CG2111A Engineering Principle and Practice II

Semester 2 2023/2024

"Alex to the Rescue"

Final Report

Team: B05-1A

Name	Student #	Sub-Team	Role
Tan Ping Hui		Software	Leader
Teo Keng Jer		Software	Software Lead
Tng Wen Xi		Hardware	Hardware Lead
Vasantharaj Frederick Amal Emerson		Firmware	Firmware Lead

Index

Section 1 Introduction

In times of natural disasters or terrorist attacks, swift search and rescue operations are imperative to enhance the chances of survival for individuals trapped under rubble and debris. In these critical situations, rescuers often face significant risks as they brave unstable and perilous environments to reach survivors. Moreover, narrow and obstructed passageways can hinder human access to these affected areas. To address these complexities, robots have been deployed to navigate through these hazardous terrains more effectively and securely, aiding in the search for signs of life.

Introducing Alex, a robotic vehicle with an array of search and rescue functionalities.

1. **Visualization:** Alex is mounted with a Light Detection and Ranging (LiDAR) sensor to scan and detect surrounding obstacles. Coupled with its integration into the Robot Operating System (ROS) network and ROS Visualisation (RViz), operators can visualize the unknown environment in real-time and pinpoint potential victim locations remotely.

2. **Secure Remote Control:** Operators can also remotely and securely control Alex's direction, distance, and speed with precision, facilitated by Transport Layer Security (TLS) communication with a remote Personal Computer (PC).

3. **Collision Prevention:** An ultrasonic sensor is employed to automatically halt Alex's movement when encountering obstacles in its path, averting potential collisions.

4. **Victim Identification:** A color sensor will differentiate green, red and white objects, emulating healthy, injured and non-victims respectively.

5. **Compact Design:** Alex's small size allows it to navigate through narrow passageways and inaccessible areas.

Section 2 Review of State of the Art

2.1 Robot Teleoperativo

Robot Teleoperativo consists of 2 modules, the field robot and the pilot station. The field robot integrates the versatile locomotion of the HyQReal robot with the dexterity and power of a robotic arm. At the pilot station, operators can make use of a 3D virtual reality based interface to perceive the actual environment the robot is in. They can then teleoperate the robotic arm with a haptic teleoperation device to perform actions such as door opening in real time. This reduces the need for humans to be on-site for rescue operations.

- Strengths: The field-pilot combination offers an immersive telepresence experience to the operator and allows them to project their hand movements to the field robot arm accurately and remotely. It is also capable of navigating challenging terrain, such as slopes and stairs.
- Limitations: The operator is limited to the front view, making it challenging to comprehensively locate or map out the entire site at once.

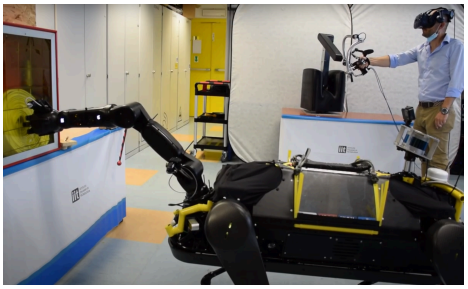


Figure 1: An Operator controlling Robot Teleoperativo

Source: Robohub

2.2 Jueying X20

Jueying X20 is a quadruped robot that features strong load capacity and scalability. Its perception system is equipped with a depth-sensing camera and a ROS-based perception computing and artificial intelligence module. Due to its great computing power and perceptual precision, it can be used in rescue operations, even in adverse weather conditions.

- Strengths: Its wide peripheral platform supports the equipment with multiple interfaces for power supply and communications, such as robotic arm, 5G communication, Global Positioning System (GPS) and Real-time kinematic positioning (RTK). It is also capable of navigating challenging terrain.
- Limitations: Although it works in controlled or pre-mapped environments effectively, its feature of a quadruped robot can pose challenges in navigating through uneven or unfamiliar terrain with debris or rubble compared to wheeled robots.

Figure 2: Jueying X20 operating under rainy conditions

Source: DEEP Robotics

Section 3 System Architecture

3.1 Core Devices and Components:

1. **Raspberry-Pi (R-Pi)**: Manages communication between Alex's Arduino and the operator's PC through Universal Asynchronous Receiver-Transmitter (UART) and TLS respectively, thereby enabling remote control. Additionally, it publishes LiDAR data for remote visualization on the ROS Network
2. **Remote PC**: To send commands to Alex and visualize its surroundings.
3. **Arduino**: Controls key components onboard Alex, such as motors, wheel encoder, ultrasonic sensor, color sensor and Ring LED.
4. **Motor Driver**: Amplifies the signal from the Arduino to power and control the motor.
5. **Motor & Encoder**: The motor powers the robot's movement, while the encoder provides feedback on speed and direction, aiding precise control.
6. **LiDAR**: Scans Alex's surroundings, aiding in victim identification and navigation.
7. **Ultrasonic Sensor**: Measure distances of obstacles ahead to avoid collisions.
8. **Color Sensor & Ring LED**: The color sensor detects object colors, while the Ring LED displays the identified color.

3.2 Deployment Diagram

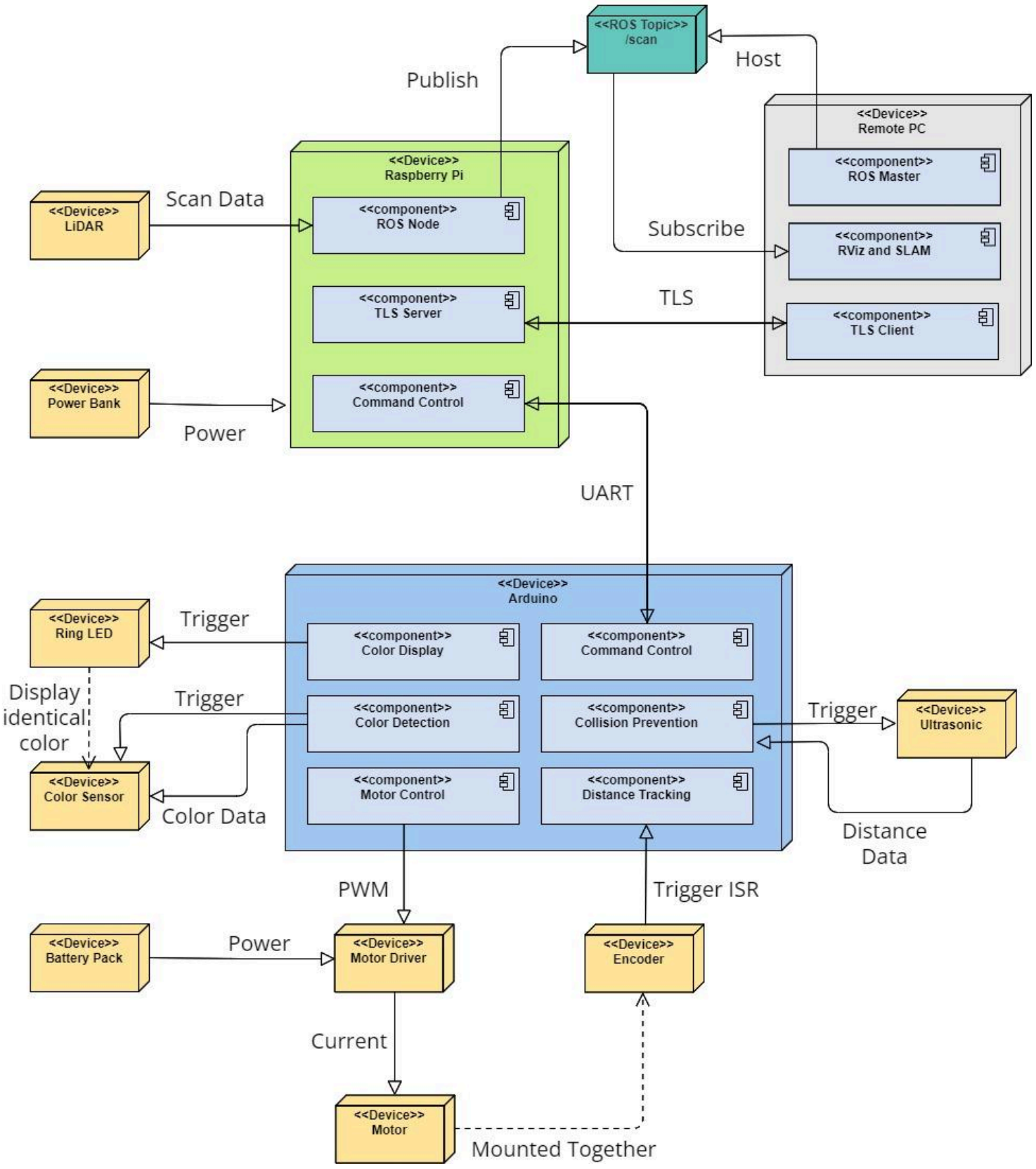
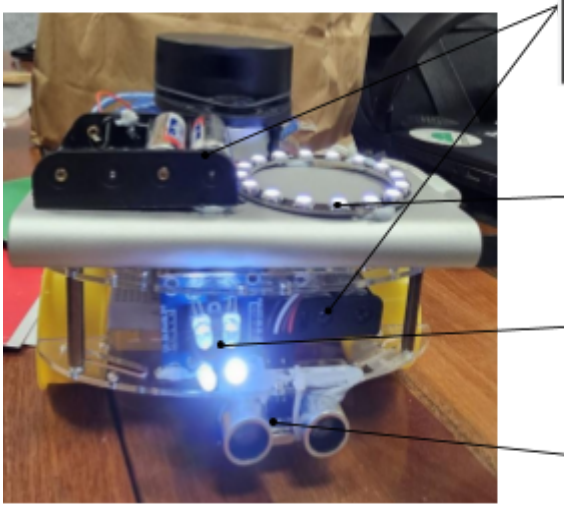


Figure 3: UML Deployment Diagram

Section 4 Hardware Design

4.1 Final Form of the System

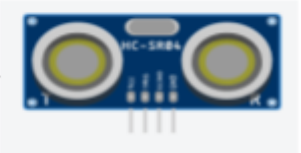
Front View



1.5V x 6 batteries
(by soldering 2
battery holders)

Neopixel Ring
Light (WS2812)

Colour Sensor
(TCS3200)



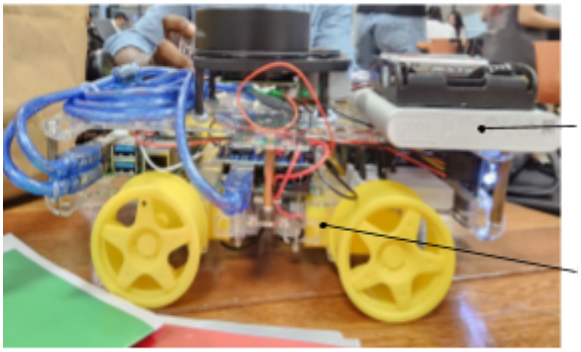
Ultrasonic
Sensor

Back View

Raspberry Pi LIDAR

Raspberry Pi 4

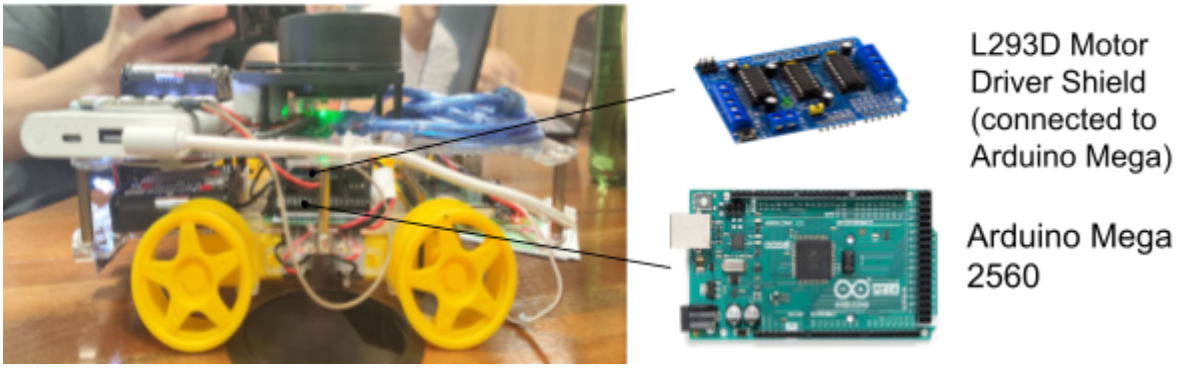
Side View (R)



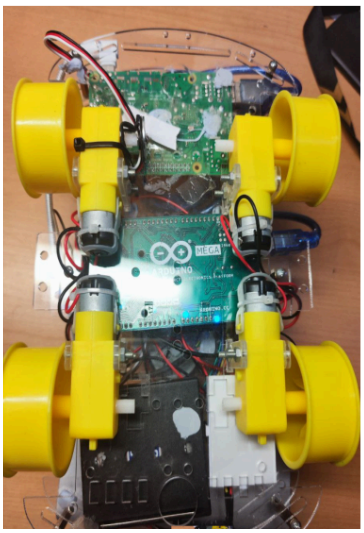
Xiaomi 10000mAh
Power Bank

DC Motor with
Encoder (Hall Effect
Sensor)

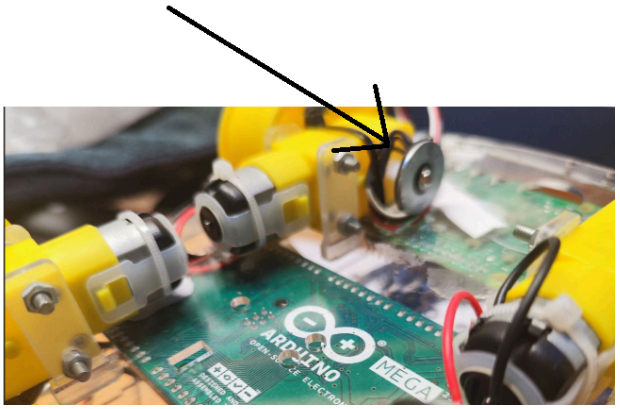
Side View (L)



BOTTOM VIEW



ZOOMED IN VIEW OF ENCODER



4.2 Non-standard Hardware Components

Alex has two additional non-standard hardware components: an ultrasonic sensor mounted at the front, and a Neopixel WS2812 LED ring light mounted on top.

- The ultrasonic sensor is used to prevent collisions while moving forward, and will be explained in detail in section 5.1.1.
- The ring light is initialized to show a rainbow gradient at the start. When Alex detects the color of an object, each pixel within the ring light will light up sequentially in the color sensed by Alex, providing real-time visual feedback.

Section 5 Firmware Design

5.1 Movement Protocol

5.1.1 Setup and Calibration

Operators control Alex's movements by specifying the distance for forward and reverse commands or the angle for left and right commands, stored in the params array within the command packet sent from the R-Pi to the Arduino. Further details are provided in Section 6.12.

Alex is equipped with a wheel encoder mounted on its back-left wheels and the hall sensor connected to an external interrupt pin (PIN18 - INT3) of the Arduino, which is configured to trigger an Interrupt Service Routine (ISR) on a falling edge. (Refer to Appendix B.1.3). As Alex moves, the wheel encoder rotates, causing the hall sensor's voltage output to drop and triggering the ISR. This updates global variables such as `leftForwardTicks`, `leftReverseTicks`, `leftForwardTicksTurns`, or `leftReverseTicksTurns`, depending on Alex's direction of movement. Alex only relies on one encoder as it was found to be more reliable than the default dual encoder setup (One left and one right). Alex relies solely on one encoder as it was found to be more reliable than the default dual encoder setup (one left and one right). Therefore, all calculations are based on the data from the left encoder, and the right encoder was removed from the setup.

To accurately compute Alex's distance traveled with the encoder, we measured and calculated key dimensions of the robot, such as its wheel circumference (`WHEEL_CIRC`) and diagonal length (`alexDiagonal`). Additionally, calibration of the encoder is essential to address variations in the number of ISR triggers per wheel revolution. Through the calibration process, we determine that setting `COUNTS_PER_REV` to 3.3 (average ISR trigger per revolution) provides the most precise and reliable distance calculation (Refer to Appendix B.1.1).

5.1.2 Forward and Backward

When the Arduino receives a forward or backward command, it invokes the `forward()` or `backward()` function, respectively. Based on the specified distance to move, the new total forward or backward distance that Alex should have moved by the end of the movement is computed and stored in the global variable `newDist`. As Alex moves, the ISR updates `forwardDist` or `reverseDist` based on `COUNTS_PER_REV` and `WHEEL_CIRC`. Alex stops when `forwardDist` or `reverseDist` reaches `newDist` by calling `stop()`. Notably, after executing every movement command, Alex calls `sendsStatus()` to provide operators with information about the ticks, distances, and ultrasonic reading (Refer to Appendix B.1.9).

5.1.3 Left and Right

For turn left and turn right commands, the `left()` and `right()` functions are called, respectively. These functions compute the number of ticks required to turn the angle specified by the user and update the global variable `targetTicks`. As Alex turns, the ISR updates `leftForwardTicksTurns` (for right turns) or `leftReverseTicksTurns` (for left turns). Alex stops when `leftForwardTicksTurns` or `leftReverseTicksTurns` reaches `targetTicks`.

5.1.4 Collision Prevention (Ultrasonic Sensor)

To prevent collisions, Alex is equipped with an ultrasonic sensor mounted at the front. The configuration for this sensor is implemented in bare-metal within the `setUltrasonic()` function (Refer to Appendix B.1.5). Within the main loop, if Alex is moving forward, the `checkUltrasonic()` function is called to retrieve the distance sensed by the ultrasonic sensor. If the sensor detects an object within 10 cm in front of Alex, regardless of whether `forwardDist` has reached `newDist`, Alex will halt its forward movement to avoid a collision.

To allow Alex to continue moving forward despite this restriction, a 100 ms delay is implemented. This delay allows Alex to advance approximately 2 cm forward before `checkUltrasonic()` is called, ensuring finer control over Alex's forward movement when it is in close proximity to an obstacle (Refer to Appendix B.1.9).

5.2 Victim Identification (Color Sensor)

In the simulated environment, victims are either “green” or “red”, and a dummy victim is indicated by the color white. As such, Alex needs to recognize the colors red, green, and white. The TCS3200 color sensor is mounted at the front of Alex. Configuration for the color sensor is done in bare-metal in the function `setColor()` , where it is configured to have a frequency scaling of 20% (Refer to Appendix B.1.4). The color sensor consists of 16 photodiodes each for red, green, and blue (RGB), and an additional 16 photodiodes without filters. It converts the photodiodes’ readings into a square wave with frequency proportional to the color’s light intensity. The `colorId[]` array stores the duration for which the output frequency pin stays low for each color (Refer to Appendix B.1.4). As the intensity of a color increases, the output frequency increases, and the duration decreases. This enables the color sensor to measure the intensity of the RGB colors.

After calibration, it was observed that white colors consistently result in durations of less than 2000 µs for red, green, and blue. Thus, if all three values in the `colorId[]` array fall below 2000, Alex categorizes the object as white. Else, if the red value is the lowest among the three, Alex identifies the object as red. Lastly, if neither condition is met, Alex detects the object as green (Refer to Appendix B.1.4).

5.3 Communication Protocol

The R-Pi and the Arduino communicate through UART serial communication. Messages and responses are sent between the R-Pi to the Arduino in the form of TPacket data structures, which contain the packet type, the command, a string containing the data, and a params array of 16 integers.

The Arduino receives a packet from the R-Pi by reading in data from the serial port in the function `readSerial()` (Refer to Appendix B.1.7), and deserializes the packet in the function `readPacket()`. The function `handlePacket()` then passes the address of the TPacket to the function `handleCommand()`, which then calls the relevant functions depending on the command type (Refer to Appendix B.1.8).

When the Arduino sends a packet to the R-Pi, it first serializes the packet before sending it to the R-Pi (Refer to Appendix B.1.8). The R-Pi deserializes the packet (Refer to Appendix A.1.2) and the function `handleUARTPacket()` then calls the appropriate function (`handleResponse()`, `handleErrorResponse()`, or `handleMessage()`), depending on the packet type.

5.4 Additional features

Additional features of Alex include the ultrasonic sensor explained in section 5.1.1 and the ring light mentioned in section 4.2.

The ring light is programmed using the Adafruit NeoPixel library, and two functions `colorWipe()` and `rainbowCycle()` were used (Refer to Appendix B.1.6). `rainbowCycle()` is used to initialize the rainbow gradient, while `colorWipe()` is used to display the color detected.

Section 6 Software Design

6.1 Teleoperation Protocol

6.1.1 TLS Communication

To ensure secure remote control over Alex, TLS communication is implemented for data exchange between operators and the robot. TLS effectively mitigates security risks, including unauthorized access or hijacking, which could compromise search and rescue efforts, particularly in scenarios involving terrorist attacks.

Both the server's (Alex's R-Pi) and the client's (Operator's Remote PC) digital certificates are signed by a trusted Certificate Authority (Self-Created) in advance. This enables both parties to validate the authenticity of each other's certificates in `createServer()`, completing the initial handshake process before establishing a secure connection (Refer to Appendix A.1.4).

6.1.2 Movement Commands

In the TLS client program, stringent control measures are implemented to ensure precise movement of Alex and minimize the potential for operator errors. Operators are restricted to issuing one command at a time, specifying direction ("f" for forward, "b" for back, "l" for left, "r" for right, “s” for stop) and distance for Alex's movement. Invalid inputs trigger a "BAD COMMAND" message loop until corrected, while valid commands generate a "COMMAND" packet with direction and parameters (distance and speed) encrypted and sent to the R-Pi via `sslWrite()`. Notably, the speed parameter is fixed at 100 to facilitate swift movements (Refer to Appendix C.1.3).

In the TLS server program, encrypted packets are decrypted via `sslRead()`, serialized into TComms data, and transmitted to Arduino via UART for execution (Refer to Appendix A.1.3). Subsequently, Aduino will send a packet back to the R-Pi via UART, which is then relayed to the PC via TLS (Refer to Appendix A.1.2). Under normal circumstances, Arduino will respond with a “STATUS” packet containing sensor data like encoder ticks and

ultrasonic readings (Refer to Appendix C.1.2), except for the initial handshake process. This data allows operators to assess Alex's surroundings and operational status, aiding in informed decision-making for subsequent movements or actions. In the event of transmission issues, such as bad checksum, Arduino will send an "ERROR" packet, indicating faults in the UART communication (Refer to Appendix B.1.8). This prompts operators to address communication channel faults or resend the command.

6.2 Visualization

Alex's operations often take place in uncharted or unpredictable terrain, necessitating the use of Simultaneous Localization and Mapping (SLAM) Algorithm, particularly Hector SLAM (Detailed explanation in Appendix E) to map its immediate surroundings. However, SLAM algorithms demand significant computational power, which may cause the R-Pi to be overloaded and crash. Hence, to enhance system reliability, we've integrated a ROS network to enable LiDAR data to be processed remotely on the PC, leveraging its superior computational power over the R-Pi.

In this setup, the PC serves as the master node, receiving LiDAR data from Alex's onboard LiDAR (/rplidarNode) via the /scan topic. Subsequently, the PC (/rviz_mapping) subscribes to this topic and processes the LiDAR data with Hector SLAM for visualization in Rviz. Concurrently, Hector SLAM estimates Alex's current position coordinates, publishing them through the /slam_out_pose topic. Rviz can subscribe to this topic, displaying a default arrow to represent Alex's position. To further enhance visualization, we've customized the Rviz launch file to replace the arrow with a box resembling Alex's dimensions and scaled down the map grids to 5cm by 5cm. These modifications will minimize collision risks and help with victim identification.

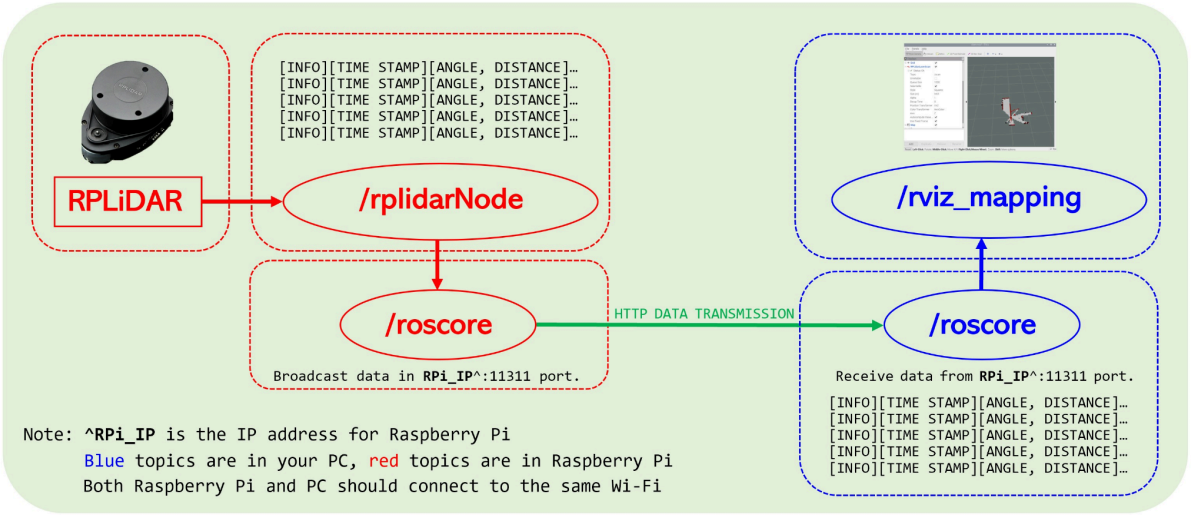


Figure 4: ROS Networking Architecture
Source: Boyd.A & Chen G.Y. ROS Networking

6.3 Color Detection

With the Rviz environment map, operators can identify objects resembling victims and direct Alex towards them using movement commands. To evaluate the color of these objects, operators activate the color sensor by issuing the command "g," prompting Alex to update the RGB values and the color variable (0-Red, 1-Green, 2-White) in the "STATUS" packet (Refer to Appendix C.1.2). The RGB values represent the raw input data from the color sensor and an algorithm (Refer to Section 5.2) will process these RGB values to derive the color variable, reflecting the perceived color of the object as interpreted by Alex. Including the raw RGB data in the status packet allows operators to evaluate the reliability of the identified color. Minimal differences in RGB values may suggest inaccuracies, prompting operators to resend the command for accurate color assessment.

Section 7 Lessons Learnt - Conclusion

7.1 Mistakes Made

7.1.1 Mistake 1: Over-Reliance on Single 9V rechargeable battery

Initially, we faced limitations with the 4xAA battery holder, as it could only provide a maximum of 6V, insufficient to power the motors for turning. This led us to experiment with 9V batteries, but their short lifespan and frequent replacements posed sustainability and budget concerns. In response, we opted for rechargeable batteries, specifically a single 9V battery by PALO, chosen for its compact size.

The day before the trial run, we conducted thorough testing by connecting the 9V battery to a power bank while powering the motor driver. This setup proved reliable, operating flawlessly for over two hours without any issues. We further validated its performance on the actual day of the trial run, where it executed all commands flawlessly during testing.

However, during the actual run, Alex encountered a setback after the first minute, losing the ability to turn and only being able to move forward and backward. This setback significantly impacted the success of our mission, underscoring the importance of selecting reliable equipment and conducting comprehensive testing.

Following this setback, we sought input from other groups regarding their power management strategies. Based on their experiences, we adopted a configuration of six 1.5V AA batteries, as they had proven to be reliable for most of them. This involved soldering two 4xAA battery holders together to provide the necessary voltage for Alex's motors. This revised approach is proven to be much more reliable and consistent in our future runs, including the final run.

7.1.2 Mistake 2: Over-Reliance on a Single Encoder for Movement Control

In section 5.1.1, we discuss how Alex depends heavily on a single encoder on its left wheel for measuring and managing movement. This dependency presents numerous risks and potential inaccuracies in managing Alex's motion. Should either the encoder or the left wheel encounter a malfunction, Alex could continue running indefinitely as the ISR would never be triggered, ultimately stalling mission progression. This scenario is highly probable in real-world contexts, where Alex navigates terrains like mud or slick surfaces, potentially causing one wheel to become immobilized. Even within our simulated environment, instances such as the wheel losing power (Refer to section 7.1.1) or the hall sensor position shifts arise, resulting in compromised movement control for Alex.

Therefore, in the context of a genuine search and rescue operation, it becomes imperative to implement a more sophisticated system to bolster the reliability of Alex's movement control. This entails integrating encoders across multiple wheels and developing firmware and software mechanisms that systematically monitor all encoder inputs to accurately assess distance traveled. Moreover, such a system could proactively detect wheel failures and initiate corrective actions to rectify future movements, ensuring a more reliable operation.

7.2 Lessons Learnt

7.2.1 Lesson 1: Efficient Resource Management

The R-Pi, while compact, faces constraints in processing power compared to larger computer systems. This limitation necessitates innovative strategies for managing computational demands, especially during complex tasks like SLAM processing, as discussed in section 6.2.

One effective strategy involves leveraging distributed computing in robotics by utilizing a remote PC for processing SLAM data. SLAM tasks, which involve mapping an unknown environment while tracking Alex's position within that map, require substantial computational resources. Offloading this task to a more powerful PC alleviates the R-Pi's burden, enabling it to concentrate on real-time control and sensor integration tasks more efficiently. Consequently, this enhances Alex's responsiveness and reliability. Moreover, this approach facilitates the incorporation of additional sensors on board Alex to bolster its functionality without straining its current hardware setup.

Our experience with the ROS network has highlighted the importance of efficient resource management. As computational demands increase, integrating additional PCs or cloud-based services can also be utilized to enhance Alex's capabilities without compromising its core functions. This approach broadens Alex's potential, particularly in scenarios with limited onboard resources. By effectively leveraging external resources, future enhancements can unlock new possibilities, making Alex a more versatile and capable robotic platform for various applications in dynamic environments.

7.2.2 Lesson 2: Handling Sensor Data Under Different Operating Conditions

Integrating and managing sensor data across diverse environmental conditions presents significant challenges for our team. Ensuring reliable sensor performance amidst varying terrains, obstacles, and environmental factors like lighting necessitates extensive testing and calibration, as these factors can markedly influence sensor output.

For instance, the ultrasonic sensor must adeptly adapt to different surface types and environmental conditions, which can distort sound propagation and potentially lead to misjudgments of distance ahead of Alex, resulting in collisions. To address this, both hardware design and software algorithms are pivotal. Software algorithms play a crucial role in interpreting sensor data accurately across diverse scenarios. For instance, software can dynamically adjust readings from ultrasonic sensors based on factors like temperature and humidity, which directly impact sound speed.

Similarly, we observed that the color sensor's output varies significantly based on the lighting conditions within different rooms. Calibration can be done to scale the output across various lighting conditions to ensure consistency. Otherwise, in our case, we conducted calibrations at a fixed distance from the object, which proves to be effective in producing reliable results, irrespective of lighting conditions (Refer to section 5.2)

These challenges become even more pronounced in real-world scenarios, where environmental conditions are more unpredictable and diverse. Future designs must incorporate rigorous and possibly automated recalibration routines to address these issues. Automated routines could periodically check and adjust critical components' calibration without human intervention. For example, if there is a significant disparity between ultrasonic readings and LiDAR readings, an automated system could prompt the ultrasonic sensor to adjust its angle. By tilting its angle such that its emitter and receiver are perpendicular to the obstacles ahead, the sensor could obtain more reliable readings.

The key lesson learned is the paramount importance of establishing and maintaining a robust calibration framework within robotic systems. This framework not only enhances operational efficiency but also significantly enhances reliability and safety, particularly in environments where precision is critical. By prioritizing precision and integrating advanced calibration technologies, future designs can achieve heightened functionality and adaptability.

Reference

Di Tecnologia, I. I. (n.d.). *The new IIT's "Robot Teleoperativo" (teleoperation robot) - Robohub*. https://robohub.org/the-new-iits-robot-teleoperativo-teleoperation-robot/

Jueying X20 - Define a New Boundary for Quadruped Robot Industry Applications. (n.d.). *DEEPRobotics*. https://deeprobotics.oss-cn-hangzhou.aliyuncs.com/YSC/20220701113127_6495.pdf

Appendix A - Code in R-Pi

A.1 tls-alex-server.cpp

A.1.1 Constants and Libraries Declaration (Line 1-32)

```
#include "make_tls_server.h"
#include "tls_common_lib.h"
#include "netconstants.h"
#include "constants.h"
#include "packet.h"
#include "serial.h"
#include "serialize.h"

// Port name of Arduino
#define PORT_NAME           "/dev/ttyACM0"

// Ensure same baud rate as Arduino (Alex.ino)
#define BAUD_RATE           B9600

// TLS Port Number
#define SERVER_PORT         5001

// Parameters and Filenames for TLS Handshake
#define PORTNUM 5001
#define KEY_FNAME "alex.key" // Alex's private key
#define CERT_FNAME "alex.crt" // Alex's certificate
#define CA_CERT_FNAME "signing.pem" // CA certificate name
#define CLIENT_NAME "kengjer" // PC's name

// Network buffer consists of 1 byte of packet type, and 128 bytes of data
#define BUF_LEN             129

// To show whether a network connection is active and prevent multi-connection
to keep it simple
static volatile int networkActive;
// To send back responses in sendNetworkData and handleNetworkData
static void *tls_conn = NULL;
```

A.1.2 Serial Routines with Arduino (Line 37 -165)

```
// Prototype for sendNetworkData
void sendNetworkData(const char *, int);

void handleErrorResponse(TPacket *packet)
{
    printf("UART ERROR: %d\n", packet->command);
    char buffer[2];
    buffer[0] = NET_ERROR_PACKET;
    buffer[1] = packet->command;
    sendNetworkData(buffer, sizeof(buffer));
}

// For Debugging, not used in mission
void handleMessage(TPacket *packet)
{
    char data[33];
    printf("UART MESSAGE PACKET: %s\n", packet->data);
    data[0] = NET_MESSAGE_PACKET;
    memcpy(&data[1], packet->data, sizeof(packet->data));
```

```
    sendNetworkData(data, sizeof(data));
}

void handleStatus(TPacket *packet)
{
    char data[65];
    printf("UART STATUS PACKET\n");
    data[0] = NET_STATUS_PACKET;
    memcpy(&data[1], packet->params, sizeof(packet->params));
    sendNetworkData(data, sizeof(data));
}

void handleResponse(TPacket *packet)
{
    // The response code is stored in command
    switch(packet->command)
    {
        case RESP_OK:
            char resp[2];
            printf("Command OK\n");
            resp[0] = NET_ERROR_PACKET;
            resp[1] = RESP_OK;
            sendNetworkData(resp, sizeof(resp));
        break;

        case RESP_STATUS:
            handleStatus(packet);
        break;

        default:
        printf("Boo\n");
    }
}

void handleUARTPacket(TPacket *packet)
{
    switch(packet->packetType)
    {
        case PACKET_TYPE_COMMAND:
            break; // Only we send command packets, so ignore


        case PACKET_TYPE_RESPONSE:
                handleResponse(packet);
            break;

        case PACKET_TYPE_ERROR:
                handleErrorResponse(packet);
            break;

        case PACKET_TYPE_MESSAGE:
                handleMessage(packet);
            break;
    }
}

void uartSendPacket(TPacket *packet)
{
    char buffer[PACKET_SIZE];
    int len = serialize(buffer, packet, sizeof(TPacket));
```

```
    serialWrite(buffer, len);
}

void handleError(TResult error)
{
    switch(error)
    {
        case PACKET_BAD:
            printf("ERROR: Bad Magic Number\n");
            break;

        case PACKET_CHECKSUM_BAD:
            printf("ERROR: Bad checksum\n");
            break;

        default:
            printf("ERROR: UNKNOWN ERROR\n");
    }
}

void *uartReceiveThread(void *p)
{
    char buffer[PACKET_SIZE];
    int len;
    TPacket packet;
    TResult result;
    int counter=0;

    while(1)
    {
        len = serialRead(buffer);
        counter+=len;
        if(len > 0)
        {
            result = deserialize(buffer, len, &packet);

            if(result == PACKET_OK)
            {
                counter=0;
                handleUARTPacket(&packet);
            }
            else
                if(result != PACKET_INCOMPLETE)
                {
                    printf("PACKET ERROR\n");
                    handleError(result);
                } // result
        } // len > 0
    } // while
}
```

A.1.3 Network Routines with PC (Line 171-294)

```
void sendNetworkData(const char *data, int len)
{
    // Send only if network is active
    if(networkActive)
    {
        int c; // bytes actually written to the TLS connection

        printf("WRITING TO CLIENT\n");

        if(tls_conn != NULL) {
            int nob = sslWrite(tls_conn,data,len);
            if(nob<0){
              perror("Error writing to client!!");
            }
        }
        // Network is still active if we can write more then 0 bytes.
        networkActive = (c > 0);
    }
}

void handleCommand(void *conn, const char *buffer)
{
    // The first byte contains the command
    char cmd = buffer[1];
    uint32_t cmdParam[2];

    // Copy over the parameters.
    memcpy(cmdParam, &buffer[2], sizeof(cmdParam));

    TPacket commandPacket;

    commandPacket.packetType = PACKET_TYPE_COMMAND;
    commandPacket.params[0] = cmdParam[0];
    commandPacket.params[1] = 100;

    printf("COMMAND RECEIVED: %c %d %d\n", cmd, cmdParam[0], cmdParam[1]);

    switch(cmd)
    {
        case 'f':
        case 'F':
            commandPacket.command = COMMAND_FORWARD;
            uartSendPacket(&commandPacket);
            break;

        case 'b':
        case 'B':
            commandPacket.command = COMMAND_REVERSE;
            uartSendPacket(&commandPacket);
            break;

        case 'l':
        case 'L':
            commandPacket.command = COMMAND_TURN_LEFT;
            uartSendPacket(&commandPacket);
            break;

        case 'r':
```

```
        case 'R':
            commandPacket.command = COMMAND_TURN_RIGHT;
            uartSendPacket(&commandPacket);
            break;

        case 's':
        case 'S':
            commandPacket.command = COMMAND_STOP;
            uartSendPacket(&commandPacket);
            break;

        case 'c':
        case 'C':
            commandPacket.command = COMMAND_CLEAR_STATS;
            commandPacket.params[0] = 0;
            uartSendPacket(&commandPacket);
            break;

        case 'g':
        case 'G':
            commandPacket.command = COMMAND_GET_STATS;
            uartSendPacket(&commandPacket);
            break;

        default:
            printf("Bad command\n");
    }
}

void handleNetworkData(void *conn, const char *buffer, int len)
{
    // Assumes that Arduino's response is for the most recent client
    tls_conn = conn; // This is used by sendNetworkData

    if(buffer[0] == NET_COMMAND_PACKET)
        handleCommand(conn, buffer);
}

void *worker(void *conn)
{
    int len;
    char buffer[BUF_LEN];
    while(networkActive)
    {
        len = sslRead(conn,buffer,sizeof(buffer));
         if(len <0){
             perror("Error reading from network");
         }
        networkActive=(len > 0); // There is data, network active

        if(len > 0)
            handleNetworkData(conn, buffer, len);
        else
            if(len < 0)
                perror("ERROR READING NETWORK: ");
    }
    tls_conn = NULL; // Reset tls_conn to NULL.
    EXIT_THREAD(conn);
}
```

```
void sendHello()
{
    TPacket helloPacket; // Send a hello packet
    helloPacket.packetType = PACKET_TYPE_HELLO;
    uartSendPacket(&helloPacket);
}
```

A.1.4 Main Function (Line 296-324)

```
int main()
{
    pthread_t serThread;

    printf("\nALEX REMOTE SUBSYSTEM\n\n");

    printf("Opening Serial Port\n");
    // Open the serial port
    startSerial(PORT_NAME, BAUD_RATE, 8, 'N', 1, 5);
    printf("Done. Waiting 3 seconds for Arduino to reboot\n");
    sleep(3);

    printf("DONE. Starting Serial Listener\n");
    pthread_create(&serThread, NULL, uartReceiveThread, NULL);

    printf("Starting Alex Server\n");

    networkActive = 1;

      // Start network thread with client authentication and sending Alex's
certificate.
         createServer(KEY_FNAME,CERT_FNAME,PORTNUM,  &worker,  CA_CERT_FNAME,
CLIENT_NAME,1);

    printf("DONE. Sending HELLO to Arduino\n");
    sendHello();
    printf("DONE.\n");

    // Loop while the server is active
    while(server_is_running());
}
```

Appendix B - Code in Arduino

B.1 Alex.ino

B.1.1 Constants and Libraries Declaration (Line 1-16)

```
#include <serialize.h>
#include <stdarg.h>
#include "packet.h"
#include "constants.h"
#include <math.h>
#include <string.h>
#include <Adafruit_NeoPixel.h>

volatile TDirection dir;
#define PI 3.141592654
#define ALEX_LENGTH 25.7
#define ALEX_BREADTH 15
#define WHEEL_CIRC 6.6 * PI
#define COUNTS_PER_REV 3.3 // Ticks per revolution from the encoder
float alexDiagonal = sqrt((ALEX_LENGTH * ALEX_LENGTH) + (ALEX_BREADTH *
ALEX_BREADTH));
float alexCirc = alexDiagonal * PI;
```

B.1.2 Alex's Movement Configuration and Function Definition (Line 22-73)

```
// Encoder Ticks
volatile unsigned long leftForwardTicks;
volatile unsigned long leftReverseTicks;
volatile unsigned long leftForwardTicksTurns;
volatile unsigned long leftReverseTicksTurns;

// Forward and backward distance traveled
volatile unsigned long forwardDist;
volatile unsigned long reverseDist;

// To track changes
unsigned long deltaDist;
unsigned long newDist;
unsigned long deltaTicks;
unsigned long targetTicks;

// Initialize Alex's internal states
void initializeState()
{
  leftForwardTicks = 0;
  leftReverseTicks = 0;
  leftForwardTicksTurns = 0;
  leftReverseTicksTurns = 0;
  forwardDist = 0;
  reverseDist = 0;
}

// Ticks Computation for turning
unsigned long computeDeltaTicks(float ang) {
   unsigned long ticks = (unsigned long) ((ang * alexCirc * COUNTS_PER_REV) /
(360.0 * WHEEL_CIRC));
  return ticks;
}
```

```
void left(float ang, float speed) {
  if (ang == 0) {
    deltaTicks = 99999999;
  } else {
    deltaTicks = computeDeltaTicks(ang);
  }
  targetTicks = leftReverseTicksTurns + deltaTicks;
  ccw(ang, speed);
}

void right(float ang, float speed) {
  if (ang == 0) {
    deltaTicks = 99999999;
  } else {
    deltaTicks = computeDeltaTicks(ang);
  }
  targetTicks = leftForwardTicksTurns + deltaTicks;
  cw(ang, speed);
}
```

B.1.3 Encoder Configuration and ISR Definition (Line 78-110)

```
void enablePullups() // Enable pull up resistors on pin 18 (Left Encoder)
{
  DDRD = 0;
  PORTD = 0b00001000; // Set Pin 18 to input
}

void setupEINT() // For Pin 18
{
  EICRA = 0b10000000; // Set to falling edge
  EIMSK = 0b00001000;
}

ISR(INT3_vect) { // Pin 18
  leftISR();
}

void leftISR() // Functions to be called by INT3 ISRs.
{
  if (dir == FORWARD) {
    leftForwardTicks++;
    forwardDist = (unsigned long) ((float) leftForwardTicks / COUNTS_PER_REV *
WHEEL_CIRC);
  }
  if (dir == BACKWARD) {
    leftReverseTicks++;
    reverseDist = (unsigned long) ((float) leftReverseTicks / COUNTS_PER_REV *
WHEEL_CIRC);
  }
  if (dir == LEFT) {
    leftReverseTicksTurns++;
  }
  if (dir == RIGHT) {
    leftForwardTicksTurns++;
  }
}
```

B.1.4 Color Sensor Configuration and Function Definition (Line 116-153)

```
#define colorOut 20
```

```
volatile unsigned long color;
volatile unsigned long colorId[3];

void setColor() {
  DDRA = 0b00001111;
  DDRD = 0;
  PORTA = 0b00000001;
}

unsigned long checkColor() {
  strip.setBrightness(0);
  strip.show();
  PORTA = 0b00000001;
  colorId[0] = pulseIn(colorOut, LOW);
  delay(100);
  PORTA = 0b00001101;
  colorId[1] = pulseIn(colorOut, LOW);
  delay(100);
  PORTA = 0b00001001;
  colorId[2] = pulseIn(colorOut, LOW);
  delay(100);
  strip.setBrightness(8);
  strip.show();
  for (int i = 0; i < 3; i++) {
    if (colorId[i] > 2000) {
      if (colorId[0] < colorId[1] && colorId[0] < colorId[2]) {
        colorWipe(strip.Color(255, 0, 0), 50);
        return 0; // Red
      } else {
        colorWipe(strip.Color(0, 255, 0), 50);
        return 1; // Green
      }
    }
  }
  colorWipe(strip.Color(255, 255, 255), 50);
  return 2; // White
}
```

B.1.5 Ultrasonic Configuration and Function Definition (Line 159-171)

```
void setUltrasonic() {
  DDRA |= 0b00010000;
}

unsigned long checkUltrasonic() {
  PORTA &= ~(1<<4);
  delayMicroseconds(2);
  PORTA |= (1<<4);
  delayMicroseconds(10);
  PORTA &= ~(1<<4);
  long duration = pulseIn(27, HIGH);
  return duration * 0.034 / 2;
}
```

B.1.6 Ring Light Configuration and Function Definition (Line 177-203)

```
#define RING 31
Adafruit_NeoPixel strip = Adafruit_NeoPixel(16, RING, NEO_GRB + NEO_KHZ800);
```

```
void setRing() { // Setup
  strip.begin();
  strip.show(); // Initialize all pixels to 'off'
  strip.setBrightness(8);
}

void colorWipe(uint32_t c, uint8_t wait) { // To display a fixed color
  for(uint16_t i=0; i<strip.numPixels(); i++) {
      strip.setPixelColor(i, c);
      strip.show();
      delay(wait);
  }
}

void rainbowCycle() { // To display all colors on ring light
  uint16_t i, j;

  for(j=0; j<256*5; j++) { // 5 cycles of all colors on wheel
    for(i=0; i< strip.numPixels(); i++) {
         strip.setPixelColor(i, Wheel(((i * 256 / strip.numPixels()) + j) &
255));
    }
    strip.show();
  }
}
```

B.1.7 Setup and start codes for serial communications (Line 209-232)

```
void setupSerial() // Set up the serial connection
{
  Serial.begin(9600); // To replace later with bare-metal
  // UBRR3L = 103;
}

void startSerial() // Start the serial connection
{
  // Using Arduino Wiring
  // To be replaced with bare-metal code
}

int readSerial(char *buffer) // Read the serial port
{
  int count = 0;
   while (Serial.available()) // This will be replaced later with bare-metal
code.
    buffer[count++] = Serial.read();
  return count;
}

void writeSerial(const char *buffer, int len) // Write to the serial port
{
  Serial.write(buffer, len); // Replaced later with bare-metal code
}
```

B.1.8 Alex Communication Routines with R-Pi (Line 238-416)

```
void handleCommand(TPacket *command)
{
  switch (command->command)
  {
```

```
    // For movement commands, param[0] = distance, param[1] = speed.
    case COMMAND_FORWARD:
      forward((double) command->params[0], (float) command->params[1]);
      break;
    case COMMAND_TURN_LEFT:
      left((double) command->params[0], (float) command->params[1]);
      break;
    case COMMAND_TURN_RIGHT:
      right((double) command->params[0], (float) command->params[1]);
      break;
    case COMMAND_REVERSE:
      backward((double) command->params[0], (float) command->params[1]);
      break;
    case COMMAND_STOP:
      stop();
      break;
    case COMMAND_GET_STATS:
      color = checkColor();
      sendStatus();
    default:
      sendBadCommand();
  }
}

void handlePacket(TPacket *packet)
{
  switch (packet->packetType)
  {
    case PACKET_TYPE_COMMAND:
      handleCommand(packet);
      break;

    case PACKET_TYPE_RESPONSE:
      break;

    case PACKET_TYPE_ERROR:
      break;

    case PACKET_TYPE_MESSAGE:
      break;

    case PACKET_TYPE_HELLO:
      break;
  }
}

void waitForHello()
{
  int exit = 0;
  while (!exit)
  {
    TPacket hello;
    TResult result;
    do
    {
      result = readPacket(&hello);
    } while (result == PACKET_INCOMPLETE);

    if (result == PACKET_OK)
    {
```

```
      if (hello.packetType == PACKET_TYPE_HELLO)
      {
        sendOK();
        exit = 1;
      }
      else
        sendBadResponse();
    }
    else if (result == PACKET_BAD)
    {
      sendBadPacket();
    }
    else if (result == PACKET_CHECKSUM_BAD)
      sendBadChecksum();
  } // !exit
}

TResult readPacket(TPacket *packet)
{
  char buffer[PACKET_SIZE];
  int len = readSerial(buffer); // Reads in data from the serial port
  if (len == 0)
    return PACKET_INCOMPLETE;
  else
    return deserialize(buffer, len, packet); // deserializes to packet
}

void sendStatus()
{
  TPacket statusPacket;
  statusPacket.packetType = PACKET_TYPE_RESPONSE;
  statusPacket.command = RESP_STATUS;
  statusPacket.params[0] = leftForwardTicks;
  statusPacket.params[1] = leftReverseTicks;
  statusPacket.params[2] = leftForwardTicksTurns;
  statusPacket.params[3] = leftReverseTicksTurns;
  statusPacket.params[4] = forwardDist;
  statusPacket.params[5] = reverseDist;
  statusPacket.params[6] = colorId[0];
  statusPacket.params[7] = colorId[1];
  statusPacket.params[8] = colorId[2];
  statusPacket.params[9] = color;
  statusPacket.params[10] = checkUltrasonic();
  sendResponse(&statusPacket);
}

// For debugging (To PC), not used in mission
void sendMessage(const char *message)
{
  TPacket messagePacket;
  messagePacket.packetType = PACKET_TYPE_MESSAGE;
  strncpy(messagePacket.data, message, MAX_STR_LEN);
  sendResponse(&messagePacket);
}

// For debugging (In Arduino), not used in mission
void dbprintf(char* format, ...) {
  va_list args;
  char buffer[128];
  va_start(args, format);
```

```
  vsprintf(buffer, format, args);
  sendMessage(buffer);
}

void sendBadPacket() // Wrong magic number
{
  TPacket badPacket;
  badPacket.packetType = PACKET_TYPE_ERROR;
  badPacket.command = RESP_BAD_PACKET;
  sendResponse(&badPacket);

}

void sendBadChecksum() // Wrong Checksum
{
  TPacket badChecksum;
  badChecksum.packetType = PACKET_TYPE_ERROR;
  badChecksum.command = RESP_BAD_CHECKSUM;
  sendResponse(&badChecksum);
}

void sendBadCommand() // Invalid Command
{
  TPacket badCommand;
  badCommand.packetType = PACKET_TYPE_ERROR;
  badCommand.command = RESP_BAD_COMMAND;
  sendResponse(&badCommand);
}

void sendBadResponse()
{
  TPacket badResponse;
  badResponse.packetType = PACKET_TYPE_ERROR;
  badResponse.command = RESP_BAD_RESPONSE;
  sendResponse(&badResponse);
}

void sendOK()
{
  TPacket okPacket;
  okPacket.packetType = PACKET_TYPE_RESPONSE;
  okPacket.command = RESP_OK;
  sendResponse(&okPacket);
}

void sendResponse(TPacket *packet)
{
  char buffer[PACKET_SIZE];
  int len = serialize(buffer, packet, sizeof(TPacket)); // Serialise Packet
  writeSerial(buffer, len); // Send to R-Pi
}
```

B.1.9 Main Function (Line 417-509)

```
void setup() {
  cli();
  setupEINT();
  setColor();
  setUltrasonic();
  setupSerial();
```

```
  startSerial();
  setRing();
  enablePullups();
  initializeState();
  rainbowCycle();
  sei();
}

void loop() {

  TPacket recvPacket; // This holds commands from the Pi
  TResult result = readPacket(&recvPacket);

  if (result == PACKET_OK) {
    handlePacket(&recvPacket);
  }
  else if (result == PACKET_BAD)
  {
    sendBadPacket();
  }
  else if (result == PACKET_CHECKSUM_BAD)
  {
    sendBadChecksum();
  }

  if (deltaDist > 0)
  {
    if (dir == FORWARD)
    {
      unsigned long distance = checkUltrasonic();
      delay(100);
      if ((distance <= 10 && distance != 0) || forwardDist > newDist)
      {
        deltaDist = 0;
        newDist = 0;
        stop();
        sendStatus();
      }
    }
    else if (dir == BACKWARD)
    {
      if (reverseDist > newDist)
      {
        deltaDist = 0;
        newDist = 0;
        stop();
        sendStatus();
      }
    }
    else if (dir == STOP)
    {
      deltaDist = 0;
      newDist = 0;
      stop();
      sendStatus();
    }
  }

  if (deltaTicks > 0) {
    if (dir == LEFT) {
```

```
      if (leftReverseTicksTurns >= targetTicks) {
        deltaTicks = 0;
        targetTicks = 0;
        stop();
        sendStatus();
      }
    }
    else {
      if (dir == RIGHT) {
        if (leftForwardTicksTurns >= targetTicks) {
          deltaTicks = 0;
          targetTicks = 0;
          stop();
          sendStatus();
        }
      }
      else {
        if (dir == STOP) {
          deltaTicks = 0;
          targetTicks = 0;
          stop();
          sendStatus();
        }
      }
    }
  }
}
```

Appendix C - Code in PC

C.1 tls-alex-client.cpp

C.1.1 Constants and Libraries Declaration (Line 1-13)

```
#include "make_tls_client.h" // Routines to create a TLS client
#include "netconstants.h" // Network packet types
#include "constants.h" // Packet types, error codes, etc.

static volatile int networkActive=0; // To check if network is running.

// Parameters and Filenames for TLS Handshake
#define SERVER_NAME "192.168.139.41" // R-Pi IP address
#define PORT_NUM 5001
#define CA_CERT_FNAME "signing.pem" // CA certificate name
#define CLIENT_CERT_FNAME "laptop.crt" // PC's certificate
#define CLIENT_KEY_FNAME "laptop.key" // PC's private key
#define SERVER_NAME_ON_CERT "signer.com" // R-Pi's name
```

C.1.2 Network Routines with R-Pi (Line 19-139)

```
void handleError(const char *buffer)
{
    switch(buffer[1])
    {
        case RESP_OK:
            printf("Command / Status OK\n");
            break;

        case RESP_BAD_PACKET:
            printf("BAD MAGIC NUMBER FROM ARDUINO\n");
            break;

        case RESP_BAD_CHECKSUM:
            printf("BAD CHECKSUM FROM ARDUINO\n");
            break;

        case RESP_BAD_COMMAND:
            printf("PI SENT BAD COMMAND TO ARDUINO\n");
            break;

        case RESP_BAD_RESPONSE:
            printf("PI GOT BAD RESPONSE FROM ARDUINO\n");
            break;

        default:
            printf("PI IS CONFUSED!\n");
    }
}

void handleStatus(const char *buffer)
{
    int32_t data[16];
    memcpy(data, &buffer[1], sizeof(data));

    printf("\n ------- ALEX STATUS REPORT ------- \n\n");
    printf("Left Forward Ticks:\t\t%d\n", data[0]);
    printf("Left Reverse Ticks:\t\t%d\n", data[1]);
    printf("Left Forward Ticks Turns:\t%d\n", data[2]);
    printf("Left Reverse Ticks Turns:\t%d\n", data[3]);
    printf("Forward Distance:\t\t%d\n", data[4]);
```

```
    printf("Reverse Distance:\t\t%d\n", data[5]);
    printf("Red:\t\t%d\n", data[6]);
    printf("Green:\t\t%d\n", data[7]);
    printf("Blue:\t\t%d\n", data[8]);
    printf("Colour:\t\t%d\n", data[9]);
    printf("Distance:\t\t%d\n", data[10]);
    printf("\n---------------------------------------\n\n");
}

// For debugging, not used in mission
void handleMessage(const char *buffer)
{
    printf("MESSAGE FROM ALEX: %s\n", &buffer[1]);
}

void handleNetwork(const char *buffer, int len)
{
    int type = buffer[0]; // The first byte is the packet type
    switch(type)
    {
        case NET_ERROR_PACKET:
        handleError(buffer);
        break;

        case NET_STATUS_PACKET:
        handleStatus(buffer);
        break;

        case NET_MESSAGE_PACKET:
        handleMessage(buffer);
        break;

        case NET_COMMAND_PACKET:
        handleCommand(buffer);
        break;
    }
}

void sendData(void *conn, const char *buffer, int len)
{
    int c;
    printf("\nSENDING %d BYTES DATA\n\n", len);

    if(networkActive)
    {
        int count = sslWrite(conn,buffer,len);
        if(count<0){
           perror("Error writing to server");
        }
        networkActive = (c > 0);
    }
}

void *readerThread(void *conn)
{
    char buffer[128];
    int len;

    while(networkActive)
    {
```

```
        len = sslRead(conn,buffer,sizeof(buffer));
        if(len<0){
            perror("Error reading from server");
        }
        printf("read %d bytes from server.\n", len);

        networkActive = (len > 0);
        if(networkActive)
            handleNetwork(buffer, len);
    }
    printf("Exiting network listener thread\n");

    stopClient();
    EXIT_THREAD(conn);
    return NULL;
}

void connectToServer(const char *serverName, int portNum)
{

createClient(SERVER_NAME,PORT_NUM,1,CA_CERT_FNAME,SERVER_NAME_ON_CERT,1,CLIENT
_CERT_FNAME,CLIENT_KEY_FNAME, readerThread,writerThread);
}
```

C.1.3 Scanning Operators' Command (Line 145-214)

```
void flushInput()
{
    char c;
    while((c = getchar()) != '\n' && c != EOF);
}

void getParams(int32_t *params)
{
    printf("Enter distance/angle in cm/degrees (e.g. 50) and power in %% (e.g.
75) separated by space.\n");
     printf("E.g. 50 75 means go at 50 cm at 75%% power for forward/backward,
or 50 degrees left or right turn at 75%%  power\n");
    scanf("%d", &params[0]);
    params[1] = 100;
    flushInput();
}

void *writerThread(void *conn)
{
    int quit=0;

    while(!quit)
    {
        char ch;
           printf("Command (f=forward, b=reverse, l=turn left, r=turn right,
s=stop, c=clear stats, g=get stats q=exit)\n");
        scanf("%c", &ch);

        // Purge extraneous characters from input stream
        flushInput();

        char buffer[10];
        int32_t params[2];
```

```
        buffer[0] = NET_COMMAND_PACKET;
        switch(ch)
        {
            case 'f':
            case 'F':
            case 'b':
            case 'B':
            case 'l':
            case 'L':
            case 'r':
            case 'R':
                        getParams(params);
                        buffer[1] = ch;
                        memcpy(&buffer[2], params, sizeof(params));
                        sendData(conn, buffer, sizeof(buffer));
                        break;
            case 's':
            case 'S':
            case 'g':
            case 'G':
                    params[0]=0;
                    params[1]=0;
                    memcpy(&buffer[2], params, sizeof(params));
                    buffer[1] = ch;
                    sendData(conn, buffer, sizeof(buffer));
                    break;
            case 'q':
            case 'Q':
                quit=1;
                break;
            default:
                printf("BAD COMMAND\n");
        }
    }
    printf("Exiting keyboard thread\n");
    stopClient();
    EXIT_THREAD(conn);
    return NULL;
}
```

C.1.4 Main Function (Line 220-234)

```
int main(int ac, char **av)
{
    if(ac != 3)
    {
        fprintf(stderr, "\n\n%s <IP address> <Port Number>\n\n", av[0]);
        exit(-1);
    }

    networkActive = 1;
    connectToServer(av[1], atoi(av[2]));

    while(client_is_running()); // Prevent main from exiting

    printf("\nMAIN exiting\n\n");
}
```

Appendix D - Component Design

High Level Steps:

1. Initialization
2. Send command from PC
3. Arduino carries out the command
4. Repeat step 2 until mission objectives are achieved

Step 1. Initialization

a. R-Pi set up TLS server, PC setup TLS client.
b. R-Pi performs handshake with Arduino by sending a hello packet through UART.
c. Arduino polls till a hello packet is received correctly and responds by sending an ok packet.

Step 2. Send command from PC

a. Based on the RViz and the ultrasonic sensor reading in the status packet received, the operator decides the direction and distance of Alex's next movement.
    i. If Alex is facing and within 2 cm of an object, the user sends an get status command ("g") for Alex to detect the color of the object.
    ii. Else, the user sends a movement command to navigate Alex towards the object with additional parameters such as distance or angle.
b. PC (client) sends this information to the R-Pi (server) through TLS, which will be serialized into a command packet and sent to Arduino through UART.

Step 3. Arduino carries out the command

a. Arduino receives and deserializes the command packet.
    i. If it is a movement command, the motor will move accordingly towards the direction.
        1. Left/Right: Compute deltaTicks and targetTicks.
            - As the wheels turn, ISR updates left/right forward and reverse tick turns. Alex will stop once targetTicks is reached.
        2. Forward/Backward: Compute deltaDist and newDist.
            - As the wheels turn, ISR updates forward/backward ticks and dist. Alex will stop once newDist is reached.
    ii. If it is an object identification command, Alex takes in RGB data inputs from the color sensor and updates the data in the status packet.
b. Arduino takes in input from the ultrasonic sensor and stores the data in the status packet.
c. Arduino will send the serialized status packet to R-Pi through UART, which is then deserialized by R-Pi and sent to PC through TLS.

Step 4. Repeat step 2 until mission objectives are achieved

Appendix E - Hector SLAM

Hector SLAM is used in robotics for mapping unknown environments and simultaneously localizing the robot within that map.

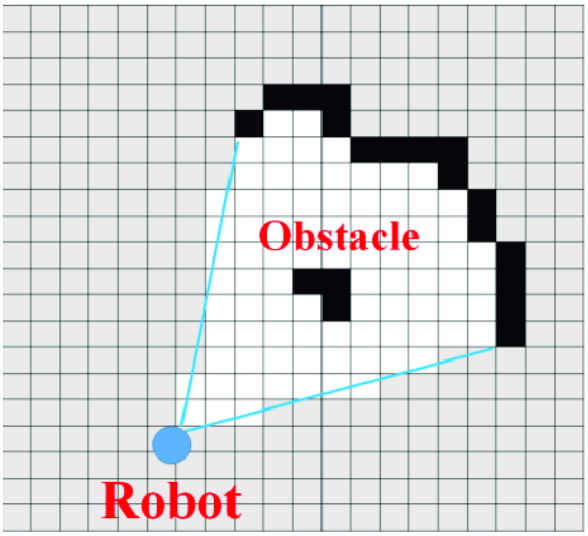
The breakdown of how Hector SLAM work is as follows:

1) Initialisation: Hector SLAM begins with an initial estimate of the robot's pose (position and orientation) relative to the environment based on the LiDAR data.

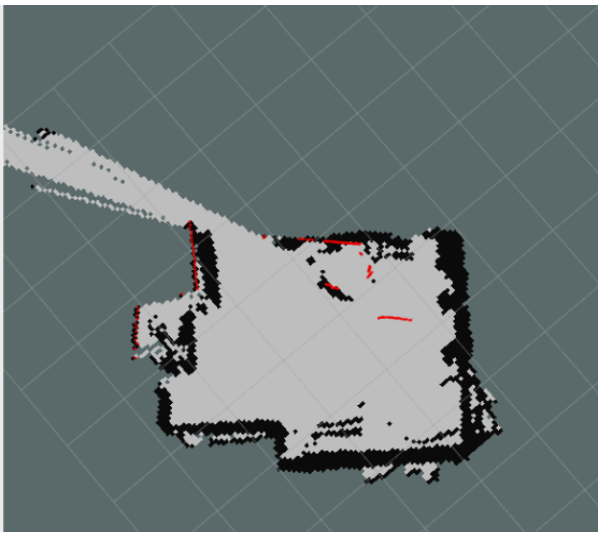
2) Mapping: As Alex traverses the environment, Hector SLAM continuously integrates incoming LiDAR measurements to build a 2D occupancy grid map. The occupancy grid map represents the likelihood of each grid cell being occupied by an obstacle, allowing Alex to navigate safely while avoiding collisions. In addition to obstacle detection, Hector SLAM also captures information about open spaces, enabling Alex to plan efficient paths through the environment.

3) Localisation: Hector SLAM employs sophisticated scan matching techniques to accurately localize Alex within the environment. By comparing consecutive LiDAR scans, Hector SLAM estimates Alex's motion (both translation and rotation) between scans. This motion estimate is used to update Alex's pose within the map, ensuring that the robot's position and orientation are accurately represented in real-time.

4) Optimisation: When Alex revisits previously explored areas, Hector SLAM detects loop closures by identifying similar features in the environment. Loop closures are crucial for correcting accumulated errors in the robot's pose estimate and improving the consistency of the map. After detecting loop closures, Hector SLAM performs optimization to refine both the map and Alex's trajectory, ensuring alignment with loop closure constraints and minimizing discrepancies.



Occupancy Grid Map
Source: Scientific Figure on ResearchGate



RViz with Hector SLAM