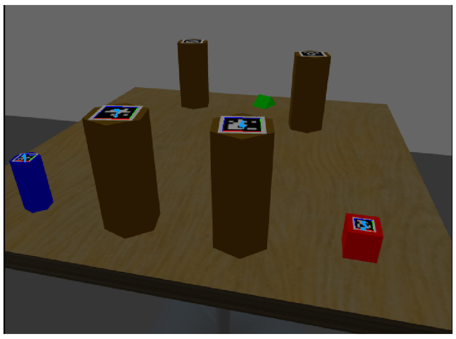
Edoardo Bastianello: 2053077
Stefano Binotto: 2052421
Gionata Grotto: 2052418
Note: the length of this report slightly exceeds the max length since we put a lot of images.

ASSIGNMENT 2 REPORT

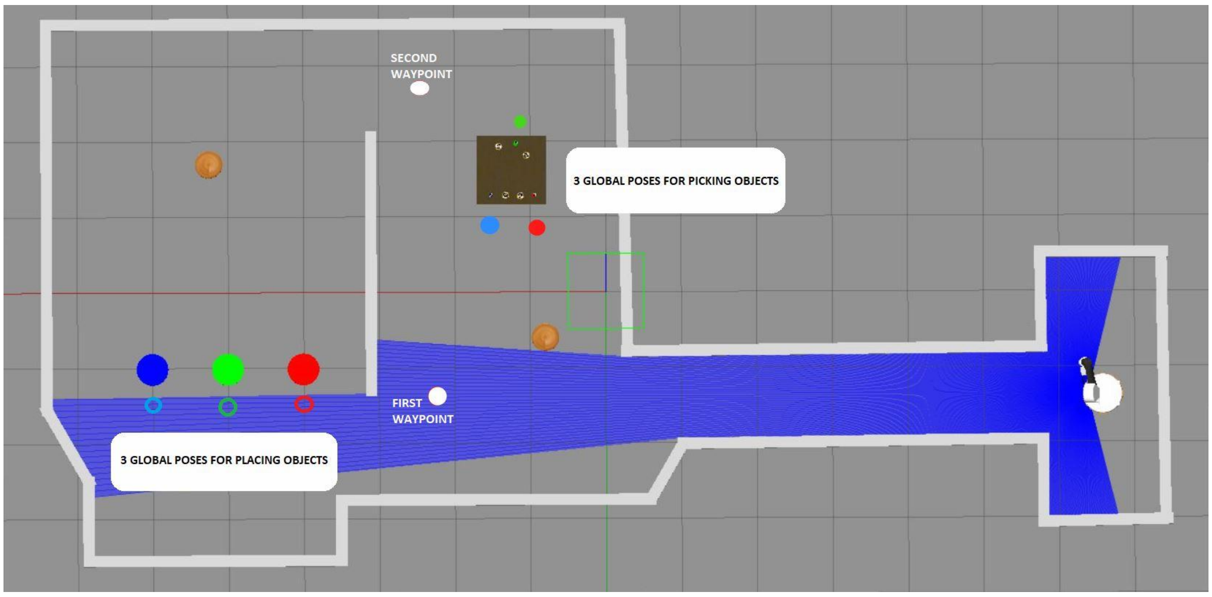
CLIENT - “client_human_navigate.cpp”:



The first thing our client does is lowering the head in order to be able to detect all the needed obstacles and objects to pick. The correct angle of the head to grant a proper vision has been set -0.50 radiants. To set this value we tested the detection several times with rviz for different poses around the table (*Figure on the right*: camera view from the position we set to pick the red cube). Then the client calls *Human_node* in order to get the order of the objects to pick. Next, for each object to pick, the client manages the following tasks:

- NAVIGATION TO THE TABLE;
- DETECTIONS OF THE OBJECTS AND THE OBSTACLES ON THE TABLE;
- PICK THE OBJECT;
- NAVIGATION TO THE CORRECT CYLINDRICAL TABLE;
- PLACE THE OBJECT ON THE CYLINDRICAL TABLE

NAVIGATION:


SECOND WAYPOINT
3 GLOBAL POSES FOR PICKING OBJECTS
FIRST WAYPOINT
3 GLOBAL POSES FOR PLACING OBJECTS

The navigation is managed by calling the server implemented in the *robot_srv.cpp* file. We used a few waypoints to correctly navigate the map. In particular:

- First waypoint: we set this waypoint to facilitate the navigation from the table to the cylindrical tables and vice versa. In addition, this waypoint was also useful to avoid the robot getting stuck between the wall on the right and the cylindrical orange obstacle.
- Second waypoint: we set this waypoint because the column of the table is much smaller than its surface, so the robot couldn’t detect it and consequently it kept colliding with it. In particular, this waypoint was only used to reach the pose to pick the green object.

Then, we also set three global poses around the table in order to simplify the detection and the picking of the objects (colored dots in the figure above). Also, we set three additional global poses in

front of the cylindrical tables in order to simplify the placing of the objects (colored circumferences in the figure above).
Depending on the object to be picked, the sequence of waypoints and global positions that the robot follows is one of the two reported below:

- If the object to pick is the red cube or the blue hexagon: first waypoint → red/blue global position for picking → red/blue global position for placing
- Instead, if the object to pick is the green triangle: first waypoint → second waypoint → green global position for picking → second waypoint → first waypoint → green global position for placing

DETECTION:

The detection is managed by calling the server implemented in the *detection_server.cpp* file. The server subscribes to the tag_detections topic and then converts the resulting poses of the detected tags from the camera frame */xtion_rgb_optical-frame* to the robot frame */base_footprint,* since this is the same frame we used to deal with the moveit framework. We chose the global poses around the table that maximize the number of detected tags in order to give the robot the maximum amount of information about its surroundings. In this way, it was possible to avoid the collisions with all the detected objects during the pick maneuver.

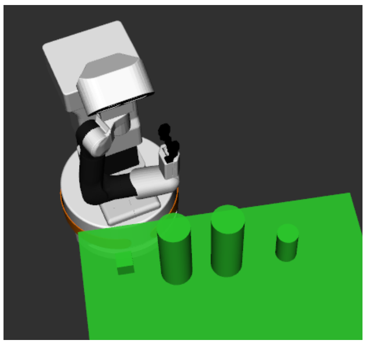


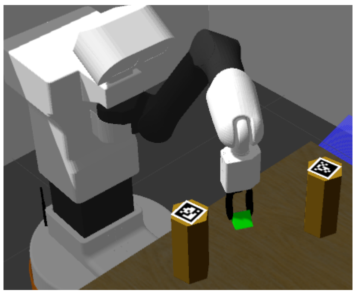
Figure: *collision objects created from the global pose to pick the red cube. Notice how, along with the red cube, the robot also detected the two adjacent hexagonal obstacles and the blue hexagon. In this way, during the cube picking procedure, the probability of collisions was minimized.*

For the making of the collision objects, we set the objects in this way: the green triangle as a cone, the blue hexagon and all the obstacles as cylinders and the red cube and the table as boxes. For all of them, we had to increase a little bit all their dimensions, especially for the hexagonal obstacles in order to prevent the arm from colliding with them during the pick phase.

PICK:

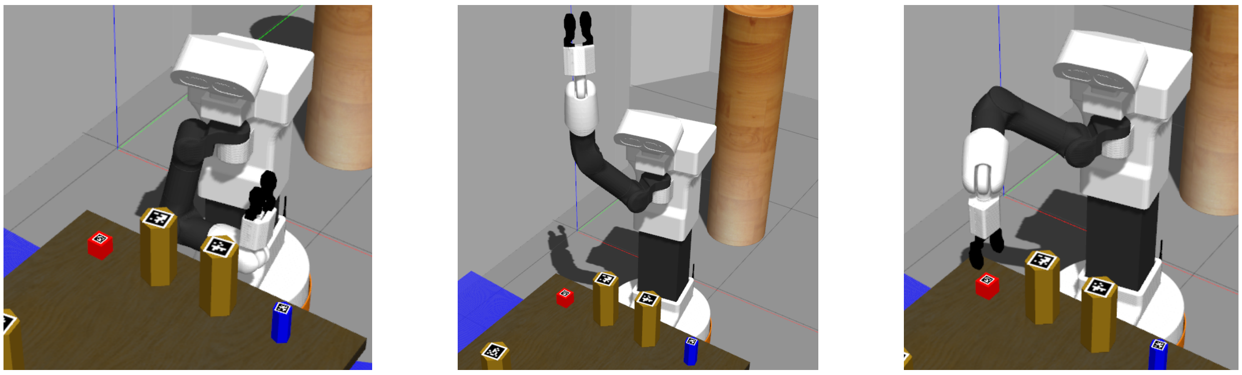
The pick phase is managed by calling the server implemented in the *pick_server.cpp* file.
In order to manage the picking of the objects we set 3 different configurations of the robot arm:

- **safe:** this configuration was specified in the joint space in order to reach every time the same pose for every joint. This pose was set in order to achieve a safe navigation (and for this it has to occupy the least amount of space possible) while at the same time providing enough room for the object that the robot is holding.
- **intermediate:** this was specified in the joint space as well for the same reasons as the safe configuration. We chose this configuration in order to minimize the risk of collisions with the table and the objects. For this reason, we raised both the torso and the arm of the robot. In this way, we were able to minimize the amount of horizontal movements in the proximity of the obstacles.
- **target:** this configuration instead is specified in the cartesian space, indeed the goal of this configuration is to reach a <u>position for the gripper</u> slightly above the object to pick (we set the position above the blue hexagon slightly higher than the cube and the triangle since the hexagon is higher). Since from the detection we're able to get the pose of the object, we can use the cartesian space to directly set a target pose for the gripper instead of setting the value of each joint. As for the <u>orientation of the gripper</u>, instead, we set it perpendicular to the plane of the table while keeping in consideration also the



orientation of the object to pick (see how, in the figure above, the gripper adapts to the orientation of the green triangle).

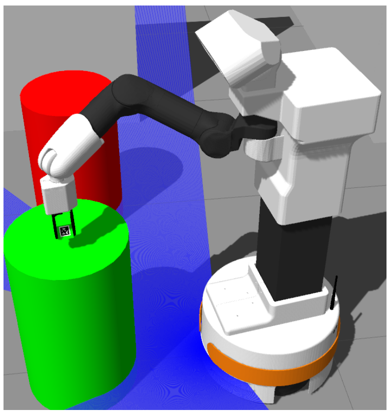
To reach the pose above the object, we used this sequence of configurations:
safe → intermediate → target



Figures (from left to right): *safe configuration, intermediate configuration, target configuration*

At this point, we performed a cartesian vertical path to have the object between the two pincers (see figure on the right). This sequence of operations was executed to ensure a safe pick procedure.
Then, we removed the object from the collision objects, we attached it to the link *arm_7_link* of the robot using the *gazebo_ros_link_attacher* plugin and, finally, we closed the gripper.
After doing that, we executed the following sequence of configurations:
configuration obtained after cartesian path → target→ intermediate → safe

PLACE:



The place phase is managed by calling the server implemented in the *place_server.cpp* file.
We got the dimensions and the positions of the three cylindrical tables directly from gazebo in order to create their respective collision objects, with incremented dimensions. After that, in order to place the object, we still used the same *safe* and *intermediate* configurations that we used during the pick procedure. Then, we also used a new version of the *target* configuration (see figure on the right), that, instead of positioning the gripper above the object, it positions the gripper above the center of the cylindrical table (the gripper is still oriented vertically). So, to reach the pose of the gripper from which we released the object, we followed this sequence of configurations:
safe → intermediate → new_target
Next, we opened the gripper and then we used the gazebo_ros_link_attacher plugin to detach the object.

In the end, we followed this sequence of configurations to return to the safe configuration:
new_target → intermediate → safe

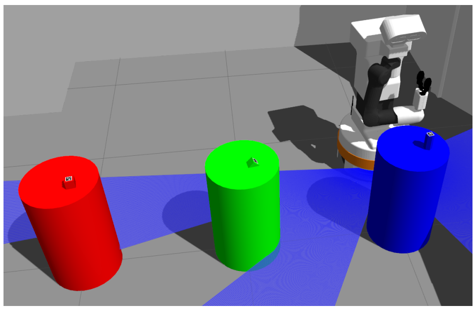


Figure*: final result with each object placed correctly on the right cylindrical table.*

Note 1: we released the object from slightly above the surface of the cylindrical table in order to ensure safety and avoid possible collisions with the table (especially since the robot is holding an object that sticks out past the robot's geometry).
Note 2: we released the hexagon from a slightly higher position than the cube and the triangle to ensure safety since the hexagon is higher.

ASSUMPTIONS AND PARAMETERS TUNING:

- We assumed that the poses of the table and of each one of the cylindrical tables are known in advance.
- To enlarge the collision objects of the obstacles and the objects on the table, we did several tests and we found out that we had to enlarge the collision objects of the table and the cylindrical tables by a considerable amount, while the collision objects of the obstacles and the objects on the table needed a much smaller increment. In addition, we found out that increasing the size of the width/radius of the collision objects was much more effective than increasing the size of the height.
- We assumed that the dimensions of the objects on the table were known in advance. To get the values of their dimensions we used gazebo.

OTHER CONSIDERATIONS:

- Conversion from origin of gazebo to map frame: since the poses of the table and of the cylindrical tables in gazebo are expressed with respect to the origin of gazebo, we had to manually convert those coordinates into coordinates expressed with respect to the *map* frame (and then we converted them into coordinates expressed with respect to the *base_footprint* frame using tf2 to pick/place the object). However, after converting the coordinates from the gazebo origin to the *map* frame, we found out that they were slightly shifted with respect to the x axis. For this reason, we corrected them by compensating the shift by applying an offset equal to -0.005 towards the opposite side (convertOriginToMap function in utils.cpp).
- Multiple inverse kinematic solutions: since we used inverse kinematics to reach a pose to pick the object, different executions of the project can result in different trajectories. This is due to the fact that inverse kinematics doesn't have a unique solution. This is even more noticeable since the whole moveit framework is probabilistic. That’s why the outcome could change among different executions.

CODE AND COMMANDS TO RUN THE CODE:

The name of the files and the commands to run the code are reported in the README file in the BitBucket repository.