Bin Picking Cell Control

Background

You are tasked with planning and programming the control of a Robotic Cell for bin picking application at the warehouse. A cell is a secured environment in which a robotic arm safely operates. Such a cell may contain doors to allow humans to access the safety environment, as well as a stack-light to communicate the state of the system. The workers of the warehouse will interact with the Robotic Cell using a Human-Machine Interface (HMI). The image below illustrates one example of these Robotic Picking Cells.

As you don’t have a cell like this at home, we won’t ask you to implement the whole system to control it. However, you’ll have several sub-tasks related to this context. It won’t be necessary to simulate the robot or cameras, we’re only interested in the other parts of this scenario.

Tasks

1. API Call handler

When we deploy a robot to a customer, it's quite normal that we use API calls to manage the communication between the Warehouse Management System (WMS) and our Robotic Cell. We receive a request from the WMS specifying what we have to do, and we respond to it once the picking is done.

Please implement the server at the WMS side, which sends the requests in this format:

```
HTTP POST http://<ip-adress>:8080/pick

JSON
Request Body
{
   "pickId": 123,
   "quantity": 4
}
```

You can implement it to be synchronous or asynchronous, up to you. Once you can send the pick requests, implement the client side as well. It receives the request, performs the pick (in a fake way, as there's no robot), and generates the response in the following format:

```
HTTP POST http://<ip-adress>:8081/confirmPick

JSON
Request Body
{
   "pickId": 123,
   "pickSuccessful": true,
   "errorMessage": null,
   "itemBarcode": 123
}
```

Feel free to generate an `errorMessage` in case the `pickSuccessful` is false, but this is not necessary. The `pickId` should be the same one as the request, and the `itemBarcode` should be the number provided by the barcode scanners. The `pickSuccessful` should be true if the robot was capable of picking and placing the item.

2. Scanner ROS 2 node

It's very common that we have to scan the item's barcode while picking it. In these cases, we use one or multiple barcode scanners to read the number, and then make it available for the system.

Implement a ROS 2 node for this task that constantly publishes a barcode of 5 random numbers. Additionally, implement a service that doesn't take any value as request, but returns the most recent barcode scanned by the scanner (in this case the number returned by the service is equal to the one that has been published by the topic). You should use this number as part of your response.

3. Door handle ROS 2 node

All cells are built with at least one door to allow the workers to access the cell. The system that manages the whole cell should be aware whether the door is open or closed. In case it's open, all the requests should be responded with `pickSuccessful` as false, as when the door is open the robot is not allowed to move at all.

Implement a ROS 2 node for this task that constantly publishes the state of the door handle of the cell. The value published should be boolean, in which is true when the door is closed, and false when the door is open. To mock the fact of the door being opened or closed, you can: either kill the node and run it again to change the boolean value it publishes; or you can implement a service that simply switches the current value that is published.

4. Emergency button ROS 2 node

In some cases, the workers must press the emergency button (e-button) to force the robot to stop immediately. Once is pressed, the robot stops and the cell doesn't go back to the operational mode until the e-button is released. The system that manages the whole cell should be aware whether the e-button has been pressed or not. If so, all the requests should be responded with `pickSuccessful` as false, as when the e-button is pressed the robot is not allowed to move at all.

Implement a ROS 2 node for this task that constantly publishes the state of the e-button of the cell. The value published should be boolean, in which is true when the button is pressed, and false when it's not. To mock the fact of the button being pressed, implement one service that changes the state of the button to true (which mimics the button being pressed), and another service to reset the button back to false, as if the button has been released.

5. Stack-light ROS 2 node

It’s quite common in the industry that machinery are equipped with a stack-light, indicating the current state of the system. For our robot picking cell, we also use a stack-light to let the workers know how the system is performing.

Implement a ROS 2 node for this task that constantly publishes the state of the stack light of the cell. The published value should be 0, operational, if the door is closed and the system is receiving requests. It should be 1, paused, if the door is open and the robot can’t perform any movements. It should be -1, emergency, if the emergency button has been pressed.

6. HMI

The last peace of the puzzle is to allow the workers to see what’s happening in the touchscreen available at the picking cell. The HMI is quite useful to help the workers to understand if there’s any problem in the cell or if the picking process is going smooth.

Implement a HMI based on your preference. You can use a browser-based application or any other framework/library to display the information. For this moment, we don’t care about style or beautiful level. However, the information should be consistent to what is happening and updated in real-time.

In your HMI, you must show:

- The request info
- The response info
- The state of the emergency button
- The state of the door handle
- The state of the stack-light:
  - If the value is 0, please draw a green shape (circle, squared, etc)
  - If the value is 1, please draw a yellow shape
  - If the value is -1, please draw a red shape

Requirements

1. **Project Submission**
   - Submit the complete project as a **ZIP file**.

- After receiving the task, you have at maximum 5 **days** to submit the task. We care about both the speed and quality of your submission.

2. **Version Control**
    - Your project must include a version control system (e.g., Git). Please also include a README file describing how to setup/install your solution and run in a new machine (extra libraries/packages to bin installed, CAD files/meshes to be downloaded, etc). The version control could be just local, it's not mandatory to make it available online. If you decide to upload it online (on GitHub, GitLab, etc), please make the repository private and ask Sereact who should be invited to your repo.
3. **Video Demonstration**
    - Provide a screen-recorded video demonstration. Please do not record yourself or any private/sensitive data.
4. **References to Open-Source Code/Resources**
    - You may use open-source repositories to make your life easier, but ensure to mention them in the README and understand how they work. It's not mandatory, but it would add value to your solution if you could explain why your decision behind your choice, i.e., why you have selected a certain repository over the others.
5. **ROS Compatibility (Optional)**
    - While not mandatory, please use **ROS2 Humble**, as ROS Noetic is about to reach EOF.
6. **Docker (Optional)**
    - Even though it's not mandatory, feel free to provide your solution compatible with Docker.
7. **Decision-Making**
    - If there are any ambiguities or unclear instructions, you are free to make reasonable decisions. Do not wait for our responses. However, please provide all the information about these moments, such as what was the doubt, your decision, and why your choice was the best one on your point of view.