南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机网络实验报告

实验 1：利用流式套接字编写聊天程序

贾景顺

年级：2022 级

专业：计算机科学与技术

指导教师：张建忠　徐敬东

2025 年 11 月 23 日

摘要

本次实验基于 TCP/IP 协议族，利用 C++ 语言及 Windows 平台的 Winsock2 网络库，设计并实现了一个基于控制台的多人在线聊天系统。程序架构分为服务器端（$Server$）与客户端（$Client$），二者通过流式套接字（$SOCKET\_STREAM$）建立可靠的面向连接通信。在核心实现上，程序采用了 C++11 标准的多线程技术（$std :: thread$），使得服务器能够并发处理多个客户端的连接请求与消息转发，客户端能够实现全双工的消息发送与后台接收。为了保证多线程环境下的数据一致性，实验引入了互斥锁（$std :: mutex$）机制，有效解决了并发访问全局用户列表时的资源竞争问题。此外，系统还实现了带有时间戳的消息广播、非阻塞式服务器控制以及基于 UTF-8 编码的中文环境适配，确保了交互界面的友好性与程序的健壮性。

关键字：Socket 编程，TCP 协议，Winsock2，多线程

目录

一、 实验要求 **1**

二、 协议设计 **1**

2.1 用户连接与注册 2

2.2 聊天信息广播 3

2.3 用户连接断开 4

三、 模块功能说明 **5**

3.1 服务器端模块 5

3.1.1 SOCKET 绑定与监听 5

3.1.2 客户端线程管理 6

3.1.3 消息全体广播 7

3.2 用户端模块 8

3.2.1 用户连接 8

3.2.2 广播消息接收 9

四、 运行环境 **9**

五、 程序运行演示 **10**

5.1 连接失败 10

5.2 单独连接 10

5.3 多端连接 10

5.4 连接退出 11

一、 实验要求

1. 设计聊天协议，并给出聊天协议的完整说明。
2. 利用 C 或 C++ 语言，使用基本的 Socket 函数进行程序编写，不允许使用 CSocket 等封装后的类。
3. 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。
4. 完成的程序应能支持英文和中文聊天。
5. 采用多线程，支持多人聊天。
6. 编写的程序应结构清晰，具有较好的可读性。
7. 在实验中观察是否有数据包的丢失，提交程序源码、可执行代码和实验报告。

二、 协议设计

本实验设计的 Socket 多人聊天程序是基于 **TCP（Transmission Control Protocol）协议**和**流式套接字（Stream Socket）**实现的。TCP 是一种面向连接的、可靠的、基于字节流的传输层通信协议。在数据传输之前，通信双方必须先建立连接（三次握手），这为聊天室内消息的有序到达和完整性提供了底层保障。流式套接字为应用程序提供了一个双向的、有序的、无重复的数据流服务，非常适合文本聊天这种对即时性和准确性要求较高的应用场景。

本系统的应用层通信协议主要包含三个阶段：用户连接与注册、聊天消息的广播处理、用户断开与资源回收

2.1 用户连接与注册

当用户启动客户端程序并选择连接服务器时，首先触发底层的 TCP 连接请求。交互流程如图 1 所示。

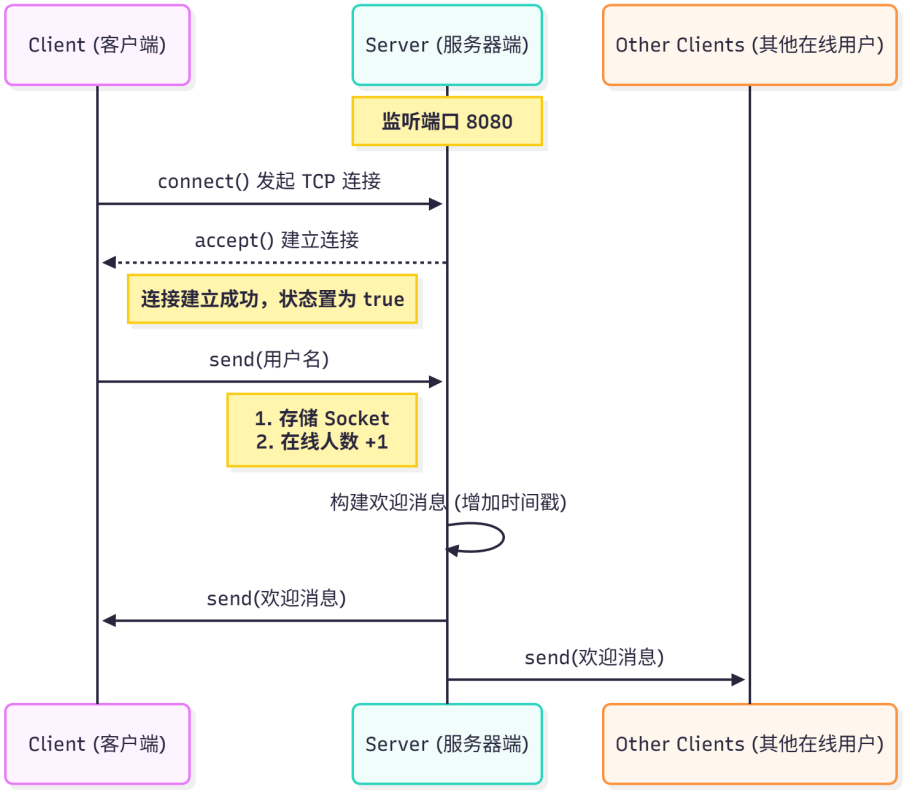

图 1: 用户连接与注册

服务器端（Server）启动后，主线程通过 *listen* 函数进入监听状态，等待连接。客户端（Client）调用 *connect* 函数向服务器发起连接请求，服务器端的 *handle_connections* 线程通过 *accept* 函数响应请求，完成 TCP 连接的建立。

客户端在连接成功后，立即通过 *send* 函数将用户输入的用户名（Username）发送至服务器。这是连接后的第一个数据包，用于标识该 Socket 的归属。

服务器接收到用户名后，首先更新内部的在线人数计数器（*client_count*），随后构建一条包含时间戳的欢迎消息。该消息不仅发回给新用户，也通过 *send_message_to_all* 函数广播给当前列表中的所有已连接用户，以通知群组有新成员加入。

2.2 聊天信息广播

在聊天会话阶段，系统采用“客户端封装内容，服务器附加时间”的协作模式，确保所有用户看到的消息格式统一且具有时效性。消息传输时序如图 2 所示。

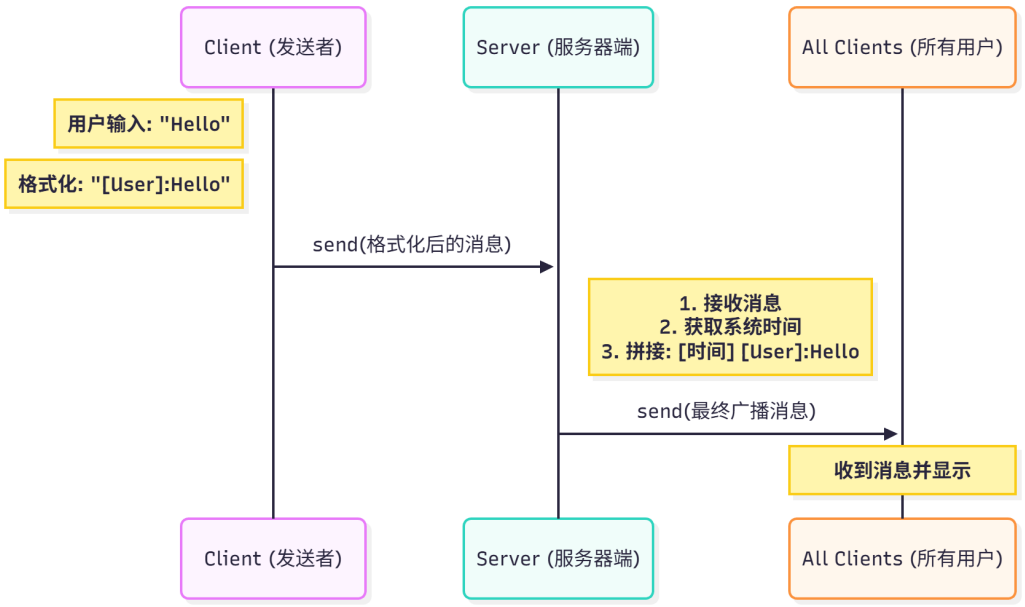

图 2: 聊天信息广播

用户在客户端输入消息后，客户端程序首先在本地进行预处理，将用户名作为前缀拼接到消息内容前，格式为 [用户名]: 消息内容。随后，客户端调用 $send$ 函数将封装好的字符串发送至服务器。

服务器通过 $recv$ 函数接收到数据流后，调用 $time$ 和 $strftime$ 函数获取当前的系统时间，格式化为 [YYYY-MM-DD HH:MM:SS]。服务器将时间戳附加在收到的消息头部，形成最终的广播报文。同时，服务器遍历维护的 $client\_sockets$ 列表，将最终报文发送给包括发送者在内的所有客户端。客户端的后台接收线程（$receive\_messages$）收到数据后，直接将其打印至控制台。用户输入：

消息格式示例：

- 用户 $user1$ 输入：Hello World

- 客户端发送：[user1]:Hello World
- 服务器广播：[2025-10-24 00:00:00] [user1]:Hello World

2.3 用户连接断开

程序设计了明确的断开连接协议，分为用户主动退出和被动断开两种情况。

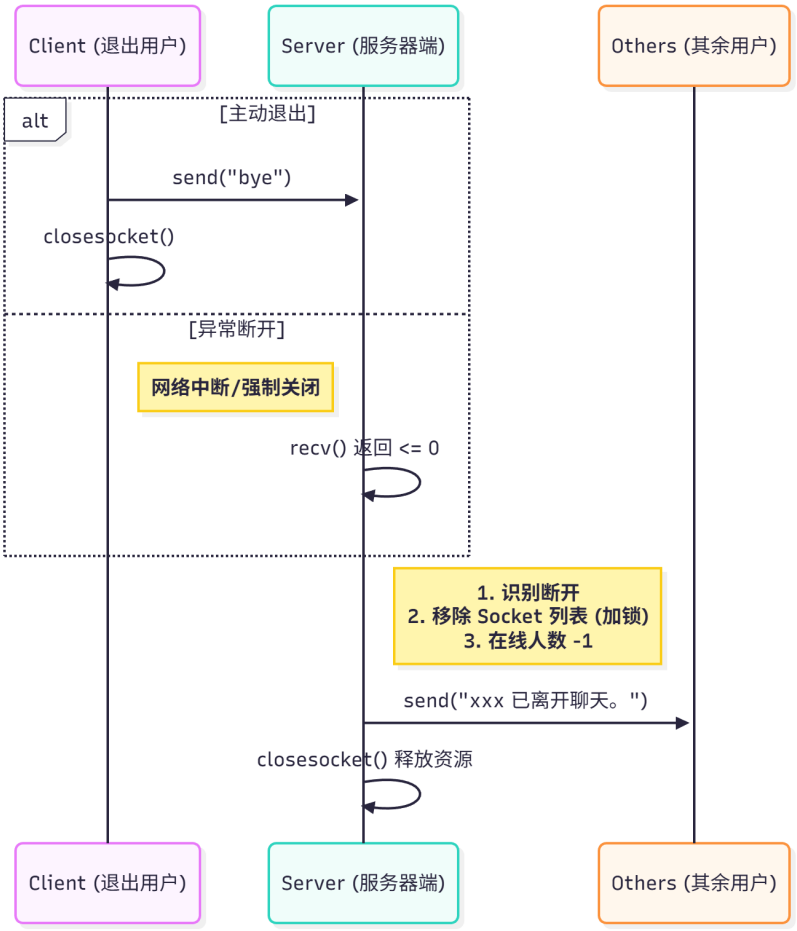

图 3: 用户连接断开

当用户在客户端输入特定指令 *bye* 时,客户端首先向服务器发送字符串"bye",随后关闭本地 Socket 并终止接收线程。服务器若接收到内容为"bye\n" 或"bye"的数据包，判定为用户主动退出；若 *recv* 函数返回值为 0 或负数，表明客户端已异常断开（如网络中断或强制关闭程序）。

一旦确认连接断开，服务器首先构建“用户已离开”的通知消息并广播给剩

余用户。随后，利用 *std* :: *mutex* 互斥锁安全地从全局 Socket 列表中移除该客户端，减少在线人数计数，并关闭对应的 Socket 句柄以释放系统资源。

三、 模块功能说明

3.1 服务器端模块

3.1.1 SOCKET 绑定与监听

SOCKET 绑定与监听

```
    // 绑定Socket
    sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    server_addr.sin_port = htons(PORT);
    if (bind(server, (sockaddr*)&server_addr, sizeof(
        server_addr)) == SOCKET_ERROR) {
        cout << "Bind failed." << endl;
        closesocket(server);
        WSACleanup();
        return 1;
    }
    // 监听Socket
    if (listen(server, 3) == SOCKET_ERROR) {
        cout << "Listen failed." << endl;
        closesocket(server);
        WSACleanup();
        return 1;
    } else {
        cout << "服务器启动成功! \n正在监听端口: " << PORT << "
            ..." << endl;
    }
    // 启动处理客户端连接的线程
    thread connection_thread(handle_connections, server);
    monitor_keyboard();
    closesocket(server);
    connection_thread.join();
```

程序通过填充 *sockaddr_in* 结构体来定义通信协议族为 IPv4（*AF_INET*）、绑定本地回环地址（127.0.0.1）以及指定服务端口。随后调用的 *bind* 函数将创建的套接字与上述地址信息进行强关联，若端口已被占用则立即终止程序，而 *listen* 函数则将该套接字从默认的主动发送状态切换为被动监听状态，参数 3 指

定了请求队列的长度，标志着服务器已准备好接收来自传输层的连接请求。

另外，程序利用 C++11 的 *std :: thread* 创建了一个名为 *connection_thread* 的新线程专门执行 *handle_connections* 函数，该函数内部包含阻塞式的 *accept* 调用。这一设计使得服务器能够在一个独立的执行流中等待和处理新的客户端连接，而不会阻塞主线程的运行。主线程随即可立即执行 *monitor_keyboard* 函数来监听控制台输入的 *close* 指令，从而实现了对服务器的非阻塞式控制。

3.1.2 客户端线程管理

客户端线程管理

```
void handle_client(SOCKET client_socket)
{
    char name[NAME_SIZE];
    char buffer[BUFFER_SIZE];
    int name_len = recv(client_socket, name, NAME_SIZE - 1, 0);
    name[name_len] = '\0';
    {
        lock_guard<mutex> lock(client_mutex);
        client_count++;
    }
    string welcome_msg = string(name) + "已加入聊天，当前共有"
       + to_string(client_count) + "位用户在线。";
    send_message_to_all(welcome_msg, client_socket);
    while (server_running) {
        memset(buffer, 0, BUFFER_SIZE);
        int bytes_received = recv(client_socket, buffer,
           BUFFER_SIZE, 0);
        if (bytes_received > 0)
        {
            buffer[bytes_received] = '\0';
            string message = string(buffer);
            if(string(buffer)=="bye\n")
            {
                string leave_msg = string(name) + "已离开聊
                   天。";
                send_message_to_all(leave_msg, client_socket);
                break;
            }
            else
            {
                send_message_to_all(message, client_socket);
            }
        }
        if (bytes_received <= 0) {
            string leave_msg = string(name) + "已离开聊天。";
```

```
33                send_message_to_all(leave_msg, client_socket);
34                break;
35            }
36        }
37        {
38            lock_guard<mutex> lock(client_mutex);
39            client_sockets.erase(remove(client_sockets.begin(),
                  client_sockets.end(), client_socket), client_sockets
                  .end());
40            client_count--;
41        }
42        closesocket(client_socket);
43    }
```

该模块是服务器端处理并发用户会话的核心执行单元，函数执行伊始，首先通过阻塞式的 $recv$ 调用接收客户端发送的第一个数据包，即用户昵称，从而完成应用层的身份握手。在此阶段，代码实现的一个关键要素是对全局共享资源的安全访问：为了更新当前在线人数 $client_count$，程序引入了 C++ 标准库中的 $lock\_guard < mutex >$ 机制，确保进入作用域时自动锁定互斥量 $client\_mutex$，并在作用域结束时自动解锁。这一设计有效防止了多线程环境下因竞态条件导致的数据不一致问题，确保了在线人数统计的原子性。

随后，函数持续监听来自客户端的数据流，调用 $send_message_to_all$ 实现全网广播。

3.1.3 消息全体广播

消息全体广播

```
1   void send_message_to_all(const string &message, SOCKET
        sender_socket) {
2       lock_guard<mutex> lock(client_mutex);
3       char time_buf[80];
4       time_t now = time(nullptr);
5       strftime(time_buf, sizeof(time_buf), "[%Y-%m-%d %H:%M:%S]",
            localtime(&now));
6       string msg=string(time_buf) + " " + message;
7       cout<<msg<<endl;
8       for (auto &socket : client_sockets) {
9           send(socket, msg.c_str(), msg.size(), 0);
10      }
11  }
```

将单一用户发送的信息或系统通知分发给当前所有在线的客户端, 通过引入 *std :: lock_guard* 对互斥量 *client_mutex* 进行加锁，构建了一个受保护的临界区, 确保了在后续遍历 *client_sockets* 列表的过程中服务器的稳定。

3.2 用户端模块

3.2.1 用户连接

用户连接

```
void connect_server()
{
    sockaddr_in client_addr;
    client_addr.sin_family = AF_INET;
    client_addr.sin_port = htons(PORT);
    client_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    client = socket(AF_INET, SOCK_STREAM, 0);
    if (connect(client, (sockaddr*)&client_addr, sizeof(
        client_addr)) == SOCKET_ERROR) {
        cout << "无法连接到服务器。" << endl;
        return;
    }
    else
    {
        connected = true;
        cout << username << "成功连接到服务器! " << endl;
        cout << "您可以开始发送消息了, 输入'bye'以断开连接。"
            << endl;
        send(client, username.c_str(), username.size(), 0);
        thread receive_thread(receive_messages);
        receive_thread.detach();
    }
}
```

函数封装了客户端发起 TCP 连接请求及初始化会话状态的完整流程。首先，代码通过配置 *sockaddr_in* 结构体指定了目标服务器的 IPv4 地址与监听端口，在此过程中，*inet_addr* 与 *htons* 函数确保了 IP 地址与端口号符合网络字节序的标准格式。随后执行的 *connect* 函数是建立链路的关键，它利用创建好的流式套接字向服务器发起三次握手请求，若返回值为 *SOCKET_ERROR*，则表明网络不可达或目标拒绝连接，程序将输出错误提示并终止后续操作。

3.2.2 广播消息接收

广播消息接收

```
void receive_messages()
{
    char buffer[5000];
    while (connected) {
        memset(buffer, 0, sizeof(buffer));
        int bytes_received = recv(client, buffer, sizeof(buffer
            ) - 1, 0);
        if (bytes_received > 0)
        {
            buffer[bytes_received] = '\0';
            string message(buffer);
            cout << message << endl;
        }
        else
            break;
    }
}
```

函数进入受 *connected* 标志位控制的 while 循环体，每次执行接收操作前，均调用 *memset* 对 5000 字节的缓冲区进行彻底清零。随后循环调用 *recv* 函数，数据成功接收后，立即利用返回值 *bytes_received* 在字节流末尾显式添加空字符 \0，并将其转换为 *string* 对象并输出至控制台。

四、 运行环境

本实验的编译与测试在 Windows11 系统下进行，采用 Visual Studio Code 并配置 Microsoft Visual C++ (MSVC) 编译器（cl.exe）进行代码的构建。

另外,在网络编程中,链接器必须将程序与 Windows Sockets 2 库($ws2\_32.lib$)进行链接才能解析 *socket*, *bind*, *connect* 等符号。在本实验的代码实现中，通过在源码头部添加预处理指令 $\#pragma\ comment(lib, "ws2\_32.lib")$，指示链接器自动在默认库路径中搜索并链接该静态库。

五、 程序运行演示

5.1 连接失败

如图所示，在服务器未启动的情况下造成连接失败，在用户端报错后返回上一界面。

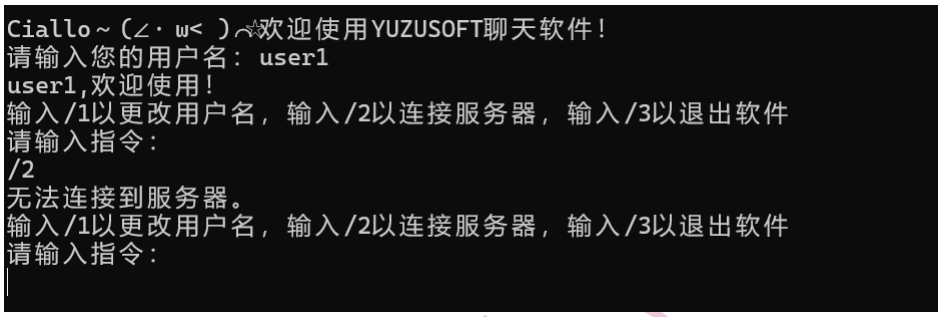

图 4: 用户连接失败

5.2 单独连接

如图 5 所示，在服务器成功启动后，用户能够成功连接并发送中英文消息，并在服务器接收后广播。

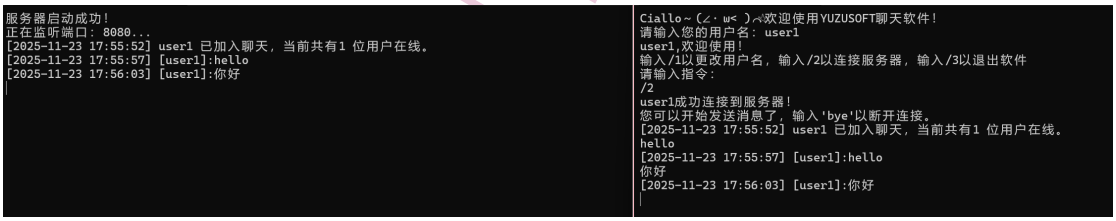

图 5: 用户单独连接

5.3 多端连接

如图 6 所示，多个用户均能够正确连接到服务器，并支持中英文的用户名及聊天，同时服务器也能正确记录用户加入信息以及在线用户数。在实验过程中并未观察到丢包现象。

图 6: 用户多端连接

5.4 连接退出

如图 7 所示，输入 *bye* 后用户退出当前聊天频道并返回上一级，同时服务器更新在线人数信息。

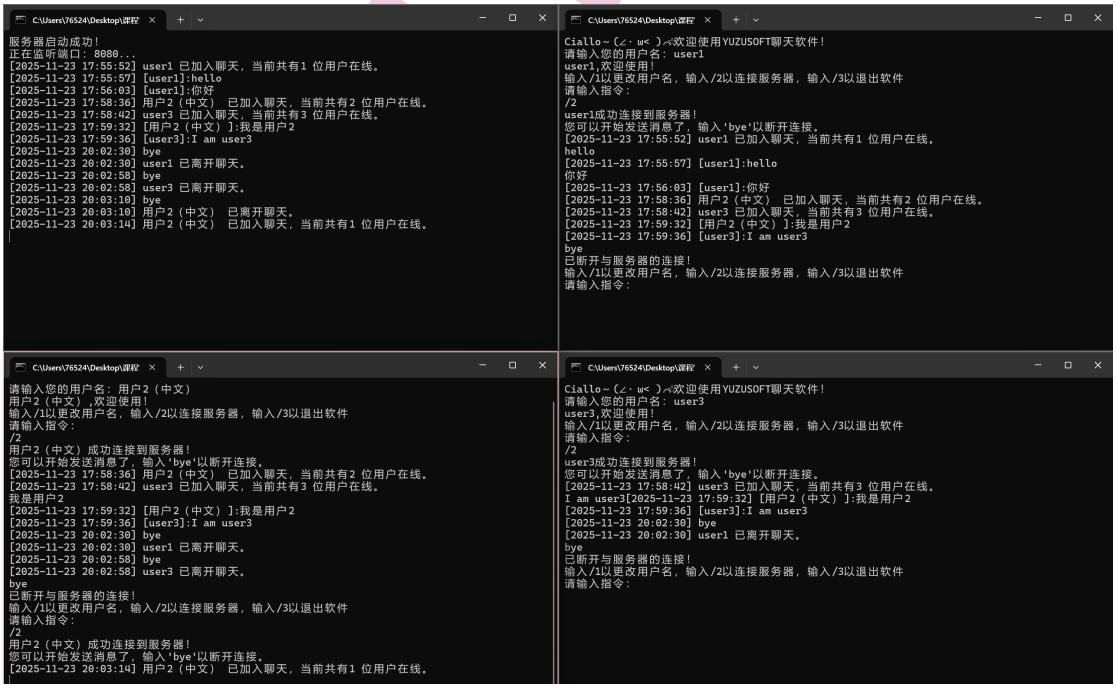

图 7: 用户断开连接

如图 8 所示，服务器端输入 *close* 指令后，服务器关闭。

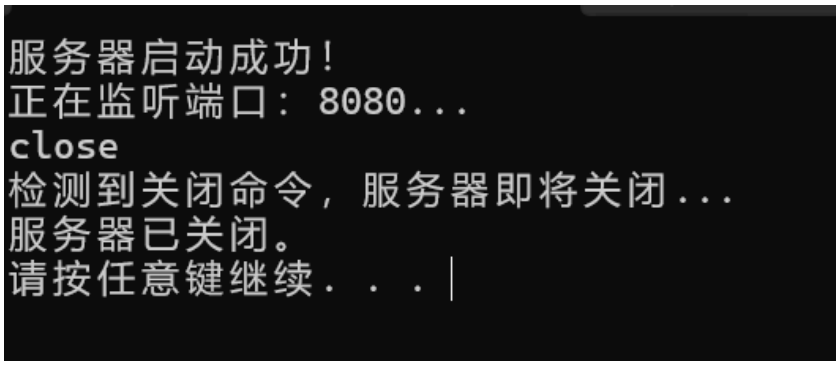

图 8: 服务器关闭