COMP308: COMPUTER NETWORKS

‘Real-Time Chat Application using Socket Programming’ Project Report

Instructor

Asst. Prof. Abdulkadir KÖSE

Group Members

Tolgahan KELEŞ - 2311051801

Elif Nur YILMAZ - 2111051034

Sude KARAOĞLAN - 110110285

Amine Sudenur SANCAK - 2014011014

Submission Date

09.06.2024

ABDULLAH GUL UNIVERSITY

KAYSERI, 2024

TABLE OF CONTENT

THE AIM OF THE PROJECT

The main goal of this project is to create a real-time chat application that makes use of a server to facilitate communication between numerous clients. The following are the goals:

- **Create Real-Time Communication:** Socket programming allows clients to communicate with each other in real-time.
- **Create a Scalable Server:** Design a server that can handle several client connections at once and effectively handle dynamic joins and leaves in order to create a scalable server.
- **Provide an Easy-to-Use GUI:** Use Tkinter to create a graphical user interface that makes it simple for users to join a conversation, send messages, and exit the application.
- **Ensure Robust Session Management:** Use server-side logic to disseminate messages, manage chat sessions, and handle client disconnections.
- **Ensure a Seamless User Experience:** Assemble all parts together to give users a responsive and easy-to-use chat experience.

By accomplishing these objectives, the project illustrates how socket programming and GUI development can be used practically to create a useful and engaging conversation application.

FEATURES:

- **Real-Time Communication:**

The chat application enables instant message exchange among multiple clients through efficient socket programming, ensuring minimal delays and offering a smooth chatting experience for both casual and professional use.

- **Scalable Server:**

The server efficiently handles multiple clients concurrently, ensuring scalability and seamless management of dynamic connections. This robust design maintains performance and reliability, even with a growing number of clients.

- **User-Friendly GUI:**

The chat application's GUI, built with Tkinter, features a message display window, an input area, and essential controls for joining, sending messages, and exiting, offering an intuitive design accessible to users of all levels without requiring technical knowledge.

- **Session Management:**

The server tracks connected clients, broadcasts messages to all participants, and notifies existing clients when new users join or leave, fostering a sense of community and maintaining an organized chat environment.

FUNCTIONAL REQUIREMENTS:

- **Socket Programming:**

Real-time communication between the chat app's server and clients is made possible using the socket library in Python. It maintains efficiency and security by ensuring dependable message flow between several clients and the server through specified protocols.

- **Server-Side Functionality:**

The server script initializes and uses a multi-threaded technique to handle numerous clients concurrently while listening for client connections. It ensures responsiveness and seamless operation by broadcasting and receiving messages. Resilient error handling keeps things stable and effectively handles disconnections.

- **Client-Side Functionality:**

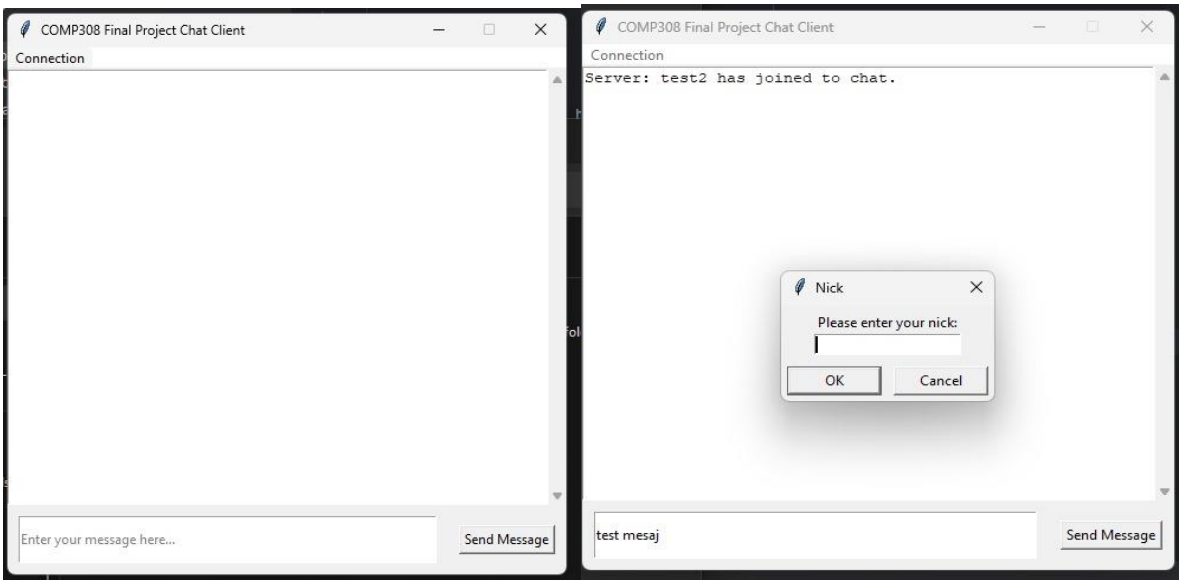
Real-time messaging is made possible on the client side by a Python script that establishes a connection between the client and the server. By processing user input and demonstrating

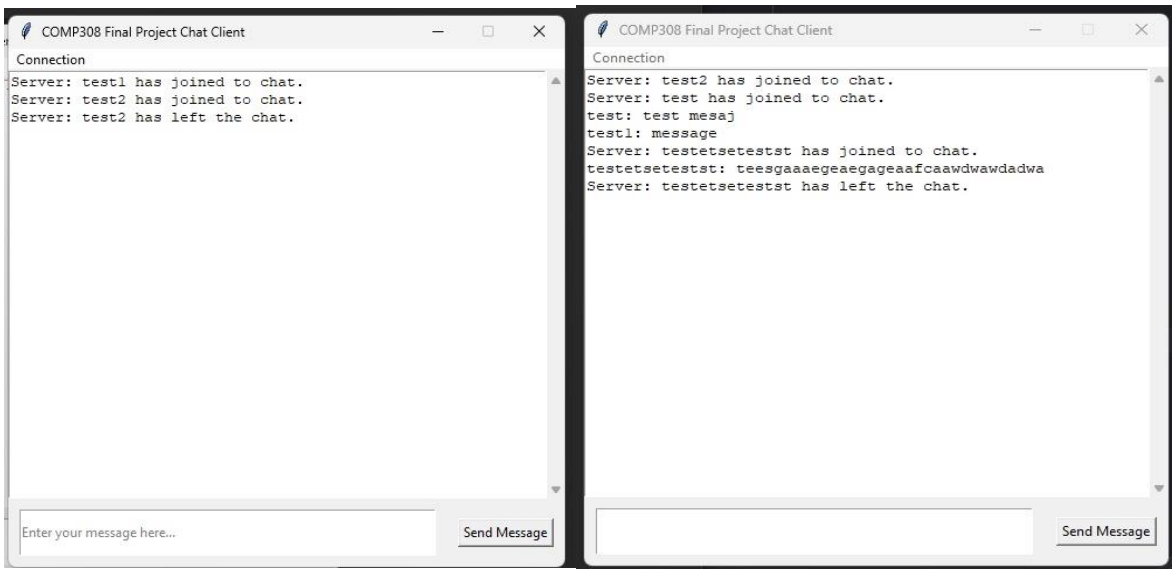
incoming messages in the chat window, this script connects with the GUI. It is simple for users to communicate through the application because of the seamless integration, which guarantees straightforward interaction.

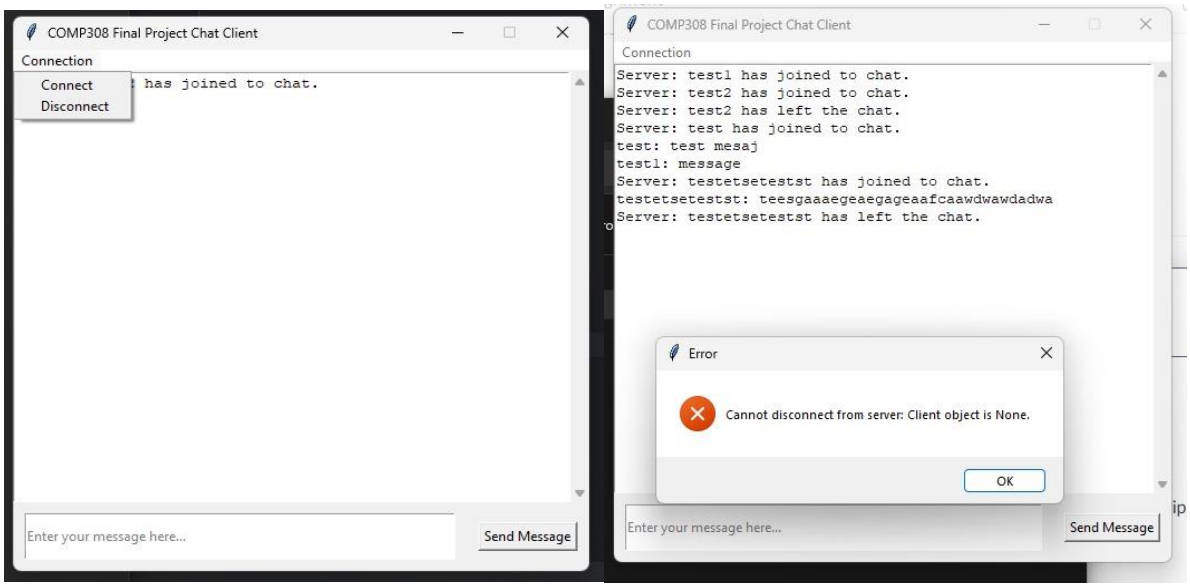
- **GUI Design:**

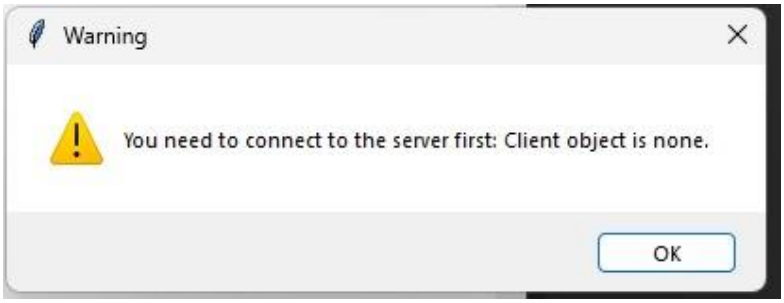
The Tkinter-created chat application's GUI has an input field and a message display area. It has buttons to enter the conversation, start a new message, and close the program. The interactive user experience is improved by the design's support for dynamic updates, which display new messages and user activity in real-time.

GUI SCREENSHOTS:




COMP308 Final Project Chat Client
Connection
Server: test1 has joined to chat.
Server: test2 has joined to chat.
Server: test2 has left the chat.
Enter your message here...
Send Message
COMP308 Final Project Chat Client
Connection
Server: test2 has joined to chat.
Server: test has joined to chat.
test: test mesaj
test1: message
Server: testetsetestst has joined to chat.
testetsetestst: teesgaaaegeaegageaafcaawdwawdadwa
Server: testetsetestst has left the chat.
Send Message


COMP308 Final Project Chat Client
Connection
Connect
Disconnect
has joined to chat.
Enter your message here...
Send Message
COMP308 Final Project Chat Client
Connection
Server: test1 has joined to chat.
Server: test2 has joined to chat.
Server: test2 has left the chat.
Server: test has joined to chat.
test: test mesaj
test1: message
Server: testetsetestst has joined to chat.
testetsetestst: teesgaaaegeaegageaafcaawdwawdadwa
Server: testetsetestst has left the chat.
Error
Cannot disconnect from server: Client object is None.
OK
Enter your message here...
Send Message


Warning
You need to connect to the server first: Client object is none.
OK

PYTHON CODES FOR SERVER SIDE

Overview

server.py is a Python script that sets up a simple TCP chat server. This server allows multiple clients to connect, exchange messages, and broadcast messages to all connected clients. The script uses the *socket* and *threading* libraries to handle network communication and manage concurrent client connections.

Class: *Server*

Attributes:

- HOST (*str*): The IP address the server will listen on. Default is *"127.0.0.1"*.
- PORT (*int*): The port number the server will listen on. Default is *8081*.
- LISTENER_LIMIT (*int*): The maximum number of clients that can connect to the server simultaneously. Default is *3*.
- splitting_char (*str*): The character used to separate the username and message. Default is *"~"*.
- active_clients (*list*): A list to store tuples of active clients' usernames and their corresponding socket objects.

Methods:

- __init__(self, host="127.0.0.1", port=8081, listener_limit=3, splitting_char="~")
    - Initializes the server with the specified host, port, listener limit, and splitting character.
- send_message_to_all(self, message)
    - Broadcasts a message to all connected clients.
    - Parameters:
        - *message* (*str*): The message to be sent to all clients.
- listen_for_messages(self, client, username)

    - Listens for incoming messages from a specific client and broadcasts them to all other clients. If the connection is closed or an error occurs, it removes the client from the active clients list.
    - Parameters:
        - client (*socket*): The client socket object.
        - username (*str*): The username of the client.
- send_message_to_a_client(client, message)
    - Sends a message to a specific client.
    - Parameters:
    - client (*socket*): The client socket object.
    - message (*str*): The message to be sent.
- client_handler(self, client)
    - Handles the initial connection of a client, receives the client's username, and starts a new thread to listen for messages from the client.
    - Parameters:
        - client (*socket*): The client socket object.
- main(self)
    - Sets up the server socket, binds it to the specified host and port, and listens for incoming client connections. For each new connection, it starts a new thread to handle the client.

Usage:

To run the server, simply execute the *server.py* script. The server will start listening for incoming connections on the specified host and port.

```
if __name__ == "__main__":

    server = Server()

    server.main()
```

By default, the server listens on *127.0.0.1* and port *8081*. These can be changed by modifying the *Server* class instantiation.

Example:

1. Start the server:

    python server.py

2. Connect clients to the server using a TCP client application. Each client should send their username first, followed by messages they want to broadcast.

3. The server will broadcast any received messages to all connected clients, prefixed by the sender's username.

Error Handling:

The script includes basic error handling for binding the server socket, receiving messages, and handling disconnections. If a client forcibly closes the connection or an error occurs, the server will remove the client from the active clients list and notify other clients about the disconnection.

PYTHON CODES FOR CLIENT SIDE:

1. `__init__(self, host="127.0.0.1", port=8081, splitting_char="~")`:

This method initializes the client class with default values for the host, port, and splitting character, setting up variables to store these values along with the client socket, username, message receiver callback, and connection status.

2. `_listen_for_messages_from_server(self, callback)`:

This method ensures continuous listening for incoming messages from the server while the client is connected, handling potential connection issues and errors as they arise.

3. `_communicate_to_server(self, username, on_message_received)`:

This method sends the username to the server and starts a new thread to listen for messages from the server.

4. `send_message_to_server(self, message)`:

The `send_message_to_server` method sends a message to the server only if the client is currently connected, raising exceptions if the client object is None or if the message is empty.

5. `connect(self, username, on_message_received)`:

The `connect` method establishes a connection to the server using the provided username and message receiver callback. It creates a new client socket, connects to the server, and initializes communication protocols. This method raises exceptions to handle invalid input or connection errors effectively.

6. `disconnect(self)`:

The `disconnect` method closes the client socket and sets the client object to None to terminate the connection with the server. It raises an exception if the client object is already None, ensuring proper disconnection handling.

These methods encapsulate the functionality required for a client to connect, communicate, send messages, and disconnect from a server. They handle errors and ensure proper communication between the client and server.

PYTHON CODES FOR GUI IMPLEMENTATION:

1. `__init__`:

This method initializes the main window, sets properties, creates a client instance, and a scrolled text box for displaying messages.

2. `create_message_send_box`:

It creates an entry field for typing messages with a placeholder and a send button.

3. `create_menu`:

It sets up a menu bar with options to connect and disconnect from the server.

4. `ask_nick`:

Ask nick method prompts the user to enter a nickname, handling cases where the user cancels or leaves the field empty.

5. `send_message`:

It sends the typed message to the server and clears the entry field, handling exceptions.

6. `on_message_received`:

This method displays received messages in the text box.

7. `on_connect_click`:

It handles the connection to the server, prompting for a nickname and showing errors if connection fails.

8. `on_disconnect_click`:

On disconnect click method handles disconnection from the server, showing errors if disconnection fails.

9. `show_warning`:

This displays a warning message.

10. `show_error`:

This method displays an error message.

11. `main`:

It initializes the menu and message send box, and starts the Tkinter main loop.

CONCLUSION:

This project successfully demonstrates the development of a real-time chat application using socket programming in Python. By combining server-side and client-side programming with a user-friendly GUI, the application offers an effective platform for real-time communication among multiple clients. The project highlights the practical application of network programming concepts and showcases the integration of network communication with graphical user interfaces.