Software Architecture for Robotics

Assignment – R.E. 2020/2021

Group: HRC-04
Students: Francesco Ganci (4143910) , Zoe Betta (5063114) , Lorenzo Causa (4519089), Federico Zecchi (4949035)
Tutor: Simone Macciò
Year: 2020 – 2021
Real Robot Test? Yes, we're interested in testing the code on the real robot.
Code Documentation: Code Documentation
GitHub Repository: GitHub Repository

Human Baxter Collaboration

Table of Contents

Abstract

The aim of this project is to move a robot in an environment where also a human operator is present. We want both the robot and the human to complete two tasks while working simultaneously: the human should pick all the red blocks and put them in a box, in the meantime the robot should pick up all the blue blocks and putting them in a different box on the other side

of the table. While completing the tasks the human and the robot should help each other and especially should avoid hitting each other.
In the following link we have our GitHub repository where all of our code is saved and thoroughly explained.

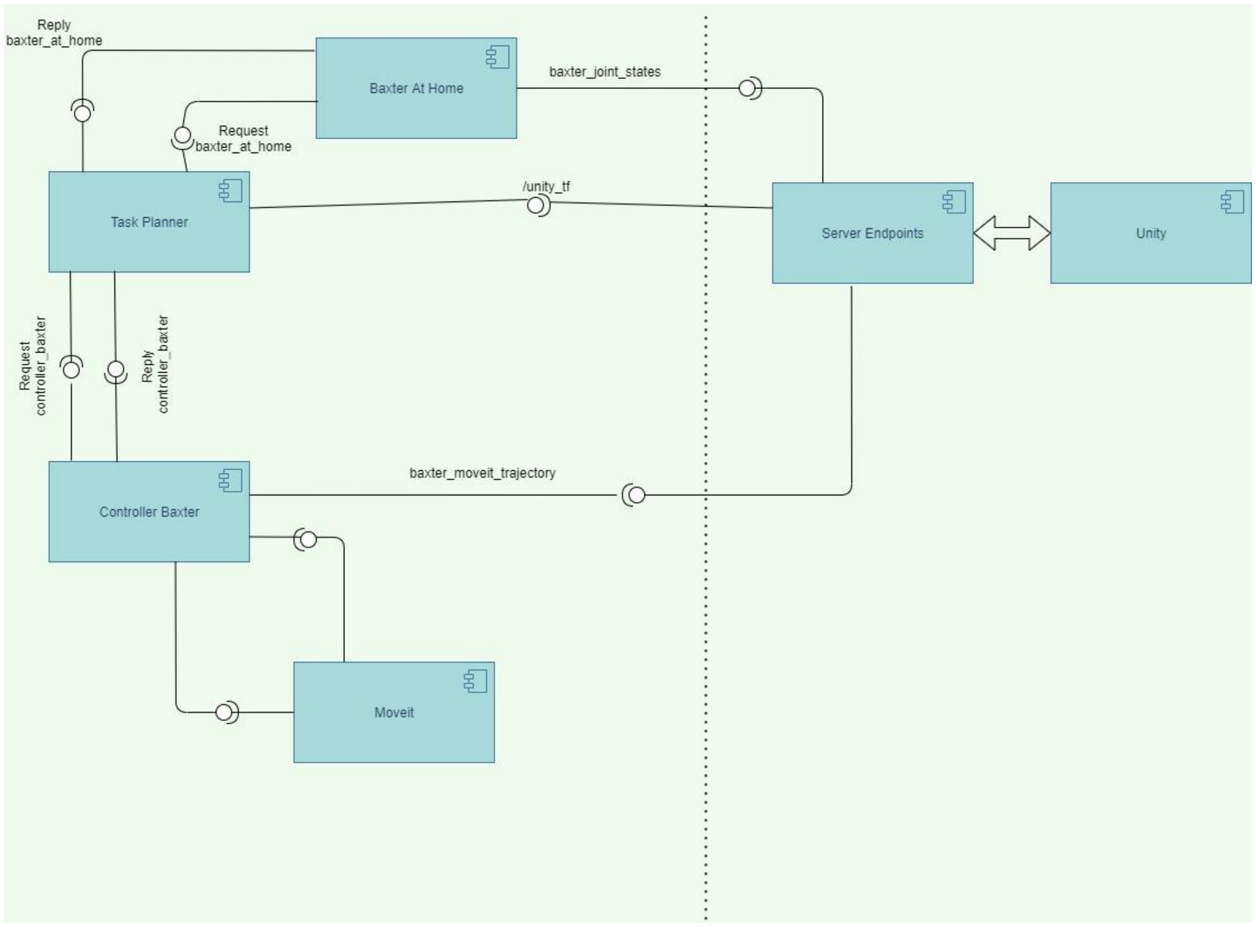
https://github.com/programmatoroSeduto/SOFAR-2020-2021-HUMAN-BAXTER-COLLABORATION.git

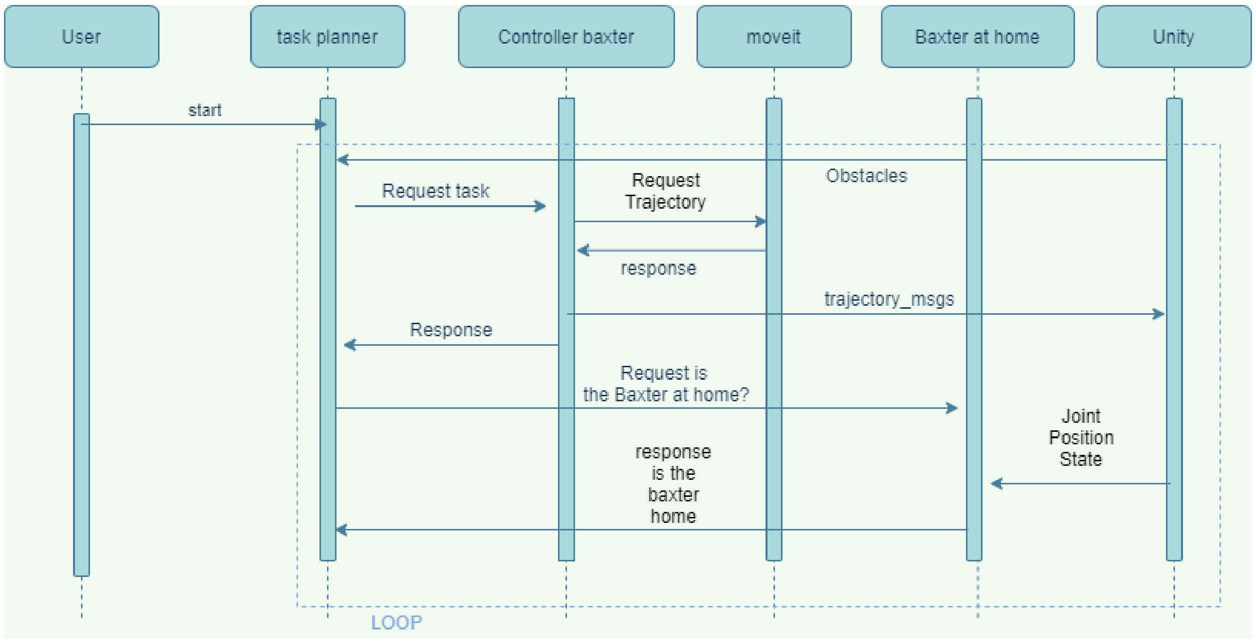
1. Introduction

This project wants to achieve the collaboration between a human and a robot in completing two separate tasks while helping each other out. We represented our system on the simulation software Unity. In particular we have a scene with: a table, and, 6 red blocks and 5 blue blocks on top of the table, a Baxter robot and a human represented opposite to Baxter. The human can execute a script that simulates his movements. We implemented our code in language C++ and python.
We decided to start our simulation by making the Baxter robot to pick up a blue block that was sitting on top of two red blocks. After that we start the human script that is now able to pick all red blocks without having any blue block that blocks them.

2. Architecture of the System

Our system to control the Baxter Robot behavior consists in four nodes, one written in python, (the task planner) and the other three in C++ (*baxter_at_home*, *controller_baxter*, and *go_to_home*). The node *go_to_home* is not shown in the UML diagram since it is not really used in the simulation but only if we have the robot in a random position and we want to move it in the home position.
On the right side of the diagram we have the unity simulation that communicates with the ROS nodes that are on the left side. The simulation sends information to ROS using the Server Endpoints. The server sends the information about the position of all the blocks, the hand of the human and the joint configuration. Those information are sent to the task planner whose object is to decide what each arm should do. If the task planner finds at least one task for one arm, a *task* is sent as a request to the server *controller_baxter*. The server tries and find a feasible trajectory using MoveIt. The c*ontroller_baxter* node returns true if a trajectory as soon as a feasible trajectory has been found, it sends false if no trajectory is acceptable. In order for the task planner to know if the task has been completed and to decide what to do next we need the Baxter at Home node. The Baxter nodes receives from unity the position of the joints and checks whether they are the same as the home configuration, only in that case it returns true to the task planner that can move on and find the new trajectory.
In the next page, we have the component UML and the sequential UML.


Reply baxter_at_home
Baxter At Home
baxter_joint_states
Request baxter_at_home
/unity_tf
Task Planner
Server Endpoints
Unity
Request controller_baxter
Reply controller_baxter
baxter_moveit_trajectory
Controller Baxter
Moveit


User
task planner
Controller baxter
moveit
Baxter at home
Unity
start
Obstacles
Request task
Request Trajectory
response
trajectory_msgs
Response
Request is the Baxter at home?
Joint Position State
response is the baxter home
LOOP

3. Description of the System’s Architecture

Our nodes have been developed to work on ROS. Here we can see in details how we decided to implement them. In order to have more information about the code of each module use the documentation provided at the following link:

https://programmatoroseduto.github.io/SOFAR-2020-2021-HUMAN-BAXTER-COLLABORATION/

Our architecture is a classic *hierarchical arch*. Initially we wanted to implement a *hybrid reactive-deliberate architecture*, but that was not possible due to limitations in the provided simulation environment.
The component *server_endpoint* is an adapter: it takes the data from Unity and returns them in a format compatible for our ROS Nodes, it also involved in a the *pub-sub* communication pattern as publisher.
The series of *task_planner* and *controller_baxter* is a configuration very similar to a computational pattern.
The servers implemented, the Controller Baxter and the Baxter at Home are Request-Process-Reply computational patterns, they receive a request from a client, process the data received and reply with the proper answer.
The set made up of *task_planner*, *server_endpoint*, *controller_baxter* forms a Request-Process-Reply computational patterns. A *pubsub* pattern is implemented in this way as well, because the “Ontology” node is distributed among all the nodes of our architecture. There is not a node *Ontology* in our architecture.

3.1. Task Planner

The module task_planner.py has been developed mainly by Francesco Ganci and Zoe Betta.
In order to be able to run this node, we need some additional packages such as *human_baxter_collaboration* and *ros_tcp_endpoint*. Those packages can be installed by following the instructions here: create a ROS workspace on a unix system, copy the folder dependencies inside the src folder of the workspace. Once installed those packages it is possible to run the node.
This node also requires an additional python script saved inside the folder includes/controller_baxter called sim_infos.py. This file contains a list of parameters needed only in the task planner file.
This file takes as inputs the data published on the topic /unity_tf or /tf, depending on one variable set in the file sim_infos.py. From that subscriber the node is able to retrieve all the position of the blocks on the table, the joint configuration, the position of the human collaborator hands in order.

Those information are used in order to decide what task to execute next. The node will run in a loop until all the blue blocks are inside the goal. The node checks for tasks for the left and right arm independently. We decided that the left arm can only pick blocks from the left side of the table or the center and can only put them in the goal box, the right arm instead can only pick objects from the right side of the table and placing them in the center. For both arms before making any decision we delete from the list of available blocks the ones that are too close to the hands of the human and the ones that are not accessible, when they are under other boxes. With the list of feasibile blocks in that moment we then decide which one to take: for the left arm we prioritize the block in the center if present, if there isn't a block in the center we decide to move the one furthest away from the hands of the human operator. For the right arm we check if there are any blocks in the center and if that is the case we don't assign any task to avoid collisions between the two Baxter arms and to avoid stacking blue blocks, if the center is free then it is chosen the block furthest away from the hands of the operator. Once the two tasks are decided we must decide if we want a parallel or sequential execution, ideally we would want always parallel execution but in practice that is not chosen if the position of the two blocks I want to move is closer than a given threshold. In order to know if the trajectory has been completed by the simulation the task planner calls the server baxter_at_home and waits for it to return True. The task planner is not able to check for obstacle while moving the robot, it cannot do a dynamic control, it is only able to do a static planning, looking for the description of the environment in the beginning of the planning and not at run time.
Since this module implements the logic of our architecture we don't receive any output from it. It closes itself when all the blocks are picked printing messages both on the screen and on the logs.

3.2. Controller Baxter

The module Controller Baxter has been developed mainly by Lorenzo Causa and Federico Zecchi. In order to be able to run this node we need some additional packages such as: human_baxter_collaboration, ros_tcp_endpoint, baxter_common and moveit_robots. Those packages can be installed by following the following instructions: create a ros workspace on a unix system, copy the folder dependencies inside the src folder of the workspace. Once installed those packages it is possible to run the node.
This module receives data by subscribing to the topic /unity_tf, also, since this node implements a server, it also receives the request that is composed of three fields of type string: arm, pos and cube. Inside the arm variable we store which arm we want to move, it can be "left" or "right". Inside the pos variable we state what the target position for the block is, this variable can assume the values "box" and "center". Finally the variable cube stores which cube we want to move, it can be "E", "M", "C", "G", "I".
The aim of this goal is finding a trajectory for the required task, in order to do this the package moveit is used. The task is always divided in five phases: the robot goes to a position right above the block, the robot moves down to the exact position of the block, the robot returns to the position right above the block, the robot moves the position above the target, the robot moves closer to the

table vertically. We decided to add the phase of returning to the position above the block after picking it up to avoid generating a trajectory that would hit other blocks on the table. After each task is executed the robot returns to the home position and the start configuration for the next trajectory is set to home.
This module publishes the trajectory on the topic baxter_moveit_trajectory. It also returns as a response to the node that called him true if it was able to find a trajectory or false in the other case.

3.3. Baxter at home

The module Controller Baxter has been developed mainly by Lorenzo Causa and Federico Zecchi. In order to be able to run this node we need some additional packages such as: human_baxter_collaboration, ros_tcp_endpoint. Those packages can be installed by following the following instructions: create a ros workspace on a unix system, copy the folder dependencies inside the src folder of the workspace. Once installed those packages it is possible to run the node.
This node subscribes to the topic baxter_joint_states in order to retrieve the current configuration of the joints of the robot. This node implements a server so it receives as input the server request that is composed of a string that states which arm we want to check and it can either be “right” or “left”.
The node retrieves the position of all the joints and compares it to the position of the joint in the home configuration. Only if the difference of all the joints from the home configuration is less than a given threshold (plus or minus 0.01) then the robot can be considered in the Home configuration.
As output this server sends a response to its client that is true if the robot is in the home configuration or false otherwise.

3.4. Go to home

The module Controller Baxter has been developed mainly by Lorenzo Causa and Federico Zecchi. In order to be able to run this node we need some additional packages such as: human_baxter_collaboration, ros_tcp_endpoint, baxter_common and moveit_robots. Those packages can be installed by following the following instructions: create a ros workspace on a unix system, copy the folder dependencies inside the src folder of the workspace. Once installed those packages it is possible to run the node.
This node subscribes to the topic baxter_joint_states to retrieve the current configuration of the Baxter’s joints.
This node is called once in the beginning of the simulation if the robot is not in the Home Configuration. In this case it plans a moveit trajectory that from the current state moves the robot to the Home configuration. We implemented it thinking about the real life implementation of the code where, contrary to what happens in simulation, the robot is almost never in the correct position when starting.

Since this node is called only once and only if needed there are not any outputs other than printing on the screen and inside the log files.

4. Installation

Since our dependencies are common for most of our packages it is possible to install them in the beginning for all of them. In order to do so it is possible to download inside the workspace the following github repositories:

- https://github.com/TheEngineRoom-UniGe/SofAR-Human-Robot-Collaboration
    - Inside the folder ROS Packages you are able to find the needed packages (human_baxter_collaboration and ros_tcp_endpoint)
- https://github.com/RethinkRobotics/baxter_common
    - Inside the folder ROS Packages you are able to find the needed packages (human_baxter_collaboration and ros_tcp_endpoint)

Also it is needed to install the package moveit for the simulation: to do so run in your terminal

- sudo apt install ros-noetic-moveit

To be able to run the code:
First of all, in your ROS environment, launch the server:

- roslaunch human_baxter_collaboration human_baxter_collaboration.launch &

On windows, launch the package MMILauncher.exe: it will let the human move in the simulation.
If the connection works fine, a print similar to this one should appear on your shell:

- ROS-Unity Handshake received, will connect to 192.168.65.2:5005

Launch the Unity environment, then "play"→ start simulation.
Here you can launch all the components of our project. Make sure the simulation is running before launching these components.
First of all, launch baxter_at_home node:

- rosrun controller_baxter baxter_at_home > /dev/null &

Then, launch the node controller_baxter which allows the robot to move:

- rosrun controller_baxter controller_baxter > /dev/null &

Last step: launch the task manager. After this, the robot will immediately start moving.

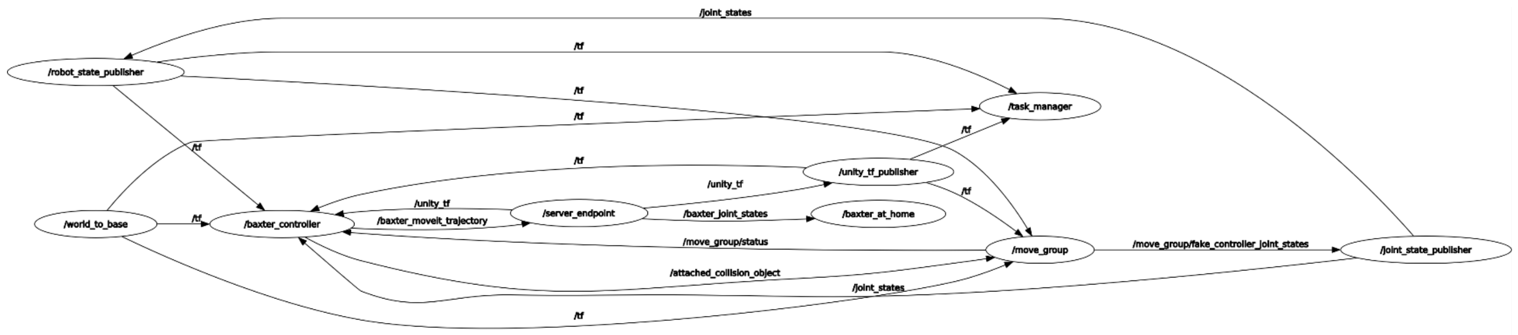
- rosrun controller_baxter task_manager.py

5. System Testing and Results

We tested our system both in simulation and with the help of the real robot. We have some videos of the simulations available at the following links:

- https://drive.google.com/file/d/1bliD6EbrQrFFnVxbXdXl74VSJtnRRKKW/view
- https://drive.google.com/file/d/1p-_naDokhO7L7R_C7RtqwC0Slp7nUhX5/view

We started the simulations by letting the robot pick one blue block that was sitting on top of one red block so that the human was able to pick up its red blocks without having to move the blue ones. After the robot has successfully moved that block the human starts moving. We noticed that the human movements are way faster with respects to the ones of the baxter robot, this causes the human to finish before the robot its task. In all of our simulations we never had a collision happen due to our policy of picking the blocks furthest away from the operator. We also notices that sometimes the Baxter would drop a block in the middle of the execution of the task probably this is due to the physics of the system. If we tried to accelerate the Baxter this behavior would increase, so we decided to keep the starting speed. Even if some blocks were dropped the Baxter was in any case capable of completing the task of putting all blocks in the goal.



6. Recommendations

For the realization of this system, we worked under the assumption that the Baxter robot could only move the blue blocks and the human cooperator could only move the red blocks. We also decided that the left arm could only pick blocks from the center or left side of the table and move them in the goal while the right arm could only move blocks from the right side of the table to the center.

To see if a block was accessible, we only looked at the planar coordinates since if two blocks are superimposed they can’t compenetrate and so one is on top of the other.

Another particularly important assumption we made was that a block could never fall down the table, in that case the Baxter robot would not be able to retrieve it and it would never finish its tasks.

We would like to make an improvement to the whole architecture by dividing the sent trajectory in more steps in order to check in between steps for the new position of the robot. That has not been possible in this first implementation since the simulation would significantly slow down reducing the time the robot and the human work together, we decided to focus on this last part considering it the main purpose of this system.

In order to move the real robot, we had to change our code, in fact during the simulation the robot would always return to the home position after finishing a goal while the real Baxter robot would stay in the target configuration. This created some problems since we had to impose also in the simulation for the Baxter robot to stay in the last position known but this created problems with the policy to decide the next tasks. In the simulation behavior when the right arm would drop the block in the center then it would move the Home configuration freeing the area for the left arm to pick it up, this does not happen anymore. To know if a task was done being executed it was checked if the arm was in the home configuration, that is no more true so we have no way of determining when a task has completed its execution.

We are working on a version of this code that solves those problems.