Lab02 - 02/11/2023

Understanding ROS 2: services, actions, parameters and launch files

*This lab activity contains some **Advanced Tasks** which are suggested for students who have a good knowledge of Python programming language. It is not mandatory to include these Tasks in your final report.*

Objectives

The aim of this lab consists of understanding the basic behaviour of ROS 2 services, actions, parameters, and launch files.
The main objectives can be summarised as:

- Getting familiar with services and actions
- Implementing custom interfaces package
- Writing a service
- (Advanced) Writing an action server/client
- Getting familiar with parameters and parameter files
- Using launch files to run multiple nodes

Tutorial

In this part of the lab, you will have to follow **all the steps** to get familiar with all the tools needed for the final exercise and the following activities.

For further insights, you should study the official ROS 2 Humble documentation (https://docs.ros.org/en/humble/index.html).

A. Requirements

Before proceeding with the lab, it's essential to have a solid understanding of the fundamentals of services, actions, parameters, and launch files in ROS 2. To ensure you're well-prepared, we recommend completing the following tutorials:

- Basic knowledge of services (here).
- Basic knowledge of parameters (here).
- Basic knowledge of actions (here).
- Basic knowledge of launch files (here).

B. Writing a simple service and 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 server node. A service is defined inside a file with `.srv` extension and has the following structure:

```
# Request
---
# Response
```

To create a simple server/client package, follow the ROS 2 guide here following all the steps.

C. Implementing custom interfaces

In ROS 2, when creating communication components, such as topics or services, you must declare the type of data you are exchanging, such as topic messages or service types. Typically, standard data types are readily available, and you won't need to create them from scratch.
However, you can create your custom interface package if your applications needs one.
Follow this tutorial to understand the creation of an interface package and its structure; notice that, for this case, you will implement a C++ package, but no C++ programming is required for this sake.
For a more detailed tutorial, you can read here.

D. Creating an action

Actions are defined in action files (`.action`) as follows:

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

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

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

The action message can be inserted in an interface package that you've seen previously. To see how to create an action and insert it in an interface package, read here.

E. Writing an action server and client

Follow this ROS 2 guide to write an action server and client. You should analyze the code and understand each passage.

F. Launch files

Until now, you have opened a new terminal for every node you run. Since you run at most two or three nodes simultaneously, it probably wasn't such a big deal, but if the system

becomes larger and mode nodes run together, it could become unmanageable. Launch files allow you to simultaneously start up and configure several executables containing ROS 2 nodes. Hence, the system will start by just executing the `ros2 launch` command.
You can find here the guide about creating a launch file (always consider the Python version).
You can integrate the launch files in your ROS 2 packages: follow the guide here.

G. Parameters

The objective of the usage of parameters in ROS 2 is the possibility of running a node with different configurations without directly changing the node's code but, rather, writing them in a configuration file.
Follow this tutorial to learn how to declare a parameter and assign it a default value directly in a node class. Using this case, if a configuration file is not used or parameters are not defined in a launch file, they are automatically set to a default value.
Usually, the best option consists in using a configuration file (e.g.:`configuration.yaml`) in a `config` folder located in your package. Then, when you launch the node(s) using a launch file, you can assign a value to the node's parameter by specifying the configuration file you want to use. You can find how to set parameters in a launch file and how to load parameters from a YAML file at point 2 of this guide.

Exercise

The objective of the exercise is to control the turtlesim to navigate through a set of known poses. Once a new goal is generated, the turtle will perform the following movements:

1. Rotate towards the goal direction
2. Move along its x-axis until the goal is reached
3. Rotate towards the desired orientation

Task 1

Create a package called `lab02_pkg`, which will contain the code for this lab.
Create another package called `lab02_interfaces`. This package will contain the necessary custom interfaces (message (`.msg`), service (`.srv`), and action (`.action`) files), which you will need during this lab.

Task 2

Implement a service node called `/turtle1/compute_trajectory`, which, given the actual pose and a goal pose, calculates the orientation of the robot towards the goal and the distance to travel. Try to call the service using the command line. The node should subscribe to the position topic of the turtle to obtain the current position, therefore, in the service message, you should only send the goal pose. Use the service message structure you find below.

Hint: try to explore the arguments of the constructor of the node to assign the namespace 'turtle1'.

```
# Request
float64 x
float64 y
---
# Response
float64 distance 0.0
float64 direction 0.0
```

Task 3

Once you evaluate the rotation towards the goal, call from the terminal the `/turtle1/rotate_absolute` action, which rotates the turtle in place to the desired angle. Try to rotate the turtle with different angles and verify its correct rotation.

(Advanced) Task 4

For this advanced task, you should use the template available on Portale della Didattica and write your code in the gaps.

Implement an action server similar to the one you used in Task 3, to make the turtle move forward for a given distance from its starting point.
The action should be called `/turtle1/move_distance`, and its definition is the following:

```
# Goal
float64 distance
---
# Result
float64 elapsed_time_s
float64 traveled_distance
---
# Feedback
float64 remaining_distance
```

The action server should:

- Receive the distance to travel
- Move forward at a velocity of 0.5 m/s

The action node should publish the velocities command on the `/turtle1/cmd_vel` topic with a frequency of 10 Hz.
The movement should stop when the turtle travels the required distance. To do so, the action node should subscribe to the topic of the turtle's position (`/turtle1/pose`) and should stop when the turtle is 'near enough' to the goal position. Define a suitable threshold.

At the end of the movement, return the elapsed time and the actual travelled distance.
The feedback of the action should be published at 2 Hz and should contain the remaining movement to be done.

Hint: this node needs an updated position of the turtle while executing the action server callback. To ensure that, two different callback groups must be declared: a mutually exclusive callback group to manage the subscription and a reentrant callback group for the service callback. In this way, the two callbacks are processed in parallel, and the action server callback will not block the subscription callback. Read carefully this page before proceeding. To spin a node with multiple callback groups, a multi-thread executor is required, as shown here.

(Advanced) Task 5

Create a ROS 2 node named `/goal_generator` responsible for generating goal poses. These goal poses should have (x, y, θ) coordinates within the turtlesim workspace (x ∈ [0, 11], y ∈ [0, 11]). You can either pre-define a list of goals or generate them randomly.
Once you generate a goal, call the service */turtle1/compute_trajectory* and perform the resulting motion using the two actions */turtle1/move_distance* and */turtle1/rotate_absolute*.

Task 6

Write a launch file called `turtle.launch.py`, which starts the following nodes:

- Turtlesim node
- Turtle teleop key
- The service node

Hint: to open a dedicated terminal for the turtle_teleop_key, so you can input the commands, add the argument `prefix=["xterm -e"]` to the Node inside the launch file.

(Advanced) Task 6b

Add to the launch file the *compute trajectory server* and the *move distance action server*.

Task 7

Create the folder config and the file `parameters.yaml` inside it. Configure the parameters *scale_angular* and *scale_linear* inside the file you just created. Those parameters are referred to the `turtle_teleop_key` node and change the maximum rotation and translation velocity Then, modify the previous launch file to get the parameters from this file when launching the node.

(Advanced) Task 7b

Make the move distance action server configurable by adding a parameter to control the velocity and one to control the goal tolerance. Add the parameters to `parameters.yaml` and launch your node with the parameters contained in the configuration file.

Report

At the end of LAB03, you must write a comprehensive report of the first 3 lab activities.

Request for each task:

- Task 1: nothing
- Task 2: describe the implementation.
- Task 3: report the target angle and the reached angle. Make some examples. Comment on the obtained result.
- (Advanced) Task 4: describe the design of your action server.
- (Advance) Task 5: describe the implementation of the client node. Report the results obtained with some goals. (terminal logs or plots)
- Task 6/6b: describe the purpose of the launch file.
- Task 7: verify that the parameters are set inside the node.
- Task 8: list the parameters you introduced in your nodes and describe their purpose.

You should also deliver on the Portale della Didattica a zipped folder containing the packages that you developed, named:

```
sesars_lab<lab_number>_<group_number>.zip
```