LAB1：利用Socket编写聊天程序

2113824	杨浩甫	计算机科学技术

实验准备

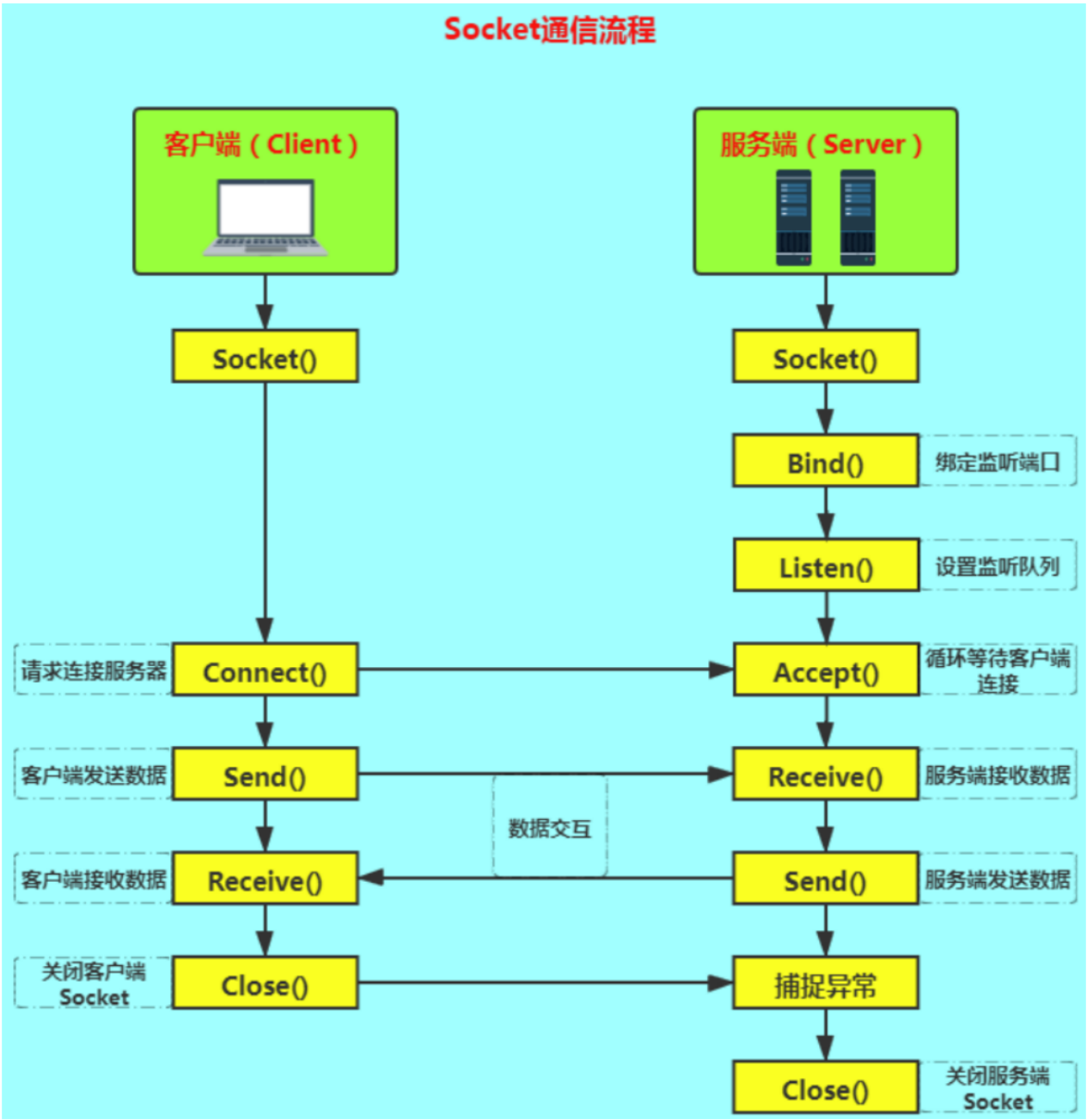
服务器实现目标：

实现了一个基于SOCKET的多线程服务器，用于监听客户端连接和处理消息通信。服务器绑定到特定端口，接受客户端连接，为每个客户端启动独立的线程，实现实时聊天功能，同时广播连接和断开消息给其他客户端。

客户端实现目标：

客户端程序使用WinSock库，允许用户连接到服务器并进行实时聊天。它首先初始化WinSock，创建一个套接字，尝试连接到服务器，然后接受用户输入的用户名并发送到服务器。客户端通过一个独立线程监听服务器消息，并在主线程中发送用户输入的消息。当用户输入"exit"时，客户端关闭连接，并清理WinSock资源。

设计框架：



聊天协议说明

用户验证协议：

1. 用户首次连接到服务器时需要提供一个用户名。
2. 服务器接收用户名并将其与用户的IP地址绑定，用于标识用户身份。

用户消息协议：

1. 用户可以向服务器发送消息，消息可以包括中文或英文文本。
2. 服务器将收到的消息转发给其他已连接的客户端，以便实现聊天功能。
3. 每条消息包含以下信息：
    - 用户发送的消息内容：用户的文本消息，可以是聊天内容。
    - 用户名称：用于标识消息的发送者，确保其他用户知道消息来源。
    - 用户发送消息的时间：用于记录消息发送的时间戳，以便用户和服务器都能知道消息的时间。

用户离线协议：

1. 当用户决定退出聊天时，用户可以发送消息为“exit”。
2. 这个特殊的消息表示用户希望结束聊天，关闭对话框。
3. 服务器会根据接收到的“exit”消息来断开用户与服务器的连接，从而结束聊天会话。

实验设计

服务器端网络连接：

我们采用了常见的套接字编程模式，以便在网络上监听和处理客户端请求。主要涉及下面几个步骤：

1. **初始化网络库：**

   首先使用`WSAStartup`函数来初始化WinSock库，以确保正确初始化网络库和资源分配。

2. **创建套接字：**

使用`socket`函数创建套接字。在这里，我们创建了一个IPv4地址族的流式套接字，并让系统自动选择协议。

3. **绑定服务器：**

   使用`bind`函数将服务器套接字绑定到指定的IP地址和端口。在你的设计中，服务器监听所有可用的IP地址（`INADDR_ANY`），并指定了监听的端口。

4. **进入监听状态：**

   使用`listen`函数将服务器套接字设置为监听状态。这允许服务器等待客户端连接请求。

5. **等待客户端连接：**

   在一个无限循环中，服务器等待客户端的连接请求。当有客户端请求连接时，`accept`函数接受该连接，并创建一个新的套接字，用于与客户端通信。

6. **服务器状态输出：**

   在成功初始化网络库、创建套接字、绑定服务器和进入监听状态后，我们会输出一些系统消息，以指示服务器正在监听特定IP地址和端口，并等待客户端连接请求。

```
    //WinSock库初始化，确保网络库的正确初始化和资源分配。
    WSADATA wsaData;
    int result = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (result != 0) {
        cout << "WSAStartup 初始化失败原因: " << result << endl;
        return 1;
    }

    //创建套接字，IPv4，流式套接字，自动选择协议
    SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (serverSocket == INVALID_SOCKET) {
        cout << "套接字创建失败" << endl;
        WSACleanup();
        return 1;
    }

    sockaddr_in serverAddress;//服务端地址
    serverAddress.sin_family = AF_INET;//连接方式
    serverAddress.sin_port = htons(PORT);//服务器监听端口
    serverAddress.sin_addr.S_un.S_addr = htonl(INADDR_ANY);//指定服务器监听的IP地址(实际就是本机地址)d。

    //绑定服务器
```

```
    result = bind(serverSocket, (SOCKADDR*)&serverAddress,
sizeof(serverAddress));
    if (result == SOCKET_ERROR) {
        cout << "绑定服务器失败" << endl;
        closesocket(serverSocket);
        WSACleanup();
        return 1;
    }

    //进入监听状态
    result = listen(serverSocket, MaxClient);
    if (result == SOCKET_ERROR) {
        cout << "进入监听失败" << endl;
        closesocket(serverSocket);
        WSACleanup();
        return 1;
    }

    cout << "[system]:服务器正在监听 " << IP << ":" << PORT << endl;

    cout << "[system]:正在等待客户端连接o(*^@^*)o" << endl;

    cout << "-----------------------------------------" << endl;
```

客户端网络连接：

这部分的设计与服务器端的设计类似，主要有两点不同：

- 与服务器不同，客户端不需要绑定到特定的IP地址和端口，而是需要指定服务器的地址以进行连接。我们创建了一个`sockaddr_in`结构，设置了服务器地址的各个部分，包括地址族、端口和IP地址。
- 客户端是主动发起连接的一方，而服务器处于被动等待连接的状态。使用`connect`函数，客户端尝试连接到服务器。

```
    WSADATA wsaData;
    int result = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (result != 0) {
        cerr << "WSAStartup 初始化失败:  " << result << endl;
        return 1;
    }
```

```
    SOCKET clientSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (clientSocket == INVALID_SOCKET) {
        cerr << "套接字创建失败" << endl;
        WSACleanup();
        return 1;
    }

    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(PORT);
    serverAddress.sin_addr.S_un.S_addr = inet_addr(IP);

    //与服务器建立连接
    result = connect(clientSocket, (SOCKADDR*)&serverAddress,
sizeof(serverAddress));
    if (result == SOCKET_ERROR) {
        cerr << "服务器连接失败" << endl;
        closesocket(clientSocket);
        WSACleanup();
        return 1;
    }
```

多线程设计：

在服务器运行时，它会不断等待连接，接受连接请求，然后在单独的线程中处理每个客户端的通信。这确保了服务器能够同时处理多个客户端，使我们的聊天程序能够支持多用户。

```
    vector<thread> clientThreads;

    while (true) {
        SOCKET clientSocket = accept(serverSocket, nullptr,
nullptr);
        if (clientSocket == INVALID_SOCKET) {
            cerr << "接受客户端连接失败" << endl;
        }
        else {
            // 启动一个新线程来处理客户端
            //这一行通过emplace_back方法将一个新的线程对象添加到
clientThreads容器中。
```

```
            //这个线程将执行HandleClient函数，处理与刚刚连接的客户端的通信。
            //HandleClient函数将在独立的线程中运行，这样服务器可以同时处理多
个客户端的连接请求。
            clientThreads.emplace_back(HandleClient, clientSocket);
        }
    }
```

同时，在客户端我们会启动一个新线程用于接收服务器消息：

```
thread receiveThread(ReceiveMessages, clientSocket);
```

时间戳设计：

我们通过`localtime`函数获取当前时间戳并将其格式化成可读的日期和时间格式，然后将时间戳与一条消息拼接在一起，以便在聊天应用中显示消息的时间。

```
            // 获取当前时间并格式化
            time_t rawtime;
            struct tm* timeinfo;
            char buffer[80];
            time(&rawtime);
            timeinfo = localtime(&rawtime);
            strftime(buffer, sizeof(buffer), "%Y-%m-%d %H:%M:%S",
timeinfo);

            // 拼接时间和消息
            string fullMessage = string(buffer) + " " +
broadcastMessage;
            cout << fullMessage << endl;
```

客户端消息处理：

消息发送：

我们允许客户端用户在命令行输入消息，通过`send`函数将消息内容（存储在`message`中）以字符串的形式发送到服务器的套接字`clientSocket`。且提供 "exit" 作为退出程序的命令。

```
string message;
while (true) {
    getline(cin, message);
    if (message == "exit") {
        break;
    }
    send(clientSocket, message.c_str(), message.size(), 0);
    cout << endl;
}
```

消息接收：

我们使用函数 `ReceiveMessages` 接收来自服务器的消息，持续监听并将消息在命令行上显示，直到服务器关闭连接。

```
    string message;
    while (true) {
        getline(cin, message);
        if (message == "exit") {
            break;
        }
        send(clientSocket, message.c_str(), message.size(), 0);
        cout << endl;
    }
```

服务器消息处理：

我们在服务器端设计了消息处理函数`HandleClient` 。首先建立客户端与服务器的连接，然后通过接收用户名来标识客户端。一旦标识成功，它允许用户发送消息，并在服务器上广播这些消息，同时也能够处理用户的离线操作。同时，这个函数还负责在每条消息中添加时间戳，以确保消息带有时间信息。

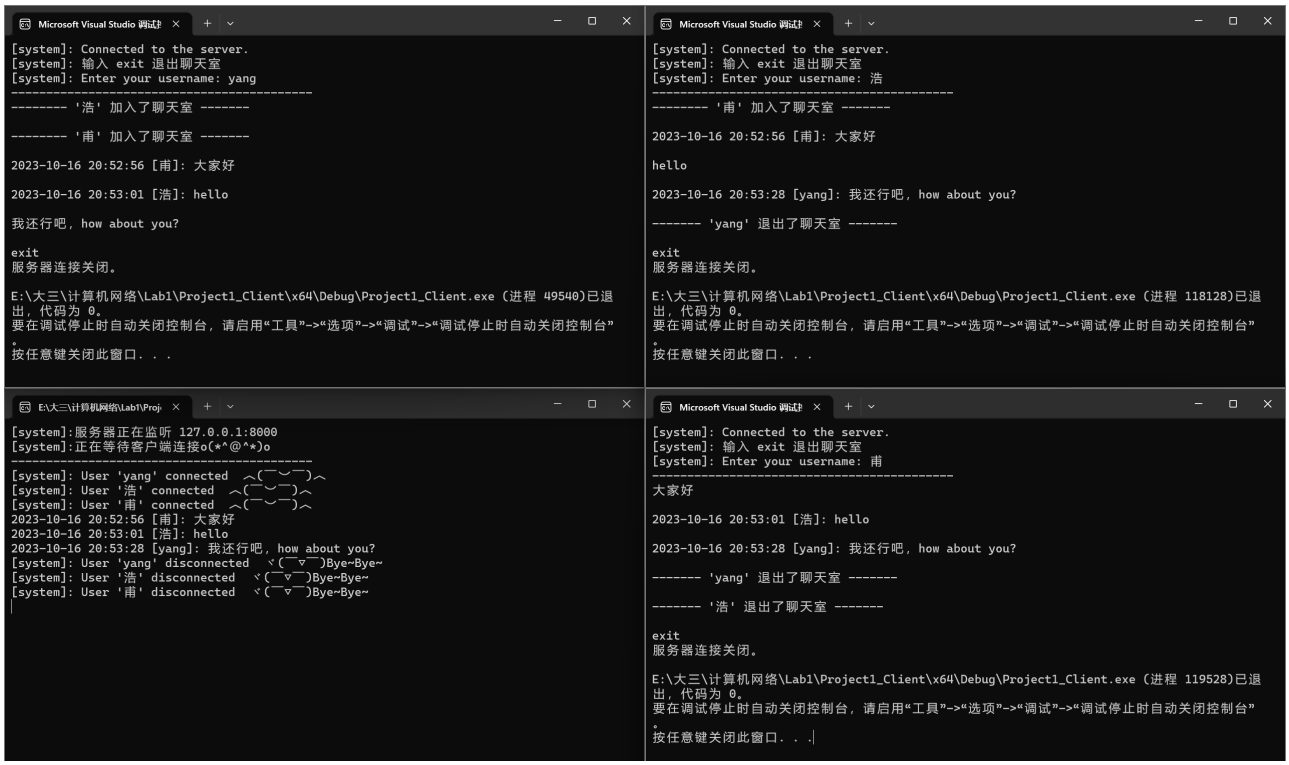
```
void HandleClient(SOCKET clientSocket) {
    char buffer[MaxBufSize];
    string username;
    int bytesReceived;

    // 接收客户端用户名
    bytesReceived = recv(clientSocket, buffer, MaxBufSize, 0);
    if (bytesReceived == SOCKET_ERROR) {
```

```
        cerr << "接受用户名失败" << endl;
    }
    else {
        // 确保接收到的数据以 null 终止
        buffer[bytesReceived] = '\0';
        username = buffer;
        cout << "[system]: User '" << username << "' connected
ヘ(￣～￣)ヘ" << endl;
    }


    // 向其他客户端广播用户连接消息
    {
        lock_guard<mutex> lock(clientMutex);
        for (const ClientInfo& client : connectedClients) {
            if (client.clientSocket != clientSocket) {
                string message = "-------- '" + username + "' 加入了
聊天室 -------";
                send(client.clientSocket, message.c_str(),
message.size(), 0);
            }
        }
        // 添加新用户到已连接客户端列表
        connectedClients.push_back({ clientSocket, username });
    }

    while (true) {
        bytesReceived = recv(clientSocket, buffer, MaxBufSize, 0);
        if (bytesReceived == SOCKET_ERROR || bytesReceived == 0) {
            // 客户端断开连接
            {
                lock_guard<mutex> lock(clientMutex);
                auto it = connectedClients.begin();
                while (it != connectedClients.end()) {
                    if (it->clientSocket == clientSocket) {
                        cout << "[system]: User '" << it->username
<< "' disconnected  ヾ(￣▽￣)Bye~Bye~" << endl;
                        connectedClients.erase(it);
                        break;
                    }
                    ++it;
                }
```

```
            }

            // 向其他客户端广播用户离开消息
            {
                lock_guard<mutex> lock(clientMutex);
                for (const ClientInfo& client : connectedClients) {
                    string message = "------- '" + username + "' 退
出了聊天室 -------";
                    send(client.clientSocket, message.c_str(),
message.size(), 0);
                }
            }

            // 关闭套接字和线程
            closesocket(clientSocket);
            break;
        }
        else {
            // 处理接收到的消息
            string message(buffer, bytesReceived);
            string broadcastMessage = "[" + username + "]: " +
message;
            //cout << broadcastMessage << endl;
            // 向其他客户端广播消息

            // 获取当前时间并格式化
            time_t rawtime;
            struct tm* timeinfo;
            char buffer[80];
            time(&rawtime);
            timeinfo = localtime(&rawtime);
            strftime(buffer, sizeof(buffer), "%Y-%m-%d %H:%M:%S",
timeinfo);

            // 拼接时间和消息
            string fullMessage = string(buffer) + " " +
broadcastMessage;
            cout << fullMessage << endl;

            {
                lock_guard<mutex> lock(clientMutex);
                for (const ClientInfo& client : connectedClients) {
```

```
                    if (client.clientSocket != clientSocket) {
                        send(client.clientSocket,
fullMessage.c_str(), fullMessage.size(), 0);
                    }
                }
            }
        }
    }
}
```

实验结果展示



实验总结

在本次实验中，我获得了宝贵的经验，学习了Socket编程的基本原理和相关函数的使用，同时提高了对C++语言的熟练度。最初，我在处理`send`和`recv`函数时感到困惑，不清楚如何正确使用它们以及它们的参数类型。但通过积极查阅资料和不断的实践，我逐渐理解了这些函数的工作原理，并成功地解决了问题。

这次实验不仅让我初步掌握了网络编程的基本概念，还帮助我了解了套接字、连接、监听、客户端-服务器通信等重要概念。这将有助于我在未来的编程项目中更好地处理网络通信任务。

总之，本次实验丰富了我的编程技能和网络编程知识，增加了自信心，使我能够更好地利用C++和Socket编程来构建网络应用程序。