Høgskulen
på Vestlandet

HVL Robotics Lab

Robtino Guide

Submitted by:
Ozan Gülbaş

Lecturer: Associate Professor Gizem Ateş Venås,

6.11.2024

Contents

1 Robotino

*Robotino is a mobile robot system from **Festo Didactic**. Robotino provides all the sensors, actuators and software interfaces you would expect from a modern state of the art mobile robot system. An easy to use REST-API(requires OS V4 image) and a C/C++ library let you access all sensors and actors of Robotino in a network transparent manner.*[**6**]

The Robotino itself is a computer that you can install an operating system, mostly Linux however in the wiki it's said Windows installation is also possible, and has a variety of integration with default development systems, which we will focus on the Robot Operating System (ROS) through this documentation.

First, as said above, Robotino has its hardware, which we can install an operating system, however, the Robotino libraries and software need to be installed to use the robot's peripheral devices, sensors, motors, etc.. This Robotino software and libraries are closed-source, meaning we cannot modify or clone the related software, however, the Robotino API2 library and Robotino Rest API are the interface tools provided by the Robotino team for people to integrate Robotino into their projects.

1.1 Robotino API2

Robotino API2 is a C/C++ library that allows to interface the Robotino's sensors and actors from another program [**6**]. This library may be the most important for our purposes because the ROS2 library (*RTO-ROS 2, Rahul-K-A*) [**11**] we will use for developing ROS applications with Robotino is actually a bridge between this library and our ROS packages.

1.2 Robotino Rest API

This is Robotino's web interface based on a RESTful server listening on port 80, which we can use to interface through the web. RESTful means that the server uses HTTP (Hyper Text Transfer Protocol) for sending and receiving data and REST is an acronym for REpresentational State Transfer.

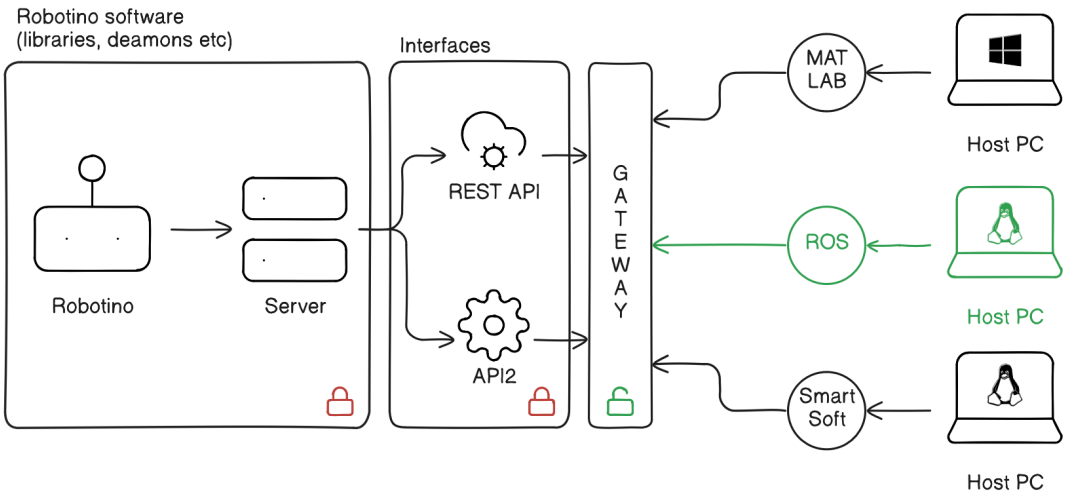

Figure 1.1: Robotino Application Development Diagram

Now we have an understanding of how to develop applications with Robotino, in the next section we have given a task and we will try to implement it using the knowledge we have.

1.3 Notes on Robotino OS

When I started working on Robotino, it had the Ubuntu 18.04 server version installed in it. However, somewhere in the testing and debugging things went wrong and I had to reinstall Robotino's software. During this process of tearing your hair out and questioning every atom in the universe, I have gained some knowledge about Robotino's operating system (therefore Ubuntu and Unix) and I will pour that information here so that in the future when you are reading this, they might be helpful and save some hair for you.

1.3.1 Robotino does not have access to the World Wide Web

Robotino has an internet connection with the Robotics Lab's wifi, however, it is mainly for its connection with your ROS Nodes, or other development environments. If you ever need to SSH into Robotino and try to install some packages, most probably you end up with errors about unresolved mirrors, and failed fetch requests to those mirrors. Don't worry though, this one has an easy fix, **HOWEVER, *you should know that the Robotino's software is outdated, thus, if you install new packages or upgrade existing packages, the***

dependencies may break the Robotino's core packages by updating their dependencies.

The Fix

1. Check your `/etc/reslov.conf` file. Normally, Robotino only has access to Festo's chosen nameservers. Thus, some packages may be out of reach. Add Google's nameservers at the bottom of your configuration:

   ```
   nameserver 8.8.8.8
   nameserver 8.8.4.4
   ```

   Remember, `resolv.conf` *file resets on every boot, thus you may need to do this again.*

2. Check your nameservers by pinging google.com: `ping google.com`. If you get **Name or service not known**, try to ping 8.8.8.8 or 8.8.4.4. If you get **Network is unreachable**, most likely you need to add a default gateway.

3. Check if you have a existing default gateway with: `ip route`. If nothing is showing up with `default` or `0.0.0.0`, add a default gateway with:

   ```
   sudo ip route add default via <gateway_ip>
   ```

   You can use the gateway IP of any device connected to the same network.

Because Robotino's software is outdated, the core packages need specific versions of some packages. Furthermore, some of the packages, like **librealsense**, may update itself when you do basic `apt-get update; apt-get upgrade`. To prevent this, I did **hold** some packages that gave me trouble so it might not be a problem for you, however, if you had a similar error caused by a different package, this can give you some idea about the problem.

1.3.2 You can connect a display if you cannot SSH

Robotino has Ubuntu 20.04 desktop installed in it, so if you cannot SSH into Robotino, you can connect a display to access Robotino and look for the problem.

Pro Tip: *Try to ping Robotino before trying SSH. Sometimes, Robotino needs a nudge to start its ssh-agent. I don't know why, it's just experience :D*

1.3.3 You can connect to Robotino form any other network

There is a laptop hidden in the Lab that is used as a server and it has Proxmox installed in it. I have created a virtual machine inside Proxmox called "Avatar" and installed the necessary software to connect Robotino and control it with the GUI I wrote. To use it:

1. Install Tailscale to your computer and connect to hidden laptop. I don't remember the exact steps but you can ask for a guide to do it.
2. Head over to the port 8006 and login to Proxmox.
3. There you will see a virtual machine named "AvatarUbuntu22.04". Start the virtual machine.
4. Once you have the access to the virtual machine you can connect to the Robotino like you are in same network (because you have access to a computer(Proxmox VM) through a VPN(Tailscale) that is in the same network with the Robotino).

1.3.4 Robotino has a backup!

If you made a chain of mistakes and cannot revert it, no worries! Robotino's working Ubuntu 20.04 image with Python 3 is installed inside a USB Drive in the lab. I cannot know its whereabouts in the future but you can ask. I have used the `dd` command to copy Robotino into USB and you can search for how to install the copied image back because I never did it. Here is a link that might be useful. [**3**]

1.3.5 Clean Start

If the backup didn't work or for some reason, you have to reset everything here is what I did.

1. Follow the steps provided by Robotino Wiki on Robotino3 USB restore [**5**].
2. This gives you the Robotino with SmartSoft installed on Ubuntu 16.04. By default, it will start a bunch of terminals for SmartSoft daemons, you can close them.
3. Next, connect the Robotino to the internet and add the necessary mirrors for the Ubuntu version upgrade.

4. If you get errors about the Ubuntu version upgrade due to some dependencies, remove those packages that have these dependencies.

5. Continue upgrading Ubuntu to your desired version

6. Depending on your version install Robotino packages. **robotino-dev, robotino-deamons, rec-rpc and robotino-api2** are the mandatory ones, you may skip the other packages.

7. If a package for your Ubuntu version doesn't exist, install the latest one.

8. Don't forget to follow the instructions on the "Ubuntu XX.XX modifications" pages while installing the packages [**9**]

9. Install the related versions of the packages to your host PC as well.

This is what I did while I was trying to understand what I was doing, however, I believe instead of steps 1-5, you can directly install the desired Ubuntu version and then install the Robotino packages to make Robotino work. It's up to you to try it, I didn't have time to test this but If I needed to reset everything again, I would give this a go.

2 TASK1: Make Robotino Safe

The first task is to make Robotino crash-safe when teleoperating so that when a teleoperate command is sent through the network the robot somehow overrides the command received if it exceeds the safety conditions determined by us using the distance sensors around the robot. We can consider several options to achieve this kind of goal.

2.1 Option 1: Safe Teleop Package using ONLY ROS

This option is the first thing that comes to mind however, its performance is highly dependent on the network. The reason for that will be more clear as we explore the implementation further.

The ROS 2 library we use already has a teleoperate package we can use as a basis. When we look at the source code of the package, we see that it is quite straightforward; get keyboard inputs, map them to a Twist message, and publish the message to the `/cmd_vel` topic. We can create our own safe teleoperate node that publishes to the, for example, `/safe_cmd_vel` topic and we would be halfway down with the problem however, the other half of the problem still remains; getting the sensor data and intercepting the received message.

When we look closer at the source code of the ROS 2 library, to find how to control motors, we see inside the `rto_node` package, there is a node called 'OmniDrive' which is exactly what we are looking for. When we inspect the source code, we see that it subscribes to the `/cmd_vel` topic and deconstructs the received message to pass it to a function called `SetVelocity()`. This function comes from the Robotino API2 library, and it is a function to drive Robotino's motors. More info about the function can be found in the API reference but basically, it takes three values; velocity in the x direction (vx), velocity in the y direction, and angular velocity (omega). With this information, we can also create our `safeOmniDrive` node and write our safety logic considering the values from the distance sensors, which brings the last part of the problem, getting the sensor data.

Using the same logic above, we see that in the source code, there is a node called `DistanceSensorArray` that gets the information from the sensors us-

ing API2, with the `distancesChangedEvent()` function, and publishes to the `/distance_sensors` topic. So we can subscribe to the `/distance_sensors` topic in the `safeOmniDrive` node to get the necessary sensor data.

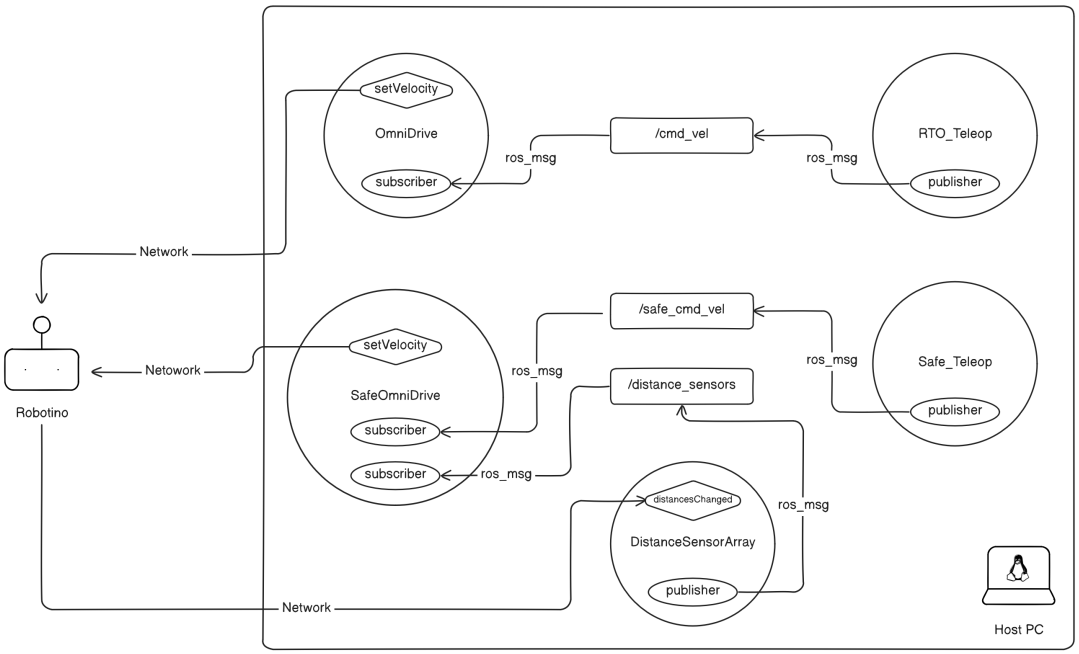

Figure 2.1: ROS Implementation diagram of Option 1.

This approach is perfectly fine with the logic however, there might be some performance issues when it is implemented in real life, mainly because of the network dependency of the solution.

2.2 Option 2: Using a custom API

This approach is a little bit less intuitive because it involves different tools to accomplish the same goal but the result is expected to be more robust as it carries the safety logic into Robotino itself, isolating it from the network problems, partly.

In this approach, the idea is to run a custom server on Robotino, beside its already running server, receive the distance sensors data, and control the motors directly on Robotino, only receiving the teleoperating commands over the network. The type of the server can be a TCP/UDP server or a Web server depending on the feasibility, however, since this approach is much harder to implement, and has the risk of not being that much different than the first option's performance,

its set aside as a second option. The implementation is visualized for easier understanding in Figure 2.2.

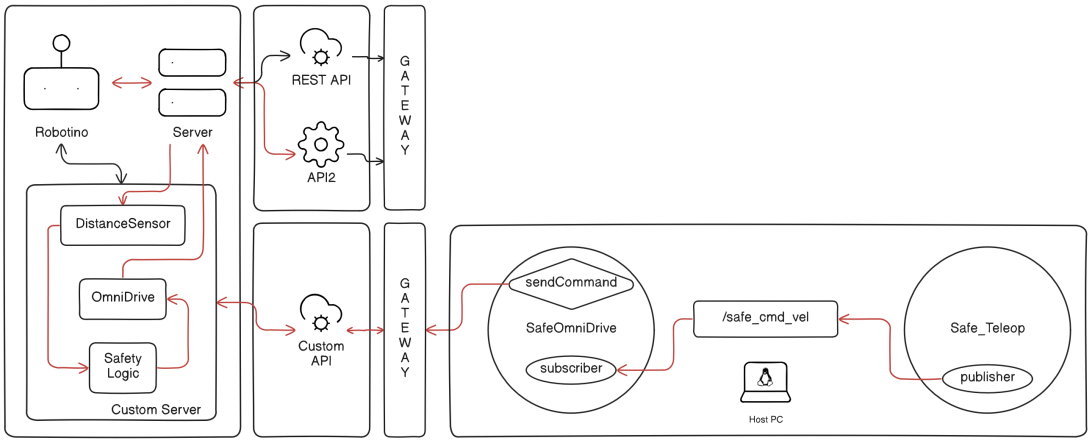

Figure 2.2: ROS Implementation diagram of Option 2.

2.3 Implementation

Since we will implement the first option, to begin with, we can get the `Safe_Teleop` node out of the way easily by copying the already existing `RTO_Teleop` node.

First things first, we need to create a new ROS package for our project since we want to modify the original package as little as possible. We create a ROS C++ package, as usual, using the commands described in the ROS2 documentation [7]. There, we define the header file and C++ file for our node called `SafeKeyboardTeleop`. The code is mostly the same as the original `KeyboardTeleop` code except for some variable names and the publisher's topic, `safe_cmd_vel` instead of `cmd_vel`. The code itself doesn't need an explanation, it just gets the keyboard input with a boilerplate and uses some C++ structures like mutex and watchdog to deal with the keyboard reading lock when multiple keys are pressed simultaneously.

For the `SafeOmniDrive` node, which is where all the magic happens, we need to define a `SafeOmniDrive` class which is a custom instance of `OmniDrive` class coming from the Robotino API2 library. The reason for this is to send velocity commands to Robotino, we need to establish some connection and this connection is handled by the Robotino API library. The `SafeOmniDrive` object we defined takes a ROS node as its parameter and we will use it to implement our safety

algorithm. We also need to define a ROS node to pass it to our `SafeOmniDrive` object, and we call it `SafeOmniDriveNode`. We define the default variables in this node to start our communication and spin our ROS node. The `initModules()` function sets our communication address which is the IP address of the Robotino, assigns a unique communication ID, and sets the necessary default parameters for the `OmniDrive` object. We set the `connectToServer` to false because the original code already opens a communication and we just need to set a unique communication ID for our custom object.

```
void SafeOmniDriveNode::initModules()
{
    com_.setAddress(hostname_.c_str());

    safe_omni_drive_.setComId(com_.id());
    safe_omni_drive_.setMaxMin(max_linear_vel_, min_linear_vel_, max_angular_vel_,
    ↪  min_angular_vel_);

    com_.connectToServer(false);
}
```

If we come back to our `SafeOmniDrive` class, we define two subscribers, one for the velocity messages comping from `safe_omni_drive` topic and one for the sensor distance messages coming from the `distance_messages` topic. The goal of this node is to implement some kind of safety algorithm that will take the incoming teleoperation messages and check if it is safe to move before executing them using the distance sensors on the Robotino.

```
SafeOmniDrive::SafeOmniDrive(rclcpp::Node* parent_node)
{
        parent_node_name_ = std::string(parent_node->get_name());
    safe_cmd_vel_sub_ =
    ↪  parent_node->create_subscription<geometry_msgs::msg::Twist>("safe_cmd_vel", 10,
    ↪  std::bind(&SafeOmniDrive::safeCmdVelCallback, this, _1));
        distance_sensors_sub_ =
        ↪  parent_node->create_subscription<sensor_msgs::msg::PointCloud>("distance_sensors",
        ↪  10, std::bind(&SafeOmniDrive::distanceSensorsCallback, this, _1));
}
```

We know that the move commands come from the `safe_cmd_vel` topic as `Twist` messages and contain three variables; velocity in the X direction, velocity in the Y direction, and angular velocity in radians. There is a function to map incoming teleoperate messages to the min/max velocity range that is defined at the start

of the `safeCmdVelCallback` function which is copied from the `cmdVelCallback` function in the `rto_teleop` package, so I will pass that. In `rto_teleop` package after this mapping algorithm, the message is published to the `cmd_vel` topic. This is the part where we can implement the safety algorithm.

Before writing our safety algorithm we need to understand the type of messages published to the `distance_sensors` topic. We can get the info with:

```
~ ros2 topic info /rto3/distance_sensors
Type: sensor_msgs/msg/PointCloud
Publisher count: 1
Subscription count: 1
```

We see that the type of the message is `PointCloud` from the `sensor_msgs` library. We can get more info about this message type with:

```
~ ros2 interface show sensor_msgs/msg/PointCloud
```

Then, we understand that the `PointCloud` message is an "Array of 3D points", represented by three 32-bit float variables, x, y, and z respectively. However, when we listen to the topic with:

```
~ ros2 topic echo /rto3/distance_sensors
```

we see that there are nine objects with x, y, and z fields. That is because Robotino has nine sensors on it with 40-degree spacing. The z field is the same for all nine, which represents the height of the sensors from the ground. The x and y values, however, are quite different when there is no object within the sensor's reach. With some investigation in the `rto_node` package, we find in the `DistanceSensorArrayROS.cpp` file, there is a section to generate this point cloud message.

```
1  void DistanceSensorArrayROS::distancesChangedEvent(const float* distances, unsigned int
   ↪  size)
2  {
3          // Build the PointCloud msg
4          distances_msg_.header.stamp = stamp_;
5          distances_msg_.header.frame_id = "base_link";
6          distances_msg_.points.resize(size);
7
8          for(unsigned int i = 0; i < size; ++i)
9          {
10                 // 0.698 radians = 40 Degrees
11                 // 0.2 is the radius of the robot
12                 distances_msg_.points[i].x = ( distances[i] + 0.2 ) * cos(0.698 * i);
```

```
13                distances_msg_.points[i].y = ( distances[i] + 0.2 ) * sin(0.698 * i);
14                distances_msg_.points[i].z = 0.05; // 5cm above ground
15        }
```

From this piece of code, we understand that the data that comes from the sensors are just distance in meters and some trigonometry is applied to turn this distance data to `PointCloud` format. The transformation is visualized below for easier understanding.

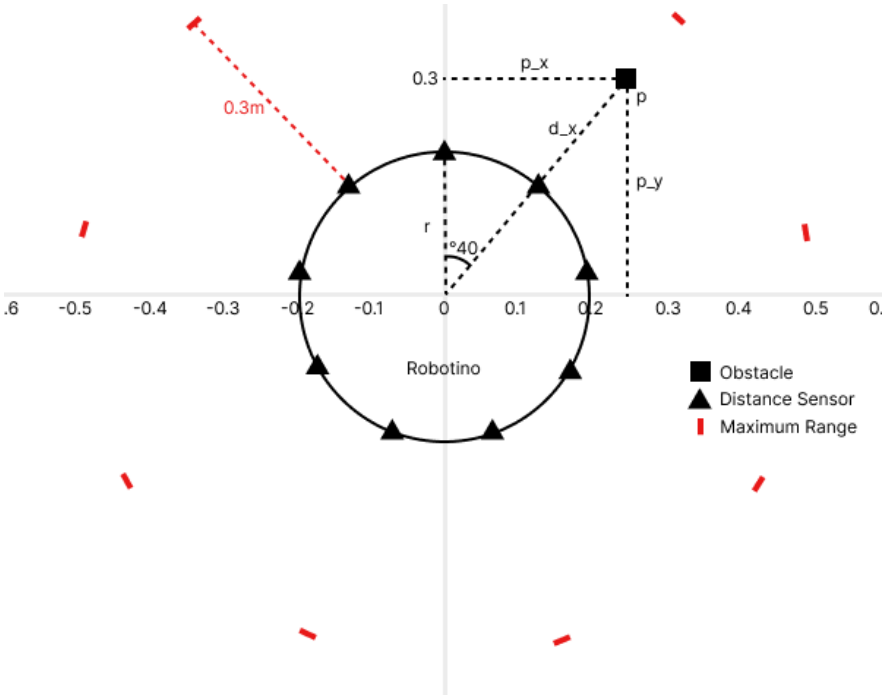

Figure 2.3: Sensor location and distance measurement graph

$$p_x = (d_x + r) \cdot cos(40 \cdot \frac{\pi}{180}) \qquad p_y = (d_x + r) \cdot sin(40 \cdot \frac{\pi}{180}) \tag{2.1}$$

In figure 2.3, only two dimensions are shown because the Z-axis is fixed and doesn't change for any of the sensors, or detected obstacles. With that, we understand that the values we see when no obstacles are in the range are the x and y positions of the sensors' maximum reach points, relative to the center of the Robotino which is the origin.

For the crash safety algorithm, I came up with a very simple idea. Since we get the positions of the detected obstacles, we can create a virtual safety circle with a radius of r_{safety}, centered at the origin, and check if the obstacle is inside or on the circle, or outside of the circle. If the obstacle is inside or on the circle, we

can choose to not execute the velocity command, and if the obstacle is outside of the circle we can execute the command as it is. This is very simple and easy to apply since there is a mathematical formula for a circle with a radius of r:

$$r^2 = x^2 + y^2 \tag{2.2}$$

and we can use this as if the sum of the obstacle's x and y coordinates is bigger than the radius of the safety radius, the obstacle is outside of the safety circle, and vice versa. The function of this check is given as:

```
bool SafeOmniDrive::isInsideSafetyBubble(double point_x, double point_y, double
↪ safety_radius)
{
        return pow(safety_radius, 2) >= pow(point_x, 2) + pow(point_y, 2);
}
```

Now, we can write our safety algorithm:

```
std::lock_guard<std::mutex> lock(distance_sensor_mutex_); // Memory safe sensor readings
if (!distance_sensor_readings_.points.empty())
{
    bool safety_breach = false;

    for (unsigned int i = 0; i < 9; ++i)
    {
        safety_breach =
            safety_breach |
            isInsideSafetyBubble(
                distance_sensor_readings_.points[i].x,
                distance_sensor_readings_.points[i].y,
                safety_radius_);
    }
    if (!safety_breach)
        setVelocity( linear_x, linear_y, angular);
}
```

The code is very simple, first, it checks if the sensor readings are received, and then it checks for all nine sensor readings to see if an obstacle is detected on or inside of the safety circle. If the `safety_breach` variable is false, which means there is no obstacle detected inside or on the safety circle, send velocity command to Robotino with `setVelocity()` function. If not, do nothing.
However, the last line, "if not, do nothing" is pretty problematic because it says to Robotino, operate as told until you detect an obstacle inside your safety circle, and as soon as you detect it, **stop** and **stay** there, until the obstacle moves away from the safety circle or someone physically moves you elsewhere. To fix this

we need an "else" statement and check if the next command received while an object is inside the safety circle is making the Robotino closer to the obstacle or move away from it. Because the operator may want to move it away from the obstacle to continue navigating safely. We can do this by calculating the position of the safety circle if the received is executed before actually executing it and checking where the obstacle falls on the safety circle. This is checked with a second variable, `not_safe_to_navigate`, with a similar logic but this time we are moving the obstacle with the received velocity commands (yes I said "by moving the safety circle" before, but it doesn't make any difference whether you move the robot or the obstacle, the math is the same, only the perspective is different). So the final code looks like this, with the addition of some debugging print statements to better understand what happens while the code is running:

```
std::lock_guard<std::mutex> lock(distance_sensor_mutex_); // Memory safe sensor readings
if (!distance_sensor_readings_.points.empty())
{
    bool safety_breach = false;
    bool not_safe_to_navigate = false;

    double distance_x = linear_x * 0.8; // Safety circle margin added for
    double distance_y = linear_y * 0.8; // compensating sensor inaccuracies

    for (unsigned int i = 0; i < 9; ++i)
    {
        safety_breach = safety_breach | isInsideSafetyBubble(
                                            distance_sensor_readings_.points[i].x,
                                            distance_sensor_readings_.points[i].y,
                                            safety_radius_);
    }
    if (!safety_breach)
        setVelocity( linear_x, linear_y, angular);
    else {

        for (unsigned int i = 0; i < 9; ++i) {
            not_safe_to_navigate = not_safe_to_navigate | isInsideSafetyBubble(
                distance_sensor_readings_.points[i].x - distance_x,
                distance_sensor_readings_.points[i].y - distance_y,
                safety_radius_
            );
        }
        if (!not_safe_to_navigate)
            setVelocity(linear_x, linear_y, angular);
        else setVelocity(0.00, 0.00, angular);
    }

    std::ostringstream os;
    os << "SafetyBreach: " << boolToString(safety_breach) << "\nSafeToNavigate: " <<
    ↪  boolToString(!not_safe_to_navigate);
    RCLCPP_INFO(rclcpp::get_logger(parent_node_name_), "%s", os.str().c_str());
}
```

This concludes this section, however, there are some things that I think need mentioning. This solution solves the problem to some degree but it is not close to perfect due to some reasons. The built-in sensors on Robotino were the only sensors I had access to and because they are infrared (IR) sensors, they have limited accuracy and range. From Festo's website: *"The infrared distance sensor allows accurate relative distance measurements of an object between 4 cm and 30 cm"*[**2**]. Furthermore, because they are IR sensors, they cover a linear field of vision, which means, that if an obstacle falls between the two sensors, Robotino cannot detect it and may still crash. Thus, though this approach provides some safety, it is still advised to operate the robot with supervision. I have added a figure to visualize the Robotino's field of vision with the IR sensors only.

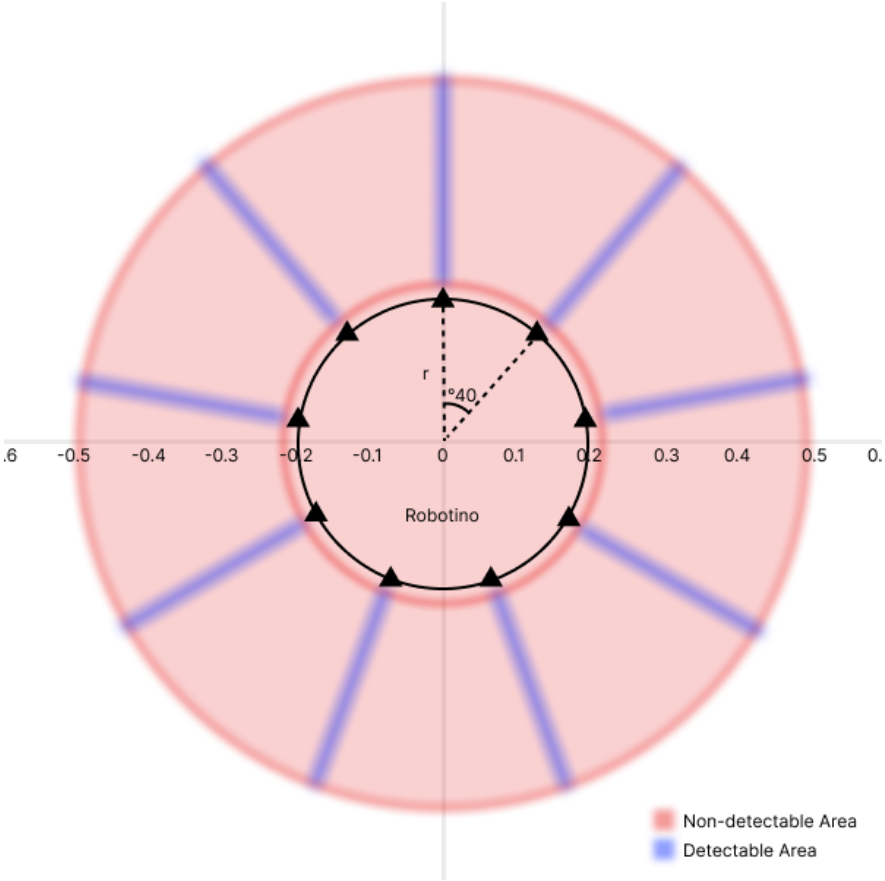

Figure 2.4: Field of view approximation of Robotino.

3 TASK2: Graphical User Interface

For the next part of the project, I've been assigned to develop a Graphical User Interface (GUI) within the ROS environment using Qt, a cross-platform user interface development tool. There are several options for developing GUI with Qt, and for this project, we decided on using PyQt5, a Python library for developing applications with Qt. Also, we used the QtDesigner tool, which is a GUI tool for designing GUIs with Qt.
I first created a virtual environment for PtQt5 using

```
python3 -m venv /gulbasozan/home/qtvenv
```

and installed PyQt5 library with

```
pip install PyQt5
```

With that, we have the necessary setup to begin developing GUI applications with PyQt5. For the basics of Qt and QtDesigner, I recommend this awesome playlist from **Tech With Tim** [**15**]. Then, we defined the necessary features of the GUI, which are a teleoperating interface with buttons that can be clickable and activated by a keyboard, a camera interface, and a command line interface (CLI).

3.1 Teleoperating Interface

The goal for this section is very simple, just the buttons that control the robot and some visual feedback when the buttons are pressed whether by mouse or keyboard to create a nice user experience. I designed the teleoperating interface on QtDesiner and exported it to a Python file using

```
pyuic5 -x <path/to/ui/file.ui> -o <path/to/python/file.py>
```

The design is shown in Figure 3.1.

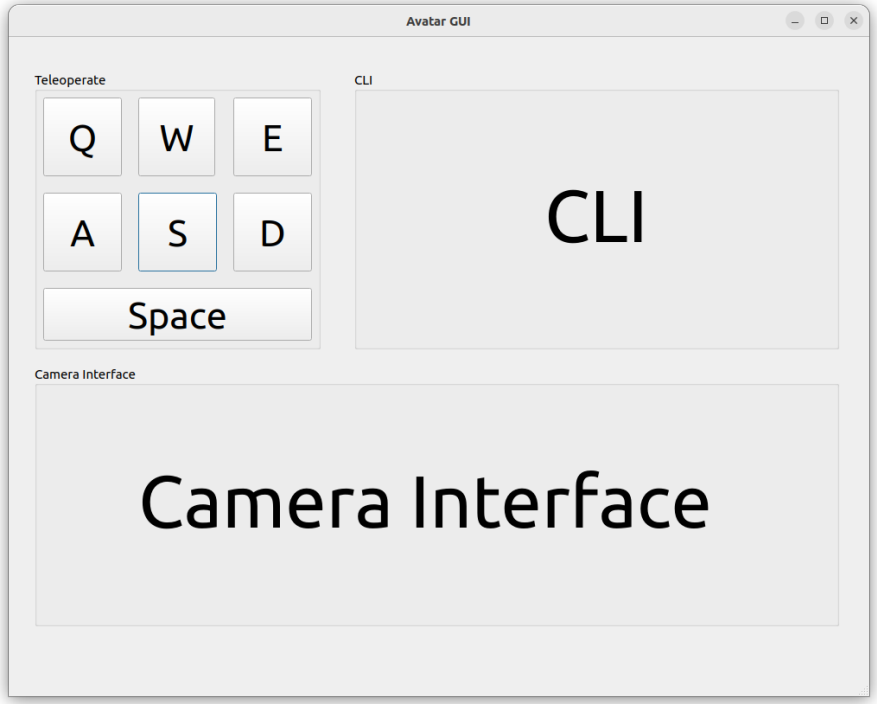

Figure 3.1: First design of the GUI.

The first step to implement this UI in our ROS environment is to create a ROS package which we can create in Python with

```
ros2 pkg create --build-type ament_python --license Apache-2.0 avatar_gui
```

We create two more directories different from the default ones: designs, which we will store .ui files we create with the QtDesigner, and assets, which we will store the necessary images that make our GUI beautiful. The UI Python file created with `pyuic5` command is stored under `avatar\_gui`, the source directory for our Python package. We also create a file named `avatar\_gui\_node.py` under this directory to write our ROS Python node template.
Next, we create a `ROSNode` object called `AvatarGUINode` and initialize a publisher to the `rto3/safe\_cmd\_vel` topic. We also write a publisher callback function that takes velocity values and publishes them.

```
1  class AvatarGUINode(Node):
2      def __init__(self):
3          super().__init__("avatar_gui")
4
5          self.safe_cmd_vel_pub_ = self.create_publisher(Twist, 'rto3/safe_cmd_vel', 10)
6          self.safe_cmd_vel_msg_ = Twist()
7
8          self.get_logger().info("AvatarGUINode initialized.")
9
```

```
10      def pubCallback(self, vel_x, vel_y, vel_omega):
11          self.safe_cmd_vel_msg_.linear.x = vel_x
12          self.safe_cmd_vel_msg_.linear.y = vel_y
13          self.safe_cmd_vel_msg_.angular.z = vel_omega
14
15          self.safe_cmd_vel_pub_.publish(self.safe_cmd_vel_msg_)
```

We also create a `UI\_AvatarGUI` instance named `AvatarGUIInstance` which takes a ROS node and `MainWindow` object. We also define a button callback function that takes a key value and adjusts necessary velocity values to publish to the ROS topic.

```
1   class AvatarGUIInstance(Ui_AvatarGUI):
2       def __init__(self, ros_node, main_window):
3           super().__init__()
4           self.ros_node_ = ros_node
5           self.setupUi(main_window)
6
7       def buttonCallback(self, key):
8           vel_x_ = 0.00
9           vel_y_ = 0.00
10          vel_omega_ = 0.00
11
12          match key:
13              case QtCore.Qt.Key_W:
14                  vel_x_ = 0.05
15              case QtCore.Qt.Key_A:
16                  vel_y_ = 0.05
17              case QtCore.Qt.Key_S:
18                  vel_x_ = -0.05
19              case QtCore.Qt.Key_D:
20                  vel_y_ = -0.05
21              case QtCore.Qt.Key_Q:
22                  vel_omega_ = -0.05
23              case QtCore.Qt.Key_E:
24                  vel_omega_ = 0.05
25              case QtCore.Qt.Key_Space:
26                  vel_x_ = vel_y_ = vel_omega_ = 0.00
27
28          self.ros_node_.pubCallback(vel_x_, vel_y_, vel_omega_)
```

For the main function, we need a different approach from the default ROS spin template because we need to run the GUI application besides the ROS node and these two applications should not intervene in each other's work and be able to communicate with each other. Thus we use a concept called **threading**. To understand threading in detail you can check this website [**13**]. What threading does basically is to create multiple separate flows of executions for applications to run on. We use threading to create a separate flow of execution for our ROS node because the Qt application needs to run on the main thread.

```
def ros_spin_thread(node):
    rclpy.spin(node)

def main(args=None):
    rclpy.init(args=args)
    node = AvatarGUINode()

    spin_thread = threading.Thread(target=ros_spin_thread, args=(node,))
    spin_thread.start()

    app = QtWidgets.QApplication(sys.argv)
    main_window = QtWidgets.QMainWindow()
    AvatarGUI = AvatarGUIInstance(node, main_window)

    main_window.show()
    app.exec_()

    node.destroy_node()
    rclpy.shutdown()
    spin_thread.join()
```

Now, if we run the code nothing happens, because we didn't initialize any kind of event handling for button clicks or keyboard presses.
For button clicks with the mouse, we need to initialize a slot connection, a feature of the Qt object. With that, we can call a function every time a button is clicked with its *<button>.clicked.connect()* property.

```
class AvatarGUIInstance(Ui_AvatarGUI):
    def __init__(self, ros_node, main_window):
        ///...///
        self.initButtonSlotConnections(self.buttonCallback)

    def buttonCallback(self, key):
        ///...///

    def initButtonSlotConnections(self, callback):
        self.forward.clicked.connect(lambda: callback(QtCore.Qt.Key_W))
        self.backward.clicked.connect(lambda: callback(QtCore.Qt.Key_S))
        self.right.clicked.connect(lambda: callback(QtCore.Qt.Key_D))
        self.left.clicked.connect(lambda: callback(QtCore.Qt.Key_A))
        self.pos_angular.clicked.connect(lambda: callback(QtCore.Qt.Key_E))
        self.neg_angular.clicked.connect(lambda: callback(QtCore.Qt.Key_Q))
        self.halt.clicked.connect(lambda: callback(QtCore.Qt.Key_Space))
```

For the keyboard control, we need to implement an event filter that will intercept the keyboard press event and run our button callback function.

```
class KeyPressFilter(QtCore.QObject):
    def __init__(self, ros_node, gui):
        super().__init__()
```

```
4           self.ros_node = ros_node
5           self.gui = gui
6
7       def eventFilter(self, obj, event):
8           vel_x_ = 0.00
9           vel_y_ = 0.00
10          vel_omega_ = 0.00
11
12          if event.type() == QtCore.QEvent.KeyPress:
13              match event.key():
14                  case QtCore.Qt.Key_W:
15                      vel_x_ = 0.05
16                  case QtCore.Qt.Key_A:
17                      vel_y_ = 0.05
18                  case QtCore.Qt.Key_S:
19                      vel_x_ = -0.05
20                  case QtCore.Qt.Key_D:
21                      vel_y_ = -0.05
22                  case QtCore.Qt.Key_Q:
23                      vel_omega_ = -0.05
24                  case QtCore.Qt.Key_E:
25                      vel_omega_ = 0.05
26                  case QtCore.Qt.Key_Space:
27                      vel_x_ = vel_y_ = vel_omega_ = 0.00
28
29              self.ros_node.pubCallback(vel_x_, vel_y_, vel_omega_)
30
31              return True
32
33          return super().eventFilter(obj, event)
34
35  def main(args=None):
36      ///...///
37
38      key_press_filter = KeyPressFilter(node, AvatarGUI)
39      app.installEventFilter(key_press_filter)
40
41      ///...///
```

Right now, all of our teleoperation functionality is working, however, there is no visual feedback to create a convenient user experience. Thus, we need a custom styling implementation for our current event handlers. We achieve this by creating two button style sheets using CSS, one for buttons' idle state and one for buttons' pressed state.

```
1   PUSH_BUTTON_IDLE_STYLESHEET = """
2               QPushButton {
3                   background-color: #6f6866;
4                   color: #eff6ee;
5                   border-radius: 16px;
6               }
7               QPushButton:pressed {
8                   background-color: #38302e
```

```
            }

    """
PUSH_BUTTON_PRESSED_STYLESHEET = """
            QPushButton {
                background-color: #38302e;
                color: #eff6ee;
                border-radius: 16px;
            }
            QPushButton:pressed {
                background-color: #38302e
            }

    """
```

Next, we need to implement these style changes between events, button clicks, or keyboard presses and releases.

```
class KeyPressFilter(QtCore.QObject):
        ///...///
        if event.type() == QtCore.QEvent.KeyPress:
            match event.key():
                case QtCore.Qt.Key_W:
                    vel_x_ = 0.05
                    self.gui.forward.setStyleSheet(PUSH_BUTTON_PRESSED_STYLESHEET)

                ///...///

                case QtCore.Qt.Key_Space:
                    vel_x_ = vel_y_ = vel_omega_ = 0.00
                    self.gui.halt.setStyleSheet(PUSH_BUTTON_PRESSED_STYLESHEET)

        ///...///

        if event.type() == QtCore.QEvent.KeyRelease:
            match event.key():
                case QtCore.Qt.Key_W:
                    self.gui.forward.setStyleSheet(PUSH_BUTTON_IDLE_STYLESHEET)
                case QtCore.Qt.Key_A:
                    self.gui.left.setStyleSheet(PUSH_BUTTON_IDLE_STYLESHEET)
                case QtCore.Qt.Key_S:
                    self.gui.backward.setStyleSheet(PUSH_BUTTON_IDLE_STYLESHEET)
                case QtCore.Qt.Key_D:
                    self.gui.right.setStyleSheet(PUSH_BUTTON_IDLE_STYLESHEET)
                case QtCore.Qt.Key_Q:
                    self.gui.neg_angular.setStyleSheet(PUSH_BUTTON_IDLE_STYLESHEET)
                case QtCore.Qt.Key_E:
                    self.gui.pos_angular.setStyleSheet(PUSH_BUTTON_IDLE_STYLESHEET)
                case QtCore.Qt.Key_Space:
                    self.gui.halt.setStyleSheet(PUSH_BUTTON_IDLE_STYLESHEET)
            return True

        return super().eventFilter(obj, event)

class AvatarGUIInstance(Ui_AvatarGUI):
```

```
38      def __init__(self, ros_node, main_window):
39          ///...///
40          self.initButtonStyleSheet(PUSH_BUTTON_IDLE_STYLESHEET)
41
42      ///...///
43
44      def initButtonStyleSheet(self, styleSheet):
45          self.forward.setStyleSheet(styleSheet)
46          self.backward.setStyleSheet(styleSheet)
47          self.right.setStyleSheet(styleSheet)
48          self.left.setStyleSheet(styleSheet)
49          self.pos_angular.setStyleSheet(styleSheet)
50          self.neg_angular.setStyleSheet(styleSheet)
51          self.halt.setStyleSheet(styleSheet)
```

With that, we now have a working, pretty-looking teleoperating interface with a convenient user experience.

Figure 3.2: Final design of the GUI, while the "W" key is pressed

3.2 Camera Interface

To have a camera stream displayed on the Qt application, we need to find a way to publish the captured camera stream to a ROS topic. If the camera is connected to a ROS running system, this is fairly easy to achieve. However, the Robotino does not have ROS installed on it and we do not want it to be installed either.

Furthermore, since we are planning to operate Robotino wirelessly, we need to find a way to send the camera stream to our host computer which leaves us with one logical option: over the network. There are several options to send video data over a network and upon some research, I came across WebRTC.

3.2.1 WebRTC

Web Real-Time Communication (WebRTC) is a protocol that defines a set of rules for the communicating peers to send and receive real-time data securely. There is also WebRTC API that is specified for Javascript and available in most browsers as a standard API. Furthermore, implementations in other languages are also possible and some of them can be found on the internet.
WebRTC communication is called serverless because the data stream is sent through peers, however, to establish this communication between peers, we need to introduce them to the minimum info possible to open a connection. This is called **Signaling** in WebRTC communication and the minimum amount of information shared to start the communication is called **Session Description Protocol (SDP)**.

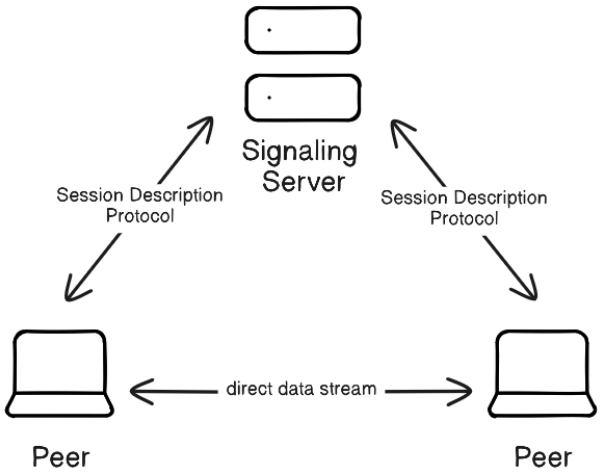

Figure 3.3: WebRTC Communication Diagram

In Figure3.3 the SDP transfer is shown as it's handled by a server called the Signaling Server. This is true and is how the signaling is handled in most cases, however, it is possible to transfer SDP without using a server, which we will use. There are lots of things going under the hood of the WebRTC and it is good to have some understanding about it. For that, you can check out this website: webrtcforthecurious.com[**4**]

Implementation Using aiortc

As I said before, there are multiple implementations of WebRTC and for this project, we will use aiortc. Aiortc is a Python implementation of WebRTC and also Object Real-Time Communication (ORTC) protocols, that uses the asyncio library in its core. We will implement a one-directional video communication using copy-paste signaling as a base for our application and try to understand each part of the code for easy modification in the future. The code generated is based on official aiortc library examples which can be found here [**1**].
Since both Robotino and Host PC run in Linux, and our GUI is based on Qt, we don't need a web application as shown in the examples. What we need is two separate scripts, one for taking a video stream from a webcam and sending it through WebRTC communication, and the other one for receiving this video stream and displaying it. For the signaling (SDP transaction) we will use copy-paste signaling which is provided in the aiortc examples. What copy-paste signaling is basically, instead of sending the SDP to a server it prints to the terminal and users can manually copy the SDPs and paste each other's terminals. It is not important who sends the first SDP for our project, however, it does affect some portion of the code (whoever sends the SDP first needs to create a media track and the answerer must consume that media track's properties). The full code can be found on GitHub however I will explain the sections of the code step by step.

MediaStreamTrack

From aiortc API Reference: *"class aiortc.**MediaStreamTrack**: A single media track within a stream."* Using this class we can create an instance of what we record from the webcam that we can send it using WebRTC and manipulate it. What I mean by manipulation is changing and arranging the properties like time stamps of the individual frames recorded by the webcam itself. This class originated from PyAV's `Stream` object and much of the media data-related stuff in aiortc is based on PyAV. So if you have some confusion or error about the video, audio encoding, playback, or save, make sure you also check PyAV's documentation here [**8**].

```
1  class VideoStreamTrack(MediaStreamTrack):
2      kind = "video"
3      _start: float
4      _timestamp: int
5
6      def __init__(self, track):
```

```
7           super().__init__()
8           self.track = track
9
10      async def next_timestamp(self) -> Tuple[int, fractions.Fraction]:
11          if self.readyState != "live":
12              raise MediaStreamError
13
14          if hasattr(self, "_timestamp"):
15              self._timestamp += int(VIDEO_PTIME * VIDEO_CLOCK_RATE)
16              wait = self._start + (self._timestamp / VIDEO_CLOCK_RATE) - time.time()
17              await asyncio.sleep(wait)
18          else:
19              self._start = time.time()
20              self._timestamp = 0
21          return self._timestamp, VIDEO_TIME_BASE
22
23      async def recv(self):
24          frame = await self.track.recv()
25          pts, time_base = await self.next_timestamp()
26          frame.pts = pts
27          frame.time_base = time_base
28          return frame
```

We create a new instance of `MediaStreamTrack` called `VideoStreamTrack` and define two functions: `next\_timestamp()` and `recv()`.
The `next\_timestamp` function adjusts the proper timestamp of the frames so that when the frames are sent, they can be reorganized in the right order with proper FPS. There is a state check at the beginning of the function, preventing excessive timestamp generation in case of errors and the variables `VIEO_PTIME`, `VIDEO_CLOCK_RATE` and `VIDEO_TIME_BASE` are defined beforehand as global variables.
The `recv()` function is how we receive individual frames of the recorded stream track and normalize it to our timebase. If we dive inside the function, we first get the current frame, change its PTS and time base, and return the normalized frame.
Both `nex_timebase()` and `recv()` already exist as a prop of `MediaStreamTrack`, however, we override these functions so that when we try to get transferred frames on the other peer, we get them with the right timing and order.

Copy-Paste Signaling

This piece of code is the same with all peers if the signaling server is not used. What it does is very straightforward, wait for the SDP object, if the received object is actually an SDP object, and mark the received object as the remote SDP. Then check if it is an "offer" or "answer" type. If it is an "offer", generate

local SDP and send it to the signaling environment (in this case it means printing it on the terminal). If it is an "answer", we don't need to do special tasks because we already have the local SDP (offer) we only need the remote SDP (answer). The creation of the offer is isolated to the "offer" peer's main function to avoid duplication.

```
1  async def consume_signaling(pc, signaling):
2      while True:
3          obj = await signaling.receive()
4
5          if isinstance(obj, RTCSessionDescription):
6              await pc.setRemoteDescription(obj)
7
8              if obj.type == "offer":
9                  #send answer
10                 await pc.setLocalDescription(await pc.createAnswer())
11                 await signaling.send(pc.localDescription)
12             elif isinstance(obj, RTCIceCandidate):
13                 await pc.addIceCandidate(obj)
14             elif obj is BYE:
15                 print("Exiting..")
16                 break
```

The remaining two conditional statements are for ICE candidates and exit. ICE candidates are added to secure connections between peers outside the same network. More info here [**10**].

"Offer" Function

The first 5 lines are for the decent debugging outputs, so we can pass that. Next, we connect to the signaling channel created on the main function and connect to the webcam. Then, we have a decorator function (learn about it here [**14**]) that we use to extend the function `on_track()` which is the function called when a new "track" is added to the channel. This part is not used as we only transfer the data one way. If we configure to transfer data both ways, then this function will called and remote track will be added to our RTC channel. Next, we add our webcam's stream to our RTC channel. One thing to notice here is that both the remote track and webcam track are added after converting them to our custom `VideoStreamTrack` object. This needs to be done to ensure that the received data is in the correct order.
Right now we have an RTC channel and a media track that is streamed over this channel. However, to send this data to another peer, we need to know some information about their RTC channel. Since we are the "offerer", the one that sets up the communication, we need to create an "offer". In lines 32 and 33, we

create an "offer" which is our SDP, in the `RTCSessionDescription` type and set it as our local description. Then we send (print to console in this case) this SDP in the hopes of getting an answer.

```
async def offer(pc, signaling):
    pc_id = "PeerConnection(%s)" % uuid.uuid4()

    def log_info(msg, *args):
        logger.info(pc_id + " " + msg, *args)

    await signaling.connect()

    # open media source
    options = {"framerate": "30", "video_size": "640x480"}
    player = MediaPlayer("/dev/video0", format="v4l2", options=options)
    log_info("Media source is created %s", player)

    @pc.on("track")
    def on_track(track):
        logger.debug("Track %s received", track.kind)

        if track.kind == "audio":
            pass
        elif track.kind == "video":
            #this should not get called - as we're only pushing one way
            logger.error("on_track - should not get video track")

            # add our locally generated video track here
            pc.addTrack(VideoStreamTrack(track))
            log_info("Track added %s to pc", player.video.kind)

    # Init media transfer interface
    pc.addTrack(VideoStreamTrack(player.video))

    # send offer
    await pc.setLocalDescription(await pc.createOffer())
    await signaling.send(pc.localDescription)

    await consume_signaling(pc, signaling)
```

Lastly, we "consume" signaling which means that we send an offer and wait for an answer. The function was explained above. When the answer is received and we set our remote description the communication begins.
The story is the same with the receiving peer but we create an image from the received frame and publish it to the relative topic inside the `recv()` function.

RTC Eventloop

This is the part where we run the RTC connection. We first create our signaling channel, RTC communication channel (PC), and a recorder for the webcam feed.

The recorder is only needed in receiving peers for this time since we only send data one way. If the recorder is not initialized, then we cannot get the transmitted data. To save the received data to a file, `MediaRecorder` is used. However, we don't want to save the data we just want to publish it to a topic, thus, we use the `MediaBlackhole` and we discard frames after we publish them in `recv()` function.

```
def rtc_eventloop():
    signaling = create_signaling(args)
    pc = RTCPeerConnection()

    # for the receiving peer
    recorder = MediaBlackhole()
    # recorder = MediaRecorder("video.mp4")

    coro = answer(pc, signaling, recorder)

    # Init event loop
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)

    # Run event loop
    try:
        loop.run_until_complete(coro)
    except KeyboardInterrupt:
        pass
    finally:
        loop.run_until_complete(pc.close())
        loop.run_until_complete(signaling.close())
        loop.run_until_complete(recorder.stop())
        loop.close()
```

The rest is boilerplate, we create a new event loop and pass our offer/answer function to that event loop.
The important part to understand for the WebRTC section is this in the sending peer (Robotino) we will run the `rtc_evenloop` function as it is, however, for the receiving peer, we need to integrate this code with a ROS node.

3.2.2 ROS and WebRTC Integration

The integration is straightforward, let the RTC event loop run on the main thread, and run the ROS node on a different thread. As I said before, we need to define our custom media track type to access the `recv()` function to convert received frames to images and publish them. For the code below we used `VideoStreamTrack` instead of `MediaStreamTrack`, but the idea is the same, the `VideoStreamTrack` is just a `MediaStreamTrack` but without the audio support. If we decide to publish the audio data if ever have one, we just change it to a `MediaStreamTrack`.

```python
# Normalize PTS to sync with the real-time webcam feed
class RemoteVideoStreamTrack(VideoStreamTrack):
    def __init__(self, track, node ):
        super().__init__()
        self.track = track
        self.node = node

    async def recv(self):
        frame = await self.track.recv() # Receive frame
        pts, time_base = await self.next_timestamp() # Adjust timestamp
        frame.pts = pts
        frame.time_base = time_base
        img = frame.to_ndarray(format="bgr24") # Convert frame to img array

        try:
            self.node.pubCallback(img) # Publish image
        except Exception as e:
            logger.info(f"Unexpected error running ROS Code {e}")
            return frame

        return frame # Return frame for disposing using MediaBlackhole
```

Inside the `recv()` function, we get the frame, normalize it, and convert it to the image. Then, we try to publish it by calling the callback function of the node object which we passed when we first added the media track inside the answer function. Finally, we return the frame so that the `MediaBlackhole` can discard it.

For the ROS node, we initialize the node as usual, create a publisher to publish the images to a topic, and set the message type as ROS `Image` type. We use the ROS Bridge library to convert the CV2 image that we get from the `recv()` function to the ROS `Image` type. If the conversion is successful, we publish the image.

```python
class RTCWebcamNode(Node):
    def __init__(self ):
        super().__init__("rtc_webcam")

        self.image_raw_pub_ = self.create_publisher(Image, "rto3/img_raw", 10)
        self.image_raw_msg_ = Image()
        self.bridge = CvBridge()
        self.get_logger().info("RTC Webcam Node initialized.")

    def pubCallback(self, img):
        try:
            self.image_raw_msg_ = self.bridge.cv2_to_imgmsg(img, "bgr8")
        except CvBridgeError as e:
            self.get_logger().info("Couldn't convert the image: {e}")
            return

        self.image_raw_pub_.publish(self.image_raw_msg_)
```

3.3 ROS and Qt Video Stream Integration

For the last part of the GUI, we need to display image messages published to the `/rto3/img\_raw` topic. Usually, it is pretty straightforward to achieve this, first, initialize the Qt Application with the label that will display incoming frames and pass that frame to the ROS node initialization function to update inside the subscriber callback function. You can find examples of this implementation on the internet, however, we need to go for a different approach. The reason for that is we initialize the ROS node before and then pass it to the Qt Application to run teleop publisher callback function when the GUI keys are pressed. How are we going to make it work then, you may ask and the answer is **PyQt Signals**.

3.3.1 PyQt5 Signals

We can think of signals as notifications for our software, that can be triggered by user actions, or carry data across the application when something happens [**12**]. There are also "slots", which are basically the signal-receiving components. We used signals before in this project, however, they were built-in signals designed for specific purposes thus, they were not called signals but used as props of other components.
When we initialized an event handler for the button clicks, we passed a callback function to the `<button>.clicked.connect()`. The function we passed was the **slot** that accepts **clicked signal**. By using this logic we can understand that we need to create a custom signal that will emit the image message when the subscriber callback function runs and create a slot so that the image label receives that emitted signal.

3.3.2 Custom PyQt Signal Implementation

A PyQt signal needs to be defined under a **QObject** instance, outside of the `__init()__` function.

```
1  class RosImageEmitter(QObject):
2      image_signal = pyqtSignal(QImage) # Needs to be defined here specifically
3
4      def __init__(self):
5          super().__init__()
```

Next, we define a subscriber callback function that will take the image message, convert it and emit it with the signal we created (`RosImageEmitter`).

```
class AvatarGUINode(Node):
    def __init__(self, emitter):
        ///...///

        self.img_raw_sub_ = self.create_subscription(Image, 'img_raw',
        ↪  self.imgRawListenerCallback, 10)
        self.img_raw_msg_ = Image()

        self.bridge_ = CvBridge() # For image conversion

        self.emitter = emitter

        ///...///
    def imgRawListenerCallback(self, msg):
        try:
            cv_img = self.bridge_.imgmsg_to_cv2(msg, "bgr8")
        except CvBridgeError as error:
            self.get_logger().error(f"Could not convert the msg to cv2 img:\n {error}")

        # Need to format image to display it on QtApp
        height, width, channel = cv_img.shape
        bytes_per_line = 3 * width
        qt_image = QImage(cv_img.data, width, height, bytes_per_line,
        ↪  QImage.Format_RGB888).rgbSwapped()

        self.emitter.image_signal.emit(qt_image)
```

The default threading that we run ROS is not compatible with the PyQt signal thus, we need to run ROS inside a special thread called `QThread` which is a specialized Qt thread.

```
class ROSThread(QThread):
    def __init__(self, node):
        super().__init__()
        self.node = node

    # Ovveride defult QThread run function
    def run(self):
        rclpy.spin(self.node)
        rclpy.shutdown()
```

In the main function, we need to pass the PyQt signal to the ROS node so that we can initialize our subscriber callback.

```
def main(args=None):
    ///...///

    # Init Image Signal Emitter
    emitter = RosImageEmitter()

    # Init ROS Node
```

```
8       node = AvatarGUINode(emitter)
9       ros_thread = ROSThread(node)
10      ros_thread.start()
11
12      ///.../// 
```

Lastly, we need a slot that will take the emitted signal and update the related label on the GUI.

```
1   class AvatarGUIInstance(Ui_AvatarGUI):
2       def __init__(self, ros_node, main_window):
3           ///.///
4
5           self.ros_node_.emitter.image_signal.connect(self.updateImage) # run updateImage when
            ↪  a signal emitted
6
7       ///.///
8       def updateImage(self, qt_image):
9           pixmap = QPixmap.fromImage(qt_image)
10          self.camera_label.setPixmap(pixmap)
```

With that now we have a nice looking GUI that we can both teleoperate with the Robotino while receiving live webcam feed with low latency shown in figure 3.4. The latency measurement code is written in source code that you can find on GitHub to test in the future.

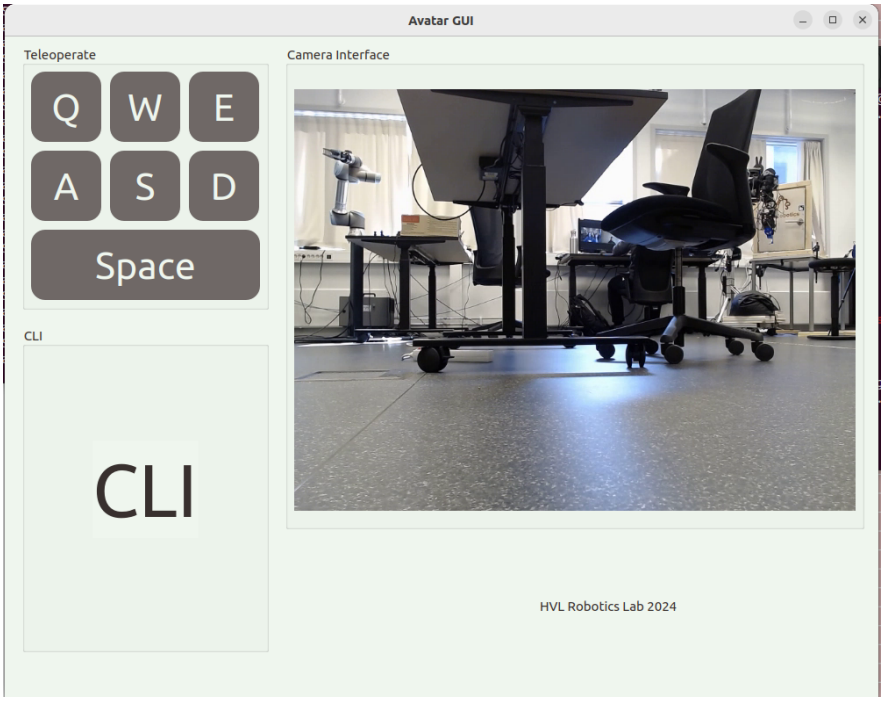

Figure 3.4: Screenshot of the GUI with working camera interface

Bibliography

[1] aiortc/examples at main · aiortc/aiortc, . URL https://github.com/aiortc/aiortc/tree/main/examples.

[2] Festo Didactic InfoPortal, . URL https://ip.festo-didactic.com/InfoPortal/Robotino/Hardware/Sensors/EN/DistanceSensors.html.

[3] how_to_setup_robotino_for_programming_and_navigation [DASL Wiki], . URL https://www.daslhub.org/unlv/wiki/doku.php?id=how_to_setup_robotino_for_programming_and_navigation.

[4] Introduction, . URL https://webrtcforthecurious.com/.

[5] Robotino3 usb restore - RobotinoWiki, . URL https://wiki.openrobotino.org/Robotino3_usb_restore.html.

[6] RobotinoWiki, . URL https://wiki.openrobotino.org/index.html.

[7] ROS 2 Documentation — ROS 2 Documentation: Humble documentation, . URL https://docs.ros.org/en/humble/index.html#.

[8] Streams — PyAV 9.0.3.dev0 documentation, . URL https://pyav.org/docs/develop/api/stream.html.

[9] Ubuntu 20 04 modifications - RobotinoWiki, . URL https://wiki.openrobotino.org/Ubuntu_20_04_modifications.html.

[10] N. Currier. Understanding ICE in WebRTC, Mar. 2022. URL https://temasys.io/guides/developers/webrtc-ice-sorcery/.

[11] R. K.A. Rahul-K-A/robotino-ros2, June 2024. URL https://github.com/Rahul-K-A/robotino-ros2. original-date: 2024-01-06T22:26:45Z.

[12] M. F. L. u. P. G. s. w. PyQt5. PyQt5 Signals, Slots and Events - pyqtSignal, pyqtSlot, Mouse Events & Context menus, May 2019. URL https://www.pythonguis.com/tutorials/pyqt-signals-slots-events/.

[13] R. Python. An Intro to Threading in Python – Real Python, . URL https://realpython.com/intro-to-python-threading/.

[14] R. Python. Primer on Python Decorators – Real Python, . URL https://realpython.com/primer-on-python-decorators/.

[15] Tech With Tim. PyQt5 Tutorial - How to Use Qt Designer, July 2019. URL https://www.youtube.com/watch?v=FVpho_UiDAY.