计算机网络Lab1

刘宇轩　2012677

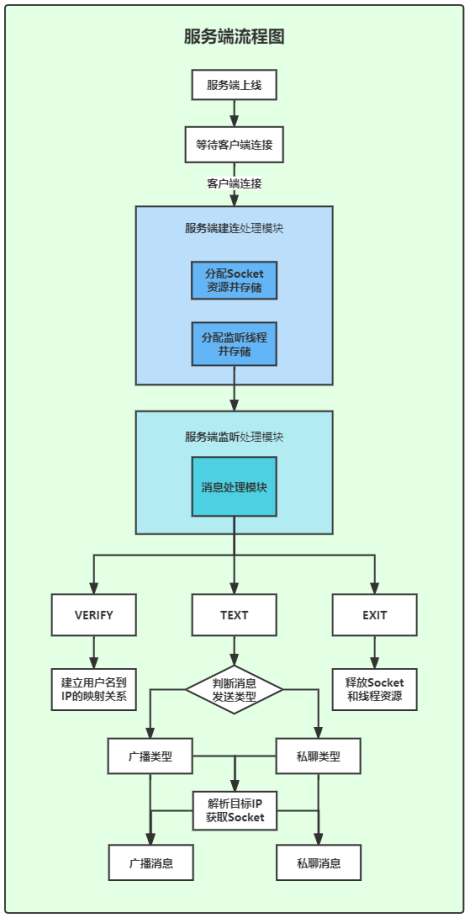
工作总览

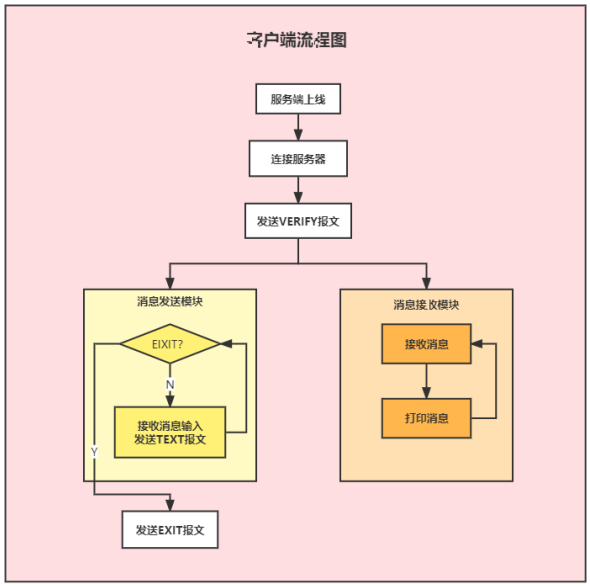
在本次实验中，基于Socket实现了一个简单的多人聊天室，支持多人同时在线聊天，并能够实现消息的群聊和用户私聊。在服务器和客户端的接收处理中采用多线程的方式，能够支持多个用户同时接入，保证了消息的并发性。

程序的整体流程如下。

服务器上线后，会监听客户端的连接请求，在收到连接请求之后，会单独开一条线程处理客户端消息的发送和接收，并且使用客户端的IP信息作为索引，存储客户端的Socket和线程Handle。当服务器接收到用户发来的`VERIFY`类型消息的时候，会建立用户名到IP信息的映射关系，方便客户端使用用户名通信。当服务器接收到用户发来的`TEXT`类型的消息的时候，会首先判断是否群发，然后根据接收端的用户名信息检索对应的Socket，并使用该Socket转发消息。而当服务器接收到用户发来的`EXIT`类型消息的时候，会根据用户名释放掉相应Socket和线程资源。

客户端上线后，会首先和服务器建立连接，建立连接成功之后，会给服务端发送一条`VERIFY`消息，其中包含了用户名。随后根据用户输入的消息以及转发选项，构建`TEXT`类型消息发送给服务器。当用户输入`EXIT`之后，会给服务器发送一条`EXIT`类型的消息，随后客户端下线。





协议设计

在本次简单的多人聊天室的设计中，主要包含了以下三种不同的协议

1. `VERIFY`用户验证协议
2. `TEXT`用户消息协议
3. `EXIT`用户离线协议

报文结构

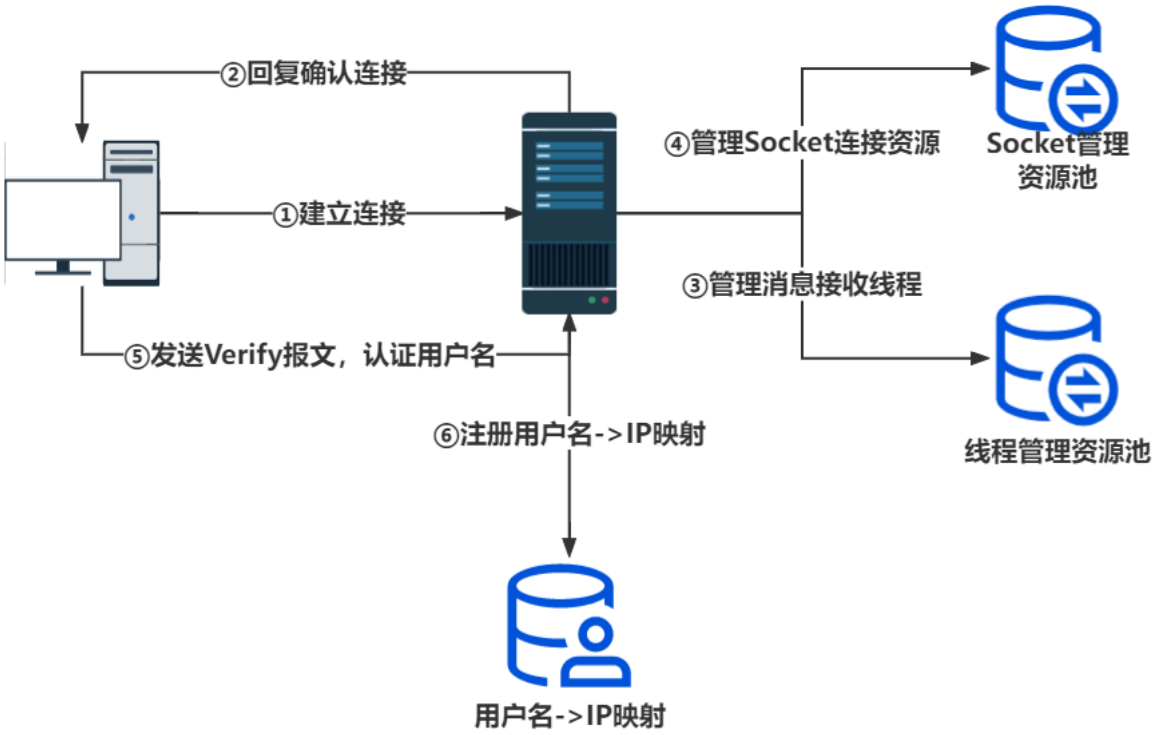
在本次实验中设计了一个报文结构体，用来存储报文的类型，发送方的用户名和IP信息，发送的时间，发送的类型，接收方的用户名和IP信息，消息的内容体。

```
enum class MessageType{
    VERIFY,
    TEXT,
    EXIT
};

struct IP{
    char IPAddress[16]{};
    unsigned short port{};
};

struct Message{
    MessageType type{};
    bool toAll{};
    time_t time{};
    char fromUsername[15]{};
    struct IP fromIP{};
    char toUsername[15]{};
    struct IP toIP{};
    char message[MAX_LEN]{};
};
```

`VERIFY`用户验证协议

服务器在接收到客户端建连之后，会产生一个Socket与客户端进行通信，并且单独申请一条新的线程，用来接收客户端发来的消息，由于此时还没有收到客户端发来的第一条消息，因此为了能够管理客户端所占用的资源，采用建立了以IP地址作为索引的资源管理表。

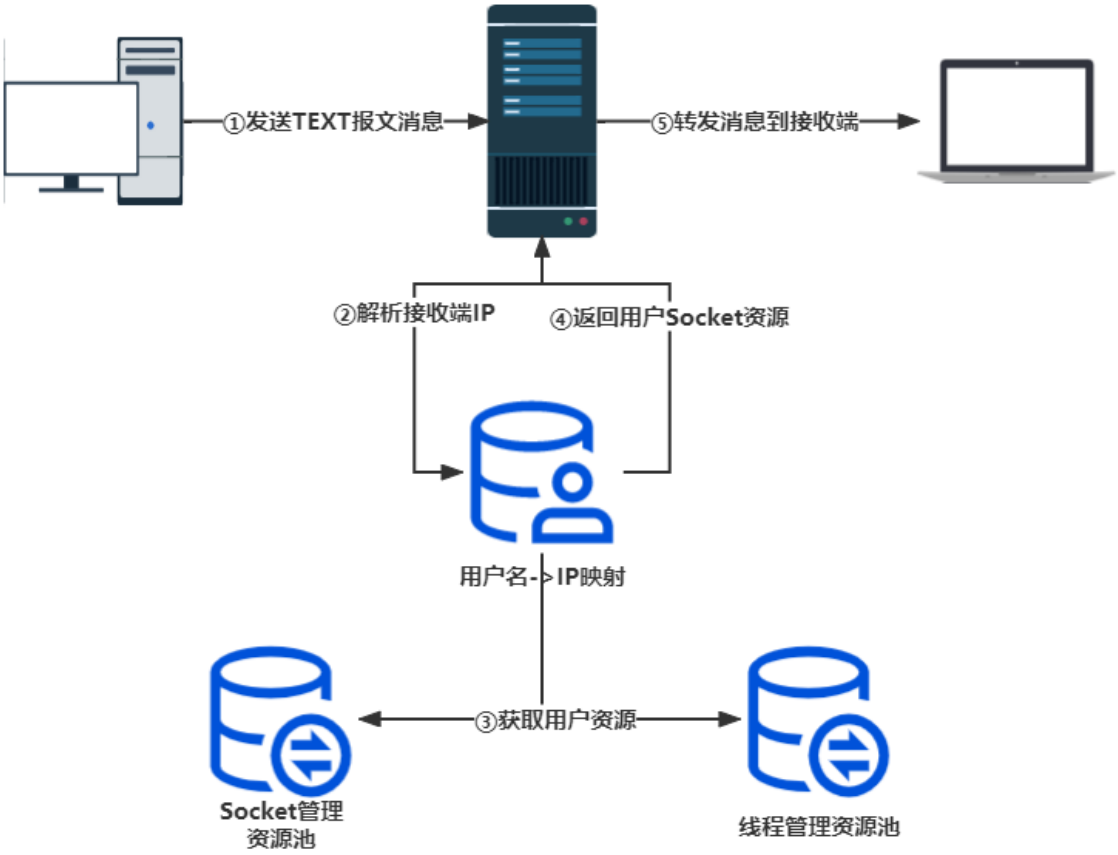
当用户和服务器建立连接之后，会首先发送一条`VERIFY`报文到服务器，报文中的消息类型字段为`VERIFY`，说明当前消息为用户登录验证消息，这条消息中只包含了用户名，用于服务器建立用户名到IP的映射关系。当用户名到IP的映射建立好之后，在后续的通信过程中，客户端就可以只通过输入接收端的用户名将消息发送，而不需要获取接收端的IP信息。



TEXT用户消息协议

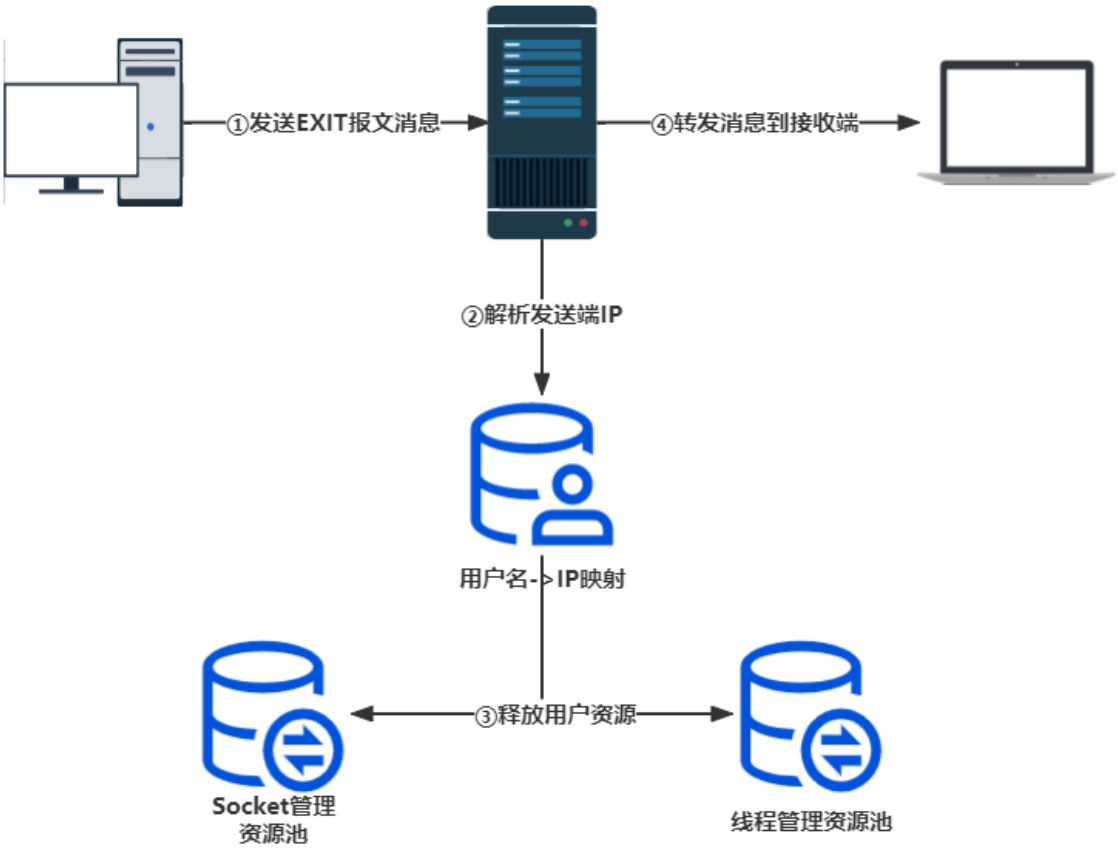
在本次实验中，支持两种不同的消息协议，一种是群聊协议，一种是私聊协议。通过用户在输入消息之后，输入的`opt`字段，来判断用户选择的消息协议。如果用户输入了`-a`，则说明用户希望将消息转发给所有人；如果用户输入`-n=[usernae]`，则说明用户希望将消息单独发送给指定用户。

服务器在接收到客户端发来的`TEXT`类型报文之后，通过之前建立的用户名到IP的映射关系，解析接收端的IP信息，并使用IP信息检索接收端所使用的Socket资源，利用这个Socket资源，将消息发送给接收端。当消息群发的时候，则需要便利整个用户列表，一次获取用户的Socket资源，并将消息逐个转发。



EXIT用户离线协议

当客户端输入了 EXIT 字符串之后，证明客户端即将离线，服务端根据消息中的用户名字段，检索出用户的IP信息，并使用该IP信息，检索出该用户占用的Socket资源和线程资源，将资源释放，然后将该用户的离线消息进行广播。



核心模块解析

服务端建连处理模块

服务器上线之后，会启动一条线程来等待客户建连。当客户连接接入后，会进入到建连处理模块中。在客户首次连接时，服务端只能够获取到客户端的IP信息，而不能够获得到相关的用户名，因此资源管理只能利用IP信息作为索引。

为了服务器能够并发处理多个客户端同时发送消息的情况，在接收到客户端的连接请求之后，服务器会单独开一条线程处理这个客户端消息的接收和发送，为了能够合理的管理线程资源，服务器用IP信息作为索引存储了线程对应的Handle，方便在用户离线之后释放占用的线程资源。对于服务器与客户端通信的Socket同样使用IP信息作为索引进行存储。在之后发送消息的时候，就可以通过IP信息直接检索出对应的Socket，使用此Socket发送消息。

```
/**
 * Thread for listening the connection
 * @param lparam server socket
 * @return
 */
[[noreturn]] DWORD WINAPI listenProc(LPVOID lparam){
    auto serverSock = (SOCKET) (LPVOID) lparam;
    int addressLen = sizeof(SOCKADDR_IN);
    while(true) {
        SOCKADDR_IN fromAddress;
        SOCKET socketConnect = accept(serverSock, (SOCKADDR
*)&fromAddress, &addressLen);
        // deal with the connection
        processConnection(socketConnect, fromAddress);
    }
}

/**
 * deal with the connection
 * @param socketConnect socket connected to the server
 * @param fromAddress connection address
 */
void processConnection(SOCKET socketConnect, SOCKADDR_IN fromAddress)
{
    auto* threadParam = new ThreadParam();
    threadParam->socket = socketConnect;
    threadParam->address = fromAddress;
    HANDLE recvThreadHandle = CreateThread(NULL, (SIZE_T) NULL,
recvProc, (LPVOID) threadParam, 0, 0);
    // get IP from the client connected
    struct IP fromIP;
    strcpy(fromIP.IPAddress,inet_ntoa(fromAddress.sin_addr));
    fromIP.port = ntohs(fromAddress.sin_port);
    std::string userIP = IP2Str(fromIP);
    // insert the handler and connection into the map
```

```
    recvHandlers.insert({userIP, recvThreadHandle});
    connections.insert({userIP, socketConnect});
    std::cout << "[CONNECT_LOG] : {IP : " << userIP << "} connected!"
<< std::endl;
}
```

服务端消息处理模块

服务端接收到客户端发来的消息之后，首先会根据来源地址将发送方的IP信息填到Message的IP数据段中。然后根据解析消息是否会被群发，如果是私聊消息的话，会将根据接收方的用户名，解析出接收方的IP信息，并填写到Message的toIP字段中。在完成了预处理工作之后，会解析消息的类型，并根据不同的消息类型调用不同的功能模块进行处理。

```
/**
 * process the message
 * @param message
 * @param fromAddress
 */
void processMessage(struct Message& message, SOCKADDR_IN fromAddress)
{
    // get IP and port from the fromAddress and set it in the message
    std::string fromIP = std::string(inet_ntoa(fromAddress.sin_addr));
    strcpy(message.fromIP.IPAddress, fromIP.c_str());
    message.fromIP.port = ntohs(fromAddress.sin_port);

    if(!message.toAll){
        std::string toUsername = message.toUsername;
        if(usernameToIP.find(toUsername)==usernameToIP.end()){
            std::cout << "[ERROR_LOG] : {User : " << toUsername << "}
not found!" << std::endl;
            return;
        }
        std::string toIP = usernameToIP[toUsername];
        message.toIP = str2IP(toIP);
    }

    // deal with different message type
    MessageType type = message.type;
    switch(type){
        case MessageType::VERIFY: {
            userVerify(message);
            break;
        }
        case MessageType::TEXT: {
            break;
        }
        case MessageType::EXIT: {
            userExit(message);
            break;
        }
    }
```

```
    printMessage(message);
    broadcastMessage(message);
}
```

对于`VERIFY`类型，服务端需要建立用户名到IP的映射关系。而对于`EXIT`类型，服务端需要释放所有与该客户端绑定的资源。

```
/**
 * user verify : set username to IP
 * @param message
 */
void userVerify(struct Message& message){
    std::string username = message.fromUsername;
    std::string userIP = IP2Str(message.fromIP);
    usernameToIP.insert({username, userIP});
    std::cout << "[VERIFY_LOG] : {User : " << username << "} from {IP : " << userIP << "} join in !" << std::endl;
}

/**
 * user exit : remove the user from the map and release the resources
 * @param message
 */
void userExit(struct Message& message){
    std::string username = message.fromUsername;
    std::string userIP = IP2Str(message.fromIP);
    CloseHandle(recvHandlers[userIP]);
    closesocket(connections[userIP]);
    connections.erase(userIP);
    recvHandlers.erase(userIP);
    usernameToIP.erase(username);
    std::cout << "[EXIT_LOG] : {User : " << username << "} from {IP : " << userIP << "} exit !" << std::endl;
}
```

而对于`TEXT`类型的消息，则需要根据用户是否希望将消息群发分别处理。如果用户希望消息群发，则遍历用户列表，依次解析用户的IP，并根据IP获取到相关的Socket资源，使用该Socket资源，将新构建拷贝的message发送给目标接收端。如果是指定用户，则只需要根据message中的toIP字段解析出用户的Socket即可。

```
/**
 * broadcast message to all users
 * @param message
 */
void broadcastMessage(struct Message& message){
    if(!message.toAll){
        struct Message newMessage{};
        newMessage.type = MessageType::TEXT;
        newMessage.time = message.time;
        strcpy(newMessage.fromUsername, message.fromUsername);
        newMessage.fromIP = message.fromIP;
```

```
        strcpy(newMessage.toUsername, message.toUsername);
        newMessage.toIP = message.toIP;
        SOCKET socketConnect = connections[IP2Str(message.toIP)];
        send(socketConnect, (char *)&message, sizeof(struct Message), 0);
        return ;
    }
    // get IP and send message to all users
    for(const auto& userInfo : usernameToIP){
        if(userInfo.first == std::string(message.fromUsername)){
            continue;
        }
        std::string username = userInfo.first;
        std::string userIP = userInfo.second;
        struct IP toIP = str2IP(userIP);
        struct Message newMessage;
        newMessage.type = message.type;
        newMessage.time = message.time;
        strcpy(newMessage.fromUsername, message.fromUsername);
        newMessage.fromIP = message.fromIP;
        strcpy(newMessage.toUsername, username.c_str());
        newMessage.toIP = toIP;
        strcpy(newMessage.message, message.message);
        SOCKET socket = connections[userIP];
        send(socket, (char *) &newMessage, sizeof(struct Message), 0);
    }
}
```

客户端消息接收模块

为了能够保证客户端在输入的时候也能够接收消息，因此需要为接收消息单独开一条线程，在这个线程中，只轮询等待消息的进入，并打印展示消息内容。

```
/**
 * Thread for receiving message
 * @param lparam ThreadParam
 * @return
 */
[[noreturn]] DWORD WINAPI recvProc(LPVOID lparam){
    auto socket = (SOCKET) (LPVOID) lparam;
    while(online) {
        struct Message msg;
        int recvLen = recv(socket, (char *) &msg, sizeof(struct Message), 0);
        if(recvLen > 0){
            printMessage(msg);
            std::cout << "============================ SEND ============================" << std::endl;
        }else{
            std::cout << "Receive failed" << std::endl;
        }
```

```
    }
}
```

客户端消息发送模块

客户端的消息发送模块要求用户不仅仅输入消息的内容，还需要输入一个`opt`字段，用来表示消息是群发还是私聊。如果用户输入了`-a`，则说明用户希望将消息转发给所有人；如果用户输入`-n=[usernae]`，则说明用户希望将消息单独发送给指定用户。

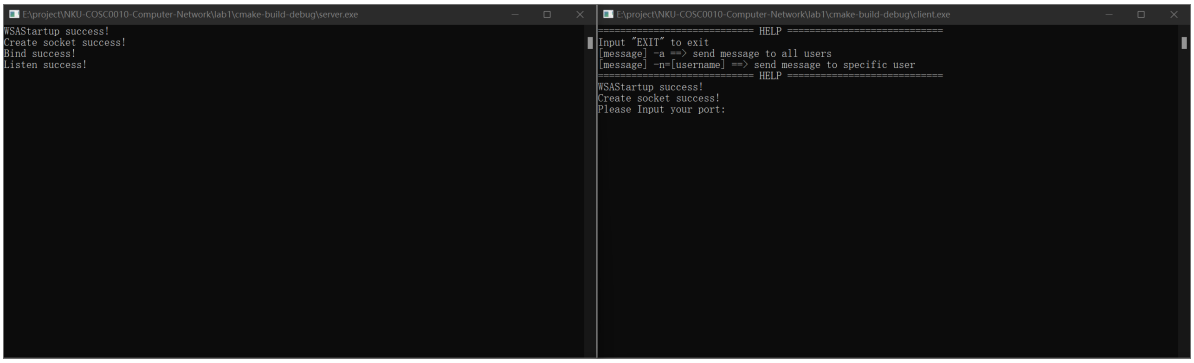
如果用户在输入消息内容的时候输入了 EXIT，证明客户端即将离线，需要给服务器发送一个`EXIT`类型的报文，通知服务器释放相关资源。

```
/**
 * Send message
 * @return
 */
bool sendProc(){
    std::cout << "============================ SEND ============================" << std::endl;
    struct Message msg;
    msg.time = time(nullptr);
    strcpy(msg.fromUsername, userName.c_str());
    std::string option;
    std::cin >> msg.message;
    if(std::string(msg.message) == "EXIT"){
        msg.type = MessageType::EXIT;
        msg.toAll = true;
        online = false;
    } else{
        msg.type = MessageType::TEXT;
        std::cin >> option;
        if(option == "-a"){
            msg.toAll = true;
        } else{
            msg.toAll = false;
            option = option.erase(0, 3);
            strcpy(msg.toUsername, option.c_str());
        }
    }
    int sendLen = send(clientSocket, (char *) &msg, sizeof(struct Message), 0);
    if(sendLen > 0){
        std::cout << "Send success" << std::endl;
    }else{
        std::cout << "Send failed" << std::endl;
    }
    std::cout << "============================ SEND ============================" << std::endl;
    if(std::string(msg.message) == "EXIT"){
        Sleep(500);
        return false;
```

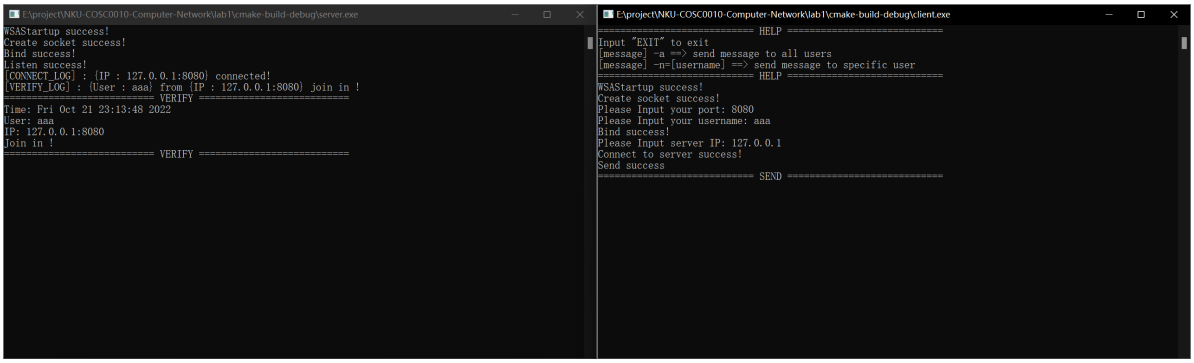
```
37      } else{
38          return true;
39      }
40  }
```

程序效果截图

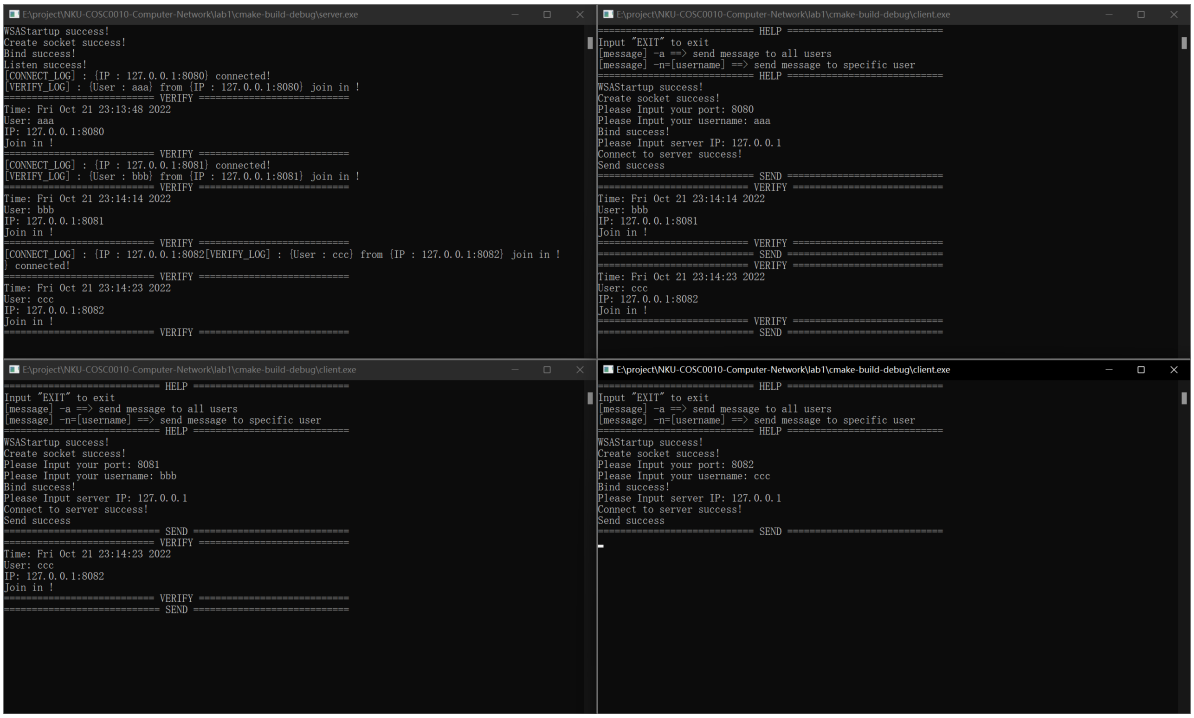
服务端客户端上线



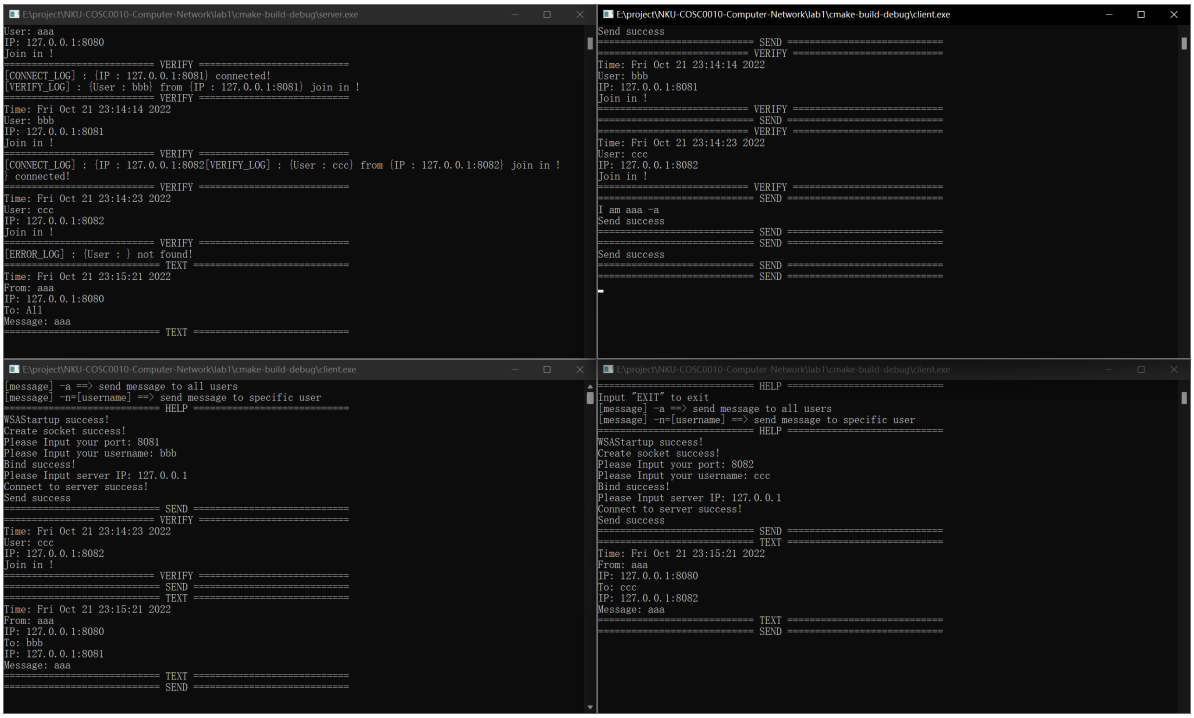
客户端连接服务器



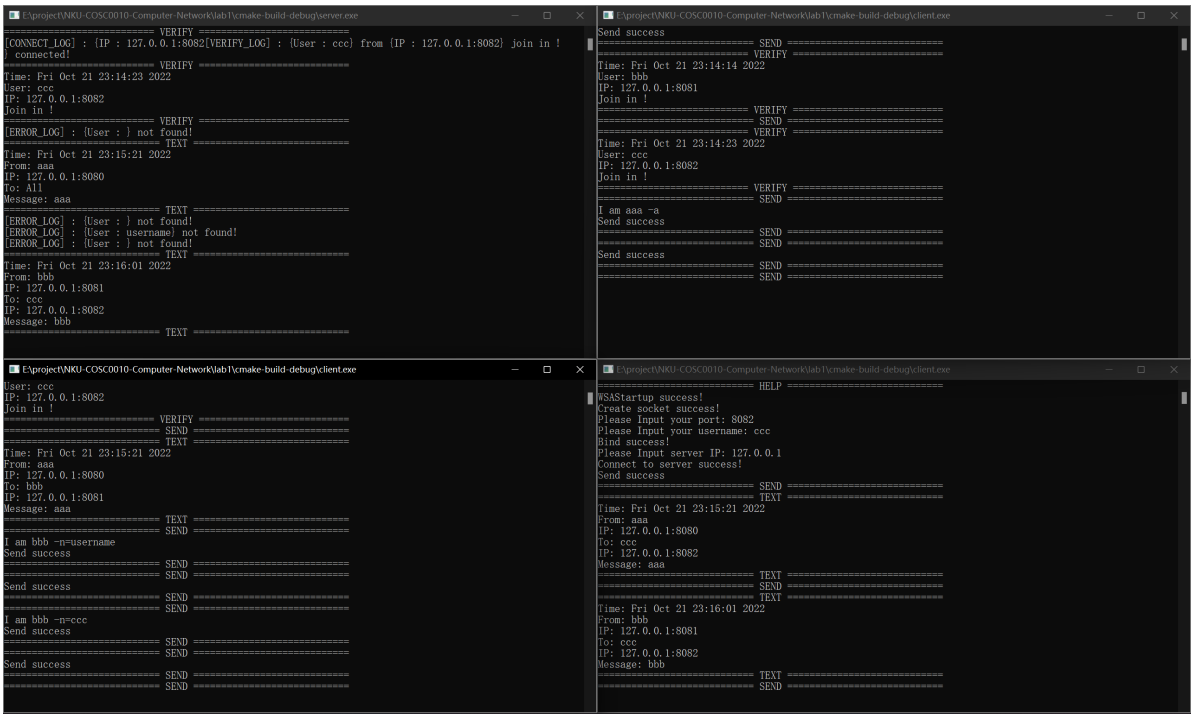
多用户连接



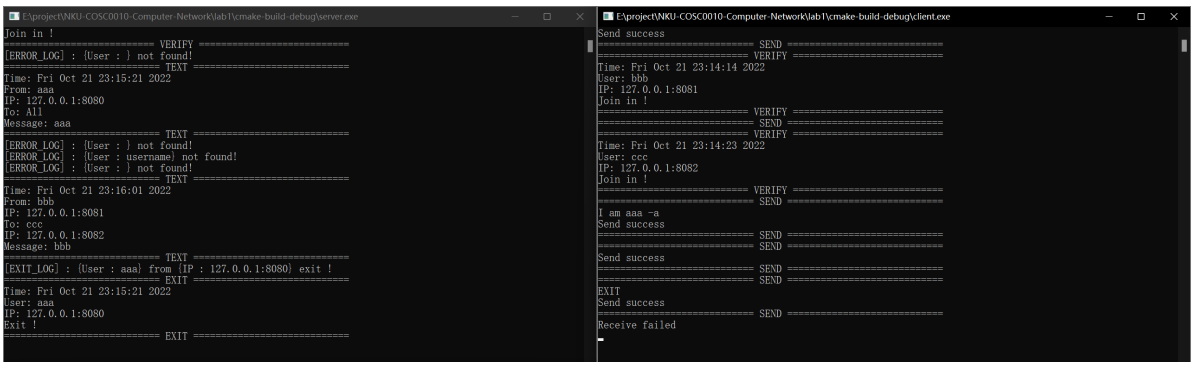
消息群发



消息私聊



用户退出



实验中遇到的问题

在用户退出的时候，有时会发生客户端程序先终止，而send函数并没有执行完，导致服务端并没有接收到客户端的EXIT消息，因此客户端下线后，服务端的监听线程就会出现报错，导致程序异常。为此，客户端在发送了EXIT消息之后，需要等待500ms，保证消息发送成功。