计算机网络实验报告

Lab1 SOCKET编程

网络空间安全学院 物联网工程专业 2212039 田晋宇

jassary08/Computer_Network (github.com)

实验要求

（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Socket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。

协议设计

一、协议原理

1. 套接字通信：

- 服务器和客户端通过流式套接字（TCP）进行通信，确保消息的可靠传输和顺序传递。TCP 提供了面向连接的、可靠的全双工通信，适合实时聊天应用。
- 服务器监听一个固定端口（8000），等待客户端连接。每当有新客户端连接时，服务器为其分配一个独立的线程来处理该客户端的通信。

2. 多线程处理：

- 服务器为每个客户端启动一个独立的线程，用于接收客户端发送的消息，并将消息广播给其他客户端。
- 服务器维护一个 `client_usernames` 映射，跟踪每个客户端的套接字和用户名，只向已经设置了用户名的客户端广播消息。

3. 消息格式：

- 每条消息由时间戳、用户名和消息正文组成。
- 系统消息与普通消息的区别在于，系统消息由服务器生成，用于通知所有用户某个客户端加入或离开聊天室。

二、协议组成要素

1. 消息格式：

- 消息格式为 `"[时间戳] 用户名: 消息内容"`。
- 系统消息格式为 `"[时间戳] 系统消息: 用户名 已进入聊天室，欢迎！"` 或 `"[时间戳] 系统消息: 用户名 已离开聊天室"`。
- 消息中的 `时间戳` 格式为 `YYYY-MM-DD HH:MM:SS`，表示消息发送的精确时间。

2. 消息内容：

- **普通消息**：客户端发送的带有用户名的文本消息，服务器接收到后，会广播给其他客户端。
- **系统消息**：服务器在以下情况下生成系统消息并广播：
  - 新用户加入聊天室时，服务器广播一条欢迎消息，通知所有用户该用户加入。
  - 用户离开聊天室时，服务器广播该用户离开的消息。

三、实现方法

根据实验要求我设计了如下的协议实现方案：

使用TCP传输协议，选用流式套接字，采用多线程方式 分别设计两个程序，一个作为服务器端（server），另一个作为客户端（client）。使用时需首先启动Server.exe，再启动若干个lient.exe，若干个client借助server完成通信。每一个功能实现过程中都需要包含错误检查和异常处理机制，一旦发生错误，打印日志以便调试。

服务器端：

1. 在服务器端使用 `socket()` 创建套接字，指定 `AF_INET`（IPv4）、`SOCK_STREAM`（流式套接字），并设置 `SO_REUSEADDR` 选项，确保套接字端口在重启服务器时能够立即重新绑定。
2. 使用 `bind()` 将服务器套接字绑定到指定的IP地址（通常是 `INADDR_ANY`）和端口号（例如 8000），使服务器在该端口上监听。
3. 使用 `listen()` 函数指定服务器最多可以接受的客户端连接数，`MaxClients` 表示最大连接数。到达该限制后，服务器拒绝新的连接请求。
4. 在一个主循环中，使用 `accept()` 函数接受来自客户端的连接请求。当有新的客户端连接时，服务器会为其创建一个新的线程来处理客户端消息，并为该客户端分配一个独立的 `socket`。每个客户端连接后，服务器为该客户端启动一个新的线程。
5. 线程函数使用 `recv()` 函数接收客户端发送的消息，并根据约定的消息格式（带有时间戳和客户端标识符）进行解析。广播时使用 `send()` 函数发送消息，遍历客户端 `socket` 列表，发送消息给所有其他连接的客户端。
6. 当客户端通过 `exit` 命令请求退出时，服务器接收 `recv()` 函数返回 `0`，表示客户端关闭连接。

客户端：

1. 在客户端使用 `socket()` 函数创建套接字，指定 `AF_INET`（IPv4）和 `SOCK_STREAM`（流式套接字），客户端将通过这个套接字连接到服务器。
2. 使用 `connect()` 函数将客户端连接到服务器端的 IP 地址（通常为 `127.0.0.1` 或本地服务器的 IP 地址）和端口号（例如 8000）。
3. 客户端发送消息：客户端在连接成功后，允许用户输入消息并通过 `send()` 函数将消息发送到服务器。
4. 客户端接收消息：客户端启动一个独立的线程用于接收服务器广播的消息。该线程使用 `recv()` 函数从服务器接收消息，并将其显示在控制台上。
5. 客户端可以通过输入 `exit` 命令退出聊天程序。当输入 `exit` 时，客户端通过 `close()` 或 `closesocket()` 函数主动关闭与服务器的连接。

Socket编程实现

一、服务器端

这部分实现一个基于 TCP 和多线程的简易聊天室服务器程序。它允许多个客户端连接服务器、发送消息，并将消息广播给其他连接的客户端。服务器使用 WinSock 库进行网络通信。

- **全局变量和常量**

```
// 全局变量与常量定义
#define PORT 8000
#define MAX_CLIENTS 10
#define BUF_SIZE 1024

vector<SOCKET> client_sockets;  // 存储客户端套接字
mutex client_mutex;  // 保护 client_sockets 的互斥锁
map<SOCKET, string> client_usernames;  // 用于存储套接字与用户名的映射
int user_count = 0;
```

  - `PORT`：服务器监听的端口号，设置为 `8000`。
  - `MAX_CLIENTS`：最大允许的客户端连接数，设置为 `10`。
  - `BUF_SIZE`：消息缓冲区大小，设置为 `1024` 字节。
  - `client_sockets`：用于存储每个连接客户端的 `SOCKET`（套接字），方便服务器管理多个客户端连接。
  - `client_mutex`：用于在多线程环境中保护共享资源 `client_sockets` 和 `client_usernames`，防止多个线程同时访问这些资源导致竞态条件。
  - `client_usernames`：使用 `map<SOCKET, string>` 存储每个客户端的套接字和对应的用户名，确保每个客户端都有一个唯一的标识。
  - `user_count`：用于统计当前在线的用户数量。

- **获取当前时间戳**

```
string get_current_time() {
    time_t now = time(0);
    struct tm time_info;
    localtime_s(&time_info, &now);  // 使用 localtime_s 替换 localtime
    char time_buffer[50];
    strftime(time_buffer, sizeof(time_buffer), "%Y-%m-%d %H:%M:%S",
&time_info);
    return string(time_buffer);
}
```

  - 获取当前时间，并将其格式化为字符串，格式为 `YYYY-MM-DD HH:MM:SS`。
  - 这个时间戳会添加到每条消息中，表示消息发送的时间。

- **广播机制**

```
void broadcast_message(const string& message, SOCKET sender_socket =
INVALID_SOCKET) {
    lock_guard<mutex> lock(client_mutex);  // 自动加锁与解锁
    for (const auto& pair : client_usernames) {  // 遍历客户端套接字和用户名的映
射
        SOCKET sock = pair.first;  // 获取客户端的套接字
        const string& username = pair.second;  // 获取用户名

        if (!username.empty()) {  // 只有设置了用户名的客户端才广播消息
```

```
            if (send(sock, message.c_str(), message.size(), 0) ==
SOCKET_ERROR) {
                cout << "----------------------------------------" <<
endl;
                cerr << "Failed to send message to client: " <<
WSAGetLastError() << endl;
                cout << "----------------------------------------" <<
endl;
            }
        }
    }
}
```

- 将消息广播给所有连接的客户端，但不会广播给发送消息的客户端（通过 `sender_socket` 排除）。
- 遍历 `client_usernames`，为每个设置了用户名的客户端发送消息。
- 使用 `send()` 函数发送消息，并进行错误处理。

- **线程函数**

这是一个线程函数，用于处理每个客户端的通信，包括接收用户名、广播消息和管理客户端断开连接的逻辑。

```
void handle_client(SOCKET client_socket) {
    char buffer[BUF_SIZE];
    string message;
    int bytes_received;

    // 从客户端接收用户名
    char name_buffer[BUF_SIZE] = { 0 };
    int name_len = recv(client_socket, name_buffer, BUF_SIZE, 0);
    if (name_len > 0) {
        user_count++;
        string username(name_buffer, name_len);
        cout << "----------------------------------------" << endl;
        cout << "客户端已加入列表,用户名称: " << username << endl;
        cout << "当前在线人数: " << user_count <<endl;
        cout << "----------------------------------------" << endl;

        // 将用户名存储在 map 中
        {
            lock_guard<mutex> lock(client_mutex);
            client_usernames[client_socket] = username;  // 将用户名存入映射中
        }
        // 广播欢迎消息
        string timestamp = get_current_time();
        string welcome_message = "[" + timestamp + "] 系统消息: " + username +
" 已进入聊天室，欢迎！当前在线人数: " + std::to_string(user_count);
        broadcast_message(welcome_message, client_socket);

        // 接收和处理消息的循环
        while (true) {
            memset(buffer, 0, BUF_SIZE);
            bytes_received = recv(client_socket, buffer, BUF_SIZE, 0);

            if (bytes_received > 0) {
```

```
                string received_message(buffer, bytes_received);

                message = received_message;

                cout << "----------------------------------------" <<
endl;
                cout << "Received message: " << message << endl;
                cout << "----------------------------------------" <<
endl;

                // 广播消息给其他客户端
                broadcast_message(message, client_socket);
            }
            else {
                // 客户端断开连接
                cout << "----------------------------------------" <<
endl;
                cout << "用户 " << username << " 断开连接: " << client_socket
<< endl;
                cout << "----------------------------------------" <<
endl;
                // 广播客户端离开消息
                string timestamp = get_current_time();
                string leave_message = "[" + timestamp + "] " + "系统消息: " +
username + " 已离开聊天室。";
                broadcast_message(leave_message, client_socket);
                user_count--;
                break;
            }
        }

        // 移除客户端并关闭连接
        closesocket(client_socket);

        lock_guard<mutex> lock(client_mutex);
        client_sockets.erase(remove(client_sockets.begin(),
client_sockets.end(), client_socket), client_sockets.end());
    }
    else {
        cerr << "Failed to receive username." << endl;
        closesocket(client_socket);
    }
}
```

- 服务器首先接收客户端发来的用户名，将其存入 `client_usernames` 中，并广播欢迎消息通知其他客户端。
- 服务器不断接收客户端发送的消息并广播给其他客户端。如果客户端断开连接，服务器会处理客户端的离开，并广播通知其他客户端。
- 当客户端断开连接时，服务器会广播该客户端离开聊天室的消息，并将其从 `client_sockets` 列表中移除。

- **主函数**

服务器的主函数，负责初始化网络、监听客户端连接、创建线程处理每个客户端。

```
int main() {
```

```
    WSADATA wsaData;

    // 初始化 WinSock 库
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        cerr << "WSAStartup failed: " << WSAGetLastError() << endl;
        return 1;
    }
    cout << "----------------------------------------" << endl;
    cout << "Socket DLL 初始化成功" << endl;
    cout << "----------------------------------------" << endl;

    // 创建服务器套接字
    SOCKET server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == INVALID_SOCKET) {
        cerr << "Socket creation failed: " << WSAGetLastError() << endl;
        WSACleanup();
        return 1;
    }
    cout << "----------------------------------------" << endl;
    cout << "Socket 创建成功" << endl;
    cout << "----------------------------------------" << endl;

    // 配置服务器地址
    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;  // 绑定到本地所有IP
    server_addr.sin_port = htons(PORT);  // 绑定到端口8000

    // 绑定套接字到指定的IP和端口
    if (bind(server_socket, (struct sockaddr*)&server_addr,
sizeof(server_addr)) == SOCKET_ERROR) {
        cerr << "Bind failed: " << WSAGetLastError() << endl;
        closesocket(server_socket);
        WSACleanup();
        return 1;
    }
    cout << "----------------------------------------" << endl;
    cout << "端口号已经绑定成功,端口号为8000" << endl;
    cout << "----------------------------------------" << endl;

    // 监听客户端连接
    if (listen(server_socket, MAX_CLIENTS) == SOCKET_ERROR) {
        cerr << "Listen failed: " << WSAGetLastError() << endl;
        closesocket(server_socket);
        WSACleanup();
        return 1;
    }
    cout << "----------------------------------------" << endl;
    cout << "开始监听，等待客户端连接中..." << endl;
    cout << "----------------------------------------" << endl;

    // 主循环：接受客户端连接
    struct sockaddr_in client_addr;
    int client_len = sizeof(client_addr);
    while (true) {
```

```
        SOCKET client_socket = accept(server_socket, (struct
sockaddr*)&client_addr, &client_len);
        if (client_socket == INVALID_SOCKET) {
            cerr << "Accept failed: " << WSAGetLastError() << endl;
            continue;
        }
        char client_ip[INET_ADDRSTRLEN];  // 用于存储客户端IP地址
        inet_ntop(AF_INET, &client_addr.sin_addr, client_ip,
INET_ADDRSTRLEN);  // 使用 inet_ntop 替代 inet_ntoa

        // 获取当前时间戳
        string timestamp = get_current_time();

        cout << "----------------------------------------" << endl;
        cout << "新连接来自: " << client_ip << ":" <<
ntohs(client_addr.sin_port) << " at " << timestamp << endl;
        cout << "----------------------------------------" << endl;

        // 将新连接的客户端加入列表
        {
            lock_guard<mutex> lock(client_mutex);
            if (client_sockets.size() < MAX_CLIENTS) {
                client_sockets.push_back(client_socket);
            }
            else {
                cout << "----------------------------------------" <<
endl;
                cout << "达到最大客户端数量，拒绝新连接。" << endl;
                cout << "----------------------------------------" <<
endl;
                closesocket(client_socket);
                continue;
            }
        }

        // 创建线程处理客户端
        thread client_thread(handle_client, client_socket);
        client_thread.detach();  // 分离线程以便它可以独立运行
    }

    // 关闭服务器套接字
    closesocket(server_socket);
    WSACleanup();
    cout << "----------------------------------------" << endl;
    cout << "服务器已关闭" << endl;
    cout << "----------------------------------------" << endl;
    return 0;
}
```

- **初始化 WinSock**：使用 `WSAStartup()` 函数初始化 WinSock 库，为使用网络套接字做准备。
- **创建服务器套接字**：使用 `socket()` 创建一个 TCP 套接字。套接字将绑定到服务器的 IP 地址和端口号（8000）。

- **绑定端口**：使用 `bind()` 函数将套接字绑定到指定的 IP 地址和端口号。成功后，服务器可以在该端口上监听客户端连接。
- **开始监听**：使用 `listen()` 函数让服务器进入监听状态，等待客户端连接。允许最大连接数为 `MAX_CLIENTS`。
- **接受客户端连接**：在一个无限循环中，服务器通过 `accept()` 接受客户端的连接。当有客户端连接时，服务器为其创建一个新线程，调用 `handle_client()` 处理客户端的消息和通信。
- **线程管理**：每个客户端连接后，服务器会启动一个独立线程，用于处理该客户端的消息接收和发送。线程使用 `detach()` 分离，以便与主线程并行运行。
- **服务器关闭**：当主循环结束时，服务器关闭监听套接字，并清理 WinSock 资源。

二、客户端

这部分实现一个简易的客户端程序，使用 TCP 协议连接到服务器，并通过多线程方式接收和发送消息。

- **接收消息线程函数**

  这是一个运行在独立线程中的函数，负责从服务器接收广播的消息，并将其显示在客户端的控制台上。

```
void receive_messages(SOCKET client_socket) {
    char buffer[BUF_SIZE];

    while (running) {
        memset(buffer, 0, BUF_SIZE);
        int bytes_received = recv(client_socket, buffer, BUF_SIZE, 0);
        if (bytes_received > 0) {
            cout << buffer << endl;  // 输出服务器广播的消息
        }
        else if (bytes_received == 0) {
            cout << "Disconnected from server." << endl;
            break;
        }
        else {
            cerr << "Error receiving message: " << WSAGetLastError() <<
endl;
            break;
        }
    }

    running = false;  // 当服务器断开连接，停止客户端
    closesocket(client_socket);
}
```

  - 使用 `recv()` 函数从服务器端接收数据，并将其存储到 `buffer` 中。
  - 检查接收到的数据：
    - 如果接收到的数据长度 `>0`，将其打印到控制台。
    - 如果 `recv()` 返回 `0`，表示服务器断开连接，打印断开信息并退出循环。
    - 如果 `recv()` 返回 `<0`，表示接收时发生错误，打印错误信息并退出循环。
  - 当服务器断开连接或发生错误时，设置全局变量 `running = false` 来停止客户端，并关闭套接字。

- **主函数**

客户端的主函数，负责初始化网络连接、与服务器通信、处理用户输入，并启动接收消息的线程。

1.初始化 WinSock 库：

```
WSAStartup(MAKEWORD(2, 2), &wsaData);
```

- 使用 `WSAStartup()` 初始化 Windows 的网络库（WinSock）。这一步是使用套接字通信的必要前提。
- 如果初始化失败，输出错误并终止程序。

2. 创建客户端套接字：

```
SOCKET client_socket = socket(AF_INET, SOCK_STREAM, 0);
```

- 使用 `socket()` 函数创建 TCP 套接字。指定 `AF_INET` 表示使用 IPv4，`SOCK_STREAM` 表示使用流式套接字（即 TCP 协议）。
- 如果套接字创建失败，输出错误并终止程序。

3. 配置服务器地址：

```
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr);  // 连接到本地服务器
server_addr.sin_port = htons(PORT);
```

- 设置服务器的 IP 地址和端口号。这里使用 `127.0.0.1`，即本地主机地址。
- `htons()` 用于将端口号转换为网络字节序。

4. 连接到服务器：

```
connect(client_socket, (struct sockaddr*)&server_addr, sizeof(server_addr));
```

- 使用 `connect()` 连接到服务器。如果连接失败，输出错误并关闭套接字。

5. 输入用户名并发送到服务器：

```
cout << "请输入你的用户名: ";
getline(cin, username);
send(client_socket, username.c_str(), username.size(), 0);
```

- 提示用户输入用户名，并将用户名通过 `send()` 函数发送给服务器。

6. 显示欢迎界面：

```
cpp复制代码cout << "****************************************" << endl;
cout << "*            南开大学聊天室                *" << endl;
cout << "****************************************" << endl;
```

- 在成功连接到服务器并发送用户名后，显示一个简单的欢迎界面。

7. 启动接收消息的线程：

```
thread receive_thread(receive_messages, client_socket);
receive_thread.detach();
```

- 启动一个新线程 `receive_thread`，运行 `receive_messages` 函数，负责接收服务器广播的消息。
- 使用 `detach()` 将线程分离，以便主线程和接收线程可以并行工作。

8. 主循环：处理用户输入并发送消息：

```
while (running) {
    string timestamp = get_current_time();
    getline(cin, message);

    if (message == "exit") {
        running = false;
        break;
    }

    string full_message = "[" + timestamp + "] " + username + ": " +
message;
    send(client_socket, full_message.c_str(), full_message.size(), 0);
}
```

- 主线程循环等待用户输入消息。
- 每条消息会加上当前的时间戳和用户名，形成完整的消息格式 `[时间戳] 用户名: 消息内容`。
- 使用 `send()` 将消息发送到服务器。
- 如果用户输入 `exit`，则停止主循环，并退出程序。

9. 清除控制台输入：

```
cout << "\033[A\033[2K";
```

- 使用 ANSI 转义序列清除刚刚输入的行。
- `\033[A` 将光标移动到上一行，`\033[2K` 清除整行，避免用户输入的消息残留在屏幕上。

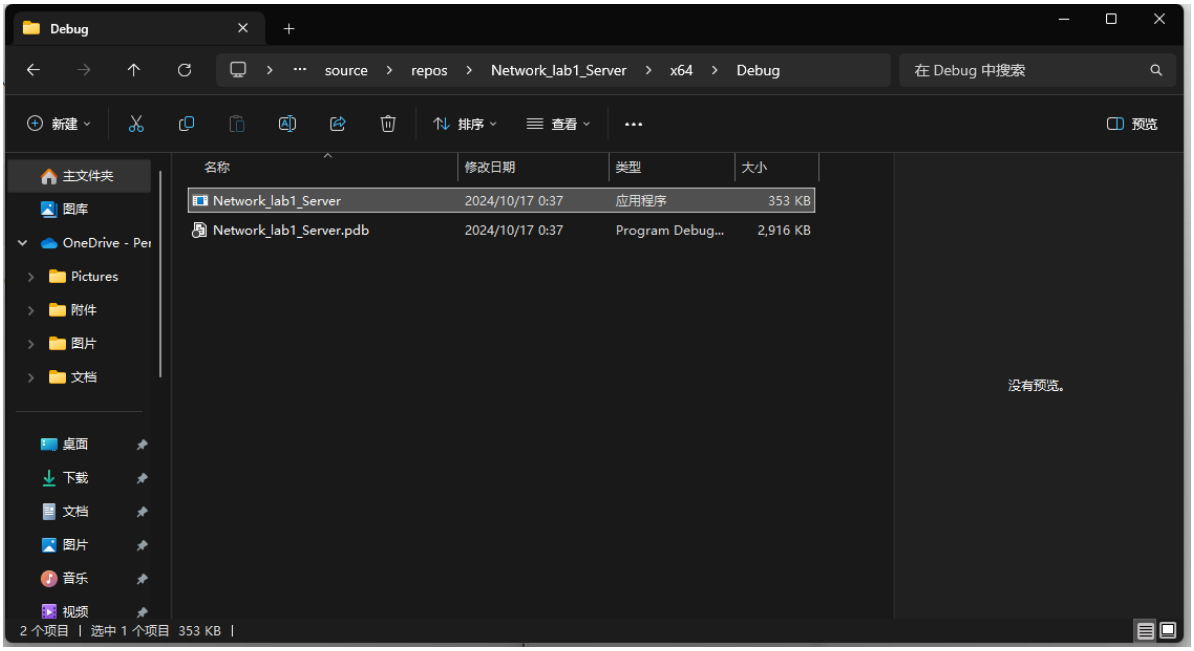
10. 关闭客户端连接：

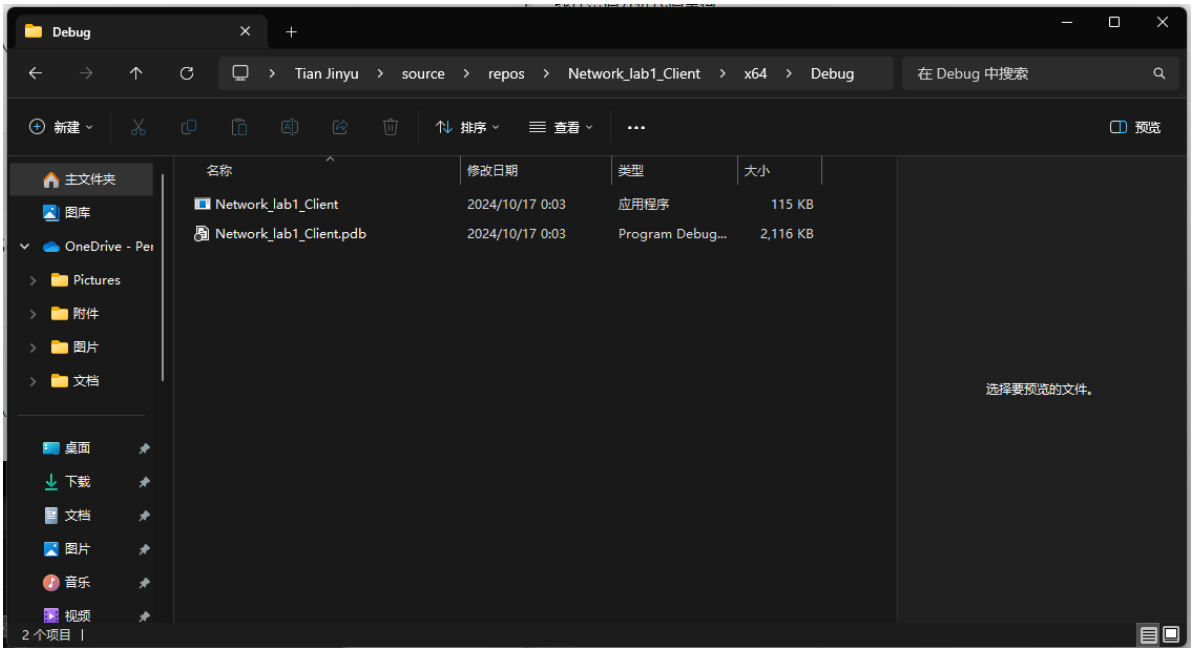
```
closesocket(client_socket);
WSACleanup();
```

- 当客户端退出时，关闭套接字，并调用 `WSACleanup()` 释放 WinSock 资源。

程序效果

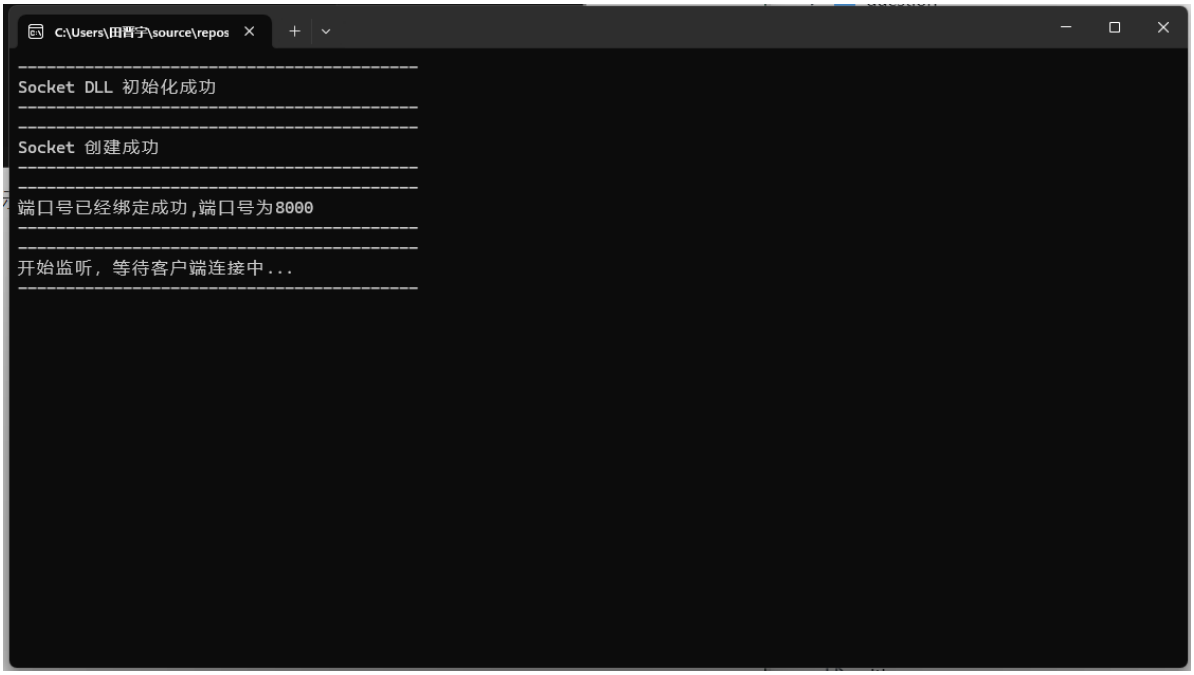
首先我们需要生成两段程序的可执行文件：



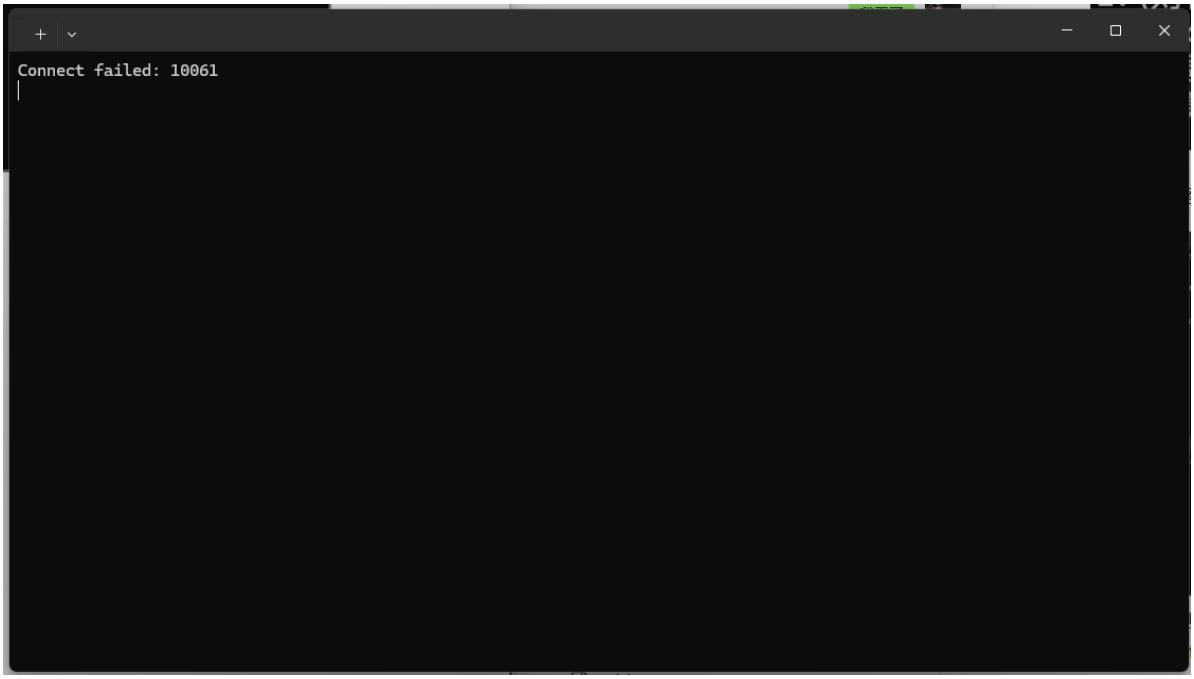


我们需要先打开服务器的可执行文件，会依次展示以下信息：

- Socket DLL初始化成功
- Socket创建成功
- 端口号已经绑定
- 开始监听

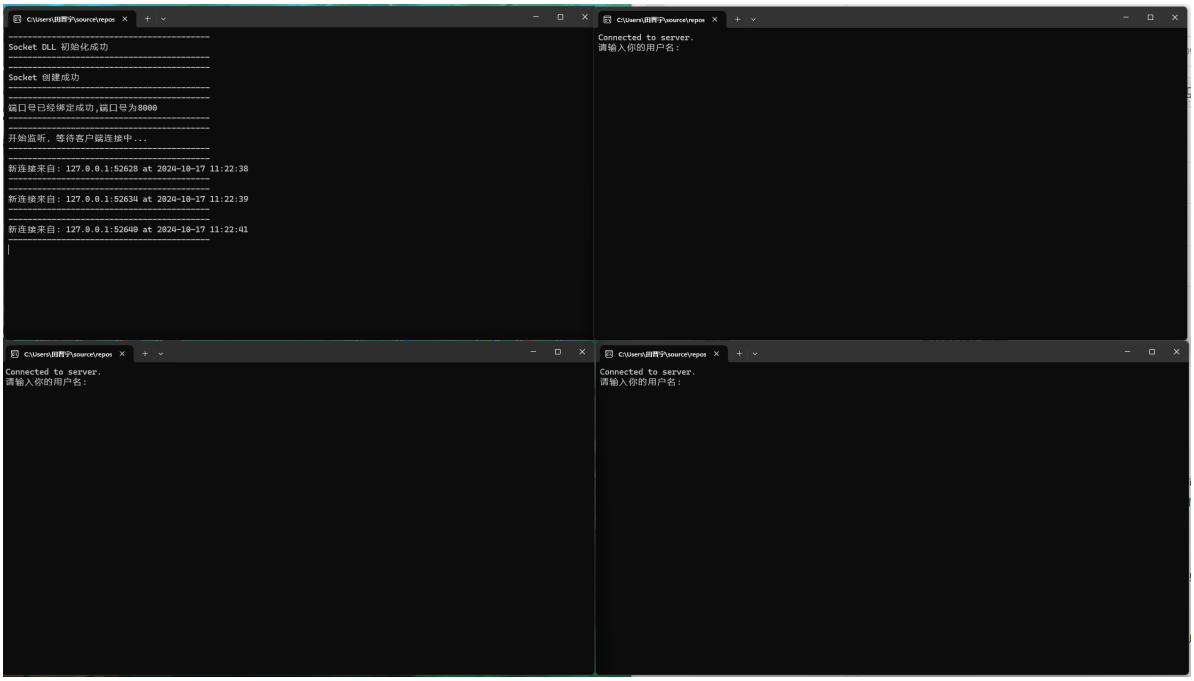


如果未打开服务器，直接打开客户端，客户端会输出连接失败的调试信息：

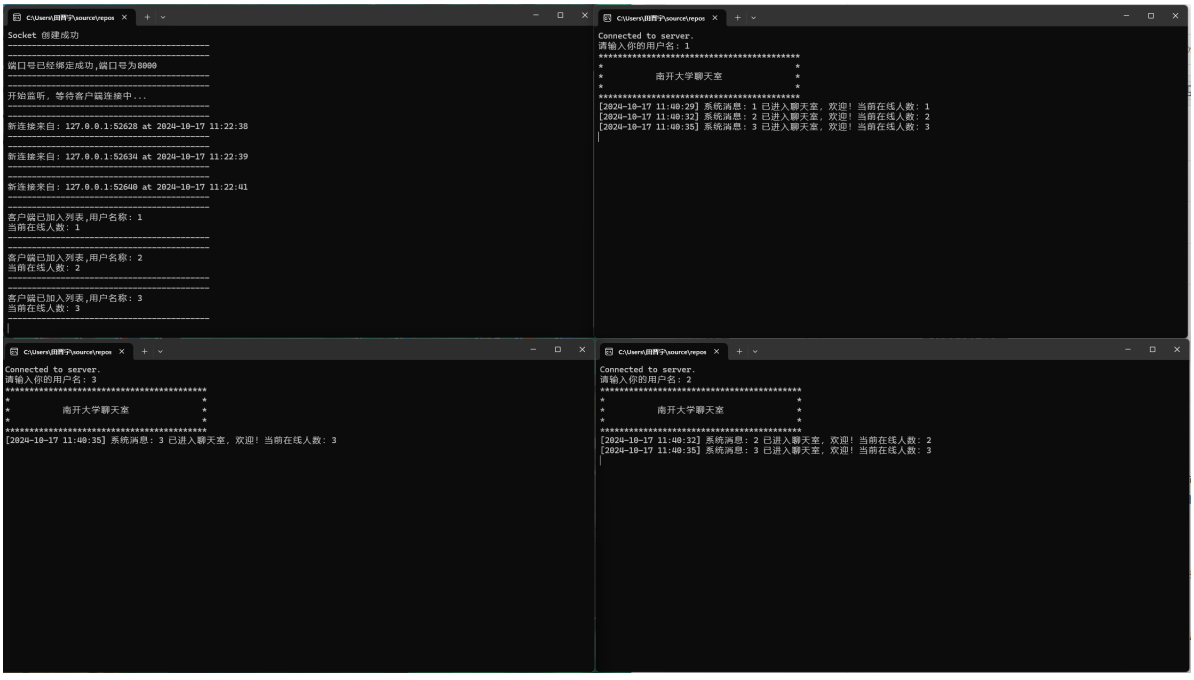


接下来我们模拟一个三人聊天的场景，我们初始化三个客户端窗口，可以看到在服务器端窗口输出了三行调试信息表明接收到了客户端的连接请求，在客户端中要求用户输入一个用户名进行注册。在之前的测试中，一次性打开多个窗口之后，服务器端只能收到最开始初始化的那个客户端的连接请求，这是因为之前我将服务器端处理用户名的逻辑放在了主函数中，而没有放到单独的线程中，之后经过修改之

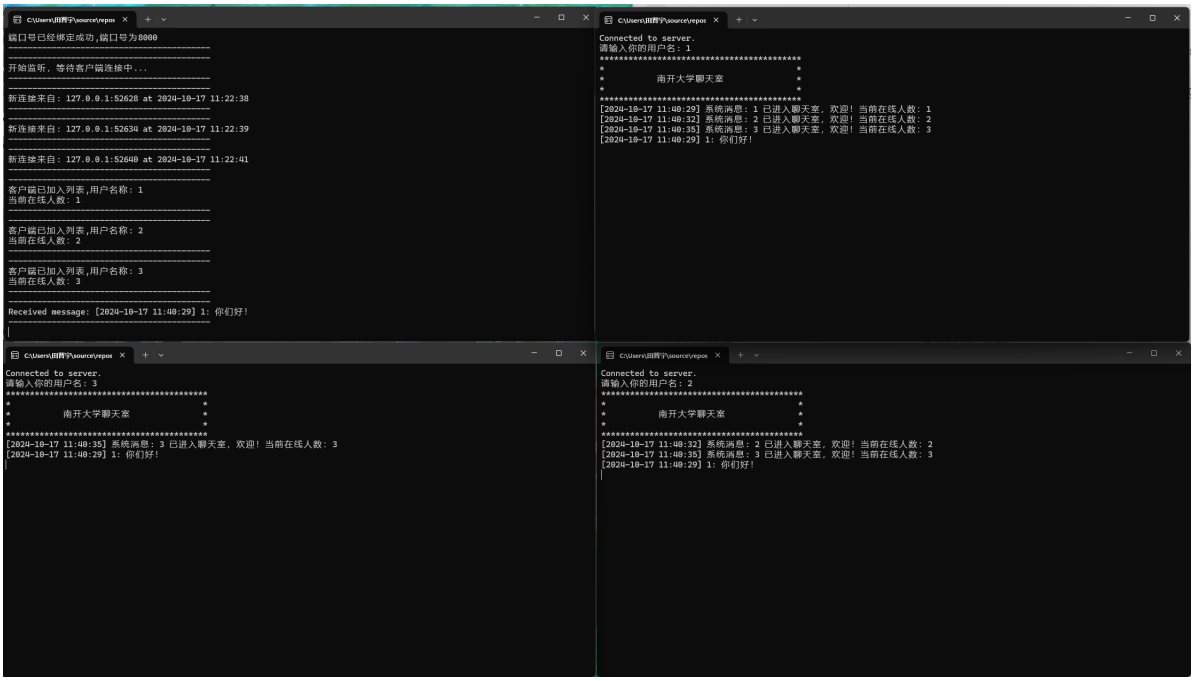
后，服务器可以正常的接收到多个请求。



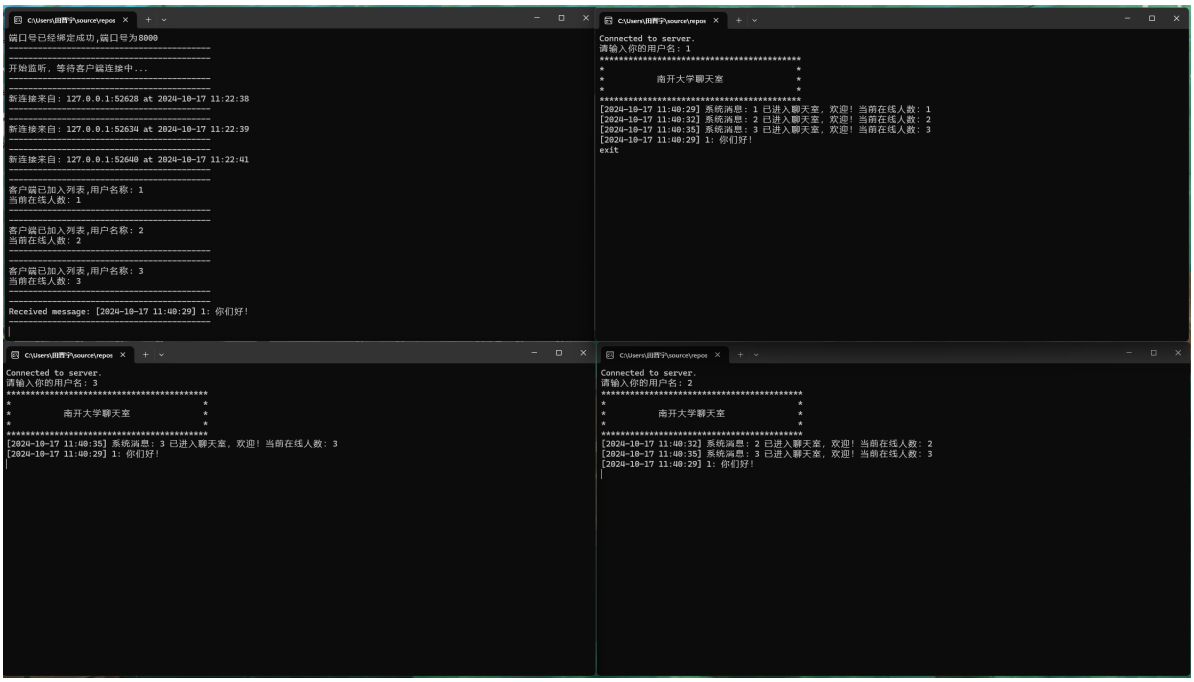
当用户成功注册后，服务器端会将该用户的欢迎信息以及当前在线人数广播给所有客户端。之前我发现广播机制会向所有已连接好的客户端进行发送消息，这将会导致用户注册完之后能够收到之前所有的广播消息，因此我们改变了广播机制的逻辑，只向已经有用户名的客户端进行广播，这样只有用户注册完毕之后才能正常的接收到消息。

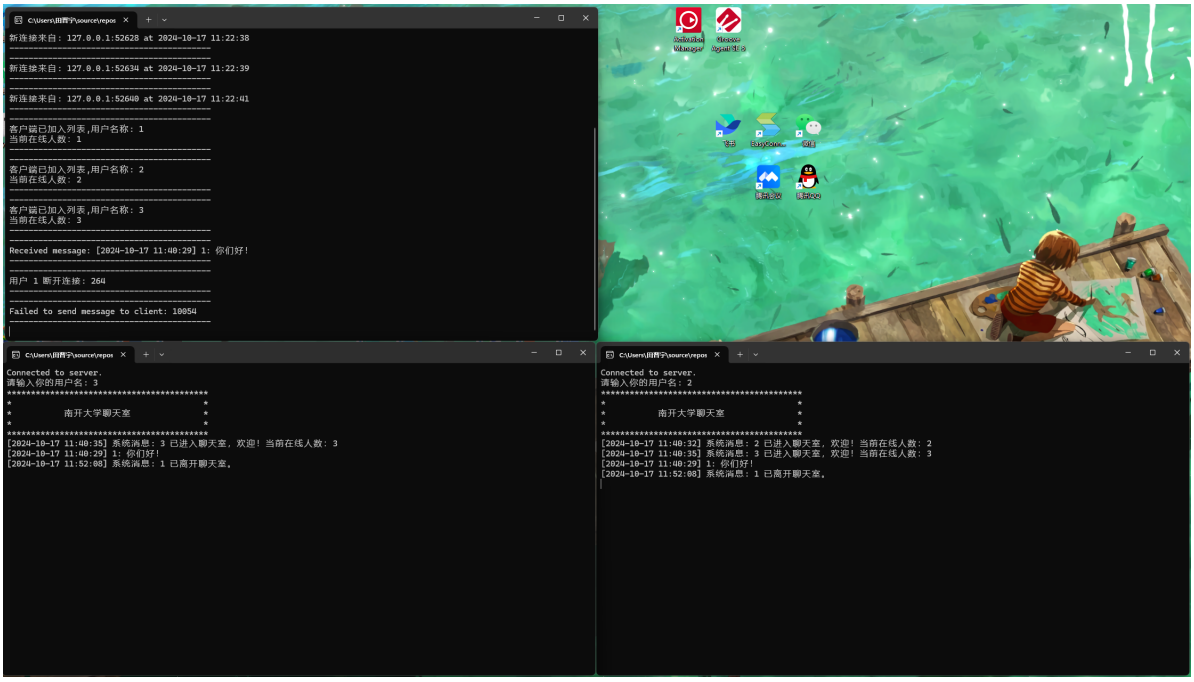


接下来我们发送一条消息，服务器可以正常的接收消息，并将这条消息广播给所有客户端。

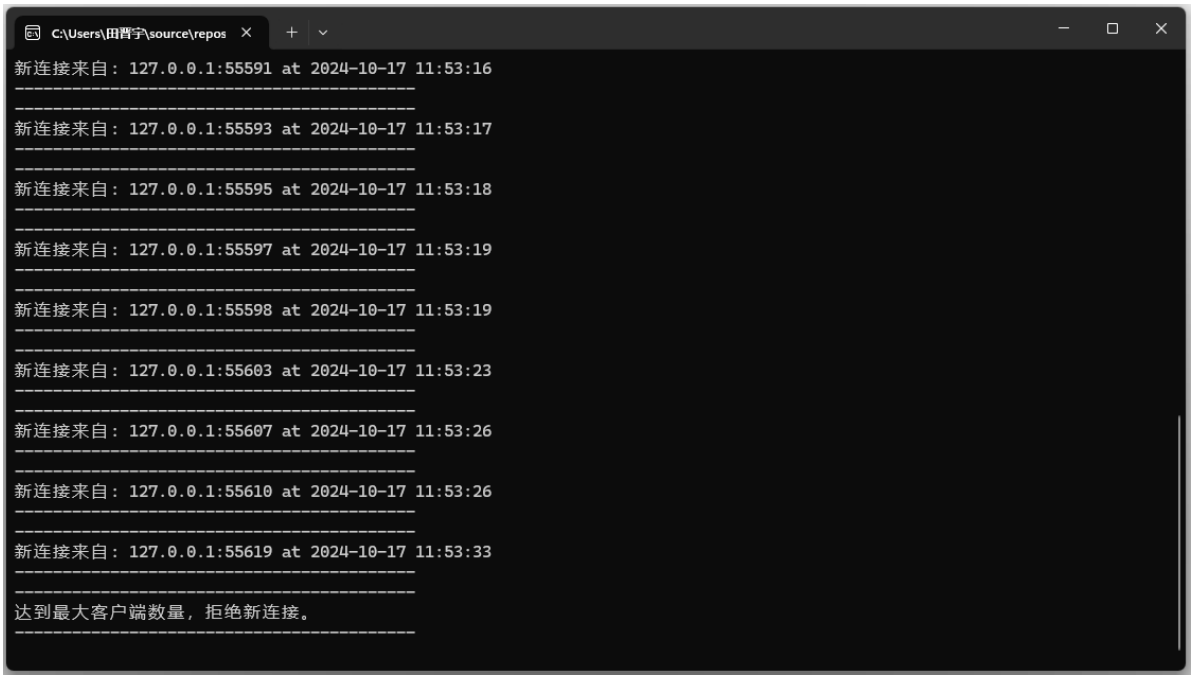


当一个客户端打算退出时，可以在聊天框内输入exit，之后服务器端便会断开与该客户端的连接，同时会广播系统消息该用户已离开聊天室。





最后我们验证服务器的最大连接数功能，当我们一次打开大于10个终端时，服务器端会显示调试信息。



心得感悟

在本次实验中，我通过实现基于TCP协议的多线程聊天室服务器，深入理解了网络编程的基础概念与实际应用。在实验中，利用套接字进行客户端与服务器端的连接，熟悉了TCP流式套接字的稳定性和可靠性。同时，通过多线程技术，实现了服务器能够同时处理多个客户端的消息收发，并保证用户之间的消息能够实时广播。

实验过程中遇到的挑战是如何保证在多客户端的环境下，服务器对共享资源的操作安全性。通过引入互斥锁机制，我解决了并发访问数据时的竞态条件。此外，通过维护用户连接和断开的状态，我学会了更好地管理服务器资源和用户状态。

此次实验不仅让我掌握了网络通信的编程技巧，还提升了我在实际开发中问题分析和解决的能力。