

# 计算机网络实验一报告

学号：2210737 姓名：阿斯雅

## 一、协议设计

我们知道网络协议的三个要素为：

1. 语法
2. 语义
3. 时序

也可以形象地把这三个要素描述为：语义表示要做什么，语法表示要怎么做，时序表示做的顺序。

而我们本次要做的是一个多人聊天程序。所以我选用客户端/服务端进程通信模型，具体说的话就是首先需要设计一个服务器端，它的主要作用就是接受客户端的发来的消息，然后再同步到各个客户端，相应的，客户端需要做的就是给服务器端发送消息然后接受服务器端发来的消息并输出。这样就可以间接的实现多个客户端也是多个进程之间的相互通信。因为本次实验要求的是使用流式socket,也就是使用TCP传输层协议，支持主机之间的面向连接的，顺序的，可靠的，全双工字节流传输。因为TCP协议已经广为人知，所以就不再这里进行介绍。我想就此次作业来介绍我的协议设计。

- 消息类型

首先因为应用程序是通过交换消息来通讯的，而我选的通讯模型是C/S模型，所以在我的实验里有两种消息类型：一个是客户端向服务端发送的聊天消息，另一个则是服务端为了同步消息而向客户端发送的信息。

其中客户端向服务端发送的消息有两种类型：一个是发送正常消息。另外一个则是发送要退出房间的消息“END”。

服务端发送给客户端的消息也有四种类型：一种消息类型是用在当房间有新用户加入，其消息的格式为：

客户【客户标号】已加入房间，当前客户数量为【客户数量】

第二种消息类型是用在同步消息的时候，其消息格式为：

客户标号【客户发的消息】--[年]-[月]-[日]--[时]:[分]:[秒]

第三种消息类型是用在当有用户需要退出房间时候，其消息格式为：

客户【客户标号】已退出房间，当前客户数量为【客户数量】

最后一种消息格式是用在当有排队的用户进到房间时候，其消息格式为：

排队客户【客户标号】已进入房间，当前客户数量为【客户数量】

- C/S对消息的动作

在确定好消息类型后需要定义不同角色对消息的动作：对客户端来说，它对消息需要两个动作，一个是需要发送自己的聊天内容，即发送消息。不能光自己发消息，也需要看到别人发的聊天内容。所以它的另一个动作就是接收服务器端发来的同步消息，并且输出到自己的终端上，这样就可以实现各个用户能看到彼此发的聊天内容。但如果它从服务端收到的消息是满员的消息的话，客户端就得要进入排队队列中进行等待，等有某个在线用于退出，再进入房间。而对于服务端来说，它对消息也需要两个动作，一个是需要接收客户端发来的消息，也就是聊天信息，并且把它保存到一个缓冲区里面。各个用户同步消

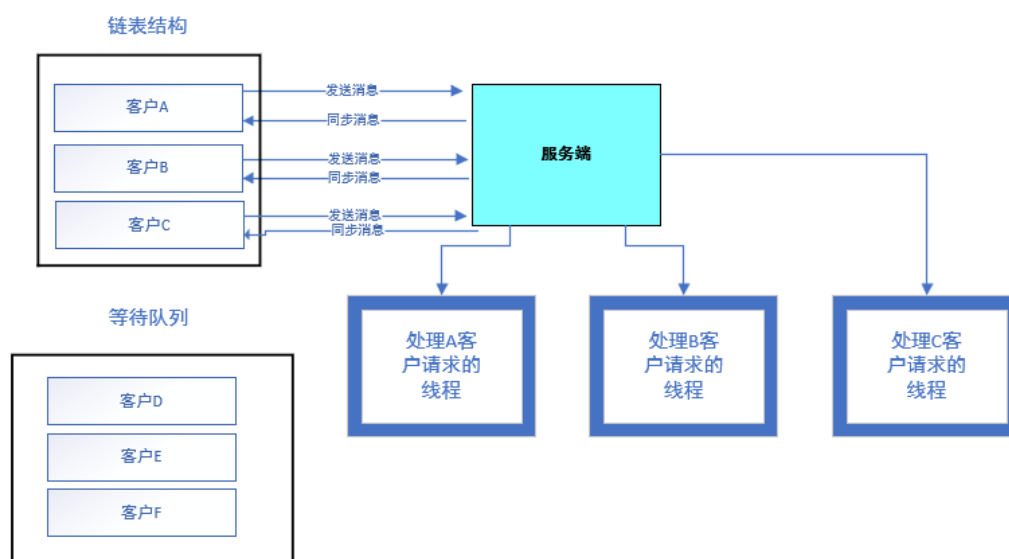
息的功能是通过服务器端来完成的，所以服务器端对消息的另一个动作就是需要给在线的每个用户转发收到的消息，即缓冲区里面的内容。

- C/S时序

为了保证消息的正确输出，接下来需要设计好服务器端和客户端的时序。首先对于服务器端来说，因为它要监听不同客户的连接请求，所以服务器端要一开始就要运行，并且要在指定的端口 8000 和 IP 地址 127.0.0.1 上等待客户端的连接请求，使用 TCP 的流式通信来接收客户端连接。每当一个客户端发送消息时，服务端会立即将消息接收存入一个缓冲区并建立一个独属于当前客户的线程。而当某个客户端需要断开连接时，服务端需要实时更新当前连接的客户端列表，确保不再向已断开连接的客户端转发消息。对于客户端来说，在服务端已经开启的情况下，客户端首先通过 TCP 协议连接到服务器端，建立通信通道。而用户在客户端输入消息后，消息会通过该通道发送到服务器端。与此同时客户端也要持续监听来自服务器端的消息。每当服务器端转发其他用户的消息时，客户端会接收到，并在用户终端上输出显示。当用户需要退出时，客户端会向服务器发送一个关闭连接的信号，然后关闭与服务器的通信通道。

- 错误处理

在聊天过程中难免会出现一些错误或者异常，比如，用户退出聊天室，聊天室已经满员等等。所以有必要做一些错误处理。在客户端可能无法连接到服务器情况下，客户端应提示用户连接错误。若在通信过程中网络中断，客户端和服务器都需要处理连接断开的情况，服务器应及时更新在线用户列表，并停止向已断开连接的客户端转发消息。虽然 TCP 协议本身保证了消息的可靠传输，但如果客户端或服务端在处理消息时发生意外，程序应当能及时检测并进行相应的恢复操作。而在房间满员的情况下，服务端应要把多出来的客户放在排队队列里面，等房间有用户退出来的时候应该把等待队列中的客户转到房间里。



## 二、代码解释

整体代码步骤可以分为以下几步：

- 客户端
  1. 导入必要的头文件
  2. 定义需要的全局变量
  3. 初始化客户端套接字和服务端地址
  4. 连接服务器并创建线程
  5. 发送消息并接收服务器发来的消息
  6. 关闭客户端套接字

- 服务器端
  1. 导入必要的头文件
  2. 定义需要的全局变量
  3. 初始化服务器端套接字和服务端地址
  4. 监听请求
  5. 接受请求并创建专属线程
  6. 同步消息
  7. 关闭服务器端套接字

接下来我将根据代码进行介绍。

- 客户端代码

首先，我们需要导入必要的头文件，如 `<winsock2.h>`，`<ws2tcpip.h>`，也需要链接 `ws2_32.lib` 库。导入这些库文件的原因是在Windows平台下开发网络应用程序时，需要导入特定的头文件和库来使用网络编程接口（即 Winsock API），就以 `WinSock2.h` 举例，它提供了用于创建和管理Socket的函数，如 `socket()`、`bind()`、`listen()`、`accept()`、`send()`、`recv()` 等基本socket函数，允许程序在网络上传输数据。而 `ws2_32.lib` 是 winsock 库的静态链接库，它实现了头文件中声明的函数。编写网络程序时，需要在编译时链接这个库，以使用 Winsock API 提供的功能。

```
#include <iostream>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <cstring>
#include <windows.h> // 用于 sleep 函数
#pragma comment(lib, "ws2_32.lib") // ws2_32.lib库
using namespace std;
```

之后在客户端里面需要定义一些全局变量，如消息缓冲区大小，客户端套接字以及要服务器端的地址。

```
#define SIZE 4096
SOCKET clientSocket; // 定义客户端socket
SOCKADDR_IN servAddr; // 定义服务器地址
#define _WINSOCK_DEPRECATED_NO_WARNINGS // 解决inet_pton错误
```

接着就是我们的主函数，在客户端主函数里面首先通过 `WSAStartup` 基本函数来初始化 winsock 库，紧接着使用 `socket` 函数来初始化客户端套接字，其中三个参数的含义分别是地址类型使用 IPV4 格式，并且使用流式套接字 TCP 协议。

```
// 初始化winsock
WSADATA wsaData;
WSAStartup(MAKEWORD(2, 2), &wsaData);

// 初始化客户端socket
clientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

之后需要初始化服务端的地址，因为服务端地址 `serveraddr` 我用的是 `SOCKADDR_IN` 结构体，所以有三个成员需要初始化，分别是：`sin_family`、`sin_port` 和 `sin_addr`。代表着地址类型，端口号和 IP 地址。

```
// 初始化服务端地址
servAddr.sin_family = AF_INET;           // 地址类型IPv4
servAddr.sin_port = htons(8000);         // 转换成网络字节序
inet_pton(AF_INET, "127.0.0.1", &servAddr.sin_addr); //IP地址
```

初始化完服务端后就可以使用 `connect` 函数来连接服务器端，并且使用 `createthread` 函数创建一个线程用来接收服务器端发来的消息。这个创建的线程会以传入的 `Thread` 函数为开始地址执行。而 `Thread` 函数主要的功能是接收服务端的消息，并且判断消息的内容，如果是房间已满的消息就表示当前客户不能进入房间，就要关闭套接字。但如果是正常的消息，就正常输出到自己的终端上。

```
// 连接服务器
connect(clientSocket, (SOCKADDR*)&servAddr, sizeof(SOCKADDR));
printWelcome();

// 创建线程接收消息
CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Thread, NULL, 0, 0);
```

```
// 接收消息并美化输出
DWORD WINAPI recvThread() {
    while (true) {
        char buffer[SIZE] = {};
        if (recv(clientSocket, buffer, sizeof(buffer), 0) > 0) {
            if (strcmp(buffer, "房间已满，不能进入") == 0) {
                cout << RED << buffer << RESET << endl;
                closesocket(clientSocket);
                break;
            }
            cout << CYAN << "[收到消息]: " << RESET << buffer << endl;
        }
        else if (recv(clientSocket, buffer, sizeof(buffer), 0) < 0) {
            cout << RED << "[错误]: 失去连接!\n" << RESET << endl;
            break;
        }
    }
    Sleep(100); // 延时100ms
    return 0;
}
```

正确连接服务器端之后，客户端就可以向服务器端发送消息。因为客户端要持续向服务端发送消息，所以需要有一个永真的 `while` 循环来接收消息到缓冲区，然后使用 `send` 函数来发送到服务端套接字。其中我是用 `cin.getline` 函数来获取用户输入的聊天信息，并且判断是否是END，如果不是就正常发送到服务端，但如果是，就代表客户端想要退出该房间，就需要用 `break` 跳出循环结束客户端程序就可。

```
// 发送消息
char Mes[SIZE] = {};
while (true) {
    cout << GREEN << "[你]: " << RESET; // 提示用户输入消息
    cin.getline(Mes, sizeof(Mes));
    if (strcmp(Mes, "END") == 0) {
        break;
    }
    send(clientSocket, Mes, sizeof(Mes), 0); // 发送消息
}
// 关闭socket
```

```

closesocket(clientSocket);
WSACleanup();
cout << RED << "已退出聊天室，再见！" << RESET << endl;

```

- 服务端代码

上述介绍的是我的客户端代码，接下来我的服务端代码。同样的还是需要导入必要的库文件，因为跟网络相关的库文件跟客户端一模一样，所以不再进行解释。而多用的几个头文件是 `<chrono>`，`<list>`，`<windows.h>`。其中 `<chrono>` 头文件用来输出日志，`<list>` 头文件用来管理多个客户，`<windows.h>` 用来美化终端。

```

#include <iostream>
#include <cstdlib>
#include <cstring>
#include <chrono>
#include <thread>
#include <list>    // 包含双向链表头文件
#include <queue>   // 包含队列头文件
#include <winSock2.h>
#include <windows.h> // 用于控制台颜色
#include <ws2tcpip.h>
#pragma comment(lib, "ws2_32.lib")    // socket库

```

我设置了最多在线用户数为3，并为每个用户创建了一个结构体，保存其地址信息和套接字信息。为了便于管理和操作这些用户，我使用了C的双向链表 `list` 来维护在线用户列表。这种设计使得在用户连接时可以方便地插入新节点，而当用户断开连接时，也能快速删除相应节点，从而有效维护在线用户的状态。

为了处理超员的情况，我还引入了一个等待队列。当在线用户达到最大容量时，新连接的用户将被加入等待队列。当有现有用户断开时，排队的用户将自动进入房间。这一机制确保了对用户连接的有序管理，同时提升了系统的灵活性。

```

//保存客户信息结构体
struct client_info {
    SOCKADDR_IN clientAddr;
    SOCKET clientSocket;
};

using namespace std;

#define SIZE 4096 //消息最大长度
#define MaxClient 3 //最多在线用户

SOCKET serverSocket;           // 服务器端socket
SOCKADDR_IN serverAddr;        // 定义服务器地址
SOCKADDR_IN clientAddr;        // 定义客户端地址
list<client_info> clientList;    //管理在线用户
queue<client_info> waitingQueue; // 管理超员用户

int addrLen = sizeof(SOCKADDR); //结构体长度

```

而在主函数开始部分的步骤跟客户端一样，首先初始化 `winsock` 库，服务端套接字和服务端地址。有一点不一样的是服务端要使用 `bind` 函数来绑定自己的套接字到指定的 IP 地址：127.0.0.1。

```
// 绑定服务器地址和服务器socket
bind(serverSocket, (LPSOCKADDR)&serverAddr, sizeof(serverAddr));
```

在正确初始化和绑定完服务端之后，服务端就可以用listen函数监听来自客户端的连接请求。

```
// 监听
listen(serverSocket, MaxClient);
cout << "服务器开始监听，等待客户端连接...\n" << endl;
```

正如前面提到的，在服务端的消息接收循环中，我使用了一个永真 while 循环来处理客户端的连接请求。首先，检查当前连接的客户端数量是否小于最大允许的用户数 `MaxClient`。如果未达到限制，我会调用 `accept` 函数接收新的客户端连接，并为其创建一个 `client_info` 结构体，保存客户端的套接字和地址信息。新用户将被插入到链表中进行管理。

接着，我会向所有在线用户广播新用户加入的消息，并通过 `CreateThread` 函数为该客户端创建一个线程，执行 `handlerequest` 函数，专门处理该客户端的消息传输。如果当前连接的客户端数量已经达到最大值，则不会直接拒绝新用户连接，而是将超员的客户端加入等待队列。此时，在控制台输出“房间已满，客户已加入排队队列”的信息。当有客户端断开连接时，会从等待队列中取出一个排队的客户端，并允许其进入房间，按照正常连接流程处理。这种机制确保了超员用户的连接请求能够有序排队，而不是立即被拒绝。

```
// 接受消息
while (true) {
    SOCKET client = accept(serverSocket, (sockaddr*)&clientAddr, &addrLen);
// 接收客户端请求
    client_info newClient;
    newClient.clientSocket = client;
    newClient.clientAddr = clientAddr;
    if (clientList.size() < MaxClient) {
        // 广播新客户端进入的信息
        char enterMsg[SIZE];
        clientList.push_back(newClient); // 添加到客户端列表

        sprintf_s(enterMsg, sizeof(enterMsg), "客户 %d 已加入房间，当前客户数量为 %d", client, clientList.size());
        for (auto& client : clientList) {
            send(client.clientSocket, enterMsg, sizeof(enterMsg), 0);
        }
        cout << enterMsg << endl;
        HANDLE Thread = CreateThread(NULL, 0,
            (LPTHREAD_START_ROUTINE)handlerequest, (LPVOID)&clientList.back(), 0, NULL); // 创建线程
        CloseHandle(Thread);
    }
    else {

        char SendBuf[SIZE] = { "房间已满，已经进入排队等待" }; // 发送缓冲区
        SetConsoleColor(14); // 设置黄色字体

        SetConsoleColor(7); // 恢复默认颜色
        send(client, SendBuf, sizeof(SendBuf), 0);

        // 将超员的客户端放入等待队列

        cout << "房间已满，客户 " << client << " 加入排队队列" << endl;
```



```
        SetConsoleColor(7); // 恢复默认颜色
        waitingQueue.push(newClient);
    }
}
```

`handlerequest` 函数在多用户聊天应用中至关重要，它不仅用于处理消息同步，还负责管理客户端的连接状态以及异常情况处理。为了更好地理解其功能，以下是对该函数的详细说明：

### 接收客户端消息：

在 `handlerequest` 中，服务端首先通过 `recv` 函数等待客户端发送消息。这是一个阻塞操作，意味着服务端线程会暂停，直到从客户端接收到数据。接收到数据后，服务端会将消息存储在接收缓冲区中，并同时记录接收到的字节数。如果 `recv` 返回的字节数大于0，则表示消息接收成功。

### 时间戳与消息格式化：

为了给每条消息加上时间戳，服务端使用了 C++ 的 `chrono` 库来获取当前的系统时间。时间被格式化为人类可读的字符串形式，比如“YYYY-MM-DD HH:MM”，这有助于用户在接收到消息时看到准确的发送时间。该时间信息会与客户端发送的消息一同被格式化，包含客户端的套接字标识符、时间戳和消息内容。

### 消息广播：

一旦消息被格式化，服务端会通过遍历 `clientList`，将消息发送给每一个已连接的客户端。这一过程实现了多用户之间的消息同步，确保所有在线用户都能看到相同的消息。这种方式避免了消息遗漏，使得每个客户端都可以实时接收到其他用户的聊天信息。

### 客户端主动断开处理：

如果 `recv` 函数返回值为 0 或者特定错误代码 `10054`（表示远程主机强制关闭了现有连接），则服务端会判定该客户端已主动断开连接。在这种情况下，服务端会将该客户端的断开事件记录到日志中，并在控制台输出消息，提示该客户端已退出房间。接下来，服务端会使用 `closesocket` 关闭该客户端的套接字连接，确保资源不会被浪费。

为了维护当前在线用户列表，服务端还会将断开的客户端从 `clientList` 中移除。在完成移除操作后，服务端会再次广播一条消息，通知其他所有在线用户该客户端已退出。此功能不仅提升了用户体验，也有助于保持用户间的实时互动。

### 处理等待队列中的用户：

当有用户断开连接时，服务端检查当前客户端是否少于最大允许数 `MaxClient`，如果有空位，则从等待队列中取出排队的用户，并将其接入聊天房间。这时，服务器会像处理正常用户一样，创建线程并广播该用户加入的消息。

### 其他错误处理：

在某些情况下，`recv` 可能返回其他错误代码，意味着通信过程中发生了意料之外的问题。当遇到这种情况时，服务端会记录该错误并在控制台输出相应的错误信息。为了保证系统的稳定性，服务端会根据错误的严重性决定是否继续处理该客户端的请求或终止当前线程。

### 线程的关闭：

当某个客户端断开连接，或者发生无法恢复的错误时，服务端会安全地终止处理该客户端的线程。通过关闭相关的套接字和清理资源，确保服务端不会因为某个客户端的异常情况而导致资源泄露或系统崩溃。

```
DWORD WINAPI handlerequest(LPVOID lpParameter) // 线程函数
{
```

```

int receByt = 0;          // 接收到的字节数
char RecvBuf[SIZE];      // 接收缓冲区
char SendBuf[SIZE];      // 发送缓冲区
client_info* clientInfo = (client_info*)lpParameter;
SOCKET ClientSocket = clientInfo->clientSocket;

// 循环接收信息
while (true) {
    Sleep(100); // 延时100ms
    receByt = recv(ClientSocket, RecvBuf, sizeof(RecvBuf), 0); // 接收信息
    if (receByt > 0) { // 接收成功

        auto time_now =
chrono::system_clock::to_time_t(chrono::system_clock::now());
        tm localTime;
        localtime_s(&localTime, &time_now); // 使用 localtime_s 替代
localtime
        char timeStr[50];
        strftime(timeStr, sizeof(timeStr), "%Y-%m-%d--%H:%M:%S",
&localTime); // 格式化时间

        SetConsoleColor(14); // 设置黄色字体

        cout << "客户 " << ClientSocket << " : " << RecvBuf << " --" <<
timeStr << endl;
        SetConsoleColor(7); // 恢复为默认颜色

        sprintf_s(SendBuf, sizeof(SendBuf), "%d: %s --%s ", ClientSocket,
RecvBuf, timeStr); // 格式化发送信息

        // 将消息同步到所有聊天窗口
        for (auto& client : clientList) {
            send(client.clientSocket, SendBuf, sizeof(SendBuf), 0);
        }
    }
    else { // 接收失败
        if (WSAGetLastError() == 10054) { // 客户端主动关闭连接
            auto time_now =
chrono::system_clock::to_time_t(chrono::system_clock::now());
            tm localTime;
            localtime_s(&localTime, &time_now); // 使用 localtime_s 替代
localtime
            char timeStr[50];
            strftime(timeStr, sizeof(timeStr), "%Y-%m-%d--%H:%M:%S",
&localTime); // 格式化时间

            SetConsoleColor(12); // 设置红色字体
            cout << "客户 " << ClientSocket << " 已退出! 时间: " << timeStr <<
endl;

            SetConsoleColor(7); // 恢复默认颜色

            // 广播客户退出的信息
            char exitMsg[SIZE];
            sprintf_s(exitMsg, sizeof(exitMsg), "客户 %d 已退出房间,当前客户数量
为 %d", ClientSocket, clientList.size()-1);
            for (auto& client : clientList) {
                if (client.clientSocket != ClientSocket) { // 不发送给已经断开的
客户端

```



```

        send(client.clientSocket, exitMsg, sizeof(exitMsg), 0);
    }
}

clientList.remove_if([ClientSocket](const client_info& client) {
    return client.clientSocket == ClientSocket;
});
closesocket(ClientSocket);

SetConsoleColor(11); // 设置蓝色字体
cout << "现在的用户数量为: " << clientList.size() << endl;
SetConsoleColor(7); // 恢复默认颜色

// 检查等待队列
if (!waitingQueue.empty()) {
    client_info nextClient = waitingQueue.front();
    waitingQueue.pop(); // 取出队列中的下一个客户端

    client_info newClient;
    newClient.clientSocket = nextClient.clientSocket;
    newClient.clientAddr = nextClient.clientAddr;
    clientList.push_back(newClient);

    char enterMsg[SIZE];
    sprintf_s(enterMsg, sizeof(enterMsg), "排队客户 %d 已进入房间，
当前客户数量为 %d", nextClient.clientSocket, clientList.size());
    for (auto& client : clientList) {
        send(client.clientSocket, enterMsg, sizeof(enterMsg),
0);
    }
    cout << enterMsg << endl;

    HANDLE Thread = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE)handlerequest, (LPVOID)&clientList.back(), 0, NULL);
    CloseHandle(Thread);
}

return 0;
}
else {
    SetConsoleColor(12); // 设置红色字体
    cout << "接受失败: " << WSAGetLastError() << endl;
    SetConsoleColor(7); // 恢复默认颜色
    break;
}
}
}
return 0;
}

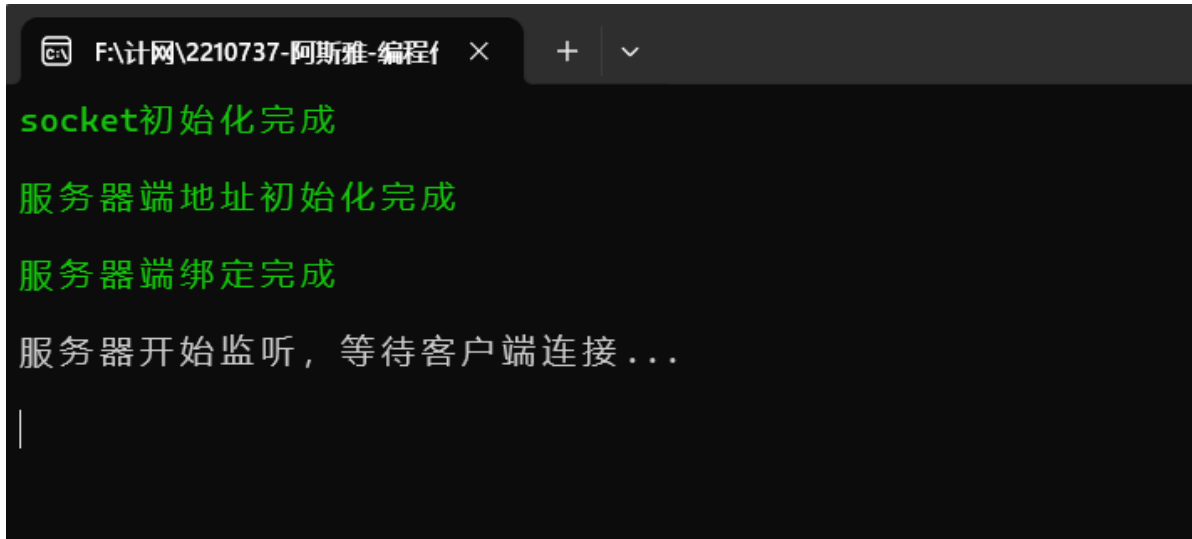
```

### 三、程序运行

如上面分析所得，因为服务器端需要持续监听客户端的请求，所以服务器端程序要先于客户端程序开启，然后客户端才能启动并连接服务端，在房间还未满的情况下成功连接之后就用户可以发送消息，如果房间已经满员，即使已经连接成功了用户也不能发送消息而是会进入到排队中，等在线用户退出就能进入到房间。

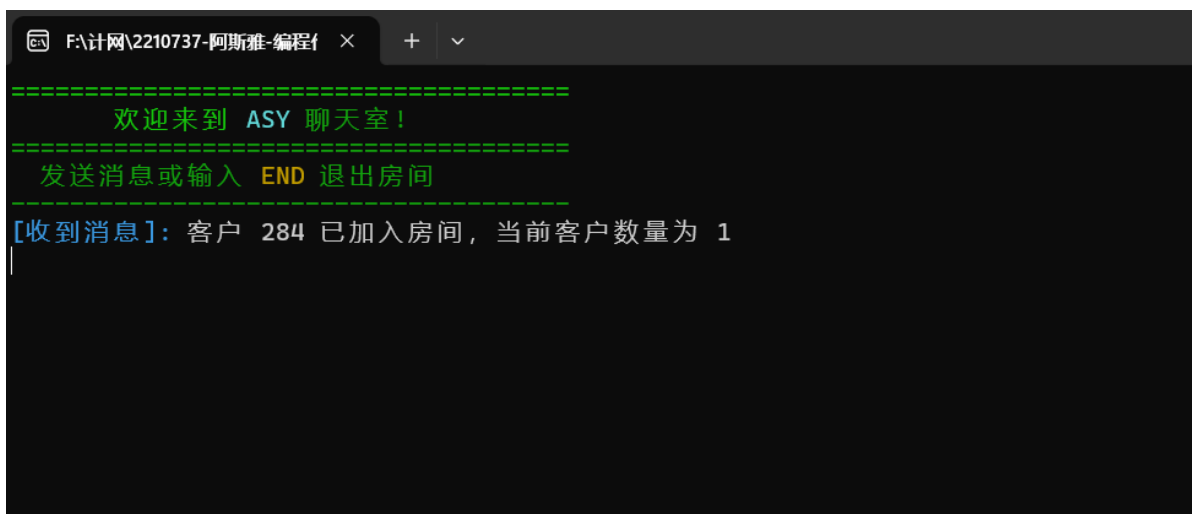
下面是我的程序运行的对于不同情况下的一些截图。

- 服务端界面



```
F:\计网\2210737-阿斯雅-编程\ >
socket初始化完成
服务器端地址初始化完成
服务器端绑定完成
服务器开始监听，等待客户端连接...
```

- 客户端



```
F:\计网\2210737-阿斯雅-编程\ >
=====
      欢迎来到 ASY 聊天室！
=====
      发送消息或输入 END 退出房间
=====
[收到消息]: 客户 284 已加入房间，当前客户数量为 1
```

- 多人聊天

```
FA计网\2210737-阿斯雅-编程 x + v
=====
欢迎来到 ASY 聊天室！
=====
发送消息或输入 END 退出房间
=====
[收到消息]: 客户 296 已加入房间, 当前客户数量
[收到消息]: 284: 1 --2024-10-16--22:24:14
2
[收到消息]: 296: 2 --2024-10-16--22:24:17

FA计网\2210737-阿斯雅-编程 x + v
=====
欢迎来到 ASY 聊天室！
=====
发送消息或输入 END 退出房间
=====
[收到消息]: 客户 284 已加入房间, 当前客户数量为 1
[收到消息]: 客户 296 已加入房间, 当前客户数量为 2
1
[收到消息]: 284: 1 --2024-10-16--22:24:14
[收到消息]: 296: 2 --2024-10-16--22:24:17

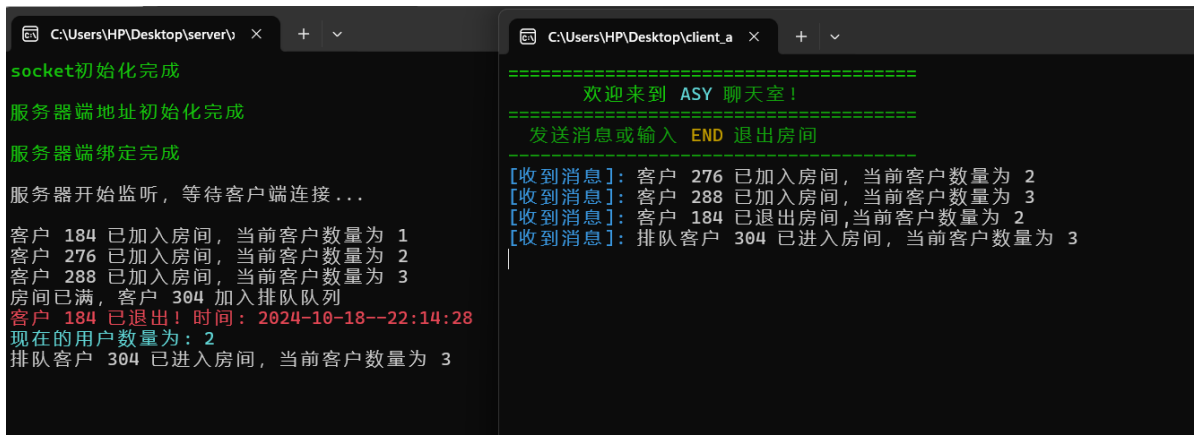
FA计网\2210737-阿斯雅-编程 x + v
=====
socket初始化完成
=====
服务器端地址初始化完成
=====
服务器端绑定完成
=====
服务器开始监听, 等待客户端连接...
=====
客户 284 已加入房间, 当前客户数量为 1
客户 296 已加入房间, 当前客户数量为 2
客户 284 : 1 --2024-10-16--22:24:14
客户 296 : 2 --2024-10-16--22:24:17
|
```

- 超员

```
C:\Users\HP\Desktop\client_a x + v
=====
欢迎来到 ASY 聊天室！
=====
发送消息或输入 END 退出房间
=====
房间已满, 已经进入排队等待
|
```

- 用户退出

```
FA计网\2210737-阿斯雅-编程 x + v
=====
欢迎来到 ASY 聊天室！
=====
发送消息或输入 END 退出房间
=====
[收到消息]: 客户 284 已加入房间, 当前客户数量为 1
[收到消息]: 客户 296 已加入房间, 当前客户数量为 2
1
[收到消息]: 284: 1 --2024-10-16--22:24:14
[收到消息]: 296: 2 --2024-10-16--22:24:17
[收到消息]: 客户 316 已加入房间, 当前客户数量为 3
END|
```

The image shows two side-by-side terminal windows. The left window, titled 'C:\Users\HP\Desktop\server\ \', displays server logs in green and red text. It shows the completion of socket initialization, server address initialization, and server binding. It then logs the arrival of clients 184, 276, and 288, reaching a capacity of 3. Client 184 is then shown logging out at 2024-10-18--22:14:28, leaving 2 active users. Client 304 is shown joining the queue. The right window, titled 'C:\Users\HP\Desktop\client\_a \', shows the client's perspective. It displays a welcome message to the 'ASY 聊天室!', a prompt to send messages or input 'END' to exit, and then logs the receipt of messages from the server regarding client 276 joining, client 288 joining, client 184 leaving, and client 304 joining the queue.

## 四、实验难点

- 用什么数据结构管理用户？

在这次实验中，使用基本的 `socket` 函数进行编程并没有带来太大的技术挑战，因为这些函数的调用方式较为直观，且可以通过网络轻松获取相关的用法说明。然而，真正的挑战在于如何有效管理和同步多个用户，特别是在用户的动态变化（连接和断开）与房间容量限制之间找到平衡。

### 数据结构的选择问题：

在聊天应用中，用户可以随时连接或退出，而在房间满员的情况下还需要拒绝新连接的用户。对于这种情况，选择合适的数据结构尤为关键。最开始，我考虑使用数组来管理用户，但数组的大小是固定的，在用户频繁变动时，数组的弹性不足，可能会导致大量的内存分配和拷贝操作，影响系统性能。尤其是在用户量大时，数组的管理和维护变得复杂且不高效。

因此，我转向了更灵活的解决方案——双向链表。双向链表具有动态性，能够在任何时刻高效地插入或删除节点，不会受到固定大小的限制，非常适合处理多用户聊天场景下的用户连接和断开管理。

### 链表的实现和挑战：

自行实现一个双向链表虽然理论上可行，但涉及到许多复杂的细节，特别是在内存管理和指针操作方面。如果处理不当，可能会导致内存泄漏、悬空指针等问题，进一步影响程序的稳定性。为了解决这个问题，我查阅了C语言标准库，并发现了 `<list>` 链表库。这个库提供了现成的双向链表实现，大大简化了我在用户管理方面的工作。

通过使用 `<list>` 库，我可以轻松地管理在线用户的添加和删除操作，并且无需担心指针操作和内存管理的问题。这不仅提高了代码的可读性，还显著减少了手动管理数据结构所带来的潜在错误风险。

### 多用户环境的灵活应对：

借助双向链表，我能够更加高效地应对多用户环境中的动态变化。例如，当用户连接时，只需在链表尾部插入一个新节点；当用户断开时，直接删除对应节点即可。房间满员时，链表结构也方便我快速遍历所有用户，确保不超出最大允许连接数。与此同时，链表的动态特性允许我在不破坏现有结构的前提下，轻松管理用户的进出，使得整个聊天应用在多用户环境下保持稳定和高效。

总体而言，双向链表的引入使得用户管理变得更加灵活和高效，解决了用户连接和断开过程中遇到的性能和扩展性问题，也使得我的聊天应用程序能够平稳运行并满足多用户环境的需求。

- 怎么处理满员的情况？

### 房间满员的处理策略：

当一个新用户尝试加入聊天室时，服务器首先检查当前的在线用户数量。通过一个简单的条件判断，如果在线人数已达到设定的最大值（3人），服务器会立即拒绝新的连接请求，并向该用户发送一条清晰的提示消息，表明“房间已满，无法加入”。这种即时反馈不仅能防止服务器承受过多的连接请求压力，还确保了用户能清楚地理解当前的情况，而不是因连接失败感到困惑。

### 提升用户体验的设计：

为了进一步优化用户体验，在客户端显示一个友好的提示界面会是个不错的增强。在房间已满的情况下，客户端可以弹出一个消息窗口，告知用户当前房间已满，建议稍后再试。这不仅比直接断开连接更加友好，还减少了用户的失望情绪。此外，客户端可以提供“排队等待”或“通知我”选项，当有用户离开房间时，系统会通知排队中的用户，有机会重新尝试连接。这种排队机制增强了聊天室的交互性和用户粘性。但在本次实验中我并没能实现。

### 资源管理和优化：

在房间已满的情况下，服务器和客户端都需要合理释放资源。服务器端在发送房间已满的消息后，会调用 `closeSocket` 函数关闭客户端的套接字连接。这不仅释放了相关的系统资源，还防止了不必要的资源占用。客户端在接收到服务器的提示消息后，也应立即关闭连接，避免无效的重试或资源浪费。

为了进一步优化资源管理，服务器还可以记录被拒绝的用户信息。这一信息可用于日志记录或后续的重新连接尝试。当有用户退出房间时，服务器可以立即向排队的用户发送通知，邀请他们重新加入。这种机制不仅能提高系统的用户管理效率，还为用户提供了更多的参与机会，避免用户因无法加入而感到失望或流失。

### 日志和系统监控：

服务器还可以记录所有尝试加入的用户和被拒绝的用户信息，方便后续的分析 and 优化。这些日志数据可以用于监控系统的运行情况、分析用户行为，甚至帮助你确定房间容量是否需要扩展。如果你发现经常有用户因房间已满而无法加入，这可能意味着你需要增加房间的容量或实现多房间管理。

### 排队队列：

#### 新用户请求时的处理：

当有新的用户尝试连接时，服务器首先检查当前在线用户数量。如果已满（即在线人数达到 `MaxClient`），则该用户不会被立即拒绝，而是会加入到等待队列中。通过使用 `queue` 数据结构存储超员用户信息，服务器能够按照“先来先服务”的原则处理排队的用户，保证每个排队用户有序等待。

新用户被加入队列后，服务器可以向该客户端发送一条排队消息，告知其正在等待进入房间，同时提示其当前的排队顺序，以增强用户的透明度和参与感。

#### 当房间有空位时的处理：

每当有现有用户主动断开连接或意外退出时，`handleRequest` 函数会处理用户退出的逻辑。此时，服务器不仅从 `clientList` 中移除该客户端，还会立即检查等待队列。如果队列中有排队用户，服务器会取出第一个用户，将其从等待队列中移到 `clientList` 中。服务器随后为该用户创建一个新的连接线程，并通过 `CreateThread` 函数执行 `handleRequest` 函数来处理该客户端的请求。与此同时，服务器也会向所有在线用户广播一条新用户进入房间的消息，通知他们当前的房间状态。