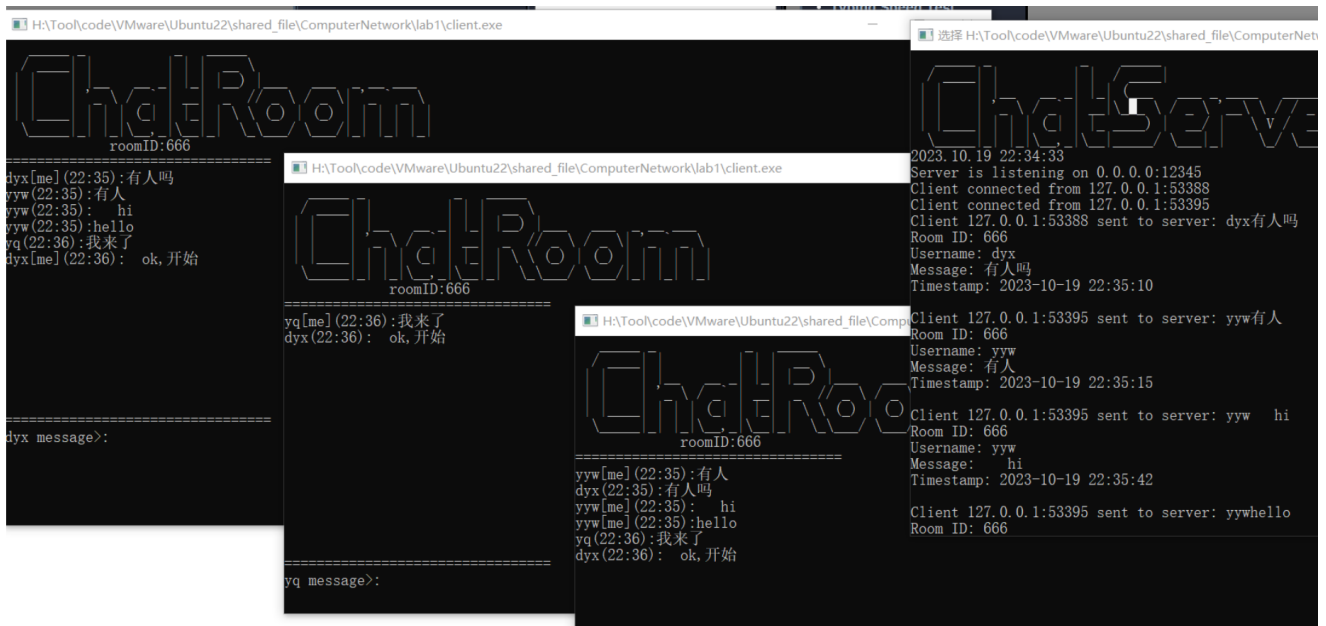


杜怡兴-2112847-socket网络编程实验报告



杜怡兴-2112847-socket网络编程实验报告

- 摘要
- 额外
- 前期准备
 - 实验环境
 - 项目组成
- 协议
 - 数据
 - 时序
 - 其他网络层
- 基础功能
- 必要的头文件
- 客户端程序
- 服务端程序
- 实验验证:
- 多线程功能
 - 新增头文件
 - 多线程服务端
 - 多线程客户端
- 消息协议
- 半可视化
- 房间机制
- 锁机制
- 实验结果
- 总结

摘要

本实验实现了：

- (1) 根据需求制定聊天协议。
- (2) 利用C++语言，使用基本的Socket函数完成程序，不使用CSocket等封装后的类编写程序。
- (3) 使用流式套接字、采用多线程方式完成程序。
- (4) 程序有半图形化的界面，有正常的退出方式。
- (5) 程序能支持双人、多人群发聊天，支持英文和中文聊天。
- (6) 代码中有详细注释，具有较好的可读性。
- (7) 实验中用wireshark观察，没有数据的丢失，实验源码含三个版本：单线程，多线程，半图形化界面。

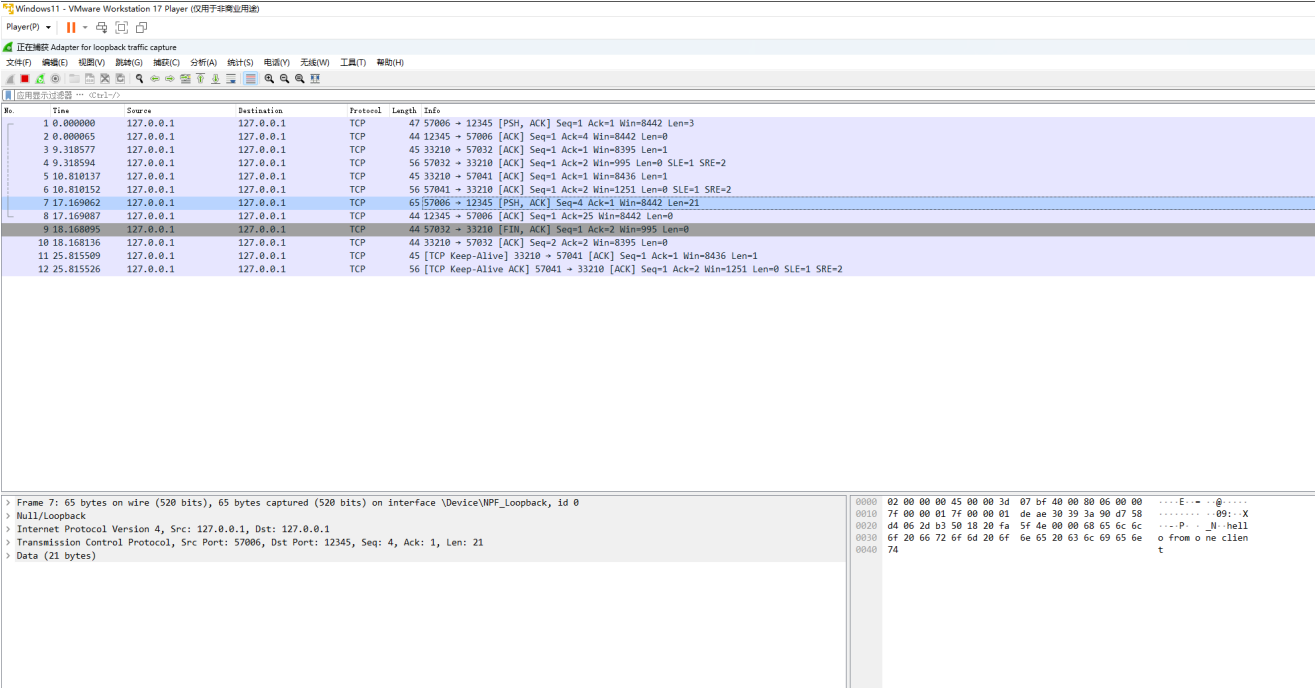
额外

本实验还实现了：

- (1)房间系统，在房间内的聊天彼此独立
- (2)锁机制保障多线程安全

前期准备

下载npcap，安装wireshark



安装gcc

实验环境

0.windows11

1.vscode编辑

2.gcc编译

编译指令:

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

项目组成

```
server.cpp-----服务端，编译后开启服务端监听
client.cpp-----客户端，编译后开启客户端监听
protocol.h-----协议头，服务端和客户端引入后都需要遵循的协议
```

协议

版本号：1.0

数据

本网络通信数据包的数据内容使用了ChatMessage结构体，依次包含用户名，消息，房间号，时间戳信息。

```
const int MAX_MESSAGE_LEN=1024;
struct ChatMessage {
    char username[20]; // 用户名
    char message[MAX_MESSAGE_LEN]; // 用户发送的消息
    int roomId; // 用户所在的房间号，暂时先不用管
    char timestamp[20]; // 添加时间戳成员
}
```

其中详细的要求如下：

类型	字段	大小	含义
char[20]	username	20 字节	用户名
char[MAX_MESSAGE_LEN]	message	最大消息长度（1024 字节）	用户发送的消息
int	roomId	4 字节	用户所在的房间号（整数类型）
char[20]	timestamp	20 字节	添加的时间戳成员，格式为 "YYYY-MM-DD HH:MM:SS"

数据编码格式：传递过程中用**字符串数组**传递，收到后读取响应数据

```
recv(clientSocket, (char *)&receivedMessage, sizeof(receivedMessage), 0);
```

时序

本实验源码有三个版本，最终版之前还有一个单线程版本

单线程时序：

第一步，启动server，服务端建立在localhost:12345,然后阻塞程序，等待一个客户端连接。

第二步，启动client，配置客户端，之后向localhost:12345请求连接。

连接成功后，客户端阻塞，等待用户输入信息。

服务端和一个客户端连接，阻塞解除，开始等待消息传输，再次阻塞。

第三步，用户输入消息，客户端解除阻塞，发送消息到服务端，等待用户输入再次阻塞。

服务端收到消息，解除阻塞，输出用户信息，再次等待消息，继续阻塞。

第四步，重复第三步

多线程时序：

第一步，启动server，服务端建立在localhost:12345,然后阻塞程序，等待一个客户端连接。

第二步，启动client，配置客户端，之后向localhost:12345请求连接。

连接成功后，建立子线程，处理服务端发给客户端其他用户的信息，期间子线程阻塞直到收到信息。

客户端再次主线程阻塞，等待用户输入信息。

服务端和一个客户端连接，创建一个新的子线程，在新的子线程里，等待客户端输入，期间子线程阻塞。

服务端主线程再次阻塞，等待客户端再次连接。

第三步，用户输入消息，客户端主线程将消息发给服务端，服务端子线程处理消息，进行广播，客户端子线程收到回复，进行处理。

第四步，重复第三步

其他网络层

1、本地回环接口

- 利用本地回环接口来模拟客户端和服务端之间的通信。

2. IP地址

本地回环接口的IP地址为 127.0.0.1 或主机名 localhost，客户端的端口例如59000由系统指定，服务端端口为 12345

3.网络采取大端序，解析采取小端序

例如IP地址需要用库函数将网络序转换成小端序（windows系统）

```
inet_ntoa(clientAddr.sin_addr)
```

基础功能

必要的头文件

接下来的所有代码默认有如下头文件：

```
#include <string>
#include <iostream>
#include <cstdio>
using namespace std;//经典头文件

#include <WinSock2.h>//网络编程必要头 socket 来自于windows socket
#include <thread>//多线程
#pragma comment(lib, "ws2_32.lib")//编译需要的链接库
#pragma warning(disable : 4996)//不要有4996相应的报错
```

客户端程序

首先定义服务端地址(本地127.0.0.1)，客户端接口

```
// 初始化 Windows 套接字库 (Winsock),即使后面没有使用,也要写
WSADATA wsaData; // windows socket api data
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
{ // Winsock Startup,需要版本号和data地址 0表示成功,非0表示失败
    // MAKEWORD(2, 2) 是一个宏,它将两个字节大小的参数合并成一个,这里表示用 Winsock 2.2 版本的库
    cerr << "WSAStartup failed." << endl;
    system("pause");
}

// 输出 Winsock 版本信息,例如Winsock version: 514.514
// std::cout << "Winsock version: " << wsaData.wVersion << "." << wsaData.wHighVersion <<
std::endl;

SOCKET clientSocket = socket(AF_INET, SOCK_STREAM, 0); // AF_INET 地址家族 表示ipv4, 流式套接
字, 0表示协议自动选择
//对于TCP套接字 (SOCK_STREAM), 操作系统将自动选择TCP协议。对于UDP套接字 (SOCK_DGRAM), 操作系统将
自动选择UDP协议。
if (clientSocket == INVALID_SOCKET)
{ // INVALID_SOCKET表示一个数值, (SOCKET)(~0),也就是-1, 如果是无效的
    cerr << "Failed to create socket." << endl;
    WSACleanup(); // windows socket api 清除Winsock库所占用的资源
    system("pause");
}
```

运行后立即尝试连接服务端

```

sockaddr_in serverAddr; // 接口地址 internet, 对应着服务器地址
serverAddr.sin_family = AF_INET; // socket info ipv4
serverAddr.sin_port = htons(12345); // 服务器端口12345
serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1"); // 设置服务器IP地址, 由于服务器是本电脑, 所以只能是127.0.0.1

if (connect(clientSocket, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) ==
SOCKET_ERROR)
{ // 根据服务器地址, 客户端接口, 建立连接
    // 建立成功返回0, 否则是一个数字对应error
    cerr << "Failed to connect to the server." << endl;
    closesocket(clientSocket); // 关闭失败连接
    WSACleanup();
    system("pause");
}

```

反复读取用户信息, 将用户信息发到服务端

```

cout << "Connected to the server." << endl;

char message[1024]; // 用户可以发的信息是有限的

while (true)
{ // 反复让用户输入信息, 提示quit退出
    cout << "Enter a message to send to the server (or type 'quit' to exit): ";
    cin.getline(message, sizeof(message)); // 读取一行

    if (strcmp(message, "quit") == 0)
    { // 如果读取到的message是quit, 退出
        break;
    }
    // 否则就是输入了要发送的信息, 通过send可以发送给服务端
    int sendResult = send(clientSocket, message, strlen(message), 0); // 需要客户端接口, 要发的信息, 长度, 额外操作
    if (sendResult == SOCKET_ERROR)
    { // 如果发送成功
        cerr << "Send failed." << endl;
        break;
    }
}
}

```

服务端程序

定义服务端接口

```

WSADATA wsaData; // windows socket api data, 2.2的库, 这里一样, 即使后面不用, 必须要初始化
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0){
    cerr << "WSAStartup failed." << endl;
    system("pause");
}

SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0); // 服务端接口, ipv4地址, 流式套接字[已经实现], 协议自动选择
if (serverSocket == INVALID_SOCKET)
{ // 接口创建失败
    cerr << "Failed to create socket." << endl;
}

```

```

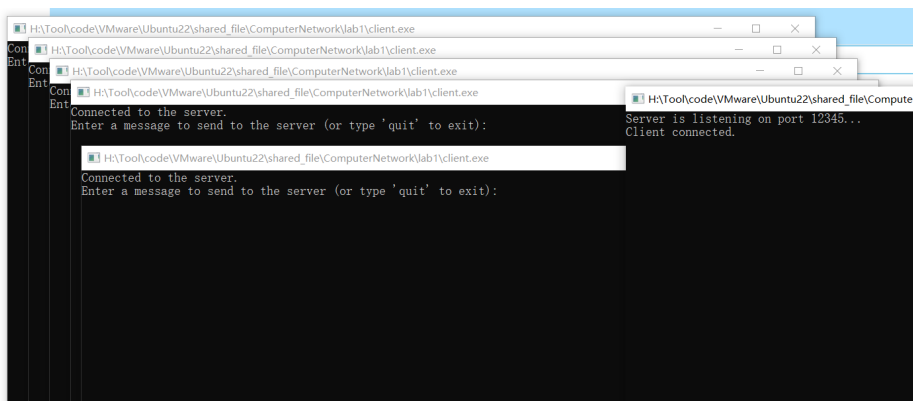
WSACleanup();
system("pause");
}

sockaddr_in serverAddr;           // 服务器地址
serverAddr.sin_family = AF_INET; // socket info ipv4
serverAddr.sin_port = htons(12345); // 服务器端口12345
serverAddr.sin_addr.s_addr = INADDR_ANY; // 设置服务器IP地址, 由于服务器是本电脑, 所以只能是127.0.0.1
// 服务器接口要和地址绑定
if (bind(serverSocket, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) == SOCKET_ERROR){
    cerr << "Bind failed." << endl; // 如果失败
    closesocket(serverSocket);      // 关闭接口, 清理接口占用
    WSACleanup();
    system("pause");
}

if (listen(serverSocket, 5) == SOCKET_ERROR)
{ // 排队的客户端只能有5个
    cerr << "Listen failed." << endl;
    closesocket(serverSocket);
    WSACleanup();
    system("pause");
}
// 获取服务端的IP地址并打印, 通常是0.0.0.0:12345, 因为服务器绑定到了所有可用的网络接口上, 端口是12345
cout << "Server is listening on " << inet_ntoa(serverAddr.sin_addr) << ":" <<
ntohs(serverAddr.sin_port) << endl;

```

listening的时候, 设置至多有5个客户端排队, 之后的连接不了



服务端开始监听客户端的连接请求:

```

SOCKET clientSocket; // 客户端接口
sockaddr_in clientAddr; // 客户端地址
int clientAddrLen = sizeof(clientAddr);
// accept 函数在服务器端一直监听客户端的连接请求【accept 函数会阻塞程序, 直到有客户端连接进来】, 连接后accept 函数会自动创建一个新的接口返回, 也就是我们的客户端接口
// 此时服务端和这个客户端建立了连接, 可以直接通信了
// 客户端连接的时候, 不需要手动填写地址, 接收连接请求的时候, 客户端地址会被自动填充
clientSocket = accept(serverSocket, (struct sockaddr *)&clientAddr, &clientAddrLen); // 服务器接口和客户端地址
// 获取客户端的IP地址并打印
cout << "Client connected from " << inet_ntoa(clientAddr.sin_addr) << ":" <<
ntohs(clientAddr.sin_port) << endl;

```

recv函数会造成一个中断，就像是cin函数一样，在收到客户端接口发来的信息之前，while其实是停止的【就像是cin在用户输入之前，while也是停止的】，知道接收到客户端信息，输出客户端信息。





```
char buffer[1024]; // 缓冲区，接收客户端信息
int recvResult;    // 收到的字符个数


while (true){
    recvResult = recv(clientSocket, buffer, sizeof(buffer), 0); // 接收客户端接口的信息，结果写入buffer，无特殊标志
    if (recvResult > 0){
        buffer[recvResult] = '\0'; // 字符串最后一位的串尾符
        cout << "Client " << inet_ntoa(clientAddr.sin_addr) << ":" << ntohs(clientAddr.sin_port) << " sent to server: " << buffer << endl;
    }
    else if (recvResult == 0)
    { // 没有信息，表示关闭
        //但是：TCP会周期性地检测连接是否仍然存在。如果客户端突然关闭了，服务器可能需要等待一段时间才能检测到连接的断开，
        cout << "Client disconnected." << endl;
        break;
    }
    else
    { // 其他是错误
        cerr << "Recv failed." << endl;
        // break;
    }
    cout << "wating for next message..." << endl;
}
```

到此为止实现了如下功能：

实验验证：

第一步，启动server，服务端建立在localhost:12345,然后阻塞程序，等待一个客户端连接。

名称	修改日期	
 .vscode	2023/10/18 18:53	
 不同版本	2023/10/18 22:34	
 2112847-杜怡兴-实验报告.md	2023/10/18 23:00	
 client.cpp	2023/10/18 23:07	
 client.exe	2023/10/18 23:17	

 H:\Tool\code\VMware\Ubuntu22\shared_file\ComputerNetwork\lab1\server.exe
Server is listening on 0.0.0.0:12345

第二步，启动client，配置客户端，之后向localhost:12345请求连接，连接成功后，客户端阻塞，等待用户输入信息。


```
H:\Tool\code\VMware\Ubuntu22\shared_file\ComputerNetwork\lab1\client.exe
Connected to the server.
Enter a message to send to the server (or type 'quit' to exit):
```

服务端和一个客户端连接，阻塞接触，开始等待消息传输，再次阻塞。

```
H:\Tool\code\VMware\Ubuntu22\shared_file\ComputerNetwork\lab1\se
Server is listening on 0.0.0.0:12345
Client connected from 127.0.0.1:53950
```

第三步，用户输入消息，客户端解除阻塞，发送消息到服务端，等待用户输入再次阻塞。

```
H:\Tool\code\VMware\Ubuntu22\shared_file\ComputerNetwork\lab1\client.exe
Connected to the server.
Enter a message to send to the server (or type 'quit' to exit): abc
Enter a message to send to the server (or type 'quit' to exit): dyx
Enter a message to send to the server (or type 'quit' to exit): 你好dyx
Enter a message to send to the server (or type 'quit' to exit):
```

服务端收到消息，解除阻塞，输出用户信息，再次等待消息，继续阻塞。

```
H:\Tool\code\VMware\Ubuntu22\shared_file\ComputerNetwork\lab1\client.exe
Connected to the server.
Enter a message to send to the server (or type 'quit' to exit): abc
Enter a message to send to the server (or type 'quit' to exit): dyx
Enter a message to send to the server (or type 'quit' to exit): 你好dyx
Enter a message to send to the server (or type 'quit' to exit):
```

第四步，重复第三步

可以见到，支持中英文通信，中间消息没有丢失

接着进行了wire shark的数据包捕获，记得选择adapter loop选项，也就是本地回环数据包，结果如下：

Windows11 - VMware Workstation 17 Player (仅用于非商业用途)

Player(P)

正在捕获 Adapter for loopback traffic capture

文件(F) 编辑(E) 视图(V) 录制(G) 捕获(C) 分析(A) 统计(S) 电话(W) 无线(W) 工具(T) 帮助(H)

应用展示过滤器

Ctrl-F

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	TCP	47	57006 → 12345 [PSH, ACK] Seq=1 Ack=1 Win=8442 Len=3
2	0.000065	127.0.0.1	127.0.0.1	TCP	44	12345 → 57006 [ACK] Seq=1 Ack=4 Win=8442 Len=0
3	9.318577	127.0.0.1	127.0.0.1	TCP	45	33210 → 57032 [ACK] Seq=1 Ack=1 Win=8395 Len=1
4	9.318594	127.0.0.1	127.0.0.1	TCP	56	57032 → 33210 [ACK] Seq=1 Ack=2 Win=995 Len=0 SLE=1 SRE=2
5	10.810137	127.0.0.1	127.0.0.1	TCP	45	33210 → 57041 [ACK] Seq=1 Ack=1 Win=8436 Len=1
6	10.810152	127.0.0.1	127.0.0.1	TCP	56	57041 → 33210 [ACK] Seq=1 Ack=2 Win=1251 Len=0 SLE=1 SRE=2
7	17.169062	127.0.0.1	127.0.0.1	TCP	65	57006 → 12345 [PSH, ACK] Seq=4 Ack=1 Win=8442 Len=21
8	17.169087	127.0.0.1	127.0.0.1	TCP	44	12345 → 57006 [ACK] Seq=1 Ack=25 Win=8442 Len=0
9	18.168095	127.0.0.1	127.0.0.1	TCP	44	57032 → 33210 [FIN, ACK] Seq=1 Ack=2 Win=995 Len=0
10	18.168136	127.0.0.1	127.0.0.1	TCP	44	33210 → 57032 [ACK] Seq=2 Ack=2 Win=8395 Len=0
11	25.815509	127.0.0.1	127.0.0.1	TCP	45	[TCP Keep-Alive] 33210 → 57041 [ACK] Seq=1 Ack=1 Win=8436 Len=1
12	25.815526	127.0.0.1	127.0.0.1	TCP	56	[TCP Keep-Alive ACK] 57041 → 33210 [ACK] Seq=1 Ack=2 Win=1251 Len=0 SLE=1 SRE=2

> Frame 7: 65 bytes on wire (520 bits), 65 bytes captured (520 bits) on interface \Device\NPF_{...} id 0

> Null/Loopback

> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

> Transmission Control Protocol, Src Port: 57006, Dst Port: 12345, Seq: 4, Ack: 1, Len: 21

> Data (21 bytes)

0000 02 00 00 00 45 00 00 3d 07 bf 40 00 80 06 00 00 ...E...-...@....

0010 7f 00 00 01 7f 00 00 01 de ae 30 39 3a 90 d7 58-...09::X

0020 d4 06 2d b3 50 18 20 fa 5f 4e 00 00 68 65 6c 6c ...P..._H-hell

0030 6f 20 66 72 6f 6d 20 6f 6e 65 20 63 6c 69 65 6e o from o ne clien

0040 74 t

如图，捕获到了客户端发送的:"hello from one client"字样

至此为止，我们实现了实验要求的：

利用C或C++语言，使用基本的Socket函数完成程序。不使用CSocket等封装后的类编写程序。使用流式套接字。程序有基本的对话界面，但不是图形界面。程序有正常的退出方式。支持英文和中文聊天。经过测试，没有数据包丢失。

多线程功能

新增头文件

需要用vector来存储每个客户端连接的线程

```
#include <vector>
vector<SOCKET> clientSockets; // 存储所有客户端套接字
```

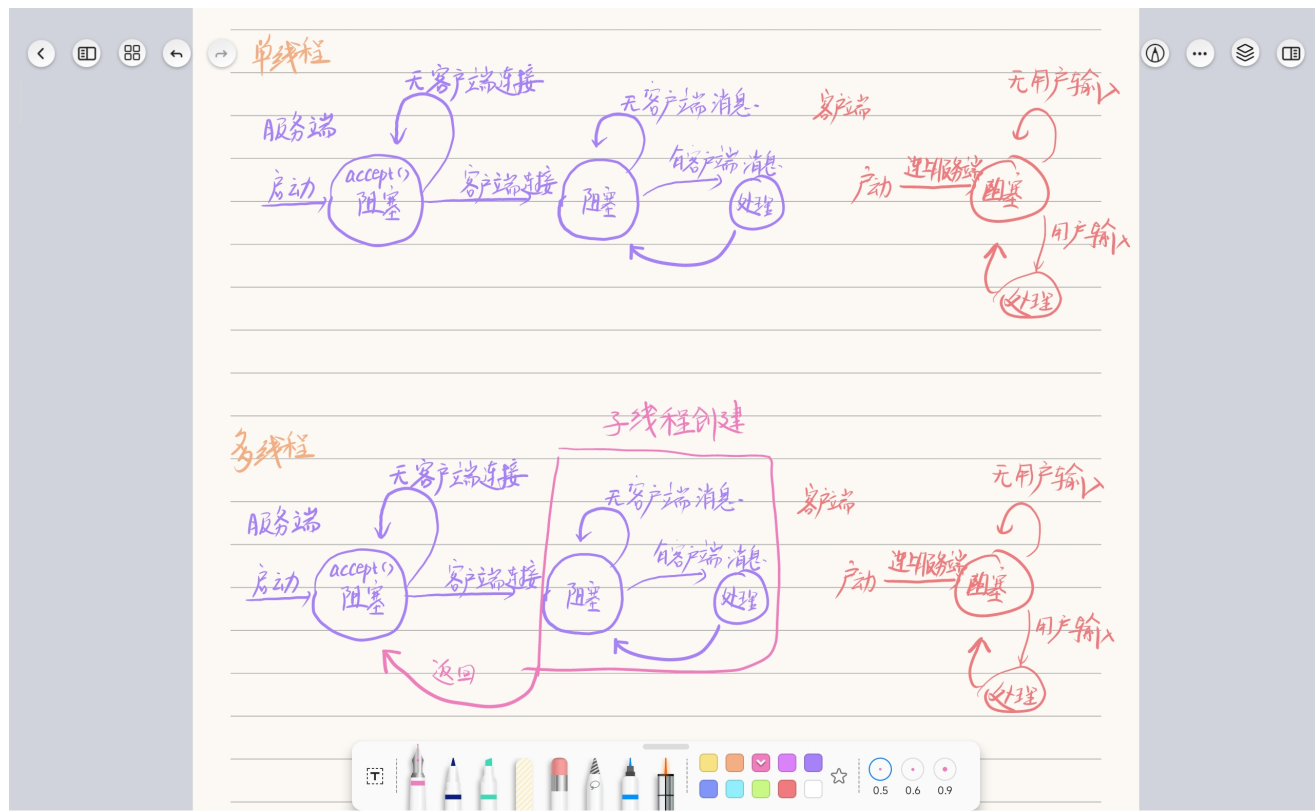
多线程服务端

开启服务端之后，依然需要accept函数阻塞，然后等待客户端连接。

和单线程的不同点在于，单线程一旦连接上，就等待客户端输入(再次阻塞)，然后把消息输出。

而多线程一旦连接上，就创建一个新的子线程，在新的子线程里，等待客户端输入（是阻塞的,但是只阻塞了子线程），然后把消息输出，但是不影响主线程，主线程继续阻塞在等待客户端连接。

如手绘状态图所示：



```
while (true) {
    SOCKET clientSocket; // 客户端接口
    sockaddr_in clientAddr; // 客户端地址
    int clientAddrLen = sizeof(clientAddr); // 地址长度
```

```

        clientSocket = accept(serverSocket, (struct sockaddr*)&clientAddr, &clientAddrLen);//中
断，直到一个客户端连接上，之后自动填写客户端地址，返回客户端接口
        if (clientSocket == INVALID_SOCKET) { //连接失败
            cerr << "An accept failed." << endl;
        } else { //否则就是连接上了，可以输出连接到的客户端的ip和端口，注意网络序转小端序
            cout << "Client connected from " << inet_ntoa(clientAddr.sin_addr) << ":" <<
ntohs(clientAddr.sin_port) << endl;
            // lock_guard<mutex> lock(mtx);
            clientSockets.push_back(clientSocket); //所有客户端接口需要保留，后面还有用
            //线程创立之后立即执行，但不会阻塞，也就是执行的同时后面的代码继续执行，thread第一个参数是执
行的函数名，第二个是函数参数
            thread t(HandleClient, clientSocket, clientAddr); //一旦连接上，开一个线程处理客户端的消息
            t.detach(); // 分离线程，不等待线程结束
        }
    }
}

```

而在子线程中,子线程内阻塞，等待客户端消息，收到后输出，和之前一样

```

void HandleClient(SOCKET clientSocket, sockaddr_in clientAddr) {
    char buffer[1024]; //客户端发送的消息
    int recvResult; //长度

    while (true) {
        recvResult = recv(clientSocket, buffer, sizeof(buffer), 0); //子线程内阻塞，等待客户端消息，
收到后写入buffer，flags=0表示没有额外操作
        if (recvResult > 0) { //收到消息
            buffer[recvResult] = '\0'; //串尾符为\0
            //输出客户端的ip和信息
            cout << "Client " << inet_ntoa(clientAddr.sin_addr) << ":" <<
ntohs(clientAddr.sin_port) << " sent to server: " << buffer << endl;
            //广播这个消息到其他所有用户
        } else if (recvResult == 0) {
            cout << "Client disconnected." << endl;
            break;
        } else {
            cerr << "Recv failed." << endl;
            break;
        }
    }

    closesocket(clientSocket);
}

```

此时实现的效果如下：允许多个客户端向服务端发起连接请求，服务端可以收到多个客户端的消息，并将客户端信息和消息显示出来。

```
H:\Tool\code\VMware\Ubuntu22\shared_file\ComputerNetwork\lab1\client.exe
Server is listening on 0.0.0.0:12345
Client connected from 127.0.0.1:60738
Client 127.0.0.1:60738 sent to server: dyx
Client 127.0.0.1:60738 sent to server: bc
Client connected from 127.0.0.1:60751
Client 127.0.0.1:60751 sent to server: another
Client connected from 127.0.0.1:60784
Client 127.0.0.1:60784 sent to server: third
Client 127.0.0.1:60738 sent to server: back

H:\Tool\code\VMware\Ubuntu22\shared_file\ComputerNetwork\lab1\client.exe
Connected to the server.
Enter a message to send to the server (or type 'quit' to exit): dyx
Enter a message to send to the server (or type 'quit' to exit): bc
Enter a message to send to the server (or type 'quit' to exit): back
Enter a message to send to the server (or type 'quit' to exit):

H:\Tool\code\VMware\Ubuntu22\shared_file\ComputerNetwork\lab1\client.exe
Connected to the server.
Enter a message to send to the server (or type 'quit' to exit): another
Enter a message to send to the server (or type 'quit' to exit):

H:\Tool\code\VMware\Ubuntu22\shared_file\ComputerNetwork\lab1\client.exe
Connected to the server.
Enter a message to send to the server (or type 'quit' to exit): third
Enter a message to send to the server (or type 'quit' to exit):
```

广播效果，服务端收到客户端消息之后，不仅要输出它，还需要将消息广播给所有这个用户之外的用户

```
//输出客户端的ip和信息,之后
BroadcastMessage(buffer, clientSocket); //广播这个消息到其他所有用户
```

广播给所有这个用户之外的用户

```
void BroadcastMessage(const char* message, SOCKET senderSocket ) { //将message发给除了发送者之外的接口
    for (SOCKET clientSocket : clientSockets) { //遍历所有客户端接口
        if (clientSocket != senderSocket) { //除了发送者接口
            int sendResult = send(clientSocket, message, strlen(message), 0); //将message发送给这个接口, flags=0没有额外操作
            if (sendResult == SOCKET_ERROR) { //一般不报错
                cerr << "Send failed." << endl;
                system("pause");
            }
        }
    }
}
```

多线程客户端

由于需要除了等待用户输入还需要等待其他用户消息，因此客户端也需要是多线程的。

在开始监听用户输入之前，需要开一个线程用来接收消息

```
thread t(ReceiveMessages, clientSocket); //用来接收消息的线程，和接口
```

接收消息输出消息的方法和服务端是一样的

```
void ReceiveMessages(SOCKET clientSocket) {
    char buffer[1024]; //消息
    int recvResult; //长度
```

```

while (true) {
    recvResult = recv(clientSocket, buffer, sizeof(buffer), 0); //子线程等待接收消息，阻塞
    if (recvResult > 0) {
        buffer[recvResult] = '\0';
        cout << "Server sent: " << buffer << endl; //将消息输出
    } else if (recvResult == 0) {
        cout << "Disconnected from the server." << endl;
        break;
    } else {
        cerr << "Recv failed." << endl;
        break;
    }
}
}
}

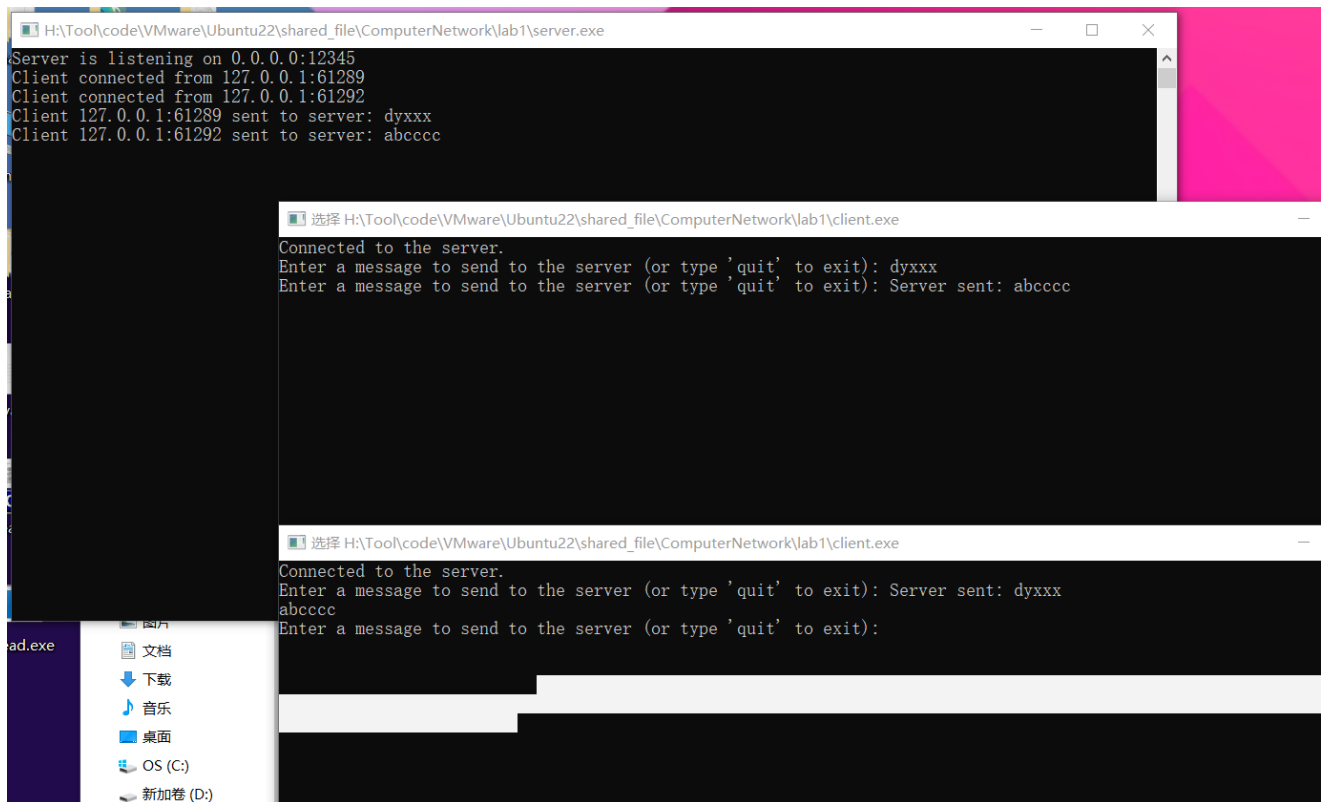
```

此时的效果是：

一个用户发送的信息，可以被另外一个用户收到，但是有两个缺点：

1. 一个用户输入的时候，可能同时收到另外一个用户的消息，而在非可视化界面，输入和输出都在同一个地方，也就是输入会被输出打断。

2. 信息是无状态的，也就是只能显示消息来自于服务器。



消息协议

为了解决刚才的问题，我们可以要求在通信时必须携带用户名，需要重新设计消息结构体，写在protocol.h，需要在client.cpp和server.cpp引入

```
#include "protocol.h"
```

其中协议规定了发送的信息的结构，除了消息外，需要用户名包含用户身份信息，房间号实现房间内通讯,时间记录用户消息时间。

其中特别要注意，这里不要用string! 而是用类似于char[20]，因为结构体在通信发送的时候，里面的string读取可能会出问题。

```
struct ChatMessage {
    char username[20]; // 用户名
    char message[MAX_MESSAGE_LEN]; // 用户发送的消息
    int roomId; // 用户所在的房间号，暂时先不用管
    char timestamp[20]; // 添加时间戳成员

    ChatMessage() {}

    // 构造函数，用于方便初始化 ChatMessage 的实例
    ChatMessage(const char* _username, const char* _message, int _roomId)
        : roomId(_roomId) {
        // 复制用户名和消息
        strncpy(username, _username, MAX_MESSAGE_LEN - 1);
        username[MAX_MESSAGE_LEN - 1] = '\0'; // 确保字符串以 null 结尾

        strncpy(message, _message, MAX_MESSAGE_LEN - 1);
        message[MAX_MESSAGE_LEN - 1] = '\0'; // 确保字符串以 null 结尾

        // 获取当前时间，并将其格式化为字符串
        auto now = std::chrono::system_clock::now();
        auto time_t_now = std::chrono::system_clock::to_time_t(now);
        strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S", localtime(&time_t_now));
    }
};
```

既然需要用户名，就意味着客户端打开的时候，必须要用户输入用户名，房间号，发送到服务端的时候是发结构体。

```
while (true)
{ // 反复让用户输入信息，提示quit退出
    cout << ">:";
    cin.getline(message.message, MAX_MESSAGE_LEN); // 读取一行,这次是到字符串里面
    //消息要最新的时间
    UpdateTimestamp(message);
    PrintChatMessage(message);
    int sendResult = send(clientSocket, (const char*)&message, sizeof(ChatMessage), 0); //
    //需要客户端接口，要发的信息，长度，额外操作

}
```

接收也是结构体

```
void ReceiveMessages(SOCKET clientSocket){
    ChatMessage receivedMessage;
    int recvResult; // 长度
    while (true){
        recvResult = recv(clientSocket, (char *)&receivedMessage, sizeof(receivedMessage), 0);
        // 子线程等待接收消息, 阻塞
        if (recvResult == sizeof(ChatMessage)){
            PrintChatMessage(receivedMessage);
            cout << receivedMessage.username << receivedMessage.message << endl;
        }
    }
}
```

此时的效果是，一个用户发送的信息【整个结构体的所有信息】会发送给自己之外的所有用户，可以被其他所有用户看见

The screenshot displays a Windows desktop environment. At the top, there's a taskbar with icons for back, forward, search, and several open applications. The main area shows two overlapping terminal windows.

The left terminal window has a title bar indicating it's running "lab1". It contains C++ code snippets from files like "protocol.h", "server.cpp", and "client.cpp". The code includes comments about receiving messages via SOCKET, defining constants like "ws_32.lib", and handling network connections. A specific line of code is highlighted: `H:\Tool\code\VMware\Ubuntu22\shared_file\ComputerNetwork\lab1\server`.

The right terminal window has a title bar indicating it's running "H:\Tool\code\VMware\Ubuntu22\shared_file\ComputerNetwork\lab1\client.exe". It shows the output of the client program, which interacts with a server. The output includes timestamps, welcome messages, prompts for username and chat room, and the exchange of messages ("hi" and "hello world").

半可视化

为了简化，我们不用可视化界面，但是尽量模拟出聊天室的效果，既然作为聊天室，就应该有聊天区和输入区两个部分，还需要考虑用户正在输入时，消息到来的情况，不能挤占输入空间，这里的策略是：将所有收到的消息存储起来，用户输入总是在最底下，每次新的消息到来的时候`system("cls")`，重新渲染页面，将过去收到的消息依次显示在上面。

[illegible]

```

ChatMessage message;
vector<ChatMessage> historyMessages;

void renderScreen(){
    system("cls");
    // 输出所有历史消息
    for (const char* line : asciiArt) {
        std::cout << line << std::endl;
    }
    cout<<"                roomID:"<<message.roomId<<endl;
    cout<<"===== "<<endl;

    for (const ChatMessage& msg : historyMessages){
        // 使用 std::string 来方便处理
        std::string timestamp(msg.timestamp);
        // 截取字符串, 保留小时和分钟部分
        std::string formattedTimestamp = timestamp.substr(11, 5);
        cout << msg.username<< (strcmp(msg.username,message.username)==0?("[me]"): "") << "("
<<formattedTimestamp <<"): "<< msg.message << endl;
    }
    // cout<<receivedMessage.username<<" "<<receivedMessage.message<<endl;
    for(int i=1;i<=15-historyMessages.size();i++){//如果消息比较少, 用空行补齐
        cout<<std::endl;
    }
    cout<<"===== "<<endl;
    cout<<message.username<<" message>:";
}

void ReceiveMessages(SOCKET clientSocket){
    ChatMessage receivedMessage;
    int recvResult; // 长度

    while (true){
        recvResult = recv(clientSocket, (char *)&receivedMessage, sizeof(receivedMessage), 0);
        // 子线程等待接收消息, 阻塞
        if (recvResult == sizeof(ChatMessage)){
            historyMessages.push_back(receivedMessage);//新增消息
            renderScreen();//渲染屏幕
            recvResult=0;//保证下一次收到信息之前, 不会再次输出
        }
    }
}

```


效果是这样的，用户输入的时候是主线程，消息来了刷新是子线程，自己的消息和收到的消息都加入vector

```
选择 H:\Tool\code\VMware\Ubuntu22\shared_file\ComputerNetwork\lab1\client.exe

ChatRoom
roomID:666
=====
dyx[me] (22:35):有人吗
yyw(22:35):有人
yyw(22:35): hi
yyw(22:35):hello
yyq(22:36):我来了
dyx[me] (22:36): ok, 开始

=====
dyx message>:
```

房间机制

在用户端登陆的时候，把房间号发给客户端，顺带加入一点延时，防止客户端没有收到就开始发消息

```
if (connect(clientSocket, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) ==
SOCKET_ERROR)
{ // 根据服务器地址，客户端接口，建立连接
// 建立成功返回0，否则是一个数字对应error
cerr << "Failed to connect to the server." << endl;
closesocket(clientSocket); // 关闭失败的连接
WSACleanup();
}

send(clientSocket, (const char *)&(message.roomId), sizeof(int), 0); // 发送房间号给服务器
cout << "Connected to the server." << endl;
```

服务端通过哈希表将 接口映射到房间号

```
clientSocket = accept(serverSocket, (struct sockaddr *)&clientAddr, &clientAddrLen); // 中
断，直到一个客户端连接上，之后自动填写客户端地址，返回客户端接口
int roomNum = 0; // 读取房间号
recv(clientSocket, (char *)&(roomNum), sizeof(roomNum), 0);
// 在接收到客户端房间号后，将客户端套接字和房间号关联
clientRoomMap[clientSocket] = roomNum;
cout << "roomID:" << clientRoomMap[clientSocket] << " ";
```

广播消息的时候，不会向消息的发起者接口广播，和发起者接口的房间号不一样的消息也不会广播

```
std::unordered_map<SOCKET, int> clientRoomMap;

void BroadcastMessage(const ChatMessage& message, SOCKET senderSocket ) { // 将message发给除了发送者
之外的接口
```

```

        // lock_guard<mutex> lock(mtx);

        for (SOCKET clientSocket : clientSockets) { //遍历所有客户端接口
            if (clientSocket != senderSocket && message.roomId == clientRoomMap[clientSocket]) { //不要发给发送者自己，不要发给
                int sendResult = send(clientSocket, (const char*)&message, sizeof(message), 0); //将message发送给这个接口，flags=0没有额外操作
                if (sendResult == SOCKET_ERROR) { //一般不报错
                    cerr << "Send failed." << endl;
                }
            }
        }
    }
}

```

效果如图，房间2内的消息只有房间2内的其他人可以收到



锁机制

我们之前添加了一个 `vector<SOCKET> clientSockets` 来存储所有客户端的套接字，所以可以使用互斥锁 `mutex mtx` 来保护这个列表，以防止多个线程同时修改它

```

#include <mutex>
mutex mtx; // 用于保护客户端套接字列表的互斥锁

```

在多个用户同时连接，都想要加入接口列表的时候，加上锁

```

        cout << "Client connected from " << inet_ntoa(clientAddr.sin_addr) << ":" <<
            ntohs(clientAddr.sin_port) << endl;
        { //由于在使用 std::lock_guard 时，不需要显式地解锁，因为 std::lock_guard 会在离开其作用域
            //时自动解锁互斥量，所以这里加上一个括号
            lock_guard<mutex> lock(mtx);
            clientSockets.push_back(clientSocket); //所有客户端接口需要保留，后面还有用
        }
    }
}

```

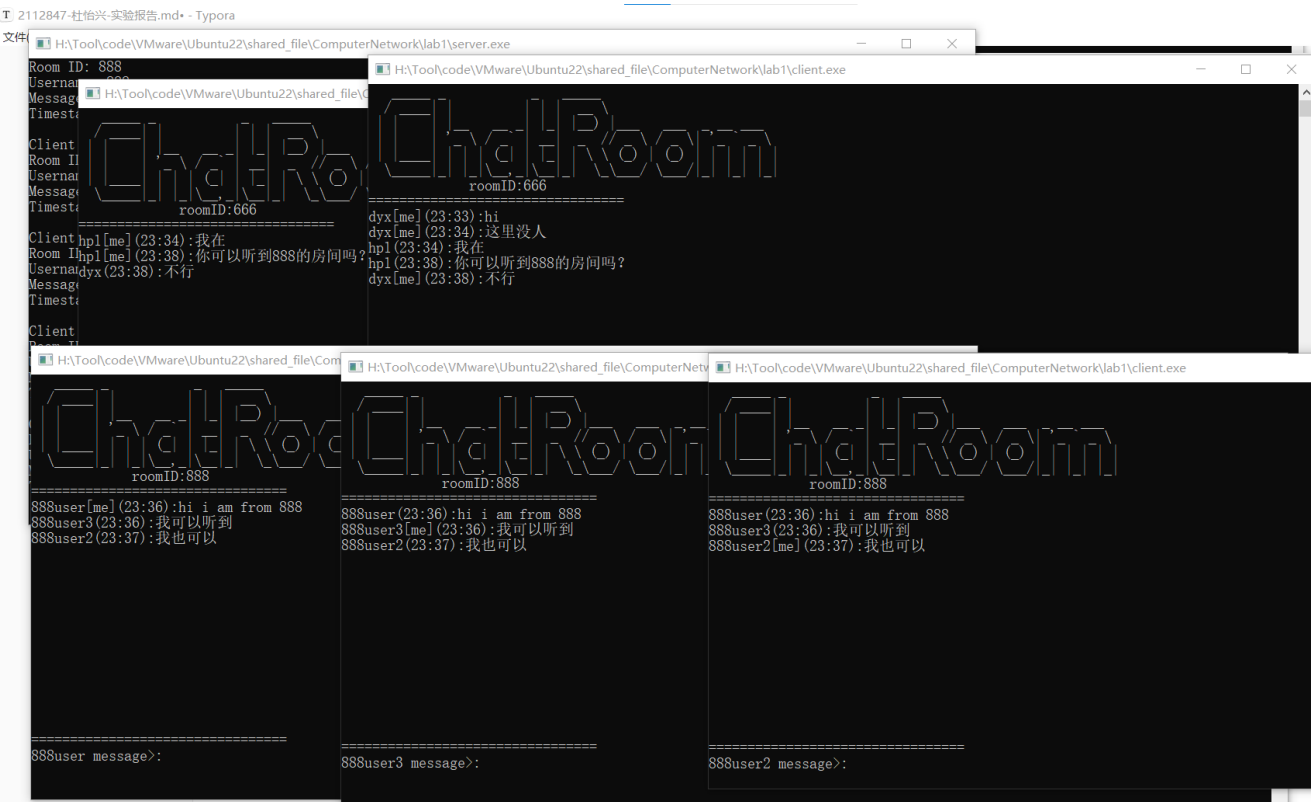
遍历接口列表的时候，加上锁

```
void BroadcastMessage(const ChatMessage& message, SOCKET senderSocket ) { //将message发给除了发送者
之外的接口
    lock_guard<mutex> lock(mtx);
    //后面省略
}
```

lock_guard不需要显式地解锁

实验结果

最后一次测试中，测试了单人发信息，此时服务器可以收到，无人回复。
后来888房间有了3人，聊天彼此可以听到。666房间有2人，聊天彼此可以听到，但是它们的聊天彼此不影响。



```
Room ID: 888
Username:
Message:
Timestamp:

Client:
Room ID:
Username:
Message:
Timestamp:

Client hpl[me] (23:34): 我在
Room ID: hpl[me] (23:38): 你可以听到888的房间吗?
Username: dyx (23:38): 不行
Message:
Timestamp:

Client:
Room ID:
Username:
Message:
Timestamp:

dyx[me] (23:33): hi
dyx[me] (23:34): 这里没人
hpl (23:34): 我在
hpl (23:38): 你可以听到888的房间吗?
dyx[me] (23:38): 不行

roomID: 666

ChatRoom

roomID: 666

888user[me] (23:36): hi i am from 888
888user3 (23:36): 我可以听到
888user2 (23:37): 我也可以

888user message>:

888user (23:36): hi i am from 888
888user3[me] (23:36): 我可以听到
888user2 (23:37): 我也可以

888user3 message>:

888user (23:36): hi i am from 888
888user3 (23:36): 我可以听到
888user2[me] (23:37): 我也可以

888user2 message>:
```

总结

本实验旨在设计和实现一个基于C++语言的聊天应用程序，涉及了网络通信、多线程编程、协议设计和用户界面开发等方面的技术。

总的来说，本实验成功地实现了一个多线程聊天应用程序，具有良好的可扩展性和可维护性。通过协议设计和网络通信，实现了用户间的消息传递。多线程编程确保了并发处理，房间系统和半图形化界面增强了用户体验。实验中的锁机制确保了程序的稳定性和安全性。

改进空间：

发送数据压缩

差错校验

加密通信