**CS-3001**

**COMPUTER NETWORKS**

**Assignment 02**

**Advanced Chat Application with Socket Programming**

**Submitted by:** Sarita Sangrez

**Roll number:** 23i-2088

**Section :** CY-A

**Date:** 24th April 2025

## TCP VERSION

**System Design**

This system implements a command-line based **chat system** over **TCP (Transmission Control Protocol)**. TCP is chosen for the following reasons:

**Reliable Delivery**: TCP ensures that all data packets arrive in the correct order and without loss, which is essential for chat communication, especially when sending private messages or files.

**Connection-Oriented Communication**: It maintains a persistent connection between client and server, making it easier to manage user sessions and states (e.g., login, message history).

**Built-in Congestion and Flow Control**: This makes the communication more stable and efficient for multiple simultaneous users.

**Code Explanation (Key Functions)**

* **Server Side**

1. **ClientInfo Struct:**  
   Stores client username, socket, and status (active).  
   **ClientInfo clients[MAX\_CLIENTS];**
2. **Message & File Management:**

**MessageHistory** stores up to 100 messages per client.

**FileHistory** stores up to 10 files per client, with Base64-decoded content.

1. **Authentication:**Function **check\_creds()** reads creds.txt and matches credentials.  
   Login session is initiated inside **handle\_client().**
2. **Command Handling in handle\_client():**

/msg <user> <text> → Stores message in history and notifies target user.

/history <num> → Retrieves the last <num> messages for the logged in user.

/file <user> <path> → Receives Base64 encoded file content, decodes, and stores it.

/filehistory <num> → Displays stored files with filename and content.

/help, /exit → Displays command list or disconnects.

1. **Base64 Decoding:**

Uses **CryptStringToBinaryA()** for decoding incoming encoded file strings.

* **Client Side**

1. **Connection Setup:**

Uses **socket(), connect()** to establish TCP connection with the server.

Prompts user for username and password for login.

1. **Command Input Loop:**

Sends commands to server via **send().**

Receives responses or data using **recv().**

1. **Command Processing:**

/msg → Sends private message.

/file → Encodes file in Base64 using **CryptBinaryToStringA()** and sends to server.

/history, /filehistory → Waits for file data from server and prints it.

/exit → disconnects.

1. **UI**   
   After receiving any message, the client prints it with a > prompt to simulate an active chat window.
2. **Base64 Encoding:**

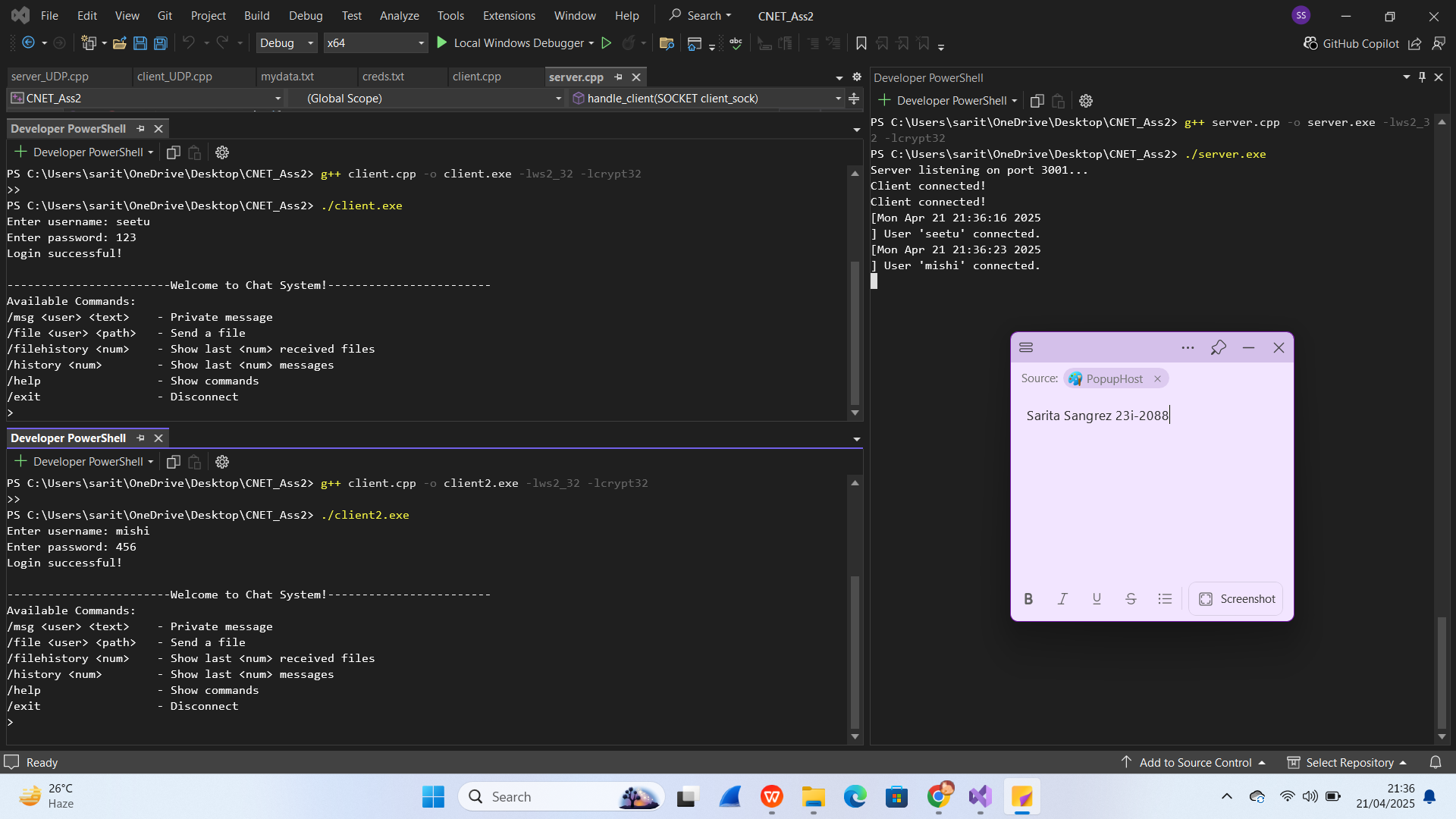
Files are converted to Base64 before sending to ensure safe text-based transmission.

**Challenges Faced**

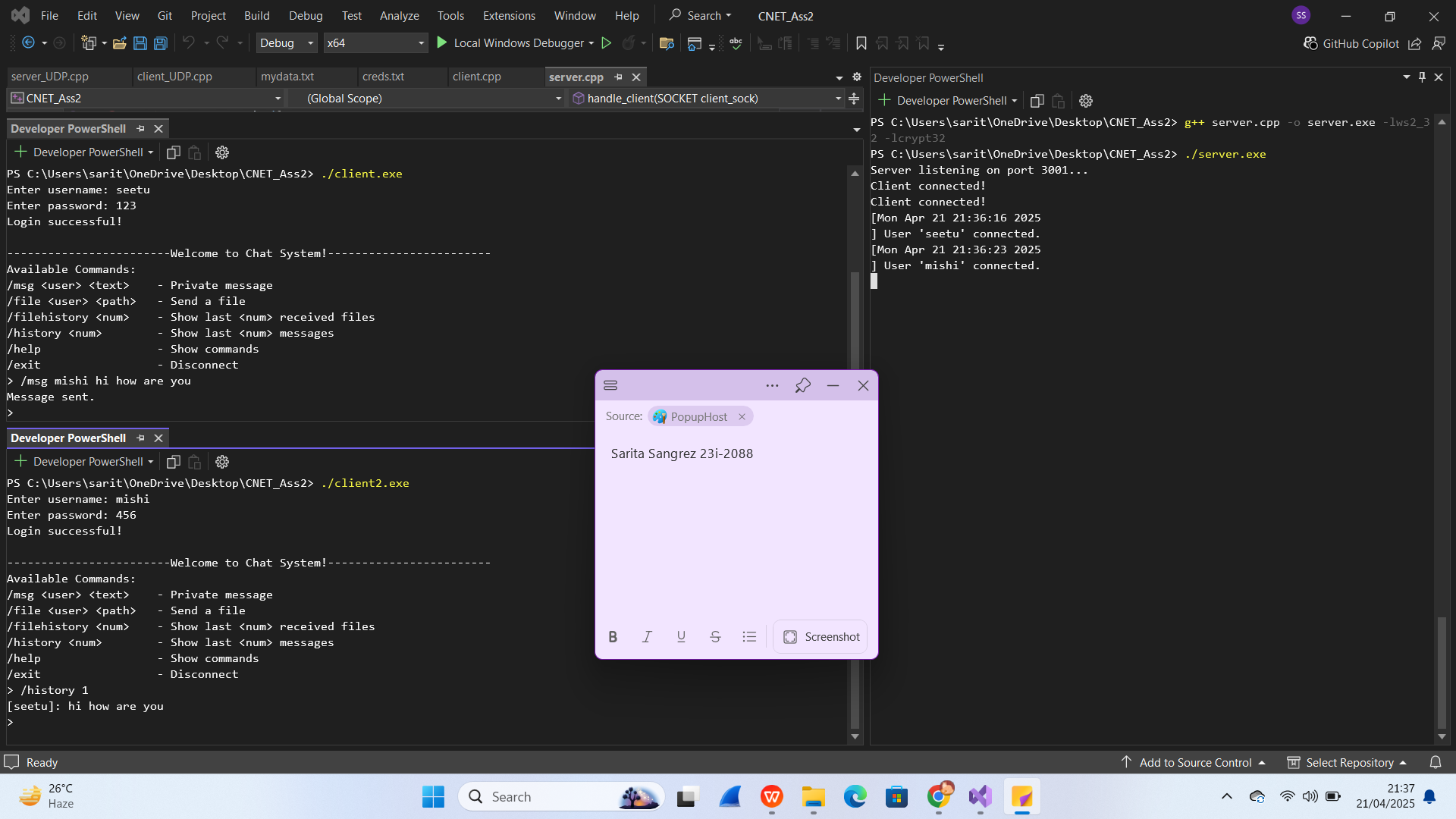
1. **Managing Multiple Clients**  
   It was tricky to handle several users at the same time, but using threads helped manage each client separately without blocking the server.
2. **Storing Messages and Files**  
   I had to create a way to store past messages and files for each user. I used fixed size arrays to keep it simple.
3. **Sending Files in Text Format**  
   I used Base64 text format before sending. So, this required careful encoding and decoding on both sides.
4. **Receiving Complete Data**  
   Sometimes, TCP doesn’t send all the data at once. I had to make sure the client and server waited and received the full message or file properly.

**Implementation (Testing + Wireshark )**

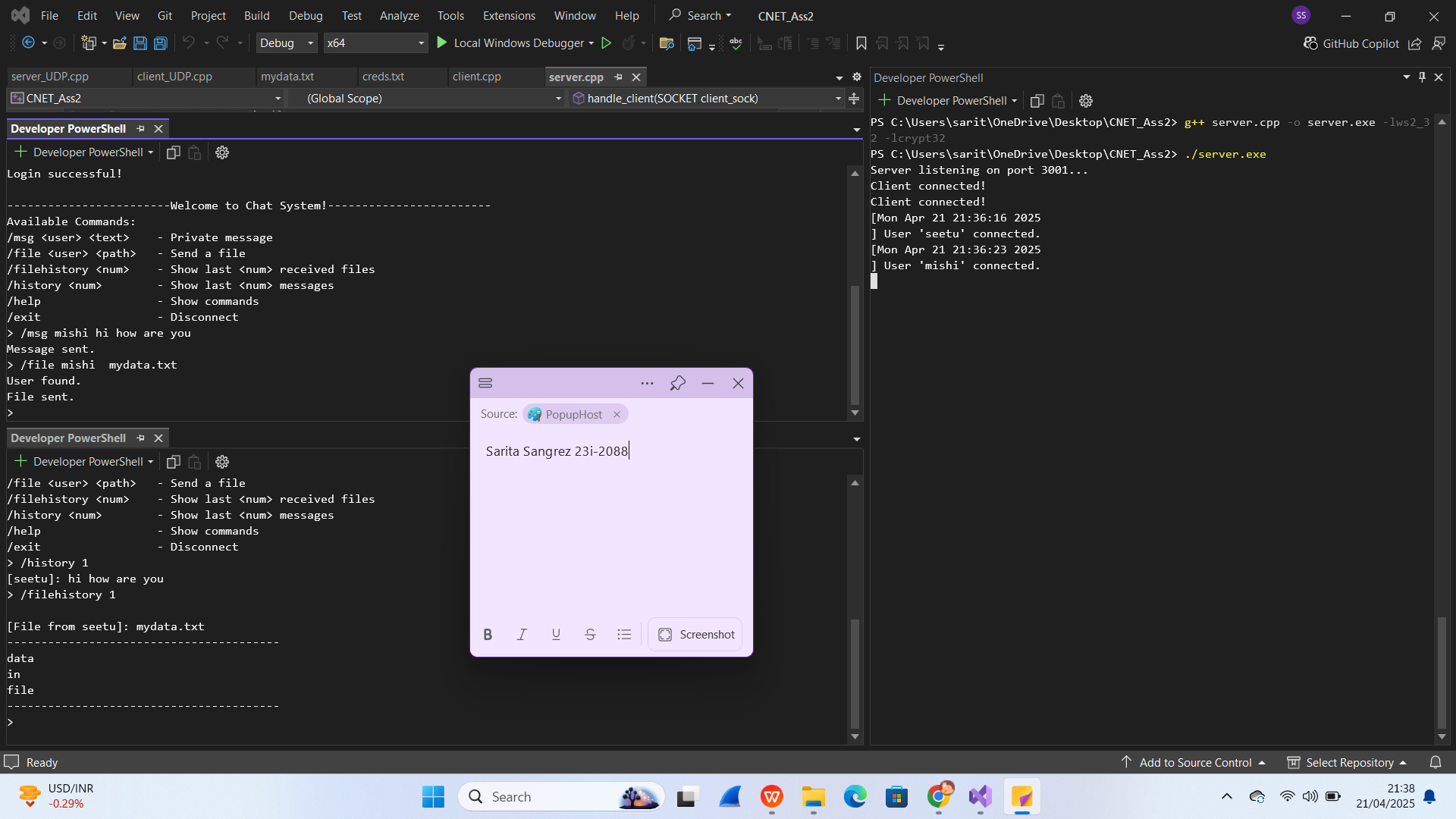
1. **User Authentication**



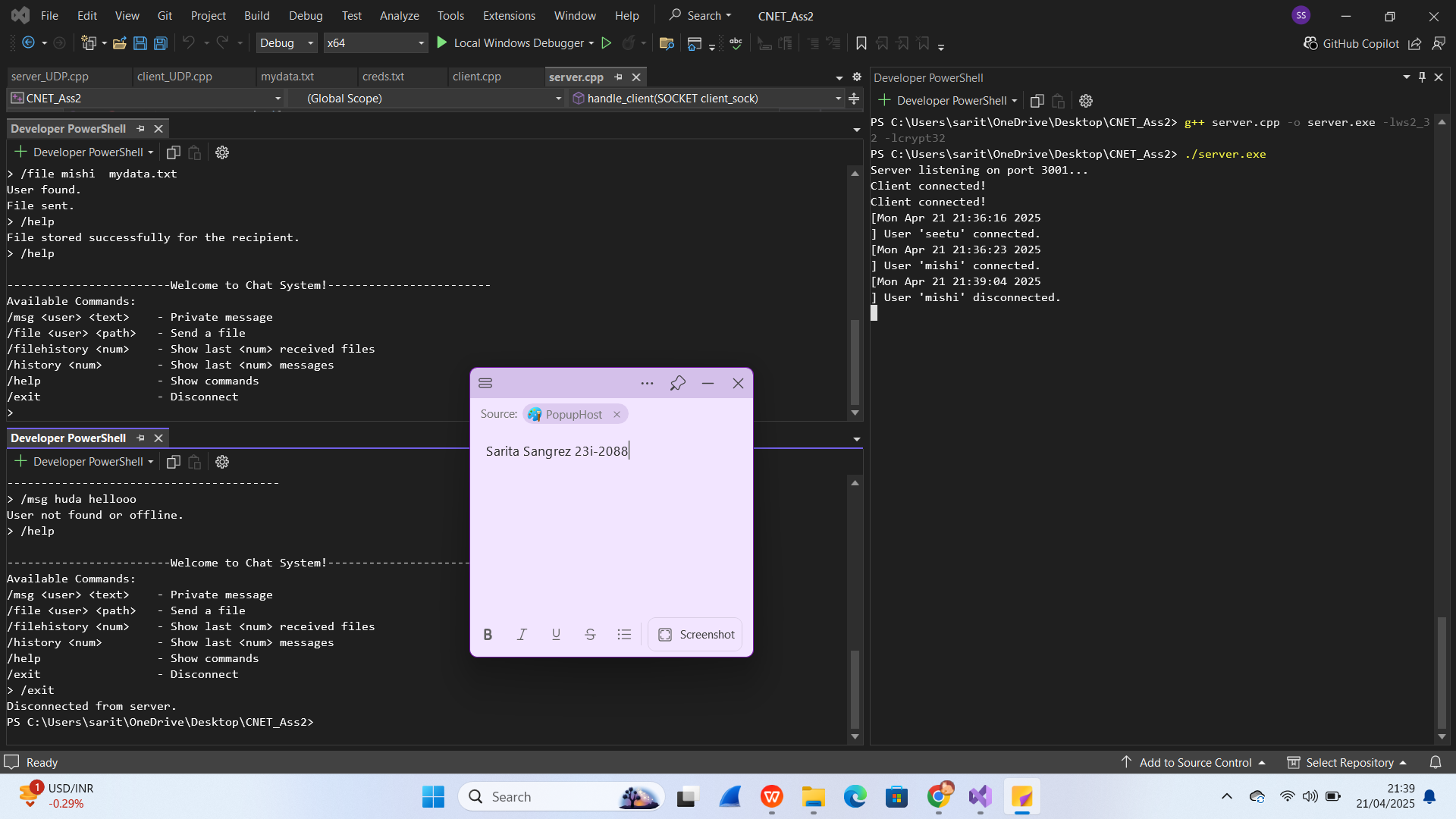
1. **Private Messaging**

****

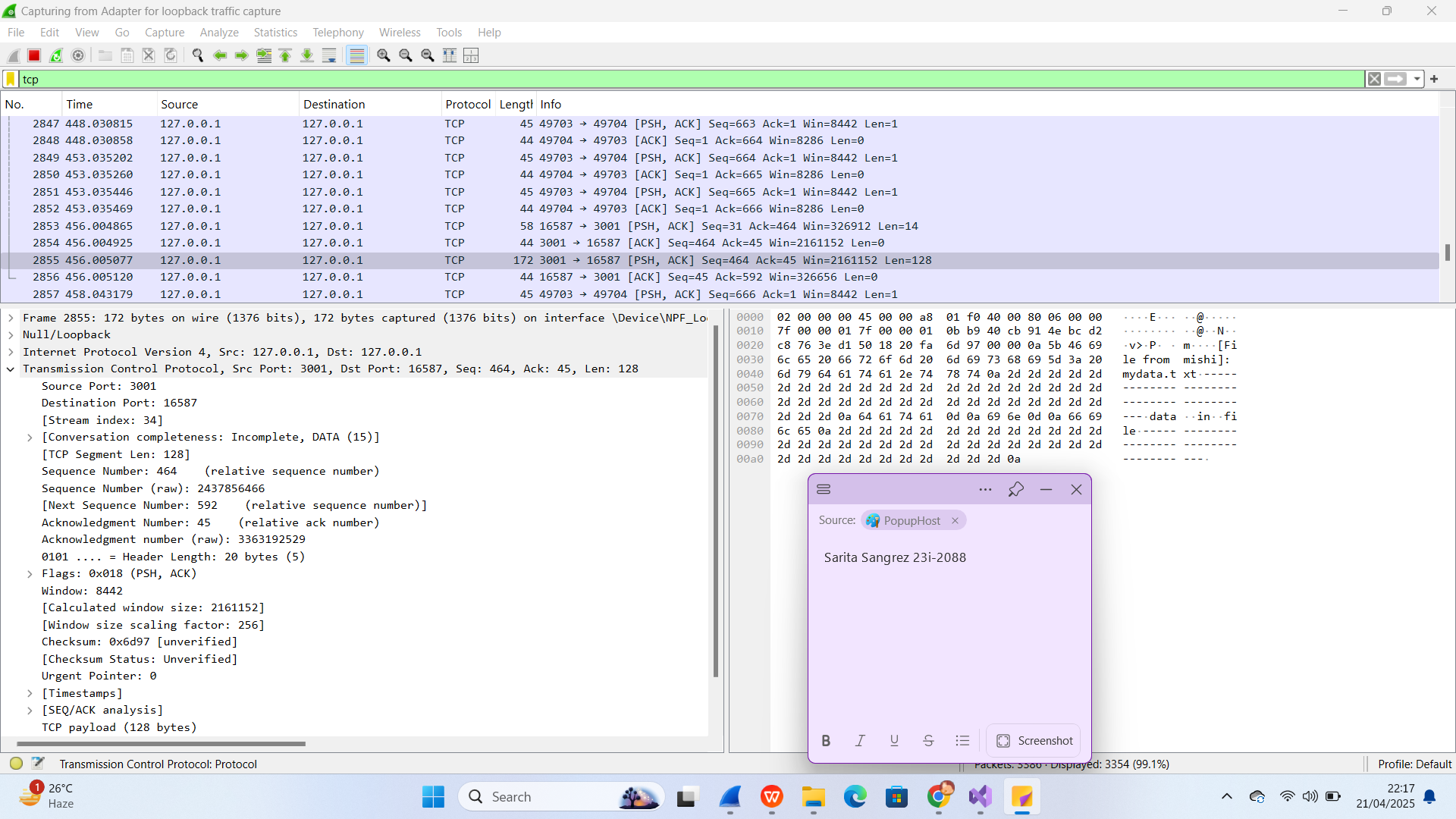
1. **File Transfer**

****

1. **Error Handling, Help Command and Exit**

****

1. **Wireshark**



## UDP VERSION

**System Design**

This part of the chat system uses UDP **(User Datagram Protocol)** for communication. Unlike TCP, UDP is connectionless and faster, making it suitable for broadcast messaging where speed matters more than reliability. UDP is chosen for the following reasons:

**No Connection Setup**: Clients can send messages directly without establishing a persistent connection.

**Low Overhead**: Less processing is required, making it lightweight and faster for broadcasting messages.

**Suitable for Broadcasting**: In this system, all messages sent using /msg are broadcasted to all connected users, which fits well with UDP's nature.

**Code Explanation (Key Functions)**

* **Server Side**

1. **Client Registration**When a client joins, it sends "client connected" and a username. The server stores this in an array of ClientInfo.

**void addClient(SOCKET sock, sockaddr\_in addr, const string& username);**

1. ****Command Processing Loop****  
   The server receives data using recvfrom() and checks the command type

/msg <text> → Broadcasts the message to all clients.

/status <online/away/busy> → Updates the sender's status and notifies others.

/exit → Marks the client as inactive and notifies others.

/help → Sends command menu to the client.

1. **Broadcasting Messages**  
   Sends formatted messages to all active clients except the sender.

**void broadcast(SOCKET sock, const char\* msg, int len, sockaddr\_in\* sender, const string& senderUsername);**

1. **User Join/Leave Notifications**  
   Automatically announces when a user joins or leaves, using a timestamp from **getTimestamp().**

* **Client Side**

1. **Setup & Username Input**

Client sends "client connected" followed by their username.

Receives and displays a welcome menu from the server.

1. **Receiving Messages**

A background thread runs **receiveMessages()** to constantly listen for incoming broadcasts.

1. **Sending Commands**  
   The main thread reads input and sends commands via **sendto()**:

/msg <text> → Broadcasts a message.

/status <status> → Updates and shares the current status.

/exit → Informs server and closes the connection.

/help → Requests and displays command menu.

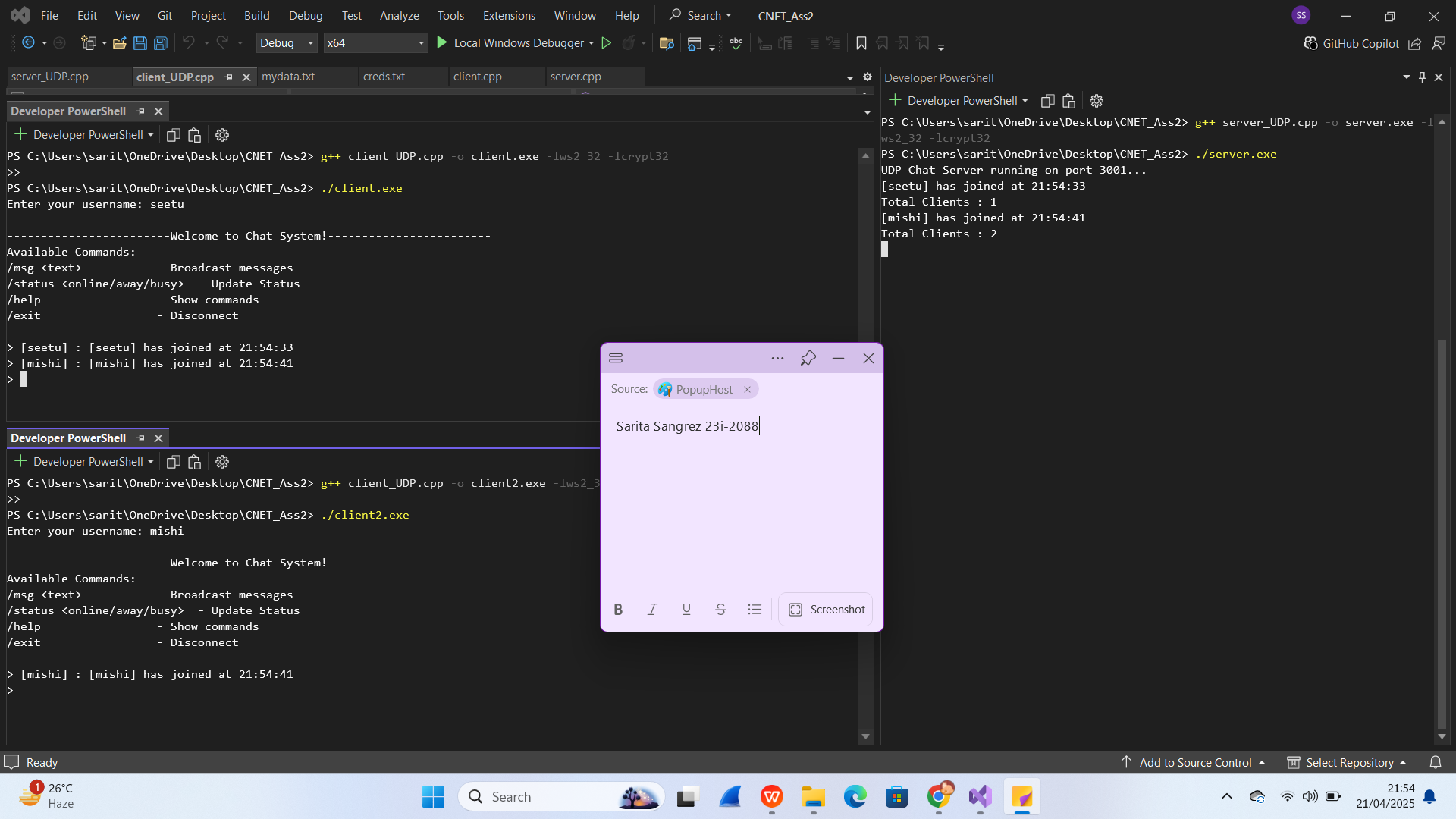
1. **UI**   
   After receiving any message, the client prints it with a > prompt to simulate an active chat window.

**Challenges Faced**

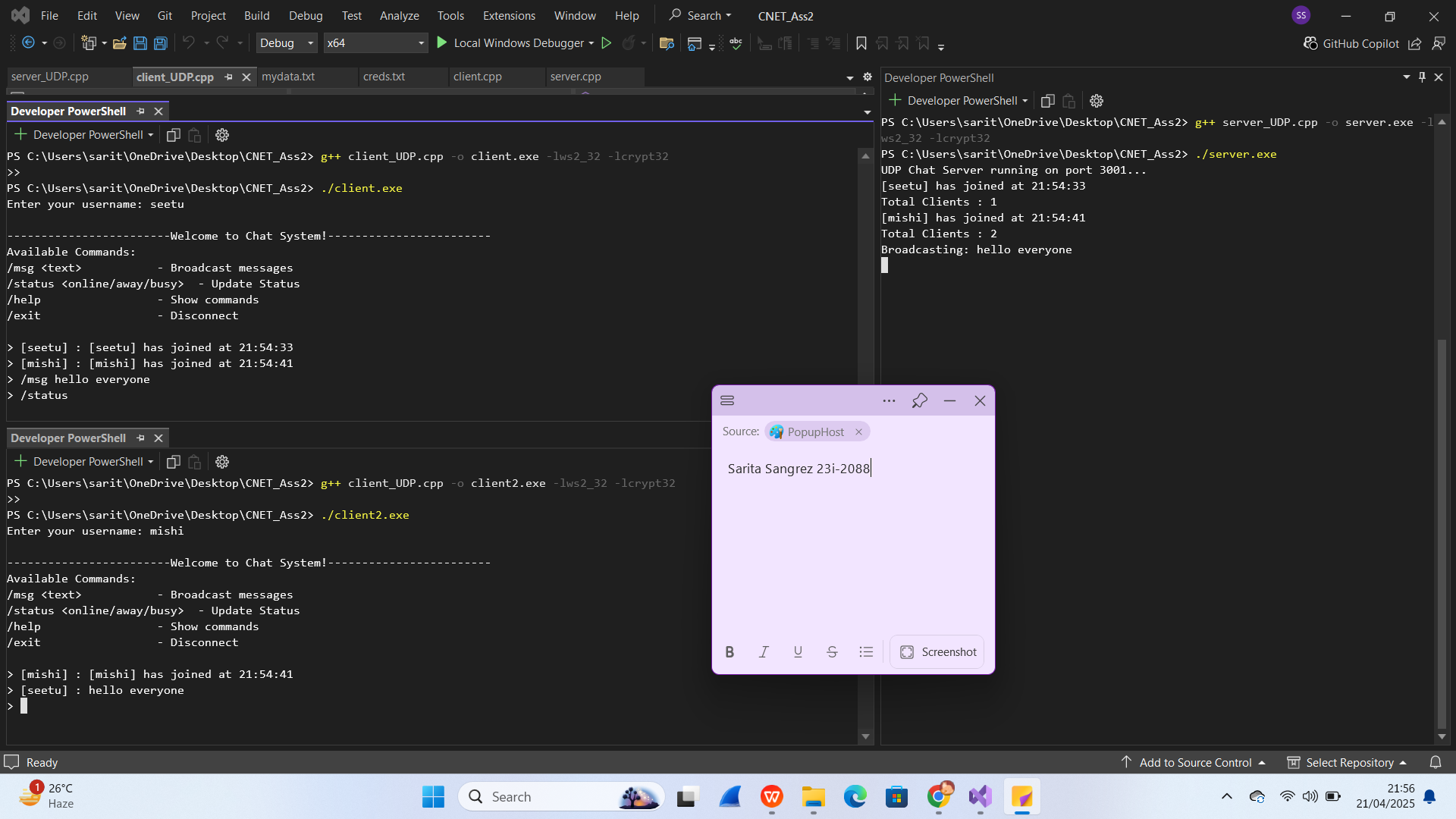
1. **Tracking Clients Without TCP Sessions**  
   Since UDP doesn’t have built-in connections, the server had to manually track clients using their IP and port.
2. **No Guaranteed Delivery**  
   Messages might be lost during transmission. This was acceptable for status updates and chat messages but not suitable for files or authentication.
3. **Identifying Clients**  
   I had to ensure that each client had a unique identifier based on their socket address and username combination.
4. **Broadcast Implementation**  
   Implementing clean and readable broadcast messages while avoiding sending to the message's sender took a lot of logical thinking.

**Testing Results**

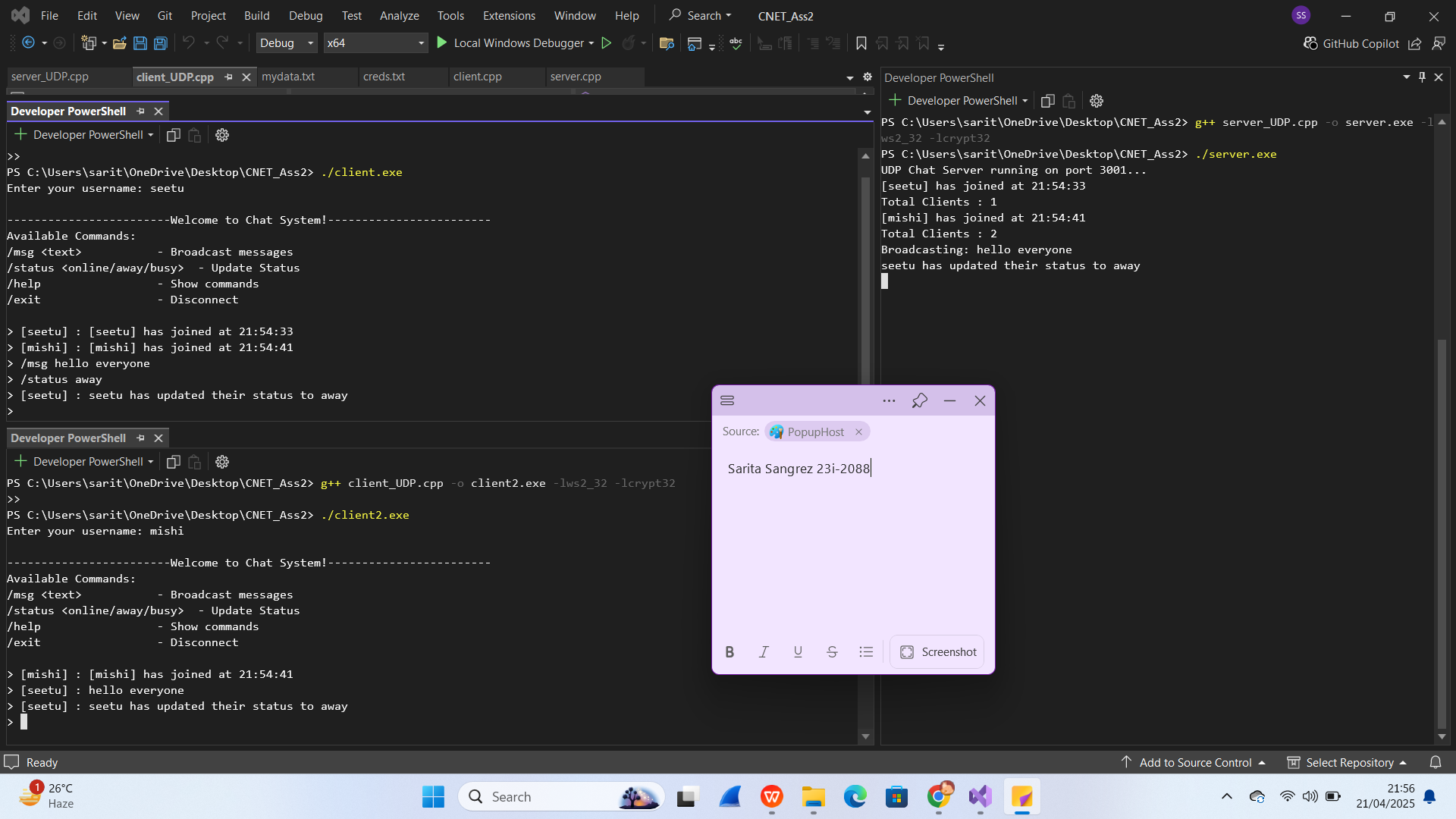
1. **Joining Message**

****

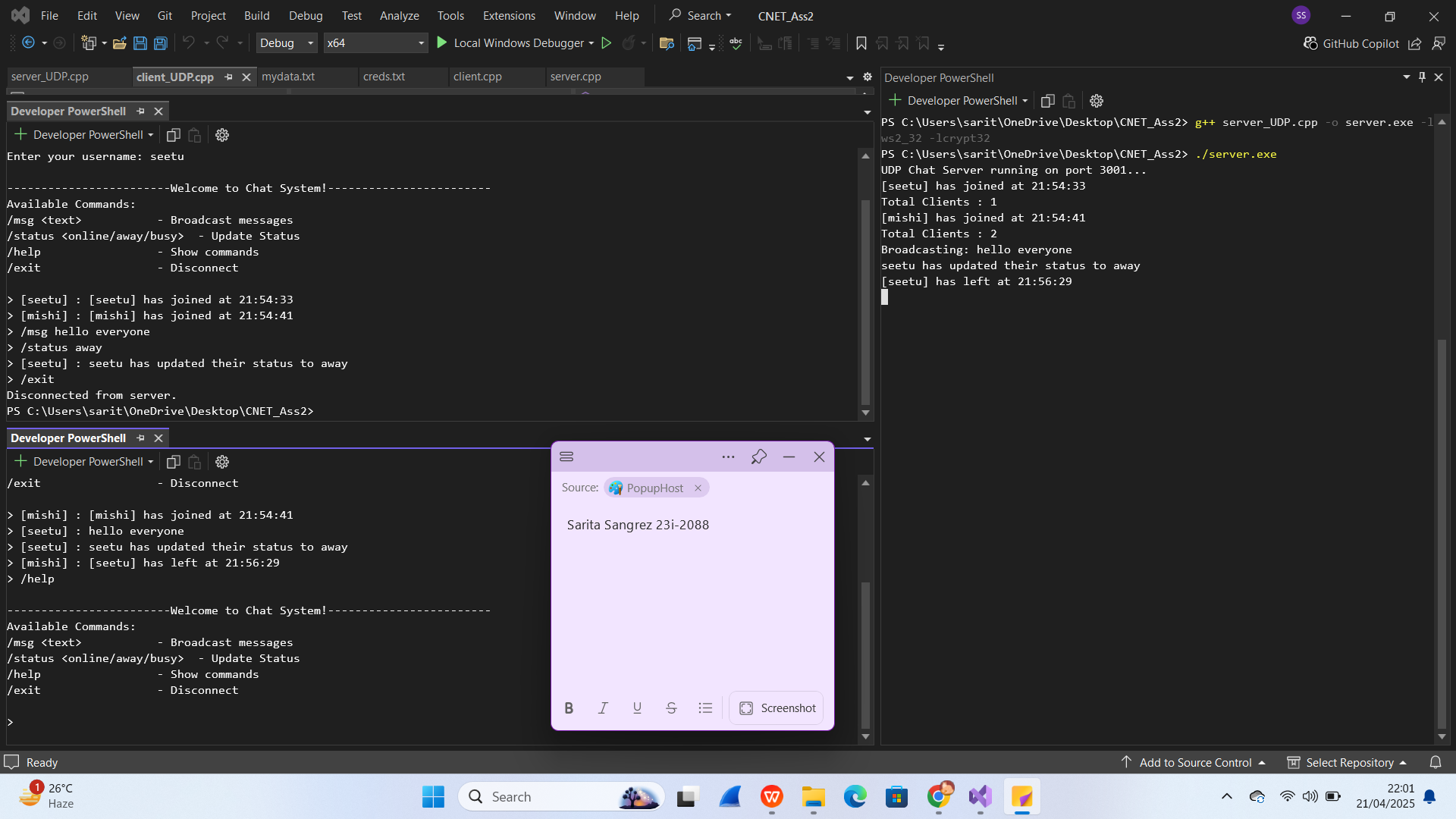
1. **Broadcast Message**

****

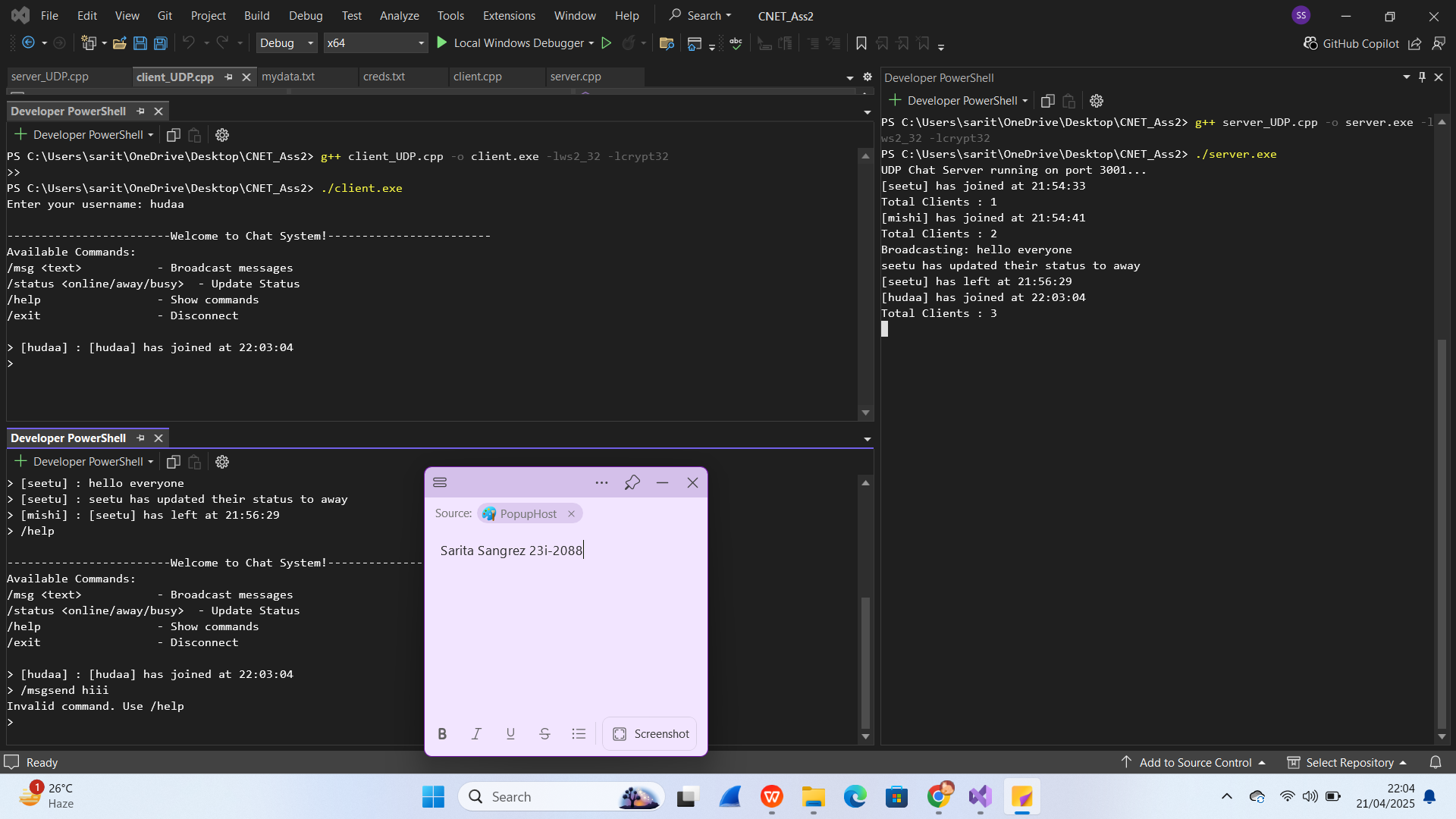
1. **Status Update**

****

1. **Help command and Exit**



1. **Error Handling**



1. **Wireshark**

