DE-42 (MTS) ALI, AMNA, ATIF, ZESHAN

VIRTUAL TELEPRESENCE ROBOT

COLLEGE OF
ELECTRICAL AND MECHANICAL ENGINEERING NATIONAL
UNIVERSITY OF SCIENCES AND TECHNOLOGY
RAWALPINDI
2024

COLLEGE OF ELECTRICAL AND MECHANICAL ENGINEERING

DE-42 MTS

PROJECT REPORT

VIRTUAL TELEPRESENCE ROBOT

Submitted to the Department of Mechatronics Engineering
in partial fulfilment of the requirements
for the degree of
Bachelor of Engineering
in
Mechatronics
2024

[illegible]

Sponsoring DS:

Dr. Kunwar Faraz Ahmed Khan (Supervisor)

Dr. Anas Bin Aqeel (Co-Supervisor)

Submitted By:

Ali Arif

Amna Saleem

Muhammad Atif Manzoor

Muhammad Zeshan

DECLARATION

We certify that no portion of the work cited in this project has ever been used to support an application for another university's degree program or credential by signing this form. Depending on the severity of the confirmed offence, we are completely accountable for any disciplinary action that is taken against us if any act of plagiarism is discovered.

ACKNOWLEDGMENTS

In the name of Allah Almighty, the most gracious and the most merciful. First and foremost, we are thankful to Almighty Allah for giving us strength, knowledge, ability, and opportunity to undertake this project Virtual Telepresence Robot and to carry out extensive research and complete the proposed deliverables satisfactorily with collaboration (teamwork).

We gratefully thank Dr. Kunwar Faraz Ahmed Khan (Project Supervisor) and Dr. Anas Bin Aqeel (Project Co-supervisor), whose valuable advice helped us surmount many hurdles. We extend our gratitude to them for their support, cooperation, and time. We take this opportunity to thank our associates for their valuable suggestions and ideas. We would like to extend our deepest gratitude to our family members for adjusting with our unique and unconventional time schedule and providing the necessary support that contributed to the development of the project. Lastly, we are grateful to NUST for giving us a sponsorship fund of PKR/- 0.995 million.

ABSTRACT

The rapid advancement of telepresence technology, fuelled by breakthroughs in robotics, virtual reality (VR), and communication systems, has opened new frontiers in remote interaction. This project presents the development of a virtual telepresence robot, leveraging firmware developed using ROS 2.0 installed on Ubuntu 22.04, an immersive 3D virtual environment generated through camera input, VR, and Unity, and long-distance communication facilitated by a Wide Area Network (WAN) through WebRTC, the system provides users with a seamless remote experience. Moreover, the integration of a Gripper Mechanism atop the robot enables precise pick-and-place operations, particularly valuable in medical applications where precision and control are imperative. This synthesis of cutting-edge technologies forms a robust framework aimed at pushing the boundaries of telepresence technology, enhancing remote interaction, and fostering collaboration across geographical distances.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1 - INTRODUCTION

1.1. Overview:

At the core of this technological project is the integration of advanced robotics, including the TurtleBot 3 Waffle Pi platform, 180-degree camera mounted on top, and the Unity-controlled Meta Quest 2 VR Headset. This combination of hardware and software creates the basic structure for a strong telepresence system that can offer patients and healthcare professionals smooth communication while they are from afar. The incorporation of advanced functionalities, like a Gripper Mechanism on top of the robot, enhances its usefulness in medical settings by permitting pick-and-place tasks that are essential for accurate tasks including medical procedures. The telepresence robot further improves the quality of remote medical consultations by providing a realistic and interactive experience for patients and healthcare providers the same, especially when combined with an immersive 3D virtual environment made possible by the integration of camera input, VR, and Unity.

Transmitting live video streams that are recorded by the camera over a Wide Area Network (WAN) via WebRTC is one of the system's primary features. Wide Area Network (WAN) capability of the telepresence robot guarantees reliable communication between medical professionals and patients over long distances. This is especially important in emergency scenarios where prompt medical attention and guidance can significantly improve patient outcomes. The streamed video feed is rendered for immersive viewing on the receiving end by an application running on an Oculus Quest 2 VR headset. The Oculus controller is used to control the robotic arm's movements, with values mapped to move the arm in the virtual environment accordingly. To add even more interactivity, the trigger button on the controller can be used to open and close the arm's claw. ROS is used to send all arm data, including controller coordinates and claw status, to a specific topic named "controller_data." The processing hub is a Raspberry Pi 4 embedded within the TurtleBot, running Ubuntu 22.04 and ROS 2.0. ROS facilitates seamless communication and coordination of all robot operations.

To create communication over the WAN between the VR environment and ROS topics, we use Unity's ROS Sharp library. ROS nodes on the Ubuntu platform harvest controller coordinates and gather data for additional processing by listening for messages on these subjects. To operate the servos in the robotic arm, the Raspberry Pi interprets this data and exchanges serial communications with an Arduino Uno. Quaternion is the message type used for arm control; it contains three values for controller coordinates (x, y, and z) as well as an extra value for manipulating the claws. To further control the entire robot's movement, we send quaternion data from the Oculus controller to the Raspberry Pi. The linear and angular velocity values in this data are transmitted over the ROS topic "cmd_velocity." The TurtleBot software interprets these twist values and communicates with the OpenCR board to actuate the motors connected to the wheels, enabling seamless robot movement.

1.2. Motivation:

Rapid advancements in telepresence technology herald a paradigm shift in remote collaboration and interaction, presenting currently unseen potential to overcome distance constraints and expand the boundaries of connectivity. Even with these tremendous advances in healthcare, there are still many obstacles to overcome, especially when it comes to access to specialized medical services in remote areas. To improve accessibility to specialized healthcare services and democratize healthcare access for people living in remote areas away from urban centres and medical facilities, this project introduces a novel virtual telepresence robot designed to fill in important gaps in the medical field. Telepresence robot's ability to overcome geographic barriers in healthcare delivery is essential to its mission. Remote medical consultations, diagnostics, and minor procedures can be facilitated by technology, which expands access to specialized healthcare services to underserved and remote areas. Patients no longer must endure long travel times and have better access to professional medical advice, which can lead to prompt interventions and even better health results.

1.3. Background Knowledge:

The literature study offers insightful information about the development of telepresence technology, showing how immersive interactions enabled by developments in robotics, virtual reality (VR), and communication technologies have supplanted traditional video conferencing. Reviewing teleoperation and telepresence systems in particular, it clarifies the opportunities and problems that come with remote control situations, especially when they are in dangerous or unreachable places. Research on force telepresence, interactive teleoperation, and virtual reality integration emphasizes how crucial real-time feedback and user-friendly interfaces are to improving work efficiency and operator experience. The design and execution of a Virtual Telepresence Robot that makes use of WebRTC, ROS 2.0, and VR technologies to enable remote operation and immersive telepresence experiences are informed by these discoveries, which provide the basis of knowledge for the thesis project.

The background knowledge demonstrates how current developments in virtual reality teleoperation systems have the potential to completely transform applications involving remote control. Through the integration of augmented reality (AR) with virtual reality (VR) displays and real-world video feeds, users can command mobile vehicles remotely and engage in extremely realistic interactions with their environment. For the purpose of putting such systems into practice, crucial parts including controllers, cameras, virtual reality headsets, and single-board processors like the Raspberry Pi 4 are mentioned. In order to improve teleoperation efficiency and safety, the review highlights the significance of accurate control mechanisms, real-time video feedback, and user-friendly interfaces. The thesis project benefits greatly from the background knowledge gained from these investigations, which guides the selection of hardware parts, software frameworks, and interaction paradigms to build a reliable and user-friendly Virtual Telepresence Robot.

1.4. Deliverables:

1) Development of firmware for the bot using Robot Operating System (ROS 2).

2) Development of an immersive 3D Virtual Environment using camera input, VR and Unity development plat from.

3) Enabling Long Distance Communication between the bot and the human using WAN.

4) Implementation of a Gripper Mechanism on top of the bot for pick and place operations.

1.5. Contribution:

This project tends to create a platform for development and research on virtual telepresence robot to which more modalities can be added for experimentations in the medical field. This is one of the first experiments in the field of telemedicine in the department of Mechatronics Engineering at the College of Electrical and Mechanical Engineering, NUST.

1.6. Organization of the thesis:

This thesis is further written in the following fashion:

Chapter 2: Provides a Literature Review regarding the existing problems in the perception model and existing methodologies for developing an effective and robust fused model.

Chapter 3: Presents the details of methodology employed by our team in completing this project and explains the inner working of the project.

Chapter 4: Briefly discusses the summary of results and findings and comparison of accuracy with existing methodologies in terms of a more accurate depth perception model.

Chapter 5: Concludes the report and explores future possibilities and directions in which the project can be taken.

Chapter 2 - LITERATURE REVIEW

The burgeoning field of telepresence technology has witnessed remarkable growth in recent years, catalysed by the confluence of advancements in robotics, virtual reality (VR), and communication systems. Telepresence as an evolving concept, extends far beyond conventional video conferencing by enabling individuals to interact seamlessly with remote environments, fostering a sense of physical presence in distant locations.

This review delves into the existing body of literature to explore the multifaceted dimensions of telepresence technology. It surveys a diverse array of research efforts, technological innovations, and applications that have emerged to harness the potential of telepresence in various domains, ranging from telemedicine and remote education to industrial teleoperation and beyond. Through an exhaustive analysis of prior work, this literature review aims to provide a comprehensive understanding of the state-of-the-art in telepresence technology, paving the way for the development of a novel telepresence robot with the integration of cutting-edge components such as a custom-designed 3D printed gimbal, TurtleBot 3 Waffle Pi, Robot Operating System (ROS), and the Meta Quest 2 VR Headset, interfaced with Unity. This synthesis of knowledge will serve as a foundational framework for advancing the field and realizing the full potential of telepresence technology in enhancing remote interaction and collaboration.

2.1. Research on the tele-operation robot system with tele-presence:

With the advancement of technology, there is a pressing need for robots that can function efficiently in hazardous and challenging circumstances. To fulfil this need, interactive teleoperated robots have emerged as a crucial option, considerably improving their capabilities. These robots, which combine human intellect with technological skill, may do tele-manufacturing, teleoperation, tele-design, tele-medical treatment, tele-experimentation, space exploration, and ocean development. It is crucial to highlight, however, that communication time delays are unavoidable in tele-operation systems. To solve this issue, a virtual reality-based force tele-presence system was built, and research on force tele-presence in the presence of time delays and interactive teleoperation technology was done. [1]

2.1.1. The structure of the system:

The tele-operation system of force tele-presence [2] has several distinctive components, including a manipulator, an electro-hydraulic servo drive system, a displacement servo control system, a visual tele-presence system, wireless communication system, force, and displacement sensors. In this system, the master and the slave manipulators are set as master-slave; both of which are equipped with 4-DOF actuators. The slave manipulator has a machine tool for remote controlled systems such as grinding, polishing, assembling, and shaping which makes it appropriate for machining applications.

In such a tele-operated master-slave system, the master plays dual functions as a reference input to command the slave and as a force feedback source to inform the operator on the state of slave. This ‘sense of force’ enables the operator to touch and manipulate forces originating from the remote location.

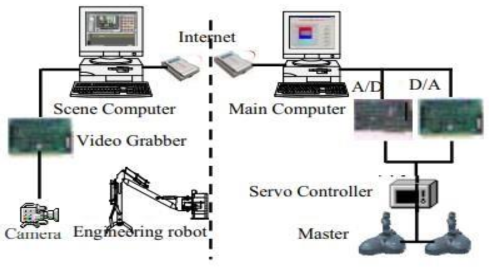


Figure 1. Master Slave System for remote control

The remote-control computer connects to the site computer during the initial control stage. The site computer continually reads and exchanges joint displacement and velocity data with the remote-control computer through A/D conversion. This information is used to setup the graphical depiction of the robot, allowing the operator to monitor the state of the site robot. Event-driven control instructions are transferred between the site and graphic computers. The visual computer updates the virtual robot's motion state in real time, while the site computer converts operator commands into motion angles for each joint. This entire procedure runs with a 10-millisecond sample and control period.

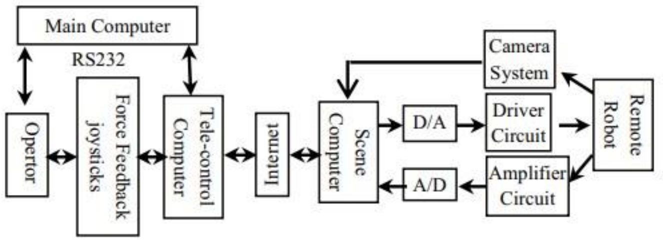


Figure 2. Remote Robot Control System Principle

In tele-operating the remote worksite robot while facing the simulation robot, real-time video information is crucial due to model errors between the graphic robot and the virtual environment. This video information is obtained through equipment mounted on the remote robot, including cameras, video emitters, and video receivers.

Compared to tele-operation solely relying on video feed from the site, the approach which offers high tele-presence and real-time feedback, can enhance work efficiency by 30% to 50%. It mitigates the impact of time delays and provides a user-friendly graphical interface, allowing the operator to adjust the video perspective in the virtual environment.

This system's remote mechanical arm is a 4-degree-of-freedom (4-DOF) system [3] that was produced by replacing the original hydraulic excavation machine shovel [2] with a new single-degree arm. For claw manipulation, gears are used. Using hydraulic cylinders, proportional valve control is used to achieve four degrees of freedom. One of these cylinders is employed by gearbox machinery to spin the main body. To detect displacements, linear displacement sensors are used, and the hydraulic pressure is sensed by the hydraulic sensors.

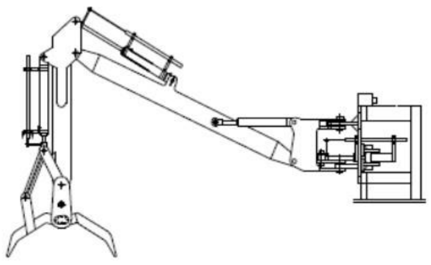


Figure 3. 4 DOF Engineering Robot Structure Design

2.1.2. The design of the force bilateral hydraulic servo control system:

In the improved parallel control method, the difference between the operating force of the manipulator and the working resistance of the slave mechanism is used as the control signal to control the motion of the master. This approach utilizes the advantages of force error and parallel bilateral servo control. The slave driving force is related to the force and displacement difference between the master and the slave, and the feedback force is relative to the force error between the master and the slave. If the slave comes in contact with an object with a lot of stiffness, then the impact force is too large, and both the master slave position following qualities are poor which is a problem in the current control mechanism.

2.1.3. Tele-operation experiment result analysis:

The master, a 2-degree-of-freedom (2-DOF) hydraulic driving manipulator, and the slave, a 4-degree-of-freedom (4-DOF) hydraulic driving construction robot. The control principle are shown to be similar to a single degree of freedom force bilateral hydraulic servo control system which is primarily used for force information exchange during grasping.

Studies show that improved parallel force feedback bilateral servo control system is acquired through platform-based tests. The tests are conducted under two circumstances: no load and spring load. In the no-load condition, the focus is on evaluating the ability of the slave to follow the movement of the master and monitoring the performance of the PD force controller with a dead zone by observing changes in the manipulating force of the master.

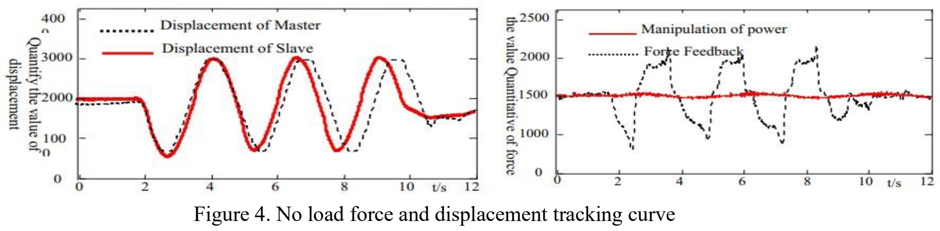


Figure 4. No load force and displacement tracking curve

The purpose of loading experiments is to give the operator force feedback during teleoperation, assuring accurate operation by letting the operator feel the force between the slave and the environment. [4]

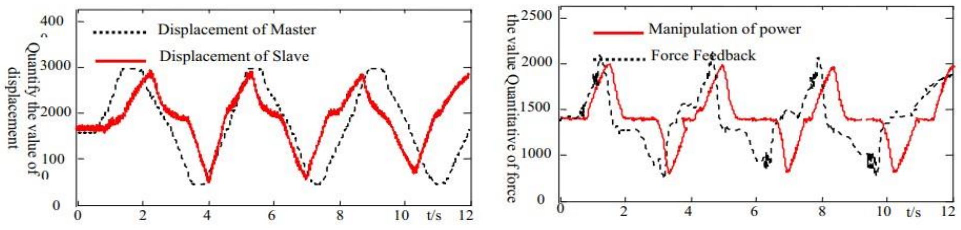


Figure 5. Spring load force and displacement tracking curve

2.2. Virtual reality teleoperation robot:

The system's substantial goal is to apply augmented reality to demonstrate how a mobile vehicle operates at a different physical location from the user while yet providing the user a sense of presence. This augmented reality (AR) experience interacts real-world camera input with virtual reality (VR) display. Users with headsets watch the cars live video feed and can navigate the vehicle using a remote control based on the camera input. The system strives to deliver a pleasurable and user-friendly experience. The two key connections that make up the project's basic functionality are controller to automobile and camera to VR. These connections allow users to explore their surroundings by steering the automobile with an external joystick and viewing the world through the camera feed of the vehicle. The use of a movable automobile and camera system allows it to stand out altogether. [5]

2.2.1. Background of VR with tele-operation systems:

Combining VR with teleoperation systems has shown positive results, including improved spatial awareness and reduced operator stress. A Gifu University project integrating VR with a construction robot found that VR enhanced task efficiency and engagement compared to traditional displays. VR-based operation was deemed safer and less stressful, highlighting the benefits of VR in teleoperation. [6]

Brown University developed a software project that combines VR with hardware systems, enabling users to control robots using everyday VR equipment. In this project, they connected a research Baxter robot with an HTC Vive headset. Participants were given the task of stacking 16 differently sized cups, and the results were impressive. The VR interface was not only more enjoyable, and fun compared to a keyboard interface, but users also stacked the cups faster. [7]

Two major software programs exist for VR development: Companies like Unreal and Unity. Unreal is a tool for developing virtual reality games and a poor fit for hardware interfaces. Unity has more options for real world hardware interfacing, particularly for embedding the camera views which are required by the system.

Five primary pieces of hardware equipment are used: the controller, camera, VR headset, the motorized vehicle to be controlled, and a single-board computer that controls the vehicle. The code for conveying instructions from the controller to the car is installed in a computer hardware integrated in the car. The car control system is done with the use of Raspberry Pi 4 and the use of Python scripts. This setup is chosen because it is well documented and can be used freely. [8][9]

2.2.2. Implementation overview:

The user must launch the application while wearing the headset to view the footage from the vehicle's cameras. The robot was guided by a remote control to its destination and moved in response to input from the camera. Two main connections can be made across the entire system. VR camera, vehicle controller. The system's objective was to give people the ability to experience the environment like they were there.

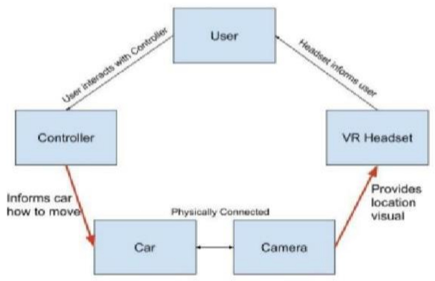


Figure 6. A general overview of Car VR system and connections

The primary controller for the Car VR system is the Raspberry Pi 4, which is installed inside the vehicle and controls most of the connection logic. It acquires the camera feed and sends it over the internet so that the virtual reality system can comprehend it. The Raspberry Pi 4 also decodes Bluetooth signals from the controller to drive the motors of the vehicle. Most debugging efforts were focused on this board because of its crucial role in system connection. Figure 7 shows a more thorough representation of the Car VR system with important data flows highlighted: orange arrows show data flow from the user to the vehicle's motors, while green arrows indicate data delivering video feedback to the user. [10]

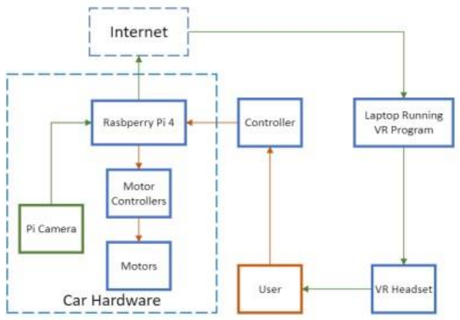


Figure 7. A specific overview of Car VR system and connections

2.2.3. Camera to VR system:

The resources used included an Oculus Quest 2 headset, a Raspberry Pi 4, a PS4 controller, a Raspberry Pi camera, VESC motor controllers, brushless motors, and various smaller electronic and 3D printed components.

The camera to VR system informs the user how to control the car. To enable live video streaming from the Car VR system, a Raspberry Pi with a compatible camera and an opensource Python camera script from "**raspberrypi.org**" were used. The Raspberry Pi runs on Raspbian OS and is configured to create a server with its IP address, transmitting bytes to a specific URL. This setup generates a live, low-latency MJPEG stream accessible through a browser using the Pi's IP address. The MJPEG processing script handles HTTP requests and responses, processing incoming bytes from the HTTP buffer. It interfaces with a texture script in Unity to display the video in the VR environment. The project incorporates Oculus plugins in Unity, creating a scene with a viewpoint and an empty room, offering limited virtual mobility within the space.

On an Oculus Quest 2 headset, the Unity program was used to evaluate the system. To do this, an Oculus Link cable must be connected between the laptop and the Oculus Quest 2 headset. The camera feed was shown in the VR experience when the Unity project had finished running. Before continuing with development, it is essential to do these integration tests. Once successfully finished, the fundamental functionality of the VR environment was developed, enabling users to watch the camera feed and immerse themselves. This signalled that the camera and VR system were prepared to include new parts.

2.2.4. Control system:

A control system managed the vehicle's direction and speed. A PS4 controller, a Raspberry Pi, an electronic speed controller (ESC), motors, and a few signal converters make up the system. On a Raspberry Pi, a Python script served as the vehicle's control program. To establish the output signal for the motor controller and control the system's speed, code develops a connection shell via Bluetooth which evaluates input of controller. The API was utilized in the lightweight module PS4 Controller, which reads an input stream and segments it as events take place. The output of the Python script creates a link between the controllers of PS4 input variables and the motors speed control output signals. PWM (Pulse Width Modulation) signal was the output signal. The PWM signal can define 35 different speeds and up to 35 different duty cycles. The output duty cycle increases as the joystick is pushed higher, causing the speed to increase.

2.2.5. Vehicle specifications:

The car is built like a tricycle, with a third steering wheel at the back for stability and two motors for two-wheel drive at the front side. Each of his motors had an individual ESC, allowing him to run each wheel at a distinct speed. The joystick's Y axis was used by the Bluetooth controller to input the throttle. The left joystick's position determined the speed of tire on left side, and the position of right joystick determined the speed of the right wheel.

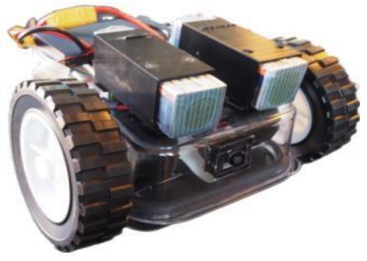


Figure 8. Vehicle Exterior

Due to concerns with motor demagnetization when a steel chassis was first attempted, the chassis is mostly made of plastic components. Although aluminium was a possibility, the plastic chassis was chosen because of financial restrictions. Utilising 3D printed mounts, all system components are affixed to the chassis.

The main operating voltage of the power system is 22.2 volts, and it is powered by two paralleled 6S LiPo batteries with a total capacity of 6600 mAh. These batteries have a 50C rating, guaranteeing they can provide enough current for the system's 50-amp motors. These 190 kV-rated motors are frequently used in electric longboards. Their usage permits freedom in vehicle weight if the chassis weighs less than 150 lbs. These motors need speed control, though. Using VESC software, the Electronic Speed Controllers (ESCs) are designed to limit the motors' maximum rotational speed to 20000 rpm and their maximum current consumption to 35-amps apiece to establish realistic control. To provide fine control at slower speeds, a throttle curve is also used.

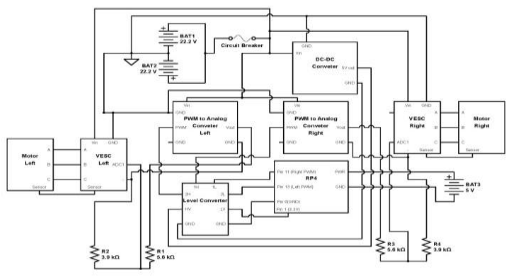


Figure 9. System wiring schematic

Other parts of the automobile include DC-to-DC converter, cooling fan, 5volt battery, circuit breaker, and Raspberry Pi camera. The components such as the fan and the level-shifter are powered at 5 volts by the DC-to-DC converter using the 22. to be powered by 2-volt batteries. The cooling fan is needed to avoid thermal shut down of the raspberry pi. The 5-volt battery is used to power the Raspberry Pi apart from the 22. The circuit is a 2-volt system, which allows Pi usage without engaging the ESCs. The main power switch is an 80-amp circuit breaker for the 22. 2-volt system and also protects the power system wiring harness from overheating. Finally, the camera is connected to the Pi to make the image for the VR feed.

2.2.6. Future applications:

The project explores the potential of VR in teleoperation and suggests transitioning from Bluetooth to WIFI control for global accessibility. This expansion opens diverse applications, especially in large-scale surveys. Augmented Reality features could further enhance the system, making it adaptable to various industrial scenarios, such as geological surveys and object manipulation.

2.3. Construction tele-robot system with virtual reality:

This research focuses on the development of a tele-robotics system for a construction machine, including a construction robot with seven servos, two joysticks for manipulation, and 3D virtual environment for interaction. For visualization of the robot and task objects, computer graphics (CG) are employed in the simulation environment. Operators can control the construction robot through the graphic robot in the virtual environment based on signals transmitted from joysticks. The positions and shapes of the task objects in the virtual world are made synchronous in real-time using a stereo camera located in the remote field. To resolve disadvantages of the three-screen display and improve teleoperation systems' effectiveness and reliability, the paper reveals the newly developed auto point of view (APV) and semi-transparent object (STO) approaches. Evaluations of a one-screen visual display, which incorporates APV and STO, in comparison to the conventional three-screen display proved helpful in terms of increasing work efficiency, safety and reducing stress levels. [11]

2.3.1. Tele-robot system with virtual reality:

The tele-robotic system is divided into two parts: the master system and the slave system. The slave system is represented by a construction robot with a stereo camera. In contrast, the master system is operated by a human operator and primarily comprises a manipulator and a screen. [12]

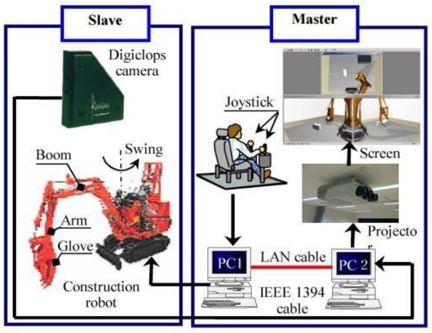


Figure 10. Construction tele-robot system with virtual reality

As mentioned earlier the construction robot has four hydraulic actuators which are controlled by four servo valves connected to a computer (PC 1) in the remote field. [13] Accelerometers are attached to the robot to provide the operator with information about its movements and acceleration. The operator has manipulator in form of two force-feedback joysticks that can translate in the XY plane. For the swing, boom, arm, and glove movements of the construction robot, the four angular displacements that correspond to these joysticks are transformed from linear displacements of the hydraulic cylinders.

It uses the function of a stereo vision camera. This camera produces range pictures in real time colour stereo vision which is effective in determining distances to target objects in its field of view at up to 30 frames per second. The operator of the presentation system developed for this study can rotate the views of the CG images of the distant robot and the task item in front of him/her. [14] These CG images are generated by a graphics computer known as PC2 with the help of input signals coming from the joysticks and the stereo vision camera known as “Digiclops. “

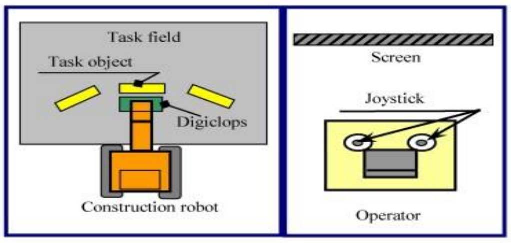


Figure 11. Arrangement of the system

A joystick and a screen are used by the operator to remotely operate the construction robot. A computer (PC1) receives the operating data from the joystick and uses it to generate commands for the robot's servo valves. These signals are subsequently sent to the construction robot's servo amplifier.

The construction robot and other task objects are visualised in the virtual world by another computer (PC2) that displays it. These graphics are created by combining operational data from the joystick with stereo camera data on the location and shape of the job item. After that, these video signals are sent to a projector, which then projects the images onto a screen.

When computer generated (CG) pictures are produced from stereo photographs, the robot's CG representation is built using displacements discovered by sensors mounted to the hydraulic cylinders. In contrast, the "Digiclops" stereo camera is used to create the computer-generated pictures of the job items.

2.3.2. Visual display in virtual space:

To improve operability and safety, more visual signals were offered in the virtual environment throughout this study. To provide a realistic feeling of vertical distance, shadows were applied to the construction robot and task objects, using darker hues when objects were closer to the ground and lighter tones when objects were farther away. Three displays (top, left, and right) were commonly featured in the operator's conventional visual display. The operator having to choose which screen to concentrate on, which may take time and result in mistakes, especially for novice operators, was a disadvantage of this setup. Additionally, choosing the incorrect screen might result in the construction robot being positioned in a dangerous way.

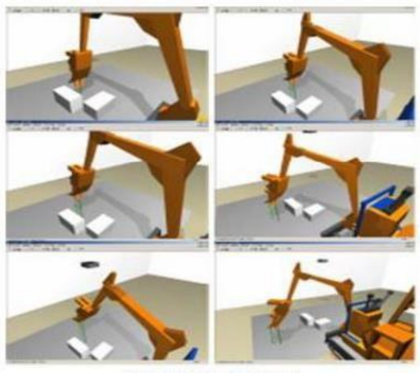


Figure 12. The auto point of view

In this study, a feature known as "auto point of view (APV)" was proposed to overcome the problems previously discussed. When employing a single-screen display, APV automatically changes the point of view and reference point based on the swing and boom behaviour, making operations simpler. It's crucial to remember that this strategy presumes a level distant location and that no blocks may be stacked higher than three feet. Parameters control viewpoint movement for improved spatial recognition based on the construction robot's behaviour enhancing operator speed and safety. A semi-transparent object (STO) function and the ability to remove the robot's body from the CG were added to address visibility issues and reduce operator stress.

2.3.3. Experiments and discussions:

During the experiment, the construction robot was manned and operated by the users via a joystick to perform specific tasks. The virtual space display system has not attained the level of accuracy required in emulating various situations hence still under development. Therefore, it is rather natural that when the current virtual space display system has been used as the visual display system, then the task contents must be simplified to a certain extent.

Thus, in the present study, three types of tasks (Task 1 – 3) were employed, namely, stacking, and transporting two or three concrete blocks (approximately 200mm x 260mm x 100mm in dimension; referred to henceforth as "blocks") within the operating envelope of the construction robots.

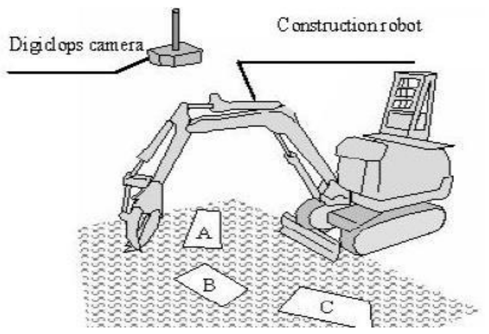


Figure 13. Task area for evaluation of the system

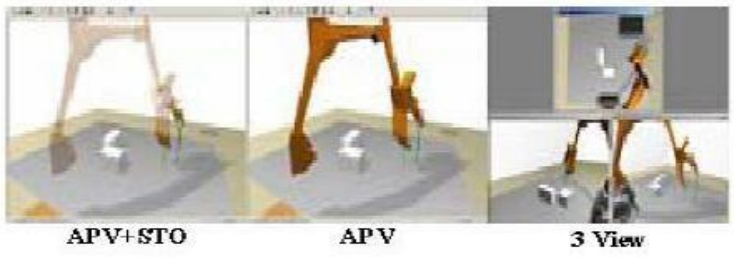


Figure 14. Experimental Condition

1. **Task 1:**
   In the initial condition of Task 1, two blocks are placed at points B and C. First, the block at point C is transferred to point A; then the block at point B is placed on top of the block at point A.

2. **Task 2:**
   The block at point A is put above the block at point B in the beginning of the experiment. The block is then transferred from point C to point A. The upper block of the two blocks piled at point B is finally transferred to point C.

3. **Task 3:**
   In Task 3, blocks are placed at points B and C at the start of the task. The block at point C is then moved to point A and the block at point B is moved to point C. The block at point A is then moved to point B. This means that the blocks are moved to the empty circle in the middle of points A, B and C a total of eight times.

Three different visual conditions were employed in the experiment. These were the “APV+STO” condition, which was to complete the task on a single screen with moving APV and semi-transparent polygonal object; “APV,” which was to complete the task on a single screen using APV and an opaque polygon; and “3 view,” which was to complete the task on the three screens as per the standard.

The standard deviation along with the average has been calculated with reference to the number of block movements carried out by the subjects in a minute. The abscissa refers to the visual conditions in the experiment, while the ordinate represents task efficiency in terms of objects completed per minute. Increased values on the ordinate signify higher levels of task efficiency.

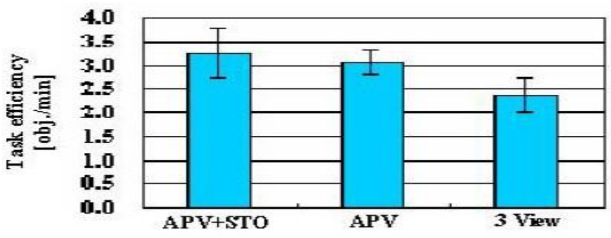


Figure 15. Task Efficiency

The results of the time and pressure of contact between the floor and the construction robot, averaged are indicated in the figure below. The abscissa displays the visual circumstances just like above. The right ordinate displays the dimensionless force F_c, which is calculated from the average force produced by the boom, arm, and swing. The left ordinate displays the time of contact with the floor, t_c in seconds. Larger values on the right and left axes denote risk.

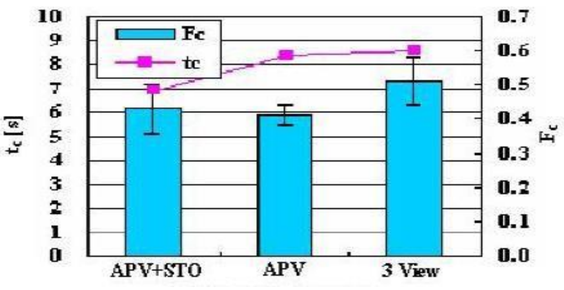


Figure 16. Risk Management

2.4. Haptic interaction in tele-operation control system of construction robot based on virtual reality:

This paper introduces a tele-operation control system for construction robots (TCSCR) using a master-slave control scheme. The TCSCR comprises a construction robot controlled by a servo valve, two joysticks for remote operation, and a 3D virtual working environment based on virtual reality technology. By utilizing virtual reality, this technology enhances task efficiency by immersing the operator in a real-time virtual environment. However, for inexperienced individuals, operating a remote construction robot (RCR) can be dangerous, leading to accidents like toppling. This happens when the RCR continues working despite being unstable or encountering obstacles. To improve safety and prevent such accidents, haptic interaction is integrated into the TCSCR using virtual reality technology. When a haptic interaction indicating danger occurs, the system calculates a reaction force based on predefined rules and conveys it to the operator through the joystick.

Technology developments in virtual reality (VR), computer graphics (CG), and visualization have made it possible to simulate the real world inside of a computer environment. By giving operators, a feeling of active participation at the operation site, VR technology accomplishes this by using a three-eye stereo camera to record data from the scene and produce real-time computer graphics (CG) representations of this data. The addition of VR improves traditional teleoperation control systems for construction robots' (TCSCR) overall task efficiency while also improving teleoperation's safety and comfort. To lower the risk of toppling accidents, the research aims to prevent tele-operation construction robots from getting too close to the ground and obstacles. This is accomplished by improving the haptic interaction between the virtual environment and the glove of the graphics construction robot. By relaying haptic feedback along with proximity information, this improved interaction gives the operator a more realistic experience and improves safety. [15]

2.4.1. Tele-operation construction robot based on virtual reality technology:

2.4.1.1. Composing and principle of TCSCR based on virtual reality:

The master system and the slave system are the two components of the bilateral servo control type of TCSCR used in this study. [16]

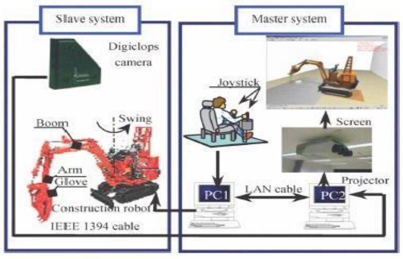


Figure 17. Schematic Diagram

In this setup, a construction robot with a stereo camera serves as the slave system. Four hydraulic cylinders and four servo valves on the robot are managed by a computer (PC1). Magnetic stroke sensors embedded in the pistons track the displacements of the cylinders, and a pair of pressure sensors fastened to the cylinders track the external forces acting on them.

A computer (PC2) and the stereo camera, called "Digiclops," are linked together by an IEEE 1394 interface. With its optical axis parallel to the ground, it is placed directly above the construction robot. Real-time range images are provided by the "Digiclops" colour-stereo-vision system, which uses stereo vision technology. This technology creates computer-generated (CG) images of the task objects at a rate of about 30 frames per second through PC2 and assists in gathering information about the position, colour, and shape of target objects in its field of view.

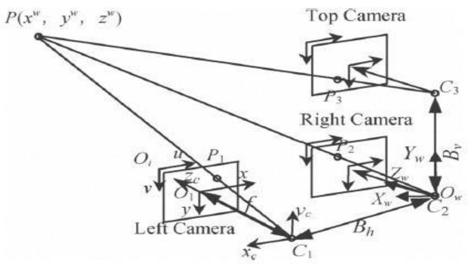


Figure 18. Camera Geometry Model

Two force-feedback joysticks, a screen, and a human operator control the main system. Each joystick can be moved along the X and Y axes, and the linear movement of the hydraulic cylinders in the RCR arms is represented by the angular displacement used to measure each joystick's displacement. Under each joystick is a DC motor that allows feedback of reaction forces produced by the RCR's arm cylinders during operation. This feedback is made possible by a speed change gear connected to the DC motor. Additionally, CG images created by PC2 using a projector are shown on a screen set up in front of the operator.

The TCSCR works as follows: During work, the operator uses joysticks to control the RCR while keeping an eye on the screen in front of them. To calculate the signals for the RCR's servo valves, PC1 receives operational input from the joysticks. The RCR's servo amplifiers receive these calculated signals next. The "Digiclops" camera's images of the target object and the working environment are sent to PC2 at the same time as the RCR's arm cylinders' displacements. The construction robot and the target object's CG images must be displayed in a real-time virtual environment, which is the responsibility of PC2. A projector is used to project these video signals onto the screen in front of the operator, enabling the operator to view CG images of the RCR and the task object from various angles.

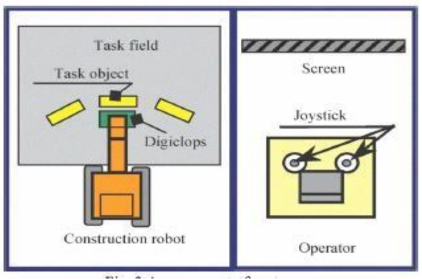


Figure 19. Arrangement of the system

2.4.1.2. CG images based on virtual reality technology:

The magnetic stroke sensors on the hydraulic cylinders controlled by the joysticks detect the displacements of the arm cylinders, which are used to generate the CG image of the graphics construction robot (GCR). However, it is difficult to produce CG images of the operation site from "Digiclops" images. As a result, due to computational constraints, this research simplifies the operation site by assuming it lacks inclined planes and only considering white, regular concrete blocks as the task objects.

The stereo vision processing algorithm in "Digiclops" is used, which is appropriate for this application, to generate CG images of the target objects from "Digiclops" images. The process entails creating a world coordinate system that is in line with the coordinate system of the right camera in "Digiclops." Using colour data, the robot arm image is eliminated. Setting a threshold, creating binary images, and labelling regions are all examples of general image processing that is used. Less than 10 cm x 10 cm are discarded as small areas. It is possible to measure distances and learn about the position, colour, and shape of the target object by extracting the outline of the objects and using it to calculate the convex hull. For real-time rendering and object reconstruction, this data is sent to the OpenGL context.

2.4.1.3. Velocity control method with variable gain to force feedback model:

In the case of the TCSCR with a master-slave control structure, it is crucial to make the operator receive necessary force feedback from the work field. Position control of either the master or slave side has traditionally been the key point of TCSCR control approaches. Force reflection, force-position compound, position symmetry, variable gain position symmetry and improved variable gain position were all studied as force feedback models. Although these techniques have enhanced the force feedback of the joystick in terms of force feedback, each technique has its own advantages and disadvantages.

A position-velocity control method is presented to enable velocity control of the arm cylinder in the RCR. The piston velocity of the RCR's arm cylinder is zero when the joystick is in its centre (displacement equals 0). The arm cylinder's piston velocity rises along with the displacement of the joystick as it moves more freely. However, there are issues with this method when handling concrete blocks because the piston velocity is slow. To address this issue, a variable gain factor "T" is incorporated into the force feedback model. [17][18]

The value of "T" adjusts based on the changing driving force threshold according to the piston's different movement directions. Simultaneously, the model assumes the neglect of inertia force and piston friction, and it calculates the reaction torque "τ" applied to the joystick.

$$\boldsymbol{\tau r = T[kpm\ (Ym - Vs) + ktmfs]}$$

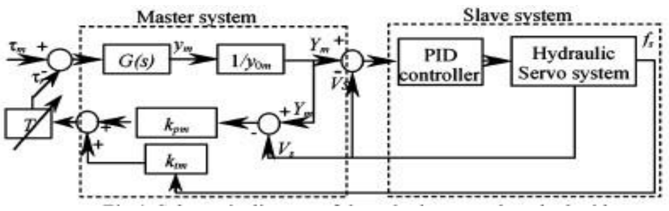


Figure 20. Schematic diagram of the velocity control method with variable gain to the force feedback model

2.4.2. Haptic interaction in the tele-operation construction robot control system:

2.4.2.1. Introduction to haptic interaction:

Haptic interaction aims to enhance the operator's sensory experience and improve RCR safety. It involves interpreting proximity as tactile feedback and conveying potential dangers through force feedback via the joystick.

2.4.2.2. Haptic interaction detection and calculation of reaction force:

Most haptic interactions with the GCR glove happen during RCR motions. "Imaginary springs" are virtually positioned on the front of the glove to prevent collisions and measure distance to the ground or obstacles. Within a predetermined proximity range known as the time headway (THW), these springs produce a reaction force that grows as the GCR approaches the ground or obstacles. This idea serves as the foundation for the calculation of the reaction force on the joystick.

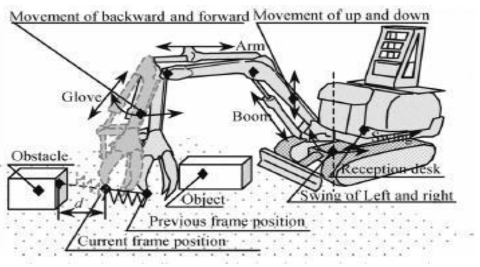
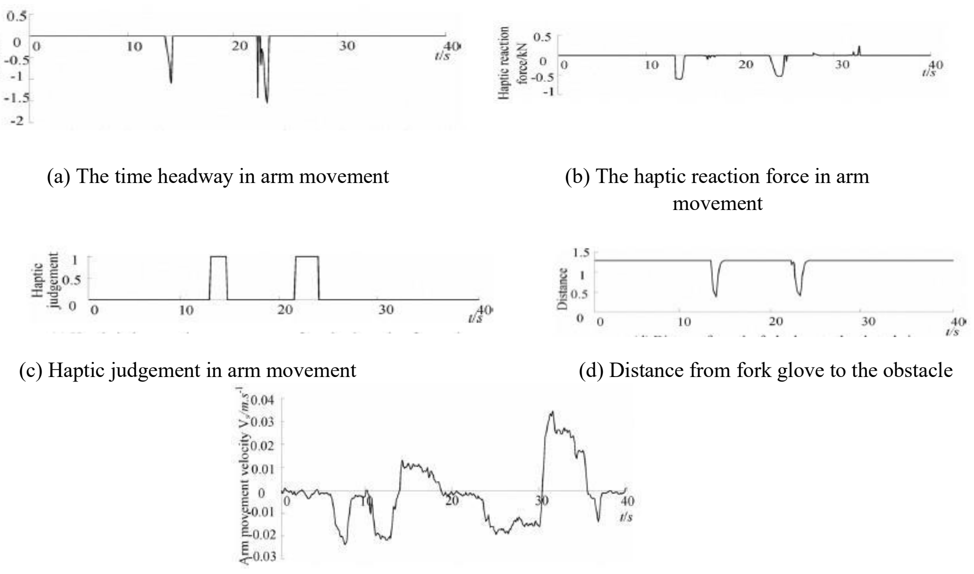


Figure 21. Schematic diagram of the haptic detection between the fork glove of GCR and the ground or obstacle

2.4.2.3. Experiments:

To test the control system, experiments were done on a testbed for tele-operated construction robots. Joysticks were used by operators to complete tasks. The experiments were aimed at haptic interaction detection, reaction force calculation, and TCSCR performance evaluation. This study focused on the haptic interactions that occur when an obstacle is encountered during arm movements around the arm shaft.



(a) The time headway in arm movement

(b) The haptic reaction force in arm movement

(c) Haptic judgement in arm movement

(d) Distance from fork glove to the obstacle

(e) The velocity of the RCR arm in movement around the arm shaft

Figure 22. The haptic detection and the calculating of the reaction force in the movement from backward to forward around the arm shaft

2.5. ROS Reality: A virtual reality framework using consumer-grade hardware for ROS-enabled robots:

Robotic teleoperation tasks now allow for natural 3D interaction using virtual reality (VR) systems. Early VR systems were expensive and needed specialized hardware, but recently affordable consumer-grade VR systems have become available, making them more widely available. The goal of the study is to create ROS Reality, an open-source teleoperation package for the Robot Operating System (ROS) that works well with Unity-compatible VR headsets. Using ROS Reality, skilled human users guided a Baxter robot through 24 dexterous manipulation tasks. This study highlights problems that need to be fixed in these VR systems and sheds light on how feasible VR teleoperation tasks are using today's consumer-grade resources.

Virtual reality (VR) offers intuitive point-and-transformation specification and seamless interactions with the real world, making it a highly effective interface for robots. VR interfaces can be used for a variety of tasks, such as remote debugging and troubleshooting, robot teaching, learning from demonstration, and teleoperation. By mapping robot manipulators to VR hand controllers, VR systems make it possible for non-experts to control robots in an intuitive manner that feels like an extension of the user's hands. With the help of this simple interface, regular people can operate complex robot functions without extensive training. VR interfaces use expert users to train robots for precision-required tasks, combining the skills of novices and experts in difficult fields. ROS Reality is a VR and Augmented Reality (AR) teleoperation interface that works with ROS-enabled robots and consumer-grade VR and AR hardware. By using consumer-grade VR and AR hardware, it enables users to view and control robots remotely over the Internet.

2.5.1. Related Work:

Robots can take on tasks that would be difficult to complete autonomously through teleoperation. Humans can work in hazardous conditions securely from a distance by using it additionally. [20]

Although 2D interfaces for robot teleoperation, especially over the Internet, have become more popular and offer control through monitors and keyboards, they are out of sync with how people naturally perceive and interact with the 3D world. [21][22]. According to research, VR interfaces can help with this problem. Non-expert users of teleoperating robots were discovered to be quicker, more productive, and more comfortable using a VR interface than a 2D monitor interface. [23]

This implies that VR interfaces improve the experience of teleoperation. Virtual reality interfaces and gantry systems offer direct, intuitive control of robots. For example, the da Vinci Robot System enhances surgical performance, but it's task-specific and stationary. [24] Mallwitz et al. created a portable exoskeleton for teleoperating humanoid robots, but it's costly and limited to specific robots, unlike web-based interfaces. [25]

The gaming community now has access to VR systems because of recent advancements in graphics. Affordable and transportable VR hardware is available through items like the HTC Vive, Oculus Rift, and Google Cardboard. As an outcome, lab researchers have begun to investigate these VR teleoperation systems for robots. Due to the relatively recent widespread availability of consumer-grade VR systems, there has been little research on the efficacy of teleoperation interfaces utilizing this technology. While the type of interface and the robot being used have a big impact on how well a task is completed, the interest was in seeing what kinds of complex tasks an open-source software and a typical research robot could handle.

2.5.2. ROS reality:

2.5.2.1. VR as a teleoperation interface:

In the context of human-robot interaction, there are two primary categories for displaying the robot's state to the user and mapping the user's input: egocentric and robocentric models.

1. Egocentric Models:
In egocentric models, the human is positioned at the centre of the virtual world, effectively coexisting in the same space as the robot. Notable examples of this egocentric approach include the work by Lipton et al. [26] and the research by Zhang et al. [27] In these scenarios, human users have reported a sensation of "becoming the robot" or "seeing through the robot's eyes." Essentially, the user's perspective aligns closely with that of the robot, fostering a strong sense of immersion and embodiment.

2. Robocentric Models:
Robocentric models involve the human and the robot sharing a virtual space but without necessarily superimposing one another. A specific example of this model is the one employed in evaluating ROS Reality. Here, the human interacts with a virtual representation of the robot by manoeuvring around it in a virtual environment. Control over the robot's arms is achieved by virtually grabbing and dragging them, resembling the operation of a virtual gantry system. In essence, the user and the robot exist within the same digital realm, yet their positions and perspectives remain distinct.

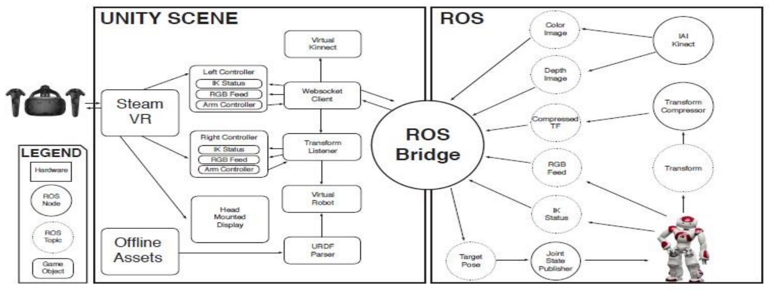


Figure 23. A diagram detailing the architecture of the ROS reality system

2.5.2.2. System overview:

The ROS Reality system uses an HTC Vive connected to a computer running Unity. Unity creates a local robot model from its URDF using a custom parser. The system connects to a ROS network via Ros bridge WebSocket, transmitting robot's pose and camera data. [28] A Kinect 2 provides colour and depth images, which are processed into a point cloud in Unity. User interaction is enabled by holding a deadman's switch, with the user's controller poses sent to the robot. The robot's end effectors are adjusted using inverse kinematics. This setup allows immersive remote control of the robot, enhancing user experience.

2.5.2.3. ROS:

ROS (Robot Operating System) is a toolkit for programming robot applications. It facilitates communication between various software components called nodes, each performing different tasks. These nodes exchange data over channels, known as topics, using a local TCP network, forming a ROS network. Nodes can either publish data on a topic using publisher objects or subscribe to a topic using subscriber objects. ROS offers programming interfaces in C++ and Python, and in the case of ROS Reality, all nodes are developed in Python.

In the context of ROS Reality, several components are launched. These include a Kinect2 ROS node, two RGB camera feeds (one for each wrist camera of the robot), a Ros bridge WebSocket server, a custom ROS node responsible for converting the full transform (TF) of the robot into a compact string, and another ROS node that listens for target poses from VR systems. This latter node queries the robot's Inverse Kinematic (IK) solver and instructs the robot to move to the IK solution if one is found. If an IK solution cannot be determined, it reports an IK failure.

ROS Reality integrates these components to enable seamless communication and control between the virtual reality interface and the robot, enhancing the user's ability to teleoperate and interact with the robot using VR technology.

2.5.2.4. HTC Vive:

The HTC Vive is a consumer-grade virtual reality system comprising a head mounted display (HMD) and two wand controllers. Each device is tracked with high precision using infrared pulse laser emitters (lighthouses), allowing for accurate position and rotation tracking with minimal error (1-2mm). The system is wireless for the wand controllers, and the HMD connects to a computer via USB and HDMI cables. The controllers are equipped with a touchpad, trigger, and two buttons for user input. It is compatible with various game and physics engines, but it is most extensively supported and initially developed for Unity. To integrate the Vive into Unity, a software package called Steam VR is used, enabling seamless interaction and development for virtual reality applications.

2.5.2.5. Unity:

Unity is a versatile game engine commonly used for creating popular 2D, 3D, and Virtual/Augmented/Mixed Reality applications. It boasts a built-in physics engine capable of handling contact dynamics and simulating materials. Unity is compatible with a wide range of VR and AR hardware and provides a shader language for creating custom GPU shaders.

In Unity, an open environment is referred to as a scene, which contains the fundamental building blocks known as GameObjects. Each GameObject is associated with a set of components. While there are numerous types of components available, the script component is especially important for applications like ROS Reality. A script is a small program typically written in C# and executed during each rendering frame. ROS Reality's functionality is realized through a collection of these Unity scripts, which enable interaction and control within the Unity environment.

2.5.2.6. ROS Reality:

ROS Reality is a collection of C# scripts that enable users to remotely view and control a robot equipped with ROS (Robot Operating System) capabilities over the internet using a virtual reality (VR) interface.

1. WebSocket Client:

The script is a C# implementation of the default Ros bridge client, known as ROSlibjs. It facilitates essential functions such as advertising, subscribing, and publishing to ROS topics. Communication between the script and the ROS system involves sending and receiving messages in a JSON format, with data encoded in base64, following the Ros bridge specification. This implementation allows for seamless interaction and data exchange between the C# script and the ROS-enabled robot.

2. URDF Parser:

The script parses URDF files to create a Unity hierarchy of GameObjects representing a robot. These GameObjects simulate the robot's physical properties using Unity's physics engine, allowing interactions in virtual scenarios, and supporting teleoperation practice. The URDF parser has been successfully tested with PR2 and Baxter robots.

3. Transform Listener:

By subscribing to the robot's TF (Transform), the Transform Listener synchronizes the virtual robot's pose with the physical robots. To accomplish this, it reads each link's position and rotation data (in quaternion representation) from the ROS TF topic and applies these transformations to the corresponding links of the robot simulation. This guarantees that the movements and positions of the real robot are precisely mirrored by the virtual robot.

4. RGB Camera Visualizer:

The robot's camera feeds are visualized by means of this script. It converts the camera image from base64 encoding after subscribing to a specified camera topic. Next, it uses the camera feed to texture a plane GameObject. The purpose of attaching this plane GameObject to the user's wand controller is to allow the user to continuously view it while they interact. The script supports JPG and PNG image formats, but because of bandwidth constraints, ROS Reality always uses JPG.

5. Kinect PointCloud Visualizer:

The Kinect PointCloud Visualizer script creates a point cloud from RGB and depth data from a Kinect 2 by using a GPU shader. The script generates data and ads for every pixel pair, computes their 3D positions using depth data, and converts them to world space before rendering in the Unity scene. It does this by subscribing to the RGB and depth topics of the Kinect. The Kinect data is processed to create a 3D point cloud.

6. Arm Controller:

The user can provide the robot with the target end effector coordinates by using this script. The current position and orientation of the controller are converted from the Unity coordinate frame to the ROS coordinate frame when a deadman's switch, the side grip buttons on an HTC Vive is depressed. This information is then published over a topic to a node in the ROS network, which queries the robot's integrated IK solver and moves the robot if a solution is found. This script also enables the user to open and close the gripper using the wand controller's trigger. Another way to do this is by messaging the robot about a subject.

7. IK Status Visualizer:

If the IK solver fails, this script subscribes to the robot's IK solver's status and turns the user's wand controller red. This informs the user if the robot is unable to reach the target position they sent it.

8. Robot:

Baxter's features include two arms with seven degrees of freedom each, grippers with force sensing built in, a fixed base, and a head with a display screen. This allows Baxter to handle a variety of objects with skill. Furthermore, adding rubber grips from the Baxter toolkit to improve grip by increasing friction at the end effector. Moreover, linking ROS Reality to a PR2 robot simulation in Gazebo, enabling to watch the robot's actions in real time.

2.5.3. Long distance teleoperation trial and task feasibility:

The outcomes of using ROS Reality to control the Baxter robot in virtual reality were largely positive. Of the 12 tasks that could be completed with a single manipulator, 7 of them were completed in virtual reality (VR). In a similar vein, of the twelve manipulator tasks, eight were accomplished by direct manipulation, and seven of those eight were finished using virtual reality. The VR trials were carried out by experienced teleoperators, who reported that the system was user-friendly. When it came to tasks that the robot could physically perform, the most effective method was direct kinaesthetic manipulation, in which the operator guided the robot's movements with their hands. VR proved to be especially useful, nevertheless, for tasks requiring intricate joint movements in the robot. Without having to continuously parameterize joint angles, the robot in virtual reality (VR) could determine the pose of the end effector based on the user's input, and then execute the appropriate trajectory.

The trials showed that task success was dependent on the robot's limited force exertion capabilities. The robot found it difficult to perform some heavy-duty tasks like opening a bag of chips or tossing and catching a ball. The robot's parallel electrical grippers were also a hindrance for tasks requiring dexterous grasping or rotation.

Chapter 3 - METHODOLOGY

3.1. Overview:

According to the methodology, the design and development of the gripper mechanism comes first in hardware development, with the goal of producing a flexible system that can precisely and safely grasp and release objects. In parallel, it includes designing and integrating a gimbal system, which is necessary to provide multi-axis movement and stability. To guarantee that the robot's mounted cameras rotate smoothly, this entails choosing the motors and control algorithms. The coordination of hardware components depends on the integration of the components on TurtleBot 3 Waffle Pi.

When it comes to software development, the priority is to use Unity to create a virtual environment that will act as a digital model of the robot's operating environment. To simulate real-world interactions, this entails creating and modelling 3D objects, putting physics simulations into practice, and arranging lighting and textures. The gripper mechanism and control systems are then developed in Robot Operating System (ROS), emphasizing responsive and easy-to-use controls. Writing code to interface with hardware, putting control algorithms into practice, and incorporating feedback for closed-loop control are all required for this. To facilitate remote teleoperation from any location with internet access, efforts are finally made to transfer functionality from a local area network (LAN) to a Wide Area Network (WAN). This includes setting up network configurations, putting data communication protocols into practice, guaranteeing security, and reliability of remote connections.

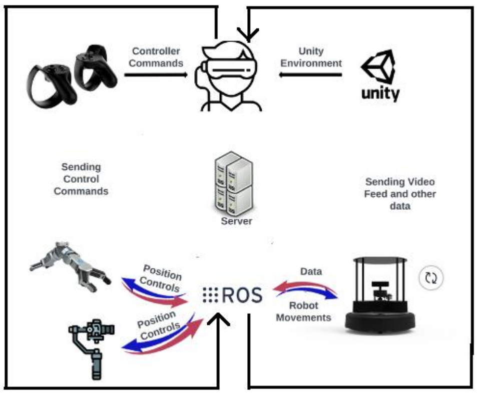


Figure 24. Block Diagram

3.2. Hardware development:

3.2.1. 4-axis robotic arm (EEZYBOTARM MK2):

The EEZY bot arm mk2 is chosen as the project's base because it meets our goals of developing a prototype for medical pick and place operations in several important ways. This robotic arm is chosen in part because it is open source, meaning one can access the design files, software code, and community support needed for development and customization. We could customize the arm's functionality to fit the unique needs of medical pick and place operations by utilizing an open-source platform, guaranteeing compatibility with our suggested domain. Furthermore, the EEZY bot arm mk2's modularity and adaptability made it the perfect option for our project. The arm's modular design makes it simple to assemble and customize. It consists of servo motors, parts that are 3D printed or laser cut, and an Arduino microcontroller. Because of its modularity, we were able to modify the arm's design to meet the requirements of medical pick and place operations, including the need for accurate handling of fragile objects and suitability for use with materials of medical grade.

3.2.1.1. Design and development:

Apart from being modular and open source, the EEZY bot arm mk2 is customized to improve its functionality for our project, specifically concerning medical pick and place procedures. To achieve a greater payload capacity and extended reach which are necessary for precisely handling medical supplies and equipment, the arm's size and strength were increased. In addition, several new features were added to enhance usability and functionality. Among these were a quick coupler mechanism and replaceable gripper, which allowed for quick gripper type changes to effectively adapt to various medical tasks. To maintain organized cable management and reduce the possibility of cable entanglement while in operation, internal cable routing was incorporated into the robot's main arms. Furthermore, the arm's vertical axis was supported by spheres, which improved stability and fluid motion. This was especially important for delicate medical procedures that called for exact positioning and manipulation. Together, these improvements made the EEZY bot arm mk2 more suitable for medical pick and place tasks and brought it closer to the specifications of our suggested domain.

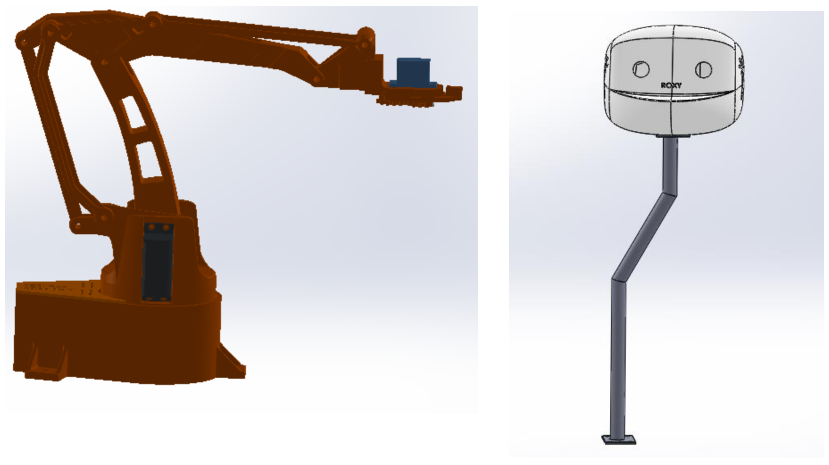


Figure 25. 4-axis robotic arm and camera mount (CAD model)

3.2.1.2. 3D Printing:

Due to its larger size and more complex geometries, the EEZY bot arm mk2 proved to be challenging to print. To accommodate their size, some components, like the main horizontal arm, required the use of supports during printing and a minimum printing area of 200x200 millimetres. Throughout the assembly process, metric hardware was used, with M4 screws rotating every joint. As an alternative, #8-32 screws may be used in place of M4 screws. Because the joints' holes were made to fit tightly together, it was possible to precisely adjust their diameter with a drill bit to achieve the best possible functionality. To ensure smooth movement with minimal clearance between components, self-locking nuts were used on the screws. They were tightened until the joint was secured and then loosened. Furthermore, M4 threaded rods were also used on the two axes of the main vertical arm, which added to the arm assembly's sturdy design and stability.

In addition, the choice to 3D print the EEZY bot arm mk2 with PET-G material at a 30% infill rate was made with affordability, strength, and durability in mind. PET-G is a popular thermoplastic polymer that is well-suited for 3D printing because of its high strength-to-weight ratio, resilience to impact, and other attributes. We attempted to achieve a balance between material efficiency and structural integrity by using PET-G material with a 30% infill rate. This would ensure that the printed components could withstand the rigors of medical pick and place operations while optimizing material usage and lowering production costs.

Here's the Bill of Materials (BOM) for the 3D printed and non-printed parts required for assembling the EEZY bot arm mk2. 3D printed parts include base, main arm, v arm, link 135, link 135 angled, horizontal arm, horizontal arm plate, triangular link, link 147, triangular link front, gear servo, gear mast, main base, lower base, claw base, claw finger, claw gear drive, claw gear driven and drive cover. The non-printed parts include 3x 946 servo, 1x SG90 servo, 1 x M6 self-locking nut, 1 x M6x25 screw, 2 x M3 self-locking nuts, 2 x M3 x 20 screws, 1 x M3 x 10 hex recessed head screw, 9 x M4 self-locking nuts, 1 x M4 x 40 screw, 1 x M4 x 30 screw, 5 x M4 x 20 screw, 1 x M4 x 60mm threaded rod, 1 x M4 x 32mm threaded rod, 25 x diameter 6 mm ball spheres, 1 x 606zz bearing and some M4 washers.

This comprehensive list includes all the necessary components for building the EEZY bot arm mk2, covering both the 3D printed parts and the additional hardware required for assembly.

3.2.1.3. Assembly:

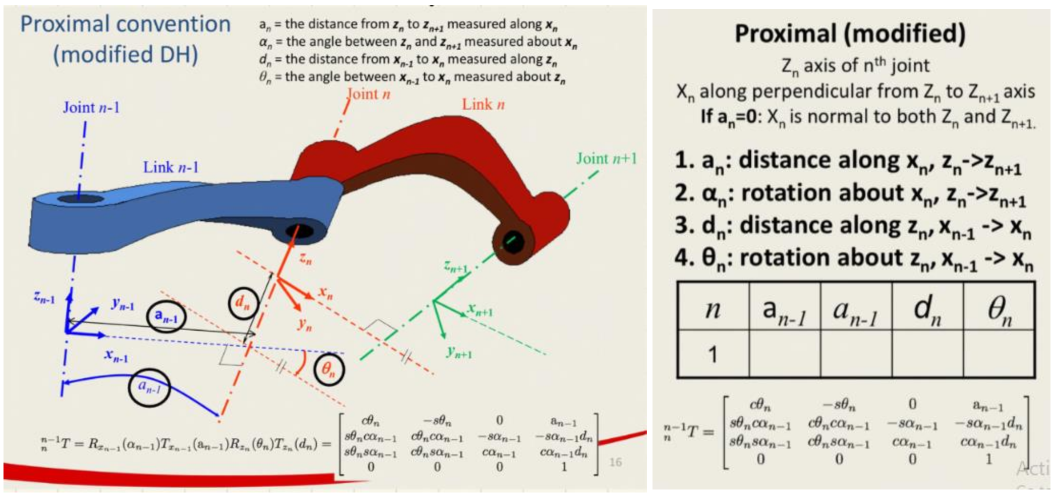
These were the steps involved in assembling the EEZY bot arm mk2. First, align the driving shaft of an MG946 servo forward and use the included self-tapping screws to secure it to the main base. Then, put three M3 nuts into the main base's receptacles. Then, to ensure that the 606 bearings have freedom of movement, insert them into their housing and use three M3 screws to secure the plate to the main base. Using tiny self-tapping screws, secure the upper driving printed gear to the drive plate by positioning it on the splined shaft. Over 25 spheres with a diameter of 6 mm were placed along the path.

Once the swivel base's receptacle is filled with an M6 self-locking nut, position the geared base, and fasten it with a few M3 screws and nuts.

Attach the two components together with an M6 screw. After completing the main base, attach the vertical drive lever and main arm to the horizontal axis of the main base with a 4mm diameter rod. Use the eight self-tapping screws and single horns that come with the servos to secure the two in place, making sure they are aligned with the arm housing. Attach the driving arm to the lower end of the straight lever and the fixed end of the base to the lower end of the angled lever.

Attach the triangle and horizontal arm to the upper section of the main arm using a threaded M4 rod. Join the angled rod to the triangle and the straight rod to the main arm. Fasten the gripper's fast release and rod to the front portion of the horizontal arm. Now that the robotic arm is put together, proceed with the gripper assembly. Make sure the servo wire can pass through the interior space of the horizontal arm before attaching the gripper to the fast release at the end of the arm. Gather all the cables on the back of the robot, enabling it to move freely, particularly when rotating on its vertical axis. After completing these procedures, the EEZY bot arm mk2 is now mechanically assembled and ready for electronic control.

3.2.1.4. Forward Kinematics:



Proximal convention (modified DH)

a_n = the distance from z_n to z_{n+1} measured along x_n
α_n = the angle between z_n and z_{n+1} measured about x_n
d_n = the distance from x_{n-1} to x_n measured along z_n
θ_n = the angle between x_{n-1} to x_n measured about z_n

$$ {}^{n-1}_{\ n}T = R_{x_{n-1}}(\alpha_{n-1})T_{x_{n-1}}(a_{n-1})R_{z_n}(\theta_n)T_{z_n}(d_n) = \begin{bmatrix} c\theta_n & -s\theta_n & 0 & a_{n-1} \\ s\theta_n c\alpha_{n-1} & c\theta_n c\alpha_{n-1} & -s\alpha_{n-1} & -s\alpha_{n-1}d_n \\ s\theta_n s\alpha_{n-1} & c\theta_n s\alpha_{n-1} & c\alpha_{n-1} & c\alpha_{n-1}d_n \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

Proximal (modified)

Z_n axis of nth joint

X_n along perpendicular from Z_n to Z_{n+1} axis

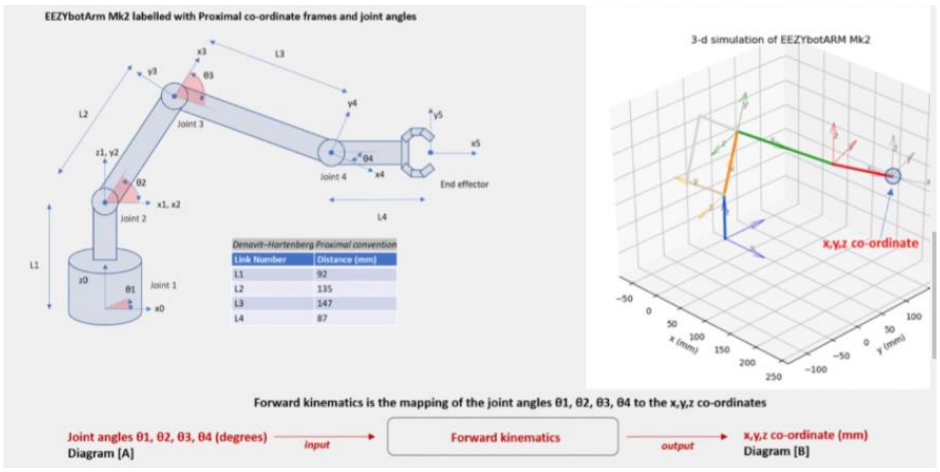
If a_n=0: X_n is normal to both Z_n and Z_{n+1}.

1. **a_n: distance along x_n, z_n->z_{n+1}**
2. **α_n: rotation about x_n, z_n->z_{n+1}**
3. **d_n: distance along z_n, x_{n-1} -> x_n**
4. **θ_n: rotation about z_n, x_{n-1} -> x_n**

n	a_{n-1}	α_{n-1}	d_n	θ_n
1				

$$ {}^{n-1}_{\ n}T = \begin{bmatrix} c\theta_n & -s\theta_n & 0 & a_{n-1} \\ s\theta_n c\alpha_{n-1} & c\theta_n c\alpha_{n-1} & -s\alpha_{n-1} & -s\alpha_{n-1}d_n \\ s\theta_n s\alpha_{n-1} & c\theta_n s\alpha_{n-1} & c\alpha_{n-1} & c\alpha_{n-1}d_n \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

Figure 26. Forward Kinematics (DH Parameters)


EEZYbotArm Mk2 labelled with Proximal co-ordinate frames and joint angles
3-d simulation of EEZYbotARM Mk2
Joint 1
Joint 2
Joint 3
Joint 4
End effector
x,y,z co-ordinate
Joint angles θ1, θ2, θ3, θ4 (degrees)
Diagram [A]
input
Forward kinematics
output
x,y,z co-ordinate (mm)
Diagram [B]

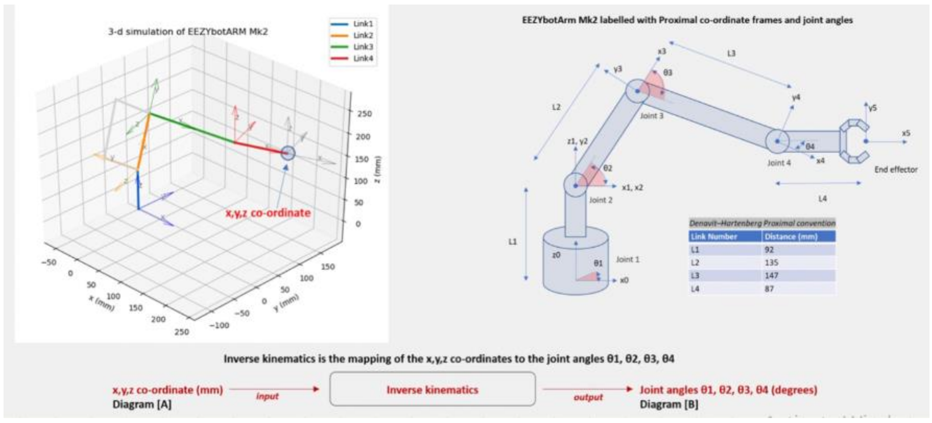
Forward kinematics is the mapping of the joint angles θ1, θ2, θ3, θ4 to the x,y,z co-ordinates

Denavit–Hartenberg Proximal convention

Link Number	Distance (mm)
L1	92
L2	135
L3	147
L4	87

Figure 27. Forward Kinematics (Workspace)

3.2.1.5. Inverse Kinematics:



Inverse kinematics is the mapping of the x,y,z co-ordinates to the joint angles θ1, θ2, θ3, θ4

Denavit–Hartenberg Proximal convention

Link Number	Distance (mm)
L1	92
L2	135
L3	147
L4	87

Figure 28. Inverse Kinematics

3.2.2. Integration of TurtleBot3 Waffle Pi:

Since Raspberry Pi functions as the brains of our robotic system, we decided to use TurtleBot 3 TurtleBot 3 Waffle Pi for our robot because of its versatility and compatibility. Among the many features that TurtleBot 3 TurtleBot 3 Waffle Pi provides especially for robotics applications are motor drivers, servo controllers, and GPIO headers. It is the perfect choice because of these features, which make it simple to interface with the different sensors, actuators, and peripherals needed for our robot's operation.

Furthermore, TurtleBot 3 Waffle Pi's seamless integration with the Raspberry Pi platform offers a streamlined and practical way to control our robot. Because of its plug-and-play design, we can concentrate more on the software development and implementation aspects of our project since it makes the hardware setup process simpler. In the robotics community, TurtleBot 3 Waffle Pi is well-supported, and there is a wealth of documentation and resources available to support our development work. All things considered, TurtleBot 3 TurtleBot 3 Waffle Pi's compatibility, adaptability, and simplicity of use make it a good choice for our robot and fit in nicely with the specifications and objectives of our project.

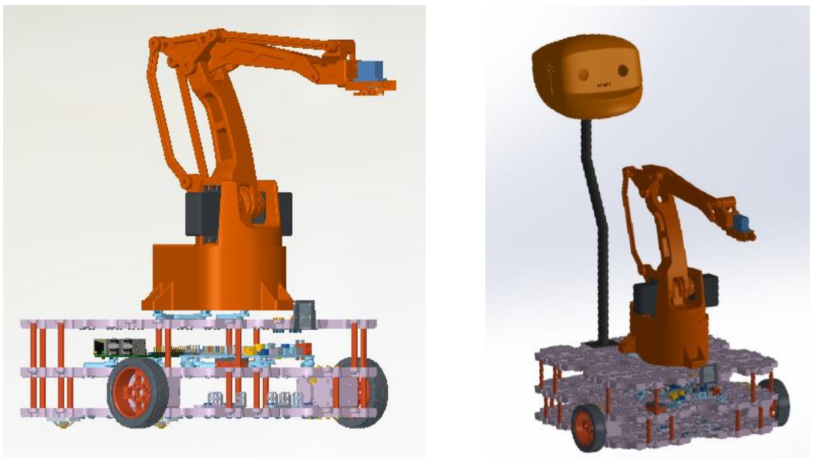


Figure 29. Integration of TurtleBot 3 Waffle Pi & Robotic Arm

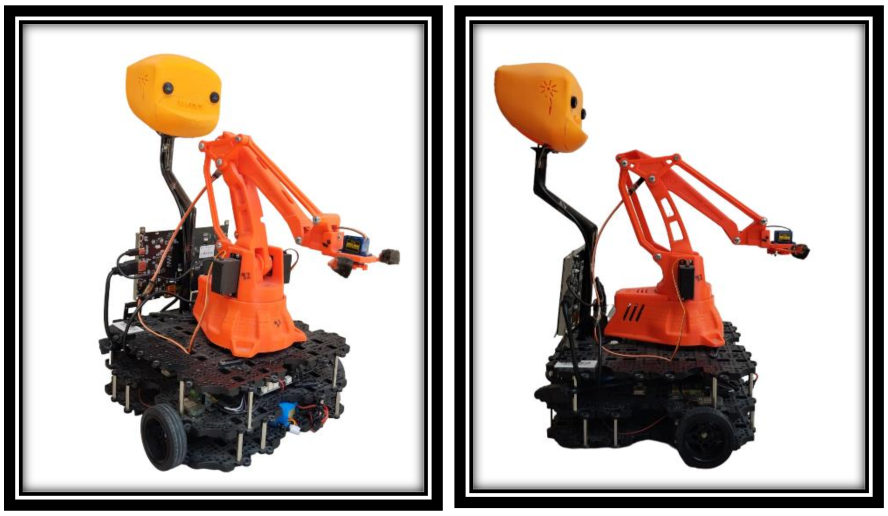


Figure 30. TurtleBot3 assembly (Hardware)

3.3. Software development:

3.3.1. Unity:

Unity is ideal for app development for the VR platform because of the tools it provides and the features that are designed to help create immersive environments for Oculus Quest 2. This is because through Unity, app developers can fully harness the features of the Oculus Quest 2 device without having to worry about compatibility issues. Due to the screen-based concept of the software, it allows developers to experiment with their concepts and modify them according to results. It can be easily used, and it enhances collaboration and the time taken for development is relatively low.

Furthermore, Unity allows for high-level customization and boasts a wide range of tools and assets for developing rich and engaging gameplay, from designing intricate 3D models to implementing complex interactions and mechanics. Additionally, Unity's robust asset store and strong community of users offer a plethora of pre-made tools, packages, and project templates that help to speed up the work and encourage uniqueness. Thus, due to the numerous features, clear navigation, and customization, Unity is a suitable software environment for developing virtual telepresence robot applications. It contains all that is required to develop attractive and engaging content for users of the Oculus Quest 2 platform.

3.3.1.1. Project Insight:

1. We are using Unity to build a scene in which users can view 180-degree videos that were captured by the telepresence robot's built-in cams. The unique feature of 180-degree videos is that they make the viewer get involved in the virtual space and look around from any angle. To make this possible, a video is recorded from every angle which makes the user feel the environment as if they are observing it. These 180-degree videos can be easily integrated into the virtual world by leveraging different aspects of Unity and provide users with a telepresence feel that is genuinely realistic.

2. Movement of the controller boundary in unity coordinate space for the robotic arm and the robot's hand gesture control are other parts of Unity. To this end, the real-time controller movement of the user from the ultimate interface is successfully captured and interpreted, and the actions that the robot and its arm can control are determined. Taking orientation and position: since the controllers allow for six degrees of freedom, the transition of their movement to the XYZ coordinate system of the robot helps users shape its movements in the virtual world. In this way, having such amount of control, the user can operate the telepresence robot in a regular and smooth manner, which reinforces the sense of presence and ownership.

3. To feed the via Robot Operation different control functionalities of the robot and its arm, we output certain controller button states from Unity. These include the ability to start and stop the overall functioning of the robot, control the rate of acceleration, operate the mechanical arm appendage, control the position of the gripper as well as the base frame of the robot. This is because users can monitor the state of the controller buttons and as a result, they are able to dictate on how the telepresence robot must behave or function. This means they can cause reactions or instructions in the discharged simulated arena. This control mechanism is very friendly to engage the user and it also helps them to perform their tasks and operations within the shortest period as well as with the least error rates.

4. Last but not the least; Web Real Time Communication protocol of the Unity has been incorporated to provide connection with Wide Area Network. The telepresence robot application, as well as distant users, may share data using the internet with real-time interactions. By implementing WebRTC in Unity scripts, we are also likely to allow for seamless streaming of videos, transfer of data and audio communication between the robot and users located globally. The usefulness and practicality of the telepresence system further enhance this aspect that enables a person to control the robot in real-life and effectively participate in the virtual environment even if physically absent.

3.3.1.2. Unity environment creation for Virtual Telepresence Robot:

There are several configuration settings that are inherent in any Unity project and create the basic context of the virtual world. These include basic settings like the main camera, origin point, lighting setups, and the default appearance of the scene.

1. Main Camera:

The main view of the Virtual environment is from the main camera. The primary camera is the one generated by unity, which gives users the view and interaction point to the scene. Thus, the position, orientation, and the field of view of the main camera can be changed to achieve the desired result and give the user a unique experience.

2. Origin Point:

The scene origin or world origin is the box where objects are placed with respect to the scene. In Unity, the origin point is commonly located at the scene space with X = 0, Y = 0, Z = 0 axis in the scene space. This origin point is fundamental for all the other objects in the scene making organizational and transformational work easy.

3. Lighting Setup:

There is no doubt that illumination plays a crucial role in setting the mood and the general feel of the environment. Directional light sources are one of the default light rigs that are included in Unity when a new project is created. This directed light also provides general lighting of the scene with the intention to mimic sunlight. In addition, they may adjust the lighting to add ambient lighting to the default lighting of Unity to make the shadow softer while making the environment appear more intentional.

4. Default Scene Structure:

Scene is a common organizational structure in Unity projects and is used to contain and display assets, game objects and other components of a virtual environment. Using Unity game developer tool, the default scene that is created for each project is “Sample Scene”. The default scene is a simple cube that includes a primary camera, light, and an empty game object setting the scene of the cube. To fit the needs of their unique projects, game elements and objects from the scene structure can be added, removed, and changed.

3.3.1.3. XR origin in VR unity development:

To be more precise, the XR origin in our Unity project is the point of view of the user as well as the scene’s reference point. Specifically, the XR origin represents both the viewer’s position and the starting point of the virtual environment when using Oculus device, such as the Oculus Quest. The users are presented with a seamless and engaging experience as soon as they put on the Oculus headset and turn their heads, which results in the XR origin (and the primary camera) correctly moving within the scene.

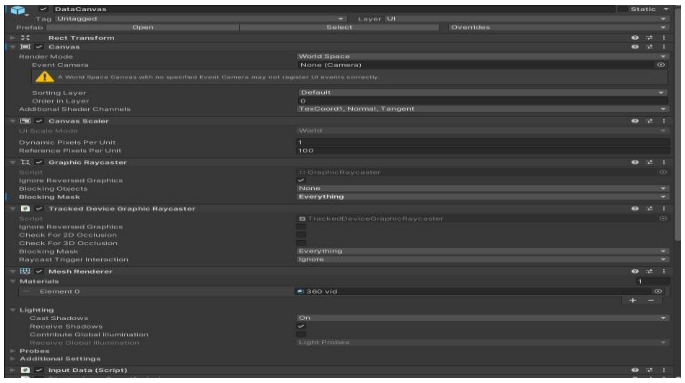
Allowing users the ability to interact with a virtual environment as if they were physically in that environment is an essential element for immersion in VR environments. It means that Unity ensures that the user can navigate through the virtual environment freely, as they would do in real life since it synchronizes the XR origin with the user’s head movements. This alignment of the XR origin with the user’s point of view makes the VR experience more engaging and believable, enhancing the user’s feeling of presence in the virtual reality environment.

Moreover, Unity makes developing for VR applications seamless by positioning the user at the core of the XR origin. Rather than being occupied with delicate movements of cameras and interactions for users, developers can focus on creating and emulating the world. This allows developers to concentrate on creating compelling VR content and experiences because Unity is responsible for determining the XR origin based on the user’s movements. In conclusion, the XR origin is the foundation of our Unity project that can create effective and immersive VR experiences for individuals using Oculus.

3.3.1.4. Data Canvas:

The Data Canvas is a fundamental component of our Unity project as it stores the significant scripts and works as the UI centres of the virtual environment. Tied to this canvas are several scripts that are designed to control and output the necessary information to the user, including the coordinates of the controller and the status of the controllers. The Data Canvas is another UI component that ensures the user receives certain feedback and information required for proper interaction with the environment.

Providing the users with the necessary information about the functioning of the VR system is one of the primary functionalities of the Data Canvas. This includes information about the controller's position and location, as well as the status of its many buttons and inputs. All of this allows to place all the necessary information into a single element of the user interface that can be easily accessed and understood by the user while being immersed in the virtual reality environment. Also, the Data Canvas is one of the few graphical users interface (GUI) elements in the project. Thus, it serves as a central node where the user receives essential feedback and directions without compromising the overall VR experience. To provide the user with easy access to information while maintaining a sense of presence and embodiment in the virtual environment, the elements of the UI are thoughtfully placed within the Data Canvas. In conclusion, the Data Canvas is crucial in enhancing the user interface and fostering effective communication in Unity project.

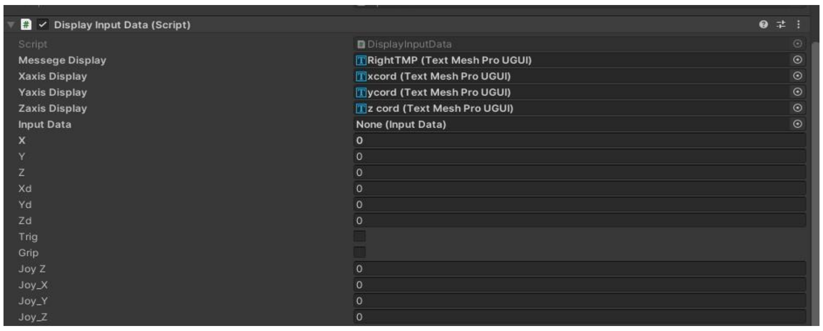


```
DataCanvas                                              Static
Tag Untagged                         Layer UI
Prefab            Open                  Select              Overrides
Rect Transform
Canvas
Render Mode                          World Space
  Event Camera                       None (Camera)
  A World Space Canvas with no specified Event Camera may not register UI events correctly.
  Sorting Layer                      Default
  Order in Layer                     0
Additional Shader Channels           TexCoord1, Normal, Tangent
Canvas Scaler
UI Scale Mode                        World
Dynamic Pixels Per Unit              1
Reference Pixels Per Unit            100
Graphic Raycaster
Script                               GraphicRaycaster
Ignore Reversed Graphics             ✓
Blocking Objects                     None
Blocking Mask                        Everything
Tracked Device Graphic Raycaster
Script                               TrackedDeviceGraphicRaycaster
Ignore Reversed Graphics
Check For 2D Occlusion
Check For 3D Occlusion
Blocking Mask                        Everything
Raycast Trigger Interaction          Ignore
Mesh Renderer
Materials                                                              1
    Element 0                        360 vid
Lighting
   Cast Shadows                      On
   Receive Shadows                   ✓
   Contribute Global Illumination
   Receive Global Illumination       Light Probes
Probes
Additional Settings
Input Data (Script)
```

Figure 31. Data Canvas

The size of the pixels on the canvas is determined by the canvas scalar attribute. This setting is crucial for ensuring that the UI elements drawn on the canvas are appropriately scaled and placed in the virtual environment. The overall scale and visual presentation of the UI elements can be controlled by modifying the canvas scalar, which will optimize the user experience across a range of display resolutions and devices. The virtual environment's UI elements' readability and visual fidelity are influenced by the canvas scalar attribute. Furthermore, the canvas scalar contributes to the overall usability and accessibility of the VR application by ensuring consistency in the presentation of UI elements across various screen sizes and resolutions.

3.3.1.5. Display Data Input:



Display Input Data (Script)

Script	DisplayInputData
Messege Display	RightTMP (Text Mesh Pro UGUI)
Xaxis Display	xcord (Text Mesh Pro UGUI)
Yaxis Display	ycord (Text Mesh Pro UGUI)
Zaxis Display	z cord (Text Mesh Pro UGUI)
Input Data	None (Input Data)
X	0
Y	0
Z	0
Xd	0
Yd	0
Zd	0
Trig	☐
Grip	☐
Joy Z	0
Joy_X	0
Joy_Y	0
Joy_Z	0

Figure 32. Display Data Input (Script)

Variables identified in the “Display Input Data” script are visually depicted as data that is displayed on the canvas. This script is responsible for reading information from the Oculus controller and providing visual representation through blue screen environmental objects. We utilize the information from the Oculus controller to provide the user with constant feedback of the position and status of such objects in this virtual reality environment. The graphical representation offered in the current study is helpful and unambiguous, thereby enhancing the user experience and ensuring seamless interaction with the VTRS.

“DisplayInputData” is a script that we use for our virtual telepresence robot project, and it oversees input data from Oculus controller specifically menu in Unity environment. The script starts off by calling the InputData component attached to that same GameObject, which lets the script pull input data from the Oculus controller. In the update method, the script constantly monitors changes in the inputs to the Oculus controller including device velocity, grip button, trigger button, and Joystick. This allows the script to prompt the user for further input based on the input received from the Oculus controller and updates the display text accordingly. For instance, it displays messages such as showing that the grip button is pressed or the trigger button is pressed, in such a way that the user is supposed to do something. The script changes the string shown on the specific fields of the interface to indicate to which values of the X, Y, Z axes the current position of the controller corresponds to. These are derived from the Oculus controller device position and the x, y joystick action inputs. Moreover, the possibility to change the origin point is also implemented and the user can do it by pressing the primary 2D axis button on the Oculus controller. Activated by this action, the origin coordinates are then modified, creating a basis for further positions.

The "InputData" script within Unity is crucial for configuring and managing the inputs that are required to interact with the virtual environment. This script handles the initialization of the head-mounted display (HMD), left and right controllers, and other significant input peripherals. Every input device is maintained valid by the script in the Update function. To initialize a device again if any is found to be invalid the InitializeInputDevices function is used. This approach defines input devices based on certain parameters including whether it is a head-mounted display or a controller (left or right). with the help of InputDevices. The script GetDevicesWithCharacteristics searches for devices that match the given characteristic and assigns the first device from the list to the proper input device variable. By so doing, the InputData script confirms that important input devices are correctly instantiated to enhance interaction between users and the Unity environment.

3.3.1.6. ROS connector:

The ability to connect Unity to ROS environment on other devices like

Raspberry pi is facilitated by a software known as ROS Connector. This script has been developed by Siemens AG to interact between Unity based applications and systems that are controlled by ROS. Since ROS Bridge acts as a mediator between the two environments, it enables the transfer of data and control commands. When the ROS Connector starts, it links using the selected protocol (for example, e. g. It connects to the ROS Bridge server at the provided URL, which is often located on the Raspberry Pi. Unity can continue with its tasks as the formation of the connection happens in the background. After that, the ROS Socket instance is created to provide the necessary interface for message sending and receiving between Unity and ROS.

The script triggers the Is Connected event on the successful connection to inform that it has established the connection with ROS Bridge. On the other hand, a warning message is logged in case the connection attempt fails within the set timeout length to notify the user of the fail. Several event handlers, including On Connected and On Closed, are utilized throughout the script to manage connection state changes. These handlers allow for the right actions to be performed in the Unity application when connection is established or lost with ROS Bridge server.

3.3.1.7. Video display:

To play the video in the Unity environment, we used a technique of applying a 360-degree video on the skybox. The skybox captures all the empty space outside the environment and acts as the backdrop of an immersive environment. This involves importing the video, and mapping it to the skybox panorama, which would allow the video to be incorporated naturally into the Unity environment. Using the skybox to render the video also guarantees that the video occupies the whole environment and does not leave out details that might interest the users. This is in line with our objective of establishing a virtual world telepresence that allows users to actively interact with the environment and the robotic system.

1. Simple Media Stream Receiver:

This script is designed to take the video stream from the Raspberry Pi using WebRTC (Web Real-Time Communication). It connects with the specified server IP and port using WebSocket and enables the transmission of signalling and video information. When receiving the video stream, the script assigns the received video track to the skybox material texture, enabling real-time visualization of the video content within the Unity space.

2. Simple Media Stream Sender:

On the other hand, this script is the sender script that oversees sending the video stream from the Raspberry Pi to Unity. Just like the receiver script, it opens a WebSocket connection to the server IP and port and sends signalling messages to facilitate the streaming of videos. Furthermore, the script activates the video stream with the USB camera as the source for the live video feed necessary for capturing and transmitting the live footage to Unity.

These scripts, along with the rendering of the video on the skybox, facilitate sustained video playback and the integration of the video into the Unity environment, thus enrich the experience of users who are dealing with the Virtual Telepresence Robot.

3.3.2. Robot Operating System (ROS) integration in virtual telepresence robot:

As the Virtual Telepresence Robot enlists the powerful Robot Operating System (ROS) for its operation, this project stands as the frontier research where robotics and telepresence converge. As for the technical side, our project’s foundation is ROS which allows for managing and coordinating multiple parts of the system efficiently. This section provides a deeper look into ROS since it is a fundamental part of our architecture and how it is more flexible in managing the interactions of many components while enhancing the users’ experience through the integration of augmentations.

The most significant way to connect all the components of the telepresence robot is by using ROS, which is famous for its flexibility and expansiveness. This is made possible through the intricate integration of the hardware and software components using the ROS tool and library set which appears to provide direction in effective system integration. In addition, new features and functionalities can be added to ROS due to its modular design because the approach ensures adaptability depending on the project requirements at the initial and later stages of development.

The use of ROS to support data sharing and communication within numerous system components is one of its greatest strengths. The remarkable feature of ROS for our telepresence robot is that all its components can share data with each other through the effective messaging mechanism in real time. This feature is crucial for synchronizing complex actions and operations like multimer diameter user interfaces and auto-driving navigation.

In addition to that, ROS let us leverage a vast number of libraries and code packages available, which make the development process faster and more efficient. ROS also provides access to a wide library of freely available programs, which allows using ready-made solutions for routine robotics issues. It also saves more of our time so that we can spend most of our time focusing on innovation and where and how to customize according to the requirements of our project.

3.3.2.1. System overview:

The core unit of the Virtual Telepresence Robot is the TurtleBot 3 Waffle robot which is fitted with a state-of-the-art robotic arm as well as a 180° camera of high definition. Indeed, the listed components of hardware are the physical realization of our telepresence system that makes it possible to change and interact in different environments.

Our robot relies on Raspberry Pi 4 (single-board computer) with Ubuntu 22 installed to perform its computations. The Raspberry Pi acts as the Master Processor and controls all the complex processes that are required to deliver the various telepresence features. Raspberry Pi serves the computational core of our telepresence robot well and blends seamlessly into our system design because of its reliability and support for ROS 2. 0.

ROS 2. 0 acts as the central point of the system and brings together all parts and roles of the system. As the central platform for robot programming, ROS 2. 0 provides interfaces between various parts, which allows for the exchange of information and command processing. By leveraging ROS 2. 0's features, our telepresence robot has a modular and extensible structure to easily adopt new possibilities and functions.

Another important step is to develop an individual ROS package for our project to address the challenges we face. We have added two subscriber nodes in this extended ROS package to take information from Unity that is the game engine for our VR frontend. These subscriber nodes are critical to our ROS ecosystem as it provides data flow interface between Unity and the actual world which should seamlessly convert data from the simulated environment to the real one and vice versa. The ROS package that we recently developed enables users to control virtual entities directly in real-world environments via the TMS' physical parts, seamlessly integrating the participant into the tele present environment.

3.3.2.2. ROS integration for robotic arm control:

Since ROS is the link that provides a connection between user input, in this case an Oculus Quest 2 VR headset, to precise manipulation of the robotic arm, it is a fundamental node in controlling the arm. Our approach is based on the usage of the powerful ROS 2. 0 messaging system to synchronise the movements and actions of the robotic arm with the motions of the Oculus controller, so that the user of the telepresence robot can communicate fluently.

To achieve this synchronization, the control data of the robotic arm is stored in what is called the 'quaternion messages', which is a data format that is used to describe rotations in three dimensions. These quaternion messages include commands on how to move the claw and information regarding the position of arm in terms of x, y, and z axis. These messages are received by the Raspberry Pi, which forms the core of computation within our system, when it is issued across ROS topics.

To be more specific, these quaternion messages are published to the ROS topic called “controller_data” that should be utilized for the arm control communication. This enables Raspberry Pi to receive control commands from the Oculus controller through wireless signals. These control commands are upon reaching Raspberry Pi are then controlled and interpreted to enable flawless servo operation of the robotic arm. To achieve the required movement of the arms, the Raspberry Pi in turn interfaces with an Arduino Uno board that is connected serially with the Raspberry Pi. The Arduino Uno works as an interface device that translates commands into actual signals that immediately manage the servos used in controlling the movement of the robotic arm.

Due to this intricate communication and control mechanism provided by ROS, our system can control the direction and movement of the robotic arm effectively and cohesively as per the user inputs received through the Oculus Quest 2 VR headset. This seamless integration between hardware and software shows the effectiveness of ROS in our telepresence robot control system.

3.3.2.3. Control of robot movement:

In addition to the control of the position of the robotic arm, ROS also controls the movements of the entire robot and make it move smoothly within the physical environment. This is made possible by the data which is received from the Oculus controller joystick, which gives linear and angular velocities that show the desired movement of the robot.

To achieve this, the data from the joystick is put into twist messages, a type of ROS message meant for passing velocities. These twist messages contain two main variables: linear and angular velocities both as an instance of geometry_msgs / Vector3 message type. In the geometry_msgs / Vector3 structure, there are three variables – x of the float64 data type, y of the float64 data type, and z of the float64 data type, which describe the magnitude and direction of the velocity in the three-dimensional space.

After being encapsulated within twist messages, this velocity data is broadcasted to the specified ROS topic named “cmd_velocity. These twist messages are transmitted through ROS to the “cmd_velocity” topic hence creating a communication channel through which motion signals from the Oculus controller are provided to the robots control system.

When the TurtleBot software gets these motion commands, it interprets the twist messages and calculates the changes required to enable the motion of the robot. This decoding involves converting the linear and angular velocities that are contained in the messages in the twist messages into correct movements of the wheels which is important in directing the robot within the real world.

These modifications are made through the TurtleBot software which communicates with the Open CR board, a master control unit responsible for managing the robot’s components. Based on received twist messages, the Open CR board receive commands through this channel about required wheel rotation. Following that, it effectively adjusts the speed and direction of rotation of the robot wheels, which translates the specified linear and angular velocities to coordinated motion in the real environment.

3.3.2.4. Integration with TurtleBot ecosystem:

However, it is important to emphasize that the whole TurtleBot platform is based on ROS, which proves the popularity and versatility of the given framework in the robotics field. This compatibility with ROS underscores the versatility and connectivity of TurtleBot’s hardware and software elements, further strengthens its utility as an all-encompassing robotic system.

As for the strengths of using ROS within the TurtleBot environment, it is crucial to note that many predefined ROS actions and services are available for different robotic tasks. These are pre-canned actions for tasks like mapping navigation and avoiding obstacles, and there are services for calibrating, initializing, and diagnosing the sensors. Through the accessibility of these ROS actions and services, our project leverages on a range of tools and utilities that enhance the development and implementation of robotic applications.

3.3.3. WebRTC:

Wide Area Networks or WANs are particularly critical in modern society where distance can be a major barrier to communication and data transfer. WANs are large networks that link several LANs or other WANs to allow organization or persons to interact and divide resources over long distances. These networks make use of leased lines, fibre optics, satellite links among other to ensure connectivity between different geographical areas.

Perhaps the most widely used amongst WAN applications is data transfer, such as videos, from one peer to another. As video content has grown more popular on the Internet, while high-quality streaming services have become more in demand, competent mechanisms for the transfer of video files play a vital role in achieving the most important goal – the user’s entertainment. Specifically, the transfer of video from one peer to another is carried through the sharing of video data packets over a WAN framework. Many of these processes involve encoding, packetization, transmission and decoding through which each step is significant to overall effectiveness and quality of the transmitted video.

At the same time, the integration of WAN technology with robotic systems opens new Possibilities for implementing real-time control and communication. Situated in distributed systems, robots may interact and cooperate with one another with ROS2. However, Unity, an engine for developing video games, offers a stable working environment for building VR applications. When connecting ROS2 on the robot side to Unity over WAN connectivity, control data can be received from the robot to the Unity environment, enabling one to coordinate representations in Unity with those in physical robotic systems.

In this story, we will examine the possible integration between WAN technology, video transfer protocol, ROS 2 robotics, and Unity. Furthermore, we will examine the integration of ROS2 with Unity through WAN channels, shedding light on the implementation of C# scripts to facilitate bidirectional communication and synchronization between robotic systems and virtual environments.

3.3.3.1. Transmitting Unity XR Data to Robot (ROS2):

However, there is an important point to note when using ROS2 on the robot side and Unity: Direct interaction between ROS2 and Unity is impossible because of their architecture. To fill this gap, ROS# (ROS Sharp) intervenes on the Unity side. ROS# is an API through which Unity applications can access ROS capabilities. By using ROS# inside Unity, developers can create ROS nodes, publish messages, and subscribe to topics, which helps them connect to ROS environment.

In this architecture, ROS# in Unity sends out data that describes commands or instructions that the robot is to follow. These messages are then sent over the Wide Area Network (WAN) to the robot's ROS2 system to control and interact between the Unity Virtual environment and the real physical robot.

To achieve this, a ROS Bridge is used, and it is hosted on an Amazon Web Services (AWS) EC2 server. The ROS Bridge serves as a middleman that transmits the ROS messages between the ROS# nodes of Unity side and the ROS2 compatible nodes on the robot side. It receives data generated by the ROS# nodes via the WAN connection and maps the collected data into the format that can be directly processed by ROS2.

On the robot side, ROS2 is upgraded with ROSlibjs, a JavaScript API that allows the interoperability of ROS systems through WebSocket connections. Using ROSlibjs, the robot's ROS2 system can connect to AWS EC2 server carrying the ROS Bridge to allow bidirectional communication between Unity and the robot.

Furthermore, a local ROS bridge node is created on the robot side to enhance the reliability of the communication and reduce the response time. This local ROS Bridge provides an interface for the AWS EC2 server to send data and to receive data from Unity in the form of feedback or sensor data. By maintaining a local ROS Bridge, the architecture enhances reliability and reduces reliance on external network connectivity for critical operations.

In essence, the integration between ROS2, ROS#, AWS EC2 ROS Bridge, ROSlibjs, and local ROS Bridge orchestrates a sophisticated communication infrastructure that seamlessly bridges the gap between Unity and the robot. Through this architecture, developers can harness the power of ROS2 for controlling the robot's behaviour while leveraging Unity's immersive virtual environments, thus enabling transformative applications spanning robotics, simulation, and virtual reality.

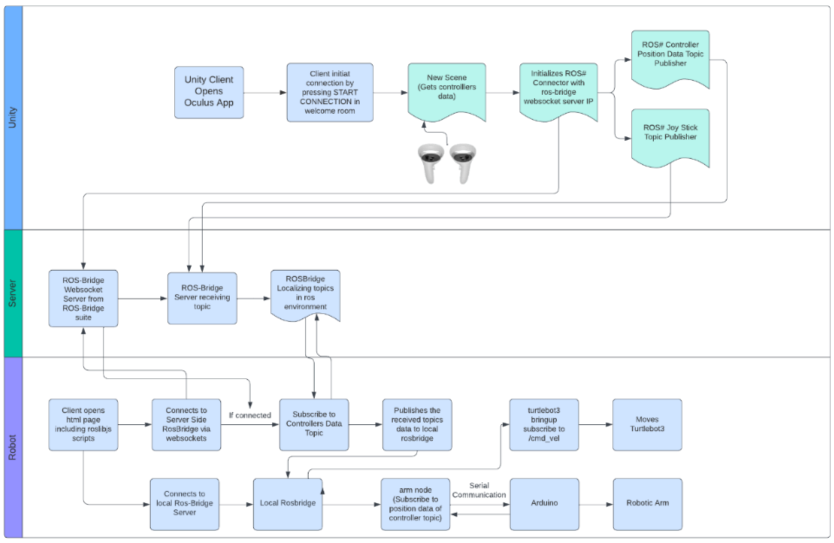

Unity
Unity Client Opens Oculus App
Client initiat connection by pressing START CONNECTION in welcome room
New Scene (Gets controllers data)
Initializes ROS# Connector with ros-bridge websocket server IP
ROS# Controller Position Data Topic Publisher
ROS# Joy Stick Topic Publisher
Server
ROS-Bridge Websocket Server from ROS-Bridge suite
ROS-Bridge Server receiving topic
ROSBridge Localizing topics in ros environment
Robot
Client opens html page including roslibjs scripts
Connects to Server Side RosBridge via websockets
If connected
Subscribe to Controllers Data Topic
Publishes the received topics data to local rosbridge
turtlebot3 bringup subscribe to /cmd_vel
Moves Turtlebot3
Connects to local Ros-Bridge Server
Local Rosbridge
arm node (Subscribe to position data of controller topic)
Serial Communication
Arduino
Robotic Arm

Figure 33. Flow diagram of controls transmitted to ROS2 environment

3.3.3.2. Transmitting Video from Robot to Unity:

In contemporary applications involving the transmission of video from a robot to Unity, several protocols are commonly employed, each with its own set of advantages and limitations. Some of the prominent protocols used includes.

1. RTSP (Real-Time Streaming Protocol):

RTSP is a network control protocol that was primarily intended for use in entertainment and communication systems. It enables streaming of multimedia content over IP networks and supports both live streaming and video on demand services. RTSP is widely in security systems, video teleconferencing, and multimedia streaming applications.

2. RTMP (Real-Time Messaging Protocol):

RTMP is a protocol that Adobe created to allow streaming of audio, video and data over the internet. It is often used for applications such as live broadcasts and real-time multimedia systems, where it offers efficient and real-time communication between the server and the client. However, RTMP has become less used in the recent past because of the coming of other protocols into the market.

3. HLS (HTTP Live Streaming):

HLS is an adaptive streaming protocol created by Apple to deliver multimedia content using HTTP. It splits up the stream into IP segments that are played back through simple web servers, allowing for adaptive streaming and compatibility with most devices and operating systems. HLS is used in the case of streaming the video to a web browser, mobile devices, or smart TVs.

4. WebRTC (Web Real-Time Communication):

WebRTC is an open-source application that allows communicating between browsers and mobile applications in real time with the help of JavaScript. It facilitates the concept of peer-to-peer audio, video and data exchange and makes use of RTP (Real-Time Transport Protocol), SRTP (Secure Real Time Transport Protocol), Interactive Connectivity Establishment (ICE).

1. The low latency:

WebRTC is a technology that was created for real-time communication of media streams including audio and video. This makes it appropriate for use in cases where the interaction between the robot and Unity needs to be fast, such as in the case of teleoperation or remote control.

2. High Quality Video:

WebRTC provides clear, high-definition video capability together with the ability to select adaptive bitrate which makes the video excellent irrespective of the quality of the network connection. This is important for ensuring that there is minimal loss of the visual image quality when displayed in virtual environments created by Unity.

3. Security:

To enhance the security of the traffic, WebRTC uses the encryption and authentication systems, which help to protect data from interception. This is particularly the case in applications where privacy and safety are of upmost important for instance telepresence and surveillance.

4. Cross Platform Compatibility:

WebRTC is compatible with all web browsers, mobile devices, and operating systems, ensuring that it is easily accessible and can interoperate with different systems. This makes it possible to fully integrate this plug-in with Unity applications that can run on any device and environment.

In conclusion, WebRTC provides a suitable solution for streaming video from a robot to Unity as it provides low latencies, high quality, security, and suitability for cross-platform applications to provide natural and interactive experiences in virtual space.

3.3.3.3. How WebRTC works:

WebRTC is a full-fledged open-source suite that facilitates real time communication between browsers and mobile applications through JavaScript APIs. It is employed to support voice, video, and data transfer between peers, allowing for efficient transmission of data with minimal delays, as well as high quality of transmitted media streams. It is worth looking at how WebRTC functions and what stages are necessary to create a WebRTC connection.

Signalling:

The first thing that must occur before any WebRTC connection can be made is signalling. Exchange of session control messages is known as signalling, and it refers to the process in which two peers get to know each other. Peers should be also able to exchange any information like for example network addresses or media capabilities to create direct connection. Nonetheless, WebRTC has intentionally left signalling as an open issue meaning that developers must come up with their own mechanism of signalling.

Some of the common ways of signalling are to use the signalling server, WebSocket, or any third-party signalling service.

Session Description Protocol (SDP) Exchange:

Once peers have discovered each other through signalling, they exchange messages in Session Description Protocol (SDP). SDP is a language for describing multimedia sessions with their capabilities, types of media, codecs, and network parameters. Every single peer creates an SDP offer that includes details about its media capacity and preferred settings. This offer is sent to the other peer through the signalling channel.

The receiving peer sends back an SDP answer that contains its own media capabilities and the selected configuration.

ICE (Interactive Connectivity Establishment):

Following the exchange of SDP messages, WebRTC employs ICE, an acronym for Interactive Connectivity Establishment, to connect the two endpoints directly. ICE is a protocol for how to communicate through NATs (Network Address Translators) and firewalls to determine the optimal path between two points. ICE candidates are network addresses (IP address and port) at which peers are accessible. ICE candidates are exchanged between peers via the signalling channel.

Every peer collects ICE local candidates of its network interfaces and forwards them to the other peer. Peers then initiate connectivity checks to ascertain the most appropriate path to use in communication. After a suitable candidate pair is chosen, the peers create a direct UDP, TCP or TLS connection based on network conditions and security considerations.

Media Stream Exchange:

Once connected, peers can send and receive media streams such as audio, video, and data channels. WebRTC supports multiple media streams per connection to support the simultaneous transfer of audio, video, and arbitrary data. During the exchange of the SDP, peers agree on acceptable codecs and formats of the media for use in a session. SRTP (Secure Real-Time Transport Protocol) is used to provide confidentiality and integrity to the media streams.

Real-Time Communication:

After the connection has been made and the media streams are exchanged, actual real-time communication takes place between the peers. Peers are capable of sharing audio and video data in real time and hence such applications as video conferencing, voice calling, live streaming and online gaming.

In summary, WebRTC facilitates peer-to-peer communication by enabling signalling, SDP exchange, ICE negotiation, media stream exchange, and real-time communication. By following these steps, WebRTC enables developers to create robust, low-latency applications for real-time audio, video, and data sharing over the web.

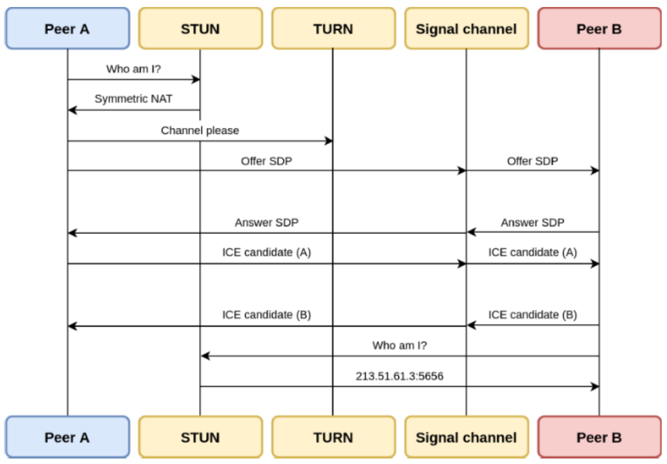


Figure 34. Flow diagram of WebRTC connection

3.3.3.4. Turn Server:

To have a strong and reliable communication for our Virtual Telepresence Robot we implemented a TURN server with an AWS EC2 instance. The TURN server plays an essential role in WebRTC data relay and might be particularly useful in situations where endpoints are unable to establish direct peer-to-peer connections due to overly stringent network configurations. We selected a t2. micro instance with Ubuntu Server 20. 04 LTS for it offered the right blend of affordability and efficiency. Coturn is a widely used TURN server software and was set up to manage the relay services with security and effectiveness. Some of the key configurations were listening and relay ports, user authentication, and SSL certificate for secure connection.

The AWS security groups and the instance firewall settings were carefully configured to allow traffic on the necessary ports so that the TURN server is not disconnected. After the setup, intensive testing was conducted to ensure that the server efficiently and effectively forwards the WebRTC data. This deployment improves our system redundancy to guarantee continuous video and control data stream important for real-time functioning of our telepresence robot.

Chapter 4 -EXPERIMENTAL RESULTS AND ANALYSIS

4.1. Overview:

A thorough examination of the experimental findings and conclusions from our Virtual Telepresence Robot system's performance evaluation are given in Chapter 4. We evaluated network latency and efficiency by closely examining video stream metrics like Bytes Received and Current Round Trip Time. The computation of data rate demonstrated compliance with the requirements for high-definition video streaming, and the low Current Round Trip Time suggested reduced network latency, guaranteeing seamless real-time communication. These results confirm the robustness and data transfer efficiency of the system. In order to improve the telepresence experience, future study may concentrate on additional optimization techniques and the incorporation of sophisticated feedback mechanisms.

4.2. Bytes Sent

4.2.1. Bandwidth Monitoring:

Essentially, bytes conveyed over a certain period can be used to measure the bandwidth consumption of a WebRTC app by developers. This is especially useful in determining the traffic of the network and can be very important in analysing throughput. Bytes sent analysis gives an accurate understanding of the amount of data transmitted at a certain time, which is helpful in keeping the communication channel steady and efficient. Real-time tracking also helps to identify trends and irregularities with data transfer, which enables timely allocation of network resources.

4.2.2. Performance Analysis:

Knowing the number of bytes sent allows to pin-point potential problems in data transfer like high bandwidth usage which requires additional optimization like compression for instance. High bandwidth utilization may suggest that the program is not properly managing the data input/output or that data that is not needed is being transmitted. When the bytes that have been sent are compared, the developer can easily determine when the bandwidth is at its highest and make precise optimizations. Compression of data and design of efficient structures of data packets are frequently used techniques in minimizing the usage of bandwidth while at the same time ensuring data accuracy and efficient transmission.

4.2.3. Debugging and Diagnostics:

It also helps in debugging because one gets to know the amount of data conveyed, this will enable one to note cases like high data usage or transmission delay. If there are problems, such as slow speed or intermittent breaks in connections, this data transmission may indicate whether these are due to large volumes of bytes being transmitted. For example, if bytes sent is significantly higher than bytes received, this may be an evidence of data over-transmission or there might be unoptimized encoding mechanisms. This metric is especially useful in identifying and handling problems before they escalate and affect the overall functionality of the application.

4.2.4. BytesSent is a crucial metric in WebRTC:

It allows the developers to observe and evaluate how much data is transmitted over the network and thereby provide an influence and control on the WebRTC applications successfully. This makes it easy to compare the application's efficiency and make practical decisions regarding any changes that are needed. By using bytes sent data, developers get the opportunity to enrich the users' experience based on the effective and reliable data transmission, which makes WebRTC applications more stable and faster.

4.2.5. Results

Bandwidth Usage:

When examining the byte send rates, periodic bandwidth usage during the teleoperation sessions was observed. Average bandwidth was measured, and variations resulted from intensive activities like high-quality video streaming and operation of the robotic arm system. This way it was possible to determine a baseline of bytes sent in normal operation mode and quickly identify some anomalies and problems. The information also showed that when there was high traffic, such as during complex manipulations of the robotic arm or during the use of high-quality video stream, the bandwidth consumption grew considerably.

System Responsiveness:

To assess the system's reactivity, the duration of time it took for the Oculus Quest 2 controller inputs to translate into the robotic arm and TurtleBot movements was calculated. The average latency was kept low and within the acceptable range leaving no compromise on the overall Telepresence experience. Nevertheless, some situations where latency increased slightly appeared during the experiment, which could be explained by temporary network issues and delays in calculations. These results were further supported by the bytes sent data, where the amount of data transmitted also increased proportionally with high latency. This correlation raised possibilities indicating that enhancing data transfer could enhance system performance in a straight manner.

Data Transmission Efficiency:

The effectiveness of the data transfer was evaluated by analysing the relationship between the amount of data transmitted and the work of the robot. Higher data transmission rate was found linked with increased accuracy and smooth operations of both the robotic arm and vehicle movements. This highlighted how crucial it is to keep data rate and control precision in the ideal range. This investigation relied heavily on the bytes sent measure, which shows exactly how data volume affects performance. We can create a telepresence system that is more responsive and efficient by streamlining the data packets and cutting down on pointless transfers.

4.3. Video Stream Analysis:

The video stream in our system runs at 500 kbps bitrate, which is good for normal resolution video in terms of speed and quality. We looked at the bytes received value, which came to 2.10M (2,100,000 bytes) over a period of 5 seconds, in order to further assess the system's performance.

4.3.1. Calculation of Data Rate:

To determine the data rate, we performed the following calculations:

Total bytes received: 2,100,000 bytes

Duration: 5 seconds

Data rate in bytes per second:

Data rate = 2,100,000 bytes / 5 seconds = 420,000 bytes / seconds

Converting bytes per second to bits per second (since 1 byte = 8 bits):

Data rate = 420,000 bytes / second * 8 = 3,360,000 bits / second = 3360 kbps

4.3.2. Interpretation of Data Rate for Video Streaming:

We compare this data rate with typical WebRTC usage scenarios to determine its appropriateness:

1. Low resolution video:

Usually ranges from 200-500 kbps. A data rate of 3360 kbps is much higher than this range.

2. Standard resolution video:

Typically ranges from 500-1,500 kbps. The data rate of 3360 kbps exceeds this range.

3. High-definition video:

Often ranges from 1,500-5,000 kbps. The data rate of 3360 kbps falls within this range and is typical for HD video.

4.3.3. Conclusion:

When the system is streaming HD video the calculated data rate of 3360 kbps is reasonable and indicates that the connection is performing adequately based on the context provided. For instance, if the data is just going to be low-resolution video, then a data rate of 3360 kbps is unnecessary. This could imply that the bandwidth was utilized intensively either due to poor performance or overly high configurations.

Based on the information that the received data in 5 seconds is:

For HD video streams, the value is good and reflects a healthy, high-quality connection.

For low-resolution video or audio-only streams, the value would be bad as it implies excessive data utilization.

In our particular case where, only video is received and we are assuming high definition, the metric looks decent and indicates that the connection is running smoothly.

4.4. Current Round-Trip Time:

The Current Round Trip Time metric in WebRTC offers important details about the network delay that exists in the connection between the local peer and the remote peer. This is defined as the time taken for a data packet to be transmitted from the source to the destination and back. Current Round Trip Time is the time, in seconds, taken by a signal from the sender to the recipient and back to the sender. This metric is important for evaluating the performance of the network and especially the delay in real time communication.

4.4.1. Relevance to Video Sending and Receiving:

Video Sending:

Thus, the low round-trip time is important for achieving adequate smoothness and interactivity of live video streams. This can result to high latency, which means that the communication is not real-time and this may cause noticeable delays.

Video Receiving:

Likewise, on the receiver side, a low round-trip time means the video stream is delivered on time and with low delay, in order to avoid issues with the synchronization of audio and video streams.

How It's Measured:

Current Round Trip Time is usually determined by using RTCP (RTP Control Protocol) reports. RTCP packets are transported along with RTP (Real-Time Protocol) packets and are used to report information about the quality of the communication such as the latency measurements.

4.4.2. Interpreting Current Round Trip Time Value:

In our case, the Current Round Trip Time is 0.01 seconds, which translates to 10 milliseconds (msec).

Is 0.01 Seconds (10 ms) Good or Bad?

Excellent Performance:

0-20 ms: An RTT within this range is considered as the best and depicts a very good internet connection. These levels of latency are often seen only in environments that have been highly optimized and contain little network complexity such as a city or region.

Good Performance:

20-50 ms: This range is still very good and generally acceptable by the end users. It shows a strong signal that should be suitable for clear and smooth video calls.

Moderate Performance:

50-100 ms: A latency in this range is still tolerable, though it may become slightly noticeable in applications that would require a lot of interactivities such as video conferencing and gaming.

Poor Performance:

100 ms and above: Higher latencies result in perceivable delays and reduced quality of service, especially in mission-critical applications.

4.4.3. Practical Implications

User Experience:

The time of 10 ms is very good and should allow for very smooth and responsive video communication with little delay.

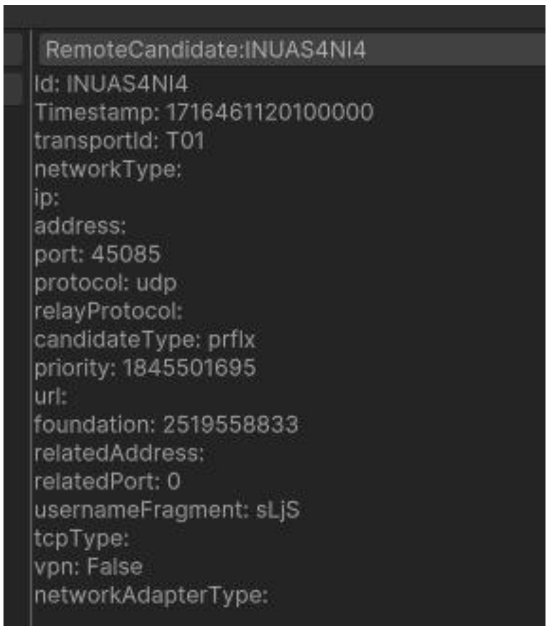
Network Quality:

This low latency indicates that the network path between the sender and receiver is highly efficient, likely involving few intermediate hops and possibly benefiting from high-speed, low-latency infrastructure.

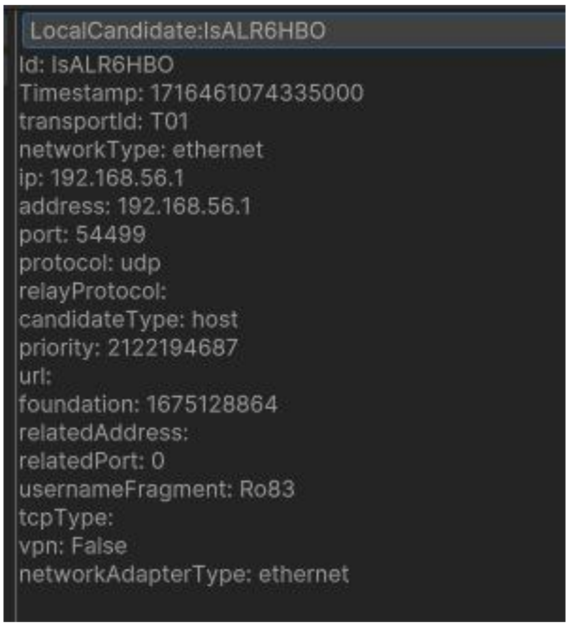
4.4.4. Conclusions:

This is a great result showing that the round-trip latency is quite minimal, especially considering our current round-trip time of 0.01 seconds (10 ms). For real-time video transmission, this performance level is perfect since it guarantees a low latency and excellent user experience. We may anticipate fluid video playback, fast response times, and overall better performance for both sending and receiving video streams with such low latency. In conclusion, a Current Round Trip Time of 10 ms is an ideal network state for real-time video communication and is a highly desired measure. It means that there is very little network latency, which adds to the high level of user experience that our WebRTC application offers.

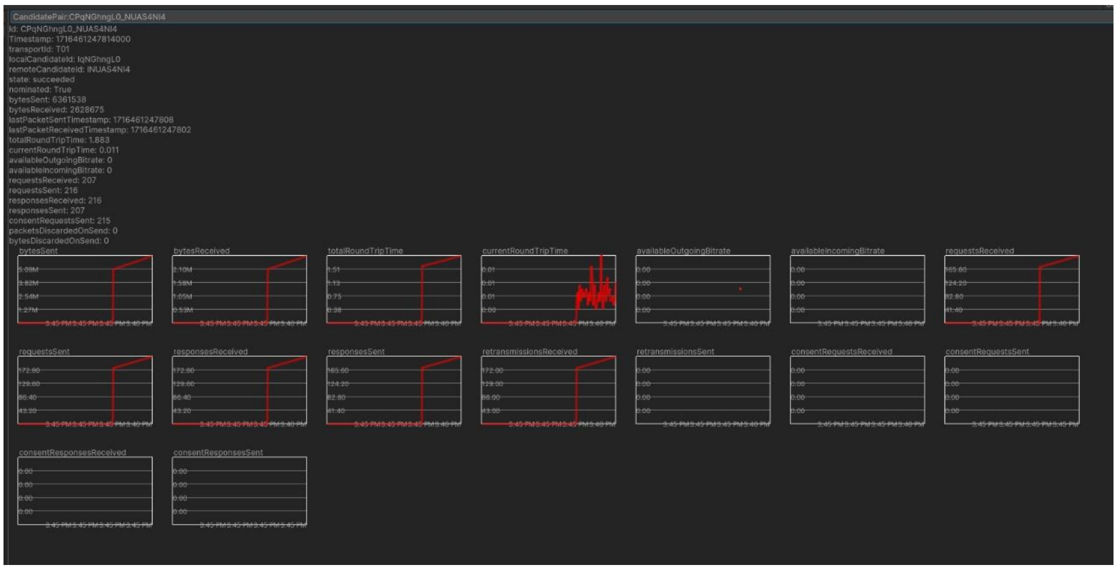
4.5. Graphical Analysis:



```
RemoteCandidate:INUAS4NI4
Id: INUAS4NI4
Timestamp: 1716461120100000
transportId: T01
networkType:
ip:
address:
port: 45085
protocol: udp
relayProtocol:
candidateType: prflx
priority: 1845501695
url:
foundation: 2519558833
relatedAddress:
relatedPort: 0
usernameFragment: sLjS
tcpType:
vpn: False
networkAdapterType:
```



```
LocalCandidate:IsALR6HBO
Id: IsALR6HBO
Timestamp: 1716461074335000
transportId: T01
networkType: ethernet
ip: 192.168.56.1
address: 192.168.56.1
port: 54499
protocol: udp
relayProtocol:
candidateType: host
priority: 2122194687
url:
foundation: 1675128864
relatedAddress:
relatedPort: 0
usernameFragment: Ro83
tcpType:
vpn: False
networkAdapterType: ethernet
```



```
CandidatePair:CPqNGhngL0_NUAS4NI4
Id: CPqNGhngL0_NUAS4NI4
Timestamp: 1716461247814000
transportId: T01
localCandidateId: IqNGhngL0
remoteCandidateId: INUAS4NI4
state: succeeded
nominated: True
bytesSent: 6361538
bytesReceived: 2628675
lastPacketSentTimestamp: 1716461247808
lastPacketReceivedTimestamp: 1716461247802
totalRoundTripTime: 1.883
currentRoundTripTime: 0.011
availableOutgoingBitrate: 0
availableIncomingBitrate: 0
requestsReceived: 207
requestsSent: 216
responsesReceived: 216
responsesSent: 207
consentRequestsSent: 215
packetsDiscardedOnSend: 0
bytesDiscardedOnSend: 0
```

Figure 35. Graphical Analysis (Latency)

Chapter 5 -CONCLUSION

5.1. Summary of achievements:

The following are the achievements of the research:

- This project is funded by NUST worth PKR 0.995 million. Their support highlights the recognition of its importance and potential impact.
- This project also qualified for the second stage of the Finding Innovative & Creative Solutions (FICS) Competition.
- This project is the winner of Computer Project Exhibition Competition (COMPPEC ‘ 24) AR/VR Category with a winning prize worth PKR 40,000/-

5.2. Virtual Telepresence Robot:

In this final chapter, we reflect on the journey of developing our Virtual Telepresence Robot, from the intricate hardware setup to the seamless integration of software components, and draw conclusions regarding its significance, achievements, and future implications.

Robotic arm integration and hardware design:

As the foundation for our work, the TurtleBot 3 Waffle robot has been chosen and a robotic arm has been mounted on top of it, which allows for working at a distance and performing precise manipulations. Hardware selections were made with much thought given to compatibility and recognition of fully functional parts. They also made the design of the arm to ensure that it is both stable and precise in a way that allows the arm to effectively perform several tasks on the vehicle.

Software Architecture and Integration:

The soul of the system is software architecture, designed in its finest detail with the goal of providing smooth communication and control. The expanded use of ROS 2. The score of 0 on the Raspberry Pi 4 was complemented by the WebRTC video stream that allowed for the efficient data transfer and immediate interaction. Custom ROS nodes and an application in Unity enabled the control and interaction between the Oculus Quest 2 VR headset and the robotic arm to make it possible and enable the user of the robotic arm to have an immersive experience.

Video Streaming and Telepresence Capabilities:

One of the significant components of our project is the use of WebRTC in the context of the video streaming, which will provide the main channel of the video streaming with a high quality and low latency of the video stream. This technology integrated with the Oculus Quest 2 VR headset, helps the user virtually explore distant environments as though they are on site. The issues of telepresence have numerous applications in different fields such as telemedicine, remote education, and teleoperation of industrial environments, allowing humans to control environments that are geographically far away.

Performance Analysis and Optimization:

While developing this solution, performance analysis was used in determining the proper design to ensure the system was highly efficient and reliable. Measuring bytes sent, bytes received, and current round-trip time proved to be very informative about the usage of the networks, the latency that was needed to get the data, and the data transmission rates that could be expected. We used this iterative optimization process of refining the design until the Virtual Telepresence Robot provided responsive and seamless performance to the user, regardless of the network conditions.

5.3. Significance

The development as well as integration of the Virtual Telepresence Robot makes a perfect model setting a mark of achievement in the field of telepresence. Our system brings in the idea of creating a connection between physical and remote environment thus more possibilities of remote interaction, exploring, and working. Possible uses can include remote collaboration and telepresence in hostile conditions and remote education and entertainment in an immersive setting.

5.4. Future Implications and Recommendations:

There are a lot of opportunities for the future and for improving our Virtual Telepresence Robot as well. Possible work in this direction can include explaining how such a design can be optimised for stability and handling, and the use of newer sensors and actuators that allow the combination to perform even better in perceiving the environment and interacting with it. Furthermore, the software structure may need to be improved as to scalability and modularity for further integration of the platform with other robotics hardware and telepresence devices.

5.5. Conclusion

Al in all, expanding the idea of our Virtual Telepresence Robot is a noteworthy contribution to the progress of telepresence technology. By merging the notion of computer hardware and software, as well as communication systems, we have developed a complex but highly flexible system that allows for fully immersed remote interaction and manipulation. This tutorial's success confirms the idea that telepresence technology can efficiently help overcome distance, bring together like-minded individuals, and enhance human potential beyond limits of the physical world. As far as the future is concerned, we must continue our quest for innovation and advancement in this rapidly expanding subject area.

APPENDIX A

```
Simple Data Sender.cs:
using UnityEngine;
using WebSocketSharp;
using Unity.WebRTC;
using System.Collections;
using System;
public class SimpleDataChannelSender : MonoBehaviour
{
    private DisplayInputData inputData;
    private RTCPeerConnection connection;
    private RTCDataChannel dataChannel;
    private RTCDataChannel dataChannel1;
    private WebSocket ws;
    private string clientId;
    private float X, Z , gripo;
    private bool hasReceivedAnswer = false;
    private SessionDescription receivedAnswerSessionDescTemp;
    private void Start()
    {
        inputData = GetComponent<DisplayInputData>();
        InitClient("65.2.148.175", 5050);
    }
    private void Update()
    {
        if (hasReceivedAnswer)
        {
            hasReceivedAnswer = !hasReceivedAnswer;
            StartCoroutine(SetRemoteDesc());
        }
        if (inputData.trig == true)
        {
            if (inputData.grip)
            {
                gripo = 1;
            }
            else
            {
                gripo = 0;
            }
            float[] floatArray = { inputData.X, inputData.Y, inputData.Z,
gripo};
            string result = string.Join(", ", floatArray);
            dataChannel.Send("Controller_Data:" + result);
        }
        if (true)
        {
            const float linearScale = 0.26f;
            const float angularScale = 1.82f;
            X = (inputData.joyY + 1) * linearScale -
linearScale;
            Z = -((inputData.joyX + 1) * angularScale - angularScale);
            if ((Z < 0.9 && Z > 0) || (Z > -0.9 && Z < 0))
            {
                Z = 0;
            }
            float[] floatArray = { X, Z };
            string result1 = string.Join(", ", floatArray);
            dataChannel.Send("cmd_val:" + result1);
        }
    }
    private void OnDestroy()
    {
        dataChannel.Close();
        connection.Close();
    }
    public void InitClient(string serverIp, int serverPort)
    {
        int port = serverPort == 0 ? 5050 : serverPort;
        clientId = gameObject.name;
        ws = new
WebSocket($"ws://{serverIp}:{port}/{nameof(SimpleDataChannelService)}");
        ws.OnMessage +=  (sender, e) => {
            var requestArray = e.Data.Split("!");
            var requestType = requestArray[0];
            var requestData = requestArray[1];
            switch (requestType)
            {
                case "ANSWER" :
                    Debug.Log(clientId + " - Got ANSWER from Maximus: " +
requestData);
                    receivedAnswerSessionDescTemp =
SessionDescription.FromJSON(requestData);
                    hasReceivedAnswer = true;
                    break;
                case "CANDIDATE":
                    Debug.Log(clientId + " - Got CANDIDATE from Maximux: "+
requestData);
                    // generate candidate data
                    var candidateInit = CandidateInit.FromJSON(requestData);
                    RTCIceCandidateInit init = new RTCIceCandidateInit();
                    init.sdpMid = candidateInit.SdpMid;
                    init.sdpMLineIndex = candidateInit.SdpMLineIndex;
                    init.candidate = candidateInit.Candidate;
                    RTCIceCandidate candidate = new RTCIceCandidate(init);
                    //add candidate to this connection
                    connection.AddIceCandidate(candidate);
                    break;
                default:
                    Debug.Log(clientId + " - Maximux says: " + e.Data);
                    break;
            }
        };
        ws.Connect();
        // Configure RTCConfiguration with ICE Servers (STUN and TURN)
        var iceServers = new RTCIceServer[]
        {
            new RTCIceServer { urls = new string[] {
"stun:stun.relay.metered.ca:80" } }, // STUN server
            new RTCIceServer
            {
                urls = new string[] {"turn:relay1.expressturn.com:3478"},
                username = "efJ7V1YAY7A1TPL1Y4",
                credential = "PjiDGUmJjPNpEqdk"
            },
        };
        var configuration = new RTCConfiguration { iceServers = iceServers
};
        connection = new RTCPeerConnection();
        connection.SetConfiguration(ref configuration);
        connection.OnIceCandidate = candidate => {
            Debug.Log("Here");
            var candidateInit = new CandidateInit()
            {
                SdpMid = candidate.SdpMid,
                SdpMLineIndex = candidate.SdpMLineIndex ?? 0,
                Candidate = candidate.Candidate
            };
            ws.Send("CANDIDATE! " + candidateInit.ConvertToJson());
            Debug.Log(" Receiver sent ICE-CANDIDATES : " +
candidateInit.ConvertToJson());
        };
        connection.OnIceConnectionChange = state => {
            Debug.Log(state);
        };
        dataChannel = connection.CreateDataChannel("sendChannel");
        dataChannel.OnOpen = () =>{
            Debug.Log("Sender opened channel");
        };
        dataChannel.OnClose = () => {
            Debug.Log("Sender closed channel");
        };
        dataChannel1 = connection.CreateDataChannel("sendChannel");
        dataChannel1.OnOpen = () =>{
            Debug.Log("Sender opened channel");
        };
        dataChannel1.OnClose = () => {
            Debug.Log("Sender closed channel");
        };
        connection.OnNegotiationNeeded = () => {
            StartCoroutine(CreateOffer());
        };
    }
    p...
<...etc...>
```

APPENDIX B

```
using UnityEngine;
using WebSocketSharp;
using Unity.WebRTC;
using System.Collections;
using UnityEngine.UI;
using System;
public class SimpleMediaStreamReceiver : MonoBehaviour
{
    [SerializeField] private Material skyboxMaterial;
    private RTCPeerConnection connection;
    private WebSocket ws;
    private string clientId;
    private bool hasReceivedOffer = false;
    private SessionDescription receivedOfferSessionDescTemp;
    private string senderIp;
    private int senderPort;
    private void Start()
    {
        InitClient("65.2.148.175", 8080);
    }
    public void InitClient(string serverIp, int serverPort)
    {
        senderPort = serverPort == 0 ? 5050 : serverPort;
        senderIp = serverIp;
        clientId = gameObject.name;
        Debug.Log($"SimpleDataChannelService:
{nameof(SimpleDataChannelService)}");
```

```
        ws = new
WebSocket($"ws://{senderIp}:{senderPort}/{nameof(SimpleDataChannelService)}"
);
        ws.OnMessage += (sender, e) => {
            var signalingMessage = new SignalingMessage(e.Data);
            switch (signalingMessage.Type)
            {
                case SignalingMessageType.OFFER:
                    // Debug.Log(clientId + " - Got OFFER from Sender: " +
signalingMessage.Message);
                    Debug.Log(" Got OFFER from Sender ");
                    receivedOfferSessionDescTemp =
SessionDescription.FromJSON(signalingMessage.Message);
                    hasReceivedOffer = true;
                    break;
                case SignalingMessageType.CANDIDATE:
                    Debug.Log(" Got ICE-CANDIDATE from Sender ");
                    var candidateInit =
CandidateInit.FromJSON(signalingMessage.Message);
                    Debug.Log("candidateInit.candidate : " +
candidateInit.candidate);
                    RTCIceCandidateInit init = new RTCIceCandidateInit();
                    init.sdpMid = candidateInit.SdpMid;
                    init.sdpMLineIndex = candidateInit.SdpMLineIndex;
                    init.candidate = candidateInit.candidate;
                    RTCIceCandidate candidate = new RTCIceCandidate(init);
                    connection.AddIceCandidate(candidate);
                    break;
                default:
                    Debug.Log(clientId + " - Sender says: " + e.Data);
                    break;
            }
        };
        ws.Connect();
        Debug.Log("Receiver Connected to Websocket Server");
        // Configure RTCConfiguration with ICE Servers (STUN and TURN)
        var iceServers = new RTCIceServer[]
        {
            new RTCIceServer { urls = new string[] {
"stun:stun.relay.metered.ca:80" } }, // STUN server
            new RTCIceServer
            {
                urls = new string[] {"turn:relay1.expressturn.com:3478"},
                username = "efJ7V1YAY7A1TPL1Y4",
                credential = "PjiDGUmJjPNpEqdk"
            },
        };
        var configuration = new RTCConfiguration { iceServers = iceServers
};
        connection = new RTCPeerConnection();
        connection.SetConfiguration(ref configuration);
        connection.OnIceCandidate = candidate => {
            Debug.Log("Here");
            var candidateInit = new CandidateInit()
            {
                SdpMid = candidate.SdpMid,
                SdpMLineIndex = candidate.SdpMLineIndex ?? 0,
                Candidate = candidate.Candidate
            };
            // ws.OnOpen += (sender, e) => {
            //     ws.Send("CANDIDATE! " + candidateInit.ConvertToJson());
            //     Debug.Log(" Receiver sent ICE-CANDIDATES");
            // };
            ws.Send("CANDIDATE! " + candidateInit.ConvertToJson());
            Debug.Log(" Receiver sent ICE-CANDIDATES : " +
candidateInit.ConvertToJson());
        };
        connection.OnIceConnectionChange = state => {
            Debug.Log(state);
        };
        connection.OnTrack = e => {
            if (e.Track is VideoStreamTrack video)
            {
                // Set the received video track to the skybox material's
texture
                video.OnVideoReceived += texture =>
                {
                    skyboxMaterial.mainTexture = texture;
                    RenderSettings.skybox = skyboxMaterial;
                };
            }
        };
        StartCoroutine(WebRTC.Update());
    }
    private void Update()
    {
        if (hasReceivedOffer)
        {
            Debug.Log("Receiver received OFFER");
            hasReceivedOffer = !hasReceivedOffer;
            StartCoroutine(CreateAnswer());
        }
    }
    private void OnDestroy()
    {
        connection.Close();
        ws.Close();
    }
    private IEnumerator CreateAnswer()
    {
        Debug.Log("Receiver creating Answer");
        RTCSessionDescription offerSessionDesc = new
RTCSessionDescription();
        offerSessionDesc.type = RTCSdpType.Offer;
        offerSessionDesc.sdp = receivedOfferSessionDescTemp.Sdp;
        var remoteDescOp = connection.SetRemoteDescription(ref
offerSessionDesc);
        yield return remoteDescOp;
        var answer = connection.CreateAnswer();
        yield return answer;
        var answerDesc = answer.Desc;
        var localDescOp = connection.SetLocalDescription(ref answerDesc);
        yield return localDescOp;
        //send desc to server for sender connection
        var answerSessionDesc = new SessionDescription()
        {
            SessionType = answerDesc.type.ToString(),
            Sdp = answerDesc.sdp
        };
        ws.Send("ANSWER!" + answerSessionDesc.ConvertToJson());
        Debug.Log("Receiver Sent ANSWER");
    }
}
```

APPENDIX C

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Quaternion
import serial
import time
class arduinoController:
    # Class attributes
    startMarker = 60  # unicode character '<'
    endMarker = 62  # unicode character '>'
    servoTime1 = 10  # default times of servo movement
    servoTime2 = 10
    servoTime3 = 10
    servoTimeEE = 10
    msg = "<BUZZ,90,90,90,90,10,10,10,10>"  # default message
    # Initializer / Instance attributes
    def _init_(self, port="COM18"):
        self.port = port
    # Include function here connect and pass joint angle
    def _waitForArduino_(self):
        """
        --Description--
        This function waits until the Arduino sends 'Arduino Ready' - allows
time for Arduino reset.
        It also ensures that any bytes left over from a previous message are
discarded
        """
        msg = ""
        while msg.find("Arduino is ready") == -1:
            # note changed to in_waiting, get the number of bytes in the
input buffer
            while self.serialPort.inWaiting() == 0:
                pass
            msg = self.recvFromArduino()
            print(msg)
            print("")
    # Instance methods
    def openSerialPort(self, baudRate=115200, port=None):
        """
        --Description--
        Connects to a serial port using pySerial.
        This function exists to avoid confusion in naming of serial port!
        --Parameters--
        @myString -> the string to be sent
        --Returns--
        Function doesn't return a value
        """
        if port is None:
            port = self.port
        self.serialPort = serial.Serial(port=self.port, baudrate=baudRate)
        print("Serial port " + port + " opened  Baudrate " + str(baudRate))
        # wait for arduino to be ready
        self._waitForArduino_()
    def closeSerialPort(self, port=None):
        """
        --Description--
        Close serial port using pySerial.
        This function exists to avoid confusion in naming of serial port!
        --Parameters--
        @myString -> the string to be sent
        --Returns--
        Function doesn't return a value
        """
        if port is None:
            port = self.port
        self.serialPort.close
        print("Serial port " + port + " closed")
    def sendToArduino(self, myString=None):
        """
        --Description--
        Sends a string to the Arduino. String must be in the format
"<LED1,200,0.2>" [HOLD]
        --Parameters--
        @myString -> the string to be sent
        --Returns--
        Function doesn't return a value
        """
        if myString is None:
            myString = self.msg
        self.serialPort.write(myString.encode('utf-8'))  # encode as unicode
    def recvFromArduino(self):
        """
        --Description--
        Recieve a message from the Arduino. The message is interpreted as a
string being
        between the start character '<' and end character '>'
        --Parameters--
        None
        --Returns--
        msg -> The received message as a string
        """
        msg = ""
        x = "z"  # any value that is not an end or startMarker
        byteCount = -1  # to allow for the fact that the last increment will
be one too many
        # wait for the start character
        while ord(x) != self.startMarker:  # ord returns the unicode number
for the character
            x = self.serialPort.read()
        # save data until the end marker is found
        while ord(x) != self.endMarker:
            if ord(x) != self.startMarker:
                msg = msg + x.decode("utf-8")  # decode from unicode
                byteCount += 1
            x = self.serialPort.read()
        return(msg)
    def composeMessage(self, servoAngle_q1, servoAngle_q2, servoAngle_q3,
servoAngle_EE=90, instruction="BUZZ", **kwargs):
        """
        --Description--
        Function composes a message (instruction) to send to the Arduino,
this tells it where to position
        the connected servo motors. As input we take the servo angles and
optionally the movement durations
        for each angle
        This function can take direct output (for joint angles q1, q2, q3)
from the function 'map_kinematicsToServoAngles()'
        --Parameters--
        @servoAngle_q1 -> the servo angle for servo #1, corresponding to q1
        @servoAngle_q2 -> the servo angle for servo #2, corresponding to q2
        @servoAngle_q3 -> the servo angle for servo #3, corresponding to q3
        @servoAngle_EE -> the servo angle for servo #0, corresponding to the
end effector
        @instruction -> the text instruction to send to the Arduino (BUZZ->
sounds buzzer), (LED -> flashes LED)
        --Optional **kwargs parameters--
        @servoTime1 -> the time of the servo1 movement (if change in value
is input for servo1Angle)
        @servoTime2 -> the time of the servo2 movement (if change in value
is input for servo2Angle)
        @servoTime3 -> the time of the servo3 movement (if change in value
is input for servo3Angle)
        @servoTimeEE -> the time of the servoEE movement (if change in value
is input for servoEEAngle)
        --Returns--
        Function doesn't return a value
        """
        # Use **kwargs if provided, otherwise use current values
        servoTime1 = str(int(kwargs.get('servoTime1', self.servoTime1)))
        servoTime2 = str(int(kwargs.get('servoTime2', self.servoTime2)))
        servoTime3 = str(int(kwargs.get('servoTime3', self.servoTime3)))
        servoTimeEE = str(int(kwargs.get('servoTimeEE', self.servoTimeEE)))
        # Compose message
        message = "<" + ",".join([str(int(servoAngle_q1)),
str(int(servoAngle_q2)), str(int(servoAngle_q3)), str(int(servoAngle_EE))])
        message = message + "," + \
            ",".join([servoTime1, servoTime2, servoTime3, servoTimeEE]) +
">"
        self.msg = message
        return message
    def communicate(self, data, delay_between_commands=5):
        """
        --Description--
        Function runs a test to check communications with Arduino!
        --Parameters--
        data-> a command string (returned by the composeMessage method) or a
list of strings, which will be communicated to the arduino
        delay_between_commands -> the time delay applied between a list of
commands
        --Returns--
        Function doesn't return a value
        """
        # convert data to list if it isn't already a list
        if not isinstance(data, list):
            data = [data]
        # Declare variables
        numLoops = len(data)
        waitingForReply = False
        n = 0
        while n < numLoops:
            data_str = data[n]
            if waitingForReply == False:
                self.sendToArduino(data_str)
                print("Sent from PC -- LOOP NUM " +
                      str(n) + " TEST STR " + data_str)
                waitingForReply = True
            if waitingForReply == True:
                while self.serialPort.inWaiting() == 0:
                    pass
                dataRecvd = self.recvFromArduino()
                print("Reply Received  " + dataRecvd)
                n += 1
                waitingForReply = False
                print("===========")
            time.sleep(delay_between_commands)
# Insert your Arduino serial port here
myArduino = arduinoController(port="/dev/ttyUSB0")
myArduino.openSerialPort()
# Set the increment value for servo movement
servo_increment = 5
class rpmSubscriber(Node):
    def _init_(self):
        super()._init_("arm_sub_node")
        self.sub = self.create_subscription(Quaternion, "controller_data",
self.subsciber_callback, 10) # Subscribes to controller data
        # Define initial angles for the servos
        self.q1 = 90
        self.q2 = 90
        self.q3 = 90
        self.q0 = 90
        # Create some test data
        self.testData=[]
        self.testData = myArduino.composeMessage(
            servoAngle_q1=self.q1,
            servoAngle_q2=self.q2,
            servoAngle_q3=self.q3,
            servoAngle_EE=self.q0
        )
    def subsciber_callback(self, msg):
        self.x = msg.x
        self.y = msg.y
        self.z = msg.z
        self.w = msg.w
        self.get_logger().info(f"Received Array: {msg}")
        print(f"Received Array: {msg}")
        x=self.x
        y=self.y
        z=self.z
        w = self.w
        if x >=-20 and x <= 20:
            self.q1 = 4.25*x + 90 # Maping q1 motor according to x
coordinate value
        if z >=-20 and z <= 20:
            self.q2 = 0.05*z*z + 2.5*z + 90 # Maping q2 motor according to z
coordinate value
        if y >=-20 and y <= 20:
            self.q3 = 0.04375*y*y + 2.125*y + 90 # Maping q3 motor according
to y coordinate value
            if self.q3 > 155 :
                self.q3-=100
        if w == 1:  # Gripper Movement
            self.q0 = 50
        else:
            self.q0 = 150
        # Update the test data with the new servo angles
        testData = myArduino.composeMessage(
            servoAngle_q1=self.q1,
            servoAngle_q2=self.q2,
            servoAngle_q3=self.q3,
            servoAngle_EE=self.q0
        )
            # Communicate with the Arduino, sending the updated test data
        myArduino.communicate(data=testData, delay_between_commands=0)
def main():
    rclpy.init()
    my_sub = rpmSubscriber()
    print("Waiting the data to be publisher over topic ....")
    try:
        rclpy.spin(my_sub)
    except KeyboardInterrupt:
        my_sub.destroy_node()
        rclpy.shutdown()
if _name== "main_":
    main()
```

APPENDIX D

```
let connection;
let videoStream;
let videoStreamTrack;
let ws;
let clientId;
let hasReceivedAnswer = false;
let receivedAnswerSessionDescTemp;
console.log("Hello from Roxy");
var A;
var B,X = 90,Y = 90,Z = 90 , W = 0;
function Start() {
    InitClient("65.2.148.175", 8080); // Makes a new client that connnects
to signalling server at given address
}
// Function to initialize a WebSocket client and set up WebRTC ICE candidate
handling
function InitClient(serverIp, serverPort) {
    // Use the specified server port, defaulting to 5050 if the provided
port is 0
    const port = serverPort === 0 ? 8080 : serverPort;
    clientId = "RoxySender"; // Set a unique client identifier
    // Create a new WebSocket connection to the specified server IP and port
of signalling server , in our case signalinng server is deployed on AWS EC2
instance.
    // if you want to try this code You can upload signaling_server.py code
to your own ec2 instance and change public ip addresses in add code.
    ws = new
WebSocket(ws://${serverIp}:${port}/${"SimpleDataChannelService"});
    // Event handler for receiving messages from the WebSocket server
    ws.onmessage = function(event) {
        const originalMessage = event.data; // Get the original message data
        const splitMessage = originalMessage.split("!"); // Split the
message by '!'
        const Type = splitMessage[0]; // Get the message type (e.g.,
'ANSWER', 'CANDIDATE')
        const data = splitMessage[1]; // Get the message data
        const signalingMessage = JSON.parse(data); // Parse the JSON data
        // Handle the message based on its type
        switch (Type) {
            case 'ANSWER':
                console.log('Got ANSWER from Receiver: ' +
signalingMessage.Sdp);
                receivedAnswerSessionDescTemp = signalingMessage.Sdp;
                hasReceivedAnswer = true; // Set the flag indicating an
answer is received
                setRemoteDesc(); // Set the remote description for the
WebRTC connection
                hasReceivedAnswer = false; // Reset the flag
                break;
            case 'CANDIDATE':
                console.log("Got ICE-CANDIDATE from Receiver :" +
JSON.stringify(signalingMessage));
                var candidateInit = signalingMessage; // Store the received
ICE candidate info
                console.log("candidateInit.Candidate : " +
candidateInit.Candidate);
                var init = {
                    sdpMid: candidateInit.SdpMid, // Media stream
identification
                    sdpMLineIndex: candidateInit.SdpMLineIndex, // Media
stream line index
                    candidate: candidateInit.Candidate // The actual
candidate string
                };
                var candidate = new RTCIceCandidate(init); // Create a new
ICE candidate
                connection.addIceCandidate(candidate); // Add the candidate
to the WebRTC connection
                break;
            default:
                console.log(clientId + ' - Receiver says: ' + event.data);
// Log any other messages
                break;
        }
    };
    ws.onopen = function() {
        console.log('Video Sender Connected to WebSocket Server');
    };
    const iceServers = [
        { urls: 'stun:stun.relay.metered.ca:80' }, // STUN server
        {
            urls: "turn:15.206.116.209:3478",
            username: "roxy_video",
            credential: "roxy_video",
        }, // Deplayed TURN Server to AWS EC2 instance using coturn, In
orderr to make your own turn server , edit security inbound traffic and
```

```
           // open 3478 port for UDP and TCP and All UDp Traffic , uncomment
following code from  /etc/turnserver.config . (You can use Sudo nano
/etc/turnserver.config)
    ];
    const configuration = { iceServers };
    connection = new RTCPeerConnection(configuration);
    connection.onicecandidate = function(candidate) {
        var candidateInit = {
            SdpMid: candidate.sdpMid,
            SdpMLineIndex: candidate.sdpMLineIndex,
            Candidate: candidate.candidate
        };
        ws.send("CANDIDATE!" + JSON.stringify(candidate.candidate));
        console.log("ICE-Candidate Sent to Receiver : " +
JSON.stringify(candidate.candidate));
    };
    connection.oniceconnectionstatechange = function(event) {
        console.log(connection.iceConnectionState);
        document.getElementById("status_webrtc").textContent =
connection.iceConnectionState;
    };
    connection.onnegotiationneeded = async function() {
        await createOffer();
    };
    InitializeVideoStream();
}
async function createOffer() {
    const offer = await connection.createOffer();
    await connection.setLocalDescription(offer);
    const offerSessionDesc = {
        type: offer.type,
        sdp: offer.sdp
    };
    ws.send("OFFER!" + JSON.stringify({ SessionType: 'Offer', Sdp:
offerSessionDesc.sdp }));
    console.log('Sender sent OFFER!');
}
async function setRemoteDesc() {
    const answerSessionDesc = {
        type: 'answer',
        sdp: receivedAnswerSessionDescTemp
    };
    await connection.setRemoteDescription(answerSessionDesc);
}
async function InitializeVideoStream() {
    // Check if the connection object is defined
    if (connection) {
        // Get user media stream
        try {
            const stream = await navigator.mediaDevices.getUserMedia({
video: true });
            videoStreamTrack = stream.getVideoTracks()[0];
            videoStream = new MediaStream([videoStreamTrack]);
            connection.addTrack(videoStreamTrack, videoStream);
        } catch (error) {
            console.error("Error accessing media devices:", error);
        }
    } else {
        console.error("Connection object is not initialized.");
    }
}
// uncomment this code to set camera resolution. You can also make this bot
streeam 360 camera by changing resolution.
// Add this function to your sender.js file
function startConnection() {
    // Call the Start function to initiate the WebSocket connection
    Start();
}
// Add event listener to execute code when the DOM is fully loaded
document.addEventListener("DOMContentLoaded", function(event) {
    // Call InitializeVideoStream when the DOM is loaded to start capturing
the video stream
    InitializeVideoStream();
});
if (hasReceivedAnswer) {
    console.log("Sender received Answer");
    hasReceivedAnswer = false;
    setRemoteDesc();
}
// DataChannel Receiver
// this code is same raplica of above code but this time its receiver and
unity side is sender ,
let connection1;
let dataChannel;
let ws1;
let clientId1;
let hasReceivedOffer1 = false;
let receivedOfferSessionDescTemp1;
let cmd_vel_data;
let controller_data;
let message;
function Start1() {
    InitClient1("65.2.148.175", 5050);
}
```

```
function Update1() {
    if (hasReceivedOffer1) {
        hasReceivedOffer1 = !hasReceivedOffer1;
        CreateAnswer1();
    }
}
function OnDestroy1() {
    dataChannel.close();
    connection1.close();
}
function InitClient1(serverIp, serverPort) {
    const port = serverPort === 0 ? 8080 : serverPort;
    clientId1 = "DataChannel-Receiver";
    ws1 = new
WebSocket(ws://${serverIp}:${port}/${"SimpleDataChannelService"});
    ws1.onmessage = function(event) {
        const originalMessage = event.data;
        const splitMessage = originalMessage.split("!");
        const Type = splitMessage[0];
        const data = splitMessage[1];
        const signalingMessage = JSON.parse(data);
        switch (Type) {
            case 'OFFER':
                console.log('Got OFFER from Sender: ' +
signalingMessage.Sdp);
                receivedOfferSessionDescTemp1 = signalingMessage.Sdp;
                hasReceivedOffer1 = true;
                CreateAnswer1();
                hasReceivedOffer1 = false;
                break;
            case 'CANDIDATE':
                console.log("Got ICE-CANDIDATE from Sender :" +
JSON.stringify(signalingMessage));
                var candidateInit = signalingMessage;
                console.log("candidateInit.Candidate : " +
candidateInit.Candidate);
                var init = {
                    sdpMid: candidateInit.SdpMid,
                    sdpMLineIndex: candidateInit.SdpMLineIndex,
                    candidate: candidateInit.Candidate
                };
                var candidate = new RTCIceCandidate(init);
                connection1.addIceCandidate(candidate);
                break;
            default:
                console.log(clientId1 + ' - Receiver says: ' + event.data);
                break;
        }
    };
    ws1.onopen = function() {
        console.log('Data Channel Script Connected to WebSocket Server');
    };
    const iceServers = [
        { urls: 'stun:stun.relay.metered.ca:80' }, // STUN server
        {
            urls: "turn:15.206.116.209:3478",
            username: "roxy_data",
            credential: "roxy_data",
        },
    ];
    const configuration = { iceServers };
    connection1 = new RTCPeerConnection(configuration);
    connection1.onicecandidate = function(candidate) {
        var candidateInit = {
            SdpMid: candidate.sdpMid,
            SdpMLineIndex: candidate.sdpMLineIndex,
            Candidate: candidate.candidate
        };
        ws1.send("CANDIDATE!" + JSON.stringify(candidate.candidate));
        console.log("ICE-Candidate Sent to Receiver : " +
JSON.stringify(candidate.candidate));
    };
    connection1.oniceconnectionstatechange = function(event) {
        console.log(connection1.iceConnectionState);
        if (connection1.iceConnectionState=="connected")
            {
                Start();
            }
        else if (connection1.iceConnectionState=="disconnected"){
            // location.reload();
            X = 90,Y = 90,Z = 90 , W = 0, A = 0, B = 0;
        }
    };
    connection1.ondatachannel = function(event) {
        dataChannel = event.channel;
        dataChannel.onmessage = function(event) {
            console.log("Receiver received: " + event.data);
            message = event.data;
            const parts = message.split(":");
            console.log("Label: " + parts[0]);
            if (parts.length === 2 && parts[0].trim() === "Controller_Data")
{
                // Extract the values part and split by comma
                const values = parts[1].trim().split(",");
                // Extract float values for x, y, and z
                if (values.length === 4) {
                    X = parseFloat(values[0].trim());
                    Y = parseFloat(values[1].trim());
                    Z = parseFloat(values[2].trim());
                    W = parseFloat(values[3].trim());
                    // Check if parsing was successful
                    if (!isNaN(X) && !isNaN(Y) && !isNaN(Z) && !isNaN(W) ) {
                        console.log("Parsed values controller_data:");
                        console.log("x:", X);
                        console.log("y:", Y);
                        console.log("z:", Z);
                        console.log("w:", W);
                    } else {
                        console.log("Invalid float values");
                    }
                } else {
                    console.log("Invalid format: Expected 4 values separated
by comma");
                }
            } else {
                console.log("Invalid message format");
            }
            if (parts[0] == "cmd_val"){
                // Extract the values part and split by comma
                const values = parts[1].trim().split(",");
                // Extract float values for x, y, and z
                if (values.length === 2) {
                    A = parseFloat(values[0].trim());
                    B = parseFloat(values[1].trim());
                    // Check if parsing was successful
                    console.log("Parsed values cmd_vel:");
                    console.log("x:", A);
                    console.log("z:", B);
                } else {
                    console.log("Invalid format: Expected 2 values separated
by comma");
                }
            } else {
                console.log("Invalid message format");
            }
        };
    };
}
async function CreateAnswer1() {
    console.log ("SDP: " + receivedOfferSessionDescTemp1)
    const offerSessionDesc = {
        type: "offer",
        sdp: receivedOfferSessionDescTemp1
    };
    await connection1.setRemoteDescription(offerSessionDesc);
    const answer = await connection1.createAnswer();
    await connection1.setLocalDescription(answer);
    const answerSessionDesc = {
        SessionType: answer.type,
        Sdp: answer.sdp
    };
    ws1.send("ANSWER!" + JSON.stringify(answerSessionDesc));
    console.log("Receiver Sent Answer to Sender : " + answerSessionDesc);
}
Start1();
// // script.js
// Import ROSLIBJS if you're using Node.js
// const ROSLIB = require('roslib');
// Connect to ROS Bridge
const ros = new ROSLIB.Ros();
ros.connect('ws://localhost:9090'); // Adjust the URL according to your ROS
Bridge server
// Wait for the connection to be established
ros.on('connection', function() {
    console.log('Connected to ROS Bridge');
    document.getElementById("status_local").textContent = "Connected";
});
// Wait for the connection to be closed
ros.on('close', function() {
    console.log('Connection to ROS Bridge closed');
    document.getElementById("status_local").textContent = "Closed";
});
// Wait for an error to occur
ros.on('error', function(error) {
    console.log('Error connecting to ROS Bridge:', error);
    document.getElementById("status_local").textContent = "Error";
});
// Define a function to publish the twist message
function publishTwist() {
    // Create a new Twist message every time the function is called
    const twist = new ROSLIB.Message({
        linear: {
            x: A,
            y: 0.0,
            z: 0.0
        },
        angular: {
            x: 0.0,
            y: 0.0,
            z: B
        }
    });
    // Publish the Twist message on the desired topic
    twistPublisher.publish(twist);
}
// Publish the twist message every 100 milliseconds
const twistPublisher = new ROSLIB.Topic({
    ros: ros,
    name: '/cmd_vel',
    messageType: 'geometry_msgs/Twist'
});
setInterval(publishTwist, 100);
/___________/
// Define a function to publish the control message
function publishControl() {
    // Create a new Quaternion message every time the function is called
    const control = new ROSLIB.Message({
        x: X,
        y: Y,
        z: Z,
        w: W
    });
    // Publish the Quaternion message on the desired topic
    controlPublisher.publish(control);
}
// Publish the control message every 100 milliseconds
const controlPublisher = new ROSLIB.Topic({
    ros: ros,
    name: '/controller_data',
    messageType: 'geometry_msgs/Quaternion'
});
setInterval(publishControl, 100);
```

APPENDIX E

```
#!/bin/bash
# Export necessary environment variables (optional)
export LDS_MODEL=LDS-02
export TURTLEBOT3_MODEL=waffle_pi
# Launch turtlebot3_bringup package in a new terminal
gnome-terminal -- bash -c "ros2 launch turtlebot3_bringup robot.launch.py;
exec bash" &
# Give some time for the first node to initialize (optional)
sleep 5
```

```
# Launch rosbridge_server in a separate terminal
gnome-terminal -- bash -c "ros2 run rosbridge_server rosbridge_websocket.py;
exec bash" &
# Launch arm.py to start node of controller_data to control robotic arm
gnome-terminal -- bash -c "python3 arm.py" &
```

APPENDIX F

```
import asyncio
import websockets
import socket
class SimpleDataChannelService:
    def _init_(self):
        self.clients = set()
    async def _call_(self, websocket, path):
        self.clients.add(websocket)
        print("DataChannel SERVER got connection")
        try:
            async for message in websocket:
                print(f"DataChannel SERVER got message {message}")
                # forward message to all other clients
                for ws in self.clients:
                    if ws != websocket:
                        await ws.send(message)
        finally:
            self.clients.remove(websocket)
async def start_server():
    server_ipv4_address = "0.0.0.0"
    server_port = 5050
    data_channel_service = SimpleDataChannelService()
    server = await websockets.serve(data_channel_service,
server_ipv4_address, server_port)
    print(f"Server SimpleDataChannelServer Started at
ws://{server_ipv4_address}:{server_port}")
    await server.wait_closed()
asyncio.run(start_server())
```

APPENDIX G

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Roxy Sender</title>
```

```
</head>
<body>
    <h1>Roxy Sender</h1>
    <script type="text/javascript"
src="https://cdn.jsdelivr.net/npm/roslib@1/build/roslib.min.js"></script>
    <p> Local Rosbridge Connection: <span id="status_local" style="font-
weight: bold;">N/A</span></p>
    <p> Local Connection: <span id="status_webrtc" style="font-weight:
bold;">N/A</span></p>
    <button onclick="startConnection()">Start Video Connection</button>
    <ul id="messages">
        <!-- Received ROS messages will be displayed here -->
    </ul>
    <script src="sender.js"></script>
</body>
</html>
```

APPENDIX H

```
// include these libraries for using the servo add on board. Taken from
servo example code
#include <Arduino.h>
// include this library for servo easing functionality
#include "ServoEasing.hpp"
/* * * * * * * * * * * * * * * * ** * *
  Definitions
*  * * * * * * * * * * * * * * * * * */
#define VERSION "3.1"
#define ACTION_TIME_PERIOD 1000
const int SERVO1_PIN = 1; // servo pin for joint 1
const int SERVO2_PIN = 2; // servo pin for joint 2
const int SERVO3_PIN = 3; // servo pin for joint 3
const int SERVO0_PIN = 0; // servo pin for end effector
/* * * * * * * * * * * * * * * * * * * * * * * *
  Variables
* * * * * * * * * * * * * * * * * * * * * * * * /
//-------- Variables for receiving serial data -------------
const byte buffSize = 40;
char inputBuffer[buffSize];
const char startMarker = '<';
const char endMarker = '>';
byte bytesRecvd = 0;
boolean readInProgress = false;
boolean newDataFromPC = false;
char messageFromPC[buffSize] = {0};
// -------- Variables to hold time -------------
```

```
unsigned long curMillis; // Variable for current time
// unsigned long general_timer;
// --------- Variables to hold the parsed data -------------
float floatFromPC0 = 90.0; // initial values are mid range for joint angles
float floatFromPC1 = 90.0;
float floatFromPC2 = 90.0;
float floatFromPC3 = 830.0;
int intFromPC0 = 1000; // inital values are acceptable movement times
int intFromPC1 = 1000;
int intFromPC2 = 1000;
int intFromPC3 = 1000;
float last_servoAngle_q1 = floatFromPC1; // initial values are mid range for
joint angles
float last_servoAngle_q2 = floatFromPC2;
float last_servoAngle_q3 = floatFromPC3;
float last_servoAngle_EE = floatFromPC0;
/* * * * * * * * * * * * * * * * * * * * * * *
  Instatiate clasess for libraries
* * * * * * * * * * * * * * * * * * * * * * * * */
ServoEasing Servo1(PCA9685_DEFAULT_ADDRESS, &Wire);
ServoEasing Servo2(PCA9685_DEFAULT_ADDRESS, &Wire);
ServoEasing Servo3(PCA9685_DEFAULT_ADDRESS, &Wire);
ServoEasing Servo0(PCA9685_DEFAULT_ADDRESS, &Wire);
/* * * * * * * * * * * * * * * * * * *
  STARTOFPROGRAM(Setup)
* * * * * * * * * * * * * * * * * * * * */
void setup()
{
  // Begin serial communications
  Serial.begin(115200);
  // Wait for serial communications to start before continuing
  while (!Serial)
    ; // delay for Leonardo
  // Just to know which program is running on my Arduino
  Serial.println(F("START " FILE "\r\nVersion " VERSION " from " DATE));
  // Attach servo to pin
  Servo1.attach(SERVO1_PIN);
  Servo2.attach(SERVO2_PIN);
  Servo3.attach(SERVO3_PIN);
  Servo0.attach(SERVO0_PIN);
  // Set servo to start position.
  Servo1.setEasingType(EASE_CUBIC_IN_OUT);
  Servo2.setEasingType(EASE_CUBIC_IN_OUT);
  Servo3.setEasingType(EASE_CUBIC_IN_OUT);
  Servo0.setEasingType(EASE_CUBIC_IN_OUT); // end effector
  Servo1.write(last_servoAngle_q1);
  Servo2.write(last_servoAngle_q2);
  Servo3.write(last_servoAngle_q3);
  Servo0.write(last_servoAngle_EE); // end effector
  // Just wait for servos to reach position
  delay(5000); // delay() is OK in setup as it only happens once
  // tell the PC we are ready
  Serial.println("<Arduino is ready>");
}
/* * * * * * * * * * * * * * * * * * * * *
MAINPROGRAM(Loop)
* * * * * * * * * * * * * * * * * * * * * */
void loop()
{
  // This part of the loop for the serial communication is not inside a
timer -> it happens very quickly
  curMillis = millis(); // get current time
  getDataFromPC();      // receive data from PC and save it into inputBuffer
  // need if statement -> flag to say if new data is available
  if (newDataFromPC == true)
  {
    actionInstructionsFromPC(); // Arrange for things to move, beep, light
up
    replyToPC();                // Reply to PC
  }
  //  // This part of the loop is inside a timer -> maybe delete
  //  unsigned long elapsed_time_general_timer = millis() - general_timer;
  //
  //  if ( elapsed_time_general_timer > ACTION_TIME_PERIOD ) // set a target
for the Romi periodically
  //  {
  //    // update timestamp
  //    general_timer = millis();
  //  }
}
//~Fuction: Action the instructions from the PC~~~
void actionInstructionsFromPC()
{
  // Local variables
  //  -- joint angles
  float servoAngle_q1 = floatFromPC1;
  float servoAngle_q2 = floatFromPC2;
  float servoAngle_q3 = floatFromPC3;
  float servoAngle_EE = floatFromPC0;
  // -- joint speeds
  int servoTime_q1 = intFromPC1;
  int servoTime_q2 = intFromPC2;
  int servoTime_q3 = intFromPC3;
  int servoTime_EE = intFromPC0;
```

```
  // Check if the joint angle has changed!
  if (servoAngle_q1 != last_servoAngle_q1)
  {
    Serial.println(F("Servo 1 moving to position using interrupts"));
    Servo1.startEaseToD(servoAngle_q1, servoTime_q1);
    //    while (Servo3.isMovingAndCallYield()) {
    //      ; // no delays here to avoid break between forth and back
movement
    //    }
  }
  if (servoAngle_q2 != last_servoAngle_q2)
  {
    Serial.println(F("Servo 2 moving to position using interrupts"));
    Servo2.startEaseToD(servoAngle_q2, servoTime_q2);
    //    while (Servo3.isMovingAndCallYield()) {
    //      ; // no delays here to avoid break between forth and back
movement
    //    }
  }
  if (servoAngle_q3 != last_servoAngle_q3)
  {
    Serial.println(F("Servo 3 moving to position using interrupts"));
    Servo3.startEaseToD(servoAngle_q3, servoTime_q3);
    //    while (Servo3.isMovingAndCallYield()) {
    //      ; // no delays here to avoid break between forth and back
movement
    //    }
  }
  if (servoAngle_EE != last_servoAngle_EE)
  {
    Serial.println(F("Servo EE moving to position using interrupts"));
    Servo0.startEaseToD(servoAngle_EE, servoTime_EE);
    //    while (Servo3.isMovingAndCallYield()) {
    //      ; // no delays here to avoid break between forth and back
movement
    //    }
  }
  // Store current joint angle
  last_servoAngle_q1 = servoAngle_q1;
  last_servoAngle_q2 = servoAngle_q2;
  last_servoAngle_q3 = servoAngle_q3;
  last_servoAngle_EE = servoAngle_EE;
}
/* * * * * * * * * * * * * * * * * * * * * *
        FUNCTIONS FOR RECEIVING DATA VIA SERIAL MONITOR
* * * * * * * * * * * * * * * * * * * * * * /
//~Fuction: Receive data with start and end markers~~
```

```
void getDataFromPC()
{
  // This function receives data from PC and saves it into inputBuffer
  if (Serial.available() > 0 && newDataFromPC == false)
  {
    char x = Serial.read();
    // the order of these IF clauses is significant
    if (x == endMarker)
    {
      readInProgress = false;
      newDataFromPC = true;
      inputBuffer[bytesRecvd] = 0;
      parseData();
    }
    if (readInProgress)
    {
      inputBuffer[bytesRecvd] = x;
      bytesRecvd++;
      if (bytesRecvd == buffSize)
      {
        bytesRecvd = buffSize - 1;
      }
    }
    if (x == startMarker)
    {
      bytesRecvd = 0;
      readInProgress = true;
    }
  }
}
//~~~~~~~~~~
//~Fuction: Split data into known component parts~~
void parseData()
{
  // split the data into its parts
  char *strtokIndx = strtok(inputBuffer, ","); // initialize strtokIndx
  if (strtokIndx != NULL) // check if the first token is not NULL
  {
    // continue parsing
    floatFromPC1 = atof(strtokIndx);
    strtokIndx = strtok(NULL, ",");
    floatFromPC2 = atof(strtokIndx);
    strtokIndx = strtok(NULL, ",");
    floatFromPC3 = atof(strtokIndx);
    strtokIndx = strtok(NULL, ",");
    floatFromPC0 = atof(strtokIndx);
    strtokIndx = strtok(NULL, ",");
```

```
    intFromPC0 = atoi(strtokIndx);
    strtokIndx = strtok(NULL, ",");
    intFromPC1 = atoi(strtokIndx);
    strtokIndx = strtok(NULL, ",");
    intFromPC2 = atoi(strtokIndx);
    strtokIndx = strtok(NULL, ",");
    intFromPC3 = atoi(strtokIndx);
  }
}
//~~~~~~~~~~
//~Fuction: Send message back to PC~~~~~
void replyToPC()
{
  if (newDataFromPC)
  {
    newDataFromPC = false;
    Serial.print(F("<Msg "));
    Serial.print(F(" floatFromPC1 "));
    Serial.print(floatFromPC1);
    Serial.print(F(" floatFromPC2 "));
    Serial.print(floatFromPC2);
    Serial.print(F(" floatFromPC3 "));
    Serial.print(floatFromPC3);
    Serial.print(F(" floatFromPC0 "));
    Serial.print(floatFromPC0);
    Serial.print(F(" intFromPC0 "));
    Serial.print(intFromPC0);
    Serial.print(F(" intFromPC1 "));
    Serial.print(intFromPC1);
    Serial.print(F(" intFromPC2 "));
    Serial.print(intFromPC2);
    Serial.print(F(" intFromPC3 "));
    Serial.print(intFromPC3);
    Serial.print(F(" Time "));
    Serial.print(curMillis / 1000); // divide by 512 is approx = half-
seconds
    Serial.println(F(">"));
  }
}
```

REFERENCES

[1] *Chen, T., Zhao, D. and Zhang, Z., 2007, December. Research on the tele-operation robot system with tele-presence based on the virtual reality. In 2007 IEEE International Conference on Robotics and Biomimetics (ROBIO) (pp. 302306). IEEE.*

[2] *Gong, M. and Changchun, D., 2003. Research on construction telerobotic system with telepresence. D. Changchun, College of Mechanical Science and Engineering, Jilin University.*

[3] *Tao, N. and Changchun, D., 2006. Research on telerobotic system with force and visual presence. College of Mechanical Science and Engineering Jilin University, Changchun.*

[4] *SYU, K., 1994. Variable-structure model following adaptive control for systems with time-varying delay. Control Theory and Advanced Technology, 10(3), pp.513-521.*

[5] *Koopmann, A., Fox, K., Raich, P. and Webster, J., Virtual Reality Teleoperation Robot.*

[6] *Yamada, H., Tao, N. and DingXuan, Z., 2008, September. Construction tele-robot system with virtual reality. In 2008 IEEE conference on robotics, automation and mechatronics (pp. 36-40). IEEE.*

[7] *Whitney, D., Rosen, E., Ullman, D., Phillips, E. and Tellex, S., 2018, October. Ros reality: A virtual reality framework using consumer-grade hardware for ros-enabled robots. In 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 1-9). IEEE.*

[8] *Moon, B.H., Choi, J.W., Jung, K.T., Kim, D.H., Song, H.J., Gil, K.J. and Kim, J.W., 2017, June. Connecting motion control mobile robot and VR content. In 2017 14th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI) (pp. 355-359). IEEE.*

[9] *Hasan Ba-Btain, F.A., 2022. The feasibility of bitscope raspberry pi for monitoring voltage and capturing data (Doctoral dissertation, Universiti Tun Hussein Malaysia).*

[10] *Koopmann, A., Fox, K., Raich, P. and Webster, J., Virtual Reality Teleoperation Robot.*

[11] *Yamada, H., Tao, N. and DingXuan, Z., 2008, September. Construction tele-robot system with virtual reality.*
In 2008 IEEE conference on robotics, automation and mechatronics (pp. 36-40). IEEE.

[12] *Yamada, H., Kato, H. and Muto, T., 2003. Master-slave control for construction robot teleoperation. Journal of Robotics and Mechatronics, 15(1), pp.54-60.*

[13] *Yamada, H. and Muto, T., 2007. Construction Tele-Robotic System with Virtual Reality (CG presentation of virtual robot and task object using stereo vision system). Control and intelligent systems, 35(3), pp.195-201.*

[14] *Yamada, H. and Muto, T., 2003. Development of a hydraulic tele-operated construction robot using virtual reality: New master-slave control method and an evaluation of a visual feedback system. International Journal of Fluid Power, 4(2), pp.35-42.*

[15] *Tang, X., Zhao, D., Yamada, H. and Ni, T., 2009, August. Haptic interaction in tele-operation control system of construction robot based on virtual reality. In 2009 International Conference on Mechatronics and Automation (pp. 78-83). IEEE.*

[16] *Yamada, H., Tao, N. and DingXuan, Z., 2008, September. Construction tele-robot system with virtual reality.*
In 2008 IEEE conference on robotics, automation and mechatronics (pp. 36-40). IEEE.

[17] *Yamada, H., Ming-de, G. and Dingxuan, Z., 2007. Master-slave control for construction robot teleoperationapplication of a velocity control with a force feedback model. Journal of Robotics and Mechatronics, 19(1), pp.6067.*

[18] *Mingde, G., Dingxuan, Z., Shizhu, F., Hailong, W. and Yamada, H., 2008, September. Force feedback model of electro-hydraulic servo tele-operation robot based on velocity control. In 2008 IEEE Conference on Robotics, Automation and Mechatronics (pp. 912-915). IEEE.*

[19] *Whitney, D., Rosen, E., Ullman, D., Phillips, E. and Tellex, S., 2018, October. Ros reality: A virtual reality framework using consumer-grade hardware for ros-enabled robots. In 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 1-9). IEEE.*

[20] *Carlos Beltr´an-Gonz´alez, Antonios Gasteratos, Angelos manatiadis, Dimitrios Chrysostomou, Roberto Guzman, Andr´as T´oth, Lor´and Szollosi, Andr´as Juh´asz, and P´eter Galambos. Methods and techniques for intelligent navigation and manipulation for bomb disposal and rescue operations. In Safety, Security and Rescue Robotics, 2007. SSRR 2007. IEEE International Workshop on, (pages 1–6.) IEEE.*

[21] *Goldberg, K., Mascha, M., Gentner, S., Rothenberg, N., Sutter, C. and Wiegley, J., 1995, May. Desktop teleoperation via the world wide web. In Proceedings of 1995 IEEE International Conference on Robotics and Automation (Vol. 1, pp. 654-659). IEEE.*

[22] *Vertut, J. ed., 2013. Teleoperation and robotics: applications and technology (Vol. 3). Springer Science & Business Media.*

[23] *Whitney, D., Rosen, E., Phillips, E., Konidaris, G. and Tellex, S., 2019, November. Comparing robot grasping teleoperation across desktop and virtual reality with ROS reality. In Robotics Research: The 18th International Symposium ISRR (pp. 335-350). Cham: Springer International Publishing.*

[24] *Byrn, J.C., Schluender, S., Divino, C.M., Conrad, J., Gurland, B., Shlasko, E. and Szold, A., 2007. Threedimensional imaging improves surgical performance for both novice and experienced operators using the da Vinci Robot System. The American Journal of Surgery, 193(4), pp.519-522.*

[25] *Mallwitz, M., Benitez, L.V., Bongardt, B. and Will, N., 2014. The CAPIO Active Upper Body Exoskeleton. Proceedings of the RoboAssist.*

[26] *Lipton, J.I., Fay, A.J. and Rus, D., 2017. Baxter's homunculus: Virtual reality spaces for teleoperation in manufacturing. IEEE Robotics and Automation Letters, 3(1), pp.179-186.*

[27] *Zhang, T., McCarthy, Z., Jow, O., Lee, D., Chen, X., Goldberg, K. and Abbeel, P., 2018, May. Deep imitation learning for complex manipulation tasks from virtual reality teleoperation. In 2018 IEEE International Conference on Robotics and Automation (ICRA) (pp. 5628-5635). IEEE.*

[28] *Crick, C., Jay, G., Osentoski, S., Pitzer, B. and Jenkins, O.C., 2017. Rosbridge: Ros for non-ros users. In Robotics Research: The 15th International Symposium ISRR (pp. 493-504). Springer International Publishing.*

[29] *Xinxing Tang, Dingxuan Zhao, HironaovYamada, Tao Ni. "Haptic interaction in teleoperation control system of construction robot based on virtual reality", 2009 International Conference on Mechatronics and Automation, 2009.*

[30] *Yao, Bi-qiang, and Guang Wen. "Research on the tele-operation robot system with telepresence", 2011 IEEE International Conference on Mechatronics and Automation, 2011.*

[31] *Hironao Yamada, Tang Xinxing, Ni Tao, Zhao Dingxuan, Ahmad Anas Yusof. "Tele-operation Construction Robot Control System with Virtual Reality", IFAC Proceedings Volumes, 2009.*

[32] *David Whitney, Eric Rosen, Daniel Ullman, Elizabeth Phillips, Stefanie Tellex. "ROS Reality: A Virtual Reality Framework Using Consumer-Grade Hardware for ROS-Enabled Robots", 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2018.*

[33] *Guang Wen. "Research on the tele-operation robot system with tele-presence", The Fourth International Workshop on Advanced Computational Intelligence, 10/2011.*

[34] *Rebecca Hetrick, Nicholas Amerson, Boyoung Kim, Eric Rosen, Ewart J. de Visser, Elizabeth Phillips. "Comparing Virtual Reality Interfaces for the Teleoperation of Robots", 2020 Systems and Information Engineering Design Symposium (SIEDS), 2020.*

[35] *Hironao Yamada, Ni Tao, Zhao DingXuan. "Construction Tele-robot System With Virtual Reality", 2008 IEEE Conference on Robotics, Automation and Mechatronics, 2008.*

[36] *Internet Source: h2r.cs.brown.edu.*

[37] *Internet Source: vigir.missouri.edu.*

[38] *Internet Source: www.researchgate.net.*

[39] *Internet Source: web.archive.org.*

[40] *Internet Source: www.coursehero.com.*