Assignment 2 Report

Group 8
Javier Alquézar Alquézar
Alejandro Cano Caldero
Jesús Moncada Ramírez

Introduction

As we were asked to do in this assignment, we have modularized our project, so we have divided our code into **three** nodes (*robot_controller_node, object_detections_node,* and *object_manipulation_node*) that will be explained separately later on. We have developed a **fourth** node (*pick_place_node*) but it is only in charge of testing the robot's arm and gripper, it must not be evaluated as it is not part of the final solution. First, we will comment on changes made in the code imported from assignment 1.

Changes in assignment 1

These changes have been made in order to adapt to the requirements of assignment 2; they are the following:

- The client (*src/client.cpp*) also asks by command line for an **orientation** to move the robot. It must be provided in degrees with respect to the *z-axis* of our robot, counter-clockwise.
- The server (*src/server.cpp*) allows moving the robot **without performing obstacle detection**. This has been done by adding a boolean attribute (*obstacleDetection*) to the goal definition of the action *action/TiagoController.action*.

Robot controller node

The **robot controller** node (*src/robot_controller.cpp*) is the central node of our project because it is in charge of performing all the operations we are asked to do by using the other two nodes.

First of all, it calls the *human_objects_srv* to retrieve the order in which the objects must be moved. After getting this information, it will execute the function *doObject* sequentially for every object id.

Firstly, this function **(1)** moves the robot to an initial position where all the destinations are reachable. Note that the ROS 2D navigation stack doesn't work properly in narrow corridors, so we had to define intermediate waypoints. After this, the node **(2)** moves the robot to the pick table, in a position where the object is easy to catch and **(3)** moves the robot's torso and head to have a wider vision of the table and its obstacles. We discovered that elevating the torso it is possible to detect more collision objects on the table. The next step is **(4)** to call the obstacle detection node to detect the objects in the table and **(5)** use the object manipulation node to catch one object among the others. Finally, **(6)** the robot is moved to the corridor, **(7)** moved to the correct drop table and then **(8)** the object is dropped. After that, the robot **(9)** returns to the initial position where it started.

In order to move the robot we have used the *assignment_1*. We are using an action server but we are not controlling its feedback to simplify the code.

Object detections node

The **object detection** node (*src/object_detections.cpp*) creates a ROS service called *object_detections*. This service must be called with an empty **request** (only including a header) and will return a **response** containing all the objects detected using the robot's camera and their poses. To have a compact data structure able to store both the id and the pose of a movable object we have defined a ROS message (*ObjectWithTag.msg*).

When this server receives a request, the following operations will: **(1)** get a message from the topic *tag_detections*, containing the AprilTag detections using the camera and its poses, **(2)** save all this information using *ObjectWithTag* messages, **(3)** transform all the poses from the camera frame (*xtion_rgb_optical_frame*) to the robot frame (*base_footprint*) using the TF2 library, **(4)** and return everything to the client.

Object manipulation node

The **object manipulation** node (*src/object_manipulation.cpp*) creates a ROS action called *object_manipulation*. When called, we will either catch or drop an object from the table. The action **goal** contains the following parameters: *mode* (0 if catching operation, 1 if dropping), *catch_id* (id of the object to catch/drop), and *objects* (all the objects detected in the table by the object detection). The **feedback** messages will contain a *mode* identifying the last operation performed, and the **result** has an *ok* variable that tells the client if the operation was completed correctly.

Note that this action server must be called in a **responsible** way, which means that, if it is called to catch an object, it will perform the operation, even if the robot already had an object on its gripper. Likewise, it will make the movement of dropping an object into the table, even if there were not any.

When called to catch an object, this action server will **(1)** add to the planning scene all the immovable collision objects (function *addImmovableCollisionObjects*). Their poses never change, so they have been written in our code; the positions were obtained by using RVIZ and the robot laser scanner, and their dimensions were obtained by using Gazebo and analyzing the *.world* files. Before being added to the planning scene, they are transformed from the *map* frame to the robot frame. After this, **(2)** the movable collision objects, all received in the goal, are also added. Once we have our planning scene ready, we can **(3)** open the gripper, **(4)** move the arm to a target position centimeters above the object, **(5)** move the arm to the object pose, **(6)** remove the collision object created for the object and **(7)** attach the object to the gripper by using the package *gazebo_ros_link_attacher*. This simple package defines two ROS services, one for attaching objects and one for detaching; they are implemented with our function *attachDetachObject.* After this, **(8)** the gripper is closed, **(9)** the arm is moved again to the target position, and finally to a **(10)** secure position. To clean the planning scene we **(11)** remove all the collision objects. In the target and catching pose the arm is moved several centimeters above the object's original pose, both distances depend on the object that is being caught and its dimensions.

When called to drop an object, the operations are the following: **(1)** adding the immovable collision objects, **(2)** moving the arm in front of the robot close to the height of the drop tables, **(3)** opening the gripper, **(4)** detaching the object from the gripper using the Gazebo plugging and **(5)** moving the arm to a secure pose. Like in the previous sequence, **(6)** all the collision objects are removed at the end of the process.