Lan Chat

(MCN Project)

By

Arya Chimulkar, Arya Mantravadi, Aryan Ghorpade, Bhargava Bhatkurse, Minsar Aga

1. Introduction

The client-server chat application is a project that aims to enable real-time communication between multiple clients through a server. The project consists of two main components: the client-side graphical user interface (GUI) and the server-side implementation. The GUI allows users to send and receive messages, while the server handles the connections, message distribution, and client management.

2. Objectives

The objectives of this project are as follows:
Develop a user-friendly GUI for the chat client that allows users to send and receive messages.
Implement a server that can handle multiple client connections and distribute messages between clients.
Enable real-time communication between clients by establishing a reliable socket connection.
Provide a simple command to stop the server gracefully.

3. Technologies Used

The project utilizes the following technologies:
Python: The programming language used for both the client-side GUI and the server-side implementation.
tkinter: A Python library for creating GUI applications.
socket: A Python library for network programming, used to establish socket connections.
threading: A Python module for creating and managing threads, enabling concurrent execution.

4. Client-side GUI

The client-side GUI is implemented using the tkinter library. It provides a user-friendly interface for users to send and receive messages. The GUI components include a message area to display the chat history, an input field to type messages, and a send button to send messages to the server. The GUI supports scrolling and automatic message display.

The client-side GUI code establishes a socket connection with the server, enabling the transmission of messages. It also includes methods to send messages, display them in the message area, and handle server disconnections. The GUI runs on the main thread, allowing users to interact with the application while messages are sent and received in the background on a separate thread.

5. Server-side Implementation

The server-side implementation handles the connections, message distribution, and client management. It creates a socket object, binds it to a specific address and port, and listens for incoming connections. Once a client connects, a new thread is created to handle that client. The server maintains a list of connected clients to broadcast messages to all connected clients except the sender.

The server-side code utilizes the socket library to handle incoming connections and messages. It includes a handle_client function that runs in a separate thread for each connected client. This function receives messages from the client, handles disconnections, prints received messages, and broadcasts them to all other connected clients using the broadcast function. The server runs in a continuous loop, accepting new connections and spawning new threads for each client.

***) Client-side Implementation**

```
import tkinter as tk
from tkinter import scrolledtext, END
from threading import Thread
import socket

class ChatGUI:
    def __init__(self):
        self.window = tk.Tk()
        self.window.title("Lan Chat(project)")

        # Text area to display messages
        self.message_area = scrolledtext.ScrolledText(self.window)
        self.message_area.pack(fill=tk.BOTH, expand=True)

        # Input field to type messages
        self.input_field = tk.Entry(self.window)
        self.input_field.pack(side=tk.LEFT, fill=tk.X, expand=True)

        # Send button
        self.send_button = tk.Button(self.window, text="Send", command=self.send_message)
        self.send_button.pack(side=tk.RIGHT)

        # self.input_field.bind('<Return>',self.send_message)

        self.client_socket = None
        self.receive_thread = None

    def run(self):
        self.window.mainloop()

    def send_message(self):
        message = self.input_field.get()
        self.input_field.delete(0, tk.END)

        if self.client_socket is not None:
            # Send the message to the server
            self.client_socket.send(message.encode('utf-8'))
            self.display_message(message, "You")

    def display_message(self, message, sender):
        formatted_message = f"{sender}: {message}\n"
        self.message_area.insert(tk.END, formatted_message)
        self.message_area.see(tk.END)

    def start_client_program(self):
        # Create a socket object
        self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        # Connect to the server
        self.client_socket.connect(('127.0.0.1', 5555))  # Replace with the server IP and port
```

```
        # Start a new thread to receive messages from the server
        self.receive_thread = Thread(target=self.receive_messages)
        self.receive_thread.daemon = True
        self.receive_thread.start()

    def receive_messages(self):
        while True:
            try:
                # Receive messages from the server
                message = self.client_socket.recv(1024).decode('utf-8')
                sender_ip = self.client_socket.getpeername()[0]
                self.display_message(message, sender_ip)
            except:
                # Handle server disconnection
                print('Disconnected from the server')
                self.client_socket.close()
                break

# Example usage
if __name__ == "__main__":
    gui = ChatGUI()
    gui.start_client_program()  # Start the client program and socket connection
    gui.run()
```

Client-side GUI:

The client-side GUI code is responsible for creating a graphical user interface that allows users to send and receive messages. It utilizes the tkinter library, which provides the necessary tools for building GUI applications in Python.

The ChatGUI class is defined, encapsulating the functionality of the GUI. In the constructor method, the GUI window is created using tk.Tk(), and the window title is set to "Lan Chat (project)".

The GUI consists of three main components: the message area, the input field, and the send button. The message area is implemented using the scrolledtext.ScrolledText widget, which provides a scrolling text area to display the chat history. The input field is a tk.Entry widget where users can type their messages. The send button is a tk.Button widget that triggers the send_message method when clicked.

The run() method starts the main loop of the GUI using self.window.mainloop(). This loop listens for events and updates the display accordingly.

The send_message() method is called when the user clicks the send button or presses Enter. It retrieves the text from the input field using self.input_field.get(), clears the input field using self.input_field.delete(), and sends the message to the server via the client socket. It also displays the sent message in the message area by calling the display_message method.

The display_message() method formats the received message and inserts it into the message area using self.message_area.insert(). The sender's name is displayed alongside the message. It also ensures that the message area is scrolled to the end using self.message_area.see(tk.END).

The start_client_program() method is responsible for setting up the client socket and initiating the connection to the server. It creates a socket object using socket.socket(socket.AF_INET,

socket.SOCK_STREAM), where AF_INET indicates the IPv4 protocol and SOCK_STREAM indicates a TCP socket. It then connects to the server using self.client_socket.connect(('127.0.0.1', 5555)), where you can replace '127.0.0.1' with the server IP and 5555 with the desired port number. Finally, it starts a separate thread to receive messages from the server by calling the receive_messages method.

The receive_messages() method runs in a separate thread and continuously receives messages from the server. It uses a while True loop to keep receiving messages until an exception occurs. It receives messages from the server using self.client_socket.recv(1024).decode('utf-8'), decodes them from bytes to a string using decode('utf-8'), and calls the display_message method to show the received message in the message area.

In the main section of the code, an instance of the ChatGUI class is created, and the start_client_program method is called to initiate the client program and establish a socket connection to the server. Finally, the run method is invoked to start the GUI's main loop.

***) Server-side Implementation**

```
import socket
import threading

def handle_client(client_socket, client_address):
    while True:
        try:
            # Receive message from the client
            message = client_socket.recv(1024).decode('utf-8')

            if not message:
                # Handle client disconnection
                print(f'{client_address[0]}:{client_address[1]} - Disconnected')
                client_socket.close()
                break

            print(f'{client_address[0]}:{client_address[1]} - {message}')

            # Broadcast the received message to all clients
            broadcast(message, client_socket)

        except ConnectionResetError:
            # Handle client disconnection
            print(f'{client_address[0]}:{client_address[1]} - Disconnected')
            client_socket.close()
            break

def broadcast(message, sender_socket):
    for client in clients:
        if client != sender_socket:
            try:
                # Send message to other clients
                client.send(message.encode('utf-8'))
            except:
                # Handle client disconnection
                client.close()
                clients.remove(client)
```

```
# Server configuration
HOST = '127.0.0.1'  # Listen on all available network interfaces
PORT = 5555

# Create a socket object
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to a specific address and port
server_socket.bind((HOST, PORT))

# Listen for incoming connections
server_socket.listen()

print(f'Server started on {HOST}:{PORT}')

clients = []
is_running = True  # Variable to control the server's running state

def stop_server():
    global is_running
    is_running = False
    # Close all client sockets
    for client_socket in clients:
        client_socket.close()
    # Close the server socket
    server_socket.close()

# Start a separate thread to listen for the stop command
def stop_thread():
    while is_running:
        command = input("Enter 'stop' to stop the server: \n")
        if command.strip().lower() == 'stop':
            stop_server()

threading.Thread(target=stop_thread).start()

while is_running:
    # Accept a new connection
    client_socket, client_address = server_socket.accept()
    print(f'New connection from {client_address[0]}:{client_address[1]}')

    # Add the new client to the list
    clients.append(client_socket)

    # Start a new thread to handle the client
    client_thread = threading.Thread(target=handle_client, args=(client_socket, client_address))
    client_thread.start()
```

Server-side Implementation:
The server-side implementation code handles the connections, message distribution, and client management. It utilizes the socket library to establish a socket connection and threading to handle multiple client connections concurrently.

The handle_client() function is defined to run in a separate thread for each connected client. It takes client_socket and client_address as input parameters. Inside the function, a while True loop is used to continuously receive messages from the client. It receives messages using client_socket.recv(1024).decode('utf-8'), decodes them from bytes to a string, and stores them in the message variable.

If the received message is empty (indicating the client has disconnected), the function handles the client disconnection by printing a "Disconnected" message and closing the client socket using client_socket.close(). Then it breaks out of the loop.

If the message is not empty, it prints the client's IP address and port number along with the received message.

The broadcast() function is defined to send a message to all connected clients except the sender. It takes the message and sender_socket as input parameters. It iterates over the clients list, which contains all the client sockets, and checks if the current client is not the sender. If so, it sends the message to that client using client.send(message.encode('utf-8')). If an exception occurs during sending (indicating client disconnection), the function closes the client socket and removes it from the clients list.

The server configuration is set by defining the HOST and PORT variables. HOST specifies the IP address to listen on, and PORT specifies the port number to listen on.

A server socket object is created using socket.socket(socket.AF_INET, socket.SOCK_STREAM). AF_INET indicates the IPv4 protocol, and SOCK_STREAM indicates a TCP socket.

The server socket is bound to the specified address and port using server_socket.bind((HOST, PORT)).

The server socket starts listening for incoming connections using server_socket.listen().

The stop_server() function is defined to stop the server gracefully. It sets the is_running flag to False, indicating that the server should stop. It then closes all client sockets by iterating over the clients list and calling client_socket.close() for each client. Finally, it closes the server socket using server_socket.close().

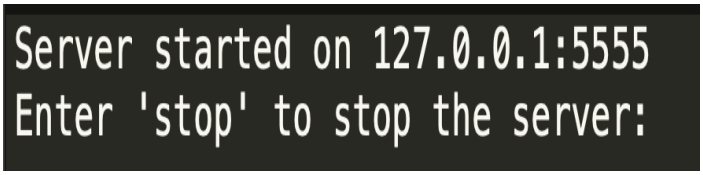
A separate thread is started to listen for the "stop" command. Inside the thread, a while loop continuously checks for user input using input(). If the input command is "stop", the stop_server() function is called to stop the server.

In the main section of the code, the server enters a continuous loop to accept incoming connections. It uses server_socket.accept() to accept a new connection, which returns the client socket and client address. The client socket is added to the clients list, and a new thread is started to handle the client by calling the handle_client() function with the client socket and address as parameters.

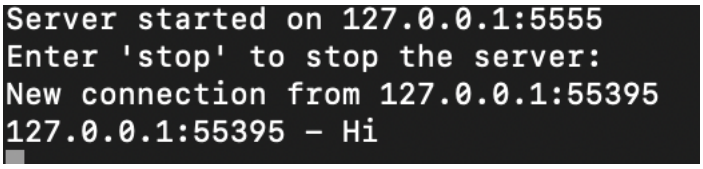
The main server loop continues to accept new connections and handle clients until the is_running flag is set to False, indicating that the server should stop.

Working demonstration:

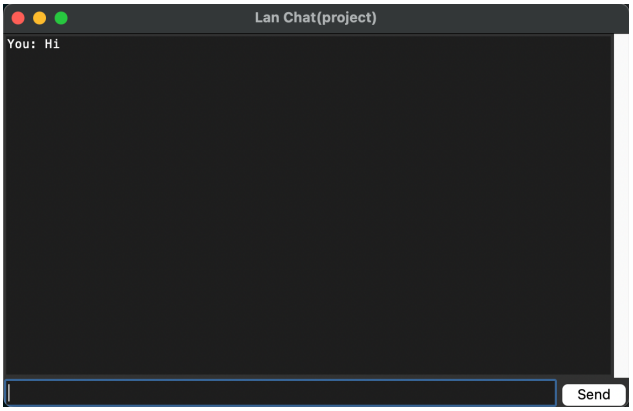
Server is started.



Client 1 is connected.

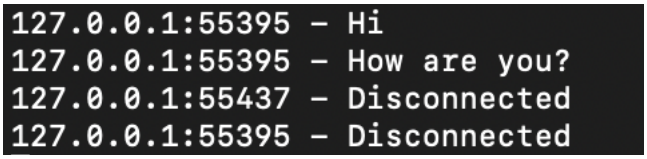


Client 1 GUI:



Client 2 GUI:

After both clients are disconnected:-



6. Conclusion

The client-server chat application project provides a basic framework for real-time communication between multiple clients. The client-side GUI offers a user-friendly interface for sending and receiving messages, while the server-side implementation handles the connections and message distribution. By utilizing sockets and threads, the application enables concurrent communication and supports multiple clients.

The project demonstrates the use of Python's tkinter library for GUI development and the socket library for network communication. It also showcases the utilization of threading to handle multiple client connections simultaneously. The project can serve as a starting point for developing more advanced chat applications with additional features like user authentication, file sharing, and encryption.

Overall, the client-server chat application project provides a practical implementation of network communication and graphical user interface, allowing users to engage in real-time conversations within a local network environment.