LAU
School of Arts and Sciences

Department of Computer Science & Mathematics

CHATX : SECURE PEER TO PEER COMMUNICATION PLATFORM

Project Report

Submitted by:

Michael Geha: 202203529

Course: Computer Networks
Instructor: Dr. Ayman Tajjedine
Date: December 30, 2025

1. Table Of Content

2. Introduction

This project presents ChatX, a secure peer-to-peer communication platform designed and implemented as part of our Computer Networks coursework. The system provides real-time text messaging and file transfer capabilities using a combination of TCP and UDP sockets. Unlike centralized messaging systems, ChatX distributes the communication load directly between clients while using a lightweight server solely for user registration and peer discovery.

The goal of this project is to translate theoretical networking concepts into a functioning system that demonstrates the practical use of socket programming, concurrency, graphical user interfaces, and message exchange protocols. The final application integrates a PyQt-based GUI, a central authentication and discovery server, and decentralized direct messaging between clients. Through this architecture, ChatX offers a complete demonstration of client–server and peer-to-peer communication in Python.

3. Project Objectives

The main objectives of the ChatX project were:

1. Implement a fully functional client server architecture for registration and online peer tracking.
2. Enable direct peer-to-peer communication using TCP for messaging and UDP for file transfer.
3. Design a responsive graphical interface that allows users to view peers, send messages, and transfer files.
4. Ensure concurrency and responsiveness through the use of multithreading.
5. Apply structured JSON protocols for communication between system components.
6. Demonstrate real-world networking concepts such as connection handling, multi-threaded listeners, packet chunking, and asynchronous updates.

These objectives guided our design decisions and ensured that ChatX remained aligned with the course's learning outcomes.

4. Individual Role and Contribution

• Michael Geha

- Developed and tested the TCP messaging system
- Implemented connection handling, message formatting, chat logging and encryption.
- Contributed to system debugging and logging
- Designed and implemented the UDP file transfer subsystem
- Integrated chunking, file reconstruction, and callback mechanisms
- Ensured reliable directory handling and progress reporting
- Implemented the GUI and user interaction layer using PyQt
- Managed the signal/slot communication and chat display logic
- Integrated client components and final application behavior

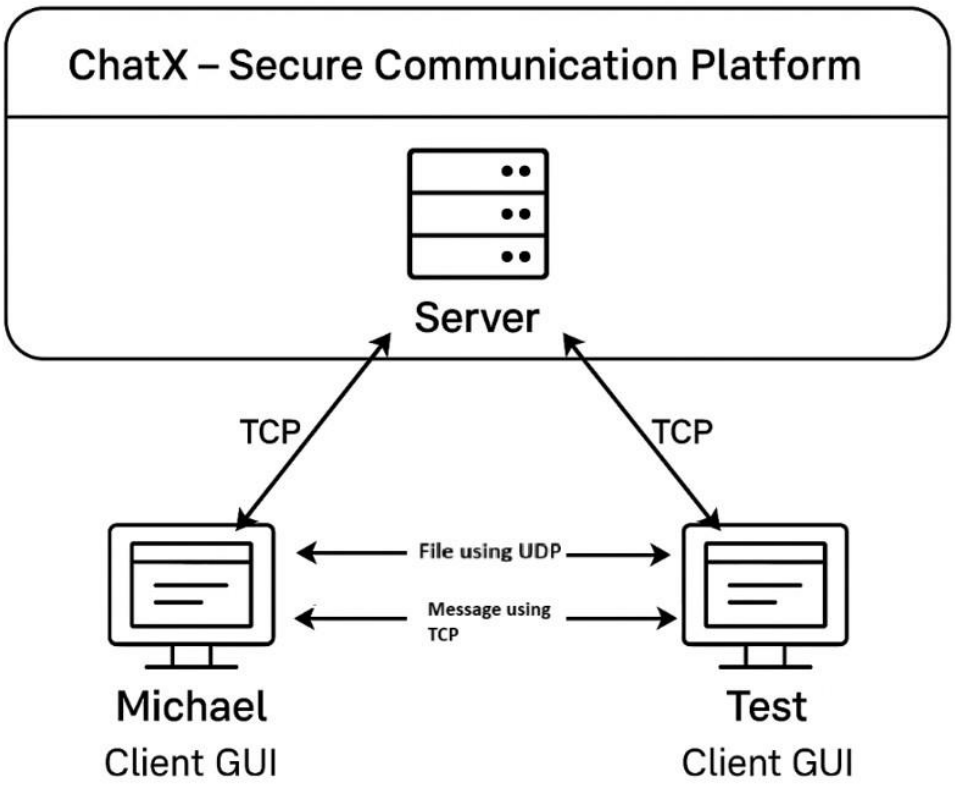
5. System Architecture

The ChatX architecture is divided into two primary layers:

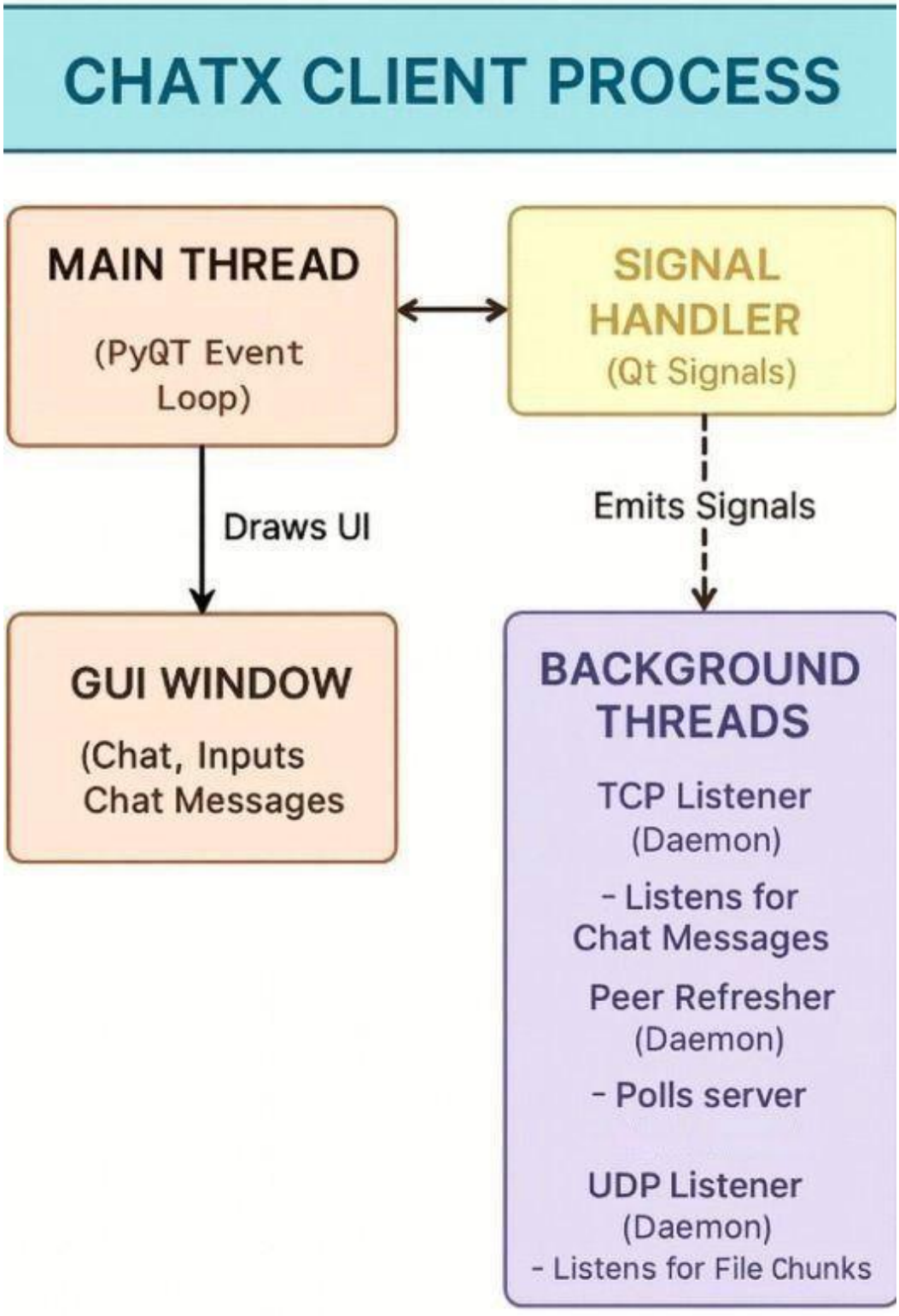
1. **Central Server (Registration + Peer Discovery)**
2. **Client Layer (GUI + TCP + UDP communication)**

This hybrid architecture was selected because it balances simplicity with realism. A fully decentralized model would require NAT traversal and more complex routing logic; instead, our system uses centralized discovery followed by decentralized message exchange.

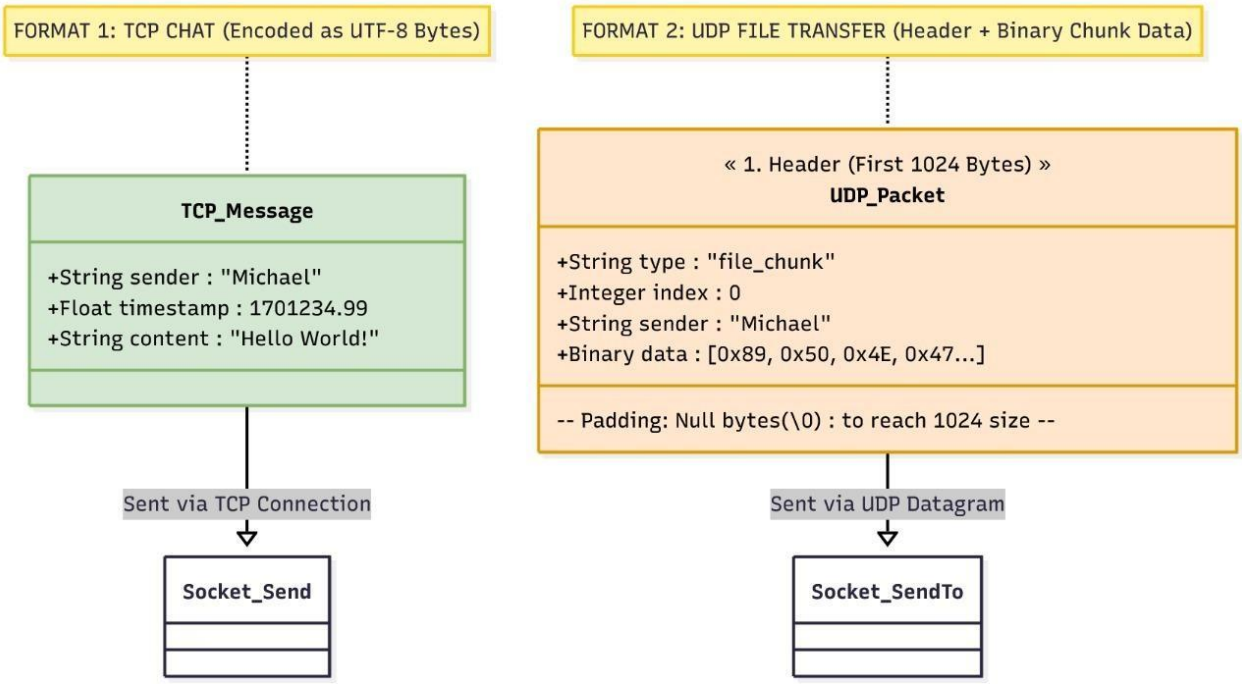
5.1 Architecture Diagram



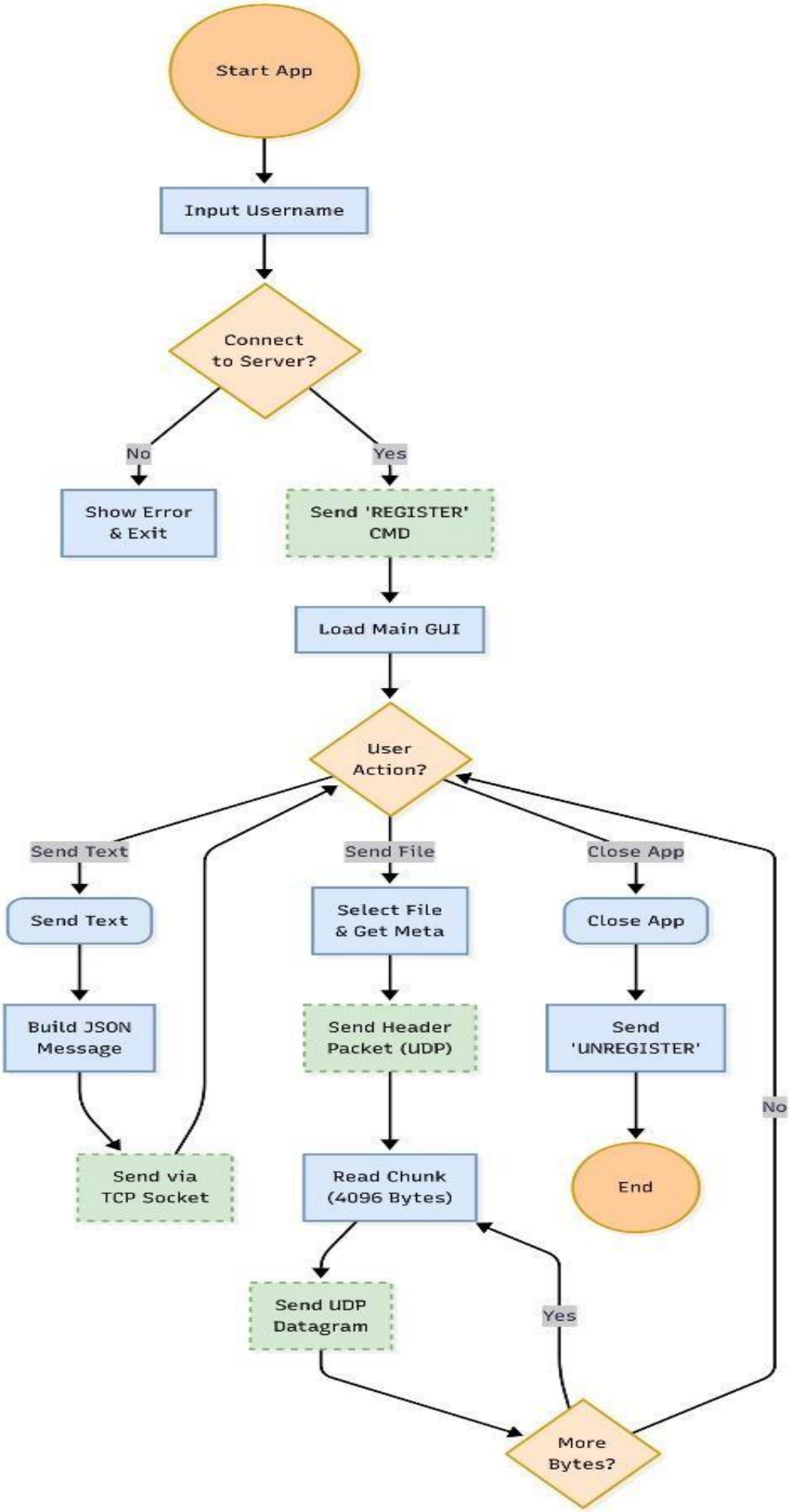
5.2 Threading Diagram



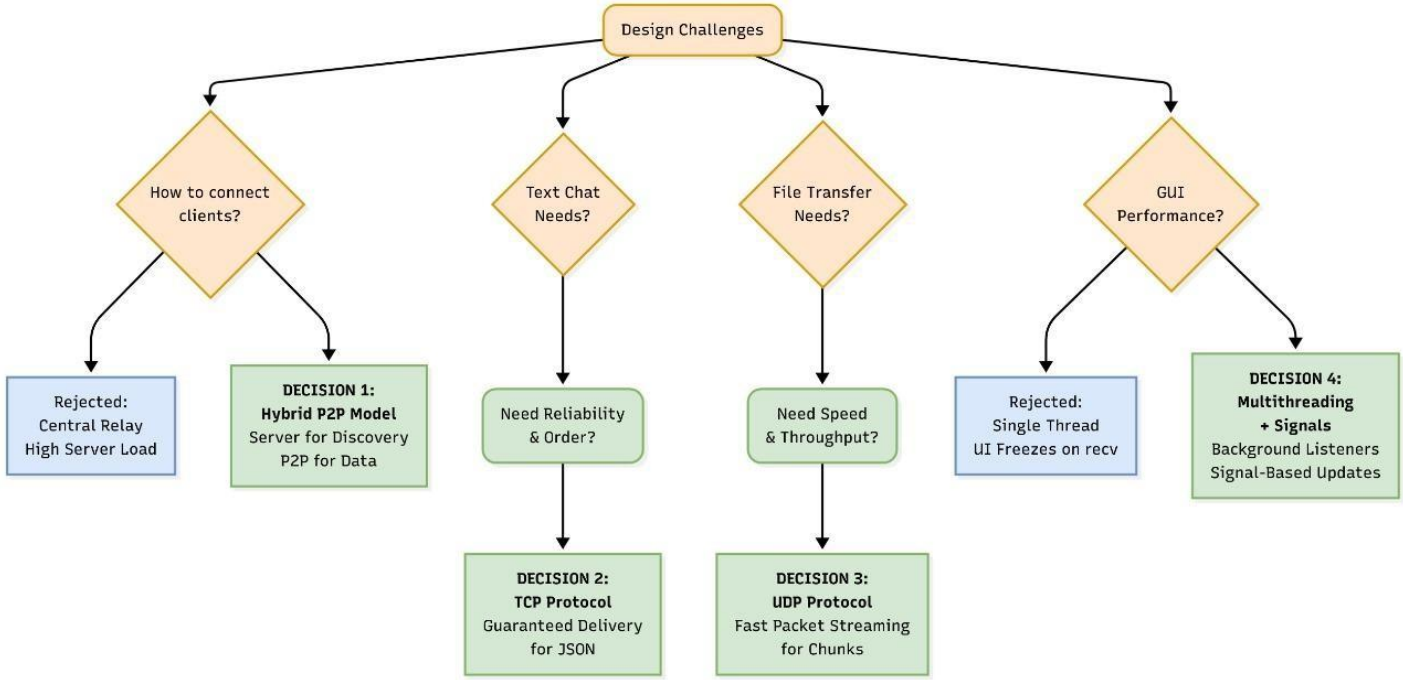
5.3 Message/File Formats Diagram



5.4 Instructions Diagram


Start App
Input Username
Connect to Server?
No
Yes
Show Error & Exit
Send 'REGISTER' CMD
Load Main GUI
User Action?
Send Text
Send File
Close App
Send Text
Select File & Get Meta
Close App
Build JSON Message
Send Header Packet (UDP)
Send 'UNREGISTER'
Send via TCP Socket
Read Chunk (4096 Bytes)
End
Send UDP Datagram
Yes
No
More Bytes?

5.5 Design Decisions Diagram



6. Server Design

The server (implemented in server.py) operates as a lightweight coordination point. Its responsibilities include:

- Receiving and processing register, get_peers, and unregister commands.
- Maintaining a synchronized dictionary of online peers.
- Managing concurrent client connections using threads.

6.1 Peer Registry Structure

The server stores peers as:

```
self.peers = {
   "username": {
      "ip": "127.0.0.1",
      "tcp_port": 5010,
      "udp_port": 6010
   }
}
```

6.2 Request Processing

The process_request() method supports:

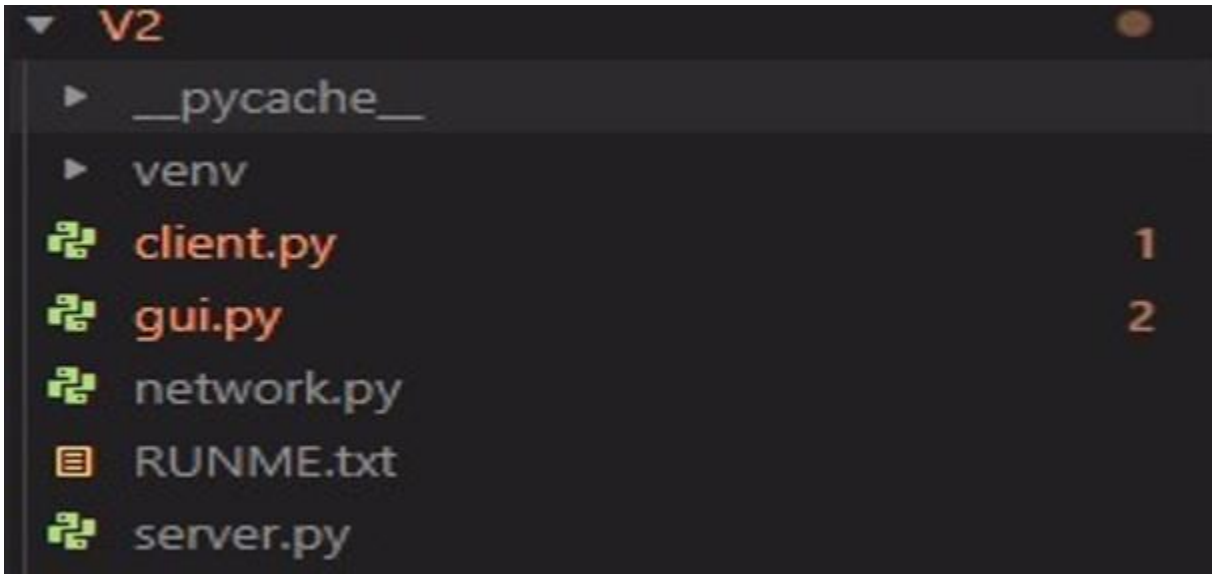
- register — Inserts the peer into the registry
- get_peers — Returns all active peers
- unregister — Removes the peer

The server does *not* relay messages or files. Its simplicity keeps performance high and avoids congestion.

6.3 Project File Structure

To maintain modularity and clarity, the ChatX project is organized into several Python modules, each responsible for a specific subsystem. The directory structure is shown below. This

organization ensures a clean separation of concerns and makes the system easier to maintain and debug.



6.4 Server Startup and Listening State

When the ChatX server is launched, it initializes the peer registry and begins listening for incoming TCP connections on port 5000. The server runs in a blocking loop where it continuously accepts and processes client requests, including registration, peer listing, and unregistration. The console output below demonstrates the server's successful initialization and readiness to handle clients.

```
(venv) root@DESKTOP-92MIPSF:~/.vscode-server/V2# python3 server.py
*** Server started successfully! ***
Listening for connections on 0.0.0.0:5000
Waiting for clients to connect...
--------------------------------
```

After startup, the server begins receiving incoming client connections. Each time a client registers, the server logs the connection details, including the client's IP address and the dynamically assigned TCP and UDP ports. This provides a clear confirmation that the registration process is functioning correctly. The example below shows a successful connection and peer registration event.

```
(venv) root@DESKTOP-92MIPSF:~/.vscode-server/V2# python3 server.py
*** Server started successfully! ***
Listening for connections on 0.0.0.0:5000
Waiting for clients to connect...
--------------------------------
[+] New connection from ('127.0.0.1', 53002)
[+] Registered peer: Michael at 127.0.0.1:48795/55359
```

In addition to the first client registration, the server also handles subsequent client connections seamlessly. When the second client (“Test”) is launched, the server logs a new TCP connection request and registers the peer with its associated TCP and UDP ports. This confirms that the server correctly maintains multiple active peers in its registry.

```
[+] New connection from ('127.0.0.1', 41720)
[+] Registered peer: Test at 127.0.0.1:59065/60735
```

7. Client Design

The client (in client.py) integrates several subsystems:

• PyQt5 GUI (gui.py)

• TCP Messaging (TCPChatClient)

• UDP File Transfer (UDPFileTransfer)

• Server Communication (ServerConnector)

• Thread Management (ThreadSafeUpdater)

The client uses a clean and modular structure, and each subsystem is initialized in the ChatXClient constructor.

7.1 Graphical User Interface

The GUI includes:

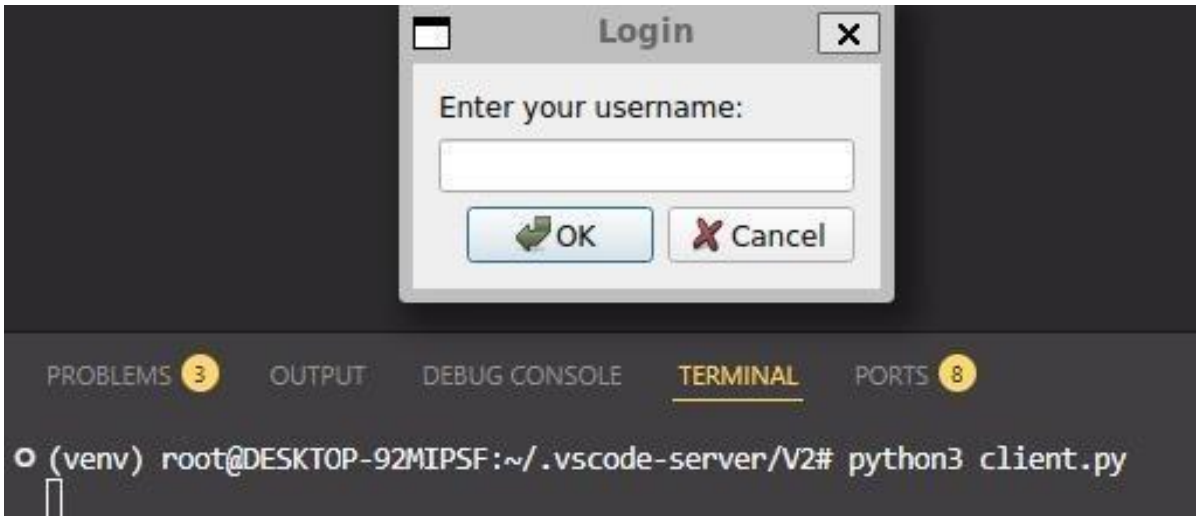
- Online peers list
- Chat display
- Message input
- File selection and progress bar
- Status bar
- Signals for thread-safe UI updates

The signal system ensures that network data received in background threads is safely displayed.

7.1.1 Client Initialization and Login Process

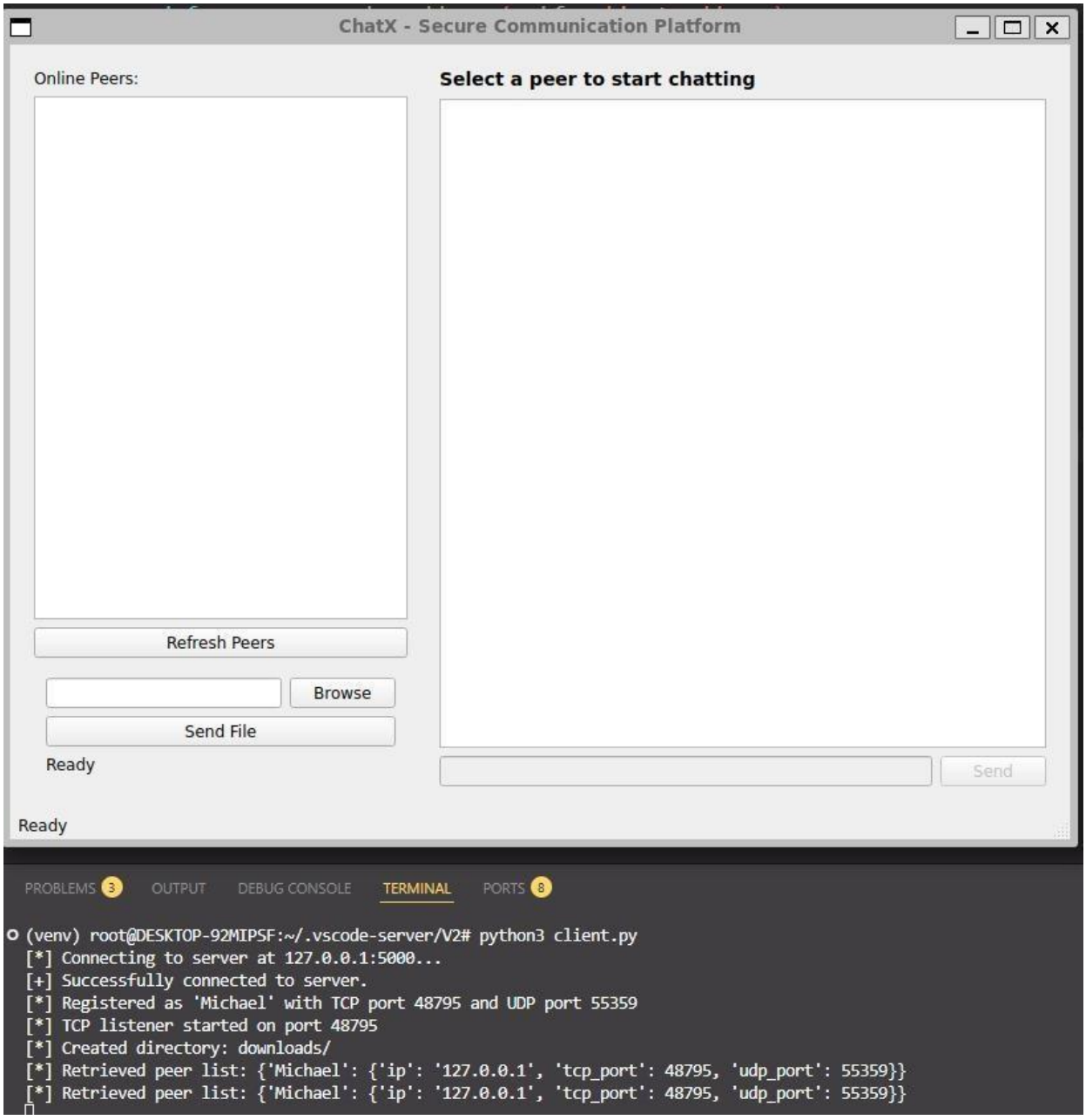
When the ChatX client application is launched, the system immediately prompts the user to enter a unique username. This login step establishes the client's identity and is essential for peer

registration on the central server. The prompt is implemented using a PyQt5 input dialog, ensuring a simple and intuitive user interaction. After the username is confirmed, the client proceeds to initialize the TCP listener, the UDP file transfer handler, and the communication session with the server.



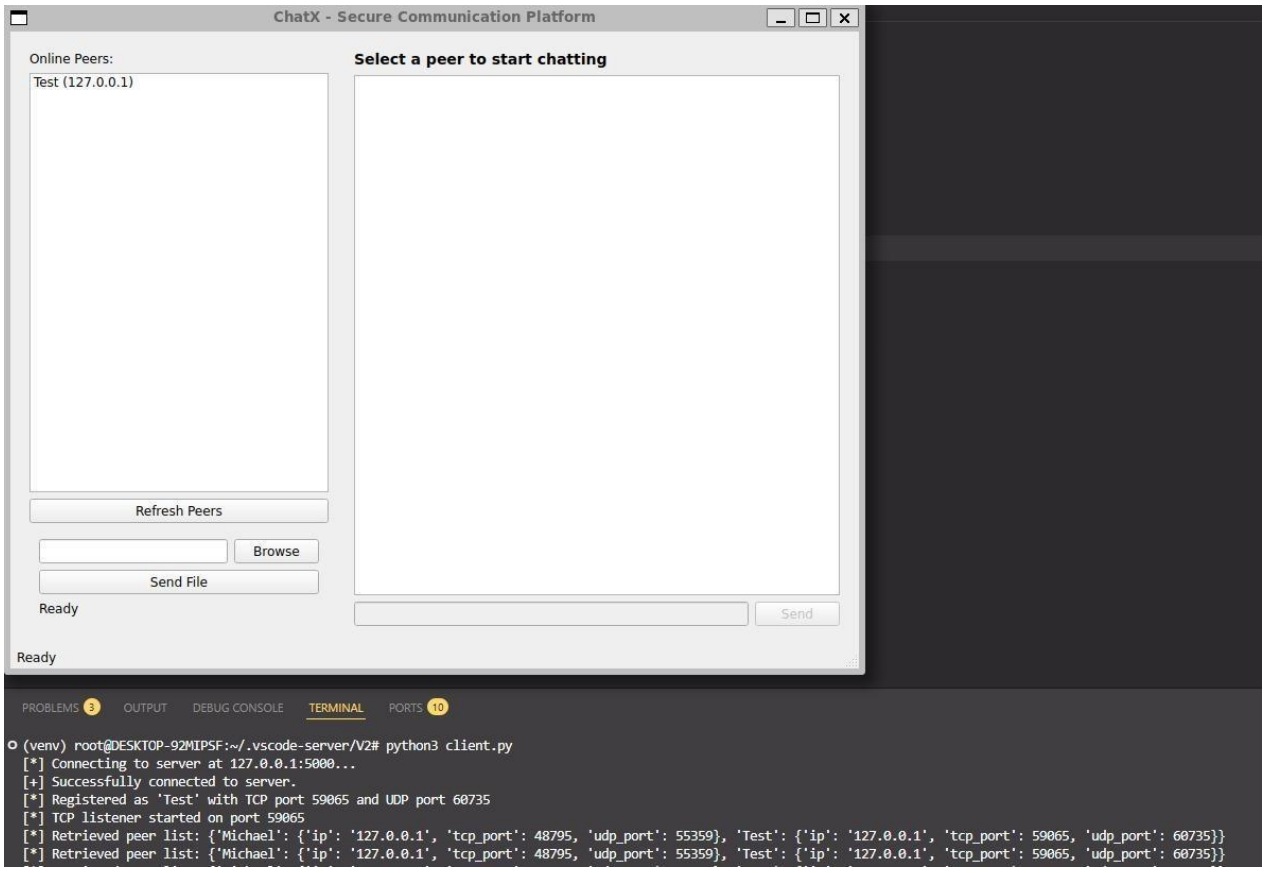
7.1.2 Main Interface After Login

After the user enters a valid username and the client registers with the central server, the main ChatX interface becomes fully active. The graphical layout displays the list of online peers on the left, the chat area on the right, and the controls for refreshing peers and sending files at the bottom. At this stage, the client completes several internal processes, including establishing a TCP listener, initializing the UDP file transfer handler, creating the local download directory, and retrieving the latest peer list from the server. The figure below shows the fully initialized client interface along with the corresponding terminal output.


ChatX - Secure Communication Platform
Online Peers:
Select a peer to start chatting
Refresh Peers
Browse
Send File
Ready
Send
Ready
PROBLEMS 3
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS 8
(venv) root@DESKTOP-92MIPSF:~/.vscode-server/V2# python3 client.py
[*] Connecting to server at 127.0.0.1:5000...
[+] Successfully connected to server.
[*] Registered as 'Michael' with TCP port 48795 and UDP port 55359
[*] TCP listener started on port 48795
[*] Created directory: downloads/
[*] Retrieved peer list: {'Michael': {'ip': '127.0.0.1', 'tcp_port': 48795, 'udp_port': 55359}}
[*] Retrieved peer list: {'Michael': {'ip': '127.0.0.1', 'tcp_port': 48795, 'udp_port': 55359}}

7.1.3 Online Peer Discovery

The ChatX interface retrieves the list of active peers through a periodic server request. Once the server returns the updated registry, the GUI displays all available users along with their corresponding IP addresses. This enables the client to select a peer and initiate a direct TCP-based chat session. The screenshot below shows a successful peer discovery event, where the client detects another active user ("Test") and updates the Online Peers panel accordingly.



7.2 TCP Messaging Subsystem

The TCPChatClient class in network.py handles:

- Hosting a local TCP listener
- Accepting incoming peer connections
- Initiating outgoing TCP connections
- Sending/receiving JSON-formatted messages

- Running message handling in dedicated threads

Message Format

{

"sender": "Michael",

"timestamp": 1732789444.452,

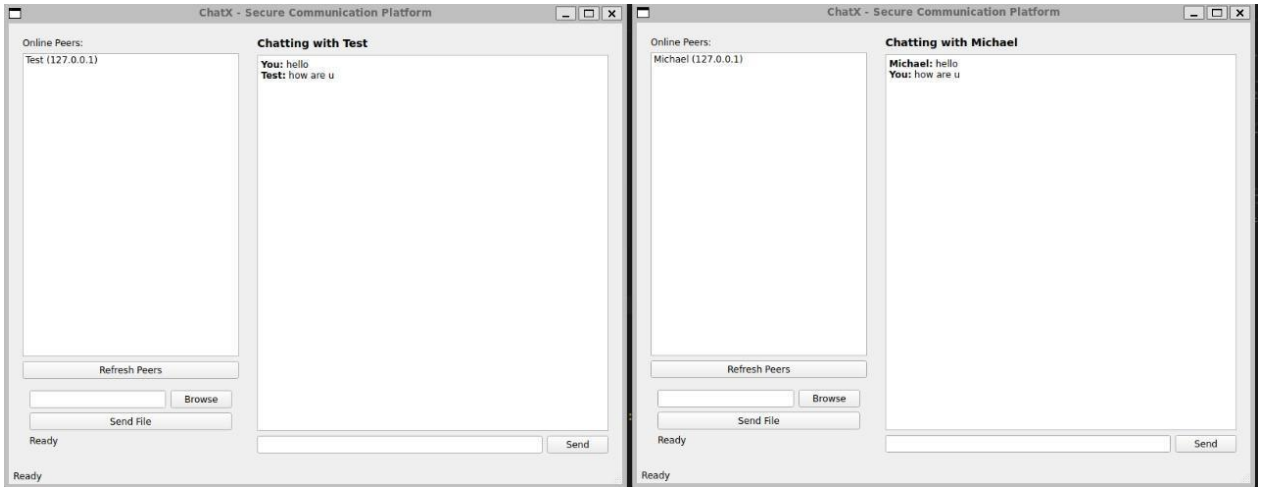
"content": "Hello!"

}

TCP guarantees reliable, ordered delivery, making it ideal for chat messages.

7.2.1 Real-Time TCP Message Exchange

Once a peer-to-peer TCP connection is established, ChatX allows users to exchange text messages in real time. Each message is encoded as a JSON object and transmitted through the dedicated TCP socket. The receiving client processes the incoming message in a background thread and updates the GUI through a thread-safe signal. The screenshot below demonstrates a successful two-way message exchange between two clients (“Michael” and “Test”), confirming that the TCP messaging subsystem is functioning correctly.



7.3 UDP File Transfer Subsystem

The UDPFileTransfer class in network.py implements:

• **Header Packet (Metadata)**

Includes filename, file size, sender, and total chunk count.

• **Chunk Transmission**

Each file is split into uniform packet payloads and transmitted sequentially.

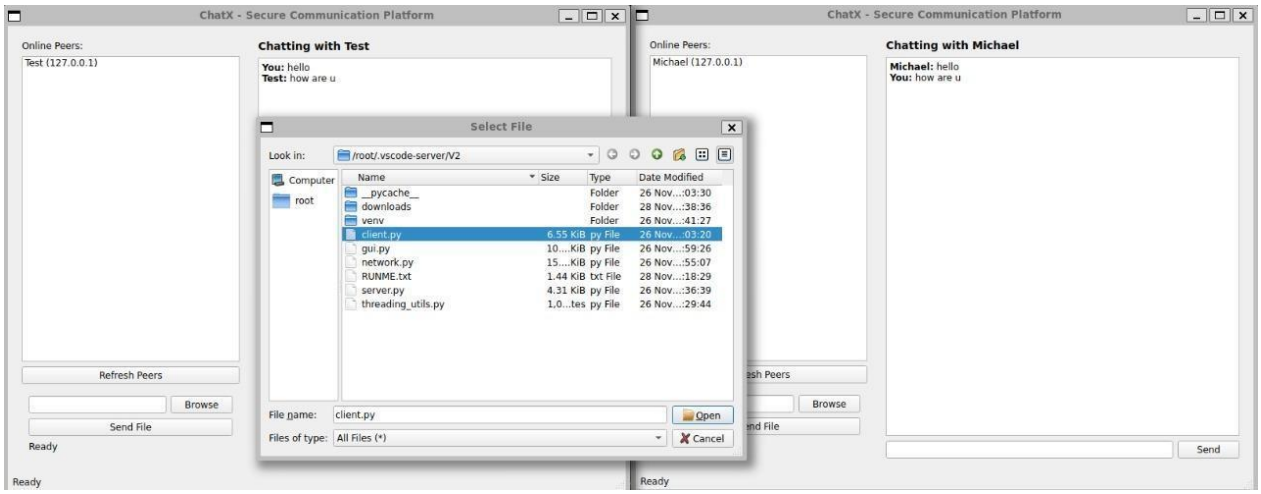
• **File Reconstruction**

The receiver writes chunks to a temporary file and renames it after completion.

UDP provides high transfer speed without blocking the chat system.

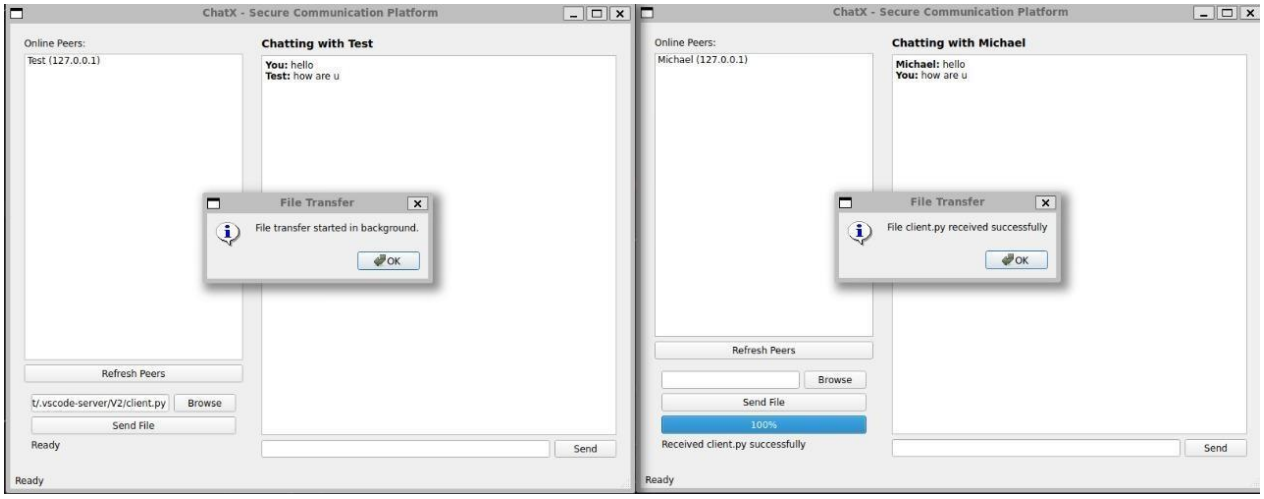
7.3.1 File Selection Interface

Before initiating a UDP-based file transfer, the sender must select the file to be transmitted. ChatX uses a standard PyQt5 file dialog to allow users to browse their local directories and choose any file type. Once the file is selected, the client extracts the file size, prepares the metadata header, and begins splitting the file into UDP packets. The screenshot below shows the file selection dialog open while two clients ("Michael" and "Test") are engaged in an active chat session, demonstrating the seamless integration between messaging and file transfer features.



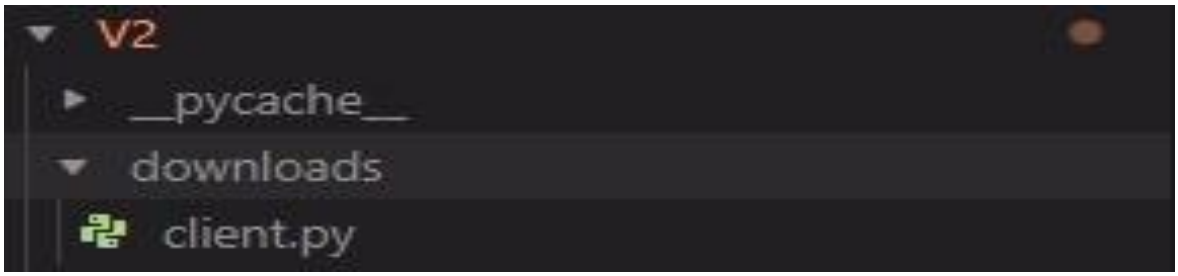
7.3.2 File Transfer Execution and Completion

Once the sender selects a file, ChatX initiates a background UDP transfer, splitting the file into individual packets and transmitting them to the peer's UDP port. The receiver reconstructs the file from these packets and updates the progress bar accordingly. Upon completion, both clients display a confirmation message. The screenshot below illustrates a successful file transfer of client.py, where the sender receives a "File transfer started" notification and the receiver confirms the successful reception with a 100% progress indicator.



7.3.3 File Reconstruction and Storage

After all UDP packets are received, ChatX reconstructs the original file by assembling the chunks in order and writing them to the local *downloads* directory. The receiver validates the file header, determines the output path, and performs the write operation once all chunks have been collected. The screenshots below confirm that the transferred file (*client.py*) was reconstructed successfully and stored in the appropriate directory. The terminal logs show the file header detection, save path resolution, and final write confirmation.




```
[*] File header received from ('127.0.0.1', 55359): filename=client.py, size=6703, chunks=2
[*] Will save incoming file to: /root/.vscode-server/V2/downloads/client.py
[+] File 'client.py' saved to '/root/.vscode-server/V2/downloads/client.py'
```

7.4 ServerConnector

The ServerConnector class manages:

- Connecting to the server
- Registering a user
- Requesting the peer list
- Unregistering gracefully

This class encapsulates server communication and simplifies client logic.

7.5 Threading & Asynchronous Processing

Each component runs in its own thread:

- TCP listener
- UDP listener
- Periodic peer refresh
- File sending thread

8. Implementation Highlights

Below are meaningful excerpts linked to actual code to demonstrate the correctness of the ChatX implementation.

8.1 Dynamic Port Allocation

From client.py (ChatXClient.find_free_port):

```
def find_free_port(self):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind(('127.0.0.1', 0))
        port = s.getsockname()[1]
    return port
```

This method lets the operating system choose a free port, ensuring multiple clients on the same machine avoid port conflicts.

8.2 Client Login, Registration, and Startup

From client.py (ChatXClient.init and init_client):

```
self.tcp_port = self.find_free_port()
self.udp_port = self.find_free_port()
self.server_connector = ServerConnector(server_host, server_port)
self.server_connector.register(self.username, self.tcp_port, self.udp_port)
self.tcp_client = TCPChatClient(self.username, self.tcp_port)
self.udp_transfer = UDPFileTransfer(self.username, self.udp_port)
```

These lines show how the client selects ports, connects to the central server, registers itself, and initializes both TCP chat and UDP file transfer components.

8.3 Multithreaded TCP Listener and Connection Handling

From network.py (TCPChatClient.start_listener and listen_for_connections):

```
self.server_socket.listen(5)
self.listener_thread = threading.Thread(target=self.listen_for_connections)
client_socket, client_address = self.server_socket.accept()
```

The TCP chat client runs a background thread that continuously accepts new incoming connections without blocking the GUI, enabling multiple simultaneous chat sessions.

From network.py (TCPChatClient.handle_connection):

```
data = client_socket.recv(4096).decode('utf-8')
message = json.loads(data)
self.message_callback(message, client_address)
```

Each connection is handled in its own thread, which receives, parses, and forwards messages to the GUI via a callback.

8.4 Encrypted Message Sending and Receiving

From network.py (TCPChatClient.encrypt_message, decrypt_message, send_message):

```
encrypted_content = self.encrypt_message(message)
message_data = {"sender": self.username, "timestamp": time.time(), "content": encrypted_content}
self.active_connections[peer_address].send(message_json)
```

Messages are encrypted before transmission and decrypted on receipt, demonstrating secure end-to-end messaging over TCP.

From network.py (TCPChatClient.handle_connection):

```
encrypted_content = message.get("content", "")
decrypted_content = self.decrypt_message(encrypted_content)
message["content"] = decrypted_content
```

This shows how incoming messages are decrypted before being passed to the application logic.

8.5 GUI-Based Peer Selection and Chat Display

From gui.py (MainWindow.update_peer_list and select_peer):

```
item = QListWidgetItem(f"{username} ({info['ip']})")
item.setData(Qt.UserRole, info)
self.peer_list.itemClicked.connect(self.select_peer)

peer_info = item.data(Qt.UserRole)
self.client.tcp_client.connect_to_peer(peer_ip, peer_port)
```

The GUI dynamically populates the peer list and uses hidden metadata to connect to a selected peer's TCP server when the user clicks on a name.

From gui.py (MainWindow.send_message and display_incoming_message):

```
if self.client.tcp_client.send_message(message, self.current_peer_address):
    self.add_message_to_chat(self.current_peer, "You", message)

sender = message.get("sender", "Unknown")
content = message.get("content", "")
self.add_message_to_chat(sender, sender, content)
```

These methods connect the GUI to the TCP layer, sending messages to the selected peer and updating the chat history for both outgoing and incoming messages.

8.6 UDP-Based File Transfer with Progress Feedback

From network.py (UDPFileTransfer.send_file and _send_file_worker):

```
send_thread = threading.Thread(target=self._send_file_worker, args=(filename, peer_ip, peer_port))
header_data = json.dumps(header).encode('utf-8')
self.socket.sendto(header_data, (peer_ip, peer_port))
self.socket.sendto(packet, (peer_ip, peer_port))
```

File transfers are performed in a background thread using UDP datagrams. A structured header plus fixed-size chunks are sent to the receiver.

From network.py (UDPFileTransfer.listen_for_files):

```
header_raw = data[:1024].split(b'\0', 1)[0].decode('utf-8', errors='ignore')
header = json.loads(header_raw)
self.file_callback("progress", filename, filesize, progress, addr)
```

The receiver reconstructs the file from incoming chunks and notifies the GUI with progress updates via a callback.

From gui.py (FileTransferWidget.update_file_status):

```
if status == "start":
    self.progress_bar.setVisible(True)
    self.progress_bar.setValue(0)
elif status == "progress":
    self.progress_bar.setValue(int(progress))
elif status == "complete":
    self.progress_bar.setValue(100)
```

These lines show how the GUI displays real-time progress of incoming file transfers using a progress bar and status label.

8.7 Central Registry Server for Peer Discovery

From server.py (Server.start):

```
self.server_socket.bind((self.host, self.port))
client_socket, client_address = self.server_socket.accept()
client_thread = threading.Thread(target=self.handle_client, args=(client_socket, client_address))
```

The central server listens on a fixed TCP port and spawns a new thread for each client, allowing concurrent registrations and requests.

From server.py (Server.process_request):

```
if command == "register":
    self.peers[username] = {"ip": client_address[0], "tcp_port": tcp_port, "udp_port": udp_port}
elif command == "get_peers":
    return {"status": "success", "peers": self.peers}
elif command == "unregister":
    del self.peers[username]
```

This method implements the registry protocol used by all clients to register, retrieve the global peer list, and unregister when they disconnect.

8.8 Full TCP Chat Core Implementation

From network.py TCPChatClient (main TCP logic: encryption, listener, handler, connect, send):

```
class TCPChatClient:
    def __init__(self, username, tcp_port):
        self.username = username
        self.tcp_port = tcp_port
        self.listening = False
        self.listener_thread = None
        self.message_callback = None

        # MUTEX LOCK: Critical for thread safety.
        self.lock = threading.Lock()

        self.server_socket = None

        # This dictionary stores all active open sockets.
        # Key: (IP, Port) tuple. Value: The socket object.
        self.active_connections = {}  # { (ip, port): socket }

        # This key must be the SAME for all clients.
        self.key = b'8P_GkC-1l0j3g5s7q9z1x3c5v7b9n1m3k5j7h9g1f3d='
        self.cipher = Fernet(self.key)

        print(f"DEBUG: My Encryption Key is {self.key}")
```

```
    # Encrypting the message
    def encrypt_message(self, message):
        return self.cipher.encrypt(message.encode()).decode()

    # Decrypting the message
    def decrypt_message(self, encrypted_message):
        try:
            return self.cipher.decrypt(encrypted_message.encode()).decode()
        except:
            return "[Decryption Failed]"

    def start_listener(self):
        """Start a TCP server to listen for incoming connections."""
        # Create a TCP/IP socket (IPv4, Stream/TCP)
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        # BIND: Reserve the specific port (e.g., 5001) on this machine.
        self.server_socket.bind(('127.0.0.1', self.tcp_port))

        # LISTEN: Tell the OS to start accepting connections on this port.
        # '5' is the backlog (queue size).
        self.server_socket.listen(5)

        self.listening = True

        # Spawn a background thread to handle the 'accept()' loop.
        self.listener_thread = threading.Thread(target=self.listen_for_connections)
        self.listener_thread.daemon = True  # Thread dies if the app closes
        self.listener_thread.start()
        print(f"[*] TCP listener started on port {self.tcp_port}")
```

Def start_listener(self): Creates a TCP server on the client machine and starts a background thread to accept chat connections from peers.

```python
def listen_for_connections(self):
    """Listen for incoming connections and spawn threads to handle them."""
    while self.listening:
        try:
            # ACCEPT: This line BLOCKS (waits) until a peer connects.
            client_socket, client_address = self.server_socket.accept()
            print(f"[+] New incoming TCP connection from {client_address}")

            # SAFETY: Lock the dictionary before adding the new connection.
            with self.lock:
                self.active_connections[client_address] = client_socket
                print(f"[*] Stored incoming connection. Active: {list(self.active_connections.

            # Create a NEW thread specifically for this one peer.
            conn_thread = threading.Thread(
                target=self.handle_connection,
                args=(client_socket, client_address)
            )
            conn_thread.daemon = True
            conn_thread.start()

        except Exception as e:
            if self.listening:
                print(f"[!] Error accepting TCP connection: {e}")
```

Def listen_for_connection : Waits for peers to connect, creates a dedicated handler thread for each incoming TCP connection.

```python
def handle_connection(self, client_socket, client_address):
    """Handle messages from a specific connection."""
    try:
        # Infinite loop to constantly listen for messages from this specific peer
        while True:
            # RECV: Block and wait for data. Read up to 4096 bytes.
            data = client_socket.recv(4096).decode('utf-8')

            # If recv returns empty data, it means the other side closed the connection.
            if not data:
                print(f"[-] Connection closed by peer {client_address}")
                break

            try:
                # Deserialize JSON string back to Python Dictionary
                message = json.loads(data)

                # The content coming in is encrypted. We must unlock it.
                encrypted_content = message.get("content", "")
                decrypted_content = self.decrypt_message(encrypted_content)

                # Update the message object with the readable text
                message["content"] = decrypted_content

                print(f"[*] Received message from {client_address}: {message}")

                sender = message.get("sender", "Unknown")
                content = message.get("content", "")
                logging.info(f"[CHAT RECEIVED] From {sender} ({client_address}): {content}")
```

```
                # Notify the GUI (via the callback function in client.py)
                if self.message_callback:
                    self.message_callback(message, client_address)
            except json.JSONDecodeError:
                print("[-] Received invalid JSON message")
    except Exception as e:
        print(f"[!] Error handling connection from {client_address}: {e}")
    finally:
        # CLEANUP: If the loop breaks (error or disconnect), remove them from the list.
        with self.lock:
            if client_address in self.active_connections:
                del self.active_connections[client_address]
                print(f"[*] Removed connection for {client_address}. Active: {list(self.active
        try:
            client_socket.close()
        except:
            pass
```

Def handle_connection(): Receives encrypted TCP messages, decrypts them using Fernet, and forwards them to the GUI.

```python
def connect_to_peer(self, peer_ip, peer_port):
    """Connect to a peer's TCP server."""
    peer_address = (peer_ip, peer_port)
    try:
        # CHECK: Do we already have an open line to this person?
        if peer_address in self.active_connections:
            print(f"[!] Already connected to {peer_address}")
            return True

        # CREATE: Make a NEW socket.
        peer_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        # CONNECT: Dial the peer's listening port (e.g., 5001).
        peer_socket.connect(peer_address)
        print(f"[+] Successfully connected to peer at {peer_address}")

        # STORE: Save this socket so we can send messages through it later.
        with self.lock:
            self.active_connections[peer_address] = peer_socket
            print(f"[*] Stored outgoing connection to {peer_address}. Active: {list(self.activ

        # LISTEN: Start a thread to listen for replies from this peer.
        conn_thread = threading.Thread(
            target=self.handle_connection,
            args=(peer_socket, peer_address)
        )
        conn_thread.daemon = True
        conn_thread.start()

        return True
    except Exception as e:
        print(f"[!] Error connecting to peer {peer_address}: {e}")
        return False
```

```python
def send_message(self, message, peer_address):
    """Send a message to a specific peer."""
    # Encrypting the message before sending it
    encrypted_content = self.encrypt_message(message)

    # Construct the message payload
    message_data = {
        "sender": self.username,
        "timestamp": time.time(),
        "content": encrypted_content
    }
    # Serialize to JSON -> Encode to Bytes
    message_json = json.dumps(message_data).encode('utf-8')

    with self.lock:
        # Look up the socket for this specific peer
        if peer_address in self.active_connections:
            try:
                print(f"[*] Sending message to {peer_address}: {message}")
                # SEND: Push bytes through the TCP pipe
                self.active_connections[peer_address].send(message_json)

                logging.info(f"[CHAT SENT] To {peer_address}: {message}")

                return True
            except Exception as e:
                print(f"[!] Error sending message to {peer_address}: {e}")
                # If sending fails, assume the connection is dead and remove it.
                del self.active_connections[peer_address]
                return False
        else:
            print(f"[!] No active connection to {peer_address}. Cannot send.")
            print(f"[*] Current active connections: {list(self.active_connections.keys())}")
            return False
```

Def send_message(): Encrypts outgoing chat messages and sends them over TCP to a specific peer.

This class implements the full TCP chat logic: it listens on a port, accepts incoming connections, spawns handler threads per peer, encrypts and decrypts messages, and sends messages to specific peers over reliable TCP sockets.

8.9 Full UDP File Transfer Core Implementation

From network.py UDPFileTransfer (main UDP logic: listener, receiver, sender):

```
class UDPFileTransfer:
    def __init__(self, username, udp_port):
        self.username = username
        self.udp_port = udp_port

        # Create a UDP socket (SOCK_DGRAM = Datagram/UDP)
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.socket.bind(('127.0.0.1', udp_port))

        self.listening = False
        self.listener_thread = None
        self.file_callback = None
        self.lock = threading.Lock()
        self.chunk_size = 4096

        # Download directory setup
        self.download_dir = "downloads"
        if not os.path.exists(self.download_dir):
            os.makedirs(self.download_dir)
            print(f"[*] Created directory: {self.download_dir}/")

    def start_listening(self):
        self.listening = True
        # Start the background thread that waits for incoming file packets
        self.listener_thread = threading.Thread(target=self.listen_for_files)
        self.listener_thread.daemon = True
        self.listener_thread.start()
```

Def start_listening(): Creates a dedicated thread to listen for incoming UDP packets used for file transfer.

```python
def listen_for_files(self):
    while self.listening:
        try:
            # recvfrom: Waits for a UDP packet.
            # We read chunk_size + 1024 to account for the header padding.
            data, addr = self.socket.recvfrom(self.chunk_size + 1024)

            try:
                # PARSING LOGIC:
                # First 1024 bytes = JSON header padded with nulls.
                header_raw = data[:1024].split(b'\0', 1)[0].decode('utf-8', errors='ignore')
                header = json.loads(header_raw)

                # CASE 1: New File Incoming
                if header.get("type") == "file_header":
                    filename = header.get("filename")
                    filesize = header.get("filesize")
                    chunks = header.get("chunks")

                    print(f"[*] File header received from {addr}: "
                          f"filename={filename}, size={filesize}, chunks={chunks}")

                    # Notify GUI: "Transfer Started"
                    with self.lock:
                        if self.file_callback:
                            self.file_callback("start", filename, filesize, 0, addr)

                    # Prepare file paths
                    temp_filepath = os.path.join(self.download_dir, f"temp_{filename}")
                    final_filepath = os.path.join(self.download_dir, filename)
                    print(f"[*] Will save incoming file to: {os.path.abspath(final_filepath)}"
```

Def listen_for_files : Receives UDP packets, parses the 1024-byte JSON header, and reconstructs incoming files chunk-by-chunk.

```
                # Loop to receive the exact number of chunks expected
                with open(temp_filepath, 'wb') as f:
                    while received_chunks < chunks:
                        data, _ = self.socket.recvfrom(self.chunk_size + 1024)
                        try:
                            # Parse packet header again
                            chunk_header_raw = data[:1024].split(b'\0', 1)[0].decode('utf-
                            chunk_header = json.loads(chunk_header_raw)

                            if chunk_header.get("type") == "file_chunk":
                                # Extract the raw binary data (everything after the first
                                chunk_data = data[1024:]
                                f.write(chunk_data)
                                received_chunks += 1

                                # Notify GUI of progress
                                with self.lock:
                                    if self.file_callback:
                                        progress = (received_chunks / chunks) * 100
                                        self.file_callback("progress", filename, filesize,
                        except json.JSONDecodeError:
                            # Invalid chunk header, ignore
                            continue

                # Transfer Done: Rename temp file to real filename
                os.rename(temp_filepath, final_filepath)

                # Notify GUI: "Transfer Complete"
                with self.lock:
                    if self.file_callback:
                        self.file_callback("complete", filename, filesize, 100, addr)

                print(f"[+] File '{filename}' saved to '{os.path.abspath(final_filepath)}'

            except json.JSONDecodeError:
                # Not a valid header packet
                continue

        except Exception as e:
            if self.listening:
                print(f"[!] Error receiving file: {e}")
```

```python
def send_file(self, filename, peer_ip, peer_port):
    """Starts the file sending process in a new thread."""
    try:
        if not os.path.exists(filename):
            return False, "File not found"

        # Use a thread so large files don't freeze the GUI
        send_thread = threading.Thread(
            target=self._send_file_worker,
            args=(filename, peer_ip, peer_port)
        )
        send_thread.daemon = True
        send_thread.start()

        return True, "File transfer started in background."
    except Exception as e:
        return False, f"Error starting file transfer: {e}"
```

```python
def _send_file_worker(self, filename, peer_ip, peer_port):
    """Worker thread that handles the actual file sending."""
    try:
        base_filename = os.path.basename(filename)
        filesize = os.path.getsize(filename)
        # Calculate how many pieces we need to send
        chunks = (filesize + self.chunk_size - 1) // self.chunk_size  # Like a ceil functio

        # STEP 1: Send the File Header (Meta-data)
        header = {
            "type": "file_header",
            "filename": base_filename,
            "filesize": filesize,
            "chunks": chunks,
            "sender": self.username
        }
        header_data = json.dumps(header).encode('utf-8')
        # Pad the header to exactly 1024 bytes using null characters (\0)
        header_data = header_data.ljust(1024, b'\0')
        self.socket.sendto(header_data, (peer_ip, peer_port))

        # STEP 2: Read file and send chunks
        with open(filename, 'rb') as f:
            chunk_index = 0
            while True:
                # Read 4KB of binary data
                chunk_data = f.read(self.chunk_size)
                if not chunk_data:
                    break

                # Create chunk header
                chunk_header = {
                    "type": "file_chunk",
                    "index": chunk_index,
                    "sender": self.username
                }
                header_bytes = json.dumps(chunk_header).encode('utf-8')
                header_bytes = header_bytes.ljust(1024, b'\0')

                # Combine Header + Binary Data
                packet = header_bytes + chunk_data

                # Blast it via UDP
                self.socket.sendto(packet, (peer_ip, peer_port))
                chunk_index += 1

    except Exception as e:
        print(f"[!] Error in file sending thread: {e}")
```

Def send_file_worker(): Sends file metadata first (header), then transmits the file in 4KB binary chunks over UDP without blocking the GUI.

This class implements the full UDP file transfer logic: it listens for incoming datagrams, parses custom headers, reconstructs files from fixed-size chunks, reports progress to the GUI, and sends files in the background using UDP packets.

8.10 Central Server Connector (Registry Client)

From network.py ServerConnector (handles registration and peer discovery with the central server):

```
class ServerConnector:
    def __init__(self, server_host, server_port):
        self.server_host = server_host
        self.server_port = server_port
        # Create TCP socket for registry communication
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    def connect(self):
        try:
            # Persistent connection to the Central Server
            self.socket.connect((self.server_host, self.server_port))
            return True
        except Exception as e:
            print(f"[!] Error connecting to server: {e}")
            return False

    def disconnect(self):
        try:
            self.socket.close()
        except:
            pass
```

Def __init__: Creates the TCP socket used for all communication with the central registry server.

Def connect(): Establishes a persistent TCP connection with the central server for registration and peer queries.

Def disconnect() : Safely closes the TCP connection when the client exits.

```python
def register(self, username, tcp_port, udp_port):
    try:
        # Construct JSON command to tell server who we are
        request = {
            "command": "register",
            "username": username,
            "tcp_port": tcp_port,
            "udp_port": udp_port
        }
        # Send the request to the server
        self.socket.send(json.dumps(request).encode('utf-8'))
        # Wait for confirmation response
        response = json.loads(self.socket.recv(1024).decode('utf-8'))
        return response.get("status") == "success"
    except Exception as e:
        print(f"[!] Error registering with server: {e}")
        return False

def get_peers(self):
    try:
        # Ask server for list of active users
        request = {"command": "get_peers"}
        self.socket.send(json.dumps(request).encode('utf-8'))
        # Receive larger buffer (4096) as peer list might be long
        response = json.loads(self.socket.recv(4096).decode('utf-8'))
        if response.get("status") == "success":
            return response.get("peers", {})
        return {}
    except Exception as e:
        print(f"[!] Error getting peers from server: {e}")
        return {}
```

Def Register : Sends the client's username and TCP/UDP ports to the server so other peers can discover it.

Def get_peers : Requests the list of all currently registered peers, enabling real-time updates in the GUI.

```
def unregister(self, username):
    try:
        # Tell server to remove us from the list
        request = {"command": "unregister", "username": username}
        self.socket.send(json.dumps(request).encode('utf-8'))
        response = json.loads(self.socket.recv(1024).decode('utf-8'))
        return response.get("status") == "success"
    except Exception as e:
        print(f"[!] Error unregistering from server: {e}")
        return False
```

Def unregister(): Notifies the server to remove the client from the active peer list when closing the application.

9. Testing and Validation

Testing covered:

• Server responsiveness

Registration and peer lookup validated using multiple simultaneous clients.

• TCP message integrity

Messages were received in order with no data loss.

• UDP file transfer accuracy

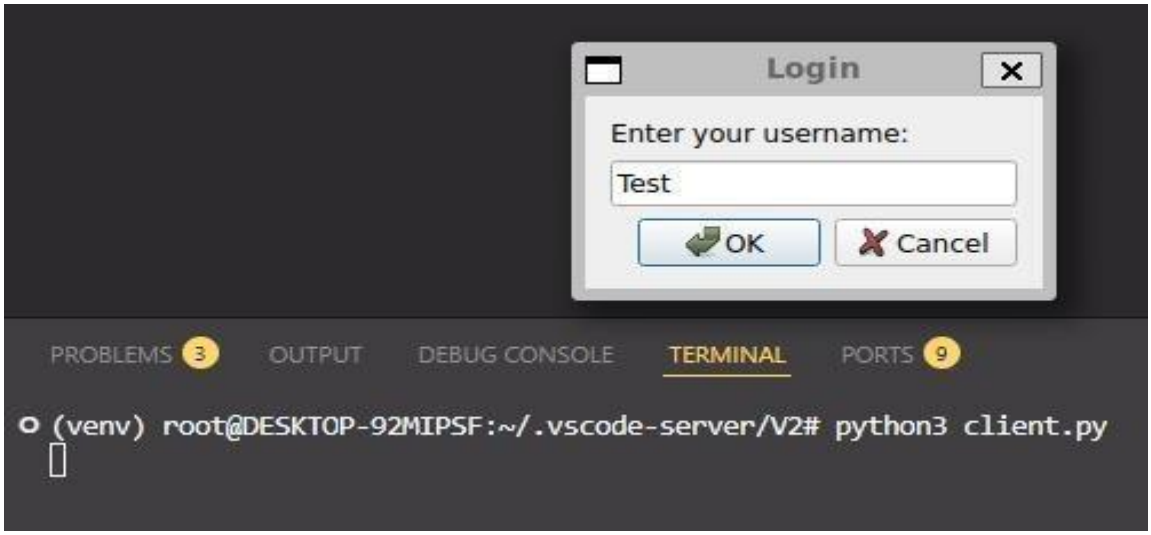
Files up to several megabytes were transferred and reconstructed correctly.

• GUI responsiveness

Thanks to multithreading, the GUI remained responsive even during large file transfers.

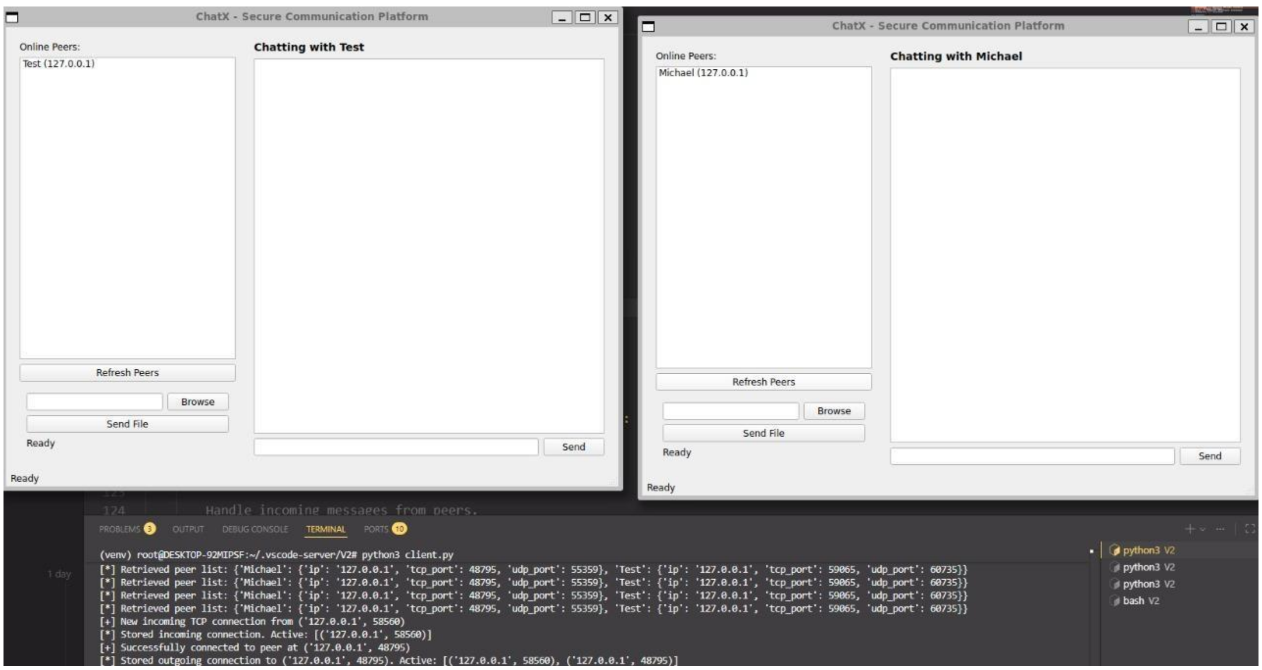
9.1 Multi-Client Initialization Test

To verify that ChatX supports multiple simultaneous clients, the application was launched from two separate terminals. Each instance displays an independent username prompt, ensuring that clients can register with unique identifiers and operate concurrently. The screenshot below shows the second client initializing using the username “Test”, confirming that the multi-client login mechanism functions correctly.



9.2 Peer-to-Peer Session Establishment Test

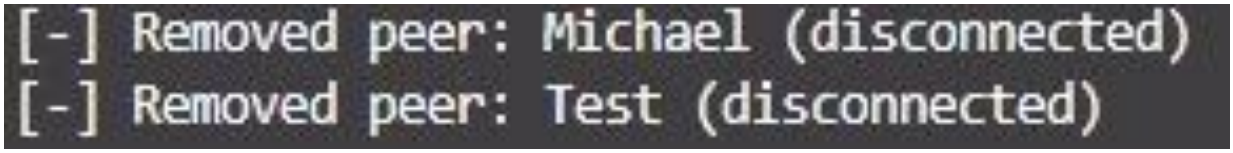
To validate the correctness of the peer-to-peer communication architecture, two ChatX clients were launched simultaneously on the same machine. Each client retrieved the updated peer list from the server and displayed the other online user. After selecting a peer, both clients successfully established a direct TCP connection and entered a chat session. The interface labels updated automatically to reflect the active conversation, confirming that the connection handling, peer lookup, and GUI updates were functioning as intended. The screenshot below shows both clients running side by side, each connected to the other.



9.3 Peer Disconnection and Cleanup

An essential part of maintaining a stable peer-to-peer system is ensuring that peers are correctly removed from the server registry when they disconnect. After both ChatX clients were closed, the server immediately detected the termination of each connection and removed the

corresponding entries from the active peer list. This prevents stale peers from appearing online and ensures that future clients always receive an accurate list of available users. The screenshot below shows the server successfully unregistering both peers.



10. Challenges Encountered

The main challenges included:

1. **Thread safe GUI updates**
   Resolved using Qt signals and slots (via the SignalHandler class).
2. **Concurrent TCP connections**
   Required careful synchronization using locks.
3. **Parsing UDP packets**
   Ensuring consistent header separation from byte payloads.
4. **Managing disconnections**
   Ensuring clean resource cleanup during shutdown.

These challenges enhanced our understanding of network robustness and system reliability.

11. Conclusion

ChatX successfully demonstrates a practical implementation of real-time communication using Python's socket programming capabilities. The combination of TCP for messaging and UDP for file transmission allowed us to explore the strengths of each protocol in a controlled environment. Through careful multithreading, structured communication protocols, and a responsive graphical interface, the project meets all requirements of the course and serves as a meaningful learning experience.

The system's modular architecture also provides a strong foundation for future enhancements such as encryption, NAT traversal, and group messaging.

12. References

- ChatX Codebase
  - client.py
  - gui.py
  - network.py
  - server.py
- Some help of AI was used