PEERCONNECT: REAL-TIME P2P CHAT APPLICATION

A PROJECT REPORT

Submitted by

Registration No	Name
24MCS0001	Tanmay Borse
24MCS0062	Aditi Acharjya
24MCS0042	Arpit Madse

DATA STRUCTURES AND ALGORITHMS LAB

MCSE501L

DATE OF SUBMISSION:

14 November 2024

OBJECTIVE

- To develop a peer-to-peer (P2P) chat application that enables real-time communication between users.
- To designed and developed to build a peer-to-peer (P2P) chat application that allows users to communicate directly and in real time over a network without the need for a centralized server.
- To provide an efficient, simple yet functional system that allows users to connect, send, and receive messages using an efficient and secure structure.
- To apply essential concepts of data structures and algorithms to develop a robust messaging application that facilitates quick and reliable communication between peers maintaining effective user connectivity.

INTRODUCTION

Background:

In the digital age, real-time communication is a cornerstone of online interaction. It is a key aspect of modern applications, from social networking to customer support. Most traditional chat applications often rely on a centralized server that handles all incoming and outgoing messages between users. However, this centralized model has some limitations. It can introduce latency (i.e. delays due to server load), increase privacy concerns, as all messages pass through a third party. And, additionally increase dependency on the server's uptime creating a single point of failure where server downtime affects all users.

A peer-to-peer chat application addresses these issues by establishing a direct communication channel between users, which allows for lower latency, improving responsiveness and maintaining enhanced privacy by avoiding intermediaries, and decreased reliance on centralized infrastructure. This project aims to build a straightforward P2P chat application, showcasing the benefits of this architecture in handling direct messaging between peers.

Problem Statement:

The primary challenge this project addresses is designing an effective method for real-time communication between users, without relying on a centralized server. This approach reduces server costs, improves message latency, and enhances user privacy. In a traditional system, a server relays messages, but in a P2P network, each peer must connect directly with others. Implementing a real-time P2P messaging system requires careful management of network connections and data structures like message queues to ensure that messages are delivered accurately and in real time. Careful implementation of the project is important to handle incoming and outgoing messages, connection states, and peer discovery, ensuring that messages are reliably transmitted.

Addressing this problem is important for several reasons:

- **Privacy:** Messages do not pass through third-party servers, reducing potential privacy risks.

- **Reduced Costs:** Serverless design means no expenses associated with maintaining a central server.
- **Latency:** Direct connections reduce the time it takes for messages to reach their destination, improving user experience. This project implements a solution by developing an efficient P2P system that maintains direct connections between users and handles real-time message transmission.

METHODOLOGY

Chosen Data Structures:

- **Arrays:** Used for buffers to store received or sent data (TCHAR buffer[256], char buffer[1024]).
- **String:** Utilized for handling strings that need dynamic memory allocation and UTF-8 conversion (std::string sendBuffer).
- **Wide Character Arrays (wchar_t):** Used for Unicode string handling before conversion (wchar_t buffer[256]).
- **vector or list:** For managing a collection of connected clients or messages.
- **map or unordered_map:** To map client identifiers (such as IP addresses or custom IDs) to socket descriptors or client-specific data.
- **queue:** Useful for maintaining a queue of incoming messages or tasks to be processed by the server.
- **Buffers (char or TCHAR arrays):** Used for sending and receiving data in the send() and recv() functions.

Windows-Specific Structures:

- **HWND:** Used to represent handles to window elements like ListBox, InputEdit, and SendButton.
- **SOCKET**: Represents a socket descriptor for network communication (SOCKET server).
- **SOCKADDR_IN:** A structure for handling IP addresses and port information in the network connection (SOCKADDR_IN addr).
- **WSADATA:** Used by WSAStartup() to store information about the Windows Sockets implementation.

Algorithm:

The core algorithm for the P2P chat application focuses on message transmission between peers. A basic handshaking protocol is implemented to establish a reliable connection, followed by real-time message exchange.

- **Connection Establishment:** When a user wants to connect to a peer, the application initiates a TCP connection, exchanging necessary information (IP address, port) for direct communication. Once connected, each peer is added to the active peer list.
- **Message Transmission:** Messages are sent to peers over established connections using TCP to ensure data integrity and order. Incoming messages are placed in an incoming queue, while outgoing messages are placed in an outgoing queue. This queue-based approach helps manage multiple messages and ensures that each message is processed in order.
- **Message Reception:** A separate thread constantly listens for incoming messages. Upon receiving a message, it is added to the incoming queue, which is then displayed to the user in real time.

Development Environment:

- **Programming Language:** C++ was chosen for its simplicity in handling sockets and threading, both of which are critical for network-based applications. The built-in libraries such as socket and threading are well-suited for implementing network protocols and concurrent operations.
- **IDE:** Visual Studio Code, due to its ease of use and versatility. Its debugging and extension capabilities simplify development and troubleshooting in C++

Libraries/Tools:

- **Socket Library:** Used to establish and manage connections between peers, providing the foundation for network communication.
- **Threading Library:** Essential for handling simultaneous send and receive operations, allowing real-time message exchange without blocking other processes.

Relevance to Course of Study:

This project aligns well with the core concepts of data structures and algorithms, as it demonstrates:

- **Queue Management:** By implementing message queues, we apply the concept of FIFO (First In, First Out), which is critical for managing the order of real-time messages.
- **Dynamic Lists:** Managing a dynamic list of active peers demonstrates efficient use of linked lists for real-time applications.
- **Hash Maps:** Using hash maps for peer lookup shows practical application of hash tables for fast access to network resources.

IMPLEMENTATION DETAILS

Code Overview:

How server works:

- Initialization: The program starts by creating the GUI and starting a server thread.
- Server Socket: The server thread sets up the Winsock library, creates a socket, binds it to a port, and listens for incoming client connections.
- Connection Handling:
    - -When a client connects, accept() completes, and a thread is created to receive data.
    - -The serverSend() function sends data when the Send button is clicked.
- Receiving Messages: The serverReceive() thread waits for data from the client and updates the list box in the GUI with received messages.

How the Client Works:

- Starts the Winsock library and creates a socket.
- Connects to the server at 127.0.0.1 on port 5555.
- Starts a thread for receiving messages from the server.
- Provides a GUI where users can type messages and click "Send" to transmit them to the server.
- Displays incoming messages in a list box.
- Handles disconnection gracefully when the server stops responding.

To run the server.cpp and Client.cpp file

1. we have first g++ complier installed in your system
2. open cmd or terminal
3. Move to the folder where server.cpp and Client.cpp files are exist
4. to compile Server.cpp file type
    a. *g++ Server.cpp -o Server -lws2_32*
    b. enter then type *Server*
    c. enter to execute server file
5. Similarly for Client.cpp file Open new cmd or terminal to compile Client.cpp file type
    a. *g++ Client.cpp -o Client -lws2_32*
    b. enter then type *Client*

Code Snippets:

Server code:

1. GUI components like ListBox, InputBox, SendButton and sendMessage are created created with CreateWindowEx function.

```
case WM_CREATE:
        // Create a List Box
        ListBox = CreateWindowEx( WS_EX_CLIENTEDGE, _T("LISTBOX"), _T(""),
        WS_VISIBLE | WS_CHILD | WS_BORDER | LBS_NOTIFY,10, 10, 500, 200,hwnd, (HMENU)1, GetModuleHandle(NULL), NULL);

        // Create Input Field
        InputEdit = CreateWindowEx(WS_EX_CLIENTEDGE, _T("EDIT"), _T(""),
        WS_VISIBLE | WS_CHILD | WS_BORDER,10, 220, 300, 25,hwnd, (HMENU)1, GetModuleHandle(NULL), NULL);

        // Create Button
        SendButton = CreateWindow( _T("BUTTON"), _T("Send"),
        WS_VISIBLE | WS_CHILD | WS_BORDER | BS_DEFPUSHBUTTON, 320, 220, 80, 22, hwnd, (HMENU)2, GetModuleHandle(NULL), NULL);

        // Add items to the list box
        SendMessage(ListBox, LB_ADDSTRING, 0, (LPARAM)_T("I am Server!"));

    break;
```

2. OnButtonClick() Function:

   Retrieves the text from the input box. Checks if the input is not empty. Calls serverSend() to send the data to the connected client.

   Formats the message as "Me: [message]" and displays it in the list box. Clears the input box after sending.

```
void OnButtonClick() {

    TCHAR buffer[256];
    GetWindowText(InputEdit, buffer, sizeof(buffer) / sizeof(buffer[0]));

    if (_tcslen(buffer) != 0 || _tcscspn(buffer, _T("\t\r\n ")) != 0){
        serverSend(&clientSocket);
        TCHAR formattedBuffer[512];
        _stprintf(formattedBuffer, _T("Me: %s"), buffer);
        SendMessage(ListBox, LB_ADDSTRING, 0, (LPARAM)formattedBuffer);
        SetWindowText(InputEdit, _T(""));
    }
}
```

3. serverSend() Function

   Sends data from the server to the client. Converts the wide-character input from the GUI to a UTF-8 std::string using WideCharToMultiByte. Sends the data using the send() function.

   If the message is "exit", it logs the disconnection and returns.

```
DWORD WINAPI serverSend(LPVOID lpParam) {

    SOCKET client = *(SOCKET*)lpParam;
    wchar_t buffer[256] = { 0 };

    GetWindowTextW(InputEdit, buffer, sizeof(buffer) / sizeof(wchar_t));

    std::string sendBuffer;
    sendBuffer.resize((wcslen(buffer) + 1) * 4);
    int charsConverted = WideCharToMultiByte(CP_UTF8, 0, buffer, -1, &sendBuffer[0], sendBuffer.size(), NULL, NULL);

    if (charsConverted > 0) {
        if (send(client, sendBuffer.c_str(), charsConverted - 1, 0) == SOCKET_ERROR) {
            std::cerr << "send failed with error " << WSAGetLastError() << std::endl;
            return -1;
        }
    }
    else{
        std::cerr << "WideCharToMultiByte failed with error: " << GetLastError() << std::endl;
        return -1;
    }
    if(wcscmp(buffer, L"exit") == 0){
        std::cout << "Client disconnected (server-side exit)." << std::endl;
        return 1;
    }
    SetWindowTextW(InputEdit, L"");

    return 1;
}
```

4. serverReceive() Function

   Continuously receives data from the client using recv(). Checks for errors and connection closure. Displays received messages in the list box. Clears the buffer after each message.

```
DWORD WINAPI serverReceive(LPVOID lpParam) {

    TCHAR buffer[1024] = { 0 };
    SOCKET client = *(SOCKET*)lpParam;

    while (true) {
        int bytesReceived = recv(client, (char*)buffer, sizeof(buffer) - sizeof(TCHAR), 0);
        if (bytesReceived == SOCKET_ERROR) {
            std::cerr << "recv function failed with error " << WSAGetLastError() << std::endl;
            return -1;
        }
        if (bytesReceived == 0) {
            std::cout << "Client Disconnected." << std::endl;
            break;
        }

        buffer[bytesReceived / sizeof(TCHAR)] = '\0';
        SendMessage(ListBox, LB_ADDSTRING, 0, (LPARAM)buffer);
        memset(buffer, 0, sizeof(buffer));
    }

    return 1;
}
```

5. WinMain function:

Entry point for the Win32 application. Registers the window class using RegisterClassEx. Creates the main application window using CreateWindowEx. Starts the ServerThread to handle the socket operations in the background.

Enters the message loop (GetMessage, TranslateMessage, DispatchMessage).

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
    HWND hwnd;
    MSG msg;
    memset(&wc, 0, sizeof(wc));
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.lpfnWndProc = WndProc;
    wc.hInstance = hInstance;
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wc.lpszClassName = "WindowClass";
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
    if (!RegisterClassEx(&wc)) {
        MessageBox(NULL, "Window Registration Failed!", "Error!", MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }
    hwnd = CreateWindowEx(
        WS_EX_CLIENTEDGE, "WindowClass", "Caption",
        WS_VISIBLE | WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, 640, 480,
        NULL, NULL, hInstance, NULL
    );
    if (hwnd == NULL) {
        MessageBox(NULL, "Window Creation Failed!", "Error!", MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }

    CreateThread(NULL, 0, ServerThread, NULL, 0, NULL);

    while (GetMessage(&msg, NULL, 0, 0) > 0) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
```

Client code:

1. clientSend() Function

- **Message Retrieval**: The function retrieves the text entered by the user in InputEdit using GetWindowTextW().
- **UTF-8 Conversion**: The wide-character (Unicode) string from InputEdit is converted to a UTF-8 formatted string using WideCharToMultiByte():
  - CP_UTF8 specifies the UTF-8 code page for conversion.
  - The converted string is stored in a std::string buffer.
- **Sending Data**: The converted UTF-8 string is sent to the server using send(). If send() fails, it logs an error and returns. The charsConverted - 1 ensures the null terminator is not sent.
- **Exit Command**: If the message is "exit", the function returns a special code (1) to indicate termination.

```
DWORD WINAPI clientSend(LPVOID lpParam) {
    SOCKET server = *(SOCKET*)lpParam;
    wchar_t buffer[256] = { 0 };

    GetWindowTextW(InputEdit, buffer, sizeof(buffer) / sizeof(wchar_t));

    std::string sendBuffer;
    sendBuffer.resize((wcslen(buffer) + 1) * 4);
    int charsConverted = WideCharToMultiByte(CP_UTF8, 0, buffer, -1, &sendBuffer[0], sendBuffer.size(), NULL, NULL);

    if (charsConverted > 0) {
        if (send(server, sendBuffer.c_str(), charsConverted - 1, 0) == SOCKET_ERROR){
            std::cerr << "send failed with error " << WSAGetLastError() << std::endl;
            return -1;
        }
    }
    else{
        std::cerr << "WideCharToMultiByte failed with error: " << GetLastError() << std::endl;
        return -1;
    }

    if (wcscmp(buffer, L"exit") == 0) {
        std::cout << "Thank you for using the application" << std::endl;
        return 1;
    }
    SetWindowTextW(InputEdit, L"");
    return 1;
}
```

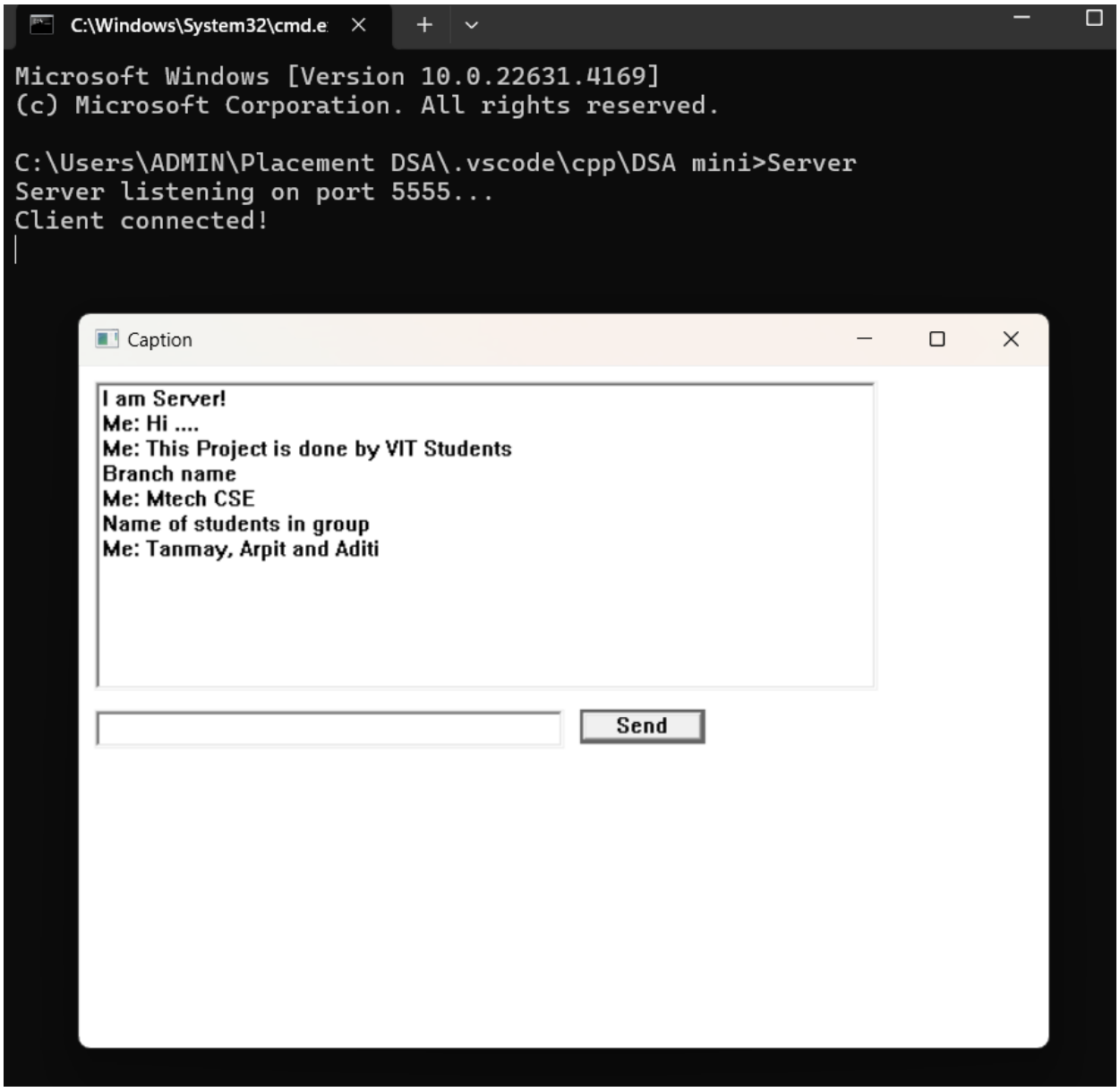
2. clientReceive **Function**

- **Data Reception**: Continuously listens for incoming data from the server using recv():
  - Reads data into buffer and handles the number of bytes received.
  - If recv() fails, logs the error and returns to stop the thread.
  - If bytesReceived is 0, it indicates that the server has disconnected, and the function breaks out of the loop.
- **Displaying Messages**: Null-terminates the received string and adds it to the ListBox using SendMessage().
- **Buffer Management**: Clears the buffer after processing to prepare for the next message.

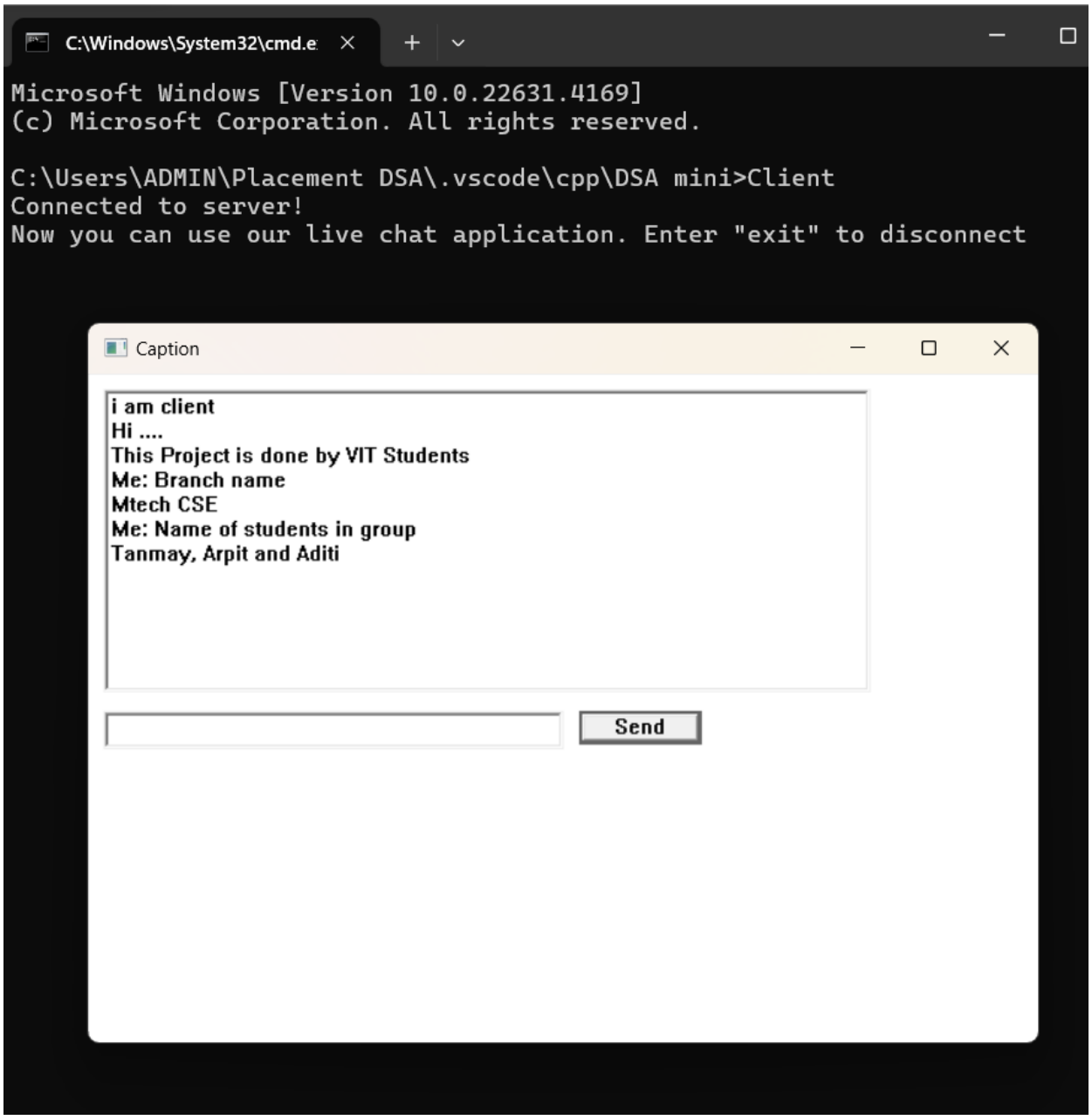
```
DWORD WINAPI clientReceive(LPVOID lpParam) {
    SOCKET server = *(SOCKET*)lpParam;
    TCHAR buffer[1024] = { 0 };

    while (true) {
        int bytesReceived = recv(server, (char*)buffer, sizeof(buffer) - sizeof(TCHAR), 0);
        if (bytesReceived == SOCKET_ERROR) {
            std::cerr << "recv function failed with error " << WSAGetLastError() << std::endl;
            return -1;
        }

        if (bytesReceived == 0) {
            std::cout << "Server Disconnected." << std::endl;
            break;
        }

        buffer[bytesReceived / sizeof(TCHAR)] = '\0';
        SendMessage(ListBox, LB_ADDSTRING, 0,  (LPARAM)buffer);
        memset(buffer, 0, sizeof(buffer));
    }

    return 1;
}
```

RESULTS AND TESTING

Server side screen:



Client side screen:



CONCLUSION

In this project, we successfully developed a peer-to-peer chat application that demonstrating the feasibility of real-time messaging. By utilizing data structures like queues, linked lists, and hash maps, we managed message flow, maintained a dynamic peer list, and facilitated efficient peer lookup.

The project demonstrated the benefits of a P2P network architecture, such as improved privacy, reduced server costs, and lower latency. Key achievements included effective use of data structures to manage message queues and connections and a streamlined message transmission algorithm.

The primary takeaway is that P2P communication is feasible and effective for real-time applications and can be implemented with minimal resources. This project also underscored the importance of choosing appropriate data structures and algorithms to handle real-time communication tasks, providing valuable hands-on experience with practical networking and data management concepts.