Mapping and Exploration

In TurtleBot ROS2

By: Rakhimov Rustambek

The aim of this project is to create an action for Mapping and Exploration. Mapping and Exploration action involves a robot autonomously navigating and mapping its environment. It encompasses tasks such as simultaneous localization and mapping (SLAM) to create a map and explore an unknown area while avoiding obstacles.

Introduction

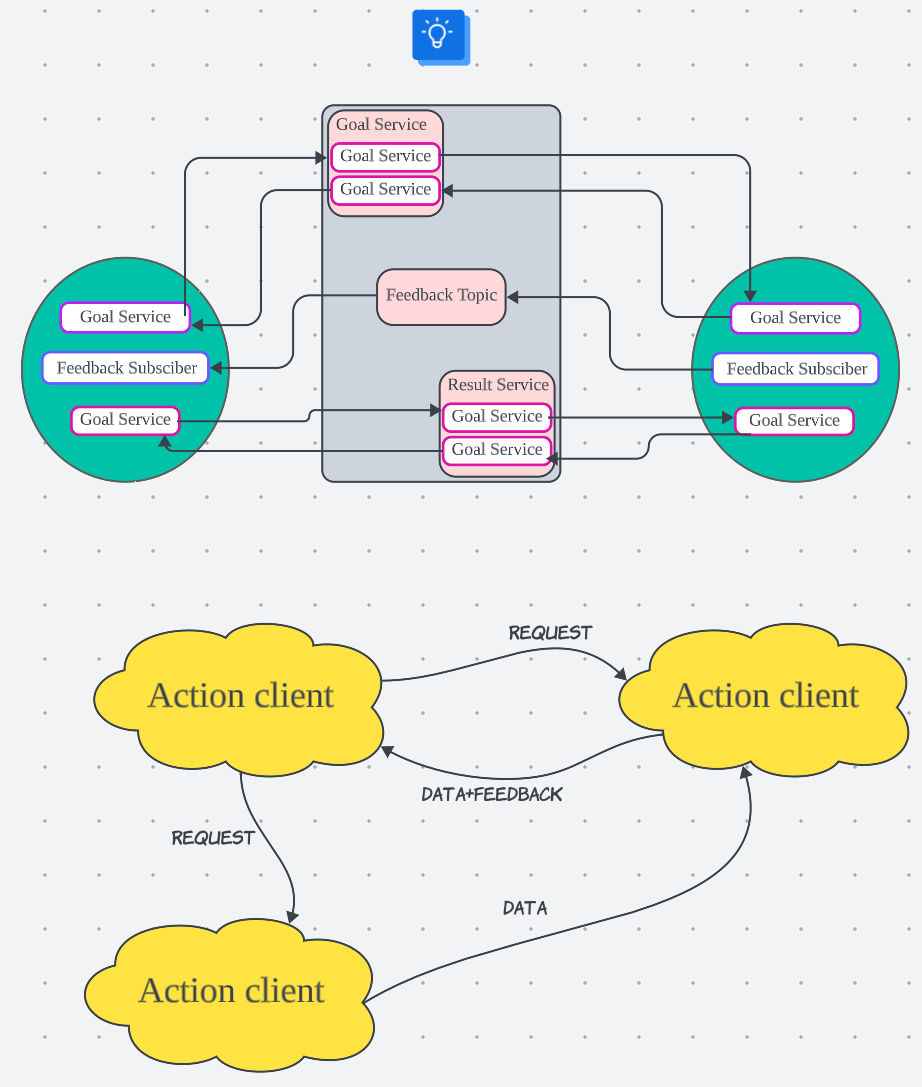
In this tutorial, we will guide you through the process of defining and building a Mapping and Exploration Action in ROS2. The Mapping and Exploration Action allows a robot, like the TurtleBot, to autonomously explore and map an unknown environment.

**Prerequisites**

Before you begin, ensure that you have the following prerequisites:

* A working ROS2 installation.
* A ROS2 package where you want to define the Mapping and Exploration Action.

Diagram of the action communication between client and server:



The diagram on the page above illustrates how the client communicates with the server as well as how the lidar is used. The chart also assumes feedback and topics are used along with the action. As you can see, Goal service, Feedback subscriber and Goal service exist, one of the 2 goal services sends data to start the action and the next is used to get the result. The feedback service is used to send feedback and make changes accordingly.

The Client side service sends a request though the topic and gets a response from the server.

Afterwards, the server sends feedback though the topics and then receives data on the results and sends back feedback. The chart below also demonstrates how the data from lidar is used to get and use data on the surrounding environment.

Step 1

* Create a package and directory for the action.
* I made a package and then a directory within it for my action.

Step 2

* Create a .action file within the repository.
* Add the file and configuration to CMakeLists.txt.
* Add the file and package dependencies to package.xml.

Step 3

* Create a client and server side python or C++ file to run the operations. I choose to use python as it has more dynamic libraries and is more familiar to me.

Step 4

* After sourcing and installing dependencies, build the package and run it.
* Connect to the server and turtlebot.

The files can be seen below:

Cmakelists:

A screenshot of a computer program

Description automatically generated

Package.xml

A screenshot of a computer program

Description automatically generated

.action file

A screenshot of a computer

Description automatically generated

Client-side Code:

#include "rclcpp/rclcpp.hpp"

#include "rclcpp\_action/rclcpp\_action.hpp"

#include "action\_tutorials\_interfaces/action/mapping\_and\_exploration.hpp"

using MappingAndExploration = action\_tutorials\_interfaces::action::MappingAndExploration;

namespace rclcpp = rclcpp;

class MappingAndExplorationClientNode : public rclcpp::Node

{

public:

explicit MappingAndExplorationClientNode()

: Node("mapping\_and\_exploration\_client")

{

action\_client\_ = rclcpp\_action::create\_client<MappingAndExploration>(

this, "mapping\_and\_exploration");

// Wait for the action server to become available

if (!action\_client\_->wait\_for\_action\_server(std::chrono::seconds(10)))

{

RCLCPP\_ERROR(get\_logger(), "Action server not available. Exiting.");

return;

}

send\_goal();

}

private:

rclcpp\_action::Client<MappingAndExploration>::SharedPtr action\_client\_;

void send\_goal()

{

auto goal = MappingAndExploration::Goal();

// Populate the goal message with the desired data

// For example, set the map and other parameters in the goal

RCLCPP\_INFO(get\_logger(), "Sending goal request");

auto send\_goal\_options = rclcpp\_action::Client<MappingAndExploration>::SendGoalOptions();

send\_goal\_options.goal\_response\_callback =

std::bind(&MappingAndExplorationClientNode::goal\_response\_callback, this, \_1);

send\_goal\_options.result\_callback =

std::bind(&MappingAndExplorationClientNode::result\_callback, this, \_1);

send\_goal\_options.feedback\_callback =

std::bind(&MappingAndExplorationClientNode::feedback\_callback, this, \_1, \_2);

action\_client\_->async\_send\_goal(goal, send\_goal\_options);

}

void goal\_response\_callback(std::shared\_future<rclcpp\_action::ClientGoalHandle<MappingAndExploration>::SharedPtr> future)

{

auto goal\_handle = future.get();

if (!goal\_handle)

{

RCLCPP\_ERROR(get\_logger(), "Goal was rejected by the action server.");

return;

}

RCLCPP\_INFO(get\_logger(), "Goal accepted by the action server.");

}

void result\_callback(const rclcpp\_action::ClientGoalHandle<MappingAndExploration>::WrappedResult &result)

{

switch (result.code)

{

case rclcpp\_action::ResultCode::SUCCEEDED:

RCLCPP\_INFO(get\_logger(), "Action succeeded");

break;

default:

RCLCPP\_ERROR(get\_logger(), "Action failed with result code %d", static\_cast<int>(result.code));

break;

}

}

void feedback\_callback(

const rclcpp\_action::ClientGoalHandle<MappingAndExploration>::SharedPtr,

const std::shared\_ptr<const MappingAndExploration::Feedback> feedback)

{

// Handle feedback from the action server

// Feedback typically contains information about the action's progress

// You can process and display this feedback as needed

}

};

int main(int argc, char \*\*argv)

{

rclcpp::init(argc, argv);

rclcpp::spin(std::make\_shared<MappingAndExplorationClientNode>());

rclcpp::shutdown();

return 0;

}

Server-side code

A screenshot of a computer

Description automatically generated