M1 Robotics and Interactive System

Ostfalia
Am Exer 1 · 38302
Wolfenbüttel

Internship Report

Development of a bartending robot

Creation of a web application using Docker, Flask and Node-Red to command a Sawyer robot in a complex scenario of interaction between several robots

24 avril 2023 - 21 juillet 2023

Presented and written by Loan BERNAT

Acknowledgments

I would like to start by thanking my training supervisors Prof. Gerndt Reinhard and Prof. Dörnbach Tobias for allowing me to work in the team.
I also thank Mr. Pereira Matheus Lara for his help and also Mrs. Bizien Véronique for all that she has done to help us get settled.

Ein großes Dankeschön an sie

Résumé - French abstract

Dans le cadre de ma formation d'ingénieur au sein de l'école UPSSITECH, j'ai eu l'occasion d'effectuer mon stage de M1 au sein du département d'Informatique de l'*université des sciences appliquées d'Ostfalia*. Ce sont les professeurs Gerndt Reinhard et Dörnbach Tobias qui m'ont encadré du 24 avril au 21 juillet et permis d'intégrer l'équipe de Programmation Visuelle (*visual programming*) du laboratoire de Robotique Centrée sur l'Homme (*human centered robotics laboratory*). La tâche qui m'a été confiée s'intègre dans un désir d'unifier l'ensemble des robots du laboratoire sous un même langage de programmation, sous un même outil. Ceci afin de simplifier la prise en main pour les étudiants lors de Travaux Pratiques, mais aussi, de faciliter le développement d'applications plus complexes présentant des interactions entre différents robots. L'outil choisi est Node-Red. Mes deux missions consistaient donc à développer une interface Node-RED pour le robot SAWYER (bras mécanique de chez Rethinks Robotics) et à réaliser une application de bartender utilisant cette interface. Pour m'aider dans mon travail, j'ai eu accès à un projet précédemment réalisé par deux étudiants allemands, M. Ruske Kai et M. Weike Michel, dans le cadre de leur mémoire de Master 2. Ce projet met en place l'interface Node-RED pour le robot Pepper (robot humanoïde de chez SoftBanks Robotics).

Pour réaliser ma première mission, j'ai donc pris beaucoup d'inspiration et d'idée auprès du projet sur Pepper et j'ai mis en place ma propre architecture logicielle pour mon application. Pour ce faire, j'ai utilisé Docker pour créer trois différents conteneurs qui me permettent d'isoler chaque partie de l'application : L'interface Node-RED (gérant l'interaction avec l'utilisateur), mon package ROS *bartender_sawyer* (communiquant avec le robot) et le rest-serveur (faisant office de lien entre les deux premiers). J'ai créé quelques premières fonctions basiques pour le Sawyer, telles que : afficher une image sur sa tablette, allumer une LED, positionner son outil à une situation (position et orientation) donnée… L'utilisateur peut définir des paramètres via l'interface Node-RED. Ces paramètres sont alors transmis au module du rest-server associé à la node exécutée. Ce dernier va ensuite construire une commande Unix afin d'exécuter le script ROS associé avec les arguments correspondants puis l'envoyer à un *listener* du ROS conteneur qui exécutera la commande dans un sous-processus. Les communications sont gérées par Socket.io, HTTP et TCP. Avec un temps moyen de communication de 0.4 secondes, je considère avoir obtenu un bon résultat. Malheureusement, je n'ai pas trouvé de comparaison possible afin de réaliser une analyse détaillée des performances de mon application. De plus, après une discussion avec mes encadrants, nous avons défini ensemble que l'optimisation de la communication n'était pas la priorité. J'ai donc pu poursuivre avec la partie centrale de mon stage : le bartender.

J'ai identifié plusieurs aspects essentiels à gérer pour mon application de bartender : précision des mouvements, vitesse d'exécution, gestion de l'environnement, gestion des priorités et robustesse. J'ai dans un premier temps émis les hypothèses suivantes : les positions des verres et des bouteilles seraient fixées et connues à l'avance, les bouteilles seraient ouvertes et les bouteilles seraient réutilisables à l'infini. Avant de commencer pleinement le développement, j'ai défini l'environnement du robot de manière à créer 3 espaces : rangement des verres, zone de préparation et rangement des bouteilles. J'ai également décidé d'utiliser des gobelets en plastique à la place des verres, pour éviter toute

détérioration de l’environnement ou du robot. Enfin, j’ai choisi 3 boissons différentes que le robot sera en mesure de servir : Coca, Eau et Limonade à l’orange. J’ai ensuite pu créer un ensemble de node appelé “outils classiques du bartender” (Sawyer Bartender Classic Tools) permettant à l’utilisateur de définir sa propre logique de commande pour le robot. Afin de gérer la mémoire de l’environnement et les priorités, j’ai implémenté une classe statique utilisant des sémaphores au sein du rest-server, permettant de centraliser et sécuriser toutes les interactions des différents threads avec les attributs importants de cette classe (liste d’attente, statut du robot…). Pour terminer, j’ai ajouté un ensemble de scripts à mon package ROS. Ces derniers mettent en place de la planification de trajectoire via la *Motion Interface* développée par Rethink Robotics. Obtenant, pour la meilleure version, un temps de préparation moyen de 1 minute 07 secondes et une précision de 100 %. Satisfait de mon résultat, j’ai souhaité retirer l’hypothèse de départ concernant la position fixe des bouteilles. J’ai donc développé un nouvel ensemble de nodes appelé “outil avancé du bartender” qui vient compléter le précédent avec deux nouvelles nodes mettant en place de la détection d’objet via Yolov5. J’ai donc créé ma propre database puis entraîné différents modèles pour enfin sélectionner le yolov5s (Petite taille) qui présentait le meilleur compromis entre véracité et vitesse. J’ai intégré ce modèle dans mon processus de planification de trajectoire via la librairie Torch et en utilisant des équations classiques de projection d’optique pour transformer les coordonnées des objets en 2D sur l’image à une profondeur et un décalage à effectuer pour le bras robotique pour saisir l’objet. À l’issue, j'obtiens un temps de préparation moyen de 1 minute 15 secondes et une précision moyenne de 80% (variant selon l’objet).

En seulement 3 mois, j’ai réussi à concevoir un robot semi-autonome capable de travailler de manière continue avec l’aide d’un superviseur (humain ou robot). Ce dernier doit compenser les limites actuelles du robot. En d’autres termes, il doit vérifier la bonne saisie des objets et remplacer les bouteilles vides par des nouvelles. Mon projet présente également de nombreuses pistes d’amélioration et de continuation possibles, telles que : améliorer le remplissage des verres en mettant en place un système capable de déterminer la durée et le degré d’inclinaison pour toujours servir la même quantité, ouvrir seul les bouteilles et jeter les vides, réalisation de cocktail ou enfin optimiser la communication.

Summary

List of Image

List of Appendix

Glossary

Node-RED : Node-RED is a programming tool for wiring together hardware devices, APIs and online services in new and interesting ways.

Docker : Docker is a platform that simplifies and accelerates the development, packaging, sharing and running of applications using containers.

Docker container : A docker container is an isolated environment to run code. It has no knowledge of your operating system or your files.

HTTP : Hypertext Transfer Protocol is a client-server communication protocol developed for the World Wide Web.

TCP : Transmission Control Protocol is a reliable, connection-oriented transport protocol that ensures the transmission without loss and in order of data between applications.

MQTT : Message Queuing Telemetry Transport is a publish-subscribe messaging protocol based on the TCP/IP protocol.

Socket.io : Socket.io is a library that enables bidirectional and low-latency communication between a client and a server.

Flask : Flask is a lightweight web framework written in Python. It provides a minimal and flexible core that supports extensions for common functionalities.

ROS : Robot Operating System is a set of tools and software libraries for developing robotic applications.

YOLO : YOLO (You Only Look Once) is a family of object detection models which use a single neural network to predict the class and location of multiple objects in an image.

Yolov5 : Yolov5 is a model from the YOLO (You Only Look Once) family for object detection.

RobotFlow : RobotFlow is an open source toolkit for mobile robotics based on the project FlowDesigner.

Pytorch: Pytorch is a Python library optimised for deep learning using GPUs and CPUs.

Introduction

As part of my second year in Robotic and Interactive Systems at the UPSSITECH engineering school of Toulouse Paul Sabatier University, I completed an internship at Ostfalia - University of Applied Science in Woffenbüttel, Germany.

From 24 April to 21 July, I worked with the team of the *Human Centered Robotic Lab* (HCR-lab) of the Computer Science Department. More precisely, with the “visual programming group”.

Indeed, the objective of this group is to create an interface with Node-Red which can be used on all robots of the laboratory. This will allow future students to work on robots without needing an in-depth knowledge of them. It will also facilitate the interaction between robots to develop more complex applications. **My task was to create this interface for the Sawyer Robot and to make an application with it : a bartender able to interact with robot waiters and to prepare orders.**

This report summarises my work during this internship but does not provide technical detail such as the code for each feature or installation and debugging instructions. I followed a research approach throughout my internship by producing functional versions for each stage and improvement. Next chapters will follow this evolutionary protocol by presenting my methodology and my results for each version.

The first part will introduce the team, the previous work, some concepts and my task. The second chapter will explain how I created the structure of the application. Then, I will describe my approach to the design of the fixed-position bartending robot in the third chapter. The fourth chapter will present how I improved my work by using object detection instead of fixed positions. Finally, I will give a technical and personal assessment and share the current limitations of my work with ideas for improvement in the concluding section.

1. Context of the internship: a robot bartender

Working in a laboratory involves collaborating with a team, building on the work of previous colleagues and creating new innovations. That's why I think it's important to present the team and the background of my work in this first section. Therefore, I will first introduce the Team. Then I will describe the objective of my task and hardware and software components of the project, namely the Sawyer robo, the Node-RED and Docker. Finally, I will review the previous work done on the Pepper robot and how it relates to my task.

1.1 : The Team

Prof. Gerndt Reinhard and Prof. Dörnbach Tobias are both professors in the computer science department of Ostfalia. Prof. Dörnbach is responsible for the visual programming group. As I mentioned in the introduction, the goal of this group is to implement a Node-RED interface for all robots :

- **Turtle Bot** : Some members work on it.
- **Pepper Robot** (Pepper or Salt version) : Two students started working on it as part of their master's thesis.
- **Baxter Robot** : Not yet implemented.
- **Temi Robot :** Not yet implemented.
- **Sawyer Robot** : I worked on it from scratch.

Another employee that is important to introduce is Mr. Pereira Matheus Lara. He helped us get started with previous work and to familiarise ourselves with the various tools available in the laboratory.

1.2 : Objectives

My task is part of a robotic bar scenario in which waiters and the bartender are robots. I was in charge of developing the bartender using the Sawyer robot.

My application must allow the robot to receive orders sent by a waiter and understand it, to communicate with waiters ("order is ready to be picked up", "order is on your tray") and of course, to prepare orders. The only requirement I had to follow was to build the application using Node-RED.

Here is the scenario : The robot receives an order and according to it, it goes to get a glass, and puts it on the bar. Then, it picks up the appropriate bottle (Water, Coca, Orange...) and pours into the glass. When the pouring is finished, Sawyer has to send a message to the waiter to say that the order is ready to be picked up. Finally, when the waiter sends a message to Sawyer saying that it is in position to take the order, Sawyer deposits the glass on the waiter's tray.

When I chose this task, I already had a few ideas for improvement :

- Multiple orders in parallel
- Opening bottles
- Using computer vision to detect bottles or glasses.
- Making Cocktails

1.3 : Hardware : The Sawyer Robot

The Sawyer robot is a robotic arm created by Rethinks Robotics. It has two operating modes : Intera 5.1 (the classic one) and Intera SDK (the advanced one). Intera 5.1 is a software which gives a graphic interface to command the robot by creating tasks with a behaviour tree (see figure 1.1). Intera SDK allows the user to command the robot using ROS. This is the mode that I used to create my application because I could not interface Node-red and the Intera software. In addition, for object detection, it was useful to create my own scripts.

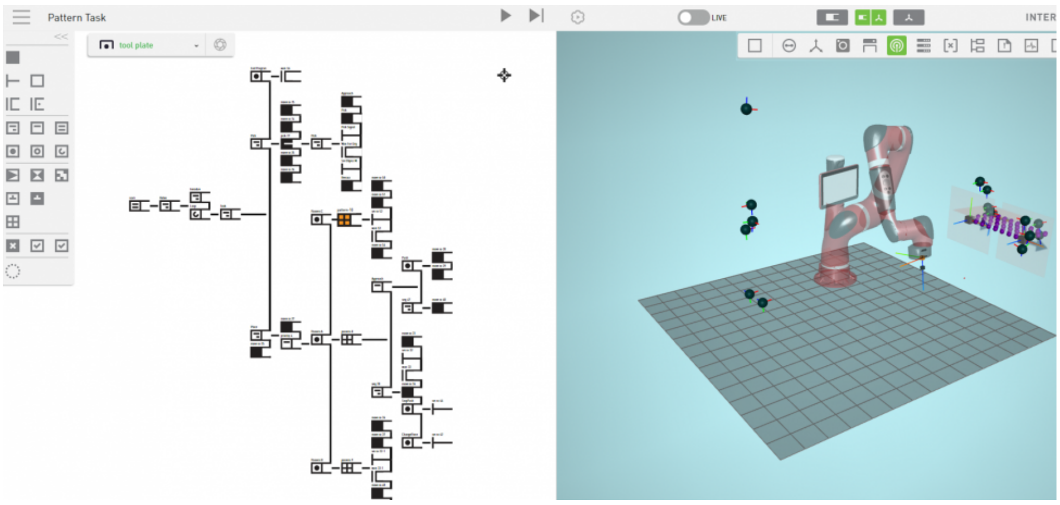


Figure 1.1 : Screen of the Intera 5.1 Software from [Rethink website]

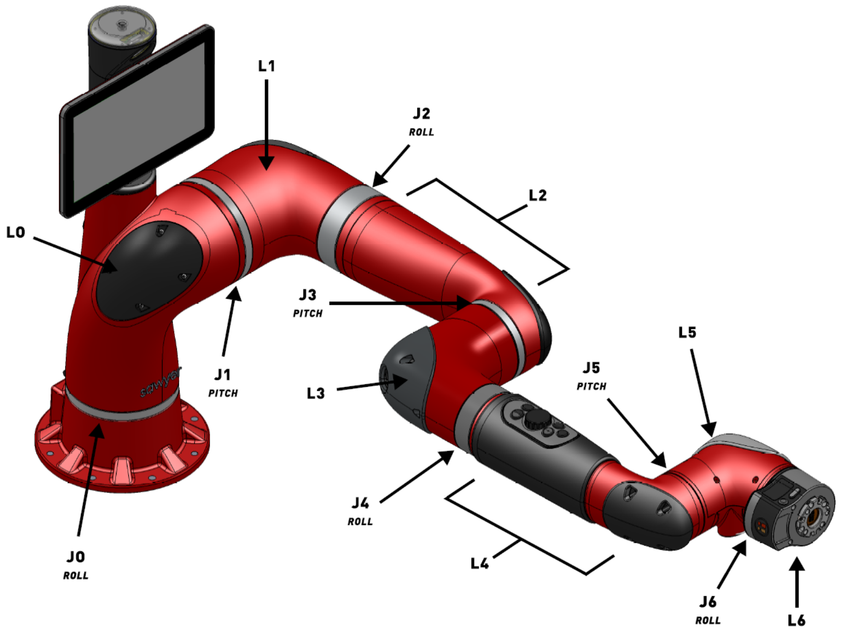


Figure 1.2 : Description of the Sawyer Robot Hardware from [Rethink website]

As you can see on figure 1.2, the robot has 7 joints numbered from 0 to 6. It has a screen representing the head and an adjustable gripper. All these joints enable it to be very precise in its movement.

1.4 : Software : Important concepts

1.4.1 : Node-RED : An IoT programming tool

To simply define Node-RED : It's a really powerful programming tool to design IoT (Internet of Things) applications. It uses visual programming with blocs that the developer can link together (see figure 1.3)

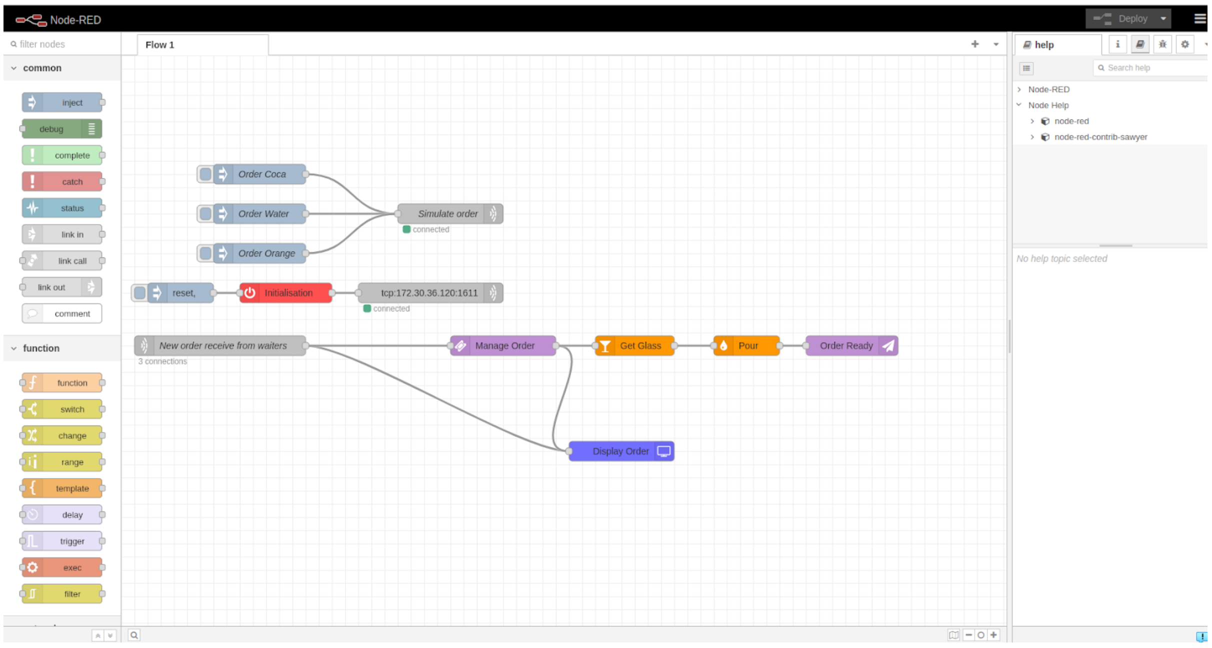


Figure 1.3 : Screenshot of the interface of Node-RED with a flow

In my case, Node-RED will allow students to create robotics applications on the sawyer quickly and easily. And more generally for the lab, this will enable all the robots to operate under the same system, so that we can design applications where several robots interact without needing to know the specific features of each robot.

An important point to understand is that multiple threads of one node can run at the same time if it receives multiple injections (entry message by the link). Messages are under JSON format. But, the Sawyer robot can't execute multiple ROS scripts at the same time. It's something that I needed to handle in my application.

1.4.2 : Docker : An application structuring tool

Docker is a containerisation platform that enables applications to be created, deployed and run in containers. Containers are lightweight, portable runtime environments that isolate applications from their underlying runtime environment.

My application is composed of 3 docker containers. I used docker-compose, which is a tool to create multi-container applications and for each container, there is a DockerFile to specify some packages to install or commands to run during the build of it.

1.5 : Study of the previous work on Pepper

As I said in 1.1, Mr. Ruske Kai and Mr. Weike Michel, as part of their Master's thesis, had already started and succeeded to implement a Node-RED interface on Pepper Robot. This interface is already used by students during the course. However, the structure of the application was intermingled. Then, professors wanted me to use their work as inspiration to make an improved version for the sawyer and at the end, integrate my work with Mr. Periou-Dezy to replace the previous version.

The application is divided into 3 docker containers (see figure 1.4) :

- **Mosquitto** controls Pepper's thermal camera.
- **Rest-server** is a rest API that receives requests from Node-RED container and executes the appropriate service on Pepper. It uses Flask Python.
- **Node-red** corresponds to the front-end. It executes the Node-RED interface.

Each container communicates using the IP address of the host machine and HTTP, MQTT or TCP protocols are used.

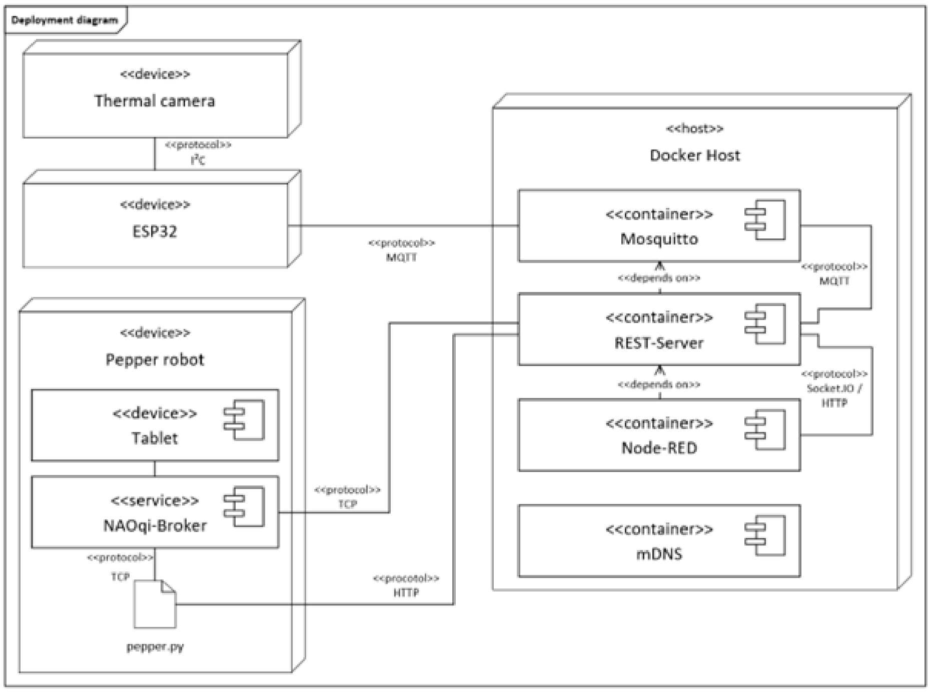


Figure 1.4 : Structure of the Peper-node-red project by Mr. Ruske Kai and Mr. Weike Michel

2. Creating basis of the application and basic nodes

The first step of the Bartender Sawyer was to adapt the pepper project, presented in 1.2, to work on the Sawyer. As I said in 1.3, I used the Intera SDK mode because it allowed me to command the robot using ROS package and python. In this section, I will describe the structure of my application and the nodes I have implemented to familiarise myself with all the concepts. I will conclude this section with a performance analysis.

2.1 : Basis of the application

Starting from the Pepper's application, I've rethought the structure by changing the mosquitto container into a ROS container and trying to simplify communications by making the robot communicate only with the ros container. Figure 2.1 shows the current structure of my application.

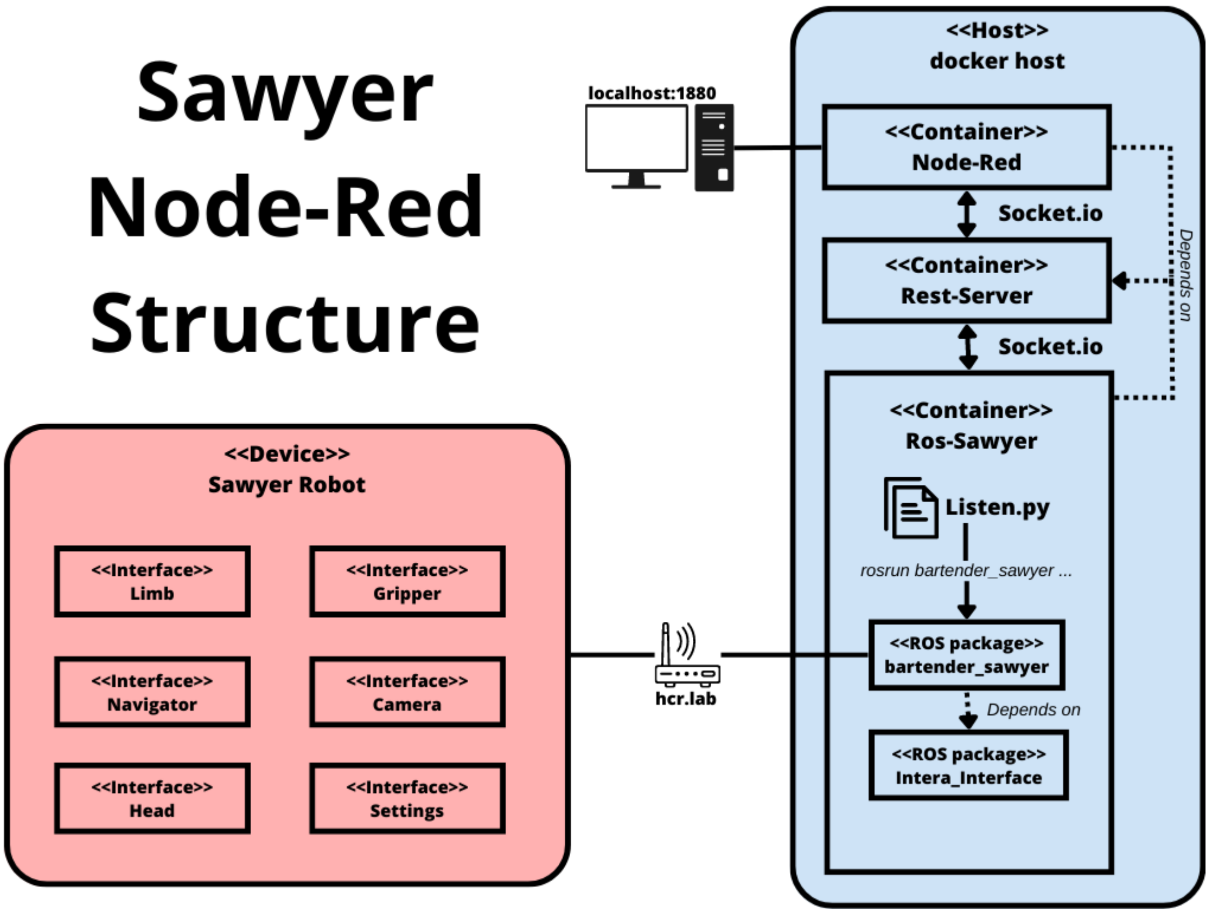


Figure 2.1 : The structure of my application. Diagram taken from my technical note

I kept the global structure of the rest-server and node-red containers. I deleted all nodes, functions and MQTT/TCP protocol from them and changed connection settings to adapt to the Sawyer. Requests between rest-server and node-red still use Socket.io and HTTP. They are named (e.g : “/sawyer/display”) and the message is composed of parameters. The rest-server function which receives this request may build an unix command, if the node needs to execute a ros script from the ros-container, according to these parameters (e.g : “rosrun bartender_sawyer display.py -u background.png”) and sends it by TCP using Socket.io to the ros-container. This container has a script, *listen.py*, which is always waiting for a connection from the rest-server to execute the command sent in a subprocess terminal using the subprocess python library. And so, executing the appropriate ROS script with given parameters.

The three containers need to be built before being executed for the first time. So I modified the old DockerFile and created my own for the ros-sawyer container. However, I have not lost sight of the fact that this application must be usable by students. It must therefore be as generic as possible and easily configurable. So I've also taken up (from the previous work) the idea of having an .env file that allows important environment variables such as the robot's IP address, communications ports etc. to be defined inside it. By the same time, I used a docker-compose file to make it easier to launch the 3 containers by typing just one command

The application can be downloaded directly from ostfalia's gitlab. All you need to do is set parameters in the .env file, compile the application by running a script and launch it by running a docker command. I give more details on it in my technical note which you can find in the appendix.

The second step was now to create some mere nodes to command the basic tools of the robot from the Node-RED interface.

2.2 : Creating the “Sawyer Tools” nodes set

To familiarise myself with Node-RED, I started by creating basic nodes which could be used to create applications other than a bartender (e.g : pick and place) :

- Display : Display image(s) on the tablet of the Robot.
- Gripper Actions : Command the gripper (Open / Close).
- Lights : Handle LEDs of the robot.
- Move To Joints Angle : Move the arm by giving an angle value for each joint.
- Move To Situation : Move the arm by giving a position and orientation for the gripper.
- Move Relative : Move the arm by giving a translation and rotation value for each axis.
- Test : Execute an unix command into the robot environment (Used for debugging)
- Wait : Wait a given number of seconds

I won't go into detail here about how these nodes work. There's no real point, because they're generally just calls to basic functions in the robot's command interfaces, they don't involve any complex scripting and they are not linked to my bartender application.

To create a node, I needed to create files in each container :

- Node-red (4 files) : One for handling when the node is called, *in javascript* ; another for the appearance and entering parameters, *in HTML* ; the last two for documentation and text, *in HTML and JSON*.
- Rest-server : One function to build the correct command depends on settings of the node, *in Python*.
- Ros-Sawyer : The script to execute, *in Python*.

To illustrate the interaction between all these files, figure 2.2 shows an example of the execution of the node Lights. The user selects parameters which will be transmitted to the rest-server's python module when the node receives an input message.

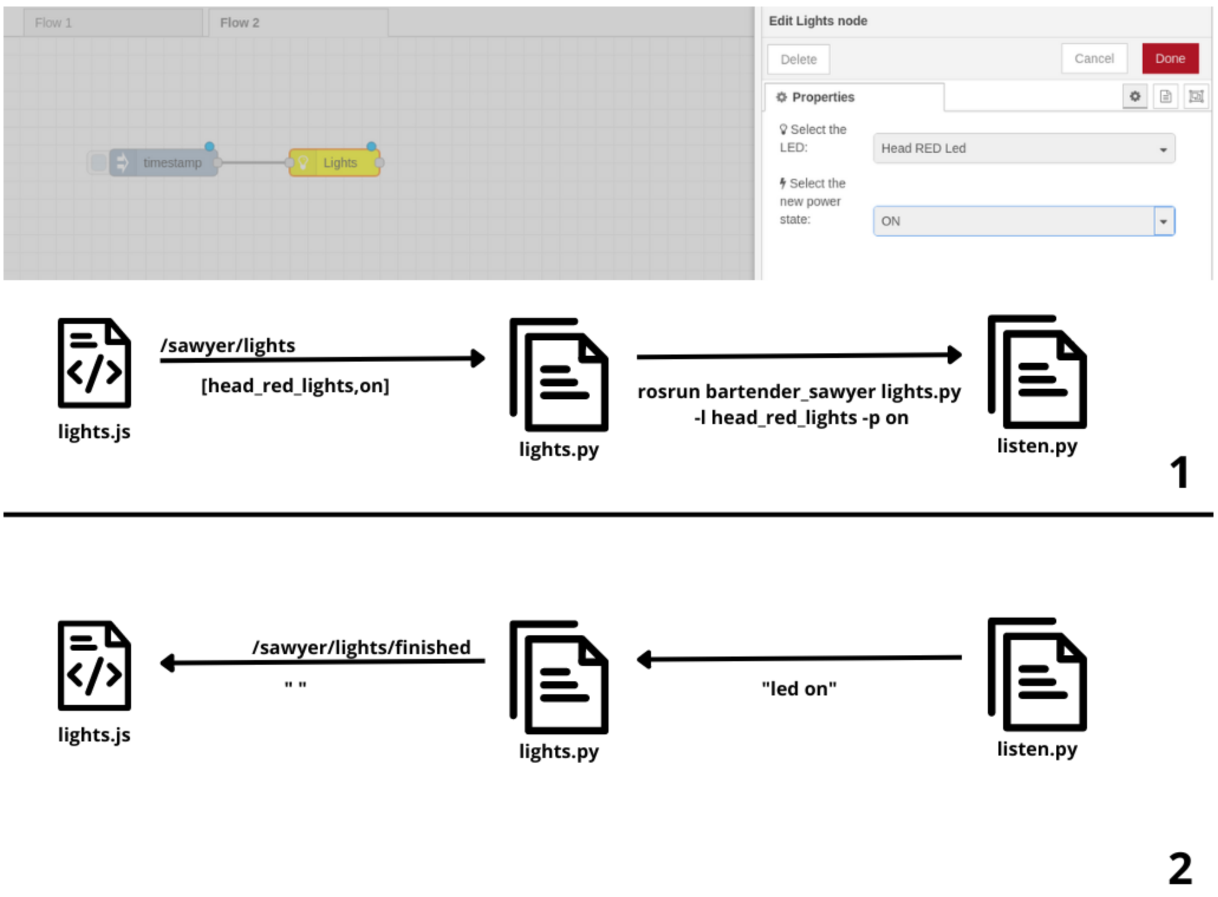


Figure 2.2 : Execution of a Lights node

One slightly ambiguous point that should not be misunderstood is that, sometimes, some ROS scripts have the same name as my rest server python modules. They do not have the same function at all (they are even in different containers). The python module is used to build a command to call the script.

Handling errors is an important thing to anticipate before starting working on the main application. Therefore, when a ros script is running, it prints logs on a virtual console. My script listen.py gets the stderr of this subprocess and sends it to the rest-server in response to the initial request. The rest-server module will then process this message and, if necessary, send the error back to the node in the node-red container, which will display an error status to warn the user.

<u>2.3 : Performance measurement</u>

As network communication wasn't something we learned in depth during my course, the results I get should be able to be improved. Indeed, the current structure is not really secure : anyone who knows the IP address and communication port of the host machine can interfere with communications. Following the advice of Prof. Dörnbach Tobias, I focused on my main task and I've left network upgrades aside.

For simple nodes, like Lights for example, the execution time is a bit less than 2s. Of these 2 seconds, 1.2 seconds are due to the ROS script being launched. Which is not really related to my structure. I measured a time of about 0.4 seconds between the call of the node in the node-red container and the execution of the command in a subprocess on my ros-container. As a barman's action time is in the order of a second, I consider 0.4 seconds to be acceptable. I did some research to find a node-RED project which “connects” ROS and Node-RED to compare this time. But I wasn't able to find something with a similar structure as mine, making the comparison irrelevant.

3. Developing the first version with fixed position

This chapter gets to the heart of the matter for our bartender. This version must perform all the basic functions of a bartender: order preparation (from receipt to delivery to the waiter's tray), workspace management and prioritisation. To achieve these tasks, I implemented a new set of nodes called “Bartender classic” for this version. Indeed, 5 different sub-versions were released. Each version represents a significant advance or a major change in the technique used. However, I will only compare the two major ones, 1.0 and 1.5, in this report. The first uses only the Limb module of the Python API to control the robot displacement and the second one uses the Motion_Interface (developed by Rethink Robotic). To make this comparison, I will use three criteria which I will also define in this section.

I will therefore first start by describing the robot’s environment and my thoughts on its design. Then I will describe each node of the “Bartender Classic” set and present how I chose to manage priority and memory. To finish, I will give you an evaluation of the two major sub-versions.

3.1 : Defining the robot’s environment

I designed the robot’s environment to be as similar as possible to a real bar but I had constraints to respect : no access to shelves or high tables and the robot must be able to move without colliding with itself in the environment. The figure 3.1 shows what the environment I designed for the sawyer looks like :

- At the left of the picture, it is the “plastic cup area”. It represents the place where the man/robot would put the glasses down after cleaning them.
- In front of the robot, it is the “bar”. This is the preparation area with 3 positions. The bartender can have a maximum of 3 orders waiting for a waiter before being blocked. I determined this number because the laboratory has only 2 Pepper, so with 3 slots, the sawyer should never get stuck. Between the bar and the plastic cup area, there is the place where the waiter should go so that Sawyer can place an order on its tray.
- At the right of the picture, it is the “bottle area”. It represents the traditional bar shelves on which all the bottles are displayed.

1.X versions of the Bartender application are designed to work with fixed positions. So I marked the positions of bottles/cups/tables with tape.

Sawyer Bartender works with 3 types of drinks : Coca, Orange Limo and Water. I have chosen these drinks because they have 3 different shapes of bottle to facilitate next steps but also easy to catch with Sawyer’s gripper. And I replaced glasses with plastic cups to avoid any deterioration problems and improve the safety. You can also see all of these objects on the 3.1 picture.

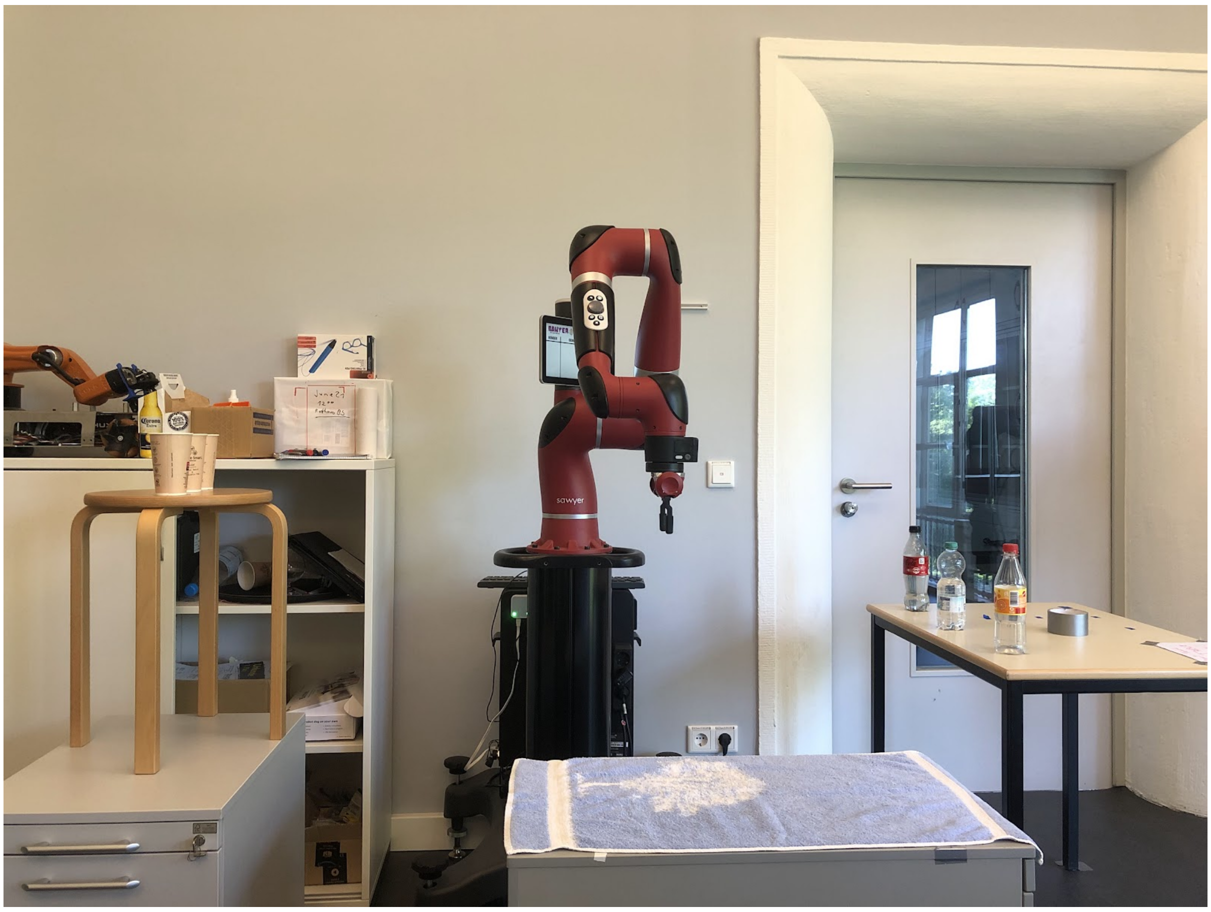


Figure 3.1 : The Sawyer Bartender's environment

3.2 : Creating the "Bartender classic" nodes set

To realise the application, I needed to find the right balance between High-Level nodes and Low-Level nodes. Because, if I created High-Level nodes, they will be usable only in my case. And if, later on, I or someone else wants to make the scenario more complex or improve on something, he will have to redo the node completely. In other hands, in only 3 months, I didn't have the time to work only with Low-Level nodes. Also, as the execution time for a node is not negligible, it would take too long to complete an action with too many nodes whereas a bartender needs to get the job done as quickly as possible (more details on my selection criterias will be given in the 3.4 section).

So, I chose to divide the task into two parts : Get the glass, Pour into the glass. Like this, I kept a "reset state" when an execution of a node is over : the robot goes back to its parking position with nothing in its gripper. Thanks to this "reset state", my nodes can be used in different application schemes. In my case, I've designed my application in this way : the robot takes a glass and deposits it into the preparation area, then takes a bottle and pours into the glass before putting the bottle back where he takes it. But with a more complex application, we could imagine that the robot first looks to see if there are glasses available on the bar before going to take one. Or even, when there is no order to prepare, it takes all the glasses available and puts them on the bar to save time later. The structure I chose is also quite high-level, because, for example, I didn't separate the bottle picking from the glass filling. This would lead to the loss of the reset state between the nodes.

Figure 3.2 shows the structure of my application on Node-RED. We can see 4 differents parts:

- The first part simulates the sending of a command received from a server. They are sent by TCP.
- The second simulates the message from a waiter in position to take a prepared order.
- The third box is just to show what waiters received from Sawyer when an order is ready to be picked up and when the order is on the waiter's tray.
- Finally, the fourth component is the main application.

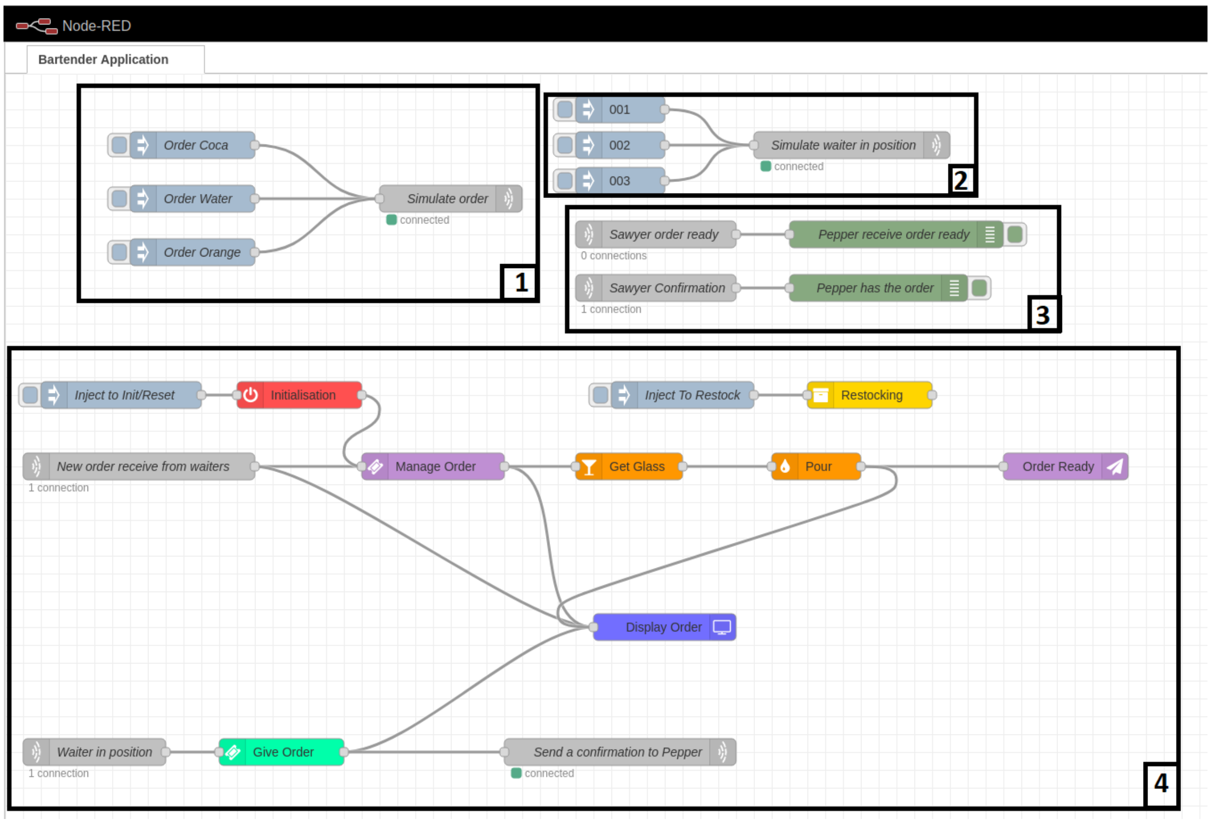


Figure 3.2 : Screenshot of version 1.5 of Bartender Sawyer Application

Now, I will present more in detail my nodes that you can see in figure 3.2. Inject nodes (blue grey) and TCP nodes (grey) are from the basic Node-RED librairie.

I created 3 types of nodes : executing nodes, managing nodes and hybrid nodes. Nodes I call "Executing nodes" are nodes which simply build a command to execute a ROS script. "Managing nodes" only manage interaction with the Bar class (see section 3.3 for more details on this class). "Hybrid nodes" execute a script and update an attribute of the Bar class.

Initialisation - *Hybrid*

The initialisation node is the global reset of my application. It executes my initialisation script : It places the robot in its parking position, calibrates the gripper and sends a “reset” message for following nodes.

Manage Order - *Managing*
Manage Order receives the order and transforms it into a message containing instructions to prepare the order. There are two important points to handle : manage multiple orders (waiting, priority) and manage available slots. These points are managed by the Bar Class with a waiting list and semaphores (more details on section 3.3).

When an order is received, a new thread is created and it waits for its turn. Afterwards, Manage Order sends back, to the node-red container, instructions for the following node to prepare the order and set the robot’s state to busy.

In parallel to this, I've made the node status in the Node-RED interface show the number of pending orders.

Order Ready - *Managing*
This node marks the end of the preparation of an order.

When a message is received, the node-red container sends to the rest-server just the ID of the order. This module interacts with the Bar class to manage a waiting list and change the robot state to free.

It also sends a message to waiters using a TCP port to prevent that the order is ready to be taken.The user needs to specify the address IP in parameters of the node. This allows the user to run everything on one computer but also to use different computers (one for sawyer, one for pepper).

Get Glass - Executing
Get Glass node sends a command to execute the ROS script get_glass.py. The node has two settings : pickup position and dropoff position. These settings can be set by the user or set on “Automatic”. In this case, node will read the input message and try to extract instructions from it. The message needs to specify two keys : ‘startpos’ and ‘endpos’. If the message isn’t correctly formatted, the node will send the message to next nodes and do nothing.

In the right case, the rest-server python module receives these 2 instructions and builds the command to send to my listen.py in ros-container. The script get_glass.py has a set of points and depending on what slots are passed in arguments, adapts its trajectory to take the cup and deposit it at the position specified.

As is said in 3.1, there are 3 different positions for picking up glasses and also 3 positions possible to prepare an order. So, to adapt the robot’s trajectory in the ROS script, I defined “global” waypoints which are common at all positions and for others (like for example, place the robot in front of a glass at a given position number) I defined a list of waypoint in which the index 0 correspond to the waypoint if the robot needs to go to the position 1, the index 1 to position 2 and index 2 to position 3.

Pour - Executing
Similar to “Get Glass”, Pour node sends a command to execute a ROS script pour.py. It also has 2 settings, ‘endpos’ (position of the glass) and ‘drink’ (the type of drink, e.g “Coca” “Orange” “Water”). These settings can also be set by the user or set on “Automatic” (extract form the input message) and the rest of the protocol is the same as with Get Glass.

The script pour.py has also a set of points and adapts its trajectory depending on arguments. Bottles are supposedly already opened. The robot follows a trajectory that minimises the tilt of the bottle (to avoid spilling it). It approaches a point above the glass and slowly descends, gradually tilting the bottle to pour it for a second. He then stops and reverses his trajectory to put the bottle back in its place.

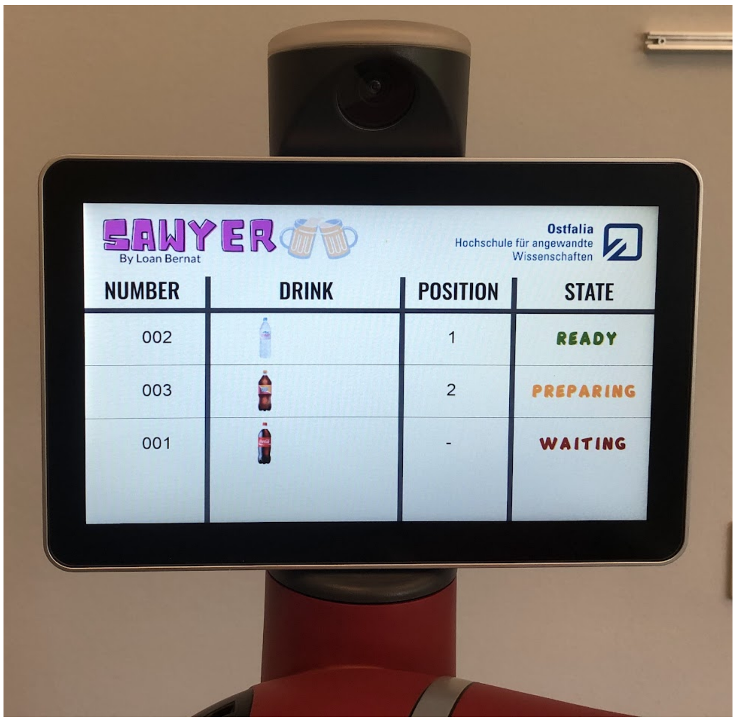
I used the same mechanism to adapt the trajectory as for Get Glass : List of waypoints.

Display Order - Hybrid
This node gives an interface to the Sawyer like an order screen. Customers can see which orders are being prepared, which orders have been completed and are waiting for the waiters to pick them up, and which ones are on hold. It doesn’t take any parameters, everything is automatic. This node can received 3 different topic of message :

- From the waiters : When it receives an order from the waiters, it adds the orders on the waiting list and, if it can be, displays the order on the screen.
- From Manage Order : The first time it receives a message formatted by a Manage Order node it changes the status of the order from “WAITING” to “PREPARING”. The second time, it changes the status from “PREPARING” to “READY”.
- From Give Order : The order is deleted from the list.

The tablet of the Sawyer can only display an image of a fixed dimension. So, I created a background image with the correct dimensions. It can display 4 orders at the same time. When a message is received by the node, the node is looking for the format of its. The Node-RED component transmits to the rest-server module 3 data extracted from the input message : Order’s ID, drink, endpos. This last parameter is equal to 0 if the message comes from a waiter (the Manage order doesn’t assign a slot to this order yet).

The python module builds the image based on a list of orders, which is represented by a dict which is an attribute of the Bar class. Once the list has been updated, the script transforms the background image into a numpy array and adds the images, numbers and status by replacing the pixel values of the background image with those of the pixels of the image to be "pasted". Figure 3.3 shows an example of the result for 3 orders. 002 is finished, 003 is in preparation and 001 is waiting to be prepared.



NUMBER	DRINK	POSITION	STATE
002		1	READY
003		2	PREPARING
001		-	WAITING

Figure 3.3 : The orders screen of Sawyer Bartender

Give order - Hybrid
This node receives a message containing an order ID from the Pepper when this one is in position to take this order.

An important thing for the “Give order” node is that the Pepper should wait as little time as possible. So, I built a priority system that allows Sawyer to stop preparing other orders to give the order that Pepper needs. Afterwards, Sawyer gets back to work normally.

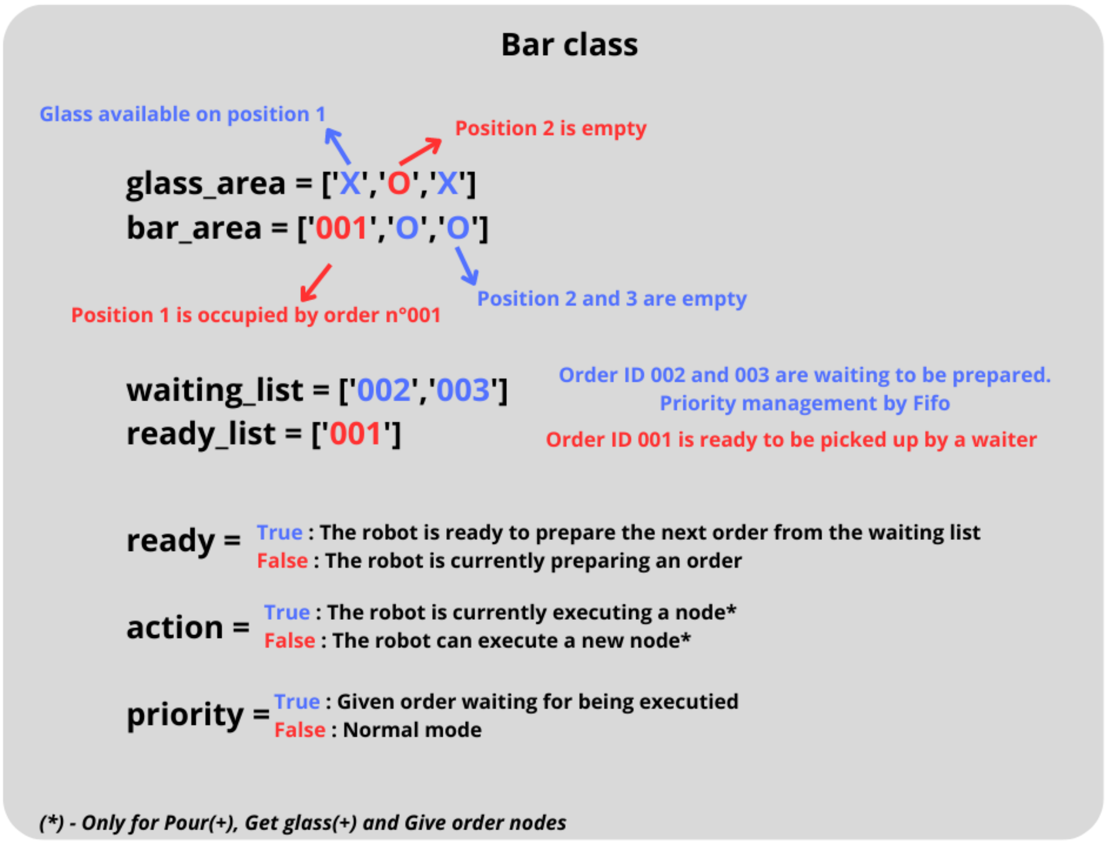
When a request is received, the module checks in the Bar class the position of the order on the bar area and sends to the Rors-container an unix command to execute the give_order.py script with the correct arguments.

Restocking - Managing
The Restocking node is used to signal to the robot that a glass has been placed in a specific position. The user sets the position via a parameter in the node.

When the node is executed, it transmits to the python module this position which will change the bar list.

3.3 : The Bar class : Priority management and memory

I started using 2 files to represent this memory but this solution wasn’t satisfying because it’s not securised : two threads can access simultaneously to the file and even if I wasn’t able to measure this precisely, I'm sure that repeatedly opening and closing files isn't the best way to go. I decided to upgrade to centralise all my internal systems and secure access to it, I created the Bar class which is a static class. Figure 3.4 shows how the class attributes are structured. In addition, It has a lot of different methods to manipulate these attributes and it also has semaphores to manage the access to them and avoid interferences between threads.


```
glass_area = ['X','O','X']
bar_area = ['001','O','O']

waiting_list = ['002','003']
ready_list = ['001']
```

Bar class
Glass available on position 1
Position 2 is empty
Position 1 is occupied by order n°001
Position 2 and 3 are empty
Order ID 002 and 003 are waiting to be prepared. Priority management by Fifo
Order ID 001 is ready to be picked up by a waiter
ready = True : The robot is ready to prepare the next order from the waiting list
False : The robot is currently preparing an order
action = True : The robot is currently executing a node*
False : The robot can execute a new node*
priority = True : Given order waiting for being executied
False : Normal mode
(*) - Only for Pour(+), Get glass(+) and Give order nodes

Figure 3.4 : Explanation of attributes of the Bar Class taken from my technical note

‘Manage Order’ node first interacts with the *waiting_list* to add the ID of the new order and the thread waits until this ID is at the index 0 and that’s *ready* is set to True. When conditions are right, the thread sets *ready* to False and interacts with *glass_area* and *bar_area* to get a picking and a deposit position for a cup.

‘Pour’ and ‘Get Glass’ nodes set *action* to True when they are running. In this way, other nodes can’t start while *action* is not set to False. In addition to this, before setting *action* to True, these nodes check if *priority* is set to true. If yes, they are waiting because they don't have the priority. Indeed, only the ‘Give Order’ node can take the priority (like explained in 3.2). This nested system allows the robot to prioritise the execution of ‘Give Order’ over others. An important hypothesis of this system is that there can't be two servers waiting to pick up an order at the same time. This is a consistent assumption because there is only one place in the environment. When the ‘Give Order’ has finished, it releases the priority and deletes from the *bar_area* and *ready_list* the order which has been given.

At the end of an order preparation, ‘Order Ready’ sets *ready* to true to allow the next waiting thread from ‘Manage Order’ to continue and add the order’s ID to the *ready_list*.

Finally, ‘Restocking’ node changes the value of *glass_area* to add a new glass at a given position. This node allows the Sawyer to not stop after using all of the three starting glass.

3.4 : Performance measurement

As I said in the introduction of this chapter, I had to make a choice between using or not the Motion Interface provided by Intera SDK. I needed to define three criterias to compare my two versions :

- **Global Execution Time :** The rapidity of execution is essential for a bartender (customers don't want to wait too long before receiving their orders).
- **Precise movements :** Sawyer must be able to avoid spilling the glasses and bottles it is carrying.
- **Resilience** : A bartender should be able to adapt its task according to the environment (no water available, no cups, error in the trajectory).

Without Motion Interface - video here
This was the first version (1.0) that I did. I measured a mean execution time of 2.12 seconds to prepare one order. Movements were jerky, as the robot stopped at each crossing point. There was no path planification.

With Motion Interface - video here
With this version (1.5), I had a mean execution time of 1.07 seconds and movements were smoother. The main constraint is that this version required a lot more waypoints and sometimes, it refused to move (error detected whereas without this interface, the movement was possible). This posed a number of problems, particularly when it came to returning to the reset position, which was not happening and was preventing the other nodes from continuing correctly.

Thus, In terms of my two first criteria, the solution with the interface is much better. But I was obliged to improve my error management, to maintain resilience at the same level as 1.0, by providing some security behaviour if the robot doesn’t want to finish its trajectory. Indeed, I created a lot of waypoints and I forced the return in the reset position by creating a script parking.py which is executed after the end of the main script (get_glass.py or pour.py). This is why in the video, you can see that the robot stops after finishing a node and then goes back to its reset position. Most of the time, if an error occurs during the planification of the trajectory, the robot goes in a reset position and tries again.

This error management, that I added, also increases processing and communication time. This is why the robot sometimes takes a long time before starting to carry out an action. But this is still largely acceptable as the preparation time is divided by 2 with this method.

To check the accuracy of my task, I carried out a series of tests changing the position of the glass and the type of drink each time. The robot caught plastic cups and bottles and poured them perfectly into the glass 100% of the time (18/18). This was hardly surprising, given that the movement was repeated to within a centimetre of the way it was configured.

To conclude on performance, this application allowed Sawyer to manage multiple orders at the same time thanks to the Bar class. This already put the application slightly ahead of scenario expectations. Version 1.5 was definitely better in the 2 first criteria but resilience still can be improved in both versions. Nevertheless, this version had a really restrictive hypothesis, namely fixed positions, which was my priority for the rest of the project.

4. Upgrading the application to 2.0 with object detection

I could choose which of the features I was going to improve from those I mentioned in 1.2. Namely, the ability to make cocktails, use computer vision to retrieve glasses and bottles or a function to open bottles autonomously. I opted for object detection because in a real bar, bottles and glasses will never be in the same place to the nearest centimetre. Therefore, I needed to make the robot more adaptable and robust to changes in its environment. To achieve this, I used a model based on the YOLO (You Only Look Once) algorithm that can detect and localise multiple objects in an image. I customised this model to recognize the objects that were relevant for my task, such as plastic cups and bottles. I also created a new set of nodes called “Bartender advanced” which supplements the “Bartender Classic” by providing new nodes using computer vision.

In the first part of this chapter, I will explain how I created my dataset and selected my model in an optimal way. Then, I will describe the integration of this model in the application thanks to a new set of nodes. Finally, I will compare this version with the previous one on the basis of the same criteria defined in section 3.4.

4.1 : Building the Object Detection system with Yolov5

The first step was to create an object detection’s system. I needed to create my own dataset because the quality of the arm’s camera is not excellent, thus I couldn't find any existing dataset which fit my constraints. To carry out this task, I wrote an ROS script to capture images every 2 seconds from the camera and while the script was running, I took several angles by varying the perspective, brightness and position of the photos (plastic cup, Coke, orange and water). I estimated that a good size of dataset is around 100 images of each object to avoid overfitting and allow a suffisant learning. Besides, to improve the quality of my dataset and avoid overfitting, I resorted to what is known as data augmentation. In other words, I've increased the size of my database by integrating transformations of the raw images: rotation, cropping, changes in brightness, etc. You can look at my dataset at this link.

I chose to use Yolo because I thought it was the most suitable for me and future students who will work on it : a lot of documentation, integration of RobotFlow, faster and good accuracy. In addition, RobotFlow allowed me to easily label my dataset by drawing a box around my object on each image (see figure 4.1) and also to use data augmentation to rotate and crop images. Then, to choose the version of Yolo to use between v5, v6 and v8. I compared the execution speed. Indeed, yolov8 is way faster than yolov5 on larger models, but I didn’t need a huge model for my object detection because this is a relatively simple use case. And for small and medium models, yolov5 is the fastest.

I used Google Colab to train 3 different models : Medium, Small and Very Small. Medium has a better accuracy score 97% on the validation (against 93% for the Small and 87% Very

Small) but the Small is twice as fast : 0.5 second to analyse an image against 1s. I chose the Small due to that because 1s for one frame is too long for a bartender with speed constraint.

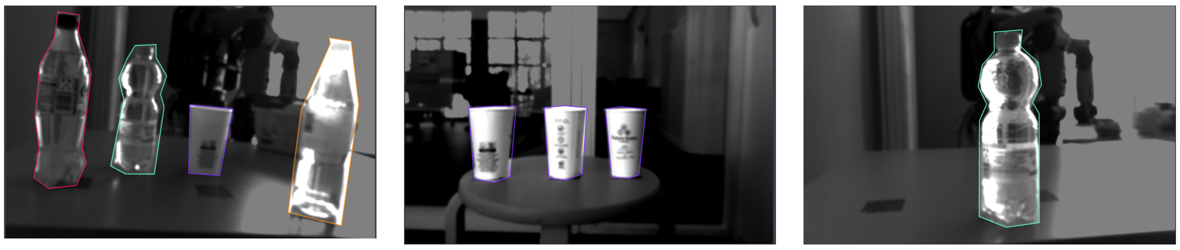


Figure 4.1 : 3 images from my datasets labelled with RobotFlows

My first idea was to position the robot's camera facing the area in which we wanted to detect bottles or cups. The robot then moves horizontally and stops in front of the first cup detected. But this scenario didn't match with the constraint of my model : the treatment time of an image is too important to do this in real-time (even 0.5s is too long). The second idea that I had was to position the robot each time at the same place and process as many images as possible for 2 seconds. After that, doing a mean between all images to define the 2D position and height of the object on the image. And then, estimate the 3D coordinate of the object (See section 4.2 for more details). Finally, my last idea was to try to implement vision-based robot control by taking an image, analysing and moving the camera robot to an estimated position in which the object would be in the centre of the next image. And repeat this action until an acceptable position. And then just catch the object with a linear translation. Unfortunately, I wasn't able to implement this last idea due to a lack of time.

4.2 : Creating the “Bartender Advanced” nodes set

I created 2 nodes : “Get Glass +” and “Pour +”, to replace “Get Glass” and “Pour” from the “Bartender Classic” set. But for this version, my development approach has changed. I wanted to focus on performance rather than the reuse of nodes. In other words, the objective here was to find the optimal way in accuracy and speed. The best way is to keep two separate nodes (not more, not less) instead of, for example, making a node to detect a particular object that returns the position, then a node that moves to that position to retrieve the object and then perform the appropriate action (pour or drop). In addition, this makes it easier to switch between fixed mode and detection mode because we use in both cases 2 nodes with the same functionalities.

As the communication process (node-red to rest-server to ros-container) of “Advanced” nodes is very similar to the old ones (see section 3.2), I only detail here the integration of detection in the ROS script. The script is the only major change here.

In this case, I used lists of waypoints only for the second part (deposit/pouring) of each movement, because the first part (taking object) depends on the detection. When the script is run it launches a thread to load the yolov5 model in a python class. During this time, the robot goes to the capture position. Once the loading of the model is finished and the robot is in position, it starts the detection function. The detection function is literally a 2 second wait during which the algorithm analyses a maximum number of images with the previously loaded model and stores in a table the coordinates of each object recognised in the image (in fact, the 2D coordinate of the detection box around the object), along with their label and a confidence score. Afterwards, the algorithm does a mean calculation for each object between all frames to have the most accurate position possible and, finally, extracts the position of the object we want. In case of there are more than one object corresponding, it extracts the one with the best confident score. At this point, the algorithm has 5 values (2d coordinate of right-bottom point of the box, 2d coordinate of top-left point of the box and confident score).

The aim was to transform these values into a horizontal and depth displacement. I researched some solutions to transform these 2D coordinates into 3D real word coordinates. One of them was to re-build the 3D environment, but it seemed too complicated for my purpose. So I chose to use optical projection equations that I learned during my course and completed it by some coefficient. As you can see in equations on figure 4.2, I needed to have intrinsic parameters of the camera to apply equations. After doing some research, I wasn't able to find these parameters. I therefore decided to do my own calibration in order to estimate them.

$$\text{depth} = \text{focale} * \text{real\_height} / \text{height}$$
$$\text{offset} = (\text{optical\_center} - \text{position}) * \text{depth} / \text{focale}$$

Figure 4.2 : Optical projection equations, focale and optical_center are intrinsic parameters, real_height is the real height of the object, height and position are calculated from the frame.

I had calibrated the camera using a chessBoard (that I printed) and openCv library to determine intrinsic parameters of the camera. I took a hundred pictures to ensure that the result was sufficiently accurate. Afterward, I applied the equations and added some coefficients to transform the estimated depth and offset in a correct movement of the robotic arm. It took me a long series of tests over several days to arrive at coefficients that I consider effective (see 4.3 to more details on performance).

Finally, when the offset and depth are determined by the algorithm, it calculates waypoints depending on them and sets a trajectory using the Motion Interface. Similar as in the old nodes, I integrated a first level of error management. If a trajectory error occurs, the robot will try to re-calculate the trajectory from a reset position or if it's impossible, go in reset mode and wait for an action from the supervisor.

4.3 : Performance measurement

I kept criterias from version 1.X to evaluate 2.0 : Resilience, Speed, Accuracy. I realised a video of this version that you can find on my website and here.

Unfortunately, resilience isn't better than before. Because of the introduction of visual detection, the robot can not be sure that it succeeded in taking the glass (or the bottle). This still poses problems for the robot's autonomy. In terms of speed, I measured a mean preparation time of 1 minute 15 seconds. It's a bit slower than the 1.5 version, but it is normal because of detection time and loading model time. I think it's still a good result and it's difficult to improve on it with such an environment and hardware constraint (one camera…). To finish on previous criteria, accuracy is, for me, not good enough with a 80% success rate (Tests Table in Appendix). Same as the speed, because of hardware constraints, it is hard to obtain a more accurate calculation with the method currently used.

On the other hand, in all the tests I carried out to measure accuracy, the model was always able to detect and position the boxes accurately. Catch errors were entirely due to small imperfections in the calculations based on the boxes, on the error rate of the position of the arm and its movements. This demonstrates the effectiveness of the model chosen and my method to extract these data, but also, the quality of the database which covers all possible situations.

5. Project evaluation: difficulties encountered, continuation and skills developed

In this final part of my report, I will evaluate my project, the difficulties I encountered, the continuation of the work and the skills I developed. I will first discuss the current constraints and limits of the project, such as the hardware constraint, the time constraints and persistent assumptions. Then, I will present a technical assessment of the application, focusing on its functionality, performance, reliability and autonomy. Finally, I will provide a personal assessment of my internship experience, highlighting my strengths and weaknesses and my professional and personal growth.

5.1 : Current constraints and limits

The biggest problem with the current application is that the quantity poured varies according to the type of bottle and the position of the glass in the preparation area. It’s normal because bottles are not always filled to the same level and can therefore flow more (¾ full) or less (¼ full) quickly at the same angle. To remedy this, a solution would be to replace the one-second wait that I implemented in the “pour” script, with a real calculation of the time and inclination based on the weight of the bottle and therefore its estimated filling percentage.

In addition, bottles are assumed to be opened and reusable. In other words, the robot doesn't throw away empty bottles and risks taking them again. This needs to be implemented for a fully automated version. Always with the detection, plastic cups may not be closer than half a centimetre to each other to avoid knocking over the others when picking one up.

When a critical error occurs, the robot stops working and must be reset by relaunching the application or executing the initialisation node. That’s not really resilient. Something that could be interesting is to implement in each node a waiting state if a critical error occurs in which the node will wait for an action from the supervisor. On the other hand, with object detection, the robot didn’t have a way to know if it really succeeded in taking a glass or not. If for any reason, the robot misses the glass that it has detected. It doesn't know and the rest of the preparation will pursue, and it will pour water over the supposed location of the glass, spilling water all over the work area.

5.2 : Technical assessment

To draw up a technical assessment of my work, I would need quantified objectives. Unfortunately, my only instruction was to go as far as possible. Throughout the project, I tried to maintain the following objectives: Clarity of code (easy to use/improve for people who will work on it later), Accuracy (working with liquids with a robot can be dangerous for it, so you have to be sure of every manipulation) and Speed (being the fastest as possible). Here's what I can say about them:

- *Clarity of code :* I think this objective has been fully achieved. Even if I could maybe optimise some functions/scripts, I've done my best to ensure that the documentation is fairly complete and the code tidy.
- *Accuracy :* It's for me the least satisfying point. I did my best with the current hardware and given the brevity of my internship to achieve a 80% success rate. To improve it significantly, we need to build a more suitable environment and perhaps add an extra camera to enable triangulation.
- *Speed :* The speed of the application can undoubtedly be improved. In particular the communication part, which I find simplistic, but that wasn't the main objective of my project. So I preferred to optimise my scripts (resource needed, execution time), which I'm much happier with.

In terms of functionality, I build an operational basis for my application but also for any other application that could be developed on the SAWYER with node-red. I implemented three different sets of nodes with different levels of abstraction for real flexibility of use. I built an internal system to manage priorities and memory entirely safely and securely. I integrated image detection for a more realistic application and adapted my error management to handle trajectory errors. **I feel I've done a good job over the last 3 months and I'm fully satisfied with the results I've achieved.**

Unfortunately, the work on waiters (Pepper Robot) was unsuccessful. In consequence, I was not able to make a complete demonstration of the interaction with a Pepper but I made a simulation of this interaction and the result was totally what I expected. At this point, Sawyer is able to work with a simulation of a waiter, a robot waiter or a human.

To conclude, I had to scale down my ambitions: no cocktail for this version of The Bartender Sawyer. I preferred to try to improve each version with alternative methods, a precise analysis of results and a determination of the better one. In consequence, even if the robot can detect objects, it can't work with total autonomy for now. It needs a supervisor who is responsible for accompanying the robot (replacing bottles, checking that glasses are correctly gripped). According to this, there are some areas of improvement :

- security : I think security and communication are the two most important points to improve in further works. Currently there is a little bug : when you stop the app and relaunch it directly, the ros-container add an error (listen.py don't succeed to connect to the right communication port)
- Safety : Add some verifications after the Get Glass or before the Pour to be sure that there is a glass on the bar and not pour on the table.
- Robot environment : It would be interesting to have something really more like a bar (shelves, drinks case…) and adapt scripts in consequence.
- New features : Implements some new features like the capacity to make cocktails or even to open bottles by itself. For cocktails, you can maybe find a way to estimate the weight of the bottle from the force of each joint to take the bottle.
- Error management : Currently, if a critical error occurs during a preparation (no cups/bottles detected, not able to follow the trajectory...), the robot stops its flow and needs an action from the supervisor (rebooting or reset). It can be interesting to add a system that allows the robot to restart itself in this case without a full reset.

5.3 : Personal assessment

I loved this internship. The subject and the freedom I was given in terms of organisation and technical choices really allowed me to thrive in my work. I was able to get a feel for research and life in a laboratory, which reinforced my desire to continue with a thesis. Thanks to this freedom in my timetable, the internship also allowed me to take time for myself, with lots of sports, time to improve my English, read philosophy and develop my own projects. This allows me to learn more discipline and autonomy in my work, even if it was already one of my strengths.

In terms of skills, I've improved both my spoken and written English (oral expression was my biggest weakness). I was also able to delve deeper into the concepts of robotics that I found too abstract in my courses and to apply my knowledge of image processing and even optics.

Webography

Solutions : Rethink Robotics : https://support.rethinkrobotics.com/support/solutions

Install Docker Engine on Ubuntu | Docker Documentation : https://docs.docker.com/engine/install/ubuntu/

Documentation : Node-RED (nodered.org) : https://nodered.org/docs/

ultralytics/yolov5: YOLOv5 🚀 in PyTorch > ONNX > CoreML > TFLite (github.com) : https://github.com/ultralytics/yolov5

Socket.IO : https://socket.io/

PyTorch : https://pytorch.org/

Roboflow : https://roboflow.com/

Flask : https://flask.palletsprojects.com/en/2.3.x/

Contributions :

My RobotFlow database : https://universe.roboflow.com/bernat-loan/bartender-sawyer-vision

Sawyer Bartender ROS package : loanBRNT/sawyer_vision_bartender

Simulation Video 1.0 : https://youtu.be/lT4WaLM3AWw

Simulation Video 1.5 : https://youtu.be/8MbCvps2-aU

Simulation Video 2.0 : https://youtu.be/Euf1ZaoABLU

Global Simulation : https://youtu.be/rq2TTOWhma8

Appendix

Tests results for bottles detection

Water	Catch	Pouring	Time
1	y	y	37,1
2	y	n	37,3
3	y	n	37,1
4	y	y	36,8
5	y	y	36,5
6	y	y	36,9
7	n	-	37,1
8	y	y	36,4
9	y	y	38,6
10	y	y	38,22
	90%	**70%**	**37.02**

Coca	Catch	Pouring	Time
1	y	y	38,2
2	y	n	40
3	y	y	39,8
4	y	y	38,24
5	y	n	37,9
6	y	y	39,9
7	y	y	38,6
8	y	y	37,9
9	y	y	38,7
10	y	y	38,1
	100%	**80%**	**38.73**

Orange	Catch	Pouring	Time
1	y	y	37,2
2	y	y	35,9
3	y	y	38
4	y	y	38,9
5	y	y	37,1
6	y	y	36,1
7	y	n	39
8	y	y	38,7
9	y	y	38,1
10	y	y	36,2
	100%	**90%**	**37.52**

Tests results for cups detection

Cups	Catch	Time
1	y	24
2	y	23
3	y	23,1
4	y	22,5
5	y	23
6	y	22,9
7	y	23,7
8	y	22,8
9	y	23,3
10	n	22,8
11	n	23,1
12	y	23,3
13	y	23,2
14	y	22,9
15	y	25,3
16	y	24,7
17	n	24,5
18	n	22,9
19	y	24,9
20	y	25,8
	80%	**23.585**

<u>Technical note</u>

Bartender sawyer with Node-red

By Loan BERNAT for my M1 internship. Contact : loan.brnt@gmail.com or +33651229452

1 - Description

A bartender application for sawyer robots using Node-Red, Docker, Flask and Yolov5. The application has two modes of use : Classic and Advanced.

- With Classic, it uses fixed positions marked with scotch.

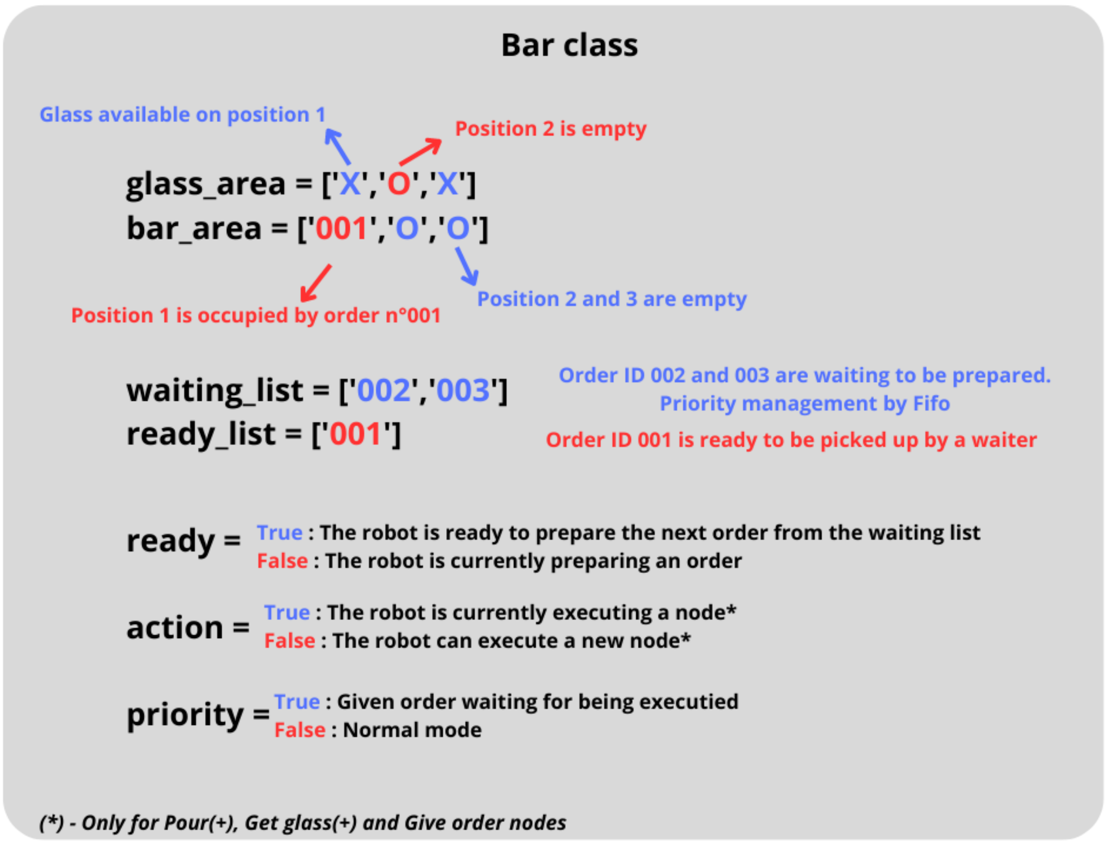
- With Advanced, it uses object detection with a personal Yolov5 datasets. You can find a complete version of it (scripts and dataset without labels) on my github page or only the dataset labelled on the robotflow website. I integrated in the application only useful scripts and models from it.

You can mix advanced and classic nodes to adapt the application to your constraints.

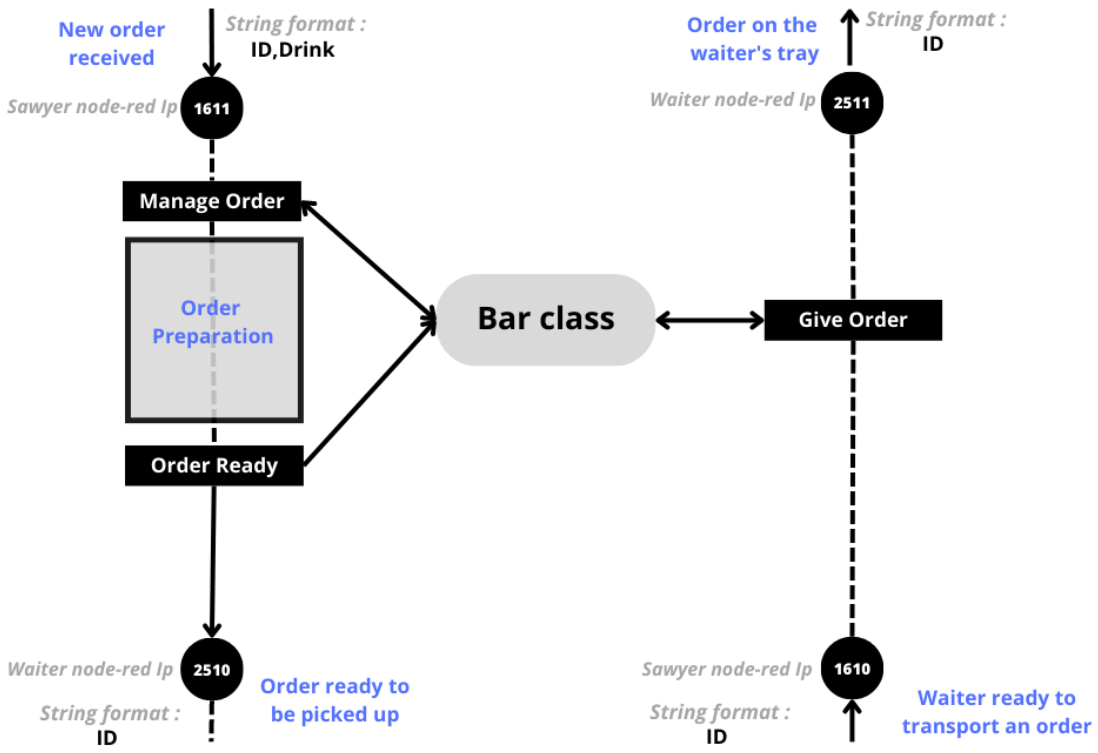
Versions:

- Fixed position V1.0 : Youtube Video
- Fixed position V1.5 : Youtube Video
- Object Detection V2.0 : Youtube Video

In both case, the application is built around the Bar class which gives at the robot a "memory" for orders and a sense of priority : It stops preparing the order it is currently working on if a waiter asks for picking up another one, to give the requested order to the waiter as quickly as possible and limit the waiter's loss of time. This is how the Bar class is structured :


Bar class
Glass available on position 1
Position 2 is empty
glass_area = ['X','O','X']
bar_area = ['001','O','O']
Position 1 is occupied by order n°001
Position 2 and 3 are empty
waiting_list = ['002','003']
ready_list = ['001']
Order ID 002 and 003 are waiting to be prepared. Priority management by Fifo
Order ID 001 is ready to be picked up by a waiter
ready = True : The robot is ready to prepare the next order from the waiting list
False : The robot is currently preparing an order
action = True : The robot is currently executing a node*
False : The robot can execute a new node*
priority = True : Given order waiting for being executied
False : Normal mode
(*) - Only for Pour(+), Get glass(+) and Give order nodes

Sawyer communicates with waiters using TCP nodes from Node-Red. See the diagram belows :



IMPORTANT : The IPs shown in the diagram aren't the IP of the robots. It's the IP of the Host machine on which flows are running. You can run everything on your computer, so the IP to use into all TCP will be your localhost. Or you can work in a team, and replace localhost by the IP address of your teammate. If you don't know your IP you can use `hostname -i` or look into the setting of your virtual machine.

For a complete description of each nodes used, please open the documentation directly included in the node-red project (run project -> open node-red interface -> documentation widget)

2 - Installation and Configuration

The project was developed on Ubuntu 20.04. You can use a virtual Machine but make sure that the bridge mode is enabled.

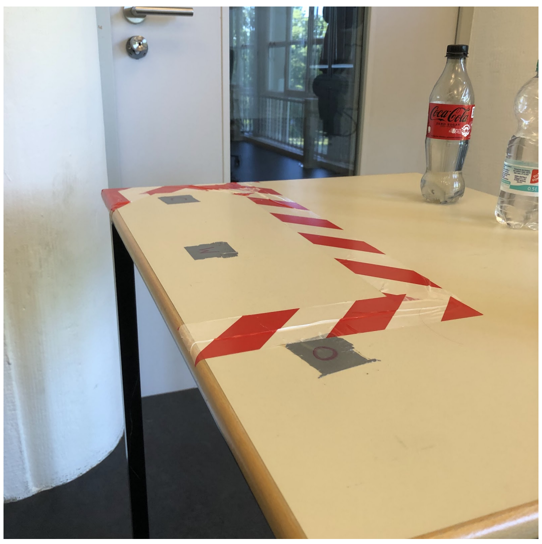
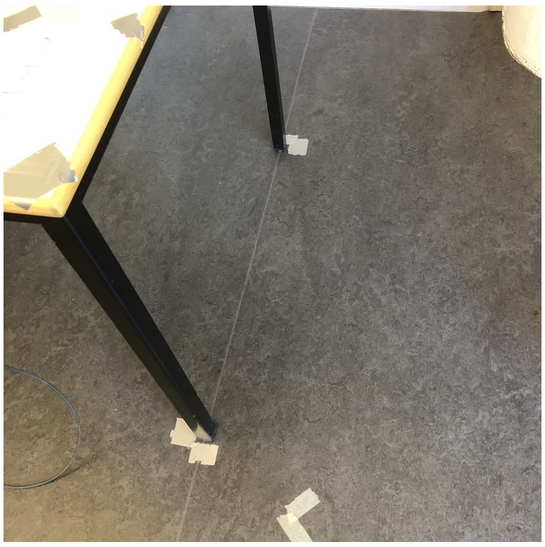
The first step is to clone this repository. Then, you will need to change some parameters in the `.env` file.

- ROS_IP = your host ip address (hostname -i or in network settings)
- ROBOT_IP = Verify that the ip is correct

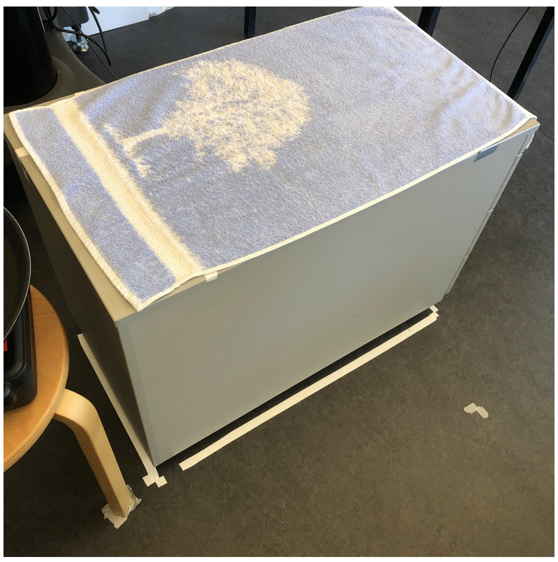
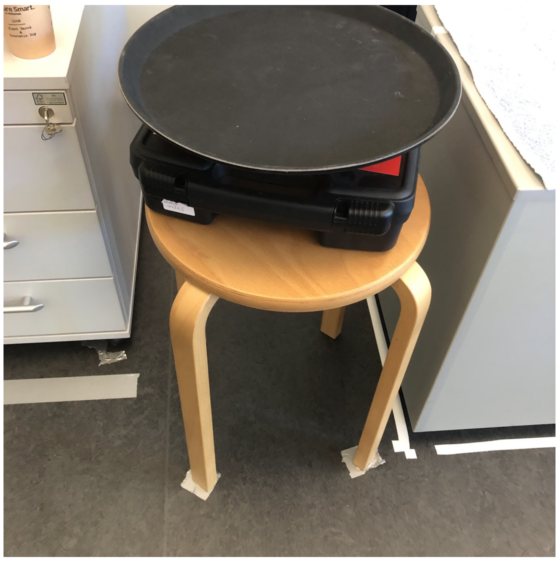
After this, make sure to already have Docker installed. Otherwise, check instructions at this link to install docker on ubuntu and then do the linux post-installation here to allow you to use docker without sudo.

Then you can build the project with `sudo ./buildContainers.sh`. It's going to take a while, so you can start preparing the robot's environment.

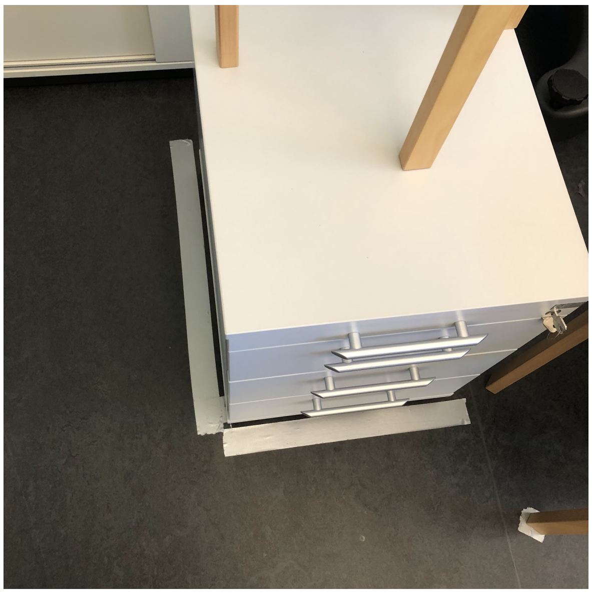
Bottles area : The table should be positioned like in the 2nd picture. The three grey tapes on the first picture mark the position of each type of bottle ("C" for Coca, "W" for Water, "O" for Orange). The red/white tape delimits an area representing the camera's field of view in object detection mode. If you are using this mode, bottles need to be inside of this area to be detected.

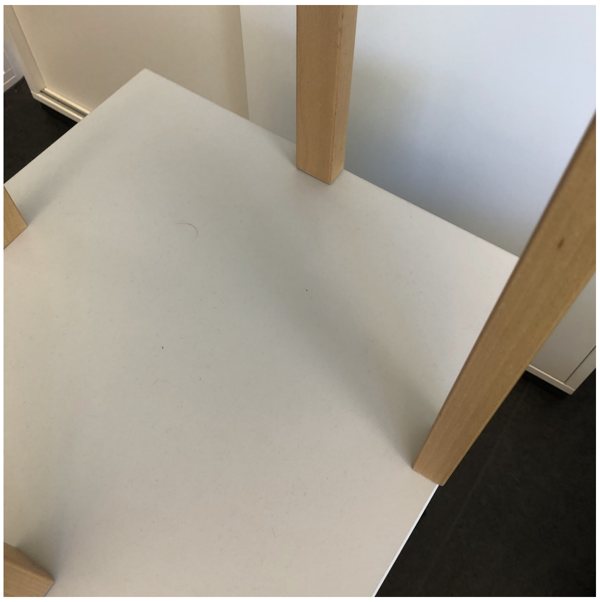



Preparation area : Position the first low table as shown in the first picture. The stool on the 2nd simulates the location of a robot picking up an order. The back legs should be placed on the tape marks and the height of the tray should be approximately 65 cm from the floor.

Glass area : The second low table should be positioned as shown in the first picture and the tool on it should have 2 its two legs on the border as shown in the second picture



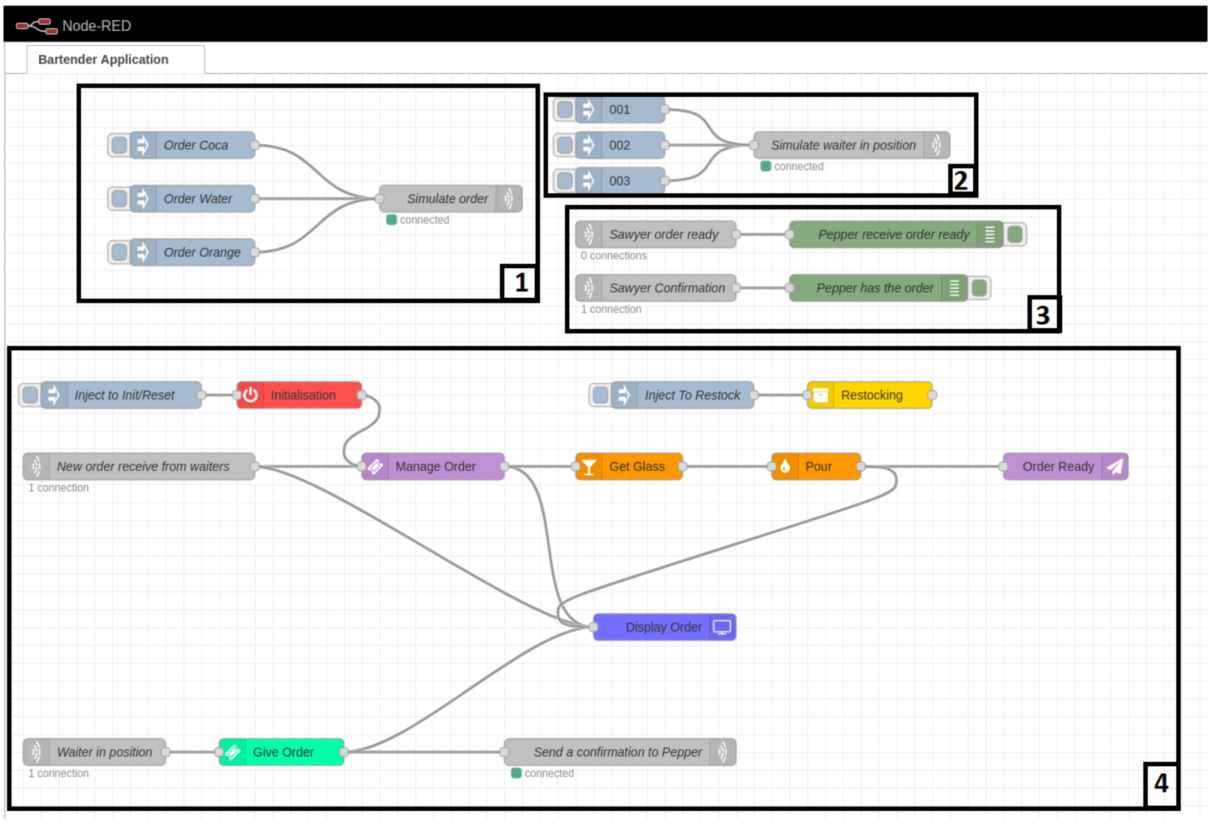


3 - Usage

Turn on the sawyer and make sure that it is in SDK mode (Black screen with SDK written on it). If not, turn it off. Then, turn it on again and press CTRL + F on the keyboard until a setting menu appears. Here, select 'boot in SDK' and reboot again.

After following the Installation instructions. You can build the project (if it's isn't already done) with `./buildContainers`. Then, you can launch the project with `docker-compose up`.

If you have some errors, contact your professor. If everything is ok, you can open the Node-Red Interface by clicking on the link in your console : localhost. And you should have a flow that looks like this :



Some importants things :

- Always verify that the robot environment is safe and the emergency button is accessible.
- Use the *initialisation* node as a reset and start protocol. That allows the robot to go back to its "parking position" before running any programs and calibrate its gripper.
- Never use directly opened bottles. Water can damage the robot. Always, make some tests with closed bottles before.

And now, you are ready to create your own application using Sawyer specific nodes!
Enjoy

4 - How to Create your own Nodes

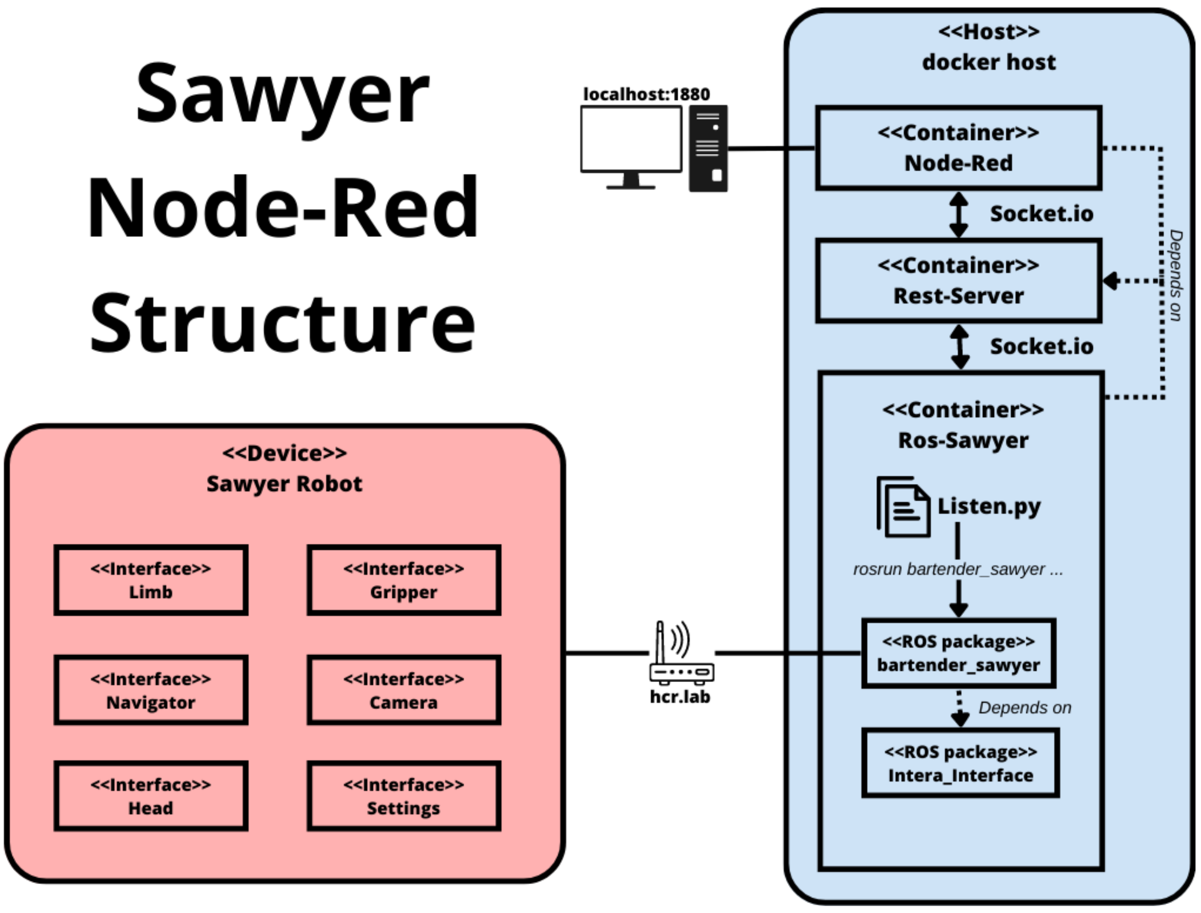
4.1 - How the project work

First of all, you need to understand the application. If you are not familiar with Docker, Flask, ROS and Node-red, I'd advise you to find out a bit more about them first.

So, the project is divided in 3 containers :

- **node-red** with all node-red stuffs (Front end),
- **ros-sawyer** in which you will find my ROS package bartender-sawyer and the original ROS package of sawyer intera_interface. This container allows me to run scripts on sawyer with a command system (Back end).

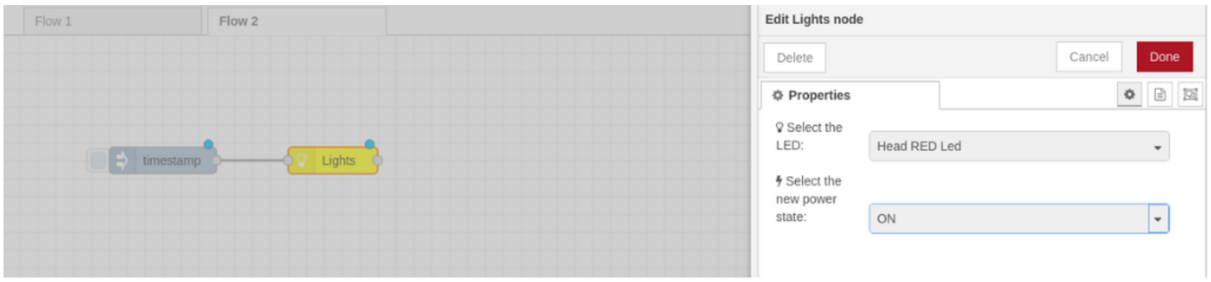
- **rest-server** which is the link between Node-Red and ROS. It receives messages from the Node-RED container with socket.io, handles the message and sends the right command to the ROS container.



The communication between all these containers is simple because they all share the same IP address as your computer. That's why the bridge mode is important!

Parameters, dependencies and environment variables are defined in DockerFile of each container and in the docker-compose.yml

To resume, this is an example of what's going on when you use the light node :

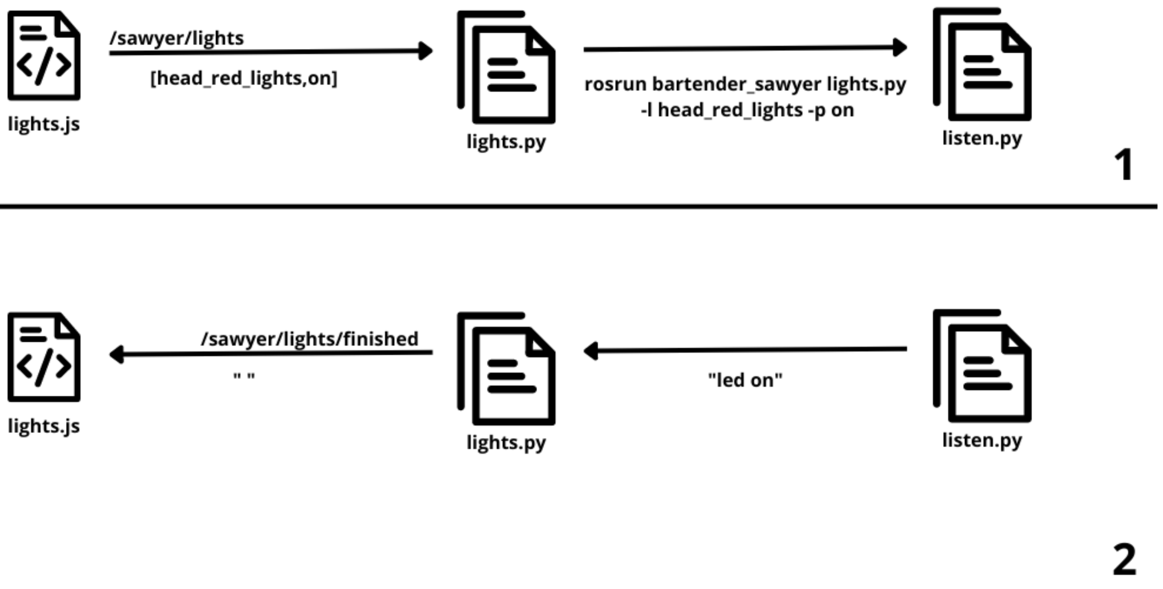


Edit Lights node

Properties

Select the LED: Head RED Led

Select the new power state: ON



Now that you understand the overall structure of the project a little better, it's time to code!

4.2 - Create the ROS script

The first thing to do is to create your ROS script. The better way to do that is to have an external folder. But for this, you need to have ROS-noetic installed and some other parameters configured. I won't go into details here, I recommend that you follow the steps on the official Rethink Robotics website, taking care to use the commands for Ubuntu 20.04 and Ros:Noetic.

You can find the python SAWYER API here.

If your script needs params, use `argparse`.

When you finish to code and test the script outside the project, copy your script into `ros-sawyer/src/bartender_sawyer/scripts/` and add your script to the `ros-sawyer/src/bartender_sawyer/CMakeList.txt`.

If you need a particular library, you need to modify the dockerfile to install them into the container.

4.3 - Create the rest-server package

For this part, you will need to create a new python file or just add a function into an existing one (if your node is linked to an existing one) inside this folder `rest-server/package/sawyer/endpoints/`.

In your function, you will need to handle different things :

- Choose the "topic" message which will activate the function when the rest-server receives a message on it. The convention that I choose to follow is '/sawyer/nom_event'.
- The communication with the ros container. So what command do you need to run your script? With parameters?
- What parameters do you need to receive from the node-red containers.

When you have finished, if needed, add your file to `rest-server/app.py`.

4.4 - Design the node

To create a node, you need **4** files into `node-red/nodes/`:

- my_own_node/my_own_node.js : The main file of your node. Inside, you will define what you want to send to your package in the rest_server.
- my_own_node/my_own_node.html : In this one, you define what your node will look like, as well as its parameters.
- my_own_node/locales/EN-US/my_own_node.json : JSON file
- my_own_node/locales/EN-US/my_own_node.html : Documentation file

I recommend you to take inspiration from one of my existing node to understand the structure of each file. You can also take a look at this link : Creating Node.

After creating and writing all these files. Don't forget to add your node to the package.json!

Now, just re-build the project and... **Congrats! You have created your first Node**

6 - Acknowledgment

This project was inspired by the work of Kai Ruske and Michel Weike.

7 - Project status

Ended

8 - Improvements

SECURITY : I think security and communication are the two most important points to improve in further works. Currently there is a little bug : when you stop the app and relaunch it directly, the ros-container add an error (listen.py don't succeed to connect to the right communication port)

SAFETY : Add some verifications after the Get Glass or before the Pour to be sure that there is a glass on the bar and not pour on the table.

ROBOT ENVIRONMENT : It is very simple for now, but it would be interesting to have something really more like a bar (shelves, drinks case…) and adapt scripts in consequence.

NEW FEATURES : Implements some new features like the capacity to make cocktails or even to open bottles by itself. For cocktails, you can maybe find a way to estimate the weight of the bottle from the force of each joint to take the bottle.

ERROR MANAGEMENT : Currently, if a critical error occurs during a preparation (no cups/bottles detected, not able to follow the trajectory...), the robot stops its flow and needs an action from the supervisor (rebooting or reset). It can be interesting to add a system that allows the robot to restart itself without a full reset.

CLASSIC FEATURES : Create some low-level nodes to command the Head rotation, the cuff, camera. Improve movement nodes by doing something more permissive.