Laboratory class #5: Path planning of mobile robots

1.- ROS Services

In comparison with ROS topics where we have one publisher and many subscribers, a ROS service is based on a request and response communication, so it is always two nodes communicating with each other. The node that provides the service is called server while the node that asks for it is called client.

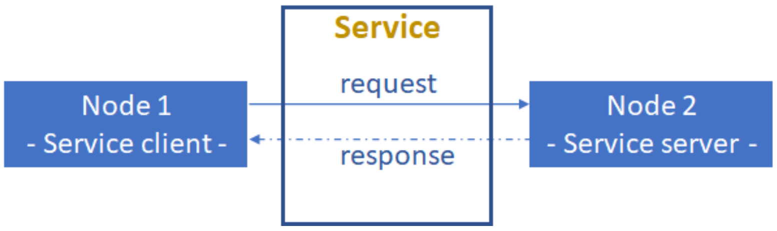

Figure 1. Example of two nodes communicating through a service

In Figure 1, the two nodes implement a communication using a service, where node 1 is the client and node 2 is the server. The service definition is done in .srv files which has the following format:

```
Request
---
Response
```

For example,

```
$ gedit /opt/ros/noetic/share/nav_msgs/srv/GetPlan.srv
```

Opens the following .srv file:

```
# Get a plan from the current position to the goal Pose
# The start pose for the plan
geometry_msgs/PoseStamped start
# The final pose of the goal position
geometry_msgs/PoseStamped goal
# If the goal is obstructed, how many meters the planner can
# relax the constraint in x and y before failing.
float32 tolerance
---
nav_msgs/Path plan
```

That can be used for a motion planner to get a global plan from a given start pose to a target pose.

Similar to ROS topics, the following commands may be used to get information about the current services:

```
$ rosservice list  # list all available services
$ rosservice type /service_name # show the type of service
$ rosservice call /service_name args # call a service with the request contents
```

Exercise 1.

Let us execute a ROS service available in the roscpp tutorial.

1. In the first console execute ros master with:

   ```
   $ roscore
   ```

2. In the second console, run a service demo node with

   ```
   $ rosrun roscpp_tutorials add_two_ints_server
   ```

3. In a new console, let us analyze and call a service

   ```
   $ rosservice list
   ```

   We can see the following service: `/add_two_ints` that adds two integer numbers.

4. In order to see the type of service

   ```
   $ rosservice type /add_two_ints
   ```

5. In order to see the service definition, we can use the following command

   ```
   $ rossrv show roscpp_tutorials/TwoInts
   ```

   Which returns the following information:

   ```
   int64 a

   int64 b

   ---

   int64 sum
   ```

   Hence, this service is called with two integers a and b and is returning their sum in the variable sum.

6. In order to call the service from console, we can execute

   ```
   $ rosservice call /add_two_ints "{a: 7, b: 8}"
   ```

   Will return

   ```
   sum: 15
   ```

Let us see the source file of the server service /add_two_ints that we executed before (https://github.com/ros/ros_tutorials/blob/noetic-devel/roscpp_tutorials/add_two_ints_server/add_two_ints_server.cpp):

```
#include "ros/ros.h"
#include "roscpp_tutorials/TwoInts.h"

bool add(roscpp_tutorials::TwoInts::Request  &req,
         roscpp_tutorials::TwoInts::Response &res )
{
  res.sum = req.a + req.b;
  ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
  ROS_INFO("  sending back response: [%ld]", (long int)res.sum);
  return true;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_server");
  ros::NodeHandle n;

// %Tag(SERVICE_SERVER)%
  ros::ServiceServer service = n.advertiseService("add_two_ints", add);
// %EndTag(SERVICE_SERVER)%

  ros::spin();

  return 0;
}
```

It can be observed that in order to create a service it is necessary to use the following command `ros::ServiceServer service = nh.advertiseService(service_name, callback_function);`. When a service request is received, the callback function is called with the request as argument, in the previous case the callback function is called `add`. The callback function implementing the service, returns in general a boolean that will be true if the service is correctly executed.

A possible client is given in the following c++ code avaible on Internet at https://github.com/ros/ros_tutorials/blob/noetic-devel/roscpp_tutorials/add_two_ints_client/add_two_ints_client.cpp:

```
#include "ros/ros.h"
#include "roscpp_tutorials/TwoInts.h"
#include <cstdlib>


int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_client");
  if (argc != 3)
  {
    ROS_INFO("usage: add_two_ints_client X Y");
    return 1;
  }

  ros::NodeHandle n;
  ros::ServiceClient client = n.serviceClient<roscpp_tutorials::TwoInts>("add_two_ints");
  roscpp_tutorials::TwoInts srv;
  srv.request.a = atoi(argv[1]);
  srv.request.b = atoi(argv[2]);
  if (client.call(srv))
  {
    ROS_INFO("Sum: %ld", (long int)srv.response.sum);
  }
  else
  {
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
  }

  return 0;
}
```

In order to create a service client, we can use `ros::ServiceClient client = nodeHandle.serviceClient<service_type>(service_name);` then the service request is created in `srv.request`. Finally, the service is called with `client.call(srv)`. In the same object used to send the request, the response will be available if the service has been correctly called. This response is available in `srv.response`.

2. ROS actions (actionlib)

Actions are similar to service calls, but can be used in the case of slow tasks since they provide the possibility to cancel the task (preempt) and also to receive feedback on the task progress.

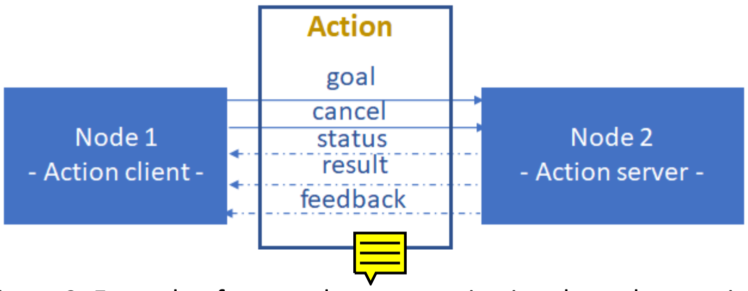

Figure 2. Example of two nodes communicating through an action

Figure 2 shows a simple structure of two nodes communicating through an action. Similar in structure to services, actions are defined in .action files, having the following format:

```
Goal
---
Result
---
Feedback
```

On the top, we have the `Goal`, followed by the `Result` and by the `Feedback` definition. As an example,

```
$ gedit
/opt/ros/noetic/share/actionlib_tutorials/action/Averaging.action
```

and the following .action file will be shown:

```
#goal definition
int32 samples
---
#result definition
float32 mean
float32 std_dev
---
#feedback
int32 sample
float32 data
float32 mean
float32 std_dev
```

This can be used to implement a simple average mechanism, where the goals are some samples and the result is the mean and the standard deviation. In the case in which the process is slow, it is possible to get feedback to know, for example, the samples that are already processed, the current mean and standard deviation.

3. ROS Parameter Server

A parameter server is accessible via network and it is used by the nodes to store and retrieve parameters during the execution. In general it is used for static constants such as configuration parameters that are globally viewable by any tool. Parameters can be defined in YAML files and can be loaded with `rosparam` command or within a launch file (http://wiki.ros.org/rosparam). The YAML files are put in general in the 'config' folder of the package folder. An example of a YAML file is the following:

```
text: "AROB lab 3"
number_int: 25
number_float: 11.7
enable_boolean: true
list_of_elements:
    - 1
    - 2
    - 3
dictionary: {
    another_text: "World",
    another_number: 3,
}
```

In order to load a yaml file, with ROS master running (roscore), use the following command:

```
$ rosparam load my_params.yaml
```

To load the YAML file with the launch file, the `<rosparam>` tag should be used followed by the name of the YAML file. As an example, the following launch file will load the config.yaml file from the config folder of the package together with the node, hence the parameters are available under the namespace of the node.

```
<launch>
<node name="name" pkg="package" type="node_type">
<rosparam command="load" file="$(find package)/config/config.yaml" />
</node>
</launch>
```

Once the previous YAML file is loaded, using the following command

```
$ rosparam list
```

Will return the list of all parameters, in particular will return

```
/dictionary/another_number
/dictionary/another_text
/enable_boolean
/list_of_elements
/number_float
/number_int
/rosdistro
/roslaunch/uris/host_osboxes__37835
/rosversion
/run_id
/text
```

With `rosparam get` we can check the value of a parameter while with `rosparam set` we can define the value of a parameter. For example

```
$ rosparam get /number_int
$ rosparam set /number_int 2
```

From your code, as seen in lab4, once the parameters are loaded from YAML into the ROS Parameter Server, the following lines can be used to load a parameter.

```
ros::NodeHandle nh("~");
double number_to_get;
nh.getParam("number_float", number_to_get);
nh.getParam("/custom_prefix/number_float", number_to_get);
```

Exercise 2.

Download arob_lab5 package from github https://github.com/luisriazuelo/arob_lab5.git and include it into your workspace. Complete the implementation of the RRT algorithm in file rrt_global_planner.cpp such that, when sending a goal through RViz or publishing a message on the topic /move_base_simple/goal, a path is computed and followed by the robot. Remember that this global planner should be registered as a plugin in ROS to be used in move_base. You can test the implementation with:

```
$ rostopic pub /move_base_simple/goal geometry_msgs/PoseStamped '{header: {stamp: now, frame_id: "map"}, pose: {position: {x: -3, y: 0, z: 0.0}, orientation: {w: 1}}}'
```

and then with,

```
$ rostopic pub /move_base_simple/goal geometry_msgs/PoseStamped '{header: {stamp: now, frame_id: "map"}, pose: {position: {x: 5, y: 5, z: 0.0}, orientation: {w: 1}}}'
```

Note: You can use **Markers** to display the tree nodes on RViz (http://wiki.ros.org/rviz/DisplayTypes/Marker).

Exercise 3.

Test again the system using also the plugin of your low level controller (implemented in lab4) as the local planner in move_base.

4. References

- ROS action tutorials: http://wiki.ros.org/actionlib_tutorials
- ROS noetic navigation: http://wiki.ros.org/navigation