南开大学
Nankai University

南　　开　　大　　学

网 络 空 间 安 全 学 院

计算机网络实验报告

简单的聊天协议设计

2014074 费泽锟

年级：2020 级

专业：信息安全

指导教师：徐敬东、张建忠

2022 年 10 月 22 日

摘要

对于本次简单聊天协议的设计，主要实现了使用流式 Socket 与多线程，通过服务端对连接该服务端且在线的用户进行实时的消息转发，达到多人在线聊天的功能。在服务端部分会记录多人聊天的 Log 信息，如果出现 Socket 连接使用时的错误，客户端会向用户报错并展示 error 信息。

通过在不同地点以及不同机器上开启服务端与多个客户端进行实验，可以证明可执行文件的正确性和可用性。

关键字：Socket，服务端，客户端，TCP/UDP，多线程

目录

一、 协议设计与代码实现

(一) 聊天协议设计

要想实现简单聊天程序的服务端与客户端的编写，首先就是要对聊天协议、输入输出的格式以及网络传输结构进行合适的设计。

1. 消息的类型

   对于消息类型的设计部分，可以确定的是在不同的客户端与服务端之间使用 Socket 进行数据传输时的参数是 char* 类型的变量。在本次的设计之中，并没有设计信息在客户端传输到服务端的过程之中再携带其余的附带信息，而是仅仅传输各个用户发送的原始消息。但是在客户端进行多个用户消息转发的过程之中，会在各个用户发送的消息之前，添加“用户_***_:”的字样，以表示多人群聊过程之中的消息是谁发送的。

   具体代码如下:

   服务端对用户消息的处理

```
strcpy_s(sendBuf, "用户--");
string ClientID = to_string(ClientSocket);
strcat_s(sendBuf, ClientID.data()); // data函数直接转换为char*
strcat_s(sendBuf, "--: ");
strcat_s(sendBuf, recvBuf);
// message_queue.push(sendBuf); //这里将消息存储到队列中
```

   接下来展示在服务端处理完消息之后，客户端接受消息的展示:

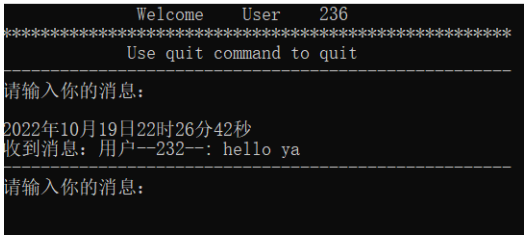


图 1: 用户接受到其他用户的消息

2. 消息的语法

   对于消息语法的设计部分，这里采用了宏定义的形式，定义了每次用户发送消息的最大长度为 100 个字符，并且为了能够提升用户的体验，在本次设计中采用了使用 getline 函数来获取用户的输入，使用该函数，用户可以在想发送的消息之中添加空格，并且该函数会自动识别换行符为输入的结束。

   具体代码如下:

   消息的语法设计

```
#define DEFAULT_BUFLEN 100

char sendBuf[DEFAULT_BUFLEN] = "";
cout << "请输入你的消息: ";
cin.getline(sendBuf, DEFAULT_BUFLEN);
// 保证可以输入空格, getline函数设置好了以换行符为结束
```

3. 消息的语义

对于消息语义的设计部分，这里采用的是在每次用户端发送或接受消息之后，输出相应的提示信息来指示具体的发送或接受的语义，在每次发送或输出消息之后都会进行相同格式信息的展示。在客户端输入 quit 字符串，即可退出与服务端的连接。

具体代码如下:

消息的语义展示

```
    cout <<endl << endl << systime.wYear << "年" << systime.wMonth << "月" <<
         systime.wDay << "日";
    cout << systime.wHour << "时" << systime.wMinute << "分" << systime.
        wSecond << "秒" << endl;
    cout << "消息已成功发送" << endl;
    cout << "------------------------------------------------------" << endl;

    cout << endl << endl << systime.wYear << "年" << systime.wMonth << "月"
        << systime.wDay << "日";
    cout << systime.wHour << "时" << systime.wMinute << "分" << systime.
        wSecond << "秒" << endl;
    cout << "收到消息: ";
    cout << recvbuf << endl;
    cout << "------------------------------------------------------" << endl;
    cout << "请输入你的消息: "; // 提示可以继续发送消息了
```

4. 消息的时序

对于消息时序的设计部分，本次设计尝试使用了消息队列，来处理用户列表发送的各个消息数据，但是考虑到目前只是基本的聊天设计，不会出现高并发的情形，而且如果使用消息队列，需要标记消息的发送者信息，会占据一定的内存，所以本次设计最终没有选择消息队列，而是使用了 map 类型的数据结构，存储所有的用户信息（无论是在线抑或是不在线的用户），并且每次当服务端接受到不同用户端的消息后，都会在子线程之中立即进行消息的转发工作。

而对于发送和接受消息时的时间展示，均采用本机的 Local 时间，而不进一步采取在数据包传输之中携带时间信息。

具体转发部分的代码:

服务端对多用户消息转发

```
for (auto it : user_map) {
    if (it.first != ClientSocket && it.second == 1) {
        sendResult = send(it.first, sendBuf, DEFAULT_BUFLEN, 0);
        if(sendResult == SOCKET_ERROR)
            cout << "send failed with error: " << WSAGetLastError() << endl;
    }
}
```

(二) 客户端设计

对于客户端的设计部分，在学习了课程讲授的 Socket 编程基本内容后，使用了 Socket 中的 connect、recv 和 send 等函数实现客户端的功能。

在客户端的设计过程之中，我们可以发现如果只使用一次 recv 函数，那么客户端只能接受一次服务端的数据，所以我们这里应该使用 while 循环来持续进行消息的 recv 和 send 操作直到用户希望退出为止。但是如果仅仅用两个 while 循环，也是不合理的。

因为如果仅仅使用两个 while 循环进行处理，那么我们只能选择在 recv 结束之后进行 send 或者是在 send 过程结束之后进行 recv，这显然是不利于用户使用的，所以在本次设计之中，选择在客户端设计两个线程，一个为 send 发送消息线程，另一个为 recv 接受消息线程，直到用户输入 quit 字符串或者退出程序为结束。

需要注意的是，在使用 CreateThread 函数进行传参时，我们需要传入一个指向 ConnectSocket 的 Socket 指针变量，因为我们在客户端只有 ConnectSocket 和服务端进行了连接操作，也可以换一种方式，那就是在两个线程之中都进行创建 Socket 并进行 connect 连接的操作，但这不是很符合讲授的知识内容，所以没有采用。

具体的客户端部分代码如下:

客户端双线程

```
    recv(ConnectSocket, user_name, 10, 0);

    // 打印进入聊天的标志
    cout << "              Welcome    User    " << user_name << endl;
    cout << "*****************************************************" << endl;
    cout << "             Use quit command to quit" << endl;
    cout << "-----------------------------------------------------" << endl;

    //————————————————————
    // 创建两个线程，一个接受线程，一个发送线程
    HANDLE hThread[2];
    hThread[0] = CreateThread(NULL, 0, Recv, (LPVOID)&ConnectSocket, 0, NULL)
        ;
    hThread[1] = CreateThread(NULL, 0, Send, (LPVOID)&ConnectSocket, 0, NULL)
        ;

    WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
    CloseHandle(hThread[0]);
    CloseHandle(hThread[1]);

    // 关闭socket
    iResult = closesocket(ConnectSocket);
    WSACleanup();
    return 0;
```

Send：用户发送消息的线程函数:

Send 线程函数

```
DWORD WINAPI Send(LPVOID lparam_socket) {

```

```
    // 接受消息直到quit退出聊天
    // flag为是否退出聊天的标志
    int sendResult;
    SOCKET* sendSocket = (SOCKET*)lparam_socket;

    while (1)
    {
        //--------------------
        // 发送消息
        char sendBuf[DEFAULT_BUFLEN] = "";
        cout << "请输入你的消息: ";
        cin.getline(sendBuf, DEFAULT_BUFLEN);    // 保证可以输入空格, getline
            函数设置好了以换行符为结束

        if (string(sendBuf) == quit_string) {
            flag = 0;
            closesocket(*sendSocket);
            cout << endl << "即将退出聊天" << endl;
            return 1;
        }
        else {
            sendResult = send(*sendSocket, sendBuf, DEFAULT_BUFLEN, 0);
            if (sendResult == SOCKET_ERROR) {
                cout << "send failed with error: " << WSAGetLastError() <<
                    endl;
                closesocket(*sendSocket);
                WSACleanup();
                return 1;
            }
            else {
                SYSTEMTIME systime = { 0 };
                GetLocalTime(&systime);
                cout <<endl << endl << systime.wYear << "年" << systime.
                    wMonth << "月" << systime.wDay << "日";
                cout << systime.wHour << "时" << systime.wMinute << "分" <<
                    systime.wSecond << "秒" << endl;
                cout << "消息已成功发送" << endl;
                cout << "
                    ----------------------------------------------------" <<
                     endl;
            }
        }
    }
}
```

在 Send 线程之中，每次用户发送消息或者接收消息之后，均会出现“请输入你的消息:”的提示字样，当用户想要退出连接，输入 quit 字符串后，会出现“即将离开聊天”的提示信息，无论消息发送成功或失败都会有相应的提示信息。

Recv 线程函数

```
DWORD WINAPI Recv(LPVOID lparam_socket) {
    int recvResult;
    SOCKET* recvSocket = (SOCKET*)lparam_socket; //一定要使用指针型变量，因为要指向connect socket的位置

    while (1) {
        char recvbuf[DEFAULT_BUFLEN] = "";
        recvResult = recv(*recvSocket, recvbuf, DEFAULT_BUFLEN, 0);
        if (recvResult > 0 && flag == 1) {
            SYSTEMTIME systime = { 0 };
            GetLocalTime(&systime);
            cout << endl << endl << systime.wYear << "年" << systime.wMonth << "月" << systime.wDay << "日";
            cout << systime.wHour << "时" << systime.wMinute << "分" << systime.wSecond << "秒" << endl;
            cout << "收到消息: ";
            cout << recvbuf << endl;
            cout << "----------------------------------------------------" << endl;
            cout << "请输入你的消息: "; // 提示可以继续发送消息了
        }
        else {
            closesocket(*recvSocket);
            return 1;
        }
    }
}
```

Recv 线程函数会一直接受来自服务端的消息，当接受完消息之后会有“可以继续发送消息”的提示字样，当程序被人为结束或者用户输入 quit 字符串后结束该线程，设置 flag 为全局变量，利于 Send 函数和 Recv 函数使用。

(三) 服务端设计

对于服务端的设计部分，在服务端的设计之中，因为服务端只是在接收到客户端发的消息之后，再进行消息到其余客户端的转发的操作，所以我们只需要一个 handlerRequest 线程函数就能实现接受与转发的功能了。

在服务端的设计之中，我还设计了一个 map 类型的数据结构，用来保存用户的 Socket 和对应的状态，使用一个 int 类型的变量来标识用户是否在线的状态，为所有在线的用户转发各个用户的消息。

除了这些功能外，为了标识不同连接到服务端的客户端，在本次设计之中，使用了在接受 Socket 创建之后返回的 int 类型的数值来对不同的客户端进行标识，并在初次连接成功后，在客户端 while 循环接受消息之前，先发送该服务端分配的用户标识信息。

如果在群聊的过程之中,你想要对单独的特殊的用户发送消息,那么你可以使用 username|message 的格式进行发送，这样你就可以实现在群聊的过程之中私聊用户的操作了。

除此之外，为了能够实现简单的通讯，即不同的客户端在不同的地点的通讯，这里对于服务

端设置的 IP 地址为 WLAN 的 IPV4 地址，这样的话如果在本机上打开服务端，就可以在该网段下实现多用户群聊功能了，而在服务端中也能看到具体的格式化的 Log 信息打印。但是因为 WLAN 的 IP 地址是经常变化的，所以需要自搜索 IP 地址，这部分需要注意。

在本次设计的展示中，服务端的设计还是写定了 IP 地址和端口号，设置恒定打开的端口为 27015 端口。

对于实现中最关键的 handlerRequest 函数，展示如下：

handlerRequest 函数实现

```
// 为每一个连接到此端口的用户创建一个线程
DWORD WINAPI handlerRequest(LPVOID lparam)
{
        SOCKET ClientSocket = (SOCKET)(LPVOID)lparam;
    user_map[ClientSocket] = 1;

    char user_name[10];
    strcpy_s(user_name, to_string(ClientSocket).data());
    send(ClientSocket, user_name, 10, 0);

    SYSTEMTIME systime = { 0 };
    GetLocalTime(&systime);
    cout << endl << systime.wYear << "年" << systime.wMonth << "月" <<
        systime.wDay << "日";
    cout << systime.wHour << "时" << systime.wMinute << "分" << systime.
        wSecond << "秒" << endl;
    cout << "Log: 用户--" << ClientSocket << "--加入聊天! " << endl;
    cout << "-----------------------------------------------------" << endl;

    // 循环接受客户端数据
    int recvResult;
    int sendResult;
    int flag = 1; // 控制是否退出该Socket的接受循环
    int judge_flag = 1; // 默认群发
    do{
        char recvBuf[DEFAULT_BUFLEN] = "";
        char sendBuf[DEFAULT_BUFLEN] = "";
        char special_message[DEFAULT_BUFLEN + 50] = "";
        recvResult = recv(ClientSocket, recvBuf, DEFAULT_BUFLEN, 0);
        judge_flag = 1;
        if (recvResult > 0) {
            char special_user_name[10] = "";
            for (int i = 0; i < DEFAULT_BUFLEN; i++) {
                if (recvBuf[i] == '|') {
                    judge_flag = 0;
                }
            }

            // temp_flag为0表明要单独发送，判断是否要单发
            if (judge_flag == 0) {
```

```
39                  int j;
40                  for (j = 0; j < DEFAULT_BUFLEN; j++) {
41                      if (recvBuf[j] == '|') {
42                          special_user_name[j] = '\0';
43                          for (int z = j + 1; z < DEFAULT_BUFLEN; z++) {
44                              special_message[z - j - 1] = recvBuf[z];
45                          }
46                          strcat_s(special_message, "( from ");
47                          strcat_s(special_message, to_string(ClientSocket).
                                data());
48                          strcat_s(special_message, ")\0");
49                          break;
50                      }
51                      else {
52                          special_user_name[j] = recvBuf[j];
53                      }
54                  }
55              }
56
57              strcpy_s(sendBuf, "用户--");
58              string ClientID = to_string(ClientSocket);
59              strcat_s(sendBuf, ClientID.data()); // data函数直接转换为char*
60              strcat_s(sendBuf, "--: ");
61              strcat_s(sendBuf, recvBuf);
62              // message_queue.push(sendBuf); //这里将消息存储到队列中
63
64              SYSTEMTIME Logtime = { 0 };
65              GetLocalTime(&Logtime);
66              cout << endl << Logtime.wYear << "年" << Logtime.wMonth << "月"
                    << Logtime.wDay << "日";
67              cout << Logtime.wHour << "时" << Logtime.wMinute << "分" <<
                    Logtime.wSecond << "秒" << endl;
68              cout << "Log: 用户--" << ClientSocket << "--的消息: " << recvBuf
                    << endl;
69              cout << "-----------------------------------------------------"
                    << endl;
70
71              if (judge_flag == 1) {
72                  for (auto it : user_map) {
73                      if (it.first != ClientSocket && it.second == 1) {
74                          sendResult = send(it.first, sendBuf, DEFAULT_BUFLEN,
                                0);
75                          if (sendResult == SOCKET_ERROR)
76                              cout << "send failed with error: " <<
                                    WSAGetLastError() << endl;
77                      }
78                  }
79              }
```

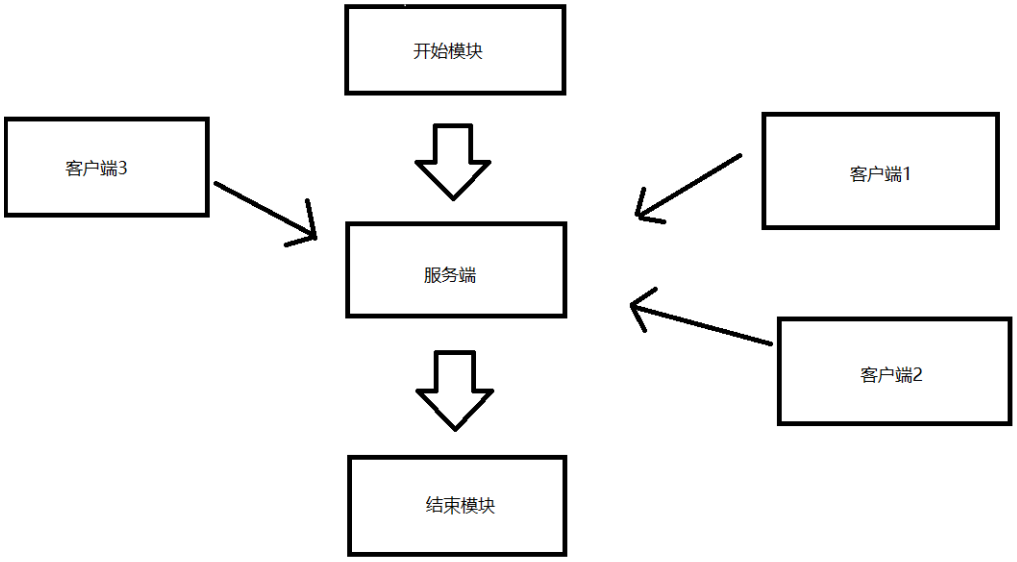
```
            else {
                for (auto it : user_map) {
                    if (to_string(it.first) == string(special_user_name) &&
                        it.second == 1) {
                        sendResult = send(it.first, special_message,
                            DEFAULT_BUFLEN, 0);
                        if (sendResult == SOCKET_ERROR)
                            cout << "send failed with error: " <<
                                WSAGetLastError() << endl;
                    }
                }
            }
        }
        else {
            flag = 0;
        }
    } while (recvResult != SOCKET_ERROR && flag != 0);

    GetLocalTime(&systime);
    cout << endl << systime.wYear << "年" << systime.wMonth << "月" <<
        systime.wDay << "日";
    cout << systime.wHour << "时" << systime.wMinute << "分" << systime.
        wSecond << "秒" << endl;
    cout << "Log: 用户--" << ClientSocket << "--离开了聊天(quq)" << endl;
    cout << "-----------------------------------------------------" << endl;

    closesocket(ClientSocket);
        return 0;
}
```

主要需要注意的点就是使用了 auto iter 迭代器来遍历 map 数据结构的用户列表，以及在这里使用 flag 变量判断用户是否退出，使用 temp_flag 变量判断是否这次的消息需要私发给特定的用户。

（四） 模块功能与模块流程图

1. 开始模块：初始化 Socket，IP 以及 Port 等内容，进行连接 connect，listen 等操作，等待客户端连接服务端。
2. 服务端模块：转发用户消息至所有的在线用户，并且在服务端留下 Log 信息, 如果在消息中添加 user_name 字段则可以实现单独发送（使用‘|’标识符)。
3. 客户端模块：可以持续接受来自服务端的消息，可以持续发送消息。
4. 结束模块：用户输入 quit 字符串表征退出连接，服务端断连。如果想要关闭服务端，则需要手动关闭。

具体的流程图展示如下：



上图仅仅展示的是宏观的各个模块的运行流程，具体的 Socket 的连接和发送流程，可参照下图：

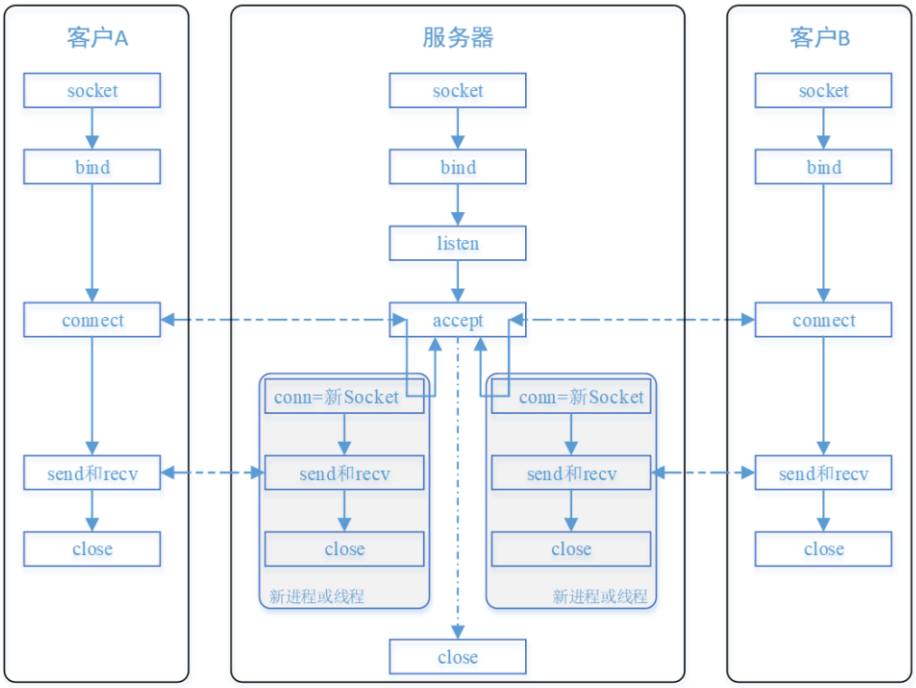


图 2: 流式 Socket 的连接流程

也体现了开始模块和结束模块的具体流程。

二、 实验验证与总结

实验验证部分主要是为了验证 Release 版本的可执行文件的逻辑正确，这里展示的是在本机上打开三个客户端互相发送群聊消息以及私聊消息的效果展示，也会展示相应的服务端的 Log

日志记录。

在本次实验中，还尝试了将客户端可执行文件发送到其他同学的计算机上运行，发现可以实现从宿舍到公教楼的通讯，并且可以实现多人通讯，但是很明显的是，消息的传输过程之中有延迟的现象。

在本次设计之中还有问题与未完成的部分，首先就是如果出现了高并发的情形，可能需要消息队列进行处理，这一部分尚未完成。其次就是延迟问题，显然这一部分还不知道咋处理。最后就是如果在用户端输入消息的过程之中，接收到了消息，那么因为是命令行展示的形式，而不是 UI 界面展示的形式，所以输入过程会被打断，但是输入的内容已经被存储在了 sendBuf 之中了，只需要继续输入想要输入的内容即可，解决该问题只需要编写合适的 UI 界面即可解决该问题了。

客户端效果图：

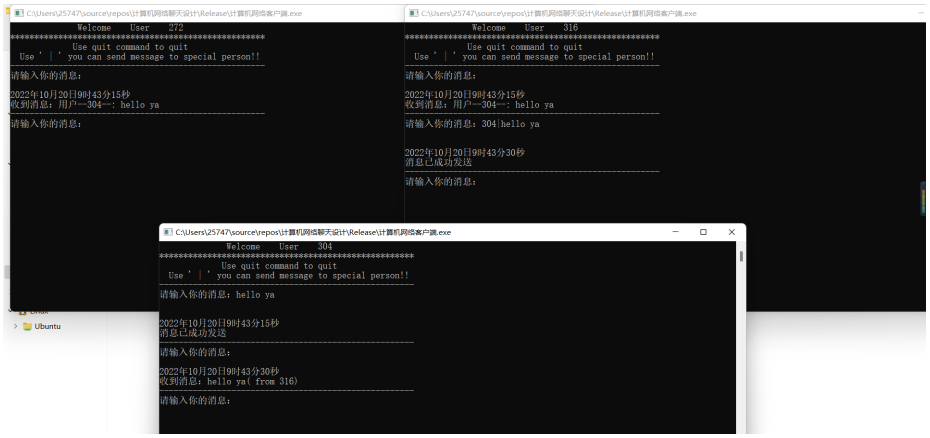


图 3: 客户端效果图

服务端效果图：

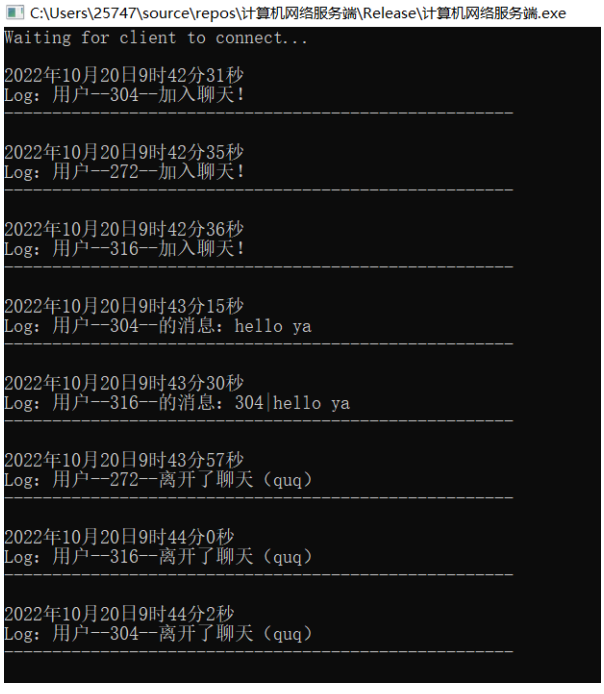


图 4: 服务端效果图

参考文献

https://blog.csdn.net/u011416077/article/details/123593428
https://www.yisu.com/zixun/740954.html
https://blog.csdn.net/li_wen01/article/details/52665505
https://cloud.tencent.com/developer/article/2098955
计算机网络/02-计算机网络-第二章-2022（第一部分）-20220924-带问题.pdf