

## 作业要求

1. 给出你聊天协议的完整说明。
2. 利用C或C++语言，使用基本的Socket函数完成程序。不允许使用CSocket等封装后的类编写程序。
3. 使用流式套接字、采用多线程（或多进程）方式完成程序。
4. 程序应有基本的对话界面，但可以不是图形界面。程序应有正常的退出方式。
5. 完成的程序应能支持多人聊天，支持英文和中文聊天。
6. 编写的程序应该结构清晰，具有较好的可读性。
7. 在实验中观察是否有数据的丢失，提交源码和实验报告。

## 聊天协议

本次实验传输层协议采用 TCP 协议，数据以流式传输。

应用层设计的聊天室中服务器和客户端之间通过套接字的 send 和 recv 函数来发送和接收文本消息，采用多线程的方式将接收消息和发送消息分开执行，要求服务器端和客户端同时在线。客户端可以向服务器端发送消息并接受服务器端传来的消息，同样服务器端也可以向客户端发送消息并接受客户端传输的消息。对于消息来说，每条消息以换行符表示结束。

当成功接收到消息时，将显示接收到的消息，并添加时间戳到它们发送的消息中，以提供消息的时间信息，进行日志记录。如果某个客户端发送消息“exit”，则证明该客户端即将结束聊天，则发送这条消息的端口则会显示“You have disconnected”，服务器端收到该条消息后会显示该客户端exit。

## 代码分析

### 服务器端代码：

1. 初始化 Winsock 库：在 main 函数开始，通过调用 WSStartup 函数来初始化 Windows 套接字库。

```
WSADATA wsaData;
WSAStartup(MAKEWORD(2, 2), &wsaData);
if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2){
    cout << "Error in initializing " << WSAGetLastError();
    exit(EXIT_FAILURE);
}
```

2. 创建服务器套接字：使用 socket 函数创建一个服务器套接字，指定了地址族（AF\_INET，表示IPv4）、套接字类型（SOCK\_STREAM，表示TCP套接字）、和协议（IPPROTO\_TCP，表示TCP协议）。

```
server = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (server == INVALID_SOCKET){
    cout << "Error in creating socket " << WSAGetLastError();
    exit(EXIT_FAILURE);
}
```

3. 绑定服务器地址：服务器通过 bind 函数将套接字与服务器的 IP 地址和端口号绑定，这样客户端可以连接到服务器。

```

SOCKADDR_IN serveraddr;
SOCKADDR_IN clientaddrs[Max];
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(PORT);
if (inet_pton(AF_INET, "127.0.0.1", &(serveraddr.sin_addr)) != 1) {
    cout << "Error in inet_pton " << WSAGetLastError();
    exit(EXIT_FAILURE);
}
if (bind(server, (LPSOCKADDR)&serveraddr, sizeof(serveraddr)) == SOCKET_ERROR){
    cout << "Error in binding " << WSAGetLastError();
    exit(EXIT_FAILURE);
}

```

4. 启动监听：使用 listen 函数开始监听客户端的连接请求，通过 Max 参数指定最大的等待连接队列长度。

```

if (listen(server, Max) != 0){
    cout << "Error in listening " << WSAGetLastError();
    exit(EXIT_FAILURE);
}

```

5. 等待客户端连接：服务器进入一个无限循环，在此期间等待客户端连接请求。当有新的客户端请求连接时，服务器接受连接请求并创建一个新的套接字来处理客户端通信。这是一个多线程服务器，因此为每个连接创建一个新的线程。

```

while (1){
    if (connect_count < Max){
        int num = 0; //可用的客户端连接
        for (num; num < Max; num++){
            if (condition[num] == 0){
                break;
            }
        }
        int addrlen = sizeof(SOCKADDR_IN); //接收客户端的地址信息
        client[num] = accept(server, (sockaddr*)&clientaddrs[num], &addrlen); //客户端连接
        if (client[num] == SOCKET_ERROR){
            cout << "The client is failed " << WSAGetLastError();
            closesocket(server);
            WSACleanup();
            exit(EXIT_FAILURE);
        }
        condition[num] = 1; //表示已占用
        connect_count++;
        auto currentTime = chrono::system_clock::now();
        time_t timestamp = chrono::system_clock::to_time_t(currentTime);
        tm localTime;
        localtime_s(&localTime, &timestamp);
        char timeStr[50];
        strftime(timeStr, sizeof(timeStr), "%Y-%m-%d--%H:%M:%S", &localTime);

        cout << "The Client " << client[num] << " is connected" << endl;
        cout << timeStr << endl;
        cout << "connect_count: " << connect_count << endl;
    }
}

```

```

        HANDLE Thread = CreateThread(NULL, 0, Threadfun, reinterpret_cast<LPVOID>
(static_cast<intptr_t>(num)), 0, NULL); //创建新线程
        if (Thread == NULL){
            cout << "The thread is failed " << WSAGetLastError();
            exit(EXIT_FAILURE);
        }
        else{
            CloseHandle(Thread);
        }
    }
    else{//已达最大连接量
        cout << "Server is full!" << endl;
    }
}
}

```

6. 处理客户端消息：对于每个客户端，服务器创建一个新线程（由 CreateThread 创建），在该线程中处理客户端发送的消息。接收到的消息将被处理，包括添加时间戳和发送给其他已连接的客户端。

```

DWORD WINAPI Threadfun(LPVOID lpParameter){
    // 从传入的参数中获取客户端的索引
    int num = static_cast<int>(reinterpret_cast<intptr_t>(lpParameter)); //从参数lpParameter中提取一个整数值
    int rece = 0; // 用于存储接收到的字节数
    char Rece[BufSize]; // 接收缓冲区
    char Send[BufSize]; // 发送缓冲区
    while (1){
        // 接收来自客户端的消息
        rece = recv(client[num], Rece, sizeof(Rece), 0);
        if (rece > 0){
            // 获取当前时间戳
            auto currentTime = chrono::system_clock::now();
            time_t timestamp = chrono::system_clock::to_time_t(currentTime);
            tm localTime;
            localtime_s(&localTime, &timestamp);
            char timeStr[50];
            strftime(timeStr, sizeof(timeStr), "%Y-%m-%d--%H:%M:%S", &localTime);
            // 打印接收到的消息和时间戳
            cout << "Client " << client[num] << ": " << Rece << endl;
            cout << timeStr << endl;
            // 格式化要发送的消息
            sprintf_s(Send, sizeof(Send), "%d: %s \n%s ", client[num], Rece, timeStr);
            // 将消息发送给所有连接的客户端
            for (int i = 0; i < Max; i++){
                if (condition[i] == 1){
                    send(client[i], Send, sizeof(Send), 0);
                }
            }
        }
        else{
            if (WSAGetLastError() == 10054){ // 如果客户端主动关闭连接
                auto currentTime = chrono::system_clock::now();
                time_t timestamp = chrono::system_clock::to_time_t(currentTime);
                tm localTime;
            }
        }
    }
}

```

```

        localtime_s(&localTime, &timestamp);
        char timeStr[50];
        strftime(timeStr, sizeof(timeStr), "%Y-%m-d--%H:%M:%S", &localTime);

        cout << "Client " << client[num] << " exit" << endl;
        cout << timeStr << endl;

        // 关闭客户端套接字，减少连接数
        closesocket(client[num]);
        connect_count--;
        condition[num] = 0;
        cout << "connect_count: " << connect_count << endl;
        return 0;
    }
    else{ // 接收失败
        cout << "Error in receiving " << WSAGetLastError();
        break;
    }
}
}
}

```

### 客户端代码：

1. 初始化 Winsock 库：与服务器端一样，首先初始化 Winsock 库。

```

WSADATA wsaData;
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0){
    cout << "Error in initializing " << WSAGetLastError();
    cout << endl;
    exit(EXIT_FAILURE);
}

```

2. 创建客户端套接字：使用 socket 函数创建一个客户端套接字，也是指定地址族（AF\_INET，IPv4）、套接字类型（SOCK\_STREAM，TCP套接字）和协议（IPPROTO\_TCP，TCP协议）。

```

client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (client == INVALID_SOCKET){
    cout << "Error in creating socket " << WSAGetLastError();
    exit(EXIT_FAILURE);
}

```

3. 设置服务器地址：客户端设置服务器的地址，包括服务器的 IP 地址和端口号，这用于连接到服务器。

```

SOCKADDR_IN servAddr; // 服务器地址
servAddr.sin_family = AF_INET; // 地址类型
servAddr.sin_port = htons(PORT); // 服务器端口号
if (inet_pton(AF_INET, "127.0.0.1", &(servAddr.sin_addr)) != 1){
    cout << "Error in inet_pton " << WSAGetLastError();
    exit(EXIT_FAILURE);
}

```

4. 连接到服务器：使用 connect 函数发起与服务器的连接请求，建立与服务器的连接。

```
if (connect(client, (SOCKADDR*)&servAddr, sizeof(SOCKADDR)) == SOCKET_ERROR){  
    cout << "Error in connecting " << WSAGetLastError();  
    exit(EXIT_FAILURE);  
}
```

5. 创建接收消息线程：客户端通过 CreateThread 函数创建一个独立线程，用于从服务器接收消息。

```
CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)receive, NULL, 0, 0);
```

6. 主循环和消息发送：在主循环中，客户端等待用户在命令行中输入消息。输入的消息将通过 send 函数发送给服务器。如果用户输入 "exit"，则客户端关闭连接并退出。

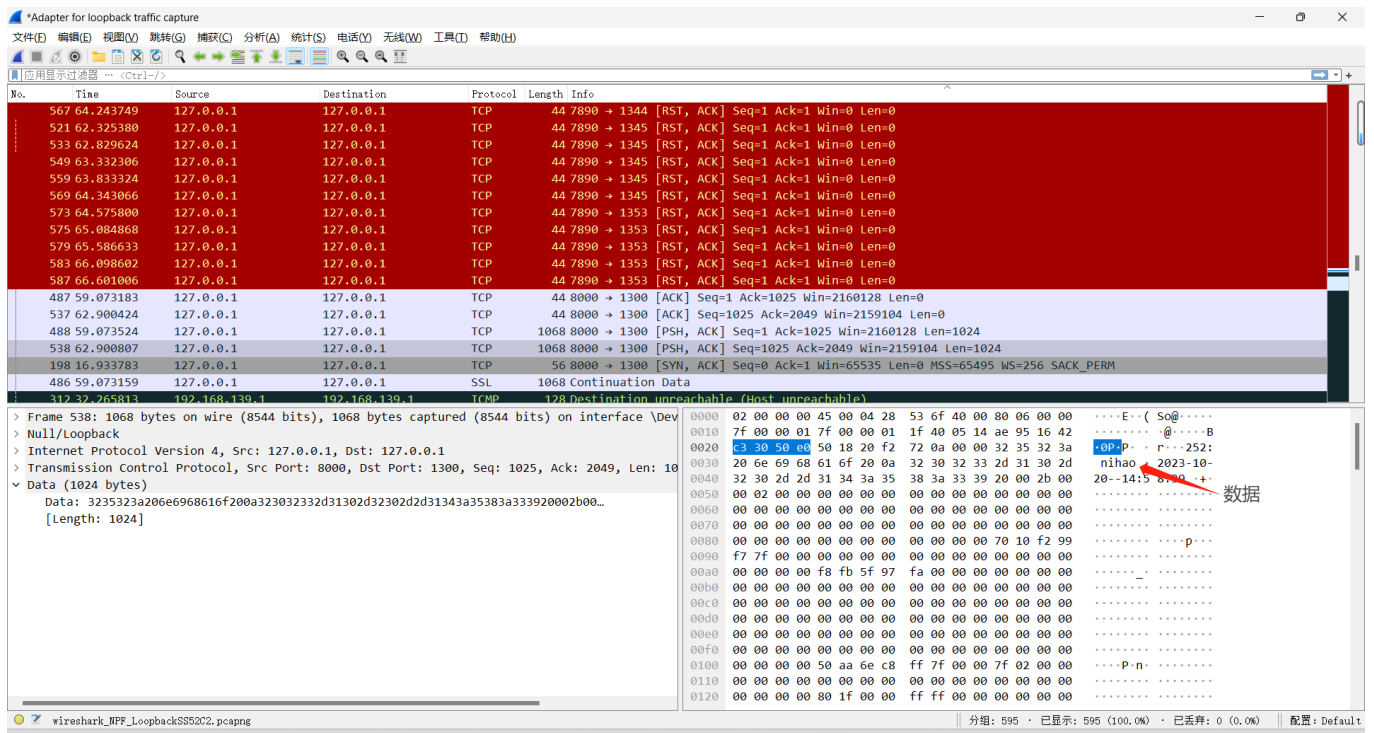
```
while (1){  
    cin.getline(message, sizeof(message));  
    if (strcmp(message, "exit") == 0){ // 输入 "exit" 退出  
        break;  
    }  
    send(client, message, sizeof(message), 0); // 发送消息  
}
```

7. 接收消息：在独立的接收消息线程中，客户端通过 recv 函数接收从服务器发送的消息，并将其显示在命令行界面上。

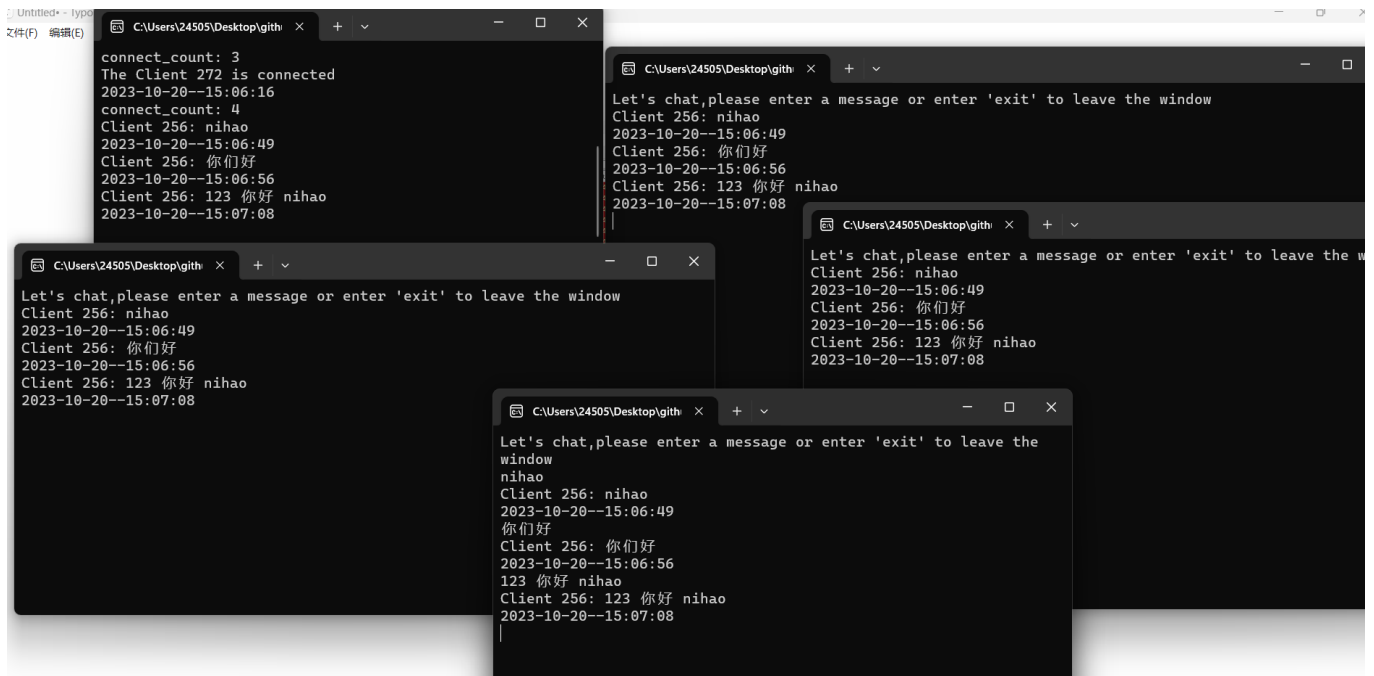
```
DWORD WINAPI receive(){  
    while (1){  
        char buffer[BufSize] = {}; // 接收数据缓冲区  
        int receivebuf = recv(client, buffer, sizeof(buffer), 0); // 接收到的字节数  
        if (receivebuf > 0){  
            cout << "Client " << buffer << endl;  
        }  
        else{  
            cout << "You have disconnected" << endl;  
            break;  
        }  
    }  
    return 0;  
}
```

## 运行情况

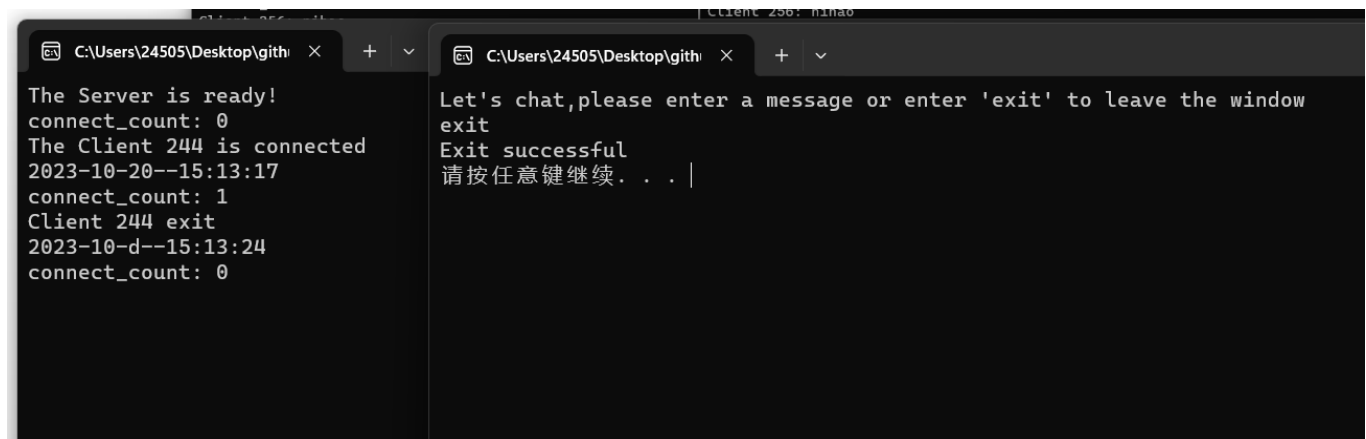
通过抓包分析，找到了完整的数据，截图如下：



聊天室运行截图如下：



离开聊天程序截图如下：



The image shows two terminal windows side-by-side. The left window displays the server's output, and the right window displays the client's input and output.

```
C:\Users\24505\Desktop\githi x + v
The Server is ready!
connect_count: 0
The Client 244 is connected
2023-10-20--15:13:17
connect_count: 1
Client 244 exit
2023-10-d--15:13:24
connect_count: 0

C:\Users\24505\Desktop\githi x + v
Let's chat,please enter a message or enter 'exit' to leave the window
exit
Exit successful
请按任意键继续. . . |
```

## 问题与分析

起初代码中存在问题时没有设置输出错误的代码，只能慢慢debug，于是加入了WSAGetLastError()的代码行，可以直观地观察到问题的出现位置。

服务器端代码中可能存在一些数据结构在两个客户端同时连接时会产生竞争的情况，目前想到的解决办法是创建一些锁来进行管理，但代码还未改进。