United International University

Project Report On

Chat Application with Multiple Clients

Group Members:

#ID01 : 011201262, Azizul Islam Nayem

Course Title: Computer Networks Laboratory

Course Code: CSE 324/CSE 3712

Section: D

Instructor: Md. Enamul Haque

Department of Computer Science & Engineering

Project Overview

This project aims to develop a chat application that enables multiple clients to connect to a server simultaneously and communicate with each other in real time. The objective is to create a simple yet effective command-line-based chat application where users can send messages to all other connected clients, demonstrating client-server architecture and threading in Python.

System Architecture

The application utilizes a client-server architecture in which the server oversees multiple client connections. It continuously listens for client requests, accepts incoming connections, and allocates a unique identifier to each client. This configuration facilitates smooth communication between clients, as the server serves as an intermediary, relaying messages from one client to all others.

Threading Implementation

The server uses threading to manage multiple client connections simultaneously. When a new client connects, the server creates a thread using Python's threading module. Each thread handles communication with a specific client, allowing messages to be sent and received without interrupting the main server. This setup ensures smooth and efficient message processing. **threading.Thread(target=handle_client, args=(client_socket, addr, client_name, available_index)).start()** -> This line creates a new thread to run the handle_client function for each connected client, enabling independent communication without affecting other clients. This keeps the chat environment responsive for everyone.

Broadcasting Mechanism

The broadcasting mechanism is designed to allow messages sent by one client to be received by all other connected clients. When a client sends a message, the server processes it and broadcasts it to all clients, except the sender. This is achieved using the broadcast function, which iterates through all connected clients and sends the message to each of them. **client['socket'].send(message)** -> This line ensures that the message is sent to each client's socket, allowing them to receive and display the message in real-time.

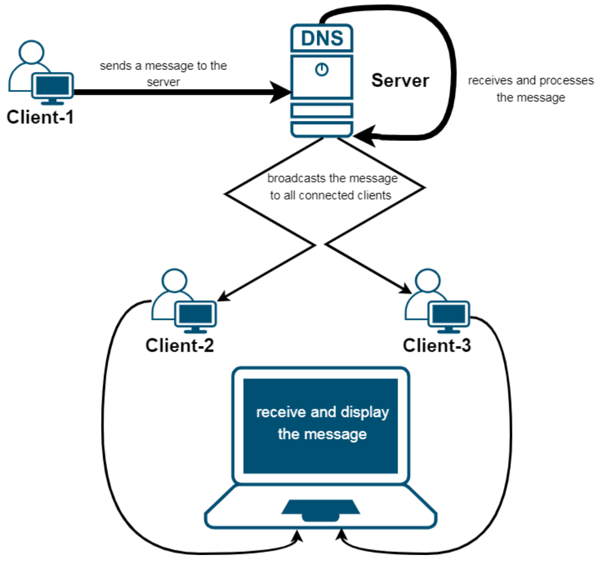
Message Handling

Message encoding and decoding are crucial for ensuring proper communication between clients and the server. In this implementation, messages are encoded using UTF-8 before being sent and decoded upon receipt. This ensures that messages maintain their integrity and can be accurately displayed to users. **client_socket.send(message.encode())** -> This line encodes the message using

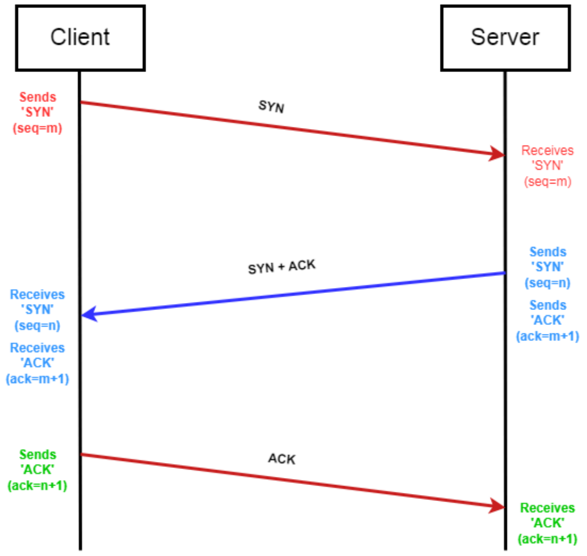
UTF-8 encoding before sending it over the network. **message = client_socket.recv(1024).decode()** -> This line decodes the received message using UTF-8 to convert it back to a string format for display or further processing.

Client-Server Communication Flow

The flow of communication begins when a client sends a message to the server. The server receives the message, processes it, and then broadcasts it to all other connected clients. The clients listen for incoming messages on their respective threads and display them in real time. The following diagram, made by me, illustrates the message broadcasting process in a client-server communication system.


DNS
Client-1
sends a message to the server
Server
receives and processes the message
broadcasts the message to all connected clients
Client-2
Client-3
receive and display the message

The following diagram, made by me, illustrates the handshaking process of communication between the client and server:



Handling Disconnections

The system is designed to handle client disconnections gracefully. When a client disconnects, the server detects the event and updates its state accordingly. The process is managed within the handle_client function using the following steps:
client_socket.close() -> Closes the connection for the disconnected client.
clients.pop(client_name) -> Removes the client from the dictionary of active clients.
client_status[client_index] = False -> Marks the client as disconnected.
broadcast(f"{client_name} has left the chat.".encode()) -> Broadcasts a message to inform remaining clients.

Challenges Encountered

During the development of this project, several technical challenges arose, particularly in managing multiple client connections and handling unexpected disconnections. One significant challenge was dealing with socket exceptions when a client disconnected unexpectedly. To address this, error handling was integrated into the **handle_client** function:

```
try:
    message = client_socket.recv(1024)
    if not message:
        break
except:
    break  -> Gracefully exit the loop on error
```

Additionally, when broadcasting messages, the server handles failures for specific clients without affecting others:

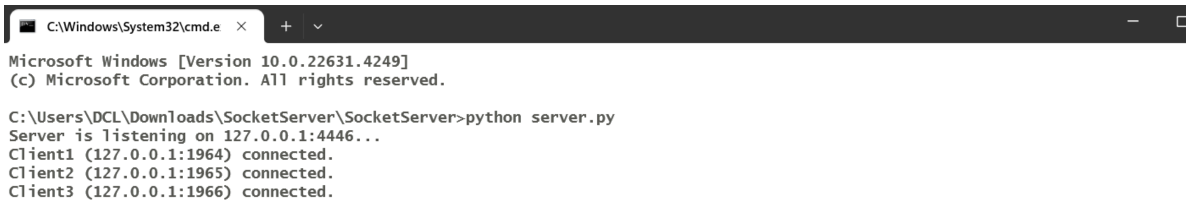
```
for client in clients.values():
    if client['socket'] != sender_socket:
        try:
            client['socket'].send(message)
        except:
            client['socket'].close()  -> Close the socket if sending fails
```

These error-handling strategies ensure the server remains stable and continues operating smoothly, even when a client disconnects unexpectedly.

Testing and Results

The system underwent thorough testing to ensure its functionality across key areas: Multiple clients successfully connected to the server simultaneously. Each connection was handled by a separate thread to ensure responsiveness:

```
client_socket, addr = server_socket.accept()
threading.Thread(target=handle_client, args=(client_socket, addr, client_name,
available_index)).start()
```

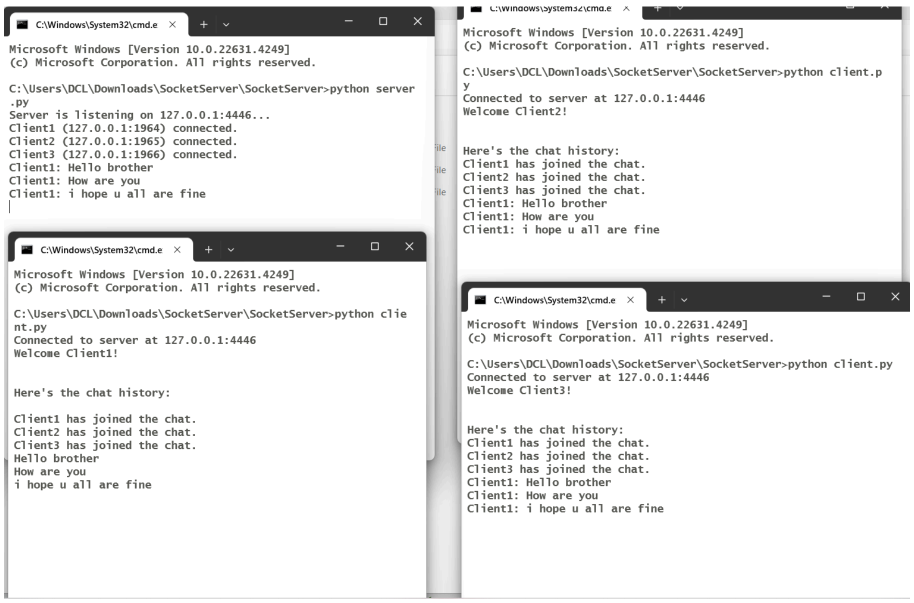


```
Microsoft Windows [Version 10.0.22631.4249]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DCL\Downloads\SocketServer\SocketServer>python server.py
Server is listening on 127.0.0.1:4446...
Client1 (127.0.0.1:1964) connected.
Client2 (127.0.0.1:1965) connected.
Client3 (127.0.0.1:1966) connected.
```

The broadcast functionality was tested to verify that messages sent by one client were received by all others (except the sender):

```
def broadcast(message, sender_socket=None):
    for client in clients.values():
        if client['socket'] != sender_socket:
            try:
                client['socket'].send(message)
            except:
                client['socket'].close()
```



```
Microsoft Windows [Version 10.0.22631.4249]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DCL\Downloads\SocketServer\SocketServer>python server
.py
Server is listening on 127.0.0.1:4446...
Client1 (127.0.0.1:1964) connected.
Client2 (127.0.0.1:1965) connected.
Client3 (127.0.0.1:1966) connected.
Client1: Hello brother
Client1: How are you
Client1: i hope u all are fine
```

```
Microsoft Windows [Version 10.0.22631.4249]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DCL\Downloads\SocketServer\SocketServer>python client.p
y
Connected to server at 127.0.0.1:4446
Welcome Client2!


Here's the chat history:
Client1 has joined the chat.
Client2 has joined the chat.
Client3 has joined the chat.
Client1: Hello brother
Client1: How are you
Client1: i hope u all are fine
```

```
Microsoft Windows [Version 10.0.22631.4249]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DCL\Downloads\SocketServer\SocketServer>python clie
nt.py
Connected to server at 127.0.0.1:4446
Welcome Client1!


Here's the chat history:

Client1 has joined the chat.
Client2 has joined the chat.
Client3 has joined the chat.
Hello brother
How are you
i hope u all are fine
```

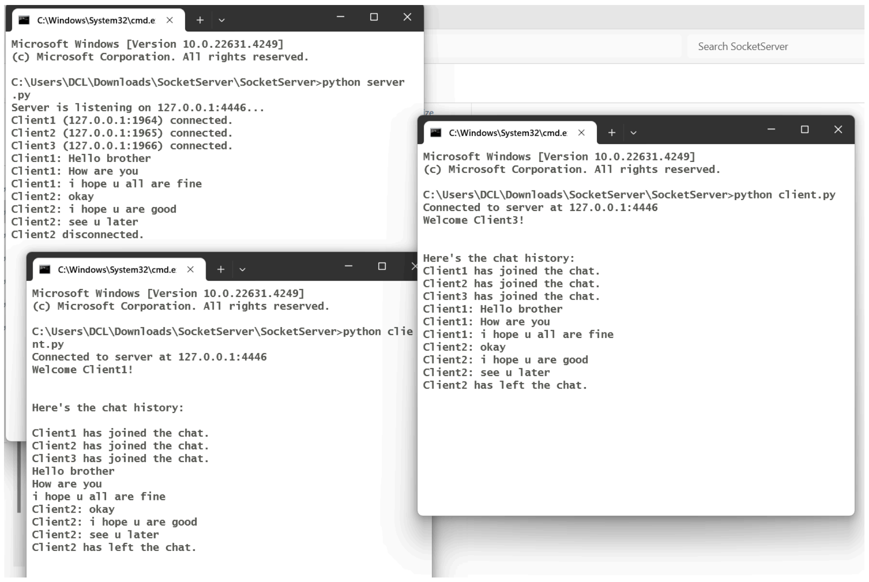
```
Microsoft Windows [Version 10.0.22631.4249]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DCL\Downloads\SocketServer\SocketServer>python client.py
Connected to server at 127.0.0.1:4446
Welcome Client3!


Here's the chat history:
Client1 has joined the chat.
Client2 has joined the chat.
Client3 has joined the chat.
Client1: Hello brother
Client1: How are you
Client1: i hope u all are fine
```

The server successfully managed client disconnections, removing the client from active connections and notifying others:

```
try:
    message = client_socket.recv(1024)
    if not message:
        break
except:
    break  -> Handle unexpected disconnections
broadcast(f"{client_name} has left the chat.".encode())
```



```
Microsoft Windows [Version 10.0.22631.4249]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DCL\Downloads\SocketServer\SocketServer>python server
.py
Server is listening on 127.0.0.1:4446...
Client1 (127.0.0.1:1964) connected.
Client2 (127.0.0.1:1965) connected.
Client3 (127.0.0.1:1966) connected.
Client1: Hello brother
Client1: How are you
Client1: i hope u all are fine
Client2: okay
Client2: i hope u are good
Client2: see u later
Client2 disconnected.
```

```
Microsoft Windows [Version 10.0.22631.4249]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DCL\Downloads\SocketServer\SocketServer>python clie
nt.py
Connected to server at 127.0.0.1:4446
Welcome Client1!


Here's the chat history:

Client1 has joined the chat.
Client2 has joined the chat.
Client3 has joined the chat.
Hello brother
How are you
i hope u all are fine
Client2: okay
Client2: i hope u are good
Client2: see u later
Client2 has left the chat.
```

```
Microsoft Windows [Version 10.0.22631.4249]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DCL\Downloads\SocketServer\SocketServer>python client.py
Connected to server at 127.0.0.1:4446
Welcome Client3!


Here's the chat history:
Client1 has joined the chat.
Client2 has joined the chat.
Client3 has joined the chat.
Client1: Hello brother
Client1: How are you
Client1: i hope u all are fine
Client2: okay
Client2: i hope u are good
Client2: see u later
Client2 has left the chat.
```

These tests confirmed the system's stability, accurate message delivery, and graceful handling of disconnections.

Conclusion and Future Work

The chat application enables real-time communication for multiple clients using Python's threading, ensuring smooth interaction and efficient message broadcasting.

Future improvements could include:

- GUI: Enhancing usability with features like chat rooms and user lists.
- User Authentication: Improving security by allowing only authorized users.
- Increased Client Capacity: Supporting larger groups or public chat rooms.

These enhancements would make the application more robust and user-friendly.