南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机网络实验报告

Socket 聊天程序

姓名：姚知言

年级：2022 级

专业：计算机科学与技术

指导教师：张建忠 徐敬东

2024 年 10 月 19 日

摘要

本次实验利用 Socket 技术，通过 C++ 语言的 winsock2 库实现多人聊天程序。程序包括用户端（client）和服务器（server）端。程序使用流式套接字和 thread 多线程技术，并通过 mutex 库完成加锁保证线程安全性。程序实现了较为友好的命令行对话界面，并能实现改名，连接/退出连接，退出程序等功能，同时增加了时间戳显示设计。

关键词：Socket，多人聊天，winsock2，多线程

目录

一、 实验要求

1. 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. 程序应该有基本的对话界面，但可以不是图形界面。程序应该有正常的退出方式。
5. 完成的程序应该支持多人聊天，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丢失，提交可执行文件、程序源码和实验报告。

二、 协议设计

本实验的 Socket 多人聊天程序是基于 TCP 协议和流式套接字实现的。TCP 是可靠的传输层协议，可以较好的保证数据的完整及准确。流式套接字（Stream Socket）是基于 TCP 的，它提供了一个持久的、面向连接的通信通道。通信双方在传输数据前必须建立连接，保证数据的可靠传输。流式套接字可以通过简单的接口实现可靠的字节流传输，应用场景广泛。

(一) 用户连接服务器

用户连接服务器的流程如图1所示。服务器建立好后一直开启 listen 监听状态，通过 accept 接收用户端的 connect 连接请求，即用户端和服务器端连接成功。

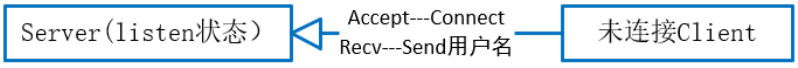

图 1: 未连接 Client 连接示意图

随后，用户端首先发送用户名到服务器。

在这一部分中，新连接用户端向服务器端发送的消息为：用户名。

例如：user1

服务器收到后，对该信息进行包装，增加时间戳和额外信息，将其广播给所有已连接的用户，通知新用户加入。

在这一部分中，服务器端向用户端发送的消息为：时间戳 + 用户名 +” 加入了聊天，当前共有”+ 当前在线人数 +” 人聊天。”。

例如：[2024-10-18 18:39:08] user1 加入了聊天，当前共有 1 人聊天。

(二) 聊天信息处理及广播

用户发送信息广播的时序如图2所示，即用户先将信息 send 到服务器端，服务器端 recv 后，经过处理在广播到所有已经连接的客户端。

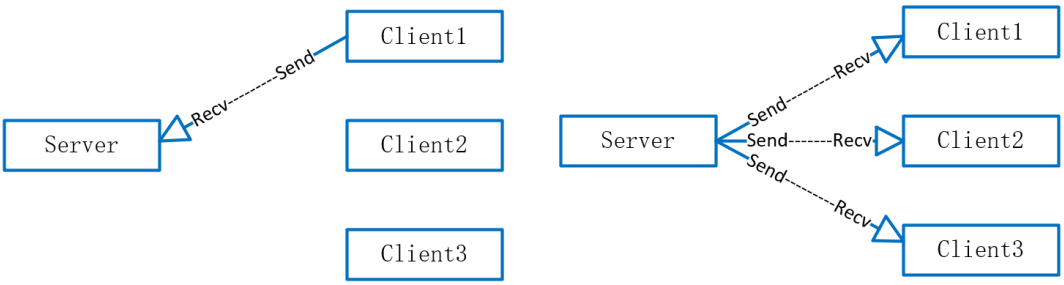

图 2: 用户端发送信息广播示意图

当用户端连接到服务器后，用户在终端中输入消息（若该消息不是退出服务器的指令），用户端将信息包装上用户名后，发送到服务器端，用于广播。

在这一部分中，用户端向服务器端发送的消息为：用户名 +”:“+ 消息。

例如：user1: 111

服务器收到后，判断该消息非退出消息，并对信息增加时间戳，将其广播给所有已连接的用户。

在这一部分中，服务器端向用户端发送的消息为：时间戳 + 用户名 +”:“+ 消息。

例如：[2024-10-18 18:46:34] user1: 111

(三) 用户断开服务器

当用户通过程序方法，即通过指定的输入”
d“方式断开服务器时（对于非正常退出情况，程序也能够正常响应，会在后续部分说明），会向服务器端发送一条”
d”。

服务器收到并识别后，会发送一条以下格式的信息：时间戳 + 用户名 +” 退出了聊天，当前共有”+ 在线人数 +” 人聊天。”。

例如：[2024-10-18 18:47:48] user1 退出了聊天，当前共有 0 人聊天。

此外，当用户因意外退出服务器时（手动关闭终端，网络连接错误等），经过 recv 函数检测到后，服务器同样会进行更新，并发送一条以下格式的信息：时间戳 + 用户名 +” 断开了连接，当前共有”+ 在线人数 +” 人聊天。”。

例如：[2024-10-18 18:47:48] user1 断开了连接，当前共有 0 人聊天。

三、 模块功能说明

(一) 通用模块

1. WSA 建立

WSA 建立

```
WSADATA wsa;
if (WSAStartup(MAKEWORD(2, 2), &wsa))
{
    cout << "\033[31m WSA 创建失败! \033[0m \n";
```

```
    return 0;
}
```

函数 *WSAStartup* 解析：

第一个参数：*WORD* 类型，指定使用的 *winsock* 版本，如 $MAKEWORD(2,2)$ 为使用 2.2 版本。

第二个参数：*LPWSADATA*，即指向 *WSADATA* 的指针类型，接收套接字实现的信息。

功能：初始化套接字库。

返回值：返回 0 表示成功，否则表示失败。

该模块在用户端和服务器端都需要搭建，作用是创建 *WSA*，存储 *socket* 数据。若创建失败则输出创建失败提示并结束程序。

2. 清理 WSA 环境

清理 WSA 环境

```
WSACleanup();
```

函数 *WSACleanup* 解析：与 *WSAStartup* 对应出现，清除 *winsock* 环境，返回 0 表示成功。

3. 创建 Socket

创建 Socket

```
//以服务器端为例，客户端类似
SOCKET ser_socket;
ser_socket = socket(PF_INET, SOCK_STREAM, 0);
```

函数 *socket* 解析：

三个参数都为 *int* 类型，第一个参数指定地址族，*PF_INET/AF_INET* 表示 *IPV4* 网络；第二个参数指定套接字类型，*SOCK_STREAM* 表示 *TCP* 协议，数据有序传输；第三个参数通常为 0，表示根据套接字类型自动选择协议。

PF_INET/AF_INET 是等价的。实际含义分别为协议族和地址族。早期的 *socketapi* 的设计者认为同一个地址族可以被多个不同的协议族使用。但实际上这个特性并未被实现，所以后来 *AF_INET* 和 *PF_INET* 可以认为没任何区别。

(二) 服务器端模块

1. 套接字绑定地址进入监听

bind 为套接字绑定地址和端口

```
struct sockaddr_in addr,u_addr;
addr.sin_family = AF_INET; //IPV4
addr.sin_port = htons(4444); //端口号为4444
addr.sin_addr.s_addr = INADDR_ANY; //接受任何IP地址的连接
bind(ser_socket, (struct sockaddr *)&addr, sizeof(addr));
if(!listen(ser_socket, 10))
```

```
7       cout<<"\033[34m␣服务器启动成功!␣\033[0m␣\n";
8   else{
9       cout<<"\033[31m␣服务器启动失败。␣\033[0m␣\n";
10  }
```

函数 *bind* 解析：

第一个参数为 *SOCKET* 类型，即创建的套接字。第二个参数为 *sockaddr* 结构体，在代码 2-4 行中指定了 IP 地址和端口等信息。第三个参数为结构体大小。

功能：把创建的套接字绑定到一个本地地址（IP 和端口）。

函数 *listen* 解析：

两个参数为创建的套接字和允许同时处理的最大连接请求数。返回 0 表示成功。

功能：对于 *TCP* 套接字，使套接字进入监听状态，等待客户端连接。

该模块用 sockaddr_in 结构体定义了协议、地址等信息，并通过 bind 函数进行绑定。随后通过 listen 函数进入监听状态，及时输出监听建立情况。

2. 新连接处理

新连接处理

```
1   //全局变量和被调函数（在后续内容中不再重复展示）
2   vector<SOCKET> client_sockets;
3   mutex clients_mutex;//互斥锁
4   void add_client(SOCKET client_socket) {
5       lock_guard<mutex> lock(clients_mutex);
6       client_sockets.push_back(client_socket);
7   }
8   //模块内容
9   while(true){
10      int u_len = sizeof(u_addr);
11      SOCKET u_socket = accept(ser_socket, (struct sockaddr*)&u_addr, &u_len);
12      add_client(u_socket);
13      thread client_thread(op_client, u_socket);
14      client_thread.detach();
15  }
16  closesocket(ser_socket);
```

函数 *accept* 解析：

第一个参数为服务器端 SOCKET，第二个参数为存储用户端地址信息的结构体指针，第三个参数为结构体的大小。

函数的返回值是接收到的 SOCKET，表示与该用户端建立连接。

函数 *closesocket* 解析：含有一个 SOCKET 参数，关闭已打开的套接字释放资源。

MUTEX 互斥锁的介绍：第 5 行 LOCK 操作是加锁操作，保证各线程不会同时更新用户列表。该操作的生存期随着所的生存期结束而结束，在本例中，当 ADD_CLIENT 函数结束后就会解锁。

THREAD 线程操作的介绍：13 行建立一个新线程，第一个参数表示调用的函数，第二个以后的参数表示函数的参数（如果需要）。14 行操作为分离线程，使得线程可以在后台独立运行，不会影响主线程的运行。

该模块的功能是当服务器端开始监听后，调用 *accept* 函数循环接收连接请求，若接收到用户套接字，则将其赋值给 *u_socket*，通过 *add_client* 将其添加到连接的用户 *socket* 数组中（该部分操作通过 *mutex* 互斥锁保证线程安全），然后通过为该用户分配一个线程，调用 *op_client* 用于处理该用户发送的信息（后续介绍），用户 *socket*，即 *u_socket* 是该函数唯一参数。最终 *detach*() 语句为分离线程，使其可以在后台独立运行。（在服务器关闭时，应调用 *closesocket* 释放资源，在本程序中也进行了设计，不过本程序并未设置服务器关关闭的方式，因此不会运行。）

3. 服务器端打印及用户端广播

服务器端打印及用户端广播

```
1  void send_message_to_all(const string &message, SOCKET sender_socket) {
2      lock_guard<mutex> lock(clients_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;
8      for (auto &socket : client_sockets) {
9          send(socket, msg.c_str(), msg.size(), 0);
10     }
11 }
```

函数 *send* 解析：

第一个参数为要发送到的 *socket*，第二个参数为消息（由于为 *const char** 类型，往往需要进行类型转换），第三个参数为消息长度，第四个参数一般为 0，表示默认发送。

该函数的功能是向对应 *socket* 发送信息。

时间戳介绍：4-5 行获取当前时间，并根据格式构建时间戳，第 6 行将时间戳与信息拼接

该模块实现接收要广播的信息（聊天信息或其他信息），并为其增加时间戳，完成拼接后对用户端列表进行遍历，发送到每一个用户端。

4. 对已有线程接收信息的处理

对已有线程接收信息的处理

```
1  //被调函数
2  void remove_client(SOCKET client_socket) {
3      lock_guard<mutex> lock(clients_mutex);
4      client_sockets.erase(remove(client_sockets.begin(), client_sockets.end(),
           client_socket), client_sockets.end());
5  }
6  //模块内容
7  void op_client(SOCKET client_socket) {
8      char buffer[BUF_SIZE];
9      char username[NAME_SIZE];
10     recv(client_socket, username, NAME_SIZE, 0);
11     num_client++;
```

```
12      string join_message=string(username) + "加入了聊天，当前共有"+to_string(
            num_client)+"人聊天。\n";
13      send_message_to_all(join_message, client_socket);
14      while (true) {
15          int receive=recv(client_socket, buffer, BUF_SIZE, 0);
16          if (receive> 0) {
17              buffer[receive]='\0';
18              string message = string(buffer);
19              if (message == "\\d") {
20                  num_client--;
21                  string exit_message=string(username)+ "退出了聊天，当前共有"+
                        to_string(num_client)+"人聊天。\n";
22                  send_message_to_all(exit_message, client_socket);
23                  remove_client(client_socket);
24                  closesocket(client_socket);
25                  break;
26              }
27              else send_message_to_all(message, client_socket);
28          }
29          else {
30              num_client--;
31              string exit_message=string(username)+ "断开了连接，当前共有"+
                    to_string(num_client)+"人聊天。\n";
32              send_message_to_all(exit_message, client_socket);
33              remove_client(client_socket);
34              closesocket(client_socket);
35              break;
36          }
37      }
38  }
```

函数 *recv* 解析：

第一个参数为通信的 *socket*，第二个参数为存放消息的缓冲区，第三个参数为最大可以接收的消息长度，第四个参数一般为 0，表示默认接收。

返回值是接收到的字节数，若小于等于 0 则为接收失败。

该函数的功能是从 SOCKET 接收数据，并保存在缓冲区中。

该模块是为新连接分配线程后的处理，首先会接收一次用户名的信息，完成一次打印和广播(10-13 行)。随后进入循环接收该连接发来的信息，若正常接收则进入 if 分支处理，判定该信息是否为退出信息。若非退出信息则正常打印和广播，否则进行退出信息广播，通过 $remove_client$ 实现线程安全的用户端 socket 移除数组操作，并关闭 socket 释放资源。若非正常接收则该用户端已经错误断开，操作与正常退出类似，不过采用了不同的输出便于观测。

(三) 用户端模块

1. 程序初始化及连接状态的切换

初始化及连接状态切换

```
cout<< "[欢迎使用Socket多人聊天程序--用户端 by 2211290姚知言] \n";
cout<< "输入你的用户名: ";
getline(cin, username);
cout<<"欢迎"<<username<<"!\n";
cout << "命令列表(在未连接状态下):\n"<< "\\c :输入服务器IP地址并连接\n"<< "
    \\r :更改用户名\n"<< "\\q :退出程序\n";
while (true) {
    getline(cin, input);
    if (!is_connected) {
        if (input=="\\c") {
            cout << "请输入服务器IP地址(默认为127.0.0.1, 若无需更改, 可直接回
                车): ";
            getline(cin, input);
            if(input=="") input="127.0.0.1";
            connect(input);
        }
        else if (input == "\\r") {
            cout << "请输入新的用户名: ";
            getline(cin, username);
            cout<<"欢迎"<<username<<"!\n";
        }
        else if (input == "\\q") {
            std::cout << "程序将在5s后退出, 感谢你的使用! \n";
            Sleep(5000);
            break;
        }
        else cout << "命令列表(在未连接状态下):\n"<< "\\c :输入服务器IP地址
            并连接\n"<< "\\r :更改用户名\n"<< "\\q :退出程序\n";
    }
    else {
        if (input == "\\d") {
            string message= "\\d";
            send(u_socket, message.c_str(), message.size(), 0);
            is_connected = false;
            cout << "已退出聊天! \n";
            cout << "命令列表(在未连接状态下):\n"<< "\\c :输入服务器IP地址并
                连接\n"<< "\\r :更改用户名\n"<< "\\q :退出程序\n";
        }
        else{
            string message = username + ": " + input + "\n";
            send(u_socket, message.c_str(), message.size(), 0);
        }
    }
}
```

该模块通过 bool 类型变量 is_connected 定义了用户连接状态。

程序启动时，会接受一次用户名输入。随后打印一次命令列表。

在未连接状态下，输入 \c 可以输入 IP 地址并进行连接（9-14 行，调用自己设计的 connect

函数，函数重载，可通过参数数量和类型区分通信 connect 函数，后续展示），输入 \r 可以更改用户名（15-19 行），输入 \q 可以退出程序（20-24 行），否则为用户重新打印命令列表提示。

在连接状态下，可以直接键入发送信息，或输入 \d 退出聊天。对于不同的输入调用不同的 send 函数，即若非退出聊天信息，则发送带有用户名的信息到服务器端中，否则仅发送 \d。

2. 连接处理

连接处理

```
1  void connect(string &ip) {
2      struct sockaddr_in addr;
3      u_socket = socket(PF_INET, SOCK_STREAM, 0);
4      addr.sin_family = AF_INET;
5      addr.sin_port = htons(4444);
6      addr.sin_addr.s_addr = inet_addr(ip.c_str());
7      if (!connect(u_socket, (struct sockaddr *)&addr, sizeof(addr))) {
8          cout << "\033[34m 服务器" << ip << "连接成功! \033[0m \n";
9          cout <<"在连接状态下, 输入\\d即可断开连接。\n";
10         send(u_socket, username.c_str(), username.size(), 0);
11         is_connected = true;
12
13         thread recv_thread(receive, u_socket);
14         recv_thread.detach();
15     }
16     else cout << "\033[31m 无法连接服务器 " << ip << " !\033[0m \n";
17 }
```

(WINSOCK 库) 函数 *connect* 解析：

第一个参数为用户端 SOCKET，第二个参数为存储服务器端地址信息的结构体指针，第三个参数为结构体的大小。

函数的作用是向服务器端发送连接请求，返回值为 0 表示成功。

整体模块的作用是当用户希望对服务器进行连接时，建立 socket，根据用户给出的 ip 地址等信息，构建 sockaddr_in 结构体，并调用 *connect* 库函数尝试连接，若连接失败则输入无法连接提示，否则提示连接成功，向服务器端发送用户名信息，切换连接状态，并开辟新线程调用 receive 函数（下一部分介绍）处理从服务器端发送的接收信息，同样分离线程到后台保证主线程操作。

3. 广播信息接收处理

信息接收处理

```
1  void receive(SOCKET u_socket) {
2      char buffer[BUF_SIZE];
3      while (is_connected) {
4          int receive = recv(u_socket, buffer, BUF_SIZE, 0);
5          if (receive > 0) {
6              buffer[receive] = '\0';
7              cout << buffer;
8          }
```

```
        else break;
    }
}
```

该模块的作用是在接收状态下，持续从服务器端接收信息，并将其打印出来。被上一模块调用，由开辟的后台线程处理。

四、 运行环境

程序在 Windows 系统，通过 g++ 编译器编译为可执行文件。

指令如下:

编译为可执行文件

```
g++ client.cpp -o client -lws2_32
g++ server.cpp -o server -lws2_32
```

指令中的 $-lws2\_32$ 的作用是告诉链接器（linker）在编译过程中链接 $ws2\_32$ 库。其指向 Winsock 2 库文件的 $ws2\_32.lib$，该库文件包含了网络通信所需的函数实现。

五、 程序运行演示

(一) 连接失败

图3展示在连接失败的情况下（本次模拟的情况为不开启服务器端），正确返回并能够正常进行后续操作。

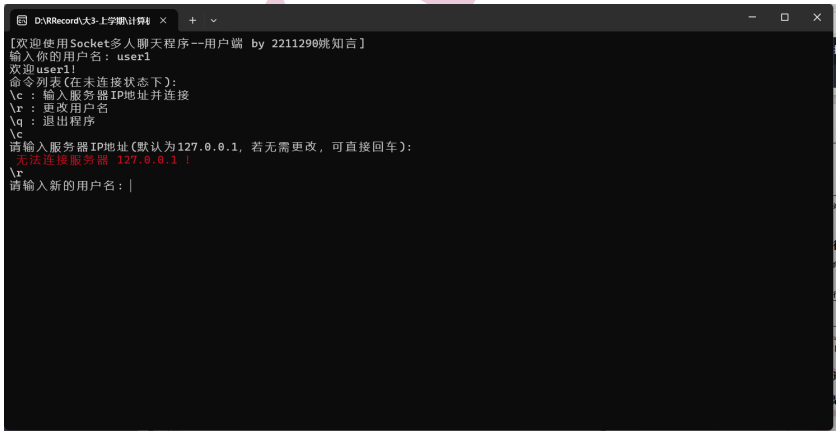

图 3: 连接失败

(二) 正常连接/反复连接展示

Step1：用户端命名 user1，并进行连接后，输出如图4所示。

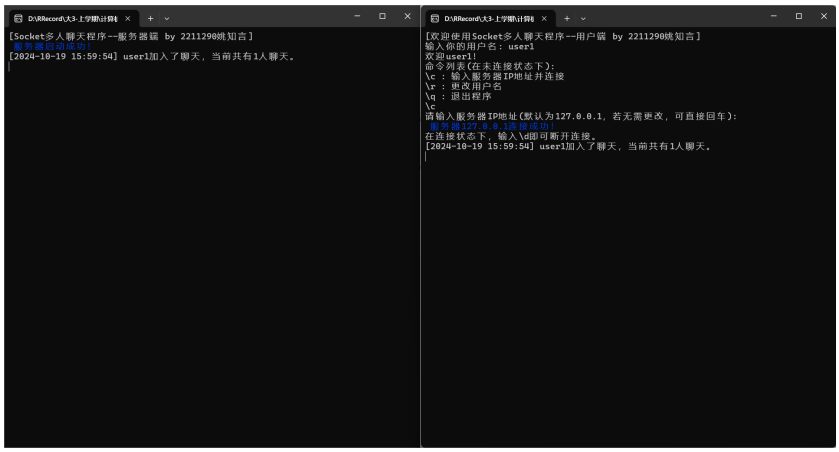

图 4: Step1

Step2：user1 发送”hello world”，随后正常断开连接，输出如图5所示。

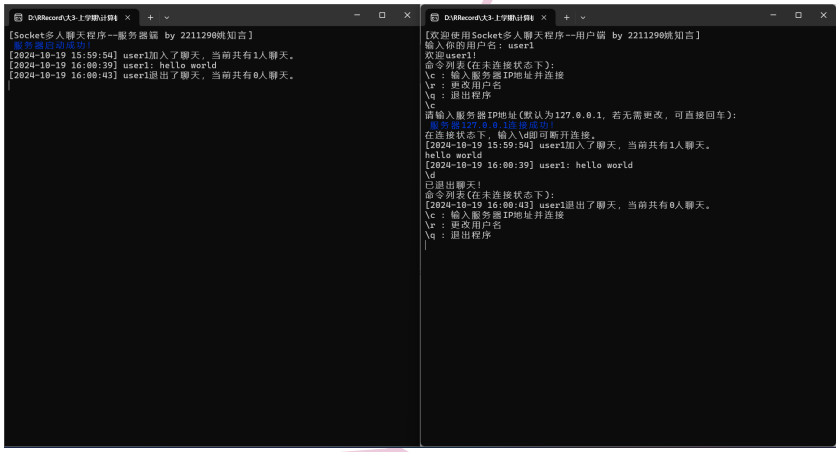

图 5: Step2

Step3:user1 更名为 user2，重新连接，发送”你好”，随后断开连接，退出程序，输出如图6所示。

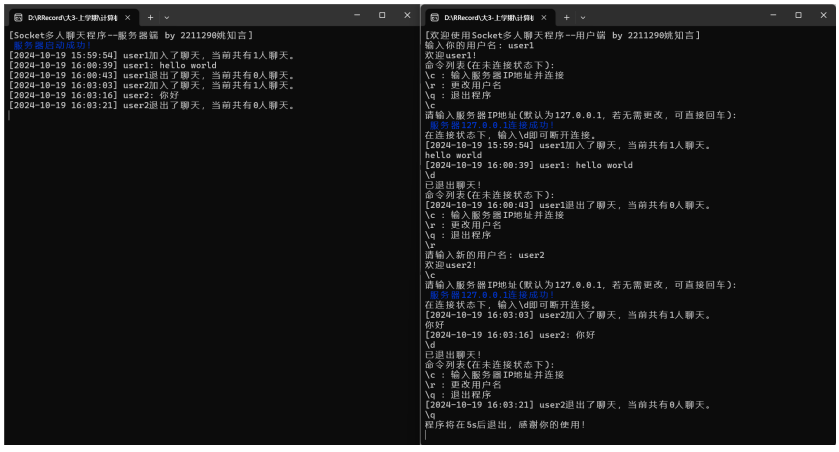

图 6: Step3

(三) 多方连接展示

Step1:test1,test2,test3 进行连接。其中 test1 采用输入 ip 地址的方式，其他采用直接回车的方式。正确输出，如图7所示。

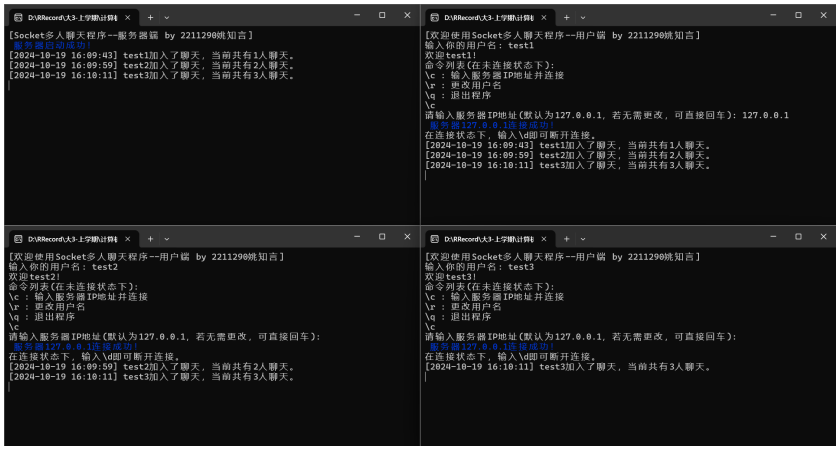

图 7: Step1

Step2:test1 发送‘111’，test2 发送‘欢迎’，能够正常传输和输出，如图8所示。

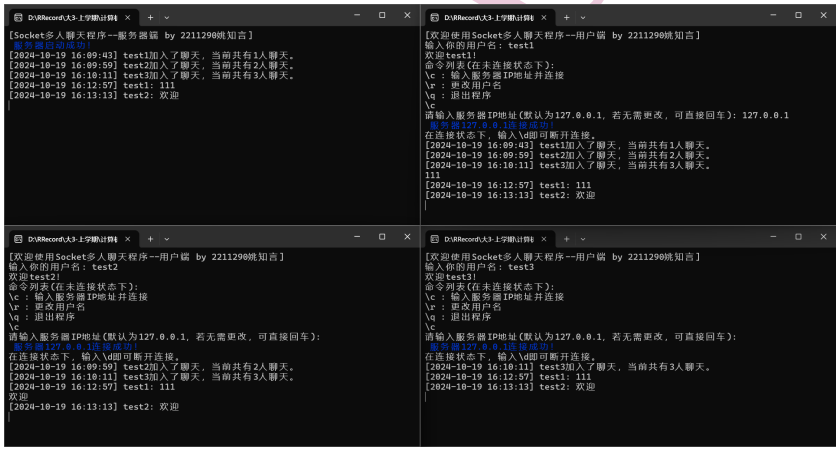

图 8: Step2

Step3:test2 退出，后 test3 发送‘222’，test1 接收到，test2 未接收到。正确显示退出信息和断开 test2 连接，如图9所示。

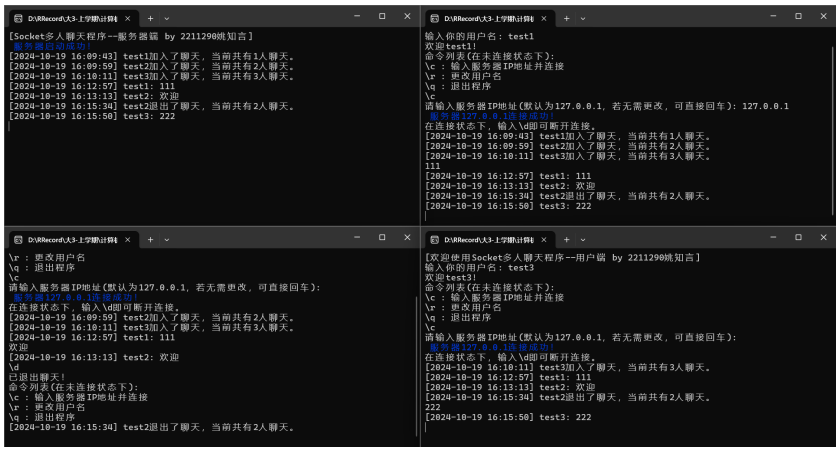

图 9: Step3

Step4: 直接终止 test3 终端运行（模拟异常退出），test1 和 server 可以正常接收，test2 因之前已经退出接收不到信息。如图10所示。

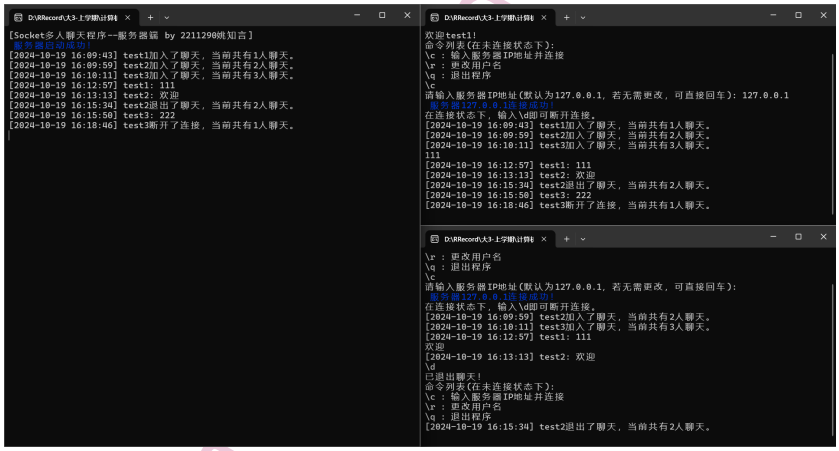

图 10: Step4

六、 问题分析

1. recv 接收含有冗余信息

   在起初进行实验时，程序往往输入冗余信息，即上一次 recv 接收到的信息。经过分析，我发现这是在一些情况下，接收的字符串只会对接收部分进行更新，且并不会设置合理的结束'\0'，导致后续读取内容超过我希望的内容。

   后来，我发现 recv 的返回值是接收到的字符个数，通过这一参数，我能够在 buffer 的合理位置添加'\0' 保证后续输出和处理的正确性。即:

解决方案

```
int receive=recv(client_socket, buffer, BUF_SIZE, 0);
buffer[receive]='\0';
```

2. thread 线程问题

起初，我未能理解如何保证服务器端同时进行 accept 和数据的接收，客户端如何同时保证发送和接收等问题。但经过对 thread 库的学习，我认识到通过开辟新线程和分离线程可结果该问题。