NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

SCSE22-0360

REMOTE TELEOPERATION WITH OBJECT DETECTION

FOO JUN MING

(U2021787E)

SUPERVISOR: DR. SMITHA KAVALLUR PISHARATH GOPI

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Submitted in Partial Fulfillment of the Requirements for
the Degree of Bachelor of Engineering (Computer Engineering)
Nanyang Technological University

2023

ABSTRACT

In today's digital age, WiFi has become an essential means of communication. However, its coverage area is limited, and it cannot reach remote areas. This project proposes Remote teleoperation of a device on NAT-type networks to resolve the issue. However, operating on NAT-type networks poses a significant challenge, primarily due to CGNAT and its dynamic IP allocation. To overcome this issue, a reliable solution is needed. VPN P2P tunneling is a powerful technology that enables secure and efficient communication over NAT-type networks, facilitating remote teleoperation. For applications requiring object detection, it is performed by the server over the NAT-type networks instead of on the device to provide cloud-based object detection to the teleoperated device, allowing for cheaper hardware implementation of said device. By leveraging these technologies, businesses can achieve enhanced productivity, cost savings, and improved performance in remote teleoperation scenarios.

The report presents the final year project to test remote teleoperation of the TurtleBot3 on NAT-type networks with a server providing object detection service through the same network. The final goal is to create a remote robot that can communicate with the operator PC through a carrier grade mobile network (CGNAT) in real-time, with low latency and high reliability.

ACKNOWLEDGEMENTS

I would like to express my utmost gratitude to my final year project supervisor, Dr. Smitha Kavallur Pisharath Gopi from the School of Computer Science and Engineering, for her unwavering support, guidance, and the checking of my well-being throughout the course of the project. Her expertise in the field of computing and advice have motivated me to achieve the goals of this project.

I would also like to express my gratitude to Mr Zulkiffle Bin Kashno for the smooth handling of the loaning process of the various equipment used for my project.

Lastly, I would like to thank my family for their incredible support and patience during the course of my project's development back at home.

TABLE OF CONTENTS

LIST OF FIGURES

List of Tables

CHAPTER 1: INTRODUCTION

1.1 Background and Motivation

Remote teleoperation refers to a process whereby an operator is operating a machine or system at a distance, granting the operator remote access to the system regardless of distance. One prominent advantage is that the operator is able to perform what used to be known as a dangerous on-site task without exposing themselves to danger. Remote teleoperation can also lower costs and enhance productivity in situations such as remote surgery, where it offers improved magnification and the ability to execute movements that human hands cannot accomplish.

With the recent rise in popularity of autonomous vehicles (AV), there is a need to implement remote teleoperation for autonomous cars, mainly used in edge cases whereby an AV is unable to make a decision and a human operator must step in to resolve the issue. From this, a reinforced learning classification model is used to imitate what a human operator would do in that specific scenario, as proposed in [1]. This makes AVs a lot safer which is why teleoperation is needed in AVs for higher safety standards. AVs are mobile and are usually not connected to the same network. Hence, there is a need to develop a remote teleoperation solution for AVs that can obtain network connectivity from Wireless Wide Area Network (WWAN) technologies like 4G cellular networks. With this solution, an operator will have much further range to perform remote teleoperation of AVs as compared to confining the AVs in smaller Wireless Area Network (WAN) coverage.

1.2 Objective

The main objective of the project is to develop a robot capable of remote teleoperation and object detection to allow the operator to reliably control the robot's movement and be updated about what the robot sees and detects through a video livestream. The video livestream is transferred over either a Wireless Local Area Network (WLAN) connection or a cellular network connection.

1.3 Scope

The project will focus on the development of the ROBOTIS TurtleBot3, a budget, portable, ROS-based programmable robot. For WLAN connection, the operator's personal computer (PC) and the TurtleBot3 is connected to the same network gateway while for cellular network connection, a subscriber identity module (SIM) dongle is used to connect the TurtleBot3 to the operator's PC via network tunneling. The network latency, network jitter and amount of data used are measured to determine whether the connection is suitable for remote teleoperation.

For the decision on which model to use for object detection, three metrics are considered: Accuracy, Mean Time taken for Inference and Mean Average Precision. The model is used to detect road signs of various classes and runs on the operator's PC due to insufficient computing resources available on the TurtleBot3.

The video livestream from the TurtleBot3's camera and the model's output are fed to the Oculus Quest 2, which is used to visualize to the operator what the TurtleBot3 is currently seeing and detecting. To determine the best control for the TurtleBot3 using the Oculus Quest 2, the joystick control and the headset movement control approaches are assessed via comparison of user experience.

CHAPTER 2: LITERATURE REVIEW

2.1 Network Communications

Fundamentally, most types of network communication follow the Open Systems Interconnection (OSI) model. The OSI model defines how data communication should be carried out between two or more systems in the network. It consists of seven abstracted layers with their own specific functions, to allow systems of different platforms or languages to communicate without the need to implement layer-specific network interfaces. Combining the functions of these seven layers, system communicating using the OSI model can interpret useful information from just a data stream of zeros and ones sent from another system. These layers include the physical, data link, network, transport, session, presentation and application layers [1].

Physical layer is responsible for handling data as raw bits over a physical medium. Data link layer is responsible for handling data frames over a network link. Network layer is responsible for routing data packets between networks. Transport layer is responsible for providing reliable data transfer between applications. Session layer is responsible for organizing communication for multiple sessions taking place at a given time. Presentation layer is responsible for the form of presenting the data to the application. Lastly, the application layer is responsible for providing user-level services to the applications [1].

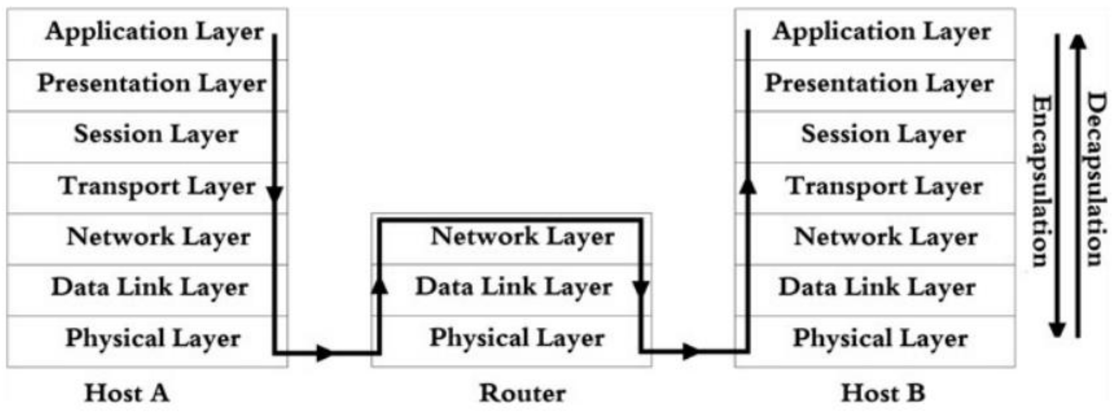


Fig. 1. Overview of OSI model and how data flows between layers. [1]

Different types of networks differ in their coverage area, connectivity and usage. Local Area Network (LAN) or WLAN covers a small area, such as a home, a single building or a campus. Connecting devices such as computers, printers and servers are in a close proximity to each other, and the internet access and resources is shared among the devices. Wide Area Network (WAN) or WWAN on the other hand, covers a larger geographical area than LAN, typically encompassing a city, region or country. It is designed to connect multiple LANs over long distances. Compared to WAN, WWAN have additional functionality of working in conjunction with mobile communication technologies like 4G mobile networks, providing wireless connectivity that is used mainly for mobile devices such as smartphones, tablets, and laptops to access the internet and other online services while on the go [2].

2.1.1 Cellular Network

A cellular network, or mobile network, is a wireless communication system that connects end nodes within a coverage area. Devices on cellular network operate by transmitting and receiving low-power radio signals, which are sent to antennas connected to radio transmitters and receivers, or base stations. To establish communication, mobile users need to be near these base stations, which are situated in cells with limited coverage areas. Providers install numerous cells, each with its own antenna, to cover a larger area and ensure seamless communication. Overlapping cells allow providers to track a user's location, ensuring uninterrupted communication [3].

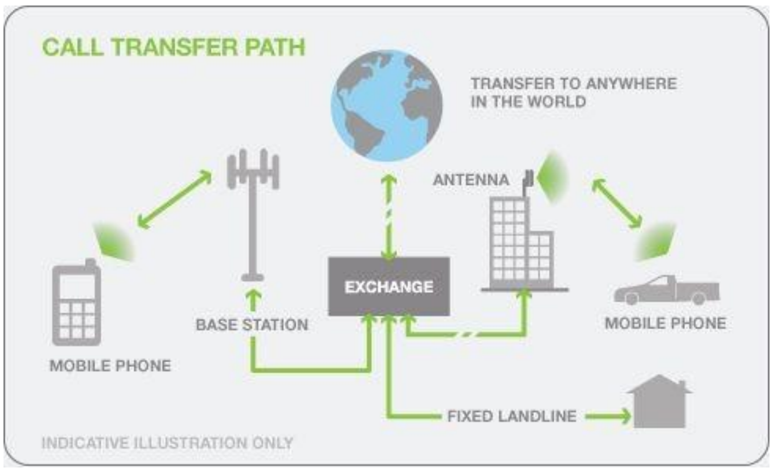


Fig. 2. Overview of cellular networks topology. [3]

Due to the rapid growth of the internet and the surging growth of the use of mobile devices, there were not enough public internet addresses to assign to every individual networking device. As such, Carrier Grade Network Address Translation (CGNAT) was developed as a solution, particularly for service providers.

2.1.2 Carrier Grade Network Address Translation (CGNAT)

CGNAT provides a practical solution to conserve public addresses, which is the primary objective of Network Address Translation (NAT). It operates by having the Internet Service Provider (ISP) assigning the end user a non-publicly routable Internet Protocol (IP) address from a private series of addresses reserved for the ISP. Instead of assigning an actual public IP address to the end user, an intermediate network operated by the ISP is utilized [4].

This allows customer networks to use their own internal network address space and route through the ISP's smaller pool of public Internet IP addresses for internet access. As a result, CGNAT reduces the number of public IP addresses an organization has to use, conserving its IPv4 addresses and allowing great scalability as ISP deploys CGNAT for multiple customers using cellular networks onto a single public IP address [4]. However, due to the dynamic allocation of public IP addresses, customers using CGNAT-type network for their devices may face hindrance in peer-to-peer (P2P) applications due to the constant changing of the device’s public IP address and the restriction of port forwarding by the ISP [5].

2.1.3 Virtual Private Network (VPN)

VPN tunneling is a technique that provides a secure means for users to connect to remote networks via the internet, enabling access to resources on the other end of the tunnel without concerns about security vulnerabilities. It establishes a P2P connection which enables the devices to communicate as if there is no central server or intermediary device to alter the connection in any means. This allows devices behind two different NAT router with different public IP addresses respectively to establish a P2P connection. In addition, VPNs provide encryption that makes it challenging for any unauthorized party to intercept or decipher data outside of the tunnel [6].

2.2 Robot Operating System (ROS)

ROS, short for Robot Operating System, is an open-source software framework specifically designed for robotics application. Despite its name, ROS is not an operating system but rather a software development kit that enables developers to implement the foundational software for any robotic application [7].

Currently there are two versions of ROS: ROS 1 and ROS 2. Since the latest distribution of the ROS 1, ROS Noetic, have an end-of-life date of May 2025 [8], developers are advised to start developing and implementing features using the ROS 2 framework instead. While ROS 1 is sufficient for most academic robotic tasks, it lacks important features for industry usage, such as security, real-time applications, certifications, and safety. In addition, ROS 1 also uses a custom middleware for communication, which can limit performance and scalability. Hence, ROS 2 is developed to address the shortcomings of the original ROS and to implement new technologies such as using Data Distribution Service (DDS) as its middleware [9].

2.2.1 ROS 2 Design

ROS 2 at its core uses the DDS standard middleware for communication between components. DDS defines a standard communication protocol that components follow for data to be shared in a distributed system environment. Under this protocol, the data is published by one component to a global data space tagged to a specific topic. Components requesting for the data from the same topic can subscribe to that topic in the global data space to retrieve the data it needs [10].

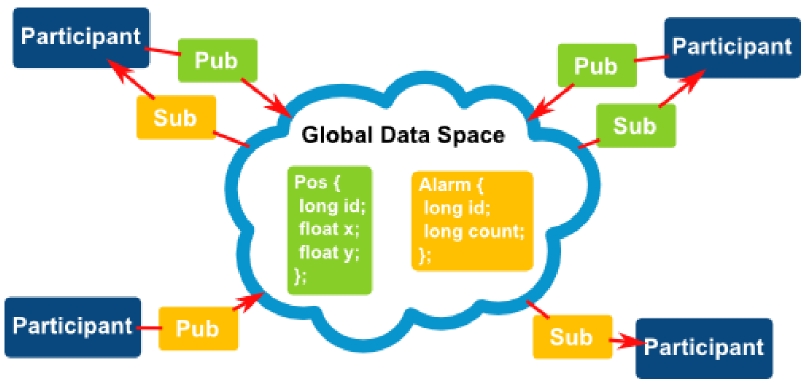


Fig. 3. Overview of DDS architecture. [10]

The fundamentals working of the DDS are mainly based on the OSI model. DDS extends on the features of the OSI model through the implementation of an interoperability protocol, known as the Real-Time Publish Subscribe Protocol (RTPS), on top of the transport layer in the OSI model. The RTPS uses a publisher-subscriber principle whereby the publisher directly transmits the data to subscribers immediately when available. Compared to the request-reply principle where a requester needs to pull the participants for data, the publish-subscribe principle allows for lower latency, higher throughput, better decoupling and a simpler distributed system [10].

To support the publish-subscribe principle of RTPS, sub-protocols Participant Discovery Protocol (PDP) and Endpoint Discovery Protocol (EDP) are used to manage publishers and subscribers. Publisher and subscribers first make themselves discoverable through the PDP, and once participants discover each other, establish endpoints using the EDP. The PDP manages a list of all participants in the system and the topics they publish or subscribe to while the EDP manages a list of all endpoints for a given topic and their communication parameters. This enables RTPS to have both unicast and multicast capabilities, allowing for automatic discovery between publisher and subscribers without needing a centralized server to manage the connection [11].

The RTPS protocol is highly configurable through Quality of Service (QoS) settings, enabling developers to fine-tune message delivery to meet specific real-time requirements, including timed delivery and publisher priority. With PDP and EDP managing participant and endpoint discovery, and a publish-subscribe API layered on top of any transport protocol, such as TCP or UDP, distributed DDS systems are network topology-independent. Publishers can publish asynchronously, and subscribers can retrieve data without the need for explicit subscription. Multiple publishers can also contribute to a topic, with subscribers receiving data prioritized by strength. In the event of a publisher failure, subscribers can fall back to the next strongest publisher [10].

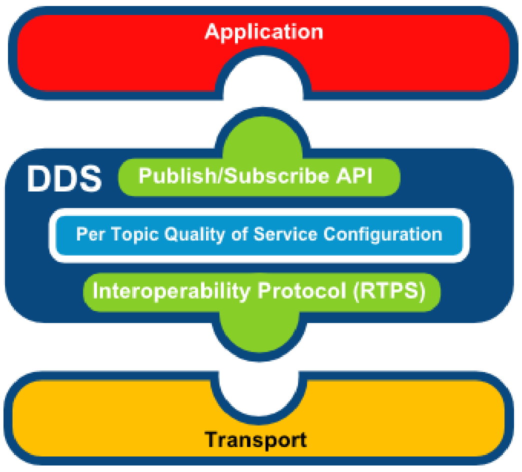

Application
DDS
Publish/Subscribe API
Per Topic Quality of Service Configuration
Interoperability Protocol (RTPS)
Transport

Fig. 4. DDS architecture in the OSI model. [10]

ROS 2 implements the above by providing a message-passing system in the form of ROS topics and ROS nodes to interface with various hardware components in the ROS system. ROS nodes are processes that performs computation while ROS topics are named buses which are used by nodes to exchange information following the DDS standard. Topics can be one-to-one, one-to-many or many-to-many [12].

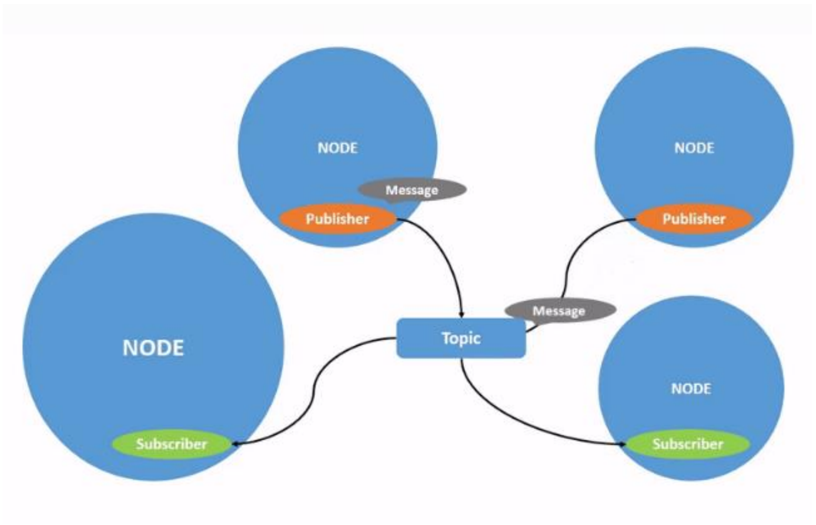


Fig. 5. Overview of ROS nodes and topics in a ROS 2 system. [12]

Besides communication through topics, ROS 2 also provide a call-and-response model called services, which only provide data when specifically requested by a service client in a node, compared to topics which provide continuous updates. However, services can only have one service server with any number of service clients, meaning unlike topics, services can only be one-to-one or one-to-many [13].

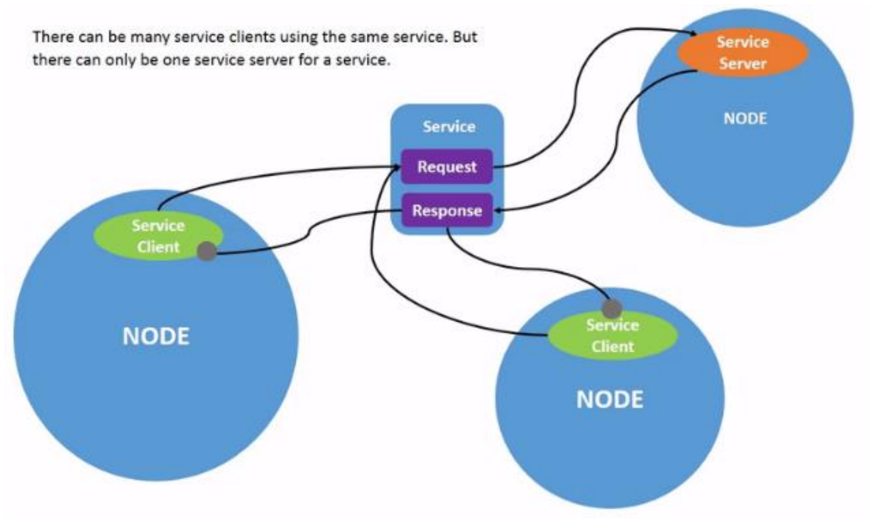

There can be many service clients using the same service. But there can only be one service server for a service.
Service
Request
Response
Service Server
NODE
Service Client
NODE
Service Client
NODE

Fig. 6. Overview of ROS nodes and services in a ROS 2 system. [13]

By combining the publisher-subscriber and call-and-response communication models, nodes can also communicate with one another through actions, which are designed for long running tasks. Actions consist of three parts: a feedback topic, a goal service, and a result service. They performed similarly to services, except that they can be cancelled unlike services and also provides feedback [14].

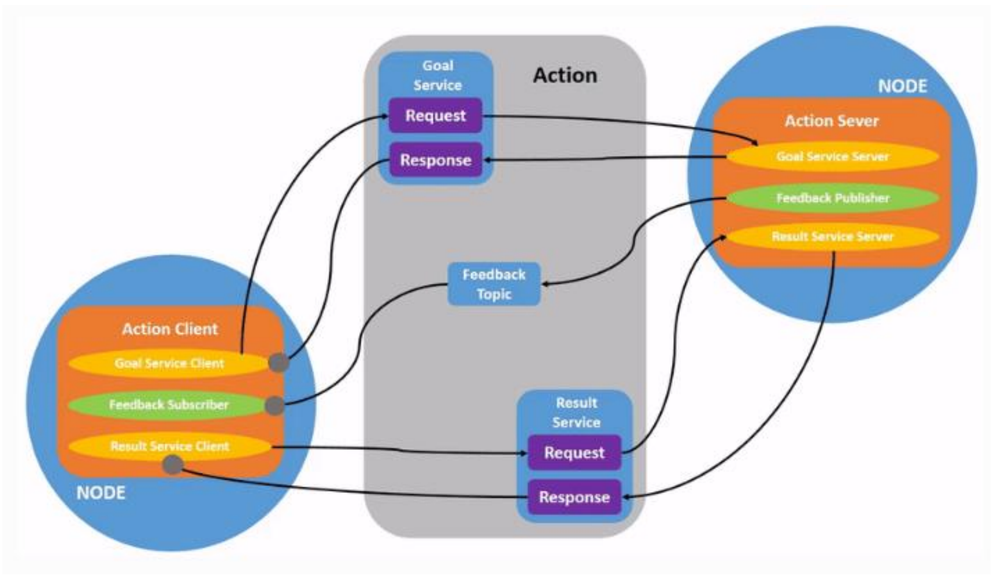


Fig. 7. Overview of ROS nodes and actions in a ROS 2 system adapted from [9]

2.1.2 ROS 2 Tools and Packages

ROS 2 also has a huge repository containing many different types of packages, with each package providing specialized functionality such as hardware abstraction, tools for testing and development, visualisation and more.

2.1.2.1 Cartographer

ROS 2 Cartographer is a package that provides real-time simultaneous localization and mapping, allowing robots to map out the surrounding environment and estimate its position on the mapped environment. To achieve this, the cartographer collects data from various sensors, such as Light Detection and Ranging (LiDAR) and Inertial Measurement Unit (IMU). Depending on its configuration settings, the cartographer can generate a simple 2D map or a complex 3D map with information about the robot's location and orientation data [15].

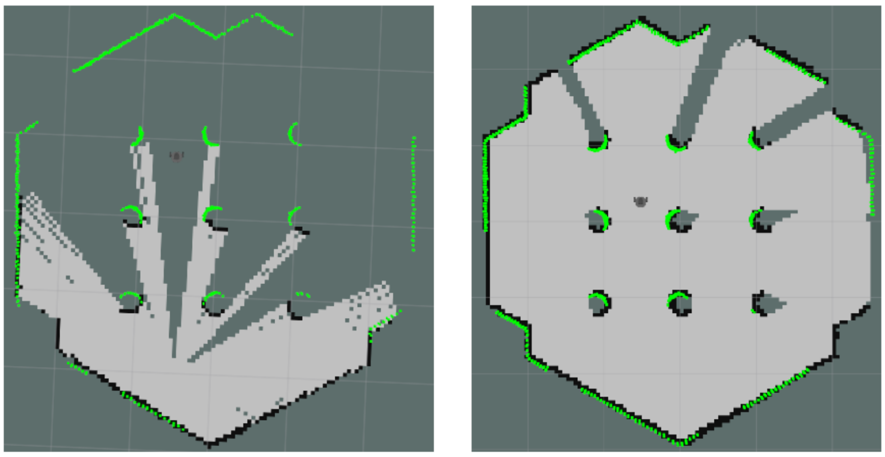


Fig. 8. Mapping of a simple 2D map using LiDAR Sensor Data, adapted from [16].

To visualize the creation of the map, a 3D visualization tool included in the cartographer package called ROS visualization (Rviz) is used. Besides visualizing the creation of the map, Rviz can also visualize robot models, sensor data and camera feed in real-time. It provides a user-friendly interface for users to interact with the various data displayed to allow the user to pan, zoom and rotate the visualization. Rviz also supports displaying data in different formats such as 3D point clouds, trajectory and field of view, making it a great tool for testing and debugging of robotic systems [17].

2.3 Computer Vision

Computer vision is a field of artificial intelligence that focuses on enabling machines to collect visual data from their surroundings to interpret, understand and analyze meaningful information from the data. Many forms of computer vision include image classification, object detection from image or video feed, image or video segmentation and more.

For this project, we are interested in the development of what the robot perceives while we are teleoperating the robot. Thus, we will be focusing on using object detection from a video feed.

2.3.1 Object Detection

Object detection is a computer vision task that automatically identifies and localize objects of interest in an image or video feed. It typically involves two main tasks: the localizing of an object which is to identify its location within the video frame and classification of the object to determine the class or category of the identified object. These tasks are usually carried out by neural networks, most notability conventional neural networks (CNN). There are many techniques to apply and use CNN to carry out object detection, such as region-based methods, single-shot methods and more. Popular examples of region-based methods include Faster Region-based CNN (Faster R-CNN) and Mask R-CNN. Popular examples of single-shot methods include You Only Look Once (YOLO) and Single Shot Detector (SSD) [18].

2.4 Virtual Reality (VR)

VR involves the utilization of computer technology to generate artificial environments, in which the user is fully immersed in a 3D experience, rather than simply viewing it on a screen. By simulating all five human senses, a computer can transport users to new worlds, limited only by the availability of content and the power of the computing system. The three main VR categories are Non-immersive Virtual Reality, Semi-Immersive Virtual Reality, and fully Immersive Virtual Reality [19].

One of the benefits of VR is its capacity to create a fully immersive experience, this allows the user to interact with the environment by tilting their head to get a better viewing angle, and operating the robot in first-person view which results in a more humanely response compared to third-person view. Three-dimensional pictures incorporate depth as an additional dimension, which makes them the most lifelike type of picture. These types of pictures are more realistic in their representation of objects and environments as they mimic how we perceive them using our own eyes. However, a two-dimensional picture is only capable of producing an illusion of depth on the surface that is being displayed [20].

CHAPTER 3: SYSTEM DESIGN

3.1 Hardware Requirements

3.1.1 TurtleBot3 Burger

The TurtleBot3 Burger is a standard platform robot developed by ROBOTIS designed to operate with the ROS framework. It is meant to be a small-sized affordable mobile robot with a modular structure for use in academic, hobby and product prototyping, thus it can be customized depending on the developer's project requirements [21].

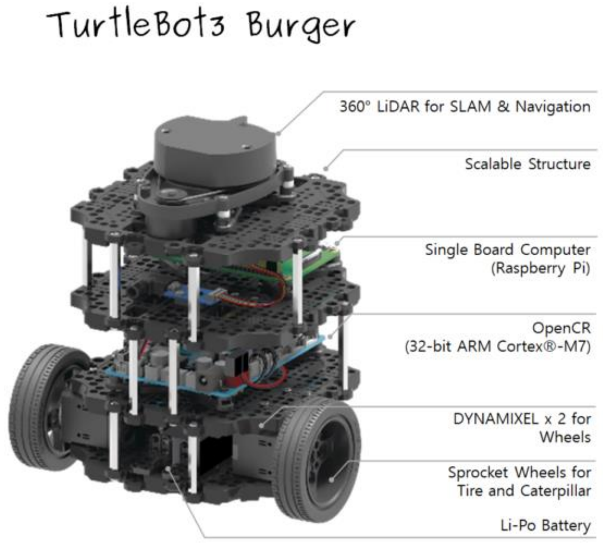

TurtleBot3 Burger
360° LiDAR for SLAM & Navigation
Scalable Structure
Single Board Computer (Raspberry Pi)
OpenCR (32-bit ARM Cortex®-M7)
DYNAMIXEL x 2 for Wheels
Sprocket Wheels for Tire and Caterpillar
Li-Po Battery

Fig. 9. TurtleBot3 Burger Hardware Components. [22]

<u>*3.1.1.1 360° Laser Distance Sensor LDS-01*</u>

The LiDAR sensor used for the Turtlebot3 is the 360° LDS-01 2D laser scanner to be used for the Turtlebot3's SLAM and Navigation. A light emitter inside the LiDAR emits infrared lasers while the LiDAR spins, allowing for a 360° laser coverage. These lasers then bounced off nearby objects and reflected to the capturing infrared sensor in the LiDAR, which will calculate the distance between the object and the LiDAR from the time it takes for the laser to travel from the emitter to the sensor. The orientation of the LiDAR at that instance is also captured. One common application of the LiDAR sensor is using its data and combining it with the orientation data from the Turtlebot3's IMU to map the robot's surrounding area using a cartography software [23].

3.1.1.2 Raspberry Pi Model 3b

The Raspberry Pi Model 3b is a small-form single-board computer that acts as the "brain" of the Turtlebot3. Compared to a conventional PC or laptop, it has many advantages such as the lower cost, smaller form factor and lower power consumption which makes it suitable for robot applications, while maintaining the same flexibility as a PC in terms of programmability.

3.1.1.3 OpenCR 1.0 Controller Board

The OpenCR 1.0 is the main controller board for the TurtleBot3. It provides additional GPIO pins and USB ports on top of the Raspberry Pi existing GPIO pins and USB ports. In the context of Turtlebot3, it is mainly used to control the current sent to the DYNAMIXEL Motors, which are responsible for the Turtlebot3's movement [24].

3.1.1.4 DYNAMIXEL XL430-W250 Motor

The DYNAMIXEL XL430-W250 Motor is used to control the Turtlebot3 movement. It consists of a fully integrated Encoder and DC Motor with network capabilities and a smart PID control algorithm. Depending on the instruction set by the user, the PID control will either be in velocity control mode or position control mode. In the context of Turtlebot3, these instructions are sent by the OpenCR board through the TTL connection bus [25].

When an instruction is received, the motor will register either a goal velocity or goal position depending on the mode, which is then converted to desired velocity or position trajectory respectively by the Profile Velocity and Profile Acceleration. The controller then calculates the PWM output based on the mode and desired parameters. The PWM output is then limited by the limiter base on the Goal PWM which will decide the final PWM value. The final PWM value is then inverted and applied to the motor [25].

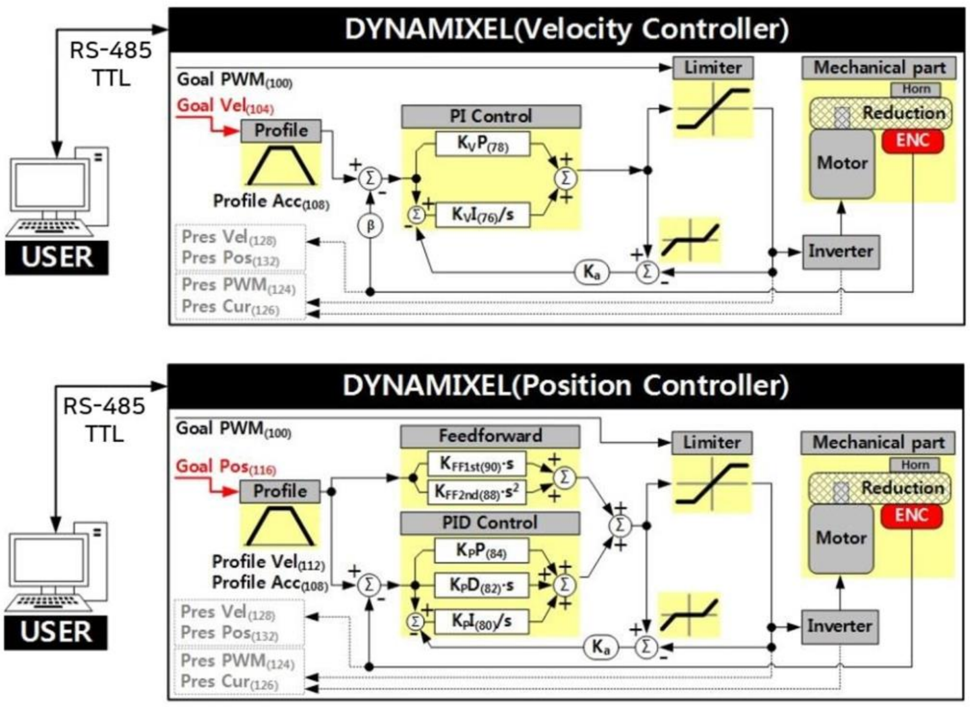


Fig. 10. DYNAMIXEL Velocity and Position Controller flowchart. [25]

3.1.1.5 Li-Po 11.1V 1800mAh LB-012 Rechargeable Battery

The Li-Po battery is used to maintain remote operation of the Turtlebot3 away from any power source. This allows the Turtlebot3 to be operated in a more remote area without any power source for a limited amount of time.

3.1.2 Raspberry Pi Camera Module v2

The Raspberry Pi Camera v2 has a Sony IMX219 8-megapixel sensor [26]. It is an addon module to the Turtlebot3 and is used to capture video stream at the direction the Turtlebot is facing. The video stream can be used for teleoperation and object detection. While it lacks depth perception, the cartographer package which estimates the robot’s position can be used to complement the lack of depth perception of the camera when used together.

3.1.3 Meta Oculus Quest 2

The Meta Quest 2 is an all-in-one virtual reality (VR) system, consisting of a Meta Quest 2 Headset, a left Touch controller, and a right Touch controller.

The headset is capable of hand tracking using the built-in inside-out cameras on the headset to capture the position and orientation of the hands and fingers. This data is then pass through computer vision algorithms to determine the type of gesture the user is portraying [27].

The headset is also capable of headset tracking by tracking the user's head movement along the 6 Degree of Freedom (DoF) in the 3D-space using the built-in inside-out cameras and the data from the headset Inertial Measurement Unit (IMU). The IMU mainly controls the rotational tracking: Roll, Pitch and Yaw of the headset, while the cameras mainly control the positional tracking. The controllers are also tracked along the 6 degrees of freedom, with the rotational tracking done by the controller's IMU and the positional tracking done by the cameras tracking the controllers' positions [28].

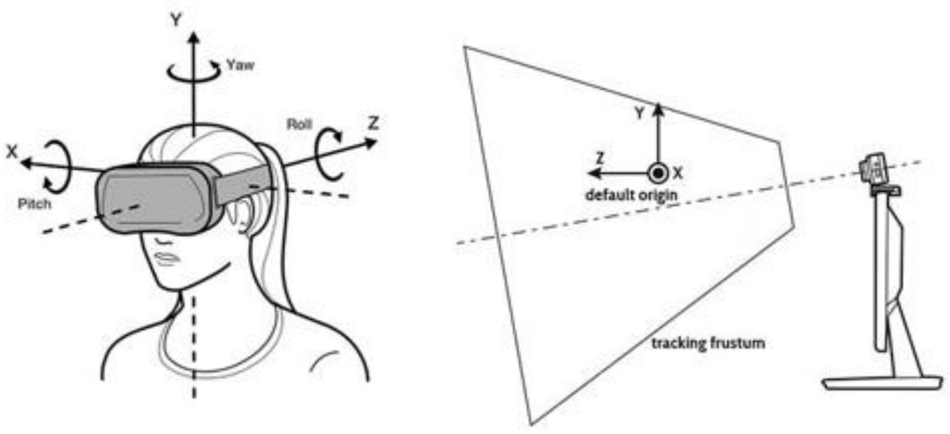


Fig. 11. Rotational and Positional Tracking of a VR Headset. [28]

3.1.4 4G LTE USB Dongle

To simulate remote teleoperation on different networks from the PC, a plug-and-play 4G LTE USB Dongle running a Qualcomm network chip [16] will be used to connect the Turtlebot3 to a mobile carrier network in a remote area.

3.1.5 Miscellaneous Hardware

A PC is required to act as a server to interface the TurtleBot3 connecting from a different network and the Meta Quest 2 together. The PC will also be used to train the object detection models. Hence it is recommended to have a PC with a discrete GPU to train the model with ease. If the PC is connected to a router, ensure that the router have port forwarding capabilities to route any incoming public connections from the TurtleBot3 to the PC on an open port.

A micro-SD card with >32GB of storage space is also required to store the RPi OS image for the TurtleBot3. A micro-SD card reader will be needed to flash the image onto the micro-SD card.

Table 1: List of Hardware Components required.

Hardware Components	**Quantity**
ROBOTIS TurtleBot3 Burger Set	1
Raspberry Pi Camera Module v2	1
Meta Quest 2 Set	1
4G LTE USB Dongle	1
>32GB Micro-SD Card	1
Micro SD Card Reader	1
Personal Computer	1
Personal Router with Port-Forwarding Capabilities	1

3.2 Software Requirements

3.2.1 ROS 2 Humble

For ROS 2, there are only two distributions that is still officially supported by the ROS developers: ROS 2 Foxy Fitzroy and ROS 2 Humble Hawksbill. Since ROS 2 Humble have an end-of-life date of May 2027 compared to ROS 2 Foxy end-of-life date of May 2023, ROS 2 Humble is chosen as the main embedded framework for this project. Humble is primarily supported on Ubuntu 22.04 or Windows 10 with Visual Studio 2019. Since most of ROS 2 packages is designed for Ubuntu, Ubuntu 22.04 is chosen as the OS for development on the PC and it comes with Python 3.10.6 pre-installed. For the DDS middleware, we are using the eprosima Fast DDS.

3.2.2 Libcamera

For the TurtleBot3 RPi, Raspbian Bullseye is the OS chosen for development due to its software library and drivers support for the Raspberry Pi Camera. The camera library software used will be the libcamera and the Picamera2, which come pre-installed with Raspbian Bullseye. As Picamera2 is only officially supported on Python 3.9, Python 3.9 will be the main programming language used for the development of the TurtleBot3 and is also pre-installed along with Raspbian Bullseye.

3.2.3 Roboflow

Roboflow is an AI-powered computer vision platform that helps developers and data scientists to create and manage computer vision models. It supports object detection and classification models. Designed for collaborative work in real-time, this browser-based tool simplifies the task of overseeing annotation projects for team members working across different work streams. It can also easily access data across models built in tensorflow, pytorch, etc and have a strong emphasis on developer-convenience. Roboflow also supports multiple file formats and provides advanced data augmentation capabilities, which helps to improve the accuracy and performance of computer vision models.

3.2.4 Ultralytics YOLOv8

For the software libraries supporting the road signs object detection of the TurtleBot3, YOLOv8 is chosen due to its high accuracy, speed, ease of use, compatibility with PyTorch and is highly configurable. As such, Ultralytics, a python software package containing all the necessary dependencies and libraries necessary for YOLOv5 such as PyTorch, OpenCV, will be used.

3.2.5 Wireguard

While the 4G Dongle specified in section 3.1 is plug-and-play, software setup is still needed for peer-to-peer connection. WireGuard is used to circumvent the issues that were faced when using P2P on the cellular network using its persistence keep-alive policy to transverse the NAT. WireGuard is designed to be used as a general-purpose VPN but is first released for the Linux Kernel. WireGuard employs advanced cryptography techniques and secure trusted constructions. It adopts careful and sound selections and has undergone scrutiny by cryptographers. WireGuard's integration within the Linux kernel and its use of fast cryptographic primitives allow for high-speed secure networking. This makes it appropriate for use in various devices, from small embedded systems like smartphones to heavily loaded backbone routers.

3.2.6 Unity 3D

Unity is a game engine and development platform that is widely used due to its powerful capabilities for creating interactive 2D and 3D experiences across multiple platforms, such as mobile, desktop, console, virtual, and augmented reality devices. One of its strong points is its device compatibility. There is a vast selection of platforms that are compatible, ranging from web and mobile devices to advanced personal computers and gaming consoles. Another strong point of unity is that user have complete control over creating and running the environment due to unity flexible engine and service ecosystem.

Table 2: List of Software Requirements and the versions used.

Software	Version
Debian-based Linux Operating System (PC)	Ubuntu 22.04 LTS (Jammy Jellyfish)
Raspberry Pi OS (TurtleBot3 RPi)	Raspbian Bullseye Lite 64-bit version 5.15
ROS 2 (PC and TurtleBot3 RPi)	ROS 2 Humble Hawksbill
Picamera2	0.3.8 Beta Release 7
Python (PC)	3.10.6 for ROS2 Humble, 3.9.13 Virtual Environments for TurtleBot3 development
Image Label and Annotation	Roboflow Website
Computer Vision Software (PC)	Ultralytics 8.0.10
Virtual Private Network (PC)	WireGuard 1.0.20210914-1ubuntu2 all
Virtual Private Network (TurtleBot3 RPi)	WireGuard 1.0.20210223-1 all
Unity-ROS TCP Endpoint	ROS2 v0.7.0
Meta Quest 2 Software	Meta Quest build 47.0
Unity Editor	2021.3.15f1

3.3 System Design Architecture

For the system network topology, the TurtleBot3 is connected to the PC server through VPN Tunneling Wireguard and FastDDS middleware, with Wireguard providing a virtual P2P connection and FastDDS reducing network usage and eliminating the instability of multicast on not so robust network. The Wireguard software at the TurtleBot3 side will periodically keep the P2P connection alive with a period of about 15 seconds so as to bypass the CGNAT-transversal when there is no network activity. The Oculus Quest 2 is connected to the PC server through a running ROS 2 Node called Unity TCP Endpoint respectively and just like the TurtleBot3, have persistence keep alive policy for the connection with the same period. As such, for the TurtleBot3 and the Oculus Quest 2 to communicate with each other, connections will have to be routed through the PC server first. Because the PC server is not situated behind a CGNAT-grade network, with its port forward capabilities, it acts as a centralized server to route P2P connections between the TurtleBot3 and Oculus Quest 2, bypassing the effect of CGNAT-grade network completely as they can use the PC server to discover each other.

For the algorithms and processes running to support the whole system architecture,when a user push a joystick or engage the head movement, the meta quest will send the inputs as twist message to topic "cmd_vel". This data then travels through the PC server via the Unity TCP endpoint to the Turtlebot3 where the Dynamixel motors listen for any changes in "cmd_vel" topic to engage the motor.

When a video frame is ready from RPI camera, the frame is then sent to topic camera. Both the PC server object detection model and the Meta Quest 2 is subscribed to this topic, as such the data for camera topic is first send to the server through VPN and FastDDS middleware. Then the object detection runs prediction on the video frame. After that, the results and the original video frame is then sent to the Oculus Quest 2 through the ROS-Unity TCP endpoint.

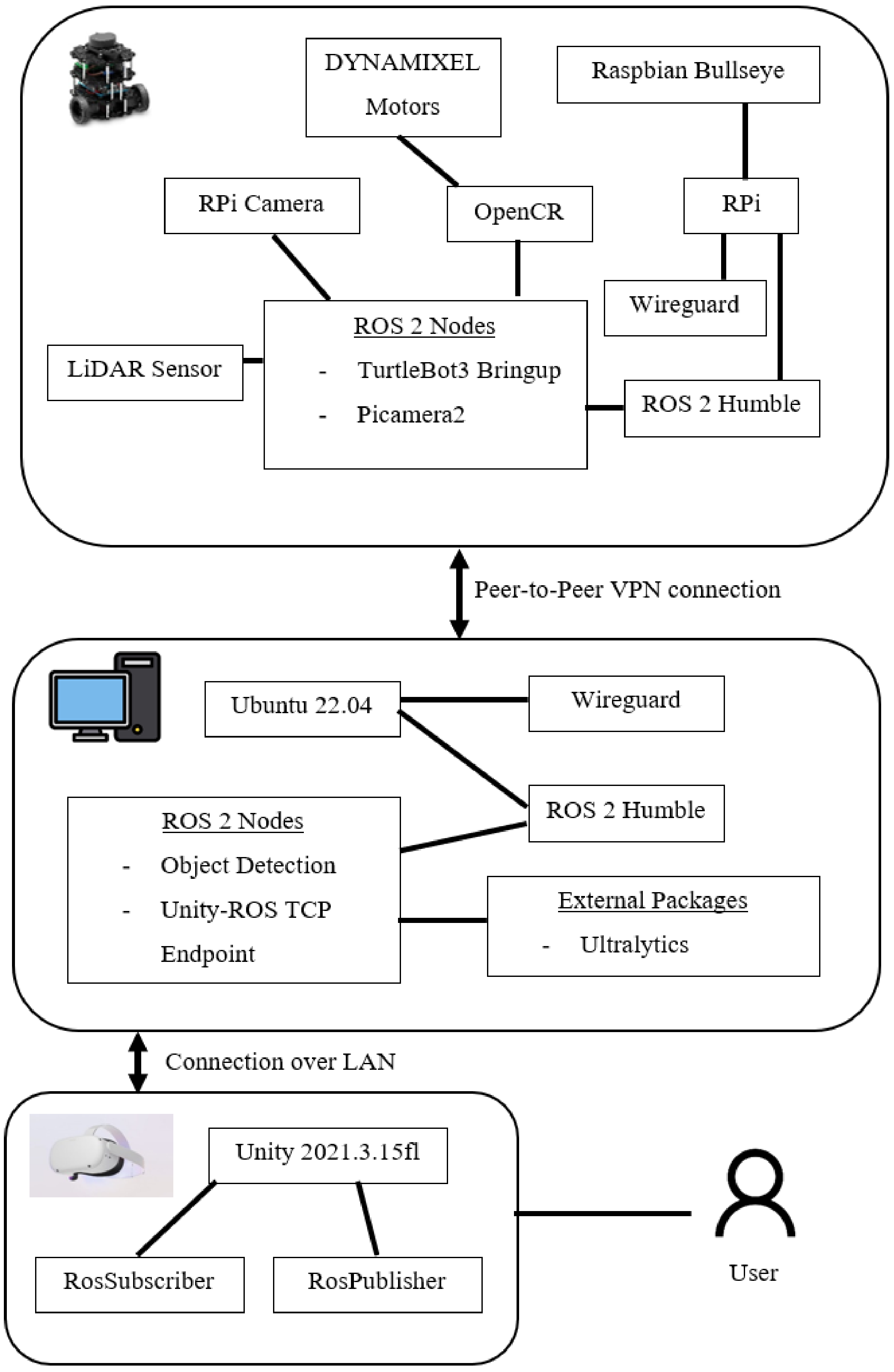


Fig. 12. Overview of System Architecture

CHAPTER 4: SETUP AND IMPLEMENTATION

4.1 Setup

4.1.1 PC Server Setup

A detailed version of setting up the PC Server can be found in Appendix A. After completing Appendix A, the PC Server should be running on Ubuntu 22.04 and have the following installed software: ROS 2 Humble, Wireguard, Ultralytics Python Package, Unity Editor, ROS-Unity TCP Endpoint ROS package and Rosboard ROS package. All packages should be built and ready for deployment. While it is not necessary, it is recommended to have a Nvidia CUDA supported Graphics Processing Unit on the PC so that training and inference of the object detection will be much faster.

4.1.2 Object Detection Setup

As the TurtleBot3 is not tall enough to detect actual road signs besides roads, a printout of the eight road sign classes is used instead. These printouts will be pasted on obstacles, with the bottom edge of the printout about 12 centimeters off the ground so that it is within the TurtleBot3's camera height.

To prepare for the training of the road sign detection model, at least 10 images of each road sign classes is taken using the TurtleBot3's camera at various lighting condition, slight angles and varying magnification. After collecting the dataset of images, Roboflow is used to annotate the images by applying a bounding box on the road sign in the image and tagging it to its respective class. After annotation, the images go through preprocessing by making sure all images are in the correct orientation through auto-orientate. Augmentation of the images of saturation adjustment of between -25% and 25%, brightness adjustment of -25% and 25%, exposure adjustment of -25% and 25% and blurring of up to 2.25 pixels is then applied randomly to the dataset. This will then generate new images which will increase the dataset size by thrice to be use for training. The new dataset can then be exported with a 70-20-10 train-validate-test split along with text annotations and yet another markup language (YAML) configuration file.

After preparing the dataset, the YOLOv8 medium sized model can then be trained for the custom road sign dataset. As Roboflow export automatically sort all the images and annotations into their test, train and validate folders respectively, training the model is as simple as running one command line in the terminal window as shown below. The model is trained for 300 epochs with an early stopping of 100, batch size of 24, image size of 384, caching set to true, translate set to 0, scale set to 0, fliplr set to 0 and mosaic set to 0. These settings were set asmost of the augmentation were already done with Roboflow and we want to detect a whole sign where orientation matters, not part of it. Hence translate, scale and fliplr is set to 0.

```
$   yolo detect train model=yolov8m.pt epochs=300 batch=24 \
    patience=100 imgsz=384 cache=True translate=0 scale=0 \
    fliplr=0 mosaic=0 data=path_to_dataset.yaml
```

4.1.2 TurtleBot3 Setup

Detailed information for TurtleBot3 Setup is presented in Appendix B. After completing Appendix B, the TurtleBot3 should be running on Raspbian Bullseye OS with the following installed software: ROS 2 Humble, TurtleBot3 Bringup ROS Packages and Wireguard. The 4G LTE USB Dongle should be plugged in into the TurtleBot3's RPi after setup.

4.1.3 Meta Quest 2 Setup

The Unity RosSubscriber and RosPublisher script which can be found in Appendix E and F is built in Unity. This will generate a .apk file which must be flash into the Meta Quest 2 to complete the setup.

4.2 Basic Operation

To begin basic operation of the TurtleBot3, all setups must be completed. Then, run the ROS-Unity TCP package and the FastDDS Discovery Server on the PC server using the following terminal commands.

```
$   ros2 run ros_tcp_endpoint default_server_endpoint --ros-args -p
    ROS_IP:=[Insert IPv4 Address here] -p ROS_TCP_PORT:=51800

$   fastdds discovery -i 0
```

Next, login into the TurtleBot3 RPI through SSH and run the following terminal command to bringup the TurtleBot3. This will activate the turtlebot3_node that controls the LiDar Sensor, Motors, PID control, battery monitoring, odometry and PiCamera2.

```
$   ros2 launch turtlebot3_bringup robot.launch.py
```

For basic operation for debugging or testing, only the PC and TurtleBot3 is required. Basic movement of the TurtleBot3 using the keyboard can be activated through the following command in the PC:

```
$   ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

4.3 Movement Operation using Oculus Quest 2

To move the TurtleBot3 using Oculus Quest 2, launch the unity app in the Oculus Quest 2. The app will display a video stream of what the TurtleBot3 is currently looking at through the RPi camera.

Move the left joystick to move the robot. To use the head orientation to rotate the robot, hold down the left joystick grip button. This will disable the left joystick left and right steering and now the head movement of the user will affect the robot's orientation.

4.4 Executing Object Detection

On the PC Server, run the object detection node using the following command:

```
$ ros2 run turtlebot3_picamera2 object_detection
```

The PC will now run the object detection model on a custom dataset consisting of 8 classes of road signs: stop, u-turn, no u-turn, keep left, keep right, no entry, ahead only and split way. Results of the inference will be send to a topic called “obj_det” and the Oculus Quest 2 can displayed the bounding boxes of the results inside the VR.

Chapter 5: Experimentation & Results

5.1 Basic Operation Analysis

After bringing up the TurtleBot3, the CPU usage of the robot is at approximately 3.12 out of the 4 available threads, and the amount of free memory is about 300MB. This shows that just the bringup operation alone is taking up a lot of resources of the TurtleBot3. Hence, it is not advisable to run the object detection model on the TurtleBot3 itself as not only will the object detection model not run well, but also it degrades the performance of the basic operation of the Turtlebot3, which may cause potential issues in the movement and data collection of the sensors. Most of the resources are taken up by the Picamera2 node and the Picamera2 node is able to broadcast a smooth video feed to the PC server at an average of 23 frames per second (fps).

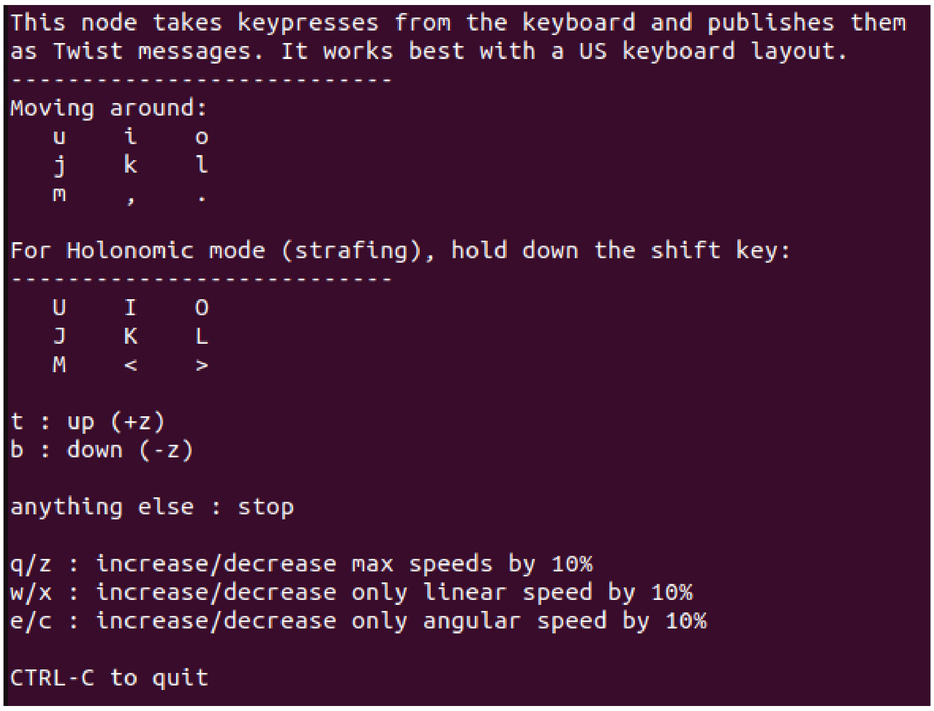
5.2 Keyboard versus VR Control Analysis

After bringing up the TurtleBot3 and running the ROS 2 keyboard teleoperation twist node on the PC, the user can control the robot remotely based on the inputs in figure 13. The robot is able to smoothly respond to the user inputs under normal conditions.

Another way to control the TurtleBot3 is to use the Oculus Quest 2 after running the Unity-ROS TCP Endpoint node and booting up the Unity application in the Oculus Quest 2. Using the Oculus Quest 2 to control the robot will naturally prevent the user from controlling the TurtleBot3 using the keyboard if the keyboard teleoperation twist node is still running as the user will be immersed in the VR world due to the VR headset blocking the user’s sight from the outside world.

Using the joystick only to move the TurtleBot3 is a little unstable as the joystick is small and very sensitive to a tiny amount of force. Thus, the robot will not be able to keep on a straight path during teleoperation by only using the joystick to move the robot. Meanwhile, the head movement of the user can only rotate and orientate the robot as the head movement does not have the degree of freedom for positional movement. Hence, the head movement can only be used to accurately pivot the robot in place.

The best way to control the robot is to combine both the joystick and head movement principles, by using the joystick to adjust the linear position of the robot and using the headset movement to precisely adjust the rotation and orientation of the robot. Comparing with the keyboard method, the joystick and head movement combo feels more immersive and allow the user to view the camera feed with object detection feedback. With the feedback from the camera feed, the user is able to make micro adjustments to the position of the robot.



```
This node takes keypresses from the keyboard and publishes them
as Twist messages. It works best with a US keyboard layout.
---------------------------
Moving around:
   u    i    o
   j    k    l
   m    ,    .

For Holonomic mode (strafing), hold down the shift key:
---------------------------
   U    I    O
   J    K    L
   M    <    >

t : up (+z)
b : down (-z)

anything else : stop

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%

CTRL-C to quit
```

Fig. 13. Overview of controls for keyboard teleoperation

5.3 Movement Analysis

Regardless of which control method is used, the TurtleBot3 is able to respond to the control inputs smoothly due to the PID control software on the Dynamixel motors. However, it requires the TurtleBot3 to be operating under normal conditions which include a flat and smooth surface, a robust internet connection between the TurtleBot3 and the server and the condition of not bumping into an obstacle.

If the TurtleBot3 encounters bumpy terrain, it can cause the robot to veer off course or get stuck and user inputs will not be as effective in controlling the robot. If the internet connection is not robust, it may cause connectivity issues whereby packets is dropped and this may affect the TurtleBot3 node running in the TurtleBot3, causing errors in the motor’s calculation. Getting the TurtleBot3 stuck or bumping into obstacles may also cause errors in the motor’s calculation. This will cause the motor to malfunction and shutdown, which will sound an alarm in the OpenCR board. At this point, the user will have to reboot the TurtleBot3 and bring it up again to solve the issue.

5.4 Network Analysis

To test the network connection between the TurtleBot3 or Oculus Quest 2 and the PC server, a ping command from the linux iputils package is use from the server to the two entities respectively. The following is the CLI command to activate the iputils ping command:

```
$ ping [IP address of entity]
```

From a round trip, the ping is able to achieve an average of 32.961 ms with 0% packet lost. When calculating a round trip from the Oculus Quest 2 to the TurtleBot3 via the server and back, adding on with a model inference time of 15ms, the ping from the TurtleBot3 to the Oculus Quest 2 and back can achieve an average of about 100ms.

5.5 Object Detection Analysis

After training the YOLOv8 model for 300 epochs with an early stopping patience of 100, the model managed to achieve a mean average precision 50 (mAP50) of 0.89135 of all classes and a mAP50-95 of 0.80272 at epoch 57. This means the object detection model is good enough for use.

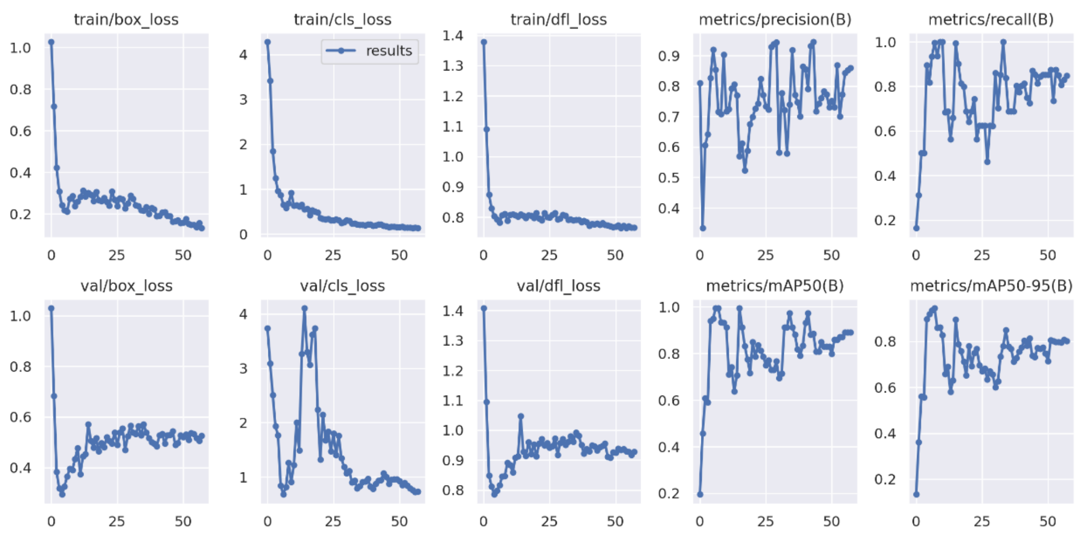


Fig. 14. Results of training model

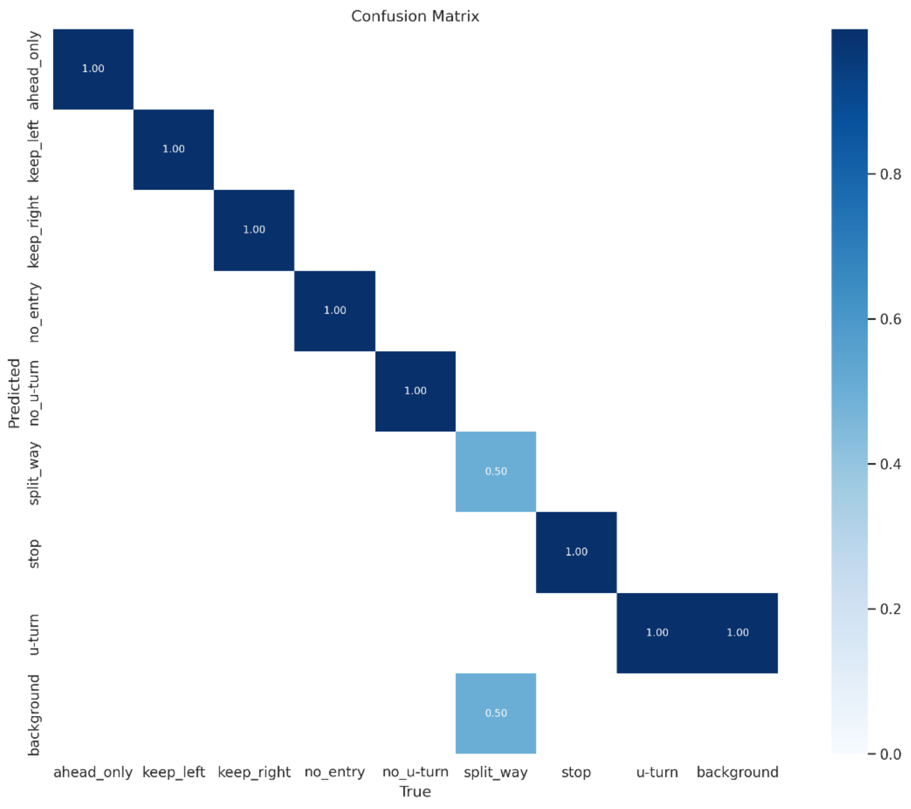


Fig. 15. Confusion Matrix

CHAPTER 6: CONCLUSION AND FUTURE WORKS

6.1 Conclusion

To conclude, this project has been successful in achieving its main objective which is develop a robot capable of remote teleoperation and object detection to allow the operator to reliably control the robot's movement and be updated about what the robot sees and detects through a video livestream over carrier grade networks.

The TurtleBot3 has good responsiveness of not more than 400ms through a 4G network and the user was able to view the video feed properly in VR, albeit there are no usage of VR elements as the Oculus Quest 2 is used more as a video projector. The object detection, while could be better, is acceptable enough to possibly apply more sophisticated algorithms to take advantage of its ability to perceive road signs.

6.2 Limitations

Achieving the above is not without its own sets of challenges. Many of ROS famous packages have not port forward from ROS 1 to ROS 2 and there is a huge API shift in the embedded market currently, such as with Raspberry Pi camera changing its core API. Thus, some packages like the picamera package have to be written from scratch despite it being readily available in ROS 1. Also, moving to ROS 1 would be just choosing which poison to consume as ROS 1 lacks real-time support and networking capabilities like DDS which may greatly impact the progress of this project.

Moreover, the TurtleBot3 is limited by its features such as its inability to climb stairs or scale terrain, which may affect certain applications of remote teleoperations. ISP may also not provide coverage on remote area and hence will affect remote teleoperation in said area. The object detection accuracy may be severely affected by environment conditions such as clutter, poorly lit areas or when objects are overlapping.

6.3 Future Works

Some future works to extend on this project includes applying SLAM, which will allow the TurtleBot3 to gain depth-perception and a sense of its surrounding areas. This will in turn allow for more advanced use case of the TurtleBot3 such as autonomous navigation, autonomous surveillance with object detection and even support the potentially expanding industry of AVs by using the TurtleBot3 as a test platform or an autonomous driving simulator.

References

[1] M. M. Alani, 'OSI Model', *Guide OSI TCPIP Models*, pp. 5–17, 2014, doi: 10.1007/978-3-319-05152-9_2.

[2] S. Yellampalli, *Wireless Sensor Networks: Design, Deployment and Applications*. BoD – Books on Demand, 2021.

[3] 'How mobile networks work', *AMTA | The Voice of the Australian Mobile Telecommunications Industry*, Mar. 03, 2020. https://amta.org.au/1041-2/ (accessed Mar. 20, 2023).

[4] R. Sabin, 'What is CGNAT and Which One is Best for My Network?', *netElastic.com*, Jan. 19, 2022. https://netelastic.com/what-is-cgnat-and-which-one-is-best-for-my-network/ (accessed Mar. 20, 2023).

[5] B. Group, 'Pros and Cons of Deploying Carrier Grade NAT', *Buy & Sell IP Addresses | Brander Group*, Jan. 04, 2023. https://brandergroup.net/2023/01/benefits-and-issues-deploying-carrier-grade-network-address-translation-cgnat/ (accessed Mar. 20, 2023).

[6] K. Kimachia, 'What Is Tunneling in Networking? Definition & How It Works', *Enterprise Networking Planet*, Mar. 03, 2023. https://www.enterprisenetworkingplanet.com/standards-protocols/what-is-tunneling/ (accessed Mar. 20, 2023).

[7] 'ROS: The ROS Ecosystem', *ROS*. https://www.ros.org/blog/ecosystem/ (accessed Mar. 18, 2023).

[8] 'Distributions - ROS Wiki', *ROS*. http://wiki.ros.org/Distributions (accessed Mar. 18, 2023).

[9] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, 'Robot Operating System 2: Design, architecture, and uses in the wild', *Sci. Robot.*, vol. 7, no. 66, p. eabm6074, May 2022, doi: 10.1126/scirobotics.abm6074.

[10] 'DDS', *eProsima The Middleware Experts*. https://www.eprosima.com/index.php/resources-all/whitepapers/dds (accessed Mar. 18, 2023).

[11] 'RTPS', *eProsima The Middleware Experts*. https://www.eprosima.com/index.php/resources-all/whitepapers/rtps (accessed Mar. 18, 2023).

[12] 'Understanding topics', *ROS 2 Documentation: Humble documentation*. https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html (accessed Mar. 18, 2023).

[13] ‘Understanding services', *ROS 2 Documentation: Humble documentation*. https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html (accessed Mar. 18, 2023).

[14] ‘Understanding actions', *ROS 2 Documentation: Humble documentation*. https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html (accessed Mar. 18, 2023).

[15] ‘Algorithm walkthrough for tuning', *Cartographer ROS documentation*. https://google-cartographer-ros.readthedocs.io/en/latest/algo_walkthrough.html (accessed Mar. 19, 2023).

[16] ‘SLAM — TurtleBot3’, *ROBOTIS e-Manual*. https://emanual.robotis.com/docs/en/platform/turtlebot3/slam/ (accessed Mar. 19, 2023).

[17] ‘rviz', *ROS*. http://wiki.ros.org/rviz (accessed Mar. 19, 2023).

[18] Z. Zou, K. Chen, Z. Shi, Y. Guo, and J. Ye, ‘Object Detection in 20 Years: A Survey’. arXiv, Jan. 18, 2023. Accessed: Mar. 19, 2023. [Online]. Available: http://arxiv.org/abs/1905.05055

[19] J. Bardi, ‘Virtual Reality Defined & Use Cases’, *3D Cloud by Marxent*, Mar. 26, 2019. https://www.marxentlabs.com/what-is-virtual-reality/ (accessed Mar. 20, 2023).

[20] E. Team, ‘What Are the Advantages and Disadvantages of Virtual Reality?', *Metaverse VR Now*, Jun. 16, 2022. https://metaversevrnow.com/vr/advantages-and-disadvantages-of-virtual-reality/, https://metaversevrnow.com/vr/advantages-and-disadvantages-of-virtual-reality/ (accessed Mar. 20, 2023).

[21] ‘Overview — TurtleBot3’, *ROBOTIS e-Manual*. https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/ (accessed Mar. 19, 2023).

[22] ‘Features — TurtleBot3’, *ROBOTIS e-Manual*. https://emanual.robotis.com/docs/en/platform/turtlebot3/features/ (accessed Mar. 19, 2023).

[23] ‘LDS-01 — TurtleBot3’, *ROBOTIS e-Manual*. https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_01/ (accessed Mar. 19, 2023).

[24] ‘OpenCR1.0 — TurtleBot3’, *ROBOTIS e-Manual*. https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_opencr1_0/ (accessed Mar. 19, 2023).

[25] ‘XL430-W250-T’, *ROBOTIS e-Manual*. https://emanual.robotis.com/docs/en/dxl/x/xl430-w250/ (accessed Mar. 19, 2023).

[26] ‘Camera’, *Raspberry Pi Documentation*. https://www.raspberrypi.com/documentation/accessories/camera.html (accessed Mar. 19, 2023).

[27] ‘Controllers and Traking’, *Meta*. https://www.meta.com/help/quest/articles/headsets-and-accessories/controllers-and-hand-tracking/ (accessed Mar. 19, 2023).

[28] A. Jaishanker, ‘Virtual Reality 101 – The different types of VR Headsets’, *YourStory.com*, Jun. 21, 2016. https://yourstory.com/2016/06/virtual-reality-headset (accessed Mar. 20, 2023).

Appendix A

1. Install Ubuntu 22.04, enable third-party boot
2. Install ROS 2 Humble (follow website)
3. Install relevant graphics drivers
    a. NVIDIA: CUDA 12.0 and CUDNN 8.8.1.3
4. Install prerequisites
    a. $sudo apt install python3-pip
5. Install packages ultralytics
    a. $sudo python -m pip install ultralytics
6. Setup WireGuard

APPENDIX B

1. Prepare SD Card
    a. Download RPI Imager
    b. Choose RPi OS Bullseye Lite 64-Bit
    c. Install Image onto SD Card
    d. After flashing, insert SD Card into Turtlebot3 RPi and boot, following the setup instructions on first boot.
2. Setting up Turtlebot3
    a. Head into raspi-config through sudo raspi-config
    b. Set Wifi connection, Auto Login on Boot, Disable Network on boot, set timezone and enable ssh
    c. Add Swap Space

```
$ sudo fallocate -l 2G /swapfile

$ sudo chmod 600 /swapfile

$ sudo mkswap /swapfile

$ sudo swapon /swapfile

$ sudo nano /etc/fstab

  - append “/swapfile swap swap defaults 0 0”
```

    d. Update Turtlebot3
3. Downloading Turtlebot3 Packages
    a. Install Pre-Requisites for Python PIP
    b. Install Pre-Requisites for ROS 2

```
$ sudo apt update

$ sudo apt upgrade -y

$ sudo apt install software-properties-common python3-pip git

$ sudo python3 -m pip install -U flake8-blind-except flake8-builtins flake8-class-
newline flake8-comprehensions flake8-deprecated flake8-import-order flake8-
```

```
quotes pytest-repeat pytest-rerunfailures opencv-python colcon-common-extensions ultralytics
```

c. Install ROS 2 Humble for RPi

```
$ wget https://github.com/Ar-Ray-code/rpi-bullseye-ros2/releases/download/ros2-0.3.1/ros-humble-desktop-0.3.1_20221218_arm64.deb
$ sudo apt install ./ros-humble-desktop-0.3.1_20221218_arm64.deb
$ source /opt/ros/humble/setup.bash
```

d. Install Turtlebot3 Packages for ROS 2 Humble (Will take about 3 hour to build)

```
$ mkdir installations
$ cd installations
$ git clone -b humble-devel https://github.com/ROBOTIS-GIT/turtlebot3.git
$ git clone -b humble-devel https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
$ git clone -b humble-devel https://github.com/ROBOTIS-GIT/hls_lfcd_lds_driver.git
$ git clone -b humble-devel https://github.com/ROBOTIS-GIT/DynamixelSDK.git
$ colcon build --symlink-install --parallel-workers 1
$ source install/setup.bash
```

e. Add to .bashrc

```
export ROS_DOMAIN_ID=30 #TURTLEBOT3
export LDS_MODEL=LDS-01
```

f. Add rules for USB?

```
$ sudo cp {PATH}/turtlebot3_bringup/script/99-turtlebot3-cdc.rules
/etc/udev/rules.d/

$ sudo udevadm control --reload-rules

$ sudo udevadm trigger
```

g. Setup OpenCR Board

```
$ sudo dpkg --add-architecture armhf

$ sudo apt update

$ sudo apt install libc6:armhf

$ export OPENCR_PORT=/dev/ttyACM0

$ export OPENCR_MODEL=burger

$ rm -rf ./opencr_update.tar.bz2

$ wget https://github.com/ROBOTIS-GIT/OpenCR-
Binaries/raw/master/turtlebot3/ROS2/latest/opencr_update.tar.bz2

$ tar -xvf ./opencr_update.tar.bz2

$ cd ~/opencr_update

$ ./update.sh $OPENCR_PORT $OPENCR_MODEL.opencr
```

h. Setup Wireguard

Appendix C

Source Code for Picamera2 node

```python
import rclpy
from rclpy.node import Node
import time
import cv2
import io

from picamera2 import MappedArray, Picamera2
from picamera2.encoders import JpegEncoder
from picamera2.outputs import FileOutput
from sensor_msgs.msg import CompressedImage
from threading import Condition


class StreamingOutput(io.BufferedIOBase):
    def __init__(self):
        self.frame = None
        self.condition = Condition()

    def write(self, buf):
        with self.condition:
          self.frame = buf
          self.condition.notify_all()

class PiCam2(Node):
  def __init__(self):
    super().__init__('picam2')
    self.publisher_camera = self.create_publisher(CompressedImage, 'camera', 1)
    self.msg = CompressedImage()
    self.msg.format = '.jpeg'

    self.camera = Picamera2()
    self.camera.configure(self.camera.create_video_configuration(
        main={"size": (512, 384)}
    ))
    self.output = StreamingOutput()

    self.camera.post_callback = self.apply_timestamp
    self.camera.start_recording(JpegEncoder(), FileOutput(self.output))
    self.timer = self.create_timer(0.010, self.update_callback)
    self.get_logger().info("PiCamera initialised.")

  def apply_timestamp(self, request):
    with MappedArray(request, "main") as m:
      timestamp = time.strftime("%Y-%m-%d %X")
      cv2.putText(
          m.array,
          timestamp,
          (0, 15),
          cv2.FONT_HERSHEY_SIMPLEX,
          0.5,
          (0, 255, 0),
          2
      )

  def update_callback(self):
    with self.output.condition:
        self.output.condition.wait()
```

APPENDIX D

Source Code for Object Detection Node

```python
import rclpy
from rclpy.node import Node
import numpy as np
import cv2
import json

from std_msgs.msg import String
from sensor_msgs.msg import CompressedImage
from ultralytics import YOLO

class ObjectDetectionNode(Node):
  def __init__(self):
    super().__init__('obj_det')

    self.publisher = self.create_publisher(String, 'obj_det_info', 1)
    self.msg = String()
    self.subscription = self.create_subscription(
      CompressedImage, 'camera', self.listener_callback, 1
      )

    directory = "/home/fjunming/"
    self.model = YOLO(directory + "best.pt")

    self.classes = [
      'ahead_only',
      'keep_left',
      'keep_right',
      'no_entry',
      'no_u-turn',
      'split_way',
      'stop',
      'u-turn'
      ]

    self.rect = []
    self.get_logger().info("Object Detection initialised.")

  def listener_callback(self, msg):
    img = cv2.cvtColor(
      cv2.imdecode(np.frombuffer(msg.data, np.uint8), cv2.IMREAD_COLOR),
      cv2.COLOR_BGR2RGB
      )

    results = self.model(img)
    self.rect = []

    for result in results:
        conf = result.boxes.conf.detach().cpu().numpy()
        boxes = result.boxes.xyxy.detach().cpu().numpy()
        cls = result.boxes.cls.detach().cpu().numpy()

        for i, c in enumerate(conf):
            if c > 0.7:
                temp = {
                      "name": self.classes[int(cls[i])],
                      "conf": int(c * 100),
                      "xy1": {"x": int(boxes[i][0]), "y": int(boxes[i][1])},
                      "xy2": {"x": int(boxes[i][2]), "y": int(boxes[i][3])},
                }
                self.rect.append(temp)
                pass


    export = {"objdetinfo": self.rect}
    self.msg.data = json.dumps(export)
    self.publisher.publish(self.msg)
    self.rect = []


def main():
  rclpy.init()
  obj_det = ObjectDetectionNode()
  rclpy.spin(obj_det)

  obj_det.destroy_node()
  rclpy.shutdown()

if __name__ == "__main__":
  main()
```

Appendix E

Source Code for Unity RosPublisher Script

```
using UnityEngine;
using Unity.Robotics.ROSTCPConnector;
using RosMessageTypes.Geometry;
using UnityEngine.XR.Interaction.Toolkit;
using UnityEngine.XR;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.Events;
using OdomMsg = RosMessageTypes.Nav.OdometryMsg;

public class RosPublisherExample : MonoBehaviour
{
    ROSConnection ros;
    public string topicSendValues = "cmd_vel";

    Vector3Msg linear = new Vector3Msg(0.0, 0.0, 0.0);
    Vector3Msg angular = new Vector3Msg(0.0, 0.0, 0.0);

    private InputDevice left_controller;
    private InputDevice head_device;

    void Start()
    {
        // start the ROS connection
        ros = ROSConnection.GetOrCreateInstance();
        ros.RegisterPublisher<TwistMsg>(topicSendValues);

        var gameControllers = new List<UnityEngine.XR.InputDevice>();
        UnityEngine.XR.InputDevices.GetDevicesWithCharacteristics(
            UnityEngine.XR.InputDeviceCharacteristics.HeadMounted, gameControllers
        );
        head_device = gameControllers[0];

        while (left_controller == null)
        {
            gameControllers = new List<UnityEngine.XR.InputDevice>();
            UnityEngine.XR.InputDevices.GetDevicesWithCharacteristics(
                UnityEngine.XR.InputDeviceCharacteristics.Left, gameControllers
            );
            if (gameControllers.Count > 0)
            {
                left_controller = gameControllers[0];
                break;
            }
            Debug.Log("Waiting for left controller connection...");
        }
    }


    private void InputDevices_deviceConnected(InputDevice device)
    {
        Debug.Log("New Connection");
        if ((device.characteristics & InputDeviceCharacteristics.Left)
            == InputDeviceCharacteristics.Left)
        {
            left_controller = device;
        }
    }

    private void Update()
    {

        var gameControllers = new List<UnityEngine.XR.InputDevice>();
        UnityEngine.XR.InputDevices.GetDevicesWithCharacteristics(
            UnityEngine.XR.InputDeviceCharacteristics.Left, gameControllers
        );
        if (gameControllers.Count > 0)
        {
            left_controller = gameControllers[0];

            //Conventional Joystick Control
            Vector2 joy_val = new Vector2();
            if (left_controller.TryGetFeatureValue(
                UnityEngine.XR.CommonUsages.primary2DAxis, out joy_val
                ))
            {
                linear.x = joy_val.y / 10.0;
                angular.z = -joy_val.x / 3.0;
            }

            //Head-movement Control
            bool triggered; //To check if left grip button pressed
            if (left_controller.TryGetFeatureValue(
                UnityEngine.XR.CommonUsages.gripButton, out triggered
                ) && triggered)
            {
                Vector3 ang_v = new();
                if (head_device.TryGetFeatureValue(
                    UnityEngine.XR.CommonUsages.deviceAngularVelocity, out ang_v
                    ))
                {
                    if (ang_v.y > 1) ang_v.y = 1;
                    else if (ang_v.y < -1) ang_v.y = -1;
                    angular.z = ang_v.y;
                }
            }

            TwistMsg msg = new TwistMsg(linear, angular);
            ros.Publish(topicSendValues, msg);
        }
    }
}
```

Appendix F

Source Code for Unity RosSubscriber Script

```
using UnityEngine;
using Unity.Robotics.ROSTCPConnector;
using CImageMsg = RosMessageTypes.Sensor.CompressedImageMsg;
using StringMsg = RosMessageTypes.Std.StringMsg;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.UI;
using TMPro;


[System.Serializable]
public class ObjectDetectInfo
{
    public Vector2 xy1;
    public Vector2 xy2;
    public string name;
    public float conf;

    public ObjectDetectInfo(Vector2 xy1, Vector2 xy2, string name, float conf)
    {
        this.xy1 = xy1;
        this.xy2 = xy2;
        this.name = name;
        this.conf = conf;
    }
}

[System.Serializable]
public class ObjectDetectInfoArray
{
    public ObjectDetectInfo[] objdetinfo;
}

public class RosSubscriberExample : MonoBehaviour
{
    public GameObject boxClonerObject;
    public GameObject spawnBoxTarget;

    public GameObject objectClassPrefab;
    public List<GameObject> boxDataList;

    public RawImage img;
    public string topicImage = "camera";
    public string topicObjDet = "obj_det_info";

    void Start()
    {
        ROSConnection.GetOrCreateInstance()
            .Subscribe<CImageMsg>(topicImage, ImageUpdate);
        ROSConnection.GetOrCreateInstance()
            .Subscribe<StringMsg>("obj_det_info", ObjdetUpdate);
    }
```

```

void ImageUpdate(CImageMsg msg)
{
    Texture2D tex = new Texture2D(384, 384);
    tex.LoadImage(msg.data);
    img.texture = tex;
}

void ObjdetUpdate(StringMsg msg)
{
    Debug.Log(msg);
    Debug.Log("screen res: " + Screen.width + " x " + Screen.height);
    if (boxDataList.Count > 0)
    {
        foreach (GameObject temp in boxDataList)
        {
            Destroy(temp.gameObject);
        }
        boxDataList.Clear();
    }

    ObjectDetectInfoArray list_info =
        JsonUtility.FromJson<ObjectDetectInfoArray>(msg.data);


    for (int i = 0; i < list_info.objdetinfo.Length; i++)
    {
        GameObject tempEditer = Instantiate(
            boxClonerObject, spawnBoxTarget.transform
        );
        boxDataList.Add(tempEditer);

        RectTransform thisCanvas = tempEditer
            .GetComponent<RectTransform>();
        RectTransform Top = tempEditer.transform
            .GetChild(0).GetComponent<RectTransform>();
        RectTransform Bottom = tempEditer.transform
            .GetChild(1).GetComponent<RectTransform>();
        RectTransform Right = tempEditer.transform
            .GetChild(2).GetComponent<RectTransform>();
        RectTransform Left = tempEditer.transform
            .GetChild(3).GetComponent<RectTransform>();

        float thickness = 1;
        Vector2 boxPoint1;
        Vector2 boxPoint2;

        boxPoint1 = list_info.objdetinfo[i].xy1;

        //Adds an end point of the box on the mouse position.
        //(Bottom-Right corner)
        boxPoint2 = list_info.objdetinfo[i].xy2;

        //Checks if the box start point is smaller than end point,
        //If true, hides.
        //Also ensures to never exceed past start point.
        if (boxPoint1.x + thickness > boxPoint2.x
            && boxPoint1.y + thickness > boxPoint2.y)
        {
            boxPoint2 = boxPoint1;
            tempEditer.SetActive(false);
        }
        else
        {
            if (boxPoint1.x + thickness > boxPoint2.x)
            {
                boxPoint2 = new Vector2(
                    boxPoint1.x + thickness, boxPoint2.y
                );
            }
            else if (boxPoint1.y + thickness > boxPoint2.y)
            {
                boxPoint2 = new Vector2(
                    boxPoint2.x, boxPoint1.y + thickness
                );
            }
            else
            {
                tempEditer.SetActive(true);
            }
        }

        //Drawing the Box.
        Vector2 finalboxPoint2 = new Vector2(
            thisCanvas.rect.width - boxPoint2.x,
            thisCanvas.rect.height - boxPoint2.y
        );

        Top.offsetMin = new Vector2(
            boxPoint1.x,
            thisCanvas.rect.height - boxPoint1.y - thickness
        ); // left, bottom

        Top.offsetMax = -new Vector2(
            finalboxPoint2.x, boxPoint1.y
        ); // right, top

        Bottom.offsetMin = new Vector2(
            boxPoint1.x, finalboxPoint2.y
        ); // left, bottom

        Bottom.offsetMax = -new Vector2(
            finalboxPoint2.x,
            thisCanvas.rect.height - finalboxPoint2.y - thickness
        ); // right, top

```

```
            Right.offsetMin = new Vector2(
                thisCanvas.rect.width - finalboxPoint2.x - thickness,
                finalboxPoint2.y
            ); // left, bottom

            Right.offsetMax = -new Vector2(
                finalboxPoint2.x, boxPoint1.y
            ); // right, top

            Left.offsetMin = new Vector2(
                boxPoint1.x, finalboxPoint2.y
            ); // left, bottom

            Left.offsetMax = -new Vector2(
                thisCanvas.rect.width - boxPoint1.x - thickness,
                boxPoint1.y
            ); // right, top

            GameObject tempEditer1 = Instantiate(
                objectClassPrefab, tempEditer.transform
            );

            tempEditer1.transform.localPosition = new Vector3(
                boxPoint1.x + 2, thisCanvas.rect.height - boxPoint1.y - 5, 0
            );
            tempEditer1.GetComponent<RectTransform>().sizeDelta
                = new Vector2(boxPoint2.x - boxPoint1.x - 5, 24);

            tempEditer1.GetComponent<TMP_Text>().text
                = list_info.objdetinfo[i].name
                + " " + list_info.objdetinfo[i].conf + "%";
        }
    }
}
```