Client

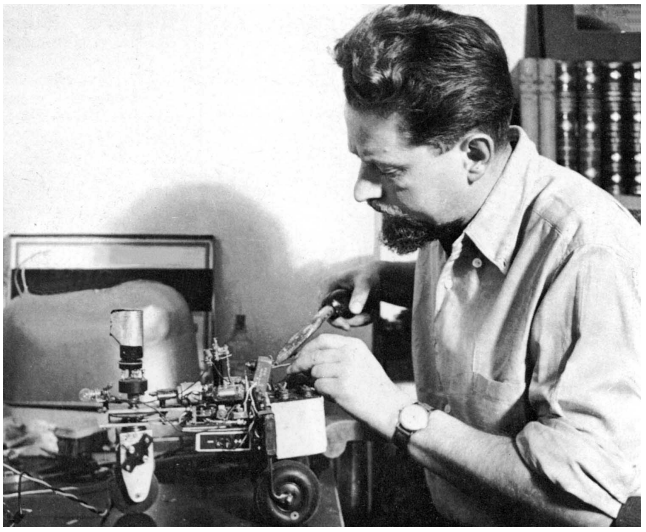
Learn about ROS Bags

Introduction

In this lab, we will be learning about a new package by using the Command Line Interface (CLI) tools. The Turtlesim package will be our test package. Next we will be implementing a simple action. Turtlesim is a lightweight simulator for learning ROS 2. It illustrates what ROS 2 does at the most basic level, to give you an idea of what you will do with a real robot or robot simulation later on.

To further explore the ROS2 Robotics Operating System we will start by relying on the turtlebot package. Why turtlebots you may ask?

William Grey Walter's influence is still felt today. He referred to his robots as 'turtles' and, as you will see, the moniker stuck. The image below is William Walter's Elsie without her protective covering.

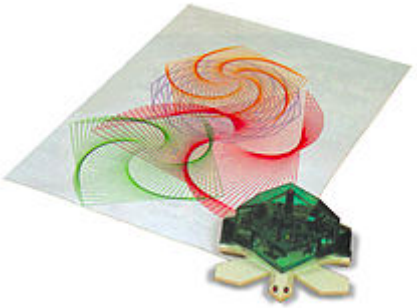


Learn more about Grey Walter's tortoises.

Long after William Walter's work with Elmer and Elsie, Dr. Seymour Papert, a professor at MIT began to use turtle robots for education. One of the characteristics of Papert's robots was their ability to draw on paper. In addition to being involved with the creation and development of MIT's turtle robots, Dr. Papert is also known as the creator and evangelist for the educational programming language LOGO.

Despite being a general-purpose language, LOGO is known for its use of "turtle graphics", a system which allows users to draw by sending simple commands to a robotic turtle. The robotic turtle mentioned here could be either a real turtle robot, or a virtual on-screen cursor within the LOGO programming environment.

Logo programming language developed by Wally Feurzeig, Seymour Papert and Cynthia Solomon in 1967. The image below shows an example of Valiant Technology's Turtle robot drawing on a sheet of paper.



Putting all together with turtlesim

Let's begin by installing the Turtlesim package and rqt:

```
ade:~$ sudo apt update
ade:~$ sudo apt upgrade
ade:~$ sudo apt install ros-foxy-turtlesim
ade:~$ sudo apt install ros-foxy-rqt*
```

Use command ros2 pkg executables to check if the package installed correctly

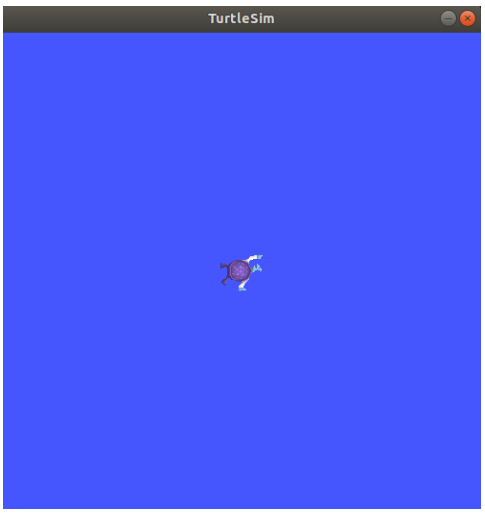
```
ade:~$ ros2 pkg executables turtlesim
ros2 pkg executables turtlesim
turtlesim draw_square
turtlesim mimic
turtlesim turtle_teleop_key
turtlesim turtlesim_node
```

You should see a list of installed turtlesim packages. The -–help option will give you more details about this command. --full-path will enable you to see where the packages are installed. Let's run the turtlesim_node.

```
ade:~$ ros2 run turtlesim turtlesim_node
```

A window such as the one depicted below will pop up and in your command line you will read something like the message below, informing about the spawning of a turtle and its position.

```
[INFO] [1666171569.815000395] [turtlesim]: Spawning turtle [turtle1] at x=[5.544445],
y=[5.544445], theta=[0.000000]
```



Open another terminal and run the draw_square node. Your turtle will begin drawing a square.

```
ade:~$ ros2 run turtlesim draw_square
```

By now, this seems magical, but I assure you it is not. Upon finding a new package you will need to understand its inner works. To do that, before going straight for the code, you should use the ROS command line interface (CLI) tools.

Using the CLI explore the available topics and services offered by that package.

```
ade:~$ ros2 node list
/draw_square
/turtlesim


ade $ ros2 node info /turtlesim
/turtlesim
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/color_sensor: turtlesim/msg/Color
    /turtle1/pose: turtlesim/msg/Pose
  Service Servers:
    /clear: std_srvs/srv/Empty
    /kill: turtlesim/srv/Kill
    /reset: std_srvs/srv/Empty
    /spawn: turtlesim/srv/Spawn
    /turtle1/set_pen: turtlesim/srv/SetPen
    /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
    /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
    /turtlesim/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /turtlesim/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /turtlesim/get_parameters: rcl_interfaces/srv/GetParameters
    /turtlesim/list_parameters: rcl_interfaces/srv/ListParameters
    /turtlesim/set_parameters: rcl_interfaces/srv/SetParameters
    /turtlesim/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  Service Clients:

  Action Servers:
```

```
    /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
  Action Clients:
```

This presents the subscribing and publishing topics offered by the node, together with the message structures. Also the available services and actions are reported. From the previous lab class you already checked a few ros2 topic commands.

Run **ros2 topic --help** to check the options.

`ade:~$ ros2 topic list -t` will show you the message types related to those topics.

```
/parameter_events [rcl_interfaces/msg/ParameterEvent]
/rosout [rcl_interfaces/msg/Log]
/turtle1/cmd_vel [geometry_msgs/msg/Twist]
/turtle1/color_sensor [turtlesim/msg/Color]
/turtle1/pose [turtlesim/msg/Pose]
```

Let's use this tool to explore topics, starting with a simple echo. Using --help notice you can also echo a message type instead.

```
ade:~$ ros2 topic echo /turtle1/pose
```

You should see a flow of pose messages conveying the **position** and **orientation** of the turtle. This output can be piped to a **.txt** file for instance (**ros2 topic echo /turtle1/pose > poses.txt**) or a **.csv** (**ros2 topic echo --csv /turtle1/pose > poses.csv**).

Topic Diagnostics

Check bandwidth utilization, particularly important when a lot of data is being handled in your system.

```
ade:~$ ros2 topic bw /turtle1/pose
```

Get the number and info of the topic subscribers

`ade$ ros2 topic info /turtle1/pose` or for more information

```
ade$ ros2 topic info /turtle1/pose -v
Subscription count: 1
Node name: draw_square
Node namespace: /
Topic type: turtlesim/msg/Pose
Endpoint type: SUBSCRIPTION
GID: 74.d0.10.01.ef.81.fc.7e.60.da.3c.e0.00.00.15.04.00.00.00.00.00.00.00.00
QoS profile:
  Reliability: RMW_QOS_POLICY_RELIABILITY_RELIABLE
  Durability: RMW_QOS_POLICY_DURABILITY_VOLATILE
  Lifespan: 9223372036854775807 nanoseconds
  Deadline: 9223372036854775807 nanoseconds
  Liveliness: RMW_QOS_POLICY_LIVELINESS_AUTOMATIC
  Liveliness lease duration: 9223372036854775807 nanoseconds
```

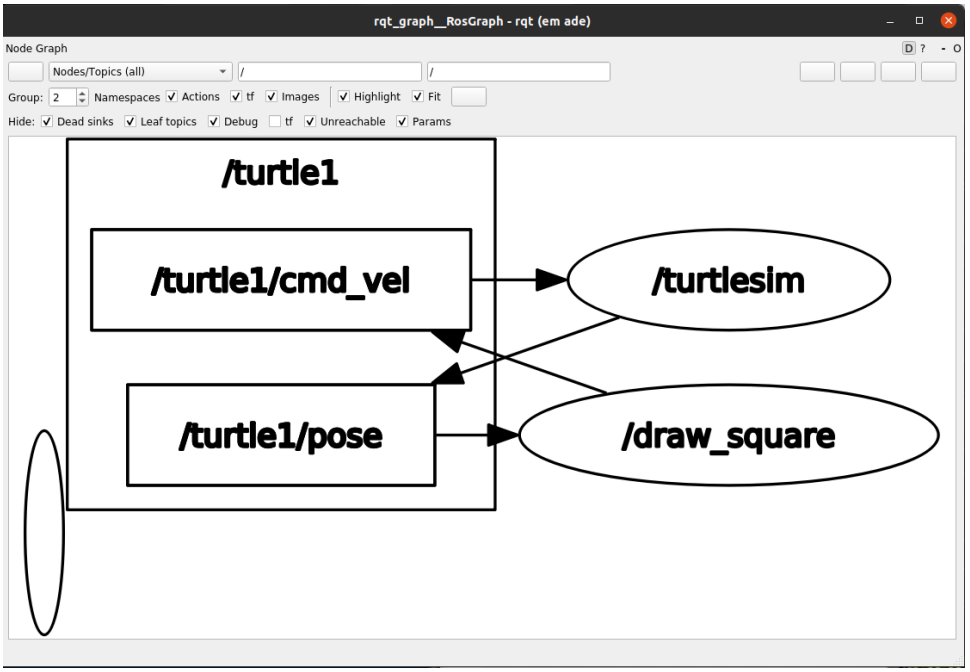
In the output it is mentioned that there is a certain message type used in that topic (turtlesim/msg/Pose). To know more about it:

```
ade$ ros2 interface show turtlesim/msg/Pose
float32 x
float32 y
float32 theta

float32 linear_velocity
float32 angular_velocity
```

There are also nice GUI tools to achieve this. Map these topics and nodes.

```
ade:~$ rqt_graph
```



Turtlesim Services

Considering what you've learned in the last lab class, list the services of turtlesim. You should get the information below:

```
/clear [std_srvs/srv/Empty]
/kill [turtlesim/srv/Kill]
/reset [std_srvs/srv/Empty]
/spawn [turtlesim/srv/Spawn]
/turtle1/set_pen [turtlesim/srv/SetPen]
/turtle1/teleport_absolute [turtlesim/srv/TeleportAbsolute]
/turtle1/teleport_relative [turtlesim/srv/TeleportRelative]
/turtlesim/describe_parameters [rcl_interfaces/srv/DescribeParameters]
/turtlesim/get_parameter_types [rcl_interfaces/srv/GetParameterTypes]
/turtlesim/get_parameters [rcl_interfaces/srv/GetParameters]
/turtlesim/list_parameters [rcl_interfaces/srv/ListParameters]
/turtlesim/set_parameters [rcl_interfaces/srv/SetParameters]
/turtlesim/set_parameters_atomically [rcl_interfaces/srv/SetParametersAtomically]
```

Using the ros2 service call (use --help) call the /reset service to reset the turtlesim.

Spawn a new turtle at a specific position.

```
ade:~$ ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta: 0.2, name:
'larry'}"
```

You can also use the rqt gui to explore the packages.

```
ade $ rqt
```

In Rqt you can also map the nodes but also interact with services and topics. In the Plugins menu, open Topic Monitor and select the pose topic. Check the info changing.

Publishing to a topic

You can also publish messages to a topic from the CLI. Stop the draw_square node, and run:

```
ade:~$ ros2 topic pub --rate 1 /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 3.0,
y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 3}}"
```

This command publishes a message to the /turtle1/cmd_vel topic every 1 second.

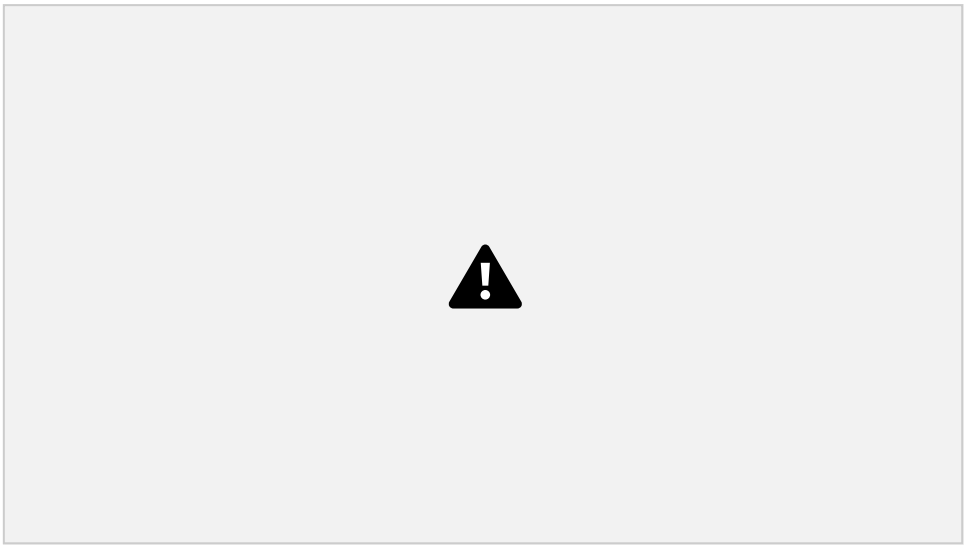
ACTIONS

Intro to actions with turtlesim

Actions are one of the communication types in ROS 2 and are intended for **long running tasks**. They consist of three parts: a **goal**, **feedback**, and a **result**.

Actions are built on topics and services. Their functionality is similar to services, except actions are preemptable (you can cancel them while executing). They also provide steady feedback, as opposed to services which return a single response.

Actions use a client-server model, similar to the publisher-subscriber model (described in the topics tutorial). An "**action client**" node sends a goal to an "**action server**" node that acknowledges the goal and returns a stream of feedback and a result.



Start by checking the ros2 action command (with **--help**)

```
ade:~$ ros2 action -h
usage: ros2 action [-h]
```

```
Call `ros2 action <command> -h` for more detailed usage.
Various action related sub-commands
optional arguments:
  -h, --help            show this help message and exit

Commands:
  info       Print information about an action
  list       Output a list of action names
  send_goal  Send an action goal

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

List actions in the system:

```
ade:~$ ros2 action list
/turtle1/rotate_absolute
```

It seems there is an action that node /turtle1 supports. Find more about it by invoking the info command with the –t option.

Find more about that message type and invoke the send goal command to trigger the action.

Follow the syntax: **ros2 action send_goal <action_name> <action_type> <values>**
<values> need to be in **YAML** format. You can also use the –f switch to get feedback while the action is running. The turtle will rotate, and the terminal output will be something like

```
Waiting for an action server to become available...
Sending goal:
     theta: 1.8
Goal accepted with ID: a05b5f7f99374d0291ac79e5f360cd3c
Feedback:
    remaining: -0.008000850677490234
Result:
    delta: 0.0
Goal finished with status: SUCCEEDED
```

Observe that there is a goal, a feedback and a result!

C++ Action (the Fibonacci sequence example)

Steps:

Define the action, test.

Write the action server, test.

Write the action client, test.

Defining an action

Actions are defined in .action files of the form:

```
# Request
---
# Result
---
# Feedback
```

An action definition is made up of three message definitions separated by ---.

- A *request* message is sent from an action client to an action server initiating a new goal.
- A *result* message is sent from an action server to an action client when a goal is done.
- *Feedback* messages are periodically sent from an action server to an action client with updates about a goal.

An instance of an action is typically referred to as a *goal*.

Step 1 - Define the action, test.

Say we want to define a new action "Fibonacci" for computing the Fibonacci sequence.

Create an action directory in our ROS 2 package tutorial_interfaces, from the last LAB class.

Inside the action directory, create a file called Fibonacci.action with the following contents:

```
int32 order
---
int32[] sequence
---
int32[] partial_sequence
```

The goal request is the order of the Fibonacci sequence we want to compute, the result is the final sequence, and the feedback is the partial_sequence computed so far.

As with services, before we start working on the code, we must setup the action definition through the **rosidl code generation pipeline**. This is accomplished by adding the following lines to our **CMakeLists.txt** before the **ament_package()** line, in the **tutorial_interfaces**:

```
find_package(rosidl_default_generators REQUIRED)
rosidl_generate_interfaces(${PROJECT_NAME}
#add to whatever interfaces you have declared before (probably RectangleArea.srv)
 "action/Fibonacci.action"
)
```

We should also add the required dependencies to our package.xml:

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<depend>action_msgs</depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

we need to depend on **action_msgs** since action definitions include additional metadata (e.g. goal IDs). We should now be able to build the package containing the Fibonacci action definition:

```
cd ~/ros2_iacos_ws
colcon build
```

By convention, action types will be prefixed by their package name and the word action. When we want to refer to our new action, it will have the full name:

```
tutorial_interfaces/action/Fibonacci
```

We can check that our action built successfully with the command line tool:

```
ade:~$ ros2 interface show tutorial_interfaces/action/Fibonacci
```

Step 2 - Write the action server, test.

Create the package that will host the action server. We can go ahead and re-use the package created in the last LAB session.

If you re-use the old package (**RECOMMENDED**) you need to add the above-mentioned dependencies to packages.xml.

```
<depend>rclcpp</depend>
<depend>example_interfaces</depend>
<depend>tutorial_interfaces</depend>
<depend>rclcpp_action</depend>
```

In **CMakeLists.txt** add the respective **find_package()** instructions for these dependencies. Add executable:

```
add_executable(fibonacci_action_server src/fibonacci_action_server.cpp)
ament_target_dependencies(fibonacci_action_server
"rclcpp"
"rclcpp_action"
"tutorial_interfaces")
```

And add **fibonacci_action_server** to the install targets at the bottom.

Create a new file **fibonacci_action_server.cpp** and paste inside:

```
#include <functional>
#include <memory>
#include <thread>
#include "tutorial_interfaces/action/fibonacci.hpp"
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"
#include "rclcpp_components/register_node_macro.hpp"

using Fibonacci = tutorial_interfaces::action::Fibonacci;
using GoalHandleFibonacci = rclcpp_action::ServerGoalHandle<Fibonacci>;

rclcpp_action::GoalResponse handle_goal(
const rclcpp_action::GoalUUID & uuid, std::shared_ptr<const Fibonacci::Goal> goal)
{
RCLCPP_INFO(rclcpp::get_logger("server"), "Got goal request with order %d", goal->order);
(void)uuid;
// Let's reject sequences that are over 9000
if (goal->order > 9000) {
return rclcpp_action::GoalResponse::REJECT;
}
return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
}

rclcpp_action::CancelResponse handle_cancel(
const std::shared_ptr<GoalHandleFibonacci> goal_handle)
{
RCLCPP_INFO(rclcpp::get_logger("server"), "Got request to cancel goal");
(void)goal_handle;
return rclcpp_action::CancelResponse::ACCEPT;
}

void execute(
const std::shared_ptr<GoalHandleFibonacci> goal_handle)
{
RCLCPP_INFO(rclcpp::get_logger("server"), "Executing goal");
rclcpp::Rate loop_rate(1);
const auto goal = goal_handle->get_goal();
auto feedback = std::make_shared<Fibonacci::Feedback>();
auto & sequence = feedback->partial_sequence;
sequence.push_back(0);
```

```
sequence.push_back(1);
auto result = std::make_shared<Fibonacci::Result>();

for (int i = 1; (i < goal->order) && rclcpp::ok(); ++i) {
// Check if there is a cancel request
if (goal_handle->is_canceling()) {
result->sequence = sequence;
goal_handle->canceled(result);
RCLCPP_INFO(rclcpp::get_logger("server"), "Goal Canceled");
return;
}
// Update sequence
sequence.push_back(sequence[i] + sequence[i - 1]);
// Publish feedback
goal_handle->publish_feedback(feedback);
RCLCPP_INFO(rclcpp::get_logger("server"), "Publish Feedback");

loop_rate.sleep();
}

// Check if goal is done
if (rclcpp::ok()) {
result->sequence = sequence;
goal_handle->succeed(result);
RCLCPP_INFO(rclcpp::get_logger("server"), "Goal Succeeded");
}
}

void handle_accepted(const std::shared_ptr<GoalHandleFibonacci> goal_handle)
{
// this needs to return quickly to avoid blocking the executor, so spin up a new thread
std::thread{execute, goal_handle}.detach();
}

int main(int argc, char ** argv)
{
rclcpp::init(argc, argv);
auto action_server_node = rclcpp::Node::make_shared("fibonacci_action_server");

// Create an action server with three callbacks
// 'handle_goal' and 'handle_cancel' are called by the Executor (rclcpp::spin)
// 'execute' is called whenever 'handle_goal' returns by accepting a goal
// Calls to 'execute' are made in an available thread from a pool of four.
auto action_server = rclcpp_action::create_server<Fibonacci>(
action_server_node,
"fibonacci",
handle_goal,
handle_cancel,
handle_accepted);
rclcpp::spin(action_server_node);
rclcpp::shutdown();
return 0;
}
```

Compile and run. (Yes... if it can’t find the executable, well.. You know. You must source your workspace!)

STEP3 - Create the action client

Add a new executable to **CMakeLists.txt** for the **fibonacci_action_client,** similarly to the previous step. Also add the executable at the bottom.

Create the **fibonacci_action_client.cpp** and paste the following code:

```
-----------------------------
#include <inttypes.h>
#include <memory>
#include "tutorial_interfaces/action/fibonacci.hpp"
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"

using Fibonacci = tutorial_interfaces::action::Fibonacci;
rclcpp::Node::SharedPtr node = nullptr;

void feedback_callback(
rclcpp_action::ClientGoalHandle<Fibonacci>::SharedPtr,
const std::shared_ptr<const Fibonacci::Feedback> feedback)
{
RCLCPP_INFO(
node->get_logger(),
"Next number in sequence received: %" PRId32,
feedback->partial_sequence.back());
}

int main(int argc, char ** argv)
{
rclcpp::init(argc, argv);
node = rclcpp::Node::make_shared("fibonacci_action_client");
auto action_client = rclcpp_action::create_client<Fibonacci>(node, "fibonacci");

if (!action_client->wait_for_action_server(std::chrono::seconds(20))) {
RCLCPP_ERROR(node->get_logger(), "Action server not available after waiting");
return 1;
}
// Populate a goal
auto goal_msg = Fibonacci::Goal();
goal_msg.order = 10;

RCLCPP_INFO(node->get_logger(), "Sending goal");
// Ask server to achieve some goal and wait until it's accepted
auto send_goal_options = rclcpp_action::Client<Fibonacci>::SendGoalOptions();
send_goal_options.feedback_callback = feedback_callback;
auto goal_handle_future = action_client->async_send_goal(goal_msg,send_goal_options);
if (rclcpp::spin_until_future_complete(node, goal_handle_future) !=
rclcpp::FutureReturnCode::SUCCESS)
{
RCLCPP_ERROR(node->get_logger(), "send goal call failed :(");
return 1;
}

rclcpp_action::ClientGoalHandle<Fibonacci>::SharedPtr             goal_handle             =
goal_handle_future.get();
if (!goal_handle) {
RCLCPP_ERROR(node->get_logger(), "Goal was rejected by server");
return 1;
}
```

```
// Wait for the server to be done with the goal
auto result_future = action_client->async_get_result(goal_handle);

RCLCPP_INFO(node->get_logger(), "Waiting for result");
if (rclcpp::spin_until_future_complete(node, result_future) !=
rclcpp::FutureReturnCode::SUCCESS)
{
RCLCPP_ERROR(node->get_logger(), "get result call failed :(");
return 1;
}

rclcpp_action::ClientGoalHandle<Fibonacci>::WrappedResult          wrapped_result          =
result_future.get();

switch (wrapped_result.code) {
case rclcpp_action::ResultCode::SUCCEEDED:
break;
case rclcpp_action::ResultCode::ABORTED:
RCLCPP_ERROR(node->get_logger(), "Goal was aborted");
return 1;
case rclcpp_action::ResultCode::CANCELED:
RCLCPP_ERROR(node->get_logger(), "Goal was canceled");
return 1;
default:
RCLCPP_ERROR(node->get_logger(), "Unknown result code");
return 1;
}

RCLCPP_INFO(node->get_logger(), "result received");
for (auto number : wrapped_result.result->sequence) {
RCLCPP_INFO(node->get_logger(), "%" PRId32, number);
}

rclcpp::shutdown();
return 0;
}
```

Compile it and run both client and server in separate terminals. See the results.

Analyze the code

It’s important that you take some time to understand the code.

By looking into the action server code, start by noticing the code structure:

As usual we start by creating a node fibonacci_action_server, and we follow by creating the actual action server. Notice what the class receives. An action server requires 6 things:

- The templated action type name: Fibonacci.
- A ROS 2 node to add the action to: this.
- The action name: 'fibonacci'.
- A callback function for handling goals: handle_goal
- A callback function for handling cancellation: handle_cancel.
- A callback function for handling goal accept: handle_accept.

See how these are handled in the code, and the correspondent client calls. Try to understand the data flow.

ROS Bag

ROS Bags are ROS's tool for recording, and replaying data, similarly to log files that let you store data along with messages. ROS systems can generate a lot of data, so you select which topics you want to bag. These are a great tool for testing and debugging your application as well.

Run the turtlesim executable, if it is not already running, and in a new terminal run the turtle_teleop_key. This one enables you to control the turtle via the keyboard.

In a third terminal, run:

```
ade:~$ ros2 bag record /turtle1/pose -o turtle
```

Move back to the terminal running the teleop executable and move the turtle around. After some time, stop the recording by doing CTRL+C.

Inspect the bag by running the command bellow. Notice the details.

```
ade:~$ ros2 bag info turtle
Files:             turtle_0.db3
Bag size:          128.8 KiB
Storage id:        sqlite3
Duration:          21.391s
Start:             Oct 20 2022 10:55:31.924 (1666259731.924)
End:               Oct 20 2022 10:55:53.316 (1666259753.316)
Messages:          1338
Topic information: Topic: /turtle1/pose | Type: turtlesim/msg/Pose | Count: 1338 |
Serialization Format: cdr
```

Let's now play the bag. First, close the turtlesim executable and all other nodes you may be running. Run a list of topics to make sure no topic is running in your system.

```
ade:~$ ros2 topic list
/parameter_events
/rosout
```

Play the bag. Run:

```
ade:~$ ros2 bag play turtle
Opened database 'turtle/turtle_0.db3' for READ_ONLY.
```

You don't see much, but a lot is happening. In another terminal run ros2 topic list and see a new topic has appeared (**/turtle1/pose**). Run an echo of that topic and see the pose values changing.

Credits

Version 1.1, Ricardo Severino, based on:

Autoware Training

ROS2 Tutorials