University of Science and Technology of Hanoi

MIDTERM REPORT

DISTRIBUTED SYSTEM

Hybrid Centralized and Peer-to-Peer Chat System Using Socket

Submitted by group 6:

Pham Minh Hieu	23BI14166
Nguyen Viet Hung	23BI14188
Nguyen Thanh Dat	23BI14091
Nguyen Hoang Long	23BI14270
Nguyen Huu Phuong	23BI14362

Under the teaching of:

Mr. Le Nhu Chu Hiep

December 2025

Table of Content

I. Project Description

This project implements a **hybrid centralized and peer-to-peer** (**P2P**) **chat system using TCP sockets**.
The system allows multiple users to communicate in real time across different computers in a distributed environment.

The system combines:

- A centralized server for user registration, login, and peer discovery.
- Direct peer-to-peer communication between clients for message delivery.

From the end-user perspective, the application provides:

- Real-time text messaging.
- User presence awareness (online users list).
- Direct communication without routing all messages through the server, improving scalability and reducing server load.

II. Architecture Design

The system follows a hybrid architecture, which consists of two main components:

1. Central Server

The server is responsible for:

- Handling user registration and authentication.
- Maintaining online user presence information.
- Providing peer address lookup (IP and port) for P2P communication.

The server does not participate in chat as a user and does not forward messages by default.

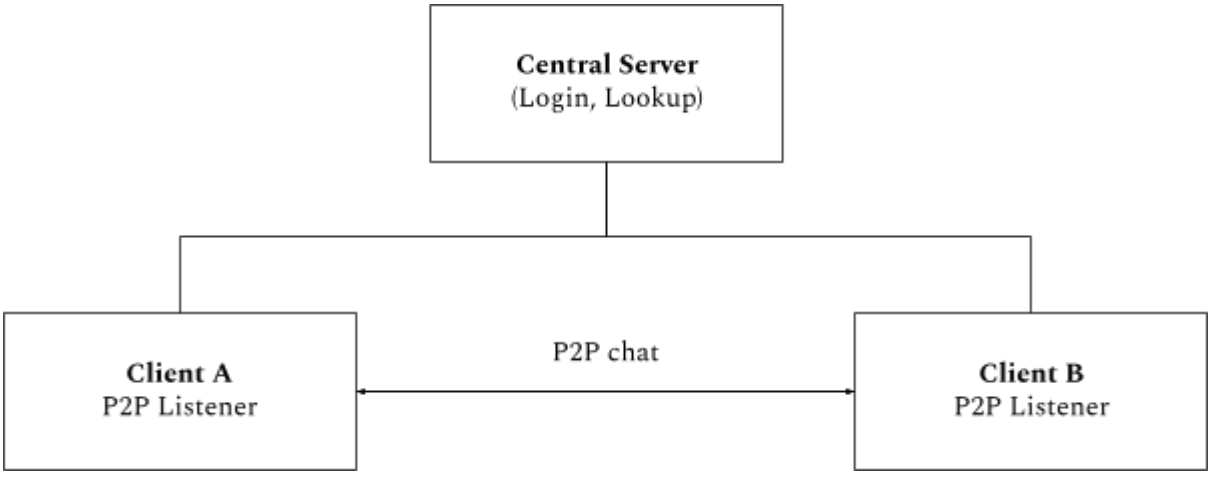
2. Clients

Each client:

- Connects to the central server for control operations.
- Opens a local TCP listening socket for peer-to-peer communication.

- Establishes direct TCP connections to other clients for message exchange.

3. Architecture Diagram



This architecture separates control plane (server) and data plane (P2P chat).

This diagram shows only 2 clients and 1 server for the logical view purpose. In reality, there can be many clients connected to the central server to chat to each other.

III. Protocol Design

All communication uses TCP sockets with JSON-based messages.

Messages are encoded in JSON format and terminated by a newline character (`\n`) to ensure correct message framing over TCP streams.

The following commands are implemented in the project:

Command	**Description**
REGISTER	Register a new user
LOGIN	Authenticate user and register P2P port
LIST	Retrieve list of online users
LOOKUP	Get IP and P2P port of a specific user

HEARTBEAT	Maintain online presence
LOGOUT	Disconnect from server

The protocol is divided into two layers:

- User-level commands, which are explicitly entered by users.
- Internal protocol messages (LOOKUP, HEARTBEAT), which are automatically generated by the client implementation and are transparent to the end user.

1. Server–Client Protocol

The server–client protocol is responsible for coordination and control, including user registration, authentication, presence management, and peer discovery.

1.1. User-Level Commands

These commands are entered directly by users during system operation.

a. Register

User command: reg <username> <password>

Client → Server:

```
{
    "type": "REGISTER",
    "username": "kali",
    "password": "123"
}
```

Server → Client:

```
{
    "type": "REGISTER_RES",
    "ok": true,
    "message": "Registered"
}
```

b. Login

User command: login <username> <password>

Client → Server:

```
{
```

```
        "type": "LOGIN",
        "username": "kali",
        "password": "123",
        "p2p_port": 6001
    }
```

Server → Client:

```
    {
        "type": "LOGIN_RES",
        "ok": true,
        "message": "Logged in",
        "your_ip": "100.72.92.42"
    }
```

The server determines the client's IP address directly from the TCP connection to prevent address faking.

c. List Online Users

User command: list

Client → Server:

```
    {
        "type": "LIST",
    }
```

Server → Client:

```
    {
        "type": "LIST_RES",
        "online": [
        {
            "username": "kali",
            "ip": "100.72.92.42",
            "p2p_port": 6001
        },
        {
            "username": "winvm",
            "ip": "100.100.250.116",
            "p2p_port": 6002
```

```
        }
        ]
    }
```

1.2. Internal Server–Client Messages

The following messages are not directly visible to users. They are automatically generated by the client to support system consistency and peer discovery.

a. LOOKUP

When a user issues a chat command, the client automatically requests peer address information from the server.

Automatically triggered message:

```
{
    "type": "LOOKUP",
    "username": "kali",
}
```

Server response:

```
{
    "type": "LOOKUP_RES",
    "ok": true,
    "username": "winvm",
    "ip": "100.100.250.116",
    "p2p_port": 6002
}
```

b. HEARTBEAT

After successful login, the client periodically sends heartbeat messages to notify the server that it is still online.

Automatically triggered message:

```
{
```

```
    “type”: “HEARTBEAT”,
}
```


Server response:


```
{
    “type”: “HEARTBEAT_RES”,
    “ok”: true
}
```


2. Client–Client Protocol

After obtaining peer address information from the server, clients communicate **directly with each other** using peer-to-peer TCP connections without passing through the server.

Chat Message

User command: chat <username> <message>

Client A → Client B:


```
{
    "type": "CHAT",
    "from": "kali",
    "to": "winvm",
    "text": "Hello from Kali!",
    "ts": 1734312345.12
}
```


Client B → Client A:


```
{
    "type": "ACK",
    “ok": true
}
```


IV. Deployment Guideline

The system follows a **hybrid architecture**, which consists of two main components:

1. Requirements

- Python 3.8 or later
- TCP/IP network connectivity (LAN or VPN). All computers must connect to the same network.
- Three computers (1 server, 2 clients). There can be more than 2 clients depending on your choice.

2. Running the Server

Command: `python server.py`

The server listens on TCP port 5000

3. Running the Clients

Command: python client.py

Each client provides:

- Server IP address
- Local P2P listening port (eg: 6001, 6002)

V. Usage Guideline

Available Commands:

Command	Description
reg <user> <pass>	Register new user
login <user> <pass>	Login to server
list	Show online users
chat <user> <message>	Send P2P message
logout	Disconnect
quit	Exit application

Example Workflow:

Step 1: User logs in.

Step 2: User requests online list.

Step 3: User selects another user and sends a message.

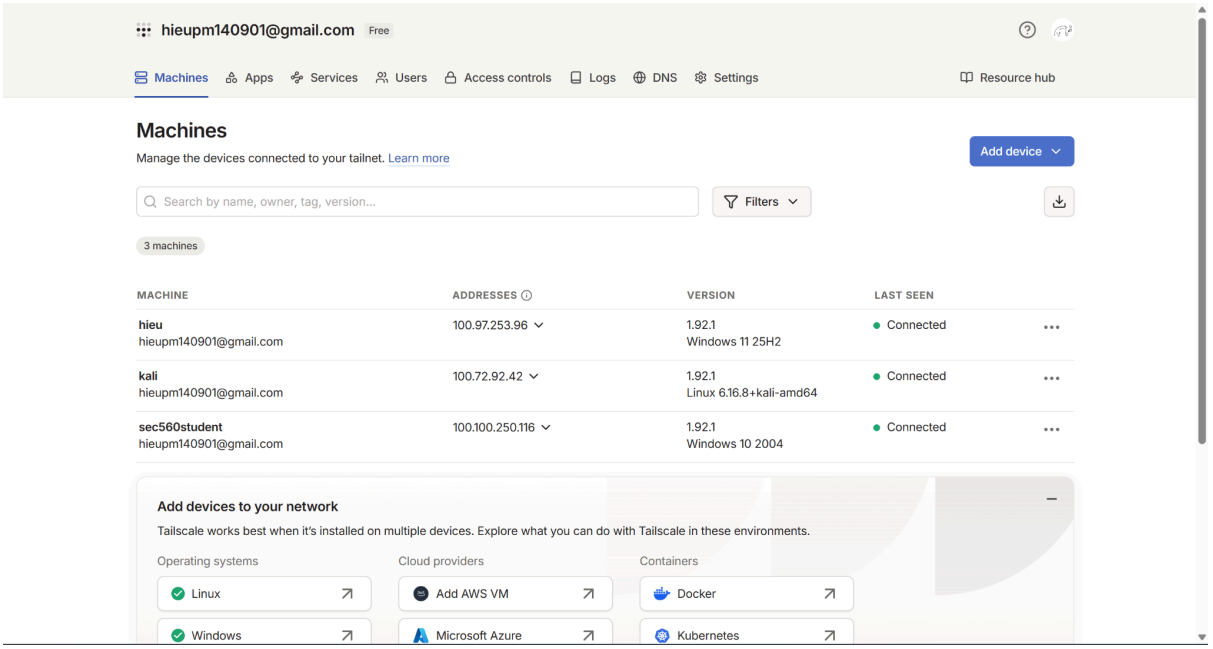
Step 4: Client establishes P2P connection automatically.

VI. Functional Demonstration

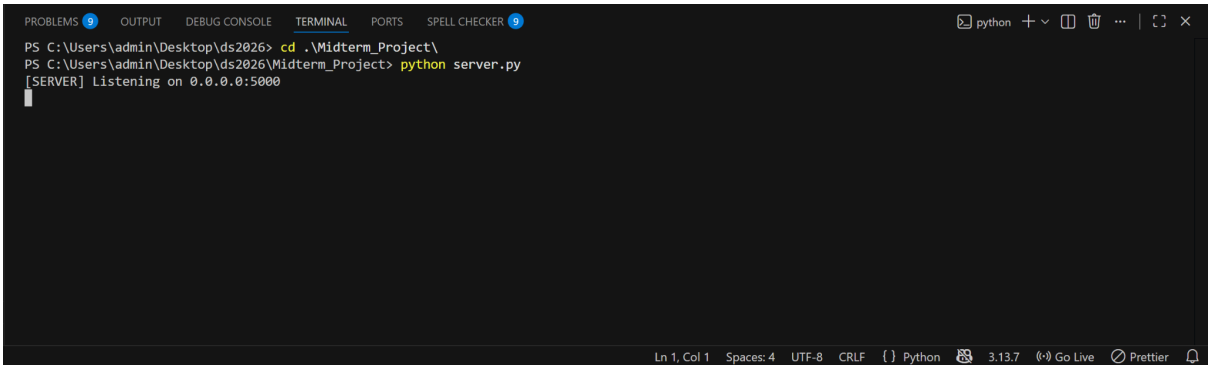
We demoed the project as an end user. We used 3 computers:

- A real machine Window for running the server.
- A virtual machine Kali acts as client A
- A virtual machine Window acts as client B

As we discussed above, all machines need to connect to the same network. So here, we used tailscale, a modern, easy-to-use mesh VPN service that creates a secure, private network (a "tailnet") between our devices. We installed tailscale on 3 machines then connected to it.

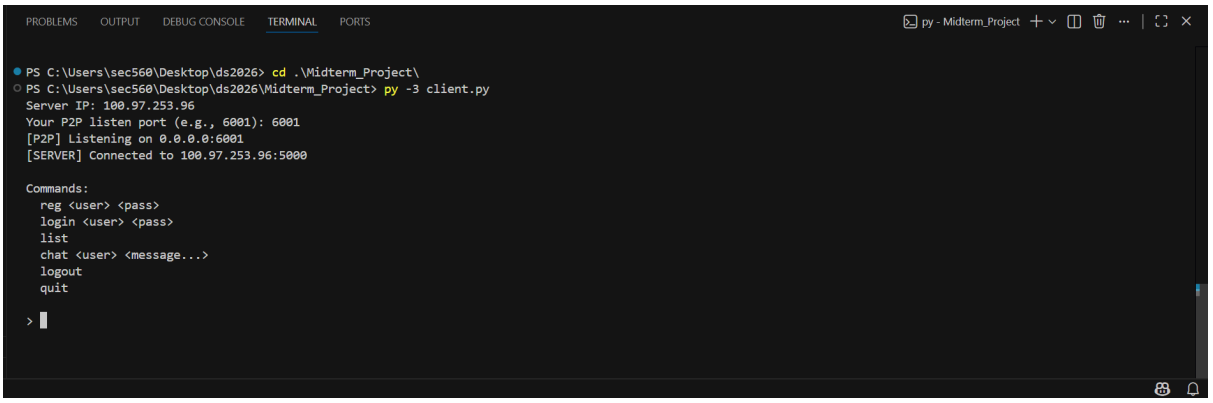


Step 1: Running Server. Here, we runned the server on Window real machine

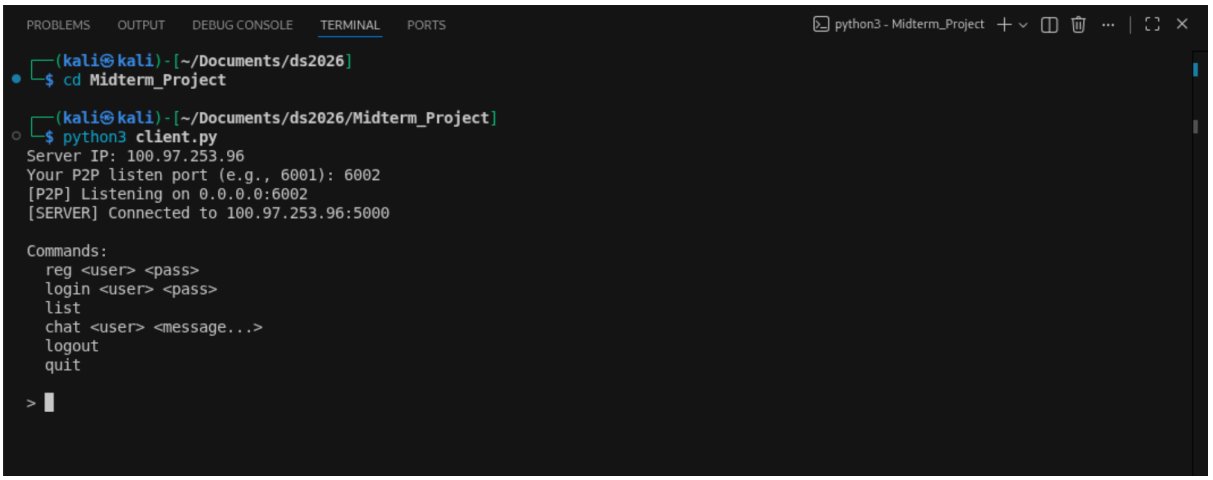


Step 2: On both virtual machines we runned the file client.py . Then, we enter the server IP address, and the P2P listen port (Kali: port 6002, Win: port 6001).

Window virtual machine:



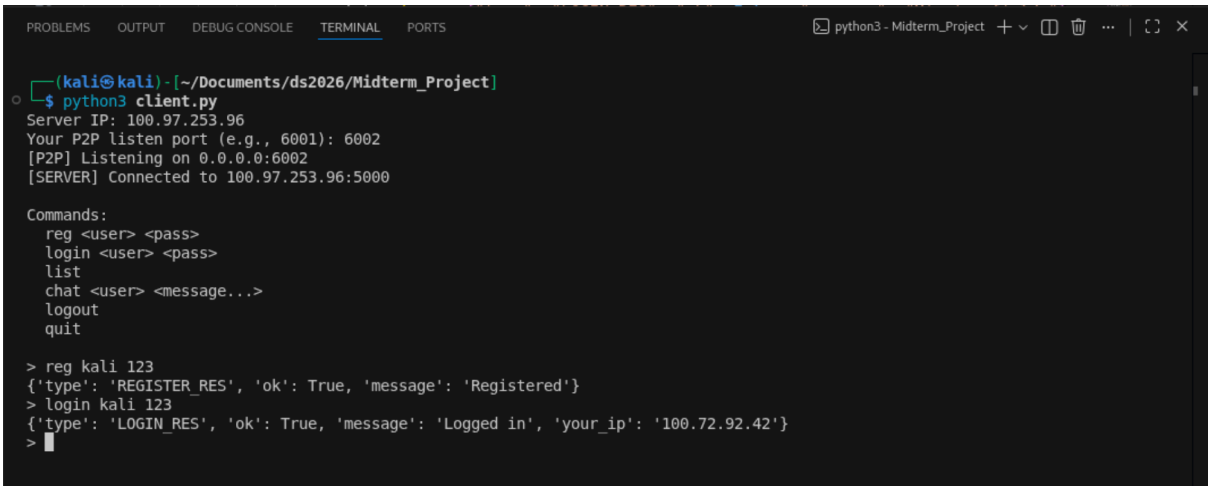
Kali virtual machine:



Step 3: Client registration and login

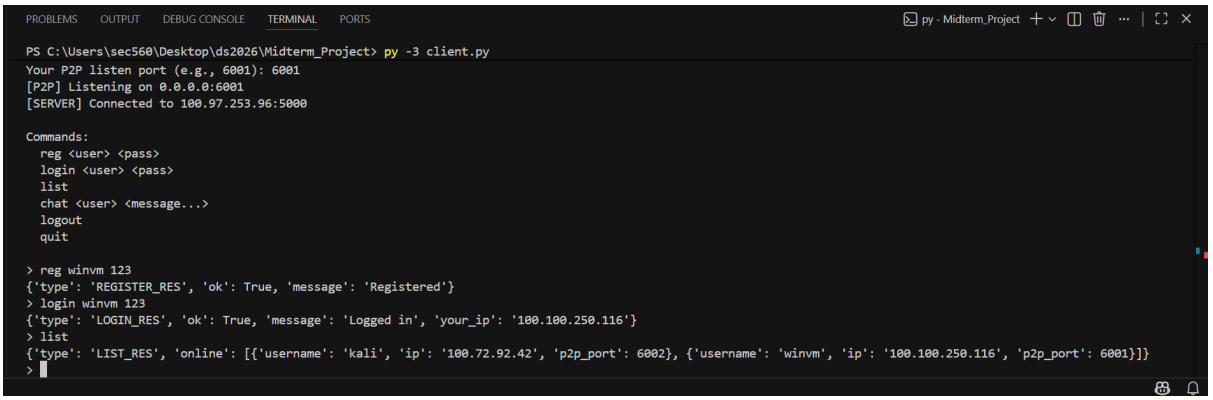
Window virtual machine:

Kali virtual machine:

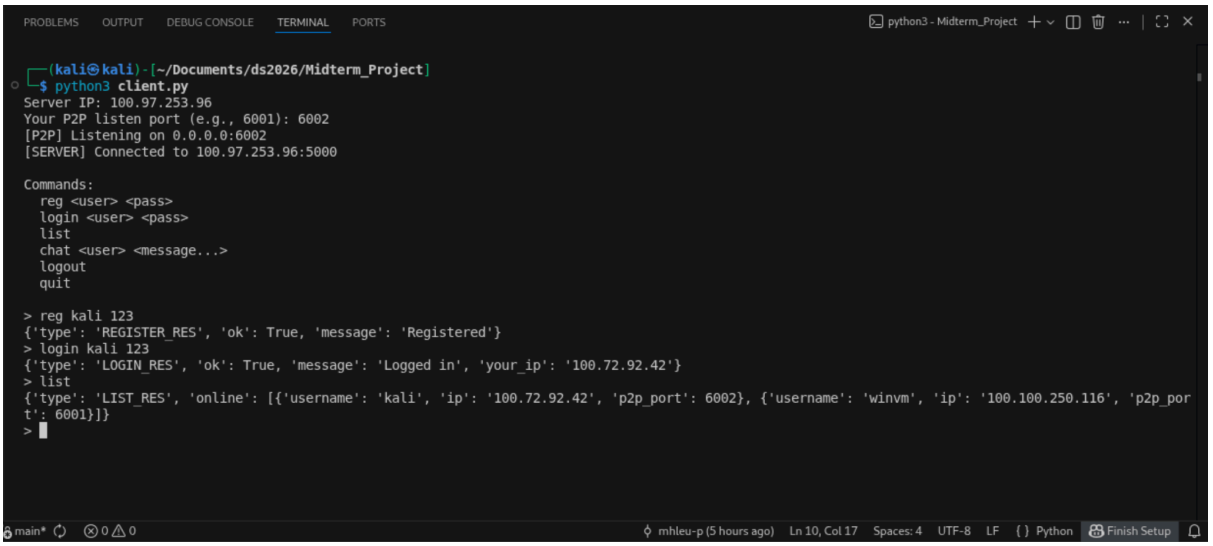


Step 4: Clients list display

Window virtual machine:



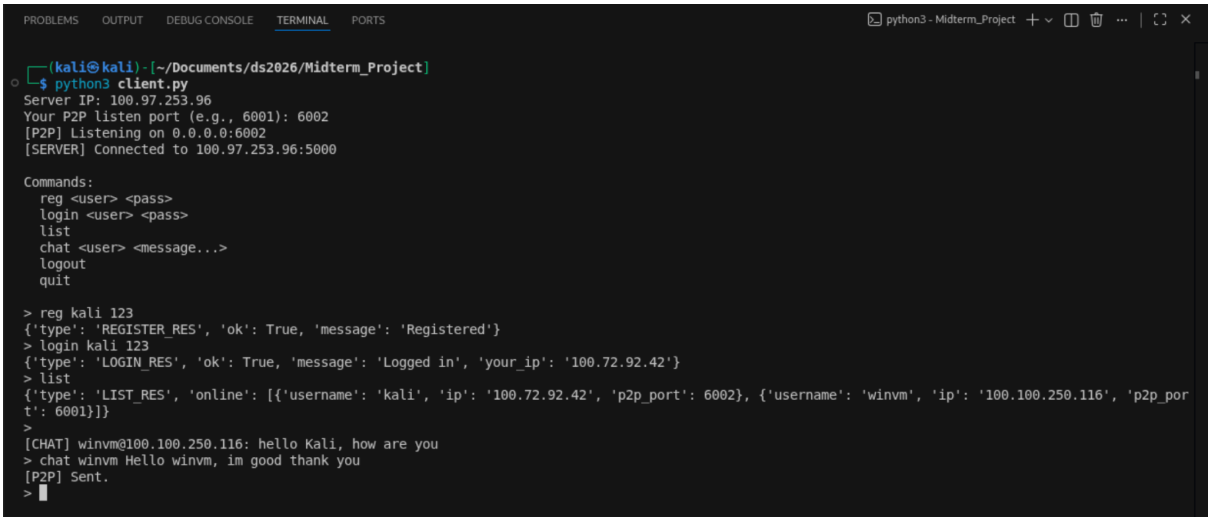
Kali virtual machine:



Step 5: Clients chat to each other

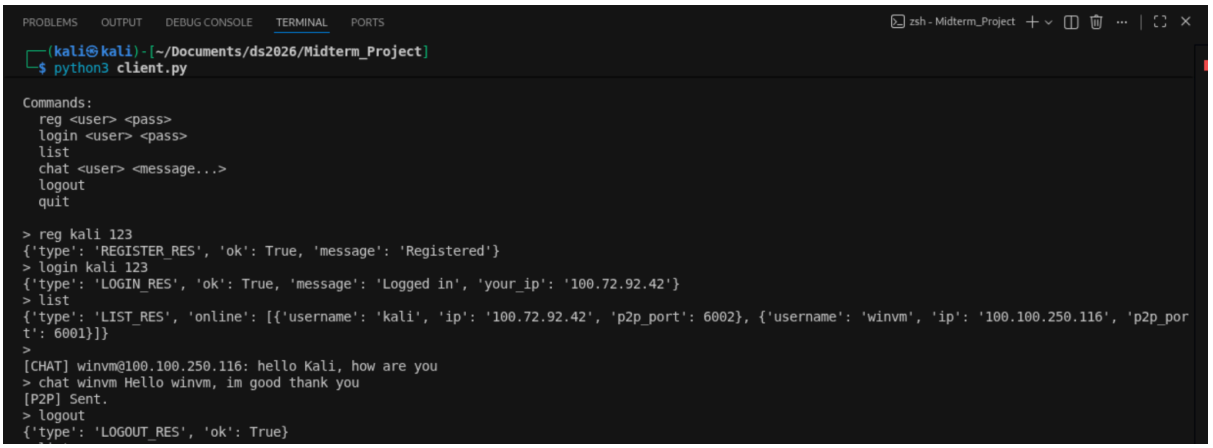
Window virtual machine:

Kali virtual machine:

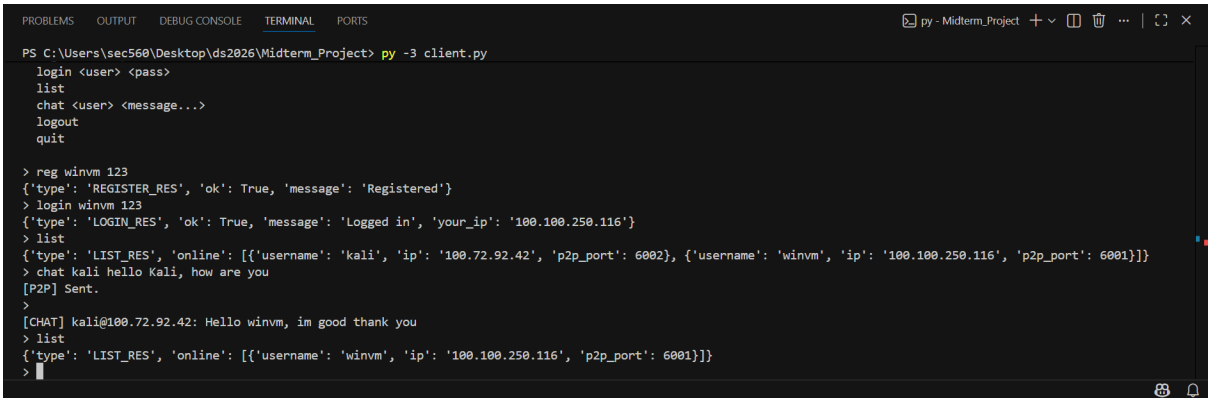


Step 6: Clients logout

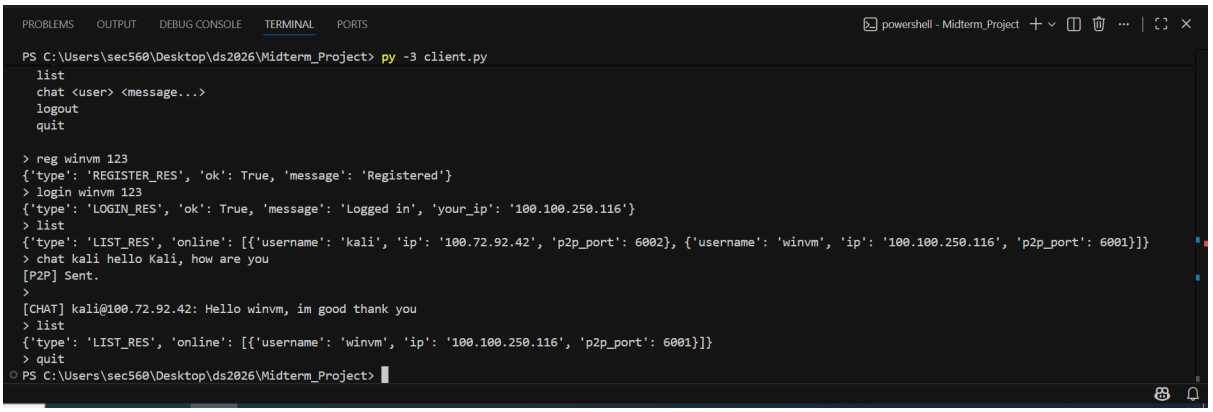
We logout in Kali machine



We listed user again in Win virtual machine



Step 7: Clients quit



VII. Contribution of Each Team Member

Member	Percentage of work
Nguyen Viet Hung	20%
Nguyen Thanh Dat	20%
Nguyen Hoang Long	20%
Nguyen Huu Phuong	20%
Pham Minh Hieu	20%

VIII. Conclusion

This project successfully demonstrates the design and implementation of a hybrid centralized and peer-to-peer chat system using socket programming.

By separating control functions (such as user management and peer discovery) from data communication, the system achieves improved scalability compared to a fully centralized architecture.
The central server maintains consistency by managing user presence information, while direct peer-to-peer connections allow efficient message delivery without unnecessary server involvement.

The system was deployed and tested across multiple computers within the same network, confirming its distributed nature and functional correctness.