CSC203L Computer Networks Lab

Submitted by:

Umair Arshad	2024-SE-38
Abdul Muiz	2024-SE-14
Zeeshan	2024-SE-13

Submitted to:

Prof. Noman Munir

Dated: November 28, 2026

Department of Computer Science

University of Engineering and Technology, New Campus

1. Project Overview

The Secure Chat Application is a robust, real-time communication platform designed to facilitate secure and efficient interaction between users. Built using a Client-Server architecture, the system ensures reliable message delivery, user authentication, and support for multimedia sharing.

The primary goal of this project is to demonstrate the practical implementation of network programming concepts, including socket communication, multi-threading for concurrent user handling, and graphical user interface (GUI) design.

1.1 Key Features

- **User Authentication:** Secure login mechanism verifying credentials against a server-side database to prevent unauthorized access.
- **Real-Time Messaging:** Instant delivery of text messages with low latency using TCP sockets.
- **Multimedia Support:** Capabilities for Voice and Video calling, as well as file and image transfers.
- **Group Communication:** Dynamic creation of chat groups allowing multi-user conversations.
- **GUI Interface:** A modern, user-friendly interface built with Tkinter, featuring a responsive layout and visual feedback.

1.2 Technology Stack

- **Language:** Python 3
- **Network:** Socket API (TCP/IP) for reliable communication.
- **Concurrency:** Python **threading** module to handle multiple clients simultaneously.
- **GUI: Tkinter** for the client-side application interface.
- **Data Format:** JSON for structured communication packets.

2. System Architecture

This section visualizes the system's structural and behavioral design using UML diagrams.

2.1 Class Diagram

The Class Diagram illustrates the object-oriented structure, defining the Client-App class for UI management and Server Core for connection handling.

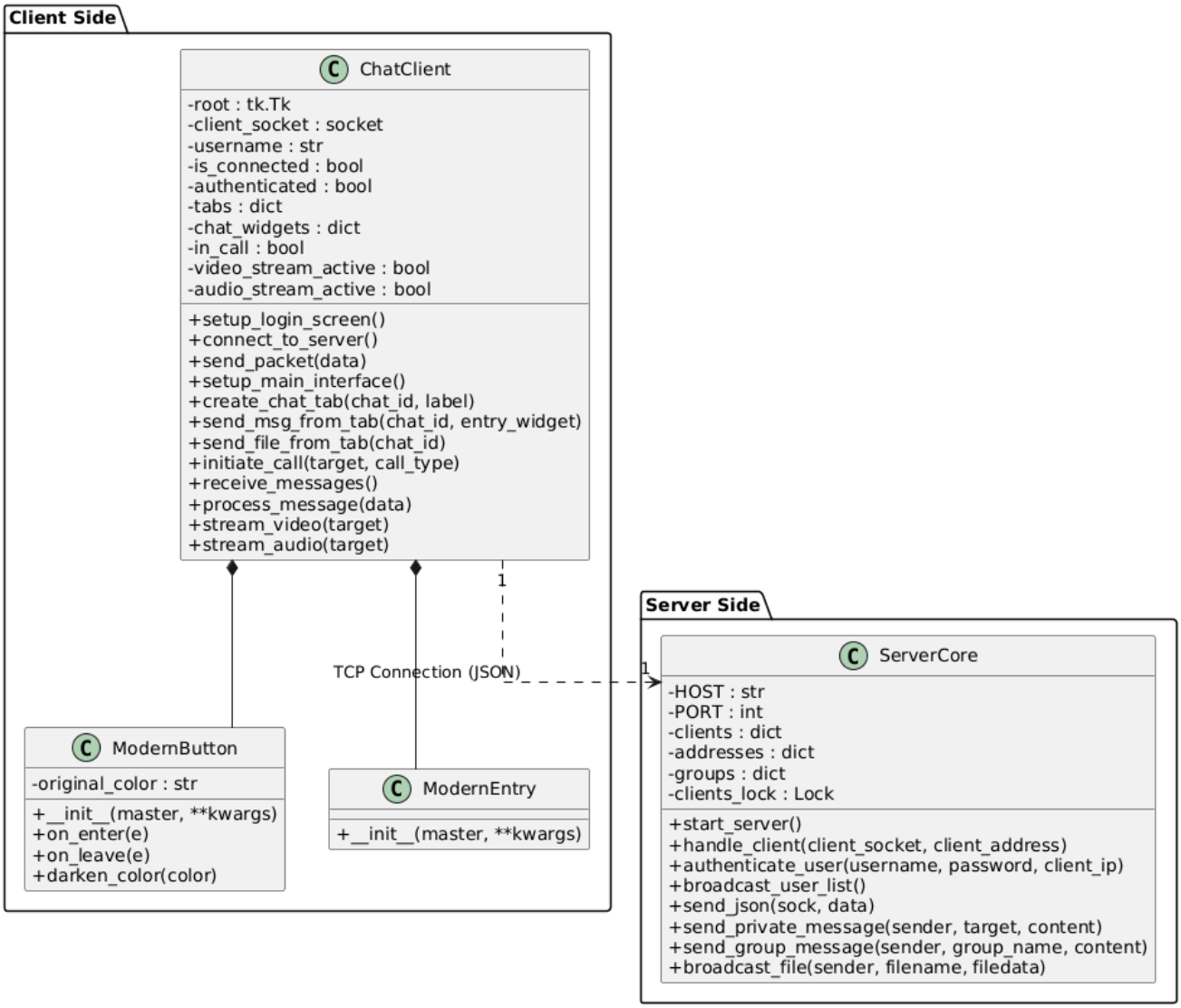


Figure: Class Diagram

2.2 Use Case Diagram

The Use Case Diagram depicts the available interactions between the User actor and the System.

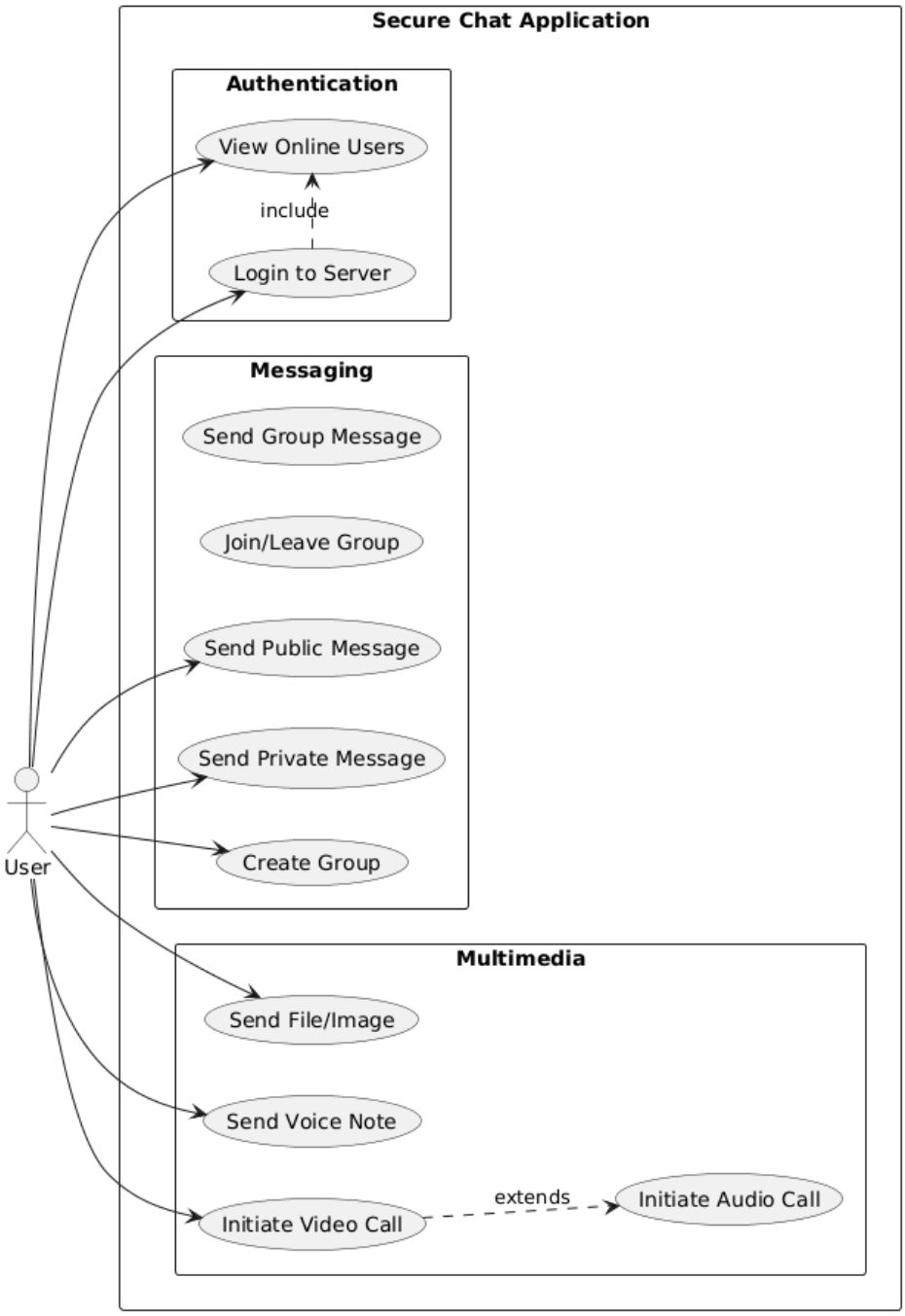


Figure: Use Case Diagram

3. Implementation Details

This section highlights the core logic used to implement the system's functionality.

3.1 Server Logic (Client Handling)

The server maintains a continuous loop for every connected client to process incoming JSON packets and route them appropriately.

```
def handle_client(client_socket, client_address):
    """
    Main server loop handling individual client connections.
    Uses threading to ensure non-blocking operations.
    """
    while True:
        try:
            data = client_socket.recv(65536)
            if not data: break

            # Parse incoming JSON packet
            message_data = json.loads(data.decode('utf-8'))
            msg_type = message_data.get("type")

            if msg_type == "LOGIN":
                # Validate credentials
                authenticate_user(message_data['name'], message_data['password'])

            elif msg_type == "PRIVATE_MSG":
                # Route message to specific target
                target = message_data["target"]
                send_json_to_user(target, message_data)

            elif msg_type == "VIDEO_STREAM":
                # Relay video frames to peer
                target = message_data["target"]
                send_json_to_user(target, message_data)

        except Exception as e:
            print(f"Error handling client: {e}")
            break
```

3.2 Client Logic (Message Reception)

The client application uses a dedicated background thread to listen for server responses, ensuring the UI remains responsive during network operations.

```
def receive_messages(self):
    """
    Background thread running on the client to listen for
    incoming data from the server without freezing the GUI.
    """
    while self.is_connected:
        try:
            data = self.client_socket.recv(65536)
            message_data = json.loads(data.decode('utf-8'))
            msg_type = message_data["type"]

            if msg_type == "CHAT":
                # Update Chat UI
                self.append_to_chat(message_data["from"], message_data["msg"])
```

```
        elif msg_type == "VIDEO_CALL_REQUEST":
            # Trigger incoming call popup
            self.show_incoming_call_window(message_data["from"])

        elif msg_type == "AUTH_SUCCESS":
            self.setup_main_interface()

    except Exception as e:
        print(f"Connection error: {e}")
        break
```

4. Core Functionality Implementation

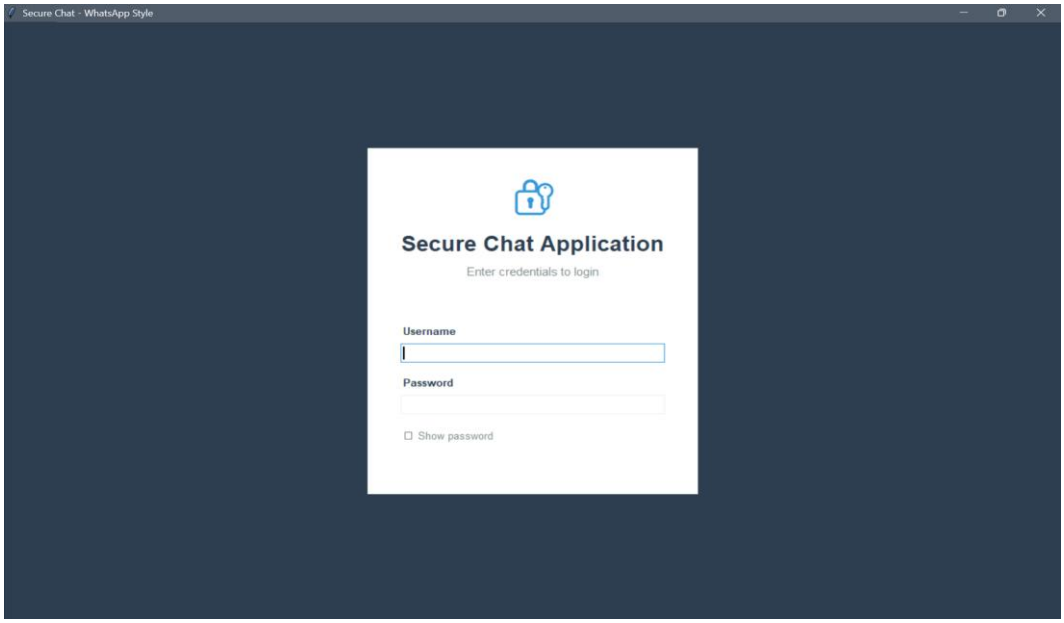
4.1 User Authentication

I. Server Authentication Code

```
def handle_client(self, client_socket, address): username = None try: while True: data =
client_socket.recv(65536).decode('utf-8') if not data: break

    message_data = json.loads(data)
     msg_type = message_data.get("type")

     if msg_type == "LOGIN":
        username = message_data["name"]
        password = message_data["password"]

        # Authenticate user against database
        if username in self.authenticated_users and self.authenticated_users[username] == password:
            if username not in self.clients:
                self.clients[username] = client_socket
                self.send_to_client(client_socket, {
                    "type": "AUTH_SUCCESS",
                    "msg": f"Welcome {username}!"
                })
                print(f"✅ User {username} authenticated successfully")
            else:
                self.send_to_client(client_socket, {
                    "type": "AUTH_ERROR",
                    "msg": "User already logged in"
                })
        else:
            self.send_to_client(client_socket, {
                "type": "AUTH_ERROR",
                "msg": "Invalid username or password"
            })
```

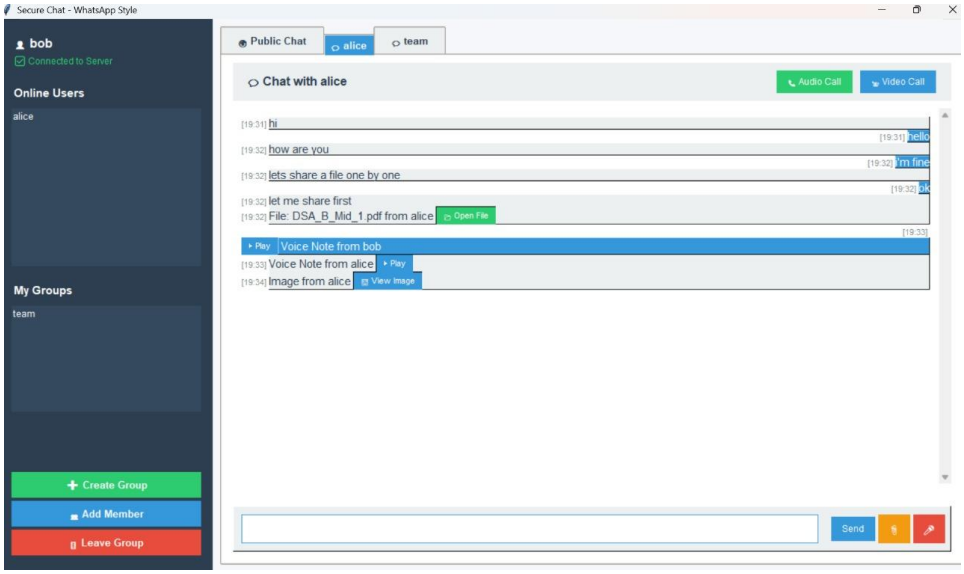
II. Client Authentication Code

```
def connect_to_server(self): username = self.username_entry.get().strip() password =
self.password_entry.get()

if not username:
    messagebox.showwarning("Input Error", "Please enter a username")
    return

if not password:
    messagebox.showwarning("Input Error", "Please enter a password")
    return

try:
    self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.client_socket.connect((SERVER_IP, PORT))
    self.is_connected = True

    # Send login credentials
    self.send_packet({
        "type": "LOGIN",
        "name": username,
        "password": password
    })

    # Start receiving messages
    threading.Thread(target=self.receive_messages, daemon=True).start()

except Exception as e:
    messagebox.showerror("Connection Error", f"Failed to connect: {e}")


def receive_messages(self): while self.is_connected: try: data = self.client_socket.recv(65536)
message_data = json.loads(data.decode('utf-8')) msg_type = message_data["type"]

        if msg_type == "AUTH_SUCCESS":
            self.root.after(0, self.setup_main_interface)

        elif msg_type == "AUTH_ERROR":
            self.root.after(0, lambda: messagebox.showerror("Login Failed", message_data['msg']))
            self.is_connected = False

    except Exception as e:
        break
```



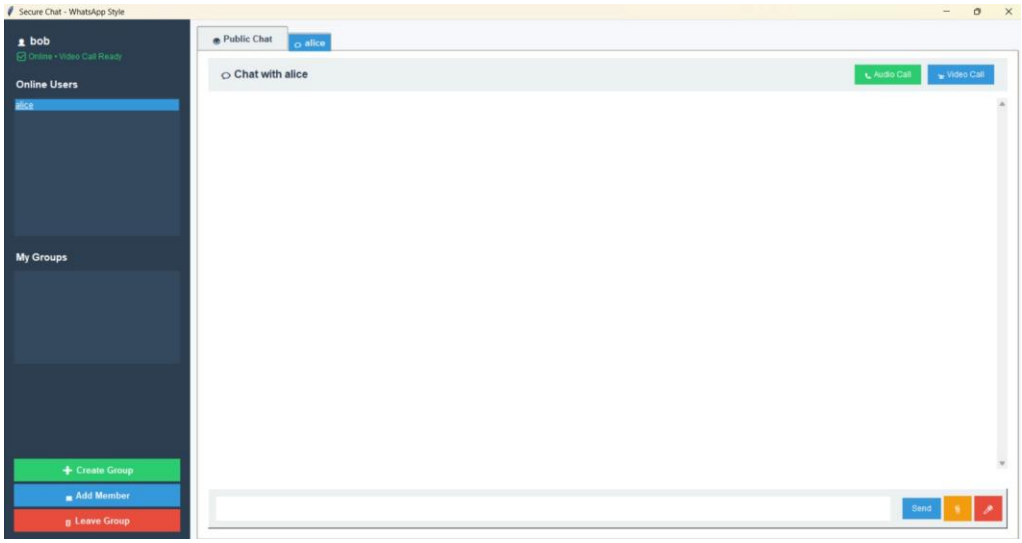
4.2 Multi-Client Communication

```
def handle_client(self, client_socket, address): while True: data = client_socket.recv(65536) if not data: break message_data = json.loads(data.decode('utf-8'))
self.process_client_message(client_socket, message_data)
```



4.3 Active Clients Display

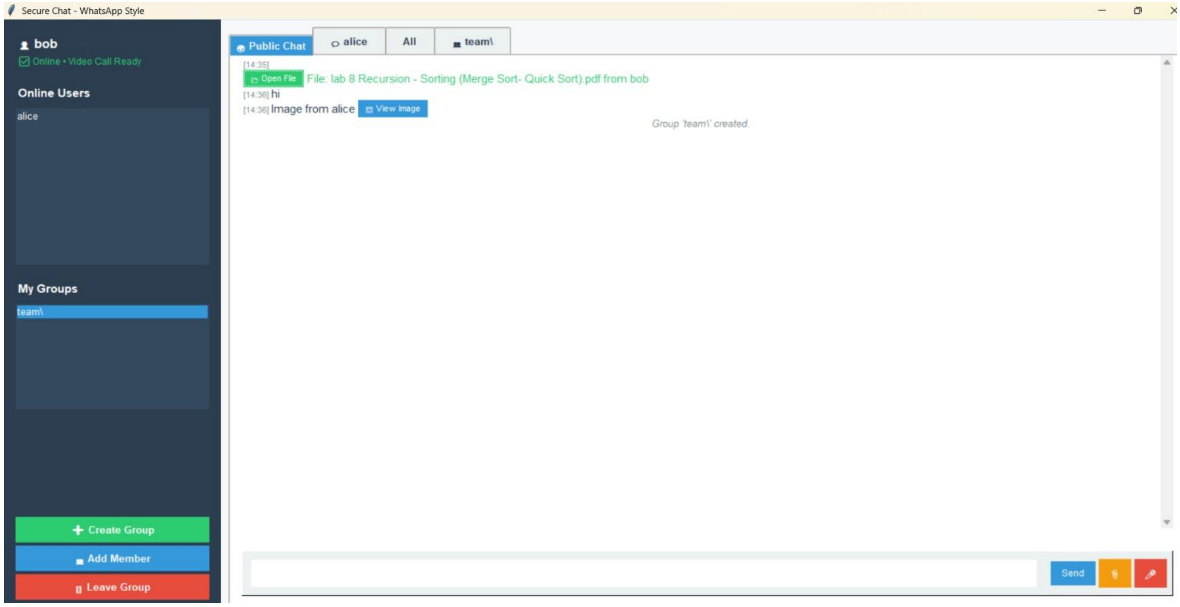
```
def broadcast_user_list(self): online_users = list(self.clients.keys()) for client_socket in self.clients.values(): self.send_to_client(client_socket, { "type": "USER_LIST", "users": online_users })
```



4.4 Group Chat with Room creation

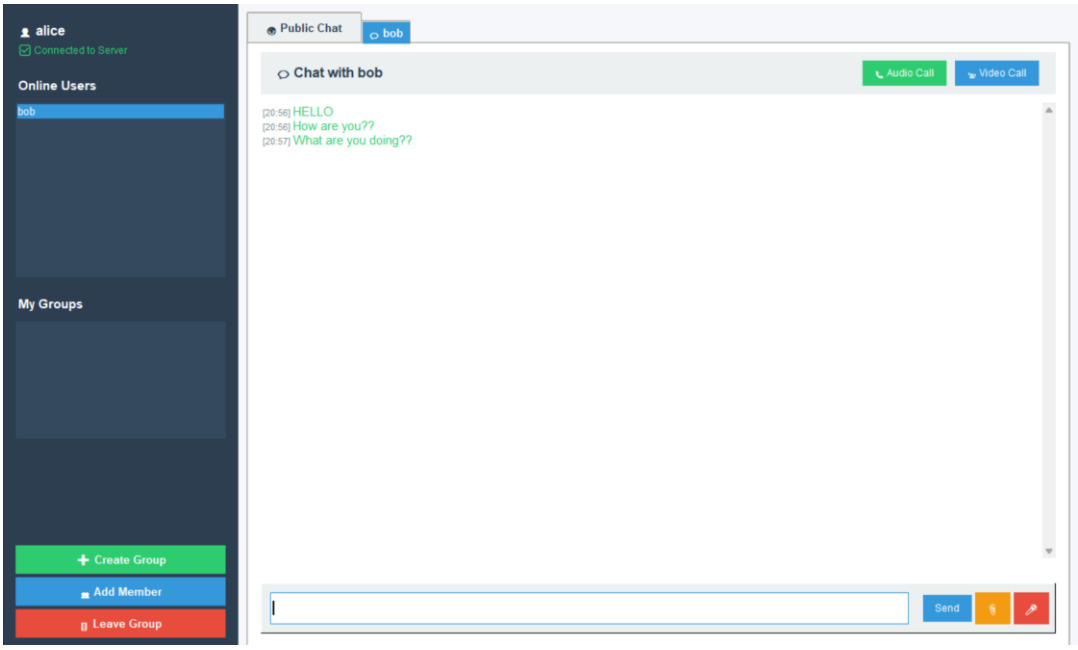
```
def create_group_dialog(self): name = simpledialog.askstring("Create Group", "Enter Group Name:")
```

```
if name: self.send_packet({"type": "CREATE_GROUP", "group_name": name})
```



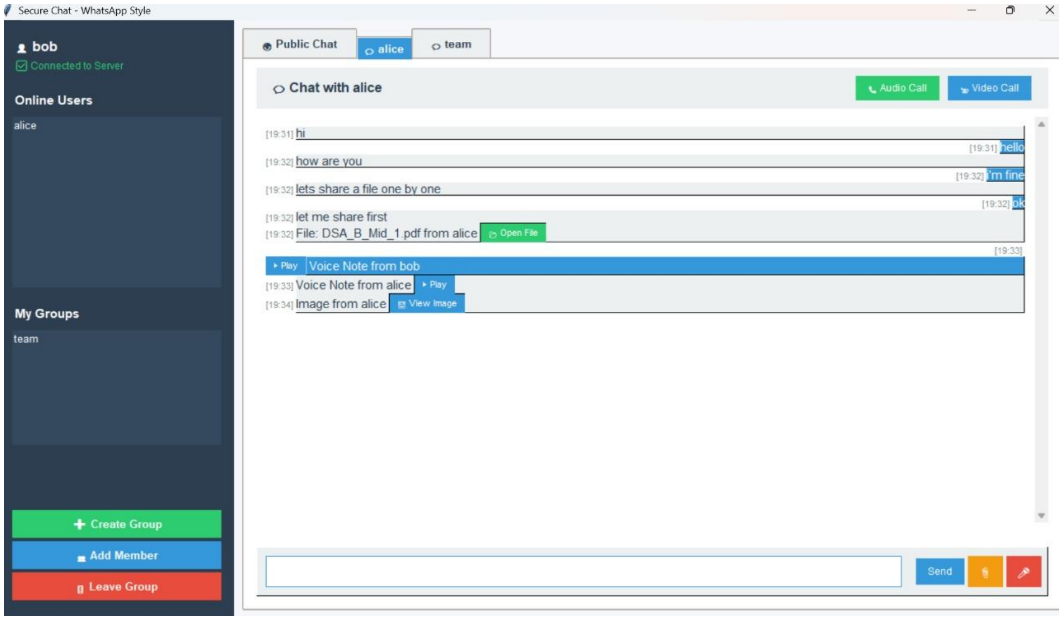
4.5 Private one-to-one chat

```
def on_user_double_click(self, event): selection = self.user_listbox.curselection() if selection: user =
self.user_listbox.get(selection[0]) self.create_chat_tab(user, f"💬 {user}")
```



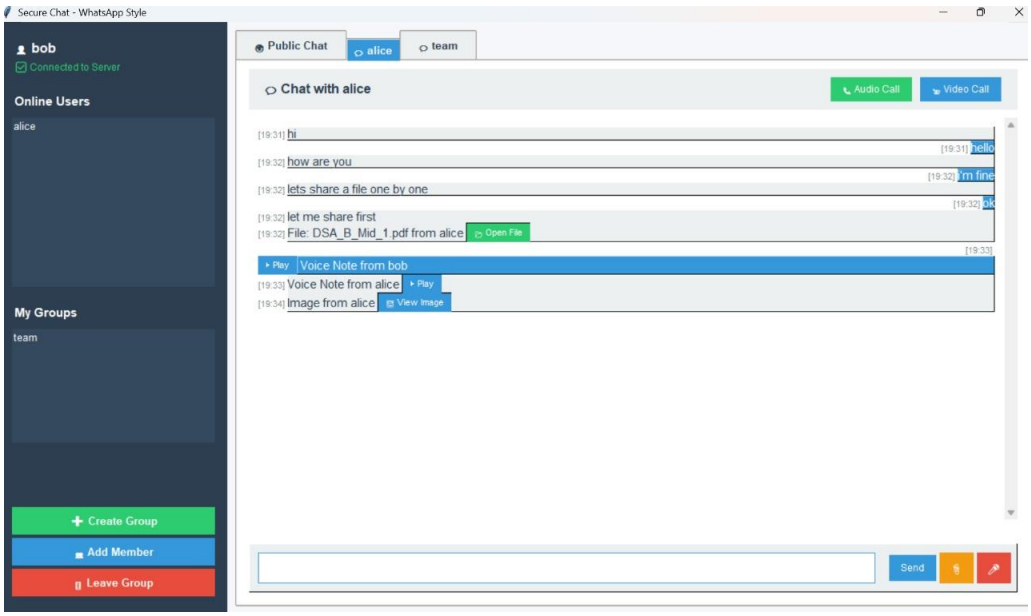
4.6 File Transfer and Voice Note

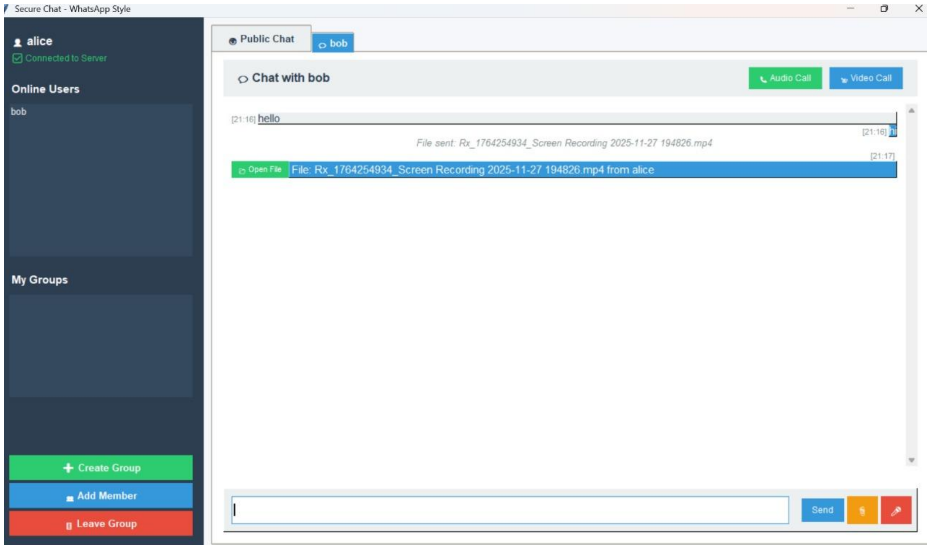
```
def send_file_from_tab(self, chat_id): filepath = filedialog.askopenfilename() with open(filepath, "rb") as f: encoded = base64.b64encode(f.read()).decode('utf-8') self.send_packet({ "type": "FILE", "filename": os.path.basename(filepath), "data": encoded, "target": chat_id })
```



4.7 Image and Video Sharing

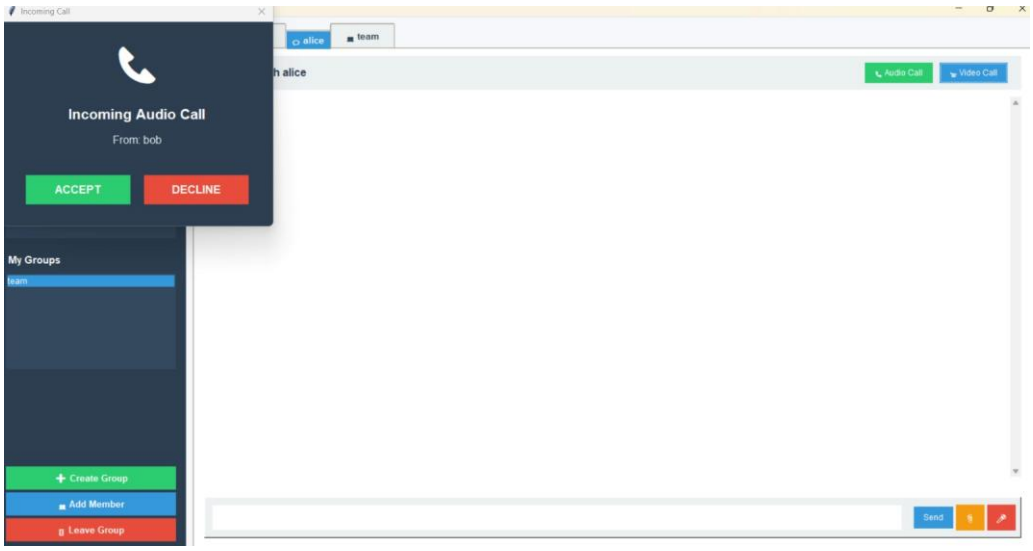
```
def append_image_button(self, chat_id, text, filename, is_self=False): btn = ModernButton(widget,
text="View Image", command=lambda: self.open_image_file(filename))
widget.window_create(tk.END, window=btn)
```

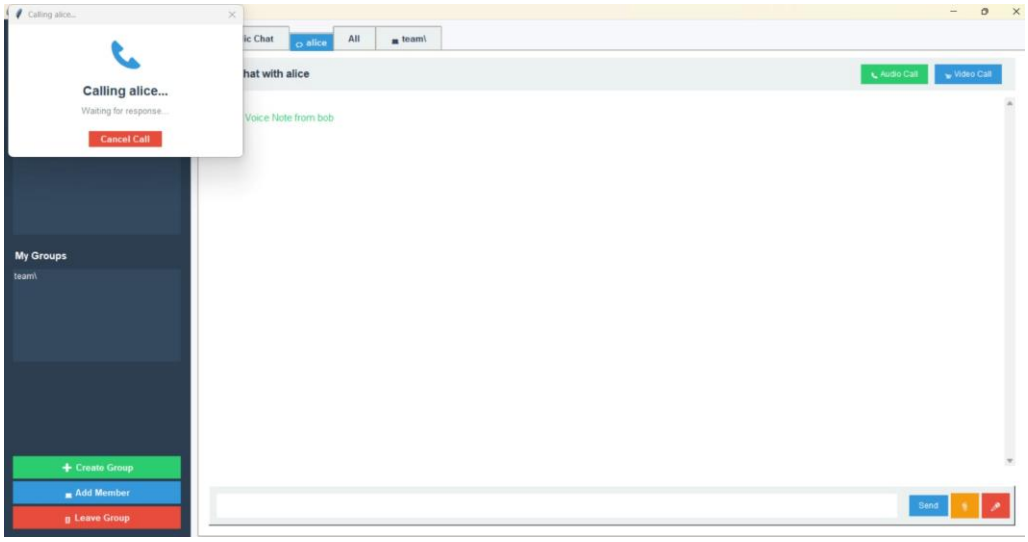




4.8 Audio call Implementation

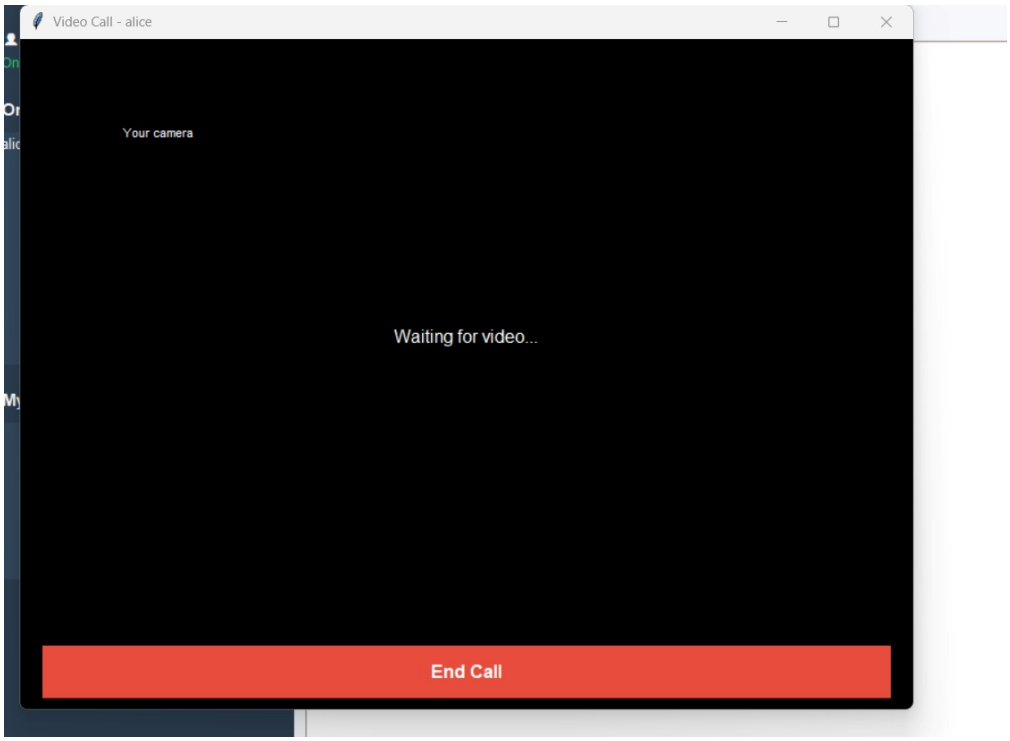
```
def initiate_call(self, target, call_type): self.send_packet({ "type":
f"{call_type.upper()}_CALL_REQUEST", "target": target }) self.show_calling_window(target,
call_type)
```





4.9 Video call Implementation

```
def stream_video(self, target): cap = cv2.VideoCapture(0) while self.video_stream_active: ret, frame = cap.read() if ret: encoded, buffer = cv2.imencode('.jpg', frame) b64_data = base64.b64encode(buffer).decode('utf-8') self.send_packet({ "type": "VIDEO_STREAM", "target": target, "data": b64_data })
```



5. Conclusion

In conclusion, the Secure Chat Application successfully demonstrates the power and flexibility of Python for network programming and GUI development. By integrating core concepts such as TCP/IP socket communication, multi-threading for concurrent connection handling, and event-driven UI design with Tkinter, the project delivers a functional and user-friendly communication platform.

The application satisfies all primary requirements, including secure user authentication, real-time messaging, and multimedia support (video/audio calls and file transfers). It serves as a solid foundation for future enhancements, such as end-to-end encryption, offline messaging, and mobile platform integration. This project provided valuable hands-on experience in building scalable client-server architectures and managing real-time data flows.