MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

Mini Project Report
of
Computer Networks LAB

Telemetry Data Pipeline for Unmanned Ground Vehicles

SUBMITTED
BY

NAME	REG NO	SECTION	ROLL NO
1. ADITYA A IYER	200905100	B	22
2. AYUSH DIXIT	200905112	B	23

Computer Networks LAB-MINI PROJECT REPORT AND GUIDELINES

NOTE: ATTACHED TITLE PAGE FORMAT AND OTHER DETAILED GUIDELINES ALONG WITH THIS MAIL

The following are the guidelines to be followed while preparing the CN LAB Mini project reports.

The mini project report around 10-15 pages in length should be submitted in Hard copy to concerned CN theory faculties during mini project evaluation. The mini project report should contain the following chapters

x. Title page with Title, Authors, Affiliations (1st Page)

x. Abstract (2nd Page)

1. Introduction

1.1 General Introduction to the topic

1.2 Hardware and Software Requirements

2. Problem definition

3. Objectives

4. Methodology

5. Implementation details

6. Contribution summary (divide the work and list who is

responsible for which part)

7. References

ABSTRACT

Telemetry is the in-situ collection of measurements or other data at remote points and their automatic transmission to receiving equipment (telecommunication) for monitoring. Telemetry is commonly used in Meteorology, Oil and Gas industry, Motor Racing and Transportation and Agriculture.

A telemeter is a physical device used in telemetry. It consists of a sensor, a transmission path, and a display, recording, or control device.In our project we demonstrate the application of telemetry in the domain of robotics.

In this context, Telemetry is any data that travels from an Unmanned Ground Vehicle(UGV) to a base station located remotely from the current position.

Telemetry can give information about the UGV's position and heading,
it can tell how instruments or other parts of the ship are functioning or can downlink scientific data gathered by the UGV and its onboard systems.
We implement the concept of developing an application and transport layer through a TCP/IP connection between a remote server and a base station.

The issue of data acquisition has been a major problem in autonomous systems for ages.The primary benefit of telemetry data is that it provides a deep understanding of the real-world performance of whatever device or application data source chooses to focus on – although, only after the raw data is analyzed and turned into actionable insights,

Recording and viewing telemetry data is a crucial part of the engineering process - accurate telemetry data helps you tune your robot to perform optimally, and is indispensable for debugging your robot when it fails to perform as expected.

Additionally, video telemetry plays a crucial role in understanding the operations of your robots requires more than staring at a wall of telemetry charts. Seeing is understanding. Using our eyes is often necessary to answer the critical questions about how to optimize a device for maximum effectiveness. In particular, video telemetry is a powerful information source used to understand the context in which you're operating. we've worked to maximize the utilization of visual recognition by implementing concepts of Computer Networks specifically tailored for video telemetry in robotics.

Some of the motives for choosing this project include :
As common as data is in our modern world on our phones and personal computers, robots historically have been limited to slow single data either by limitations of tools or device power. For robots that did have the power to do more, the owner was faced with a non-ideal trade off:

1) Capture the information on device and have to wait long after the robots operation to view what happened
2) Capture more images and have to ingest large amounts of data and take on the cost burden of storage

3) Accept capturing less data and lose real-time context

Thus this project aims to achieve all of this using the concepts of Computer Networks. We have implemented a data packet creation backend curated and optimised to our need and requirement.

1. Introduction

1.1 General Introduction to the topic

A TCP/IP connection over the internet to being used to transfer data from the remote telemetry unit to the base station.
In networking, the most used model is the 5-layer model.
The 5-Layer Networking Model consists of the following:

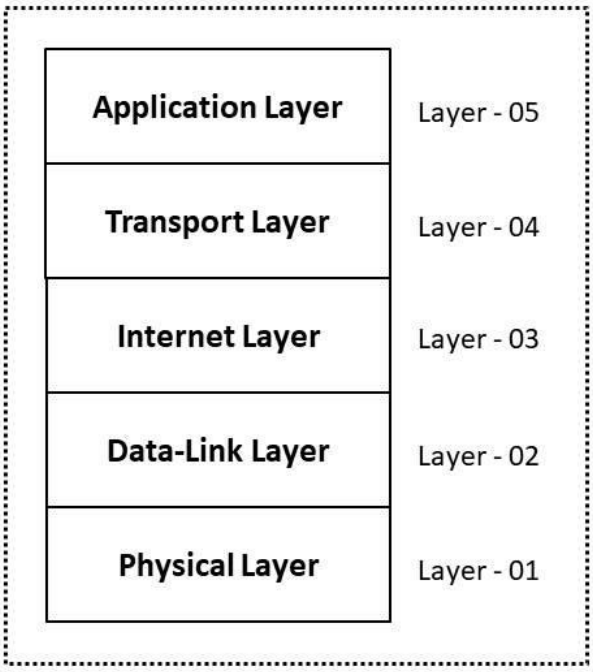


Fig 1 : OSI Model

1.Physical Layer:

This layer comprises of the Cat-6 cables (category 6, other variations are Cat-5 and Cat-5e) used to send or receive the encapsulated Ethernet frame that comprises of IP datagram and TCP segment and the Network ports to which the cables are connected to determine the connections between devices through LED's (Link LED and Activity LED). base station and remote server
This layer is a carrying path.

2.Data Link Layer:

This is the layer where connection between nodes(devices) in a network is ensured through switches (like hubs (physical layer component, but prone to collision domain) but more reliable than them in aspect of collision domain as it uses Ethernet Protocol to efficiently send data packets i.e., Ethernet frames in this layer). Ethernet protocol makes sure the node's identity (i.e. MAC address-Hardware address) to send the data packets.

3.Network Layer:

This layer uses IP (Internet Protocol) predominantly to find the correct network where the destination node is present through Routers which connects. Networks are split up into subnets by subnetting process and are represented using CIDR ID to represent the networks. Routers use routing tables to find the destination network and send the datagrams to the appropriate routers that has shorter distance to the destination IP.

4. Transport Layer:

This layer uses TCP (Transfer Control Protocol)/UDP (User Datagram Protocol). TCP makes connections to servers through system ports and to clients through ephemeral ports. Multiplexing and De-multiplexing processes are made through these ports. Then it uses acknowledgements to make sure the connection is in the proper state every time; communication is made between the nodes(devices), which leads to extra traffic but is necessary for important data transfers like phone calls. If the connection between devices is achieved on both sides, Full-Duplex is achieved, or vice versa called Simplex is achieved. But UDP does not make any acknowledgements, and it is suitable for faster transfer with a danger of data loss which does not matter a lot in the case of video streaming and radio listening. Acknowledgements are made through activity flags present in the segment fields. The Three-way handshake is used for initiating a connection, and the Four-way handshake is used to terminate a connection.

5.Application Layer:

This layer uses various protocols depending on the applications. For example, HTTP (Hyper Text Transfer Protocol) is used by web servers and webpages. In the case of the OSI model, this layer is split into the Session layer, Presentation Layer and Application Layer.

We will now define these layers in the context of this project:

In the application layer, the telemetry unit communicates through a message which is a string of an object of a class which is a custom-defined data packet curated for the application of telemetry.
Port numbers can be configured.
Error Checking is performed on the size of the packets.
ACK, RST, FIN and URG header values configure the connection state.

Transport Layer:
In the transport layer we use TCP to transfer sensor data and UDP to transfer the video feed.

Main features of TCP and UDP protocols

User Datagram Protocol (UDP)	Transmission Control Protocol (TCP)
Connectionless	Connection-oriented
“Best Effort” delivery	Reliable delivery
Low overhead	Error checking
No error checking, No flow control	Flow control

TCP Protocol
The Transmission Control Protocol (TCP) is a transport protocol that is used on top of IP to ensure reliable transmission of packets.

When sending packets using TCP/IP, the data portion of each IP packet is formatted as a TCP segment.
Each TCP segment contains a header and data. The TCP header contains many more fields than the UDP header and can range in size from 20 to 60 bytes, depending on the size of the options field.

Process of transmitting a packet with TCP/IP:

1. Establish connection

   The first computer sends a packet with the SYN bit set to 111 (SYN The second computer sends back a packet with the ACK bit set to 111 (ACK) plus the SYN bit set to 111. The first computer replies with an ACK. The SYN and ACK bits are both part of the TCP header:

2. Send packets of data

   The first computer sends a packet with data and a sequence number. The second computer acknowledges it by setting the ACK bit and increasing the acknowledgement number by the length of the received data. The sequence and acknowledgement numbers are part of the TCP header

3. Close the connection

   Either computer can close the connection when they no longer want to send or receive data. A computer initiates closing the connection by sending a packet with the FIN bit set to 1 (FIN = finish). The other computer replies with an ACK and another FIN. After one more ACK from the initiating computer, the connection is closed.

TCP header structure
TCP wraps each data packet with a header fields totaling 20 bytes (or octets). Each header holds information about the connection and the current data being sent.

The relevant header fields are as follows:

- **Source port** – The sending device's port.
- **Destination port** – The receiving device's port.
- **Sequence number** – A device initiating a TCP connection must choose a random initial sequence number, which is then incremented according to the number of transmitted bytes.
- **Acknowledgment number** – The receiving device maintains an acknowledgment number starting with zero. It increments this number according to the number of bytes received.

User Datagram Protocol (UDP)
User datagram protocol (UDP) operates on top of the Internet Protocol (IP) to transmit datagrams over a network. UDP does not require the source and destination to establish a three-way handshake before transmission takes place. Additionally, there is no need for an end-to-end connection.

Since UDP avoids the overhead associated with connections, error checks and the retransmission of missing data, it's suitable for real-time or high-performance applications that don't require data verification or correction. If verification is needed, it can be performed at the application layer.

UDP is commonly used for Remote Procedure Call (RPC) applications, although RPC can also run on top of TCP. RPC applications need to be aware they are running on UDP, and must then implement their own reliability mechanisms.

UDP header structure
UDP wraps datagrams with a UDP header, which contains four fields totaling eight bytes. The fields in a UDP header are:

- **Source port** – The port of the device sending the data. This field can be set to zero if the destination computer doesn't need to reply to the sender.

- **Destination port** – The port of the device receiving the data. UDP port numbers can be between 0 and 65,535.
- **Length** – Specifies the number of bytes comprising the UDP header and the UDP payload data. The limit for the UDP length field is determined by the underlying IP protocol used to transmit the data.
- **Checksum** – The checksum allows the receiving device to verify the integrity of the packet header and payload. It is optional in IPv4

Robot Operating System (ROS) :

ROS has been used as a middleware between the raw data obtained from the sensors and the data pipeline sending the values across to the basestation. ROS has been used to implement 2 transport layer protocols : TCPROS and UDPROS.

In **TCPROS**, Inbound connections to a TCPROS Server Socket are routed by information contained within the header fields. If the header contains the 'topic' field, it will be routed as a connection to a ROS Publisher. If it contains a 'service' field, it will be routed as a connection to a ROS Service.
A TCPROS service client is required to send the following fields:
callerid: node name of service client
service: name of the topic the subscriber is connecting to

md5sum: md5sum of the message type
type: service type

UDPROS is a transport layer (in development) for ROS Messages and Services. It uses standard UDP datagram packets to transport serialized message data. The UDPROS transport is useful when latency is more important than reliable transport. Examples in the robotics domain include teleoperation as well as streaming audio.

ROS messages are sent across the network as a series of UDPROS datagrams. The maximum size of the UDPROS datagram is negotiated in the XML-RPC connection (see above).

Each UDP datagram contains a UDPROS header, followed by message data. The format of the header is as follows:

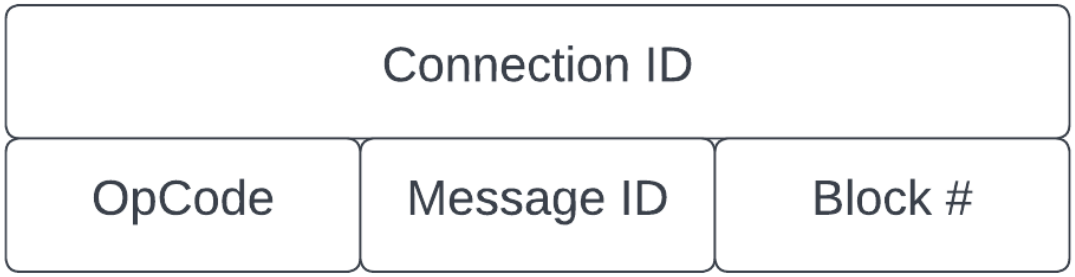


Fig 2 : UDPROS datagram format

- Connection ID - This 32-bit value is determined during connection negotiation and is used to denote which connection the datagram is destined for. This parameter allows a single socket to service multiple UDPROS connections.
- Opcode - UDPROS supports multiple datagram types. The opcode specifies the datagram type. This 8-bit field can have the following possible values:
    - DATA0 (0) - This opcode is sent in the first datagram of a ROS message
    - DATAN (1) - All subsequent datagrams of a ROS message use this opcode
    - PING (2) - A heartbeat packet is sent periodically to let the other side know that the connection is still alive
    - ERR (3) - An error packet is used to signal that a connection is closing unexpectedly
- Message ID - This 8-bit value is incremented for each new message and is used to determine if datagrams have been dropped.
- Block # - When the opcode is DATA0, this field contains the total number of UDPROS datagrams expected to complete the ROS message. When the opcode is DATAN, this field contains the current datagram number.

ROS Parameter Server :

Like the Master API, the Parameter Server API is also implemented via XMLRPC. The use of XMLRPC enables easy integration with the ROS client libraries and also provides greater type flexibility when storing and retrieving data. The Parameter Server can store basic XML-RPC scalars (32-bit integers, booleans, strings, doubles, iso8601 dates), lists, and base64-encoded binary data. The Parameter Server can also store dictionaries (i.e. structs), but these have a special meaning. The Parameter Server uses a dictionary-of-dictionary representation for namespaces, where each dictionary represents a level in the naming hierarchy. This means that each key in a dictionary represents a namespace. If a value is a dictionary, the Parameter Server assumes that it is storing the values of a namespace.

1.2 Hardware and Software Requirements

Hardware Requirments :

1. Raspberry Pi Onboard Computer
2. BNO080 9-DoF(Degree of Freedom) Inertial Measurement Unit
3. NeoMP8 RTK GPS
4. Li-Po Battery (12V,3000 Mah)
5. Arduino Uno Microcontroller
6. L298N Dual H-Bridge Motor Driver
7. 12V DC motors
8. Stainless Steel Frame chassis
9. Jumper Wire

Software Requirments :

1. Python sockets
2. PyQt GUI
3. OpenCV Image Processing Modules
4. pickle library
5. Robot Operating System

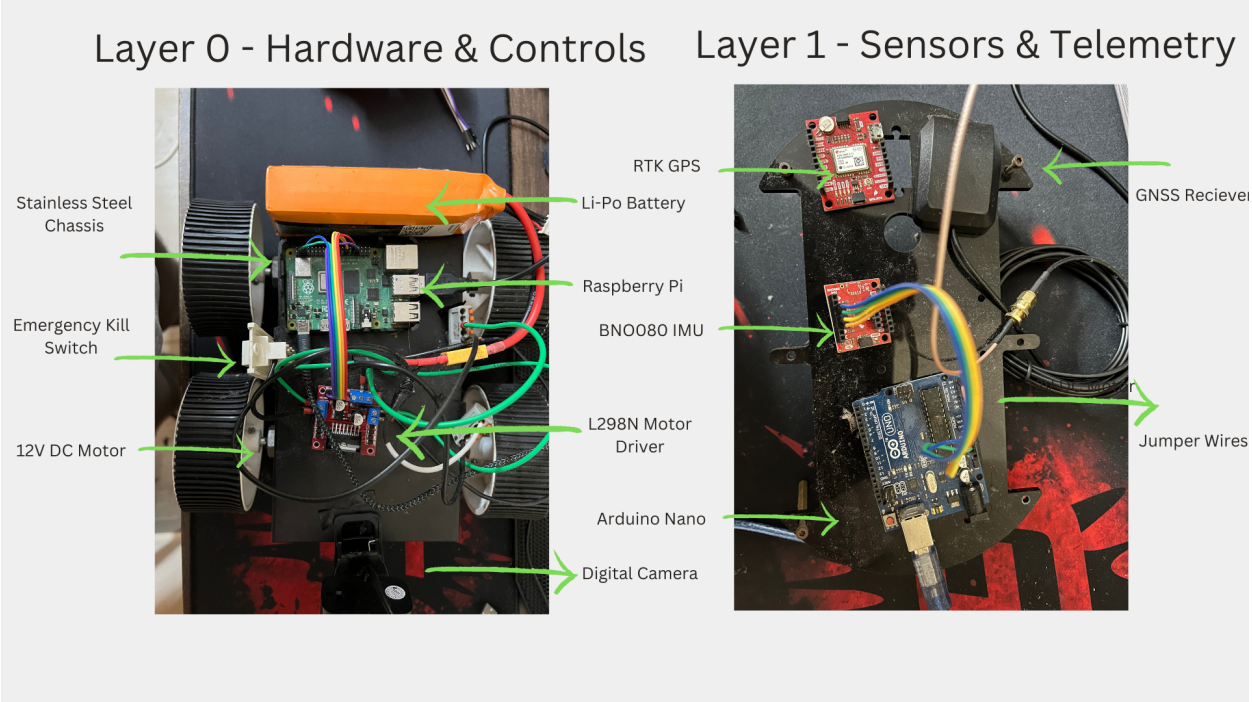


Fig 3 : Hardware Implementation

2. Problem definition

The problem statement aims to achieve effective transfer of data from a remote server to a base station operated by the user. Effective data communications between a remote station and a base station over limited bandwidth and using limited resources on the onboard computer is the problem trying to be solved by this implementation of a novel transport and application layer protocol curated specifically for Unmanned Ground Vehicles.

3. Objectives

There are 3 main goals of this project :

1) Achieve data pipeline between a remote server and a base station using multiple threads to communicate a multitude of data concurrently.
2) Enable the sending of commands for locomotion remotely from the base station using sensor data being sent on the image pipeline
3) Achieve fast image transfer between the remote server and the base station in real-time, enabling better perception of the UGV.

The sub-goals for each aspect of the data pipeline include :

1. **Simple Data Structures:**
    - The data is bing sent only after the connection is established between the remote server and the base station

- Delivery of data should be guaranteed from the receiver after the establishment of connection
- Error Checking should be performed

2. **For Video Feed:**
   - Small size data packets
   - Fast transmission

4. Methodology

The methodology followed is complex and involves multiple layers of encapsulation embedded within the Object-Oriented approach selected for this project. The data pipeline we wish to achieve is explained in Fig _.

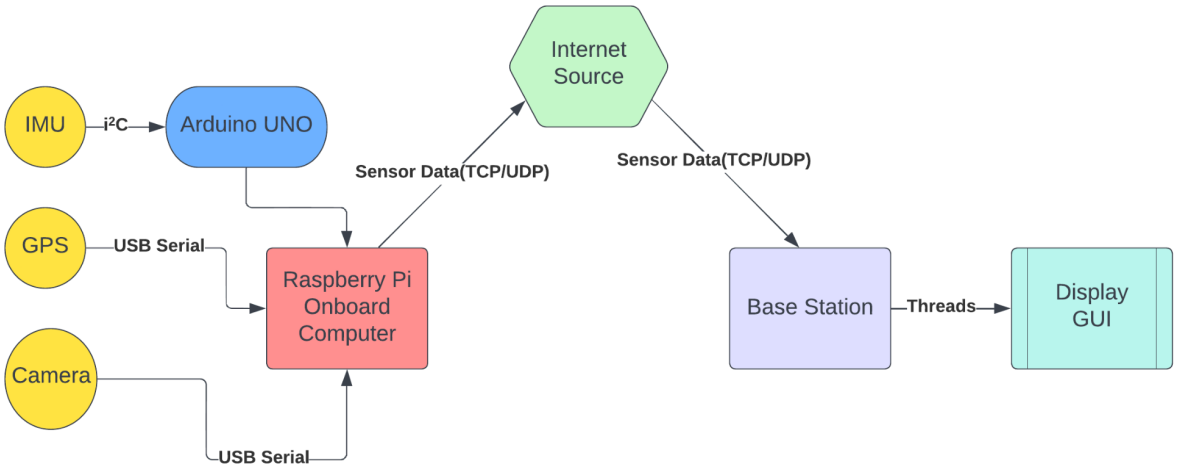


Fig 4 : Data Flow Architecture between base station and remote server

There are numerous communication protocols that could be used to communicate with the sensors individually. We decided to use USB serial for the digital camera and GPS to avoid problems with jumper wires and cross-talk among wires close to each other.
There are 3 main methodologies implemented to solve the compound problem statement stated above :

1. **Data Acquisition Pipeline :**
   This pipeline is created to access data from sensors in real-time and update the value of the last known sensor reading stored on the onboard computer. It is implemented using ROS nodes and ROS Parameter Server.

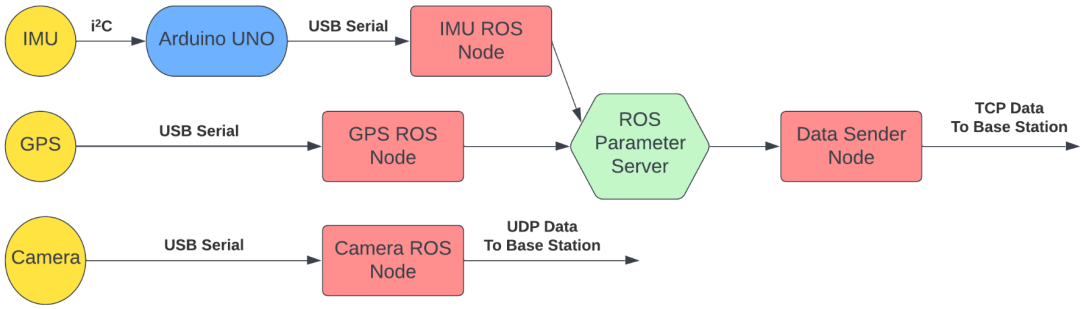


Fig 5 : Data Acquisition Pipeline

2. **TCP and UDP Socket Pipeline :**
   The second aspect of the solution is using a TCP socket to send all the sensor data across the network (assuming the base station and the UGV are connected to the same network) and a UDP socket to transfer across the network to the base station.

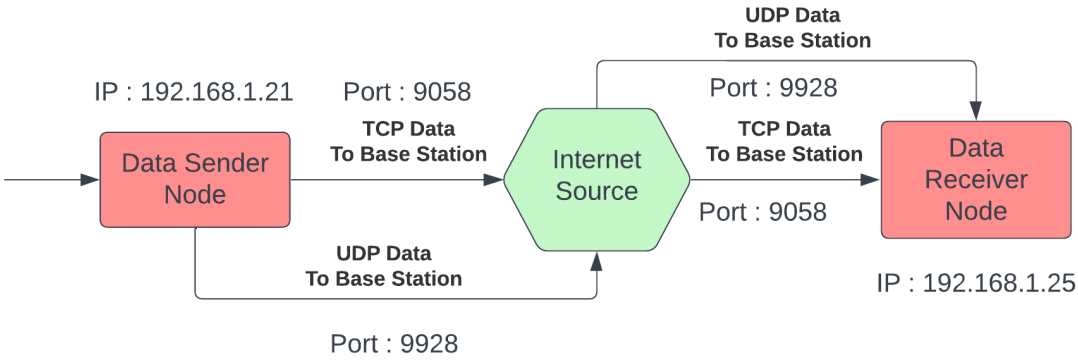


Fig 6 : Socket Pipeline

3. **Data Display Pipeline :**
   The pipeline acquires sensor data from the TCP socket and the image data from the UPD socket.It manages and handles validation functionalities such as validating checksum, checking for duplicate sequence number and reading flag bits of the TCP header. It uses the concept of sockets and multi-threading combined with PyQt GUI library.

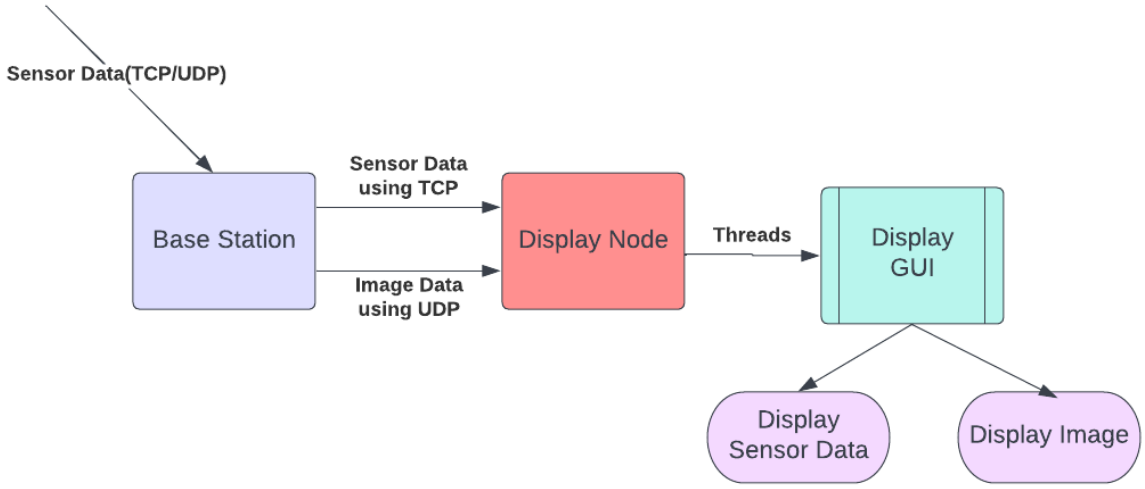


Fig 7 : Data Display Pipeline

These 3 pipelines are placed continually to establish secure communication with the remote server from the base station. They ensure data loss is minimum and maximum information is transported across the network.

5. Implementation Details

There are 2 main locations where code is being executed :

1. The remote server station (On the UGV)
2. The base station (Laptop/Workstation)

5.1 Remote Server :

1) data_packet.py

This code essentially creates the data packet that is to be sent in the data pipeline created in this project. It implements select features of the TCP data packet and adds on relevant details that are applicable to the current application. It also contains helper functions to easily validate,access and manipulate the data stored in the data packet.

Source Code :

```
import json
import sys
from datetime import datetime
import numpy as np
```

Static imports for libraries used in the code. Numpy arrays have been used to manipulate lists and arrays. datetime library has been used to get the current date and time of the sender. json is used to serialize the python objects.

```
class telemetryDataStructure:
   def __init__(self):
       self.L = []
       self.imu_roll = 0.0
       self.imu_pitch = 0.0
       self.imu_yaw = 0.0
       self.packets_sent = 0
       self.packets_recv = 0
       self.ping = 0.00
       self.signal_strength = 0.0
       self.imu_quat_x = 0.0000
       self.imu_quat_y = 0.0000
       self.imu_quat_z = 0.0000
       self.imu_quat_w = 0.0000
       self.gps_latitude = 0.000000
       self.gps_longitude = 0.000000
```

The above code snippet contains the class definition for the specific data packet designed for telemetry along with the constructor initialising all the member variables.

```
def assignValuefromList(self,L):
       self.L = L
       self.assignClassVariableValues()

   def assignClassVariableValues(self):
       self.imu_roll = np.double(self.L[0])
       self.imu_pitch = np.double(self.L[1])
       self.imu_yaw = np.double(self.L[2])
       self.packets_sent = np.double(self.L[3])
       self.packets_recv = np.double(self.L[4])
       self.ping = np.double(self.L[5])
       self.signal_strength = np.double(self.L[6])
       self.imu_quat_x = "IMU Orientation_x: "+str(np.double(self.L[7]))
       self.imu_quat_y = "IMU Orientation_y: "+str(np.double(self.L[8]))
       self.imu_quat_z = "IMU Orientation_z: "+str(np.double(self.L[9]))
       self.imu_quat_w = "IMU Orientation_w: "+str(np.double(self.L[10]))
       self.gps_latitude = "Latitude: "+str(np.double(self.L[11]))
```

```
        self.gps_longitude = "Longitude: "+str(np.double(self.L[12]))

    def printDataLog(self):
        print("Telemetry Data Log : ")
        print("Roll : "+str(self.imu_roll))
        print("Pitch : "+str(self.imu_pitch))
        print("Yaw : "+str(self.imu_yaw))
        print("Packets Sent : "+str(self.packets_sent))
        print("Packets Recv : "+str(self.packets_recv))
        print("Ping : "+str(self.ping))
        print("Signal Strength : "+str(self.signal_strength))
        print("Quat X : "+str(self.imu_quat_x))
        print("Quat Y : "+str(self.imu_quat_y))
        print("Quat Z : "+str(self.imu_quat_z))
        print("Quat W : "+str(self.imu_quat_w))
        print("Latitude : "+str(self.gps_latitude))
        print("Longitude : "+str(self.gps_longitude))
```

The above code snippet contains the helper functions to access, modify, delete and manipulate the member variables of the class TelemetryDataStructure.

```
class dataPacket:
    def __init__(self,sourceip,sourceport,destip,destport):

    # Data section of the Packet

        # {data} Contains the data to be sent to the base station
        self.data = ""
        self.data_dict = {}

    # Header of the data packet has been defined

        # {time} Contains the time of data packet creation to the header
        # Default - Contains the time the connection was initiated
        self.time = datetime.now().strftime("%m/%d/%Y, %H:%M:%S")

        # {source_ip} - Contains the destination IP address to the header
        self.source_ip = sourceip

        # {source_port} - Contains the port the packet has to be sent to
```

```
        self.source_port = sourceport

        # {dest_ip} - Contains the destination IP address to the header
        self.dest_ip = destip

        # {dest_port} - Contains the port the packet has to be sent to
        self.dest_port = destport

        # {checksum} - Check value for data integrity check
        self.checksum = self.initCheckSum()

        # {size} - Contains the size of the data packet being sent
        self.packet_size = 0

        # {seq_num} - Specifies the sequence number of the current data
packet
        # Note : must be defined externally and assigned
        self.seq_num = 0

        # Flag variables for the header
        self.ACK_FLAG = 1 # Indicates that acknowledgement number is valid.
        self.RST_FLAG = 0 # Resets the connection.
        self.URG_FLAG = 0 # Indicates that some urgent data has been
placed.
        self.FIN_FLAG = 0 # It is used to terminate the connection
```

The above code snippet contains the constructor definition for DataPacket class. This class provides a template for creating the data packet. It uses the TelemetryDataStructure class to store telemetry-specific values. The rest of the values provide more information about the packet.

```
# Driver Functions to enhance access and control

    # Function to configure details of the destination ip and port no
    def setDestDetails(self,ip,port):
        self.dest_ip = ip
        self.dest_port = port

    # Function to configure details of the source ip and port no
    def setSourceDetails(self,ip,port):
        self.source_ip = ip
        self.source_port = port
```

```
    # Function to return a str object of the class details
    def getParams(self):
        class_dict = self.__dict__
        return class_dict

    # Function to initialise checksum for the given dataPacket
    def initCheckSum(self):
        check = sys.getsizeof(self.data)
        return check

    # Function to assign sequence number to the data packet
    def assignSeqNum(self,seq):
        self.seq_num = seq

    # Function to assign data to the data segment
    def assignData(self,data):
        self.data = data

    # Function to assign packet size {In case it is variable}
    def assignPacketSize(self,packet):
        self.packet_size = sys.getsizeof(packet)

    def serializeDataSection(self,packet):
        self.data_dict = packet.data.__dict__
```

The above code snippet contains the helper functions to enhance control over the creation, modification and manipulation of the DataPacket class and its member variables.

```
# Class to provide Client and Server side programs access and
functionality to data packets
class PacketFunctions:
    # Function to validate data packets
    def validateCheckSum(self,packet):
        if(sys.getsizeof(packet.data)==packet.checksum):
            print("Packet {} validated. 0% packet
loss".format(packet.seq_num))
            return 1
        else:
            loss = sys.getsizeof(packet.data)-packet.checksum
```

```
            perc_loss = (loss/sys.getsizeof(packet.data))*100
            print("Packet {} validated. 0% packet
loss".format(str(perc_loss)))
            return 0
        return "Unable to validate checksum"
```

The third class defined in the file, namely PacketFunctions implements the validation functionality for the data packet. It also provides programmers with debugging messages in case of there being errors.

2) send_img_to_base.py

This python node is responsible for creating a UDP data packet using the image it obtains from the digital camera attached to the Raspberry Pi and send it across the socket as a datagram stream to the base station.

```
import cv2
import socket
import math
import pickle
import sys
```

Static imports for packages. The math package is used to carry out ceiling functionality within the code. pickle is a serializing library used to serialize the 3-D image (RGB) into a single 1-D array to send across the socket.

```
max_length = 65000
host = sys.argv[1]
port = int(sys.argv[2])

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

The above code describes the socket declarations in python for a UDP socket. The values of destination IP and destination port no are passed as command-line arguments to the code.

```
cap = cv2.VideoCapture(int(sys.argv[3]))
ret, frame = cap.read()
```

The above code uses OpenCV to access the camera at index specified in the command line and store it in a variable.

```
while ret:
```

```
# compress frame
retval, buffer = cv2.imencode(".jpg", frame)

if retval:
    # convert to byte array
    buffer = buffer.tobytes()
    # get size of the frame

    buffer_size = len(buffer)

    num_of_packs = 1
    if buffer_size > max_length:
        num_of_packs = math.ceil(buffer_size/max_length)

    frame_info = {"packs":num_of_packs}

    # send the number of packs to be expected
    print("Number of packs:", num_of_packs)
    sock.sendto(pickle.dumps(frame_info), (host, port))

    left = 0
    right = max_length

    for i in range(num_of_packs):
        print("left:", left)
        print("right:", right)

        # truncate data to send
        data = buffer[left:right]
        left = right
        right += max_length

        # send the frames accordingly
        sock.sendto(data, (host, port))
```

The above code serializes the image into a 1-D array and sends it to the UDP socket created before. It also prints the debugging information for the programmer to infer the exact images being processed.

3) data_sender.py

This python node collects the telemetry information from the ROS parameter server and stores it into local variables. It then calls the functions pertaining to the DataPacket class and creates the data packet. It then sends it across the TCP socket.

```
#!/usr/bin/env python3
"""
Maintainer - Aditya Arun Iyer
Last Modified - 02/09/2022 13:06 PM

The below driver code compiles the sensor and telemetry values into a
single serial parameter(str) and sends
it to the base station using a TCP socket connection

Run Location : Remote Server
"""
import socket
import time
import rospy
import threading
from data_packet import dataPacket,PacketFunctions
import json
```

Static imports and the documentation of the code facilitates the functionality of the code.

```
class Connect_Socket:

   def __init__(self):
       # Variable Declerations
       self.IP_add_c1 = "127.0.0.1"
       self.port3 = 9058
       self.imu_roll = 0.0
       self.imu_pitch = 0.0
       self.imu_yaw = 0.0
       self.packets_sent = 0
       self.packets_recv = 0
       self.ping = 0.00
       self.signal_strength = 0.0
       self.imu_quat_x = 0.0000
       self.imu_quat_y = 0.0000
       self.imu_quat_z = 0.0000
       self.imu_quat_w = 0.0000
```

```
        self.gps_latitude = 0.000000
        self.gps_longitude = 0.000000
        self.msg = ''
        self.msg_to_send = ''
        self.data_packet_msg = ''

        # Socket Declerations
        self.s1 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.s2 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.s3 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        # Thread declerations
        self.th_DataCollection =
threading.Thread(target=self.CollectDataFromServer)
        self.th_send_params = threading.Thread(target=self.send_params)
```

The code creates the Connect_Socket class with the constructor which contains 2 main threads. One of the threads scrapes the ROS parameter server to acquire the values and the second creates the data packet and sends it across the TCP socket.

```
def CollectDataFromServer(self):
        while not rospy.is_shutdown():
            time.sleep(1)
            self.imu_roll = rospy.get_param("/IMU/Roll")
            self.imu_pitch = rospy.get_param("/IMU/Pitch")
            self.imu_yaw = rospy.get_param("/IMU/Yaw")
            self.packets_sent = rospy.get_param("/Packets/Sent")
            self.packets_recv = rospy.get_param("/Packets/Recieved")
            self.ping = rospy.get_param("/ping")
            self.signal_strength = rospy.get_param("/signalStrength")
            self.imu_quat_x = rospy.get_param("/IMU/Quat/x")
            self.imu_quat_y = rospy.get_param("/IMU/Quat/y")
            self.imu_quat_z = rospy.get_param("/IMU/Quat/z")
            self.imu_quat_w = rospy.get_param("/IMU/Quat/w")
            self.gps_latitude = rospy.get_param("/GPS/latitude")
            self.gps_longitude = rospy.get_param("/GPS/longitude")
            self.msg = str(self.imu_roll) +
","+str(self.imu_pitch)+","+str(self.imu_yaw)+","+str(self.packets_sent)+"
,"+str(self.packets_recv)+","+str(self.ping)+","+str(self.signal_strength)
```

```
+","+str(self.imu_quat_x)+","+str(self.imu_quat_y)+","+str(self.imu_quat_z
)+","+str(self.imu_quat_w)+","+str(self.gps_latitude)+","+str(self.gps_lon
gitude)
            rospy.set_param('gui_msg',self.msg)
            print("Updated Message")
            time.sleep(5)
```

The CollectDataFromServer() function is continuously executed to scrape the ROS parameter server and obtain the updated data points.

```
    def createDataPacket(self,data,seq_num):
        packet =
dataPacket(self.IP_add_c1,self.port3,self.IP_add_c1,self.port3)
        packet.assignData(data)
        packet.assignSeqNum(seq_num)
        packet.initCheckSum()
        packet.assignPacketSize(packet)
        return packet
```

The function takes the data as input and crafts the data packet. It returns an object containing the packet information.

```
def send_params(self):

        # binding port and host
        self.s1.bind((self.IP_add_c1, self.port3))
        print("Waiting for client to connect")

        # waiting for a client to connect
        self.s1.listen(5)
        c, addr = self.s1.accept()
        seq_num = 0
        while True:
            self.msg_to_send = rospy.get_param('gui_msg')
            seq_num+=1
            self.data_packet =
self.createDataPacket(self.msg_to_send,seq_num)
            # sending data type should be string and encode before sending
            print("sent msg : {}".format(self.data_packet.getParams()))
            self.data_packet_msg = self.data_packet.getParams()
            self.data_packet_msg = json.dumps(self.data_packet_msg)
```

```
            c.send(self.data_packet_msg.encode())
            time.sleep(5)
```

The send_params() function obtains the message from the parameter server and crafts the data packet. It also initialises the connection and sends the data across the TCP socket.

```
def driver(self):
        self.th_DataCollection.start()
        time.sleep(3)
        self.th_send_params.start()

if __name__ == '__main__':
    rospy.init_node('test_sender')
    sock_obj = Connect_Socket()
    sock_obj.driver()
```

The driver code to run the python node.

5.2 Base Station:

1) telemetry_unit.py

This node creates the GUI elements and receives data from both the UDP as well as the TCP socket, and displays the data in a GUI curated to provide extra information about the telemetry of the UGV.

(Note : The code snippets will only mention the functions relevant to data acquisition and Computer Networks and will not include GUI functionality. For full reference : https://github.com/LavaHawk0123/AAIbot-telemetry-unit.git)

```
class Ui_MainWindow(object):

    def declare_vars(self):
        self.L = []
        self.msg = {}
        # Variable Declerations
        host = socket.gethostname()
        self.IP = socket.gethostbyname(host)
        self.IP_add = "127.0.0.1"
        self.IP_add_c2 = "172.20.10.4"
        self.IP_add_c1 = "192.168.1.101"
        self.port1 = 9928
        self.port2 = 9047
        self.port3 = 9058
```

```
        self.port4 = 12345
        self.imu_roll = 0.0
        self.imu_pitch = 0.0
        self.imu_yaw = 0.0
        self.packets_sent = 0
        self.packets_recv = 0
        self.ping = 0.00
        self.signal_strength = 0.0
        self.imu_quat_x = 0.0000
        self.imu_quat_y = 0.0000
        self.imu_quat_z = 0.0000
        self.imu_quat_w = 0.0000
        self.gps_latitude = 0.000000
        self.gps_longitude = 0.000000
        self.msg_dict = {}
        # Socket Declerations
        self.s1 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)


       # Thread declerations
        self.th_setCameraImageLabel =
threading.Thread(target=self.getCameraUDP)
        self.th_setLabelValues =
threading.Thread(target=self.getDataFromSocket)
        self.set_values_th = threading.Thread(target=self.set_values)
        self.addCoordinateThread = threading.Thread(target =
self.addItemToList)
        self.signalStrengthThread = threading.Thread(target =
self.setSignalLabels)
```

The above code illustrates the constructor of the UI class that declares the member variables, the socket initialization and thread declarations. There are a total of 5 threads:

I. *th_setCameraImageLabel*: This thread obtains the camera image from the UDP socket and sets the image label with the image.
II. *th_setLabelValues* : This thread obtains the sensor data from the TCP socket, parses it and discards unnecessary information and stores it locally.
III. *set_values_th* : This thread continuously updates the values of the labels containing the data points.
IV. *addCoordinateThread* : This thread helps add the current GPS coordinate being streamed from the UGV into a ListBox data structure for future reference
V. *signalStrengthThread* : Updates the visual representation of the signal strength of both sender and receiver.

```
def getDataFromSocket(self):
        self.s1.connect((self.IP_add, self.port3))
        while True:
            time.sleep(2)
            self.msg = self.s1.recv(1024).decode()
            print("\nRecieved Message"+ self.msg)
            self.msg_dict = json.loads(self.msg)
            self.L = self.msg_dict['data'].split(",")
            telemData = telemetryDataStructure()
            telemData.assignValuefromList(self.L)
            telemData.printDataLog()
            self.assignClassVariableValues()
```

The function used to obtain data from the TCP Socket.

```
def getCameraUDP(self):
        max_length = 65540
        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        sock.bind((self.IP_add, self.port1))
        frame_info = None
        buffer = None
        frame = None
        print("-> waiting for connection")
        while True:
            data, address = sock.recvfrom(max_length)
            if len(data) < 100:
                frame_info = pickle.loads(data)

                if frame_info:
                    nums_of_packs = frame_info["packs"]

                    for i in range(nums_of_packs):
                        data, address = sock.recvfrom(max_length)

                        if i == 0:
                            buffer = data
```

```
                        else:
                            buffer += data

                    frame = np.frombuffer(buffer, dtype=np.uint8)
                    frame = frame.reshape(frame.shape[0], 1)

                    frame = cv2.imdecode(frame, cv2.IMREAD_COLOR)
                    frame = cv2.flip(frame, 1)

                    if frame is not None and type(frame) == np.ndarray:
                        self.cv_img = frame
                        self.qt_img = self.convert_cv_qt(self.cv_img, 791,
571)
                        self.image_label.setPixmap(self.qt_img)
```

The source code of the function used to obtain and de-serialize the image obtained from the UDP socket and display it in a label.

Output Screenshots:

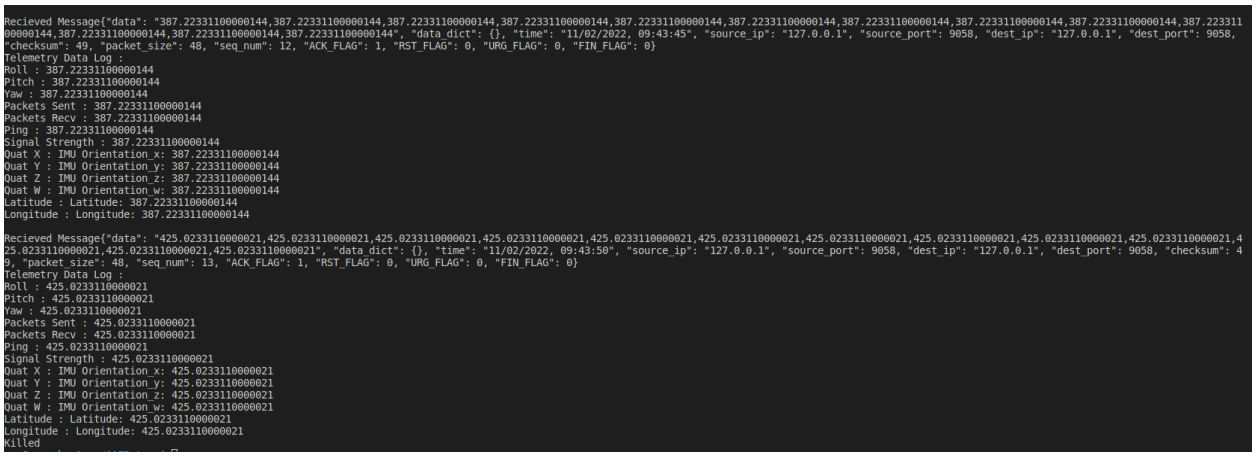
1. Data Sender node collecting data, updating the message and waiting for client to connect

```
neo@neo-ubuntu:~/AAIBot_ws$ ./launch_remote_station_testbench.sh
Base path: /home/neo/AAIBot_ws
Source space: /home/neo/AAIBot_ws/src
Build space: /home/neo/AAIBot_ws/build
Devel space: /home/neo/AAIBot_ws/devel
Install space: /home/neo/AAIBot_ws/install
####
#### Running command: "make cmake_check_build_system" in "/home/neo/AAIBot_ws/build"
####
####
#### Running command: "make -j8 -l8" in "/home/neo/AAIBot_ws/build"
####
... logging to /home/neo/.ros/log/2574e5a4-5a47-11ed-a3b3-c955c7d92574/roslaunch-neo-ubuntu-22626.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://neo-ubuntu:44201/

SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.15.14

NODES
  /
    param_testbench (telemetry/test_param_load.py)
    send_data_to_gui (telemetry/data_sender.py)

ROS_MASTER_URI=http://localhost:11311

process[param_testbench-1]: started with pid [22652]
process[send_data_to_gui-2]: started with pid [22653]
Updated Message
Waiting for client to connect
```

2. The UDP server side node sending images obtained from the camera

```
neo@neo-ubuntu:~/AAIBot_ws$ ./send_image_base.sh
Number of packs: 1
left: 0
right: 65000
Number of packs: 1
left: 0
right: 65000
Number of packs: 1
left: 0
right: 65000
Number of packs: 1
left: 0
right: 65000
Number of packs: 1
left: 0
right: 65000
Number of packs: 1
left: 0
right: 65000
Number of packs: 1
left: 0
right: 65000
```

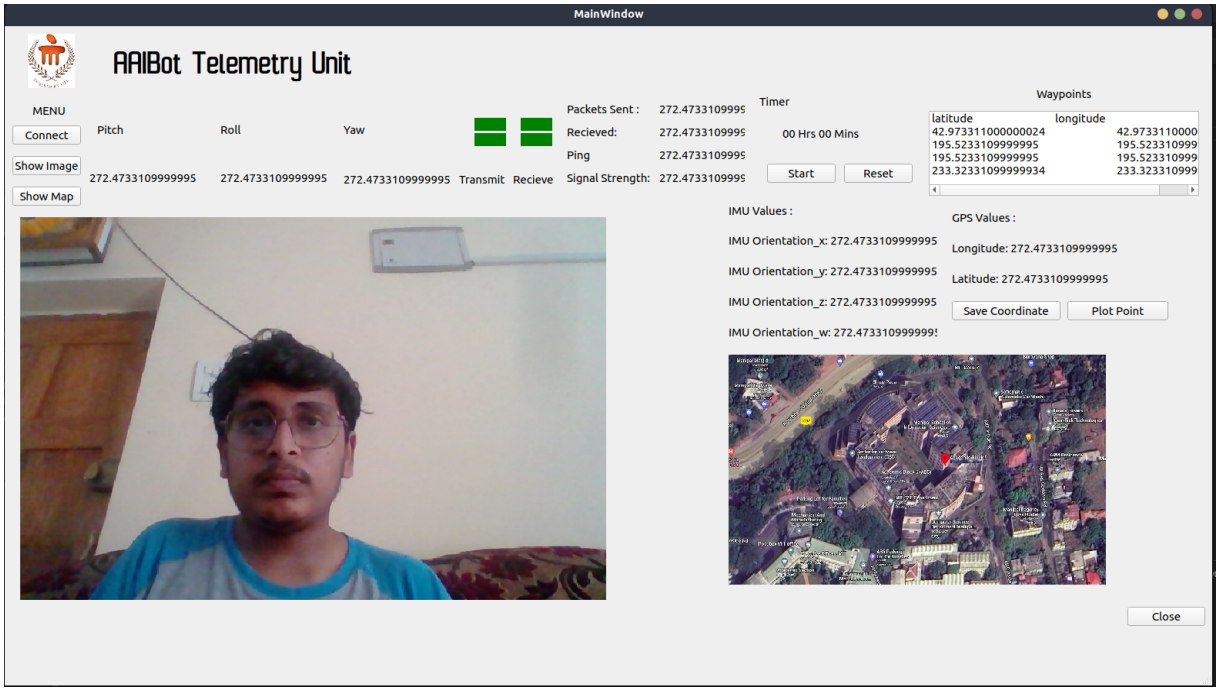
3. The Client side acknowledgement of the custom data packet received



4. The Server side acknowledgement of the custom data packet received

```
Waiting for client to connect
Updated Message
sent msg : {'data': '42.973311000000024,42.973311000000024,42.973311000000024,42.973311000000024,42.97331
1000000024,42.973311000000024,42.973311000000024,42.973311000000024,42.973311000000024,42.973311000000024
,42.973311000000024,42.973311000000024,42.973311000000024', 'data_dict': {}, 'time': '11/02/2022, 09:42:5
0', 'source_ip': '127.0.0.1', 'source_port': 9058, 'dest_ip': '127.0.0.1', 'dest_port': 9058, 'checksum':
 49, 'packet_size': 48, 'seq_num': 1, 'ACK_FLAG': 1, 'RST_FLAG': 0, 'URG_FLAG': 0, 'FIN_FLAG': 0}
sent msg : {'data': '42.973311000000024,42.973311000000024,42.973311000000024,42.973311000000024,42.97331
1000000024,42.973311000000024,42.973311000000024,42.973311000000024,42.973311000000024,42.973311000000024
,42.973311000000024,42.973311000000024,42.973311000000024', 'data_dict': {}, 'time': '11/02/2022, 09:42:5
5', 'source_ip': '127.0.0.1', 'source_port': 9058, 'dest_ip': '127.0.0.1', 'dest_port': 9058, 'checksum':
 49, 'packet_size': 48, 'seq_num': 2, 'ACK_FLAG': 1, 'RST_FLAG': 0, 'URG_FLAG': 0, 'FIN_FLAG': 0}
Updated Message
sent msg : {'data': '80.77331099999998,80.77331099999998,80.77331099999998,80.77331099999998,80.773310999
99998,80.77331099999998,80.77331099999998,80.77331099999998,80.77331099999998,80.77331099999998,80.773310
99999998,80.77331099999998,80.77331099999998', 'data_dict': {}, 'time': '11/02/2022, 09:43:00', 'source_i
p': '127.0.0.1', 'source_port': 9058, 'dest_ip': '127.0.0.1', 'dest_port': 9058, 'checksum': 49, 'packet_
size': 48, 'seq_num': 3, 'ACK_FLAG': 1, 'RST_FLAG': 0, 'URG_FLAG': 0, 'FIN_FLAG': 0}
Updated Message
sent msg : {'data': '118.57331099999982,118.57331099999982,118.57331099999982,118.57331099999982,118.5733
1099999982,118.57331099999982,118.57331099999982,118.57331099999982,118.57331099999982,118.57331099999982
,118.57331099999982,118.57331099999982,118.57331099999982', 'data_dict': {}, 'time': '11/02/2022, 09:43:0
5', 'source_ip': '127.0.0.1', 'source_port': 9058, 'dest_ip': '127.0.0.1', 'dest_port': 9058, 'checksum':
 49, 'packet_size': 48, 'seq_num': 4, 'ACK_FLAG': 1, 'RST_FLAG': 0, 'URG_FLAG': 0, 'FIN_FLAG': 0}
Updated Message
```

5. Final GUI output



6. Contribution Summary

The project being hardware-based meant that there was a large amount of workload to manage for both of us. However, we managed to evenly segregate the workload based on our strengths. Both of us were equally involved in the **ideation** and **planning** phases of the project as well as the **literature review** required for this project. With respect to individual contributions:

Aditya Iyer:

1) Writing the custom data packet structure for the TCP socket

2) Writing code for the image sender node using UDP
3) Developed Display GUI
4) Worked on integrating the project with ROS
5) Setup of electronics on the Display Bot

Ayush Dixit:

1) Worked on the functionality of Display GUI
2) Writing the code for the sender node for the TCP socket
3) Worked on building the robot for display
4) Worked on parallel programming and threading

7. References

I. http://library.isr.ist.utl.pt/docs/roswiki/ROS(2f)UDPROS.html
II. http://library.isr.ist.utl.pt/docs/roswiki/ROS(2f)TCPROS.html
III. https://www.oreilly.com/library/view/programming-python-3rd/0596009259/ch13s03.html
IV. https://wiki.python.org/moin/PyQt/Tutorials
V. http://gaia.cs.umass.edu/kurose_ross/online_lectures.htm
VI. http://wiki.ros.org/Documentation