A
Project Report
On
“Chat System for Multiple Clients”

Department of Computer Science & Engineering

NATIONAL INSTITUE OF TECHNOLOGY PATNA

University Campus, Bihar - 800005

Submitted By:

Anoop Kumar Dwivedi (2446007)

Course Code: MC460201 Course Title: Data Communication and Networks Branch: MCA-DS CSE

TABLE OF CONTENTS

LIST OF FIGURES

AIM OF THE EXPERIMENT

Design and implement a **multi-client chat system** using **Python socket programming (UDP protocol).** The system will support **real-time text messaging** and **image sharing** between multiple clients and a central server.

1. INTRODUCTION

1.1 Overview of Networking

Networking plays a crucial role in modern communication. It enables data transfer across different devices and locations, ensuring seamless connectivity. The two main types of data transfer mechanisms are **Unicast** (one-to-one communication), **Broadcast** (one-to-many communication), and **Multicast** (one-to-selected group communication). Networking protocols like TCP/IP ensure smooth communication over the internet.

1.2 Introduction to Client-Server Architecture

Client-server architecture is a fundamental model in computer networking where multiple client devices communicate with a central server to request and receive services. This model is widely used in applications such as web browsing, email, online gaming, and messaging systems.

In this architecture, the **client** is a device or application that initiates requests, while the **server** is a powerful computer or software that processes these requests and provides the required data or service. The communication between clients and servers happens over a network using protocols like **TCP/IP (Transmission Control Protocol/Internet Protocol)** or **UDP (User Datagram Protocol)**.

A simple example of client-server architecture is a **chat system**. When a user sends a message from their phone (client), the request is sent to a server, which then forwards the message to the intended recipient. The server acts as an intermediary, ensuring proper message delivery and managing multiple users simultaneously.

Components of Client-Server Architecture

1. **Client:** Requests data or services (e.g., a web browser requesting a webpage).

2. **Server:** Processes client requests and responds accordingly (e.g., a web server delivering a webpage).

3. **Network:** The communication medium connecting clients and servers (e.g., the internet or a local network).

Advantages of Client-Server Architecture

1. **Centralized Management:** Easy to update and maintain.

2. **Scalability:** Can support multiple clients efficiently.

3. **Security:** Controlled access to data and resources.

Limitations

- **Server Dependency:** If the server fails, clients cannot access services.

- **Latency Issues:** Network congestion can slow down responses.

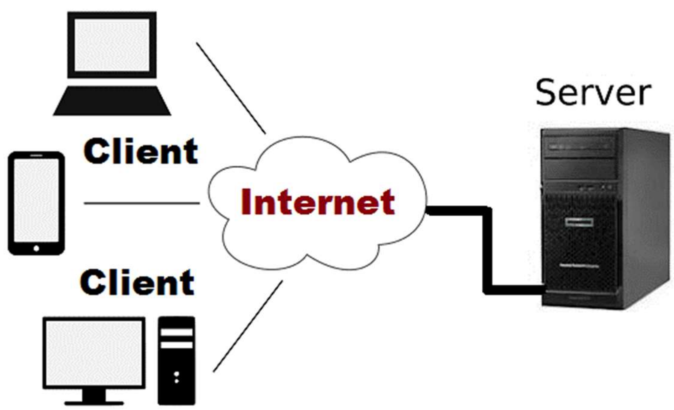


Figure 1- Components of Client-Server Architecture

Figure-1 shows the Client-Server model over internet.

1.3 UDP vs TCP Communication

- **Transmission Control Protocol (TCP)**

  TCP is a connection-oriented protocol, meaning it establishes a connection between sender and receiver before transmitting data. It ensures reliable and ordered delivery of messages.

  Key Features of TCP:

  - **Reliable** – Ensures that all data packets reach their destination correctly.
  - **Ordered Delivery** – Messages are received in the correct sequence.
  - **Error Checking & Correction** – Automatically detects and retransmits lost packets.
  - **Three-Way Handshake** – Establishes a secure connection before data transfer.

  Use Cases of TCP:

  - **Web Browsing (HTTP/HTTPS)** – Ensures complete and correct webpage loading.
  - **File Transfers (FTP)** – Guarantees all parts of a file are received.
  - **Emails (SMTP, IMAP, POP3)** – Ensures no email data is lost.

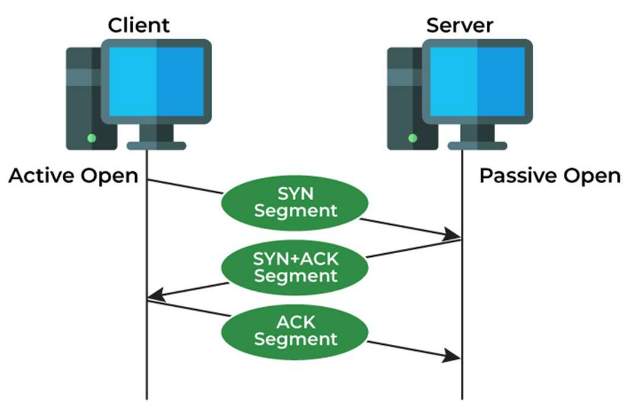


Figure 2- Transmission Control Protocol

Figure-2 shows the TCP connection establishment process, alos known as Three-Way Handshake.

- **User Datagram Protocol (UDP)**

  UDP is a connectionless protocol, meaning it sends data without establishing a connection. It prioritizes speed over reliability and does not guarantee packet delivery.

  Key Features of UDP:

- **Fast & Lightweight** – No connection setup, making it faster than TCP.
- **Unreliable** – No retransmission of lost packets.
- **No Ordering** – Packets may arrive out of sequence.
- **Low Overhead** – No extra checks, making it ideal for real-time applications.

Use Cases of UDP:

- **Online Gaming** – Reduces lag by prioritizing speed.
- **Video Streaming** (YouTube, Zoom) – Prevents buffering delays.
- **VoIP Calls** (WhatsApp, Skype) – Ensures smooth audio/video communication.

This project implements **UDP-based** communication for real-time messaging and image transfer between a server and multiple clients, making it suitable for applications that prioritize speed over reliability.

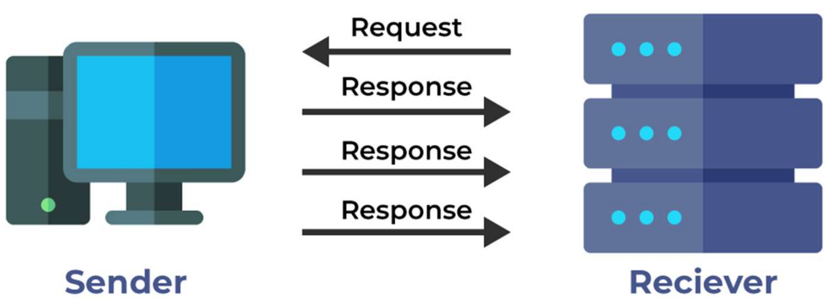


Figure 3- User Datagram Protocol

Figure-3 shows how UDP Request-Response Cycle.

1.4 Scope of the Project

The aim of this project is to implement a UDP-based chat application with:

1. **Real-time text messaging**
2. **Image transfer between clients**
3. **Multi-client support using threading**
4. **Efficient handling of large data packets**
5. **Cross-platform compatibility**

2. SYSTEM REQUIREMENTS

Hardware Requirements:

- Processor: 2 GHz or higher
- RAM: 512 MB or more
- Storage: 100 MB free space
- Network Adapter for LAN/WiFi connectivity
- Keyboard/Mouse for data input

Software Requirements:

- Python 3.x
- MS Word(Documentation)
- PIL (Python Imaging Library) for image processing
- Socket Programming Module
- Operating System: Windows/Linux/macOS

3. Functional & Non-Functional Requirement

Functional Requirements:

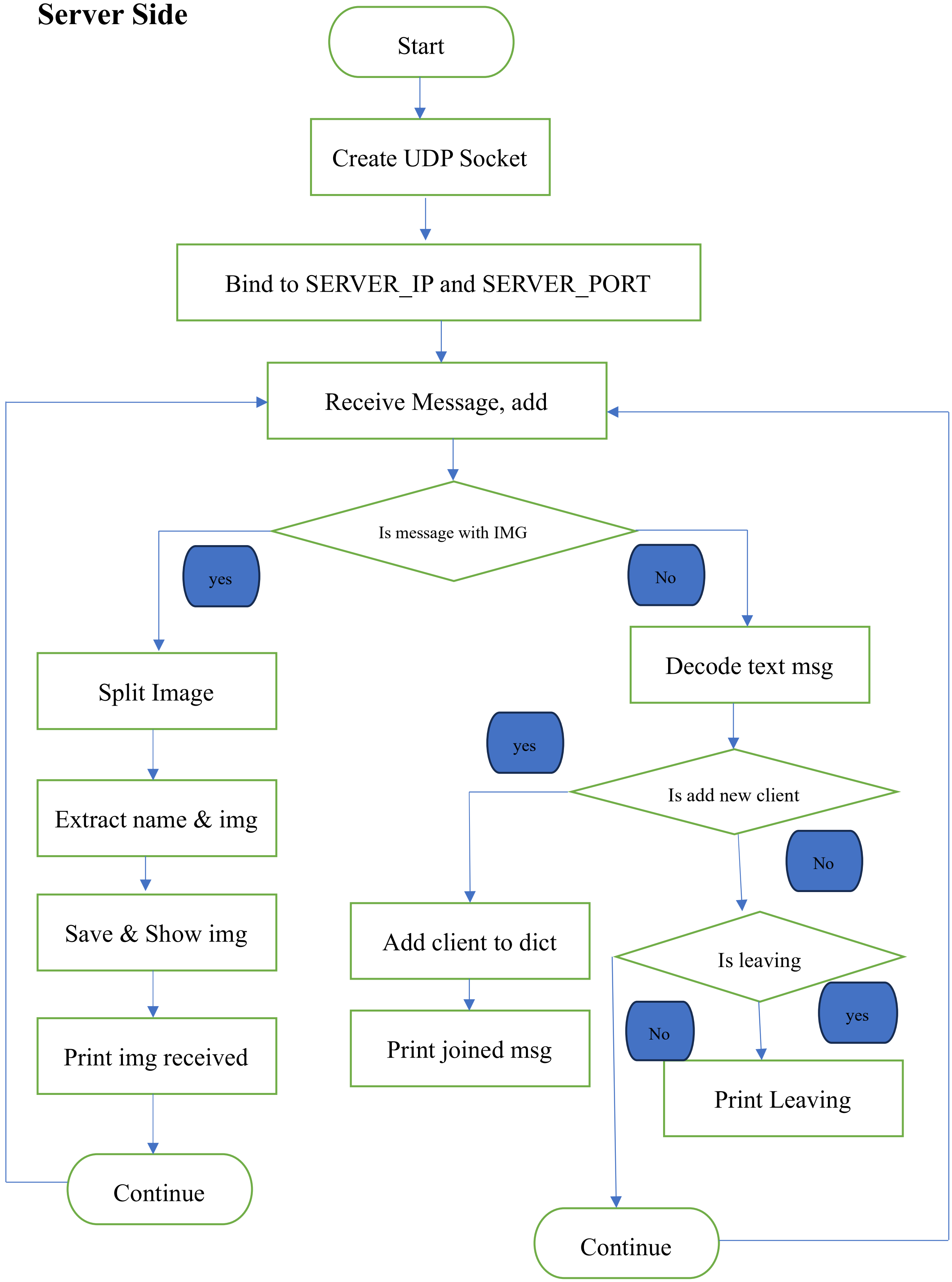
- ✓ Client can join/leave the chat.
- ✓ Server tracks all connected clients.
- ✓ Clients can send text and images.
- ✓ Server broadcast messages/images to all clients.
- ✓ Received images are saved and displayed.
- ✓ Messages include sender's name.
- ✓ Multithreaded client: send & receive simultaneously.
- ✓ Handles client disconnections.

Non-Functional Requirements:

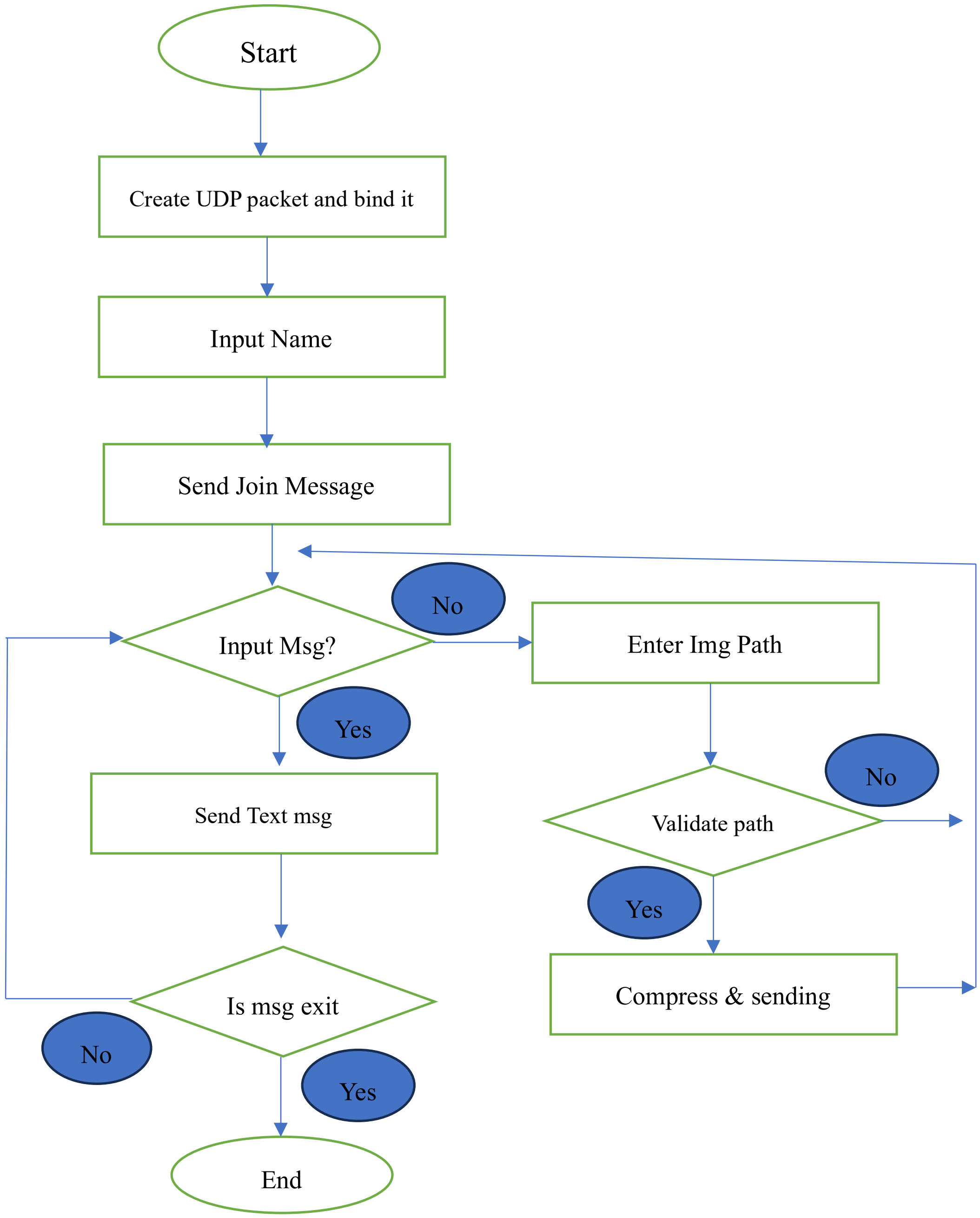
- ✓ **Performance:** Fast communication with image compression.
- ✓ **Scalability:** Supports multiple clients.
- ✓ **Usability:** Simple terminal input with clear feedback.
- ✓ **Portability:** Works across OS with Python + PIL.
- ✓ **Reliability:** Handles errors and exits cleanly.
- ✓ **Security:** Basic data validation (no encryption).
- ✓ **Maintainability:** Clean, modular code for easy updates.

4. FLOWCHART / DATA FLOW DIAGRAM

Server Side



Client Side



Start
Create UDP packet and bind it
Input Name
Send Join Message
Input Msg?
No
Enter Img Path
Yes
Send Text msg
No
Validate path
Yes
Compress & sending
Is msg exit
No
Yes
End

5. CODE

5.1 Server-Side Code

The server listens for incoming client messages and processes:

- Text messages
- Image data
- Client connections and disconnections

```
import socket  # Importing socket module for network communication
from PIL import Image  # Importing PIL (Pillow) to handle images
import io  # Importing io to handle byte streams

# Server Configuration
SERVER_IP = '127.0.0.1'  # Localhost IP address (for testing on the same machine)
SERVER_PORT = 12345  # Port number where the server will listen for client messages
BUFFER_SIZE = 65507  # Maximum size of data that can be received at once

clients = {}  # Dictionary to store connected clients in the format {address: name}

def handle_client():
    """
    Function to handle incoming messages from clients.
    It processes both text messages and image files.
    """

    # Creating a UDP socket
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    # AF_INET -> IPv4, SOCK_DGRAM -> UDP communication

    # Binding the server to the specified IP and port
    server_socket.bind((SERVER_IP, SERVER_PORT))
    print(f"Server started at {SERVER_IP}:{SERVER_PORT}")
    while True:
        # Receiving data from any client
        message, addr = server_socket.recvfrom(BUFFER_SIZE)
```

```
# Checking if the received data is an image (prefix "IMG:")
 if message.startswith(b"IMG:"):
    try:
        # Splitting the message to extract sender's name and image data
        parts = message.split(b":", 2)  # Splitting into ["IMG", sender_name, img_data]
        sender_name = parts[1].decode()  # Extract sender name
        img_data = parts[2]  # Extract image data

        # Converting received byte data into an image
        image = Image.open(io.BytesIO(img_data))

        # Creating a filename for storing the received image
        filename = f"received_image_from_{sender_name}.jpg"
        image.save(filename)  # Saving image locally

        image.show()

        print(f"Image received and saved as {filename} from {sender_name}")

        except Exception as e:
        # Handling errors if image processing fails
        print(f"Error opening image: {e}")

 else:
    try:
        # Decoding the received text message
        text_message = message.decode().strip()

        # If the client is new, store their name
        if addr not in clients:
            clients[addr] = text_message.split(" has joined the chat.")[0]
            print(text_message)
            continue
        print(text_message)
```

```
        except UnicodeDecodeError:
            # Handling errors if non-text data is received unexpectedly
            print("Received non-text data that could not be decoded.")

# Ensuring the script runs only if executed directly (not imported)
if __name__ == "__main__":
    handle_client()  # Start the server function
```

5.2 Client-Side Code

The client allows users to:

- Send and receive messages
- Transfer compressed images

```
# Importing library
import socket
import threading
import os
from PIL import Image
import io
import sys

# Client Configuration
SERVER_IP = '127.0.0.1'
SERVER_PORT = 12345
BUFFER_SIZE = 65507  # Maximum amount of data that can be sent in a single UDP packet
def compress_image(image_path):
    """
    Function to compress and resize an image before sending.
    It ensures the image size is reduced for efficient transmission.
    """
```

```
    try:
        # Open the image and convert it to RGB format
        img = Image.open(image_path).convert("RGB")

        # Resize the image while maintaining the aspect ratio (max 640x480)
        img.thumbnail((640, 480))

        # Create a byte stream to store the compressed image
        img_buffer = io.BytesIO()

        # Save the image in JPEG format with 50% quality to reduce size
        img.save(img_buffer, format="JPEG", quality=50)

        return img_buffer.getvalue()  # Return the compressed image as bytes

    except Exception as e:
        # Handle any errors during image processing
        print(f"Error processing image: {e}")
        return None

def send_message(client_socket, name):
    """
    Function to handle sending messages and images to the server.
    Users can type text messages, send images, or exit the chat.
    """
    while True:
        # Taking user input for message
        message = input("Enter message (or type 'send image' to send an image, or 'exit' to
leave): ")

        # Handling exit condition
        if message.lower() == "exit":
            # Notify server that the client is leaving
            client_socket.sendto(f"{name} has left the chat.".encode(), (SERVER_IP,
SERVER_PORT))
```

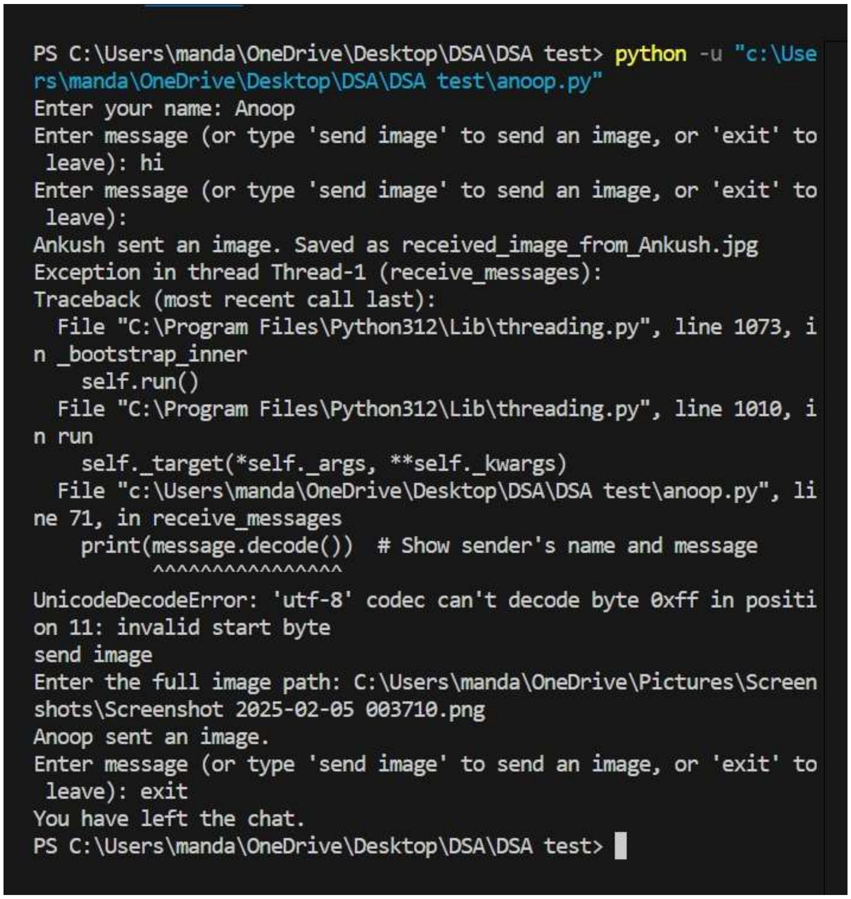
```
            print("You have left the chat.")
            client_socket.close()
            sys.exit(0)

        # Handling image sending
        elif message.lower() == 'send image':
            image_path = input("Enter the full image path: ").strip()  # Get image file path from user

            # Check if the file path is valid
            if not os.path.isfile(image_path):
                print("Error: Invalid file path.")
                continue  # Skip and ask for input again

            # Compress and prepare the image for sending
            img_data = compress_image(image_path)

            # If compression fails, skip sending
            if img_data is None:
                continue

            # Ensure the image size fits within the buffer limit
            if len(img_data) > BUFFER_SIZE - 50:  # Keeping some buffer for metadata
                print("Warning: Even after compression, the image is too large.")
                continue

            # Prefix image data with identifier and sender's name
            client_socket.sendto(f"IMG:{name}:".encode() + img_data, (SERVER_IP, SERVER_PORT))
            print(f"{name} sent an image.")

        # Handling text message sending
        else:
            # Prefix message with sender's name and send to the server
            client_socket.sendto(f"{name}: {message}".encode(), (SERVER_IP, SERVER_PORT))
```

```
def receive_messages(client_socket):
    """
    Function to continuously listen for incoming messages from the server.
    It runs in a separate thread to enable simultaneous sending and receiving.
    """
    while True:
        try:
            # Receive messages from the server
            message, addr = client_socket.recvfrom(BUFFER_SIZE)

            try:
                # Decode and display received text messages
                print(message.decode())
            except UnicodeDecodeError:
                # Handle error if received data is not a valid text message
                print("Received non-text data that could not be decoded.")

        except OSError:
            # Stop receiving if socket is closed
            break
        except Exception as e:
            # Handle other errors that may occur
            print(f"Error receiving message: {e}")
            break

def start_client():
    """
    Function to initialize the client, get user details, and start communication.
    """
    # Create a UDP socket
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    # AF_INET -> IPv4, SOCK_DGRAM -> UDP communication

    # Bind the socket to any available port on the client's machine
    client_socket.bind(('0.0.0.0', 0))  # 0 allows the OS to choose any available port
```

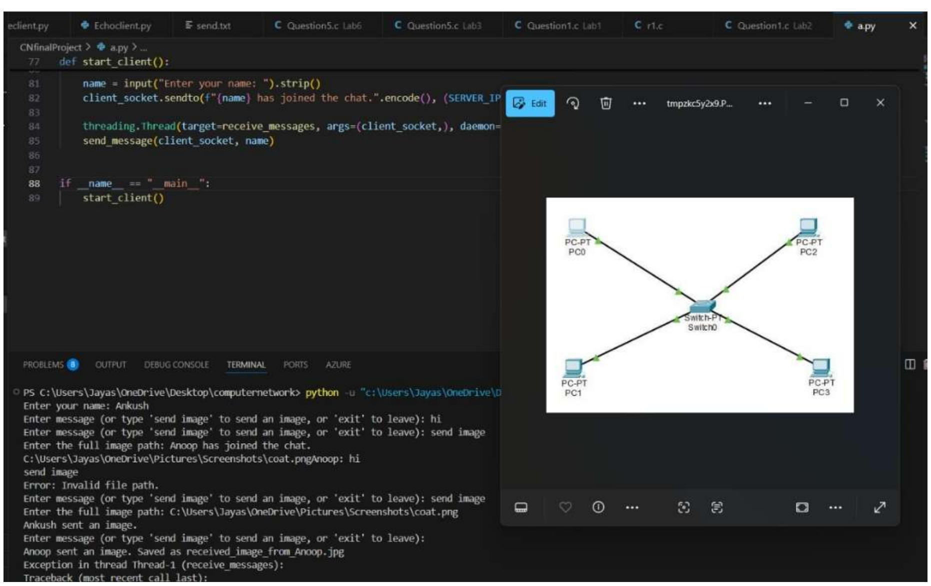
```
    # Asking user for their name
    name = input("Enter your name: ").strip()

    # Notify server that this user has joined the chat
    client_socket.sendto(f"{name} has joined the chat.".encode(), (SERVER_IP, SERVER_PORT))

    # Start a separate thread to receive messages from the server
    threading.Thread(target=receive_messages, args=(client_socket,), daemon=True).start()

    # Start sending messages (in the main thread)
    send_message(client_socket, name)

# Ensure the script runs only if executed directly (not imported)
if __name__ == "__main__":
    start_client()
```

6. OUTPUT

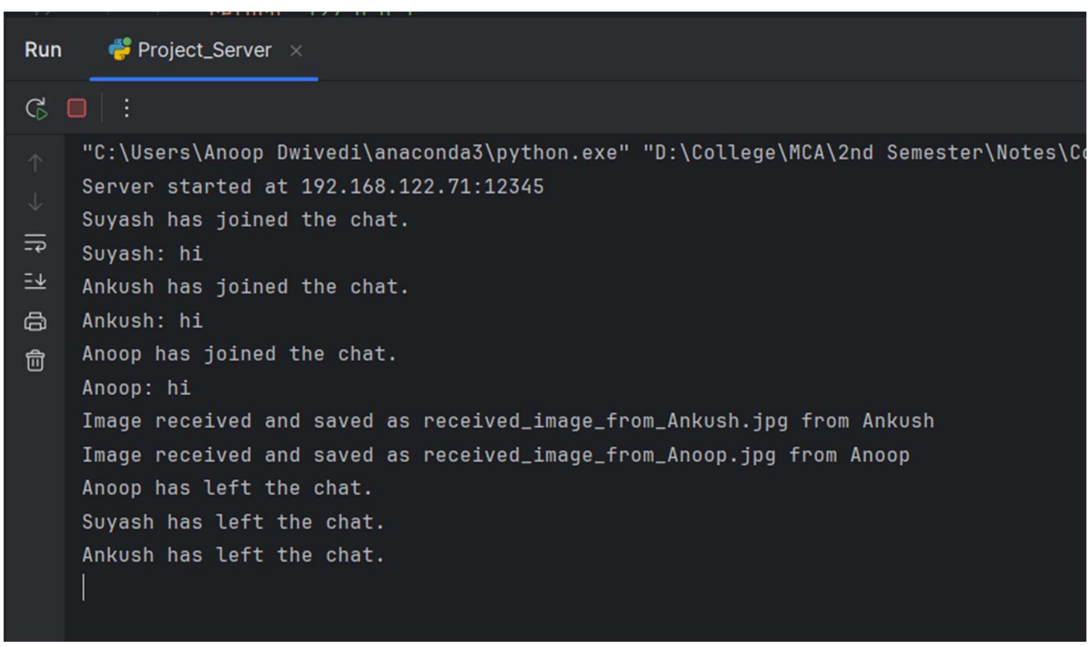
6.1 Text Message Exchange



6.2 Image Transfer Output




OUTPUT
TERMINAL
Code
PS C:\Users\manda\OneDrive\Desktop\DSA\DSA test> python -u "c:\Use
rs\manda\OneDrive\Desktop\DSA\DSA test\anoop.py"
Enter your name: Anoop
Enter message (or type 'send image' to send an image, or 'exit' to
leave): hi
Enter message (or type 'send image' to send an image, or 'exit' to
leave):
tmph5ngm06b.PNG


Run
Project_Server
"C:\Users\Anoop Dwivedi\anaconda3\python.exe" "D:\College\MCA\2nd Semester\Notes\C
Server started at 192.168.122.71:12345
Suyash has joined the chat.
Suyash: hi
Ankush has joined the chat.
Ankush: hi
Anoop has joined the chat.
Anoop: hi
Image received and saved as received_image_from_Ankush.jpg from Ankush
Image received and saved as received_image_from_Anoop.jpg from Anoop
Anoop has left the chat.
Suyash has left the chat.
Ankush has left the chat.

7. OBSERVATIONS

The UDP-based client-server chat system with image transfer provided several important insights into network communication and system performance.

1. UDP-Based Communication

- The system uses **User Datagram Protocol (UDP)**, which is **faster** than TCP but does not guarantee message delivery.
- Due to its **connectionless nature**, messages may be lost in unstable network conditions.

2. Simultaneous Sending & Receiving

- The client uses **multithreading**, allowing it to **send** and **receive** messages at the same time.
- This ensures a **real-time chat experience** without blocking message flow.

3. Image Transfer & Compression

- The system allows users to **send images**, which are **compressed** before transmission to reduce bandwidth usage.
- Compression helps in **faster transmission**, but **large images** may still exceed the buffer limit.

4. User Identification & Message Handling

- Messages include the **sender's name**, making it easy to identify users in group chats.
- The system correctly differentiates between **text messages** and **images**, ensuring proper handling.
- When users **join** or **leave**, a system message is displayed to notify others.

5. Limitations & Challenges

- **No message delivery guarantee** due to UDP's unreliable nature.
- **No encryption**, meaning messages and images are sent in plaintext, posing a security risk.
- **No message history storage**, so users cannot access old messages after exiting the chat.

6. Overall Performance

- The chat system is **fast and efficient**, making it suitable for real-time communication.
- Further improvements, such as **error correction**, **encryption**, and **a graphical interface**, could enhance usability and security.

8. LEARNING OUTCOME

Through this mini-project, I have gained valuable insights into network communication, socket programming, and multimedia data handling. The key learning outcomes include:

1. Understanding Socket Programming in Python

- Learned how to create and manage **sockets** in Python using the socket library.
- Explored **UDP-based communication** and how it differs from TCP.
- Understood how **datagram sockets** work in real-time applications.

2. Learning UDP-Based Communication Mechanisms

- Implemented **connectionless communication** using UDP sockets.
- Understood the advantages and limitations of **UDP vs. TCP** in terms of **speed, reliability, and overhead**.
- Explored **buffer size management** and its impact on data transmission.

3. Handling Multimedia Data in Networks

- Worked with the **PIL (Pillow) library** for **image processing and compression** before transmission.
- Understood the importance of **reducing file size** while maintaining acceptable quality for **efficient data transfer**.
- Implemented techniques to **differentiate between text and image messages**.

4. Implementing Multithreading for Efficient Communication

- Used **Python's threading module** to enable **simultaneous sending and receiving** of messages.
- Understood the importance of **non-blocking I/O operations** for **real-time applications**.
- Learned how **multithreading improves user experience** in chat-based applications.

9. Future Enhancements

- Implement TCP for more reliable messaging.
- Add end-to-end encryption for privacy.
- Implement a database for chat history storage

10. CONCLUSION

This mini-project provided an in-depth understanding of **client-server communication using UDP sockets**. By implementing a real-time chat system with **text and image transmission capabilities**, I explored the core principles of **network programming, socket handling, and multimedia data processing**.

One of the key takeaways was learning how **UDP differs from TCP** in terms of **speed, reliability, and connection management**. Since UDP is **connectionless and faster**, it is well-suited for applications that prioritize **low latency over guaranteed delivery**, such as real-time chat, VoIP, and gaming. However, this also introduced **challenges like packet loss and data integrity concerns**, reinforcing the importance of **error-handling mechanisms** in real-world applications.

Additionally, the project emphasized **image processing and compression** techniques. Using the **Pillow (PIL) library**, I implemented **image resizing and quality reduction** to ensure efficient transmission without exceeding buffer limitations. This knowledge is crucial for optimizing **network bandwidth** in multimedia applications.

Furthermore, I gained insights into **multithreading for concurrent message handling**, which significantly improved **responsiveness and user experience**. Implementing **basic security considerations**, such as **handling unknown data and preventing unauthorized access**, also highlighted the importance of **network security** in communication systems.

Overall, this project strengthened my understanding of **network programming, real-time data transmission, and system optimization**. It provided a practical foundation for **developing scalable and secure communication systems**, which is essential for real-world applications in networking and software development.

11. REFERENCES

- Forouzan A. Behrouz, *Data Communications And Networking (Vol.4)*, McGraw-Hill Forouzan networking series, 2007.
- Stevens, W. Richard. *UNIX Network Programming, Volume 1: The Sockets Networking API.* Addison-Wesley, 2003.
- Comer, Douglas E. *Computer Networks and Internets*. Pearson, 2014.
- Python Software Foundation, "Python Socket Programming," https://docs.python.org/3/library/socket.html
- Python Imaging Library (PIL), "Pillow Image Processing," https://pillow.readthedocs.io/
- Real Python, "Socket Programming in Python – A Complete Guide," https://realpython.com/python-sockets/
- GeeksforGeeks. "Client-Server Model in Computer Network." *GeeksforGeeks*, https://www.geeksforgeeks.org/client-server-model/.