**Christian B. Seniel  
Macadatar, Raafat  
CS3A  
  
ChatTide**

**1. Project Overview & Objectives**

**Project Name:** Simple One-to-One Chat Application

**Objective:**  
Develop a client-server chat application that enables real-time text communication between users over a TCP network. The project will allow users to host a server or connect as clients, facilitating seamless communication within a local or remote network. The system is designed to be user-friendly with a graphical interface.

**2. System Architecture**

**Components**

1. **Client:**
   * Initiates a connection to the server using a specified IP address.
   * Provides a graphical interface for users to type and send messages.
   * Displays messages received from the server in a chat window.
2. **Server:**
   * Waits for incoming client connections.
   * Facilitates bidirectional communication with the connected client.
   * Displays messages sent by the client in the chat window.
3. **Communication Layer (Protocol):**
   * Based on TCP sockets for reliable and ordered data transfer.
   * Encodes text messages as UTF-8 before transmission.
   * Handles exceptions to manage network errors.
4. **Network Layer:**
   * Manages socket connections using socket library.
   * Ensures data transmission through TCP/IP.

**3. Code Structure**

Simple One-to-One Chat Application /

├── main.py # Entry point for the application

├── client.py # Main client application

├── client\_connection.py # Handles the client's TCP connection logic

├── server.py # Main server application

├── server\_connection.py # Handles the server's TCP connection logic

├── utils.py # Shared utility functions

**4. Client-Side Implementation**

The client connects to the server, sends and receives messages, and closes the connection when done. It is responsible for handling user input and displaying messages in its UI.

* **Connect to Server:** Establish a TCP connection with the server.
* **Send Messages:** Send text messages to the server.
* **Receive Messages:** Continuously listen for and display incoming messages from the server.

**Example Code Snippet (client.py):**

import tkinter as tk

from tkinter import scrolledtext

from client\_connection import ClientConnection

from utils import handle\_error

def start\_client():

    def connect\_to\_server():

        server\_ip = ip\_entry.get().strip()

        if not server\_ip:

            handle\_error("Please enter a valid IP address.")

            return

        app.connection.connect(server\_ip)

        ip\_window.destroy()

    class ClientApp:

        def \_\_init\_\_(self, master):

            self.master = master

            self.master.title("Client - Chat Application")

            self.connection = ClientConnection(self.update\_chat)

            self.chat\_area = scrolledtext.ScrolledText(master, state='disabled', width=50, height=20)

            self.chat\_area.pack(pady=10)

            self.message\_entry = tk.Entry(master, width=40)

            self.message\_entry.pack(side=tk.LEFT, padx=(0, 10))

            self.message\_entry.bind("<Return>", self.send\_message)

            self.send\_button = tk.Button(master, text="Send", command=self.send\_message)

            self.send\_button.pack(side=tk.LEFT)

        def update\_chat(self, message):

            self.chat\_area.config(state='normal')

            self.chat\_area.insert(tk.END, message + "\n")

            self.chat\_area.config(state='disabled')

        def send\_message(self, event=None):

            message = self.message\_entry.get().strip()

            if message:

                self.update\_chat(f"You: {message}")

                self.message\_entry.delete(0, tk.END)

                self.connection.send\_message(message)

    root = tk.Tk()

    app = ClientApp(root)

    ip\_window = tk.Toplevel(root)

    tk.Label(ip\_window, text="Server IP Address:").pack(pady=5)

    ip\_entry = tk.Entry(ip\_window, width=30)

    ip\_entry.pack(pady=5)

    tk.Button(ip\_window, text="Connect", command=connect\_to\_server).pack(pady=10)

    root.mainloop()

**Example Code Snippet (client\_connection.py):**

import socket

import threading

class ClientConnection:

    def \_\_init\_\_(self, update\_chat\_callback):

        self.socket = None

        self.update\_chat\_callback = update\_chat\_callback

    def connect(self, server\_ip, port=54321):

        self.socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

        self.socket.connect((server\_ip, port))

        threading.Thread(target=self.receive\_messages, daemon=True).start()

    def send\_message(self, message):

        try:

            self.socket.send(message.encode('utf-8'))

        except Exception as e:

            self.update\_chat\_callback(f"Error sending message: {e}")

    def receive\_messages(self):

        while True:

            try:

                message = self.socket.recv(1024).decode('utf-8')

                if message:

                    self.update\_chat\_callback(f"Server: {message}")

            except Exception as e:

                self.update\_chat\_callback(f"Connection error: {e}")

                break

**5. Server-Side Implementation**

The server listens for incoming client connections, manages communication with connected clients (receiving and sending messages), and properly closes connections when the client disconnects.

* **Start Server**: Initialize the server and start listening for incoming client connections.
* **Accept Client Connection**: Accept a connection request from a client.
* **Send Messages**: Send text messages from the server to the connected client.
* **Receive Messages**: Continuously listen for and handle incoming messages from the client.
* **Close Connection**: Gracefully close the connection with the client.

**Example Code Snippet (server.py):**

import tkinter as tk

from tkinter import scrolledtext

from server\_connection import ServerConnection

from utils import handle\_error

def start\_server():

    class ServerApp:

        def \_\_init\_\_(self, master):

            self.master = master

            self.master.title("Server - Chat Application")

            self.connection = ServerConnection(self.update\_chat)

            self.chat\_area = scrolledtext.ScrolledText(master, state='disabled', width=50, height=20)

            self.chat\_area.pack(pady=10)

            self.message\_entry = tk.Entry(master, width=40)

            self.message\_entry.pack(side=tk.LEFT, padx=(0, 10))

            self.message\_entry.bind("<Return>", self.send\_message)

            self.send\_button = tk.Button(master, text="Send", command=self.send\_message)

            self.send\_button.pack(side=tk.LEFT)

            self.connection.start\_server()

        def update\_chat(self, message):

            self.chat\_area.config(state='normal')

            self.chat\_area.insert(tk.END, message + "\n")

            self.chat\_area.config(state='disabled')

        def send\_message(self, event=None):

            message = self.message\_entry.get().strip()

            if message:

                self.update\_chat(f"You: {message}")

                self.message\_entry.delete(0, tk.END)

                self.connection.send\_message(message)

    root = tk.Tk()

    app = ServerApp(root)

    root.mainloop()

**Example Code Snippet (server\_connection.py):**

import socket

import threading

class ServerConnection:

    def \_\_init\_\_(self, update\_chat\_callback):

        self.socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

        self.connection = None

        self.update\_chat\_callback = update\_chat\_callback

    def start\_server(self, host="0.0.0.0", port=54321):

        self.socket.bind((host, port))

        self.socket.listen(1)

        self.update\_chat\_callback("Server started. Waiting for a connection...")

        threading.Thread(target=self.accept\_client, daemon=True).start()

    def accept\_client(self):

        self.connection, addr = self.socket.accept()

        self.update\_chat\_callback(f"Client connected from {addr}")

        threading.Thread(target=self.receive\_messages, daemon=True).start()

    def send\_message(self, message):

        try:

            self.connection.send(message.encode('utf-8'))

        except Exception as e:

            self.update\_chat\_callback(f"Error sending message: {e}")

    def receive\_messages(self):

        while True:

            try:

                message = self.connection.recv(1024).decode('utf-8')

                if message:

                    self.update\_chat\_callback(f"Client: {message}")

            except Exception as e:

                self.update\_chat\_callback(f"Connection error: {e}")

                break

**6. Protocol Design**

1. **Connection Setup**:
   * **Purpose**: Establish a TCP connection between the client and server.
   * **Process**:
     + **Client**: The client uses the start\_client() function to connect to the server via the server's IP address and port (54321).
     + **Server**: The server listens for incoming connections using start\_server() and accepts the first client that connects.
2. **Message Exchange**:
   * **Purpose**: Send and receive text messages between the client and server.
   * **Process**:
     + **Client to Server**: The client sends messages using the send\_message() method. The message is sent via the socket to the server.
     + **Server to Client**: The server sends messages to the client using the same socket connection when it calls send\_message() from the server-side interface.
   * **Message Format**:
     + Messages are simple text messages, encoded using UTF-8. No additional formatting (e.g., headers) is applied to the messages.
3. **Error Handling**:
   * **Purpose**: Handle errors that occur during communication (e.g., message send/receive failures).
   * **Process**:
     + **Client and Server**: Both client and server check for errors when sending/receiving messages. Errors are logged in the chat window. The errors are typically displayed as "Error sending message: <error>" or "Error receiving message: <error>".
4. **Connection Termination**:
   * **Purpose**: Close the connection when either the client or server ends the communication.
   * **Process**:
     + **Client to Server**: There is no explicit "DISCONNECT" message in the current code, but the connection is closed when the program ends, or the socket is closed via error handling.
     + **Server to Client**: The server can close the connection once the client disconnects or when the server stops running.