计算机网络第一次实验报告

邢清画 2211999 物联网工程

一、实验目的

利用 **Socket** 编写一个聊天程序

二、实验要求

1.给出你聊天协议的完整说明。

2.利用 C 或 C++语言，使用基本的 Socket 函数完成程序。不允许使用 CSocket 等封装 后的类编写程序。

3.使用流式套接字、采用多线程（或**多进程**）方式完成程序。

4.程序应有基本的对话界面，但可以不是图形界面。程序应有正常的退出方式。

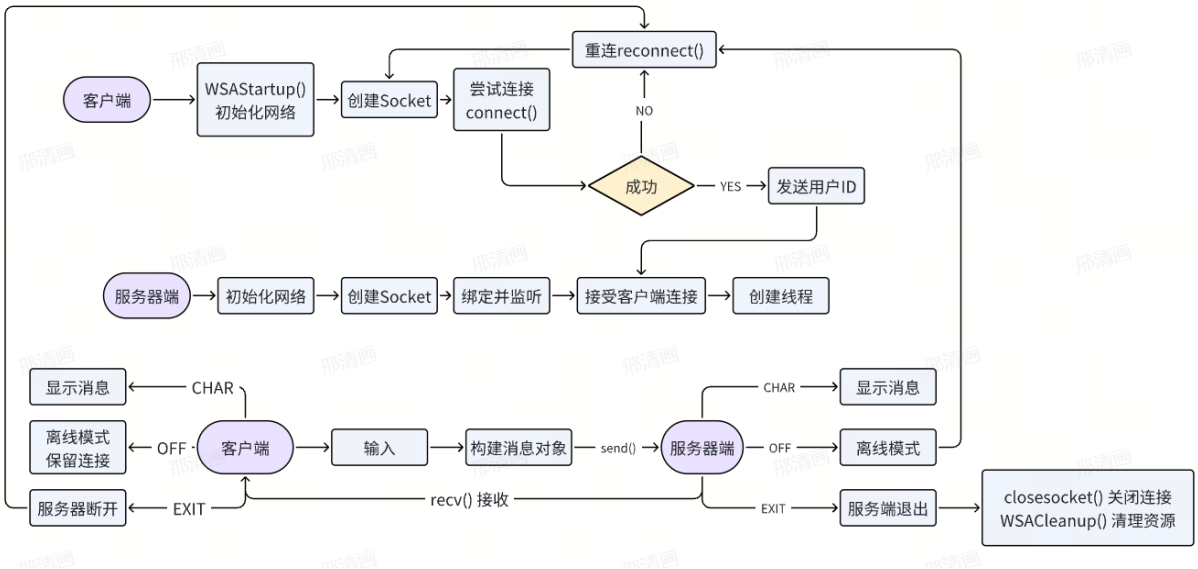
5.完成的程序应能支持多人聊天，支持英文和中文聊天。

6.编写的程序应该结构清晰，具有较好的可读性。

7.在实验中观察是否有数据的丢失，提交源码和实验报告。

三、实验过程

3.1 程序设计流程



3.1.1 服务器端（server)流程

每当一个新的客户端连接时，服务器使用 `CreateThread()` 分配两个独立的线程：

```
CloseHandle(CreateThread(NULL, 0, serveraccept, (LPVOID)&s_accept, 0, 0));
CloseHandle(CreateThread(NULL, 0, serversend, (LPVOID)&s_accept, 0, 0));
```

1. **接收消息线程 (`serveraccept`)**: 专门负责接收来自该客户端的消息。
2. **发送消息线程 (`serversend`)**: 专门负责向该客户端发送消息。

1.0 阶段：一对一聊天功能

1. **初始化网络环境**:
   - 调用 `initwsa()` 函数初始化Winsock环境，并检查是否成功。
   - 通过 `socket(AF_INET, SOCK_STREAM, 0)` 创建Socket。
2. **绑定服务器地址**:
   - 定义 `SOCKADDR_IN server_addr` 结构体并设置服务器地址和端口。
   - 使用 `bind(s_server, (SOCKADDR*)&server_addr, sizeof(server_addr))` 将Socket与地址绑定。
3. **监听连接**:
   - 通过 `listen(s_server, SOMAXCONN)` 使服务器Socket开始监听客户端连接请求。
4. **接受客户端连接**:
   - 使用 `accept(s_server, (SOCKADDR*)&accept_addr, &len)` 接受客户端连接，创建新的Socket `s_accept` 用于通信。
5. **消息传递**:
   - 创建接收消息线程 `CreateThread(NULL, 0, serveraccept, (LPVOID)&s_accept, 0, 0)` 处理客户端发送的消息。
   - 在 `serveraccept` 函数中，使用 `recv(s_accept, recvBuf, sizeof(recvBuf), 0)` 接收消息，并通过 `send()` 将消息返回给相应客户端。
6. **退出处理**:
   - 处理客户端发送的EXIT消息，在接收到退出信号后，调用 `handleClientDisconnect(s_accept)` 清理资源并关闭Socket。

2.0 阶段：多人聊天功能

1. **维护在线用户列表**:
   - 使用 `map<SOCKET, string> clientSockets` 存储当前在线客户端的Socket和对应的用户名。
   - 在 `handleClientConnect(SOCKET s_accept, const string& nickname)` 中将新用户的信息存储到 `clientSockets` 中。
2. **广播消息**:
   - 在 `serveraccept` 函数中，接收到CHAT类型的消息后，使用循环遍历 `clientSockets`，通过 `send()` 将消息广播给所有在线用户。
3. **用户上线和下线通知**:
   - 在用户连接时调用 `broadcastOnlineCount()` 广播当前在线人数信息。
   - 在 `handleClientDisconnect(SOCKET s_accept)` 中更新在线人数，并向所有客户端广播在线人数变化。
4. **消息类型处理**:

- 在 `serveraccept` 函数中，根据接收到的消息类型（如CHAT、OFFLINE、EXIT）进行相应处理，确保不同类型的消息正确响应。

3.0阶段：完善的功能与逻辑

1. **在线人数统计**:
   - 在 `broadcastOnlineCount()` 函数中，通过字符串流构建系统消息，通知所有客户端当前在线人数。
2. **离线和重连逻辑**:
   - 在 `serveraccept` 函数中处理OFFLINE消息，标记客户端为离线状态并通知其他用户。
   - 客户端若进入离线状态，服务器应记录该状态并在后续重连时恢复状态。
3. **优化消息结构**:
   - 通过 `msgToString(const message& m)` 和 `stringToMsg(const string& s)` 函数实现消息的序列化与反序列化，确保消息在客户端和服务器之间能够正确传递。
4. **结束连接的完善处理**:
   - 在接收到EXIT消息后，调用 `handleClientDisconnect(s_accept)` 处理客户端断开连接，并通过 `closesocket(s_accept)` 关闭Socket，确保资源的正确释放。

3.1.2 客户端(client)流程

客户端的实现中，主线程负责发送消息，而一个独立的线程（`clientaccept`）负责接收服务器发送的消息。**不需要**为每个用户分配多个接收线程。客户端的接收线程只是负责从服务器接收所有的广播消息，包括其他用户发送的聊天消息。

1.0 阶段：一对一聊天功能

1. **初始化网络环境**:
   - 调用 `initwsa()` 函数初始化Winsock环境，并检查初始化结果。
2. **获取用户输入**:
   - 提示用户输入昵称并存储为 `userID`，再提示输入要连接的服务器IP地址。
3. **建立连接**:
   - 创建Socket，使用 `socket(AF_INET, SOCK_STREAM, 0)` 创建Socket。
   - 通过 `connect(s_server, (SOCKADDR*)&server_addr, sizeof(SOCKADDR))` 与服务器建立连接。
4. **发送用户昵称**:
   - 连接成功后，通过 `send(s_server, userID.c_str(), userID.size(), 0)` 将用户昵称发送给服务器，以完成用户登录。
5. **消息发送与接收**:
   - 创建接收消息线程 `CreateThread(NULL, 0, clientaccept, (LPVOID)&s_server, 0, 0)`，负责接收服务器发送的消息。
   - 在主线程中，通过 `cin.getline(in, 1000)` 获取用户输入并发送消息给服务器。
6. **退出处理**:

- 当用户输入特定命令（如“exit”）时，向服务器发送EXIT消息，并调用 `closesocket(s_server)` 关闭Socket，结束通信。

2.0 阶段：多人聊天功能

1. **扩展用户输入处理**:
   - 客户端在发送消息时，不再仅发送给单一服务器，而是实现广播消息的功能。
   - 通过线程不断监听服务器返回的消息，实时更新用户界面。
2. **接收消息的处理**:
   - 在 `clientaccept` 函数中，使用 `recv(sockConn, recvBuf, 1000, 0)` 接收服务器发送的消息，并根据消息类型（如CHAT、SYSTEM_MSG等）进行处理。
3. **显示聊天记录**:
   - 当接收到CHAT消息时，格式化输出消息并显示在用户界面中。
4. **处理系统消息**:
   - 通过判断消息类型，处理服务器广播的在线人数或离线通知，确保用户可以看到当前聊天室状态。

3.0 阶段：完善的功能与逻辑

1. **离线与重连逻辑**:
   - 客户端可通过输入“off”命令进入离线模式，标记 `offlineMode` 为true，并通知服务器。
   - 当与服务器的连接断开后，通过 `reconnect(s_server, server_addr)` 函数尝试重新连接。
2. **重连处理**:
   - 在 `reconnect()` 函数中，使用**循环尝试重连**，成功后再次发送用户昵称给服务器，以恢复在线状态。
3. **优化消息结构**:
   - 通过 `msgToString(message m)` 和 `stringToMsg(string s)` 函数实现消息的**序列化与反序列化**，确保消息能够在客户端与服务器之间正确传递。
4. **结束连接的完善处理**:
   - 在收到退出或断线消息时，客户端应确保正确关闭Socket并释放资源，通过调用 `closesocket(s_server)` 完成连接清理。

四、实验分析

4.1 聊天协议设计

1.协议类型与实现方式

基于TCP协议的流式Socket，TCP的可靠性和面向连接的特性确保了消息的顺序、完整性传输，非常适合像聊天这种需要实时可靠通信的场景。可以通过 `send()` 和 `recv()` 函数进行消息的发送和接收，消息以字节流的形式传输。（通过 `socket(AF_INET, SOCK_STREAM, 0)` 创建了**TCP流式Socket**）

```
s_server = socket(AF_INET, SOCK_STREAM, 0);  // 创建TCP流式Socket
```

连接过程:

1. 服务器端：使用 `listen()` 函数将服务器Socket设为监听状态，准备接受TCP连接请求； `accept()` 函数接受来自客户端的TCP连接。

   ```
   listen(s_server, SOMAXCONN);  // 服务器开始监听
   s_accept = accept(s_server, (SOCKADDR*)&accept_addr, &len);  // 接受客户端连接
   ```

2. 客户端：使用 `connect()` 函数与服务器建立TCP连接。

   ```
   connect(s_server, (SOCKADDR*)&server_addr, sizeof(SOCKADDR));  // 客户端与服务器建立TCP连接
   ```

基于字符串序列化和反序列化的方式实现。每个消息在传输时被转换为一个字符串，并使用换行符（`\n`）来分隔不同的字段。该协议在客户端与服务器之间以字符串形式传递消息，并通过解析字符串来实现消息的处理。

实现方式:

- 每个消息在传输之前，通过 `msgToString()` 函数将 `message` 结构体序列化为字符串。
- 消息接收到后，使用 `stringToMsg()` 函数将字符串解析为 `message` 结构体，再根据消息的类型进行相应处理。

2. 消息结构

定义了一种消息结构 `message`，包含了以下四个字段：

```
struct message {
    Type type;     // 消息类型，标识消息的功能（用于区分不同的功能（聊天、系统通知、用户状态变化等）
    string msg;    // 消息内容，包含聊天信息或系统通知
    string name;   // 发送消息的用户名
    string time;   // 消息发送的时间戳
};
```

3. 消息序列化与反序列化

消息序列化: 在发送消息之前，调用 `msgToString()` 函数将 `message` 结构体转换为字符串。各个字段之间使用 `\n` 作为分隔符，以确保接收方能够正确解析消息。

```
string msgToString(const message& m) {
    stringstream s;
    s << m.type << "\n" << m.name << "\n" << m.msg << "\n" << m.time << "\n";
    return s.str();
}
```

消息反序列化: 接收到消息后，调用 `stringToMsg()` 函数将字符串解析为 `message` 结构体。通过查找 `\n` 分隔符来分割不同的字段。

```
message stringToMsg(const string& s) {
    message m;
    int pos1 = s.find('\n');
    int pos2 = s.find('\n', pos1 + 1);
    int pos3 = s.find('\n', pos2 + 1);
    int pos4 = s.find('\n', pos3 + 1);

    m.type = static_cast<Type>(stoi(s.substr(0, pos1)));
    m.name = s.substr(pos1 + 1, pos2 - pos1 - 1);
    m.msg = s.substr(pos2 + 1, pos3 - pos2 - 1);
    m.time = s.substr(pos3 + 1, pos4 - pos3 - 1);

    return m;
}
```

4. 消息类型处理

协议中定义了不同类型的消息，通过枚举类型 `Type` 进行管理。具体的消息类型包括：

```
enum Type {
    CHAT = 1,        // 普通聊天消息
    EXIT = 2,        // 客户端请求退出
    OFFLINE = 3,     // 客户端暂时离线
    SYSTEM_MSG = 4   // 系统消息，服务器广播在线人数等信息
};
```

CHAT: 用于普通的聊天消息，客户端发送消息给服务器，服务器将消息广播给其他在线用户。

EXIT: 当客户端希望完全退出时，发送EXIT消息。服务器接收到该消息后，断开与该客户端的连接，并通知其他用户。

OFFLINE: 客户端暂时离线时发送的消息，服务器接收到后不关闭连接，但会通知其他用户该客户端暂时离线。

SYSTEM_MSG: 服务器发送的系统消息，通常用于通知当前在线人数、用户上线或下线等信息。该消息不会由客户端发送。

5. 消息处理流程

服务器和客户端在接收到消息后，会根据 `message` 中的 `type` 字段来决定如何处理消息：

- **服务器端**:
  - 服务器端的 `serveraccept` 线程在接收到消息后，通过 `stringToMsg()` 将字符串转换为 `message` 结构体，并根据 `type` 字段执行相应的操作。
  - 对于 `CHAT` 类型的消息，服务器将消息广播给所有在线用户。
  - 对于 `EXIT` 或 `OFFLINE` 消息，服务器会执行相应的用户状态管理操作，如断开连接或更新在线状态。

```
if (r.type == CHAT) {
    // 广播聊天消息给所有客户端
    for (const auto& client : clientSockets) {
        send(client.first, recvBuf, recvLen, 0);
    }
} else if (r.type == EXIT) {
    // 处理客户端退出
    handleClientDisconnect(s_accept);
}
```

客户端:

- 客户端的 `clientaccept` 线程在接收到消息后，同样会通过 `stringToMsg()` 进行解析，并根据 `type` 字段处理消息。
- 对于 `CHAT` 消息，客户端会显示聊天内容；对于 `SYSTEM_MSG`，客户端会显示系统通知，如在线人数等。

```
switch (r.type) {
case CHAT:
    cout << r.time << "【" << r.name << "】:" << r.msg << endl;
    break;
case SYSTEM_MSG:
    cout << "【系统消息】: " << r.msg << endl;
    break;
}
```

6. 特殊功能设计

- **在线人数统计**:
  - 服务器端通过 `broadcastOnlineCount()` 函数定期向所有客户端发送系统消息，通知当前在线人数。这种系统消息被标记为 `SYSTEM_MSG` 类型。
- **离线和重连逻辑**:
  - 客户端可以通过发送 `OFFLINE` 消息通知服务器暂时离线。服务器接收到该消息后，记录客户端的离线状态并通知其他用户。
  - 客户端进入离线模式后会自动尝试重连，当成功重连后，客户端会重新发送 `userID` 给服务器，以恢复在线状态。

4.2 关键函数

1. `initwsa()`

这是网络编程中的标准操作，必须在任何网络操作之前执行，确保Winsock能够正常工作。

功能:初始化网络环境，它是聊天程序启动的第一步。

```
int initwsa() {
    WSADATA wsaDATA;
    if (!WSAStartup(MAKEWORD(2, 2), &wsaDATA)) {
        cout << "||【系统消息】:初始化网络环境成功!!" << endl;
        return 1;
    } else {
        cout << "||【系统消息】:初始化网络环境失败!!" << endl;
        return 0;
    }
}
```

实现:调用了 `WSAStartup()` 函数来启动Winsock库，并检查返回值，判断是否成功初始化。成功时返回1，否则输出错误信息并返回0。

2. `msgToString(const message& m)`

该函数是消息序列化的核心，确保服务器和客户端之间能够通过字符串传递消息，并正确解析。

功能:将 `message` 结构体转换为字符串，以便通过Socket进行传输。

```
string msgToString(const message& m) {
    stringstream s;
    s << m.type << "\n" << m.name << "\n" << m.msg << "\n" << m.time << "\n";
    return s.str();
}
```

实现:使用 `stringstream` 将消息的各个字段（`type`，`name`，`msg`，`time`）按顺序拼接为字符串，每个字段以 `\n` 分隔。

3. `stringToMsg(const string& s)`

这是反序列化函数，用于将字符串转换为可用的消息对象。与 `msgToString()` 配合使用，确保消息能够正确传输和处理。

功能:将收到的字符串消息转换回 `message` 结构体，以便在程序中进一步处理。

```
message stringToMsg(const string& s) {
    message m;
    int pos1 = s.find('\n');
    int pos2 = s.find('\n', pos1 + 1);
    int pos3 = s.find('\n', pos2 + 1);
    int pos4 = s.find('\n', pos3 + 1);

    m.type = static_cast<Type>(stoi(s.substr(0, pos1)));
    m.name = s.substr(pos1 + 1, pos2 - pos1 - 1);
    m.msg = s.substr(pos2 + 1, pos3 - pos2 - 1);
    m.time = s.substr(pos3 + 1, pos4 - pos3 - 1);

    return m;
}
```

实现:通过查找字符串中的换行符 `\n`，依次解析出 `type`，`name`，`msg` 和 `time` 字段，并构建 `message` 结构体。

4. `handleClientConnect()`

确保**新连接的客户端**信息能安全地被保存，并触发广播当前在线人数给其他客户端。

功能:当客户端成功连接时，服务器将该客户端加入 `clientSockets`，并增加在线人数。

```
void handleClientConnect(SOCKET s_accept, const string& nickname) {
    {lock_guard<mutex> lock(clientSocketsMutex);
     clientSockets[s_accept] = nickname; // 保存socket与昵称的对应关系
    }
    {lock_guard<mutex> lock(onlineCountMutex);
     onlineCount++;}// 新客户端连接，在线人数增加
    broadcastOnlineCount();  // 广播在线人数
}
```

实现:使用**互斥锁** `clientSocketsMutex` 来安全访问 `clientSockets`，**防止多线程同时修改数据;**将客户端Socket和昵称存入 `clientSockets`，并更新在线人数 `onlineCount`。

5. `handleClientDisconnect(SOCKET s_accept)`

确保客户**端正常断开连接**时，服务器能正确地维护客户端信息和在线人数。

功能:当客户端断开连接时，移除该客户端的信息，并广播在线人数的更新。

```
void handleClientDisconnect(SOCKET s_accept) {
    {lock_guard<mutex> lock(clientSocketsMutex);
        clientSockets.erase(s_accept);// 移除socket
    }
    {lock_guard<mutex> lock(onlineCountMutex);
        onlineCount--; // 客户端断开，在线人数减少
    }
    cout << "【系统消息】: 客户端断开连接，当前在线人数: " << onlineCount << endl;
    broadcastOnlineCount();
}
```

实现:使用**互斥锁**保护对 `clientSockets` 和 `onlineCount` 的修改，确保在**并发情况**下不会出现**数据竞争**问题。移除该客户端的Socket，并更新在线人数。

6. `broadcastOnlineCount()`

服务器广播功能的核心，用于更新并通知所有用户当前的在线情况。

功能:广播当前在线人数给所有已连接的客户端。

```
void broadcastOnlineCount() {
    lock_guard<mutex> lock(clientSocketsMutex);// 保证广播期间对socket map的安全访问

    stringstream ss;
    ss << "【系统消息】: 当前在线人数: " << onlineCount;
```

```
    message systemMessage;
    systemMessage.type = SYSTEM_MSG;
    systemMessage.name = "系统";
    systemMessage.msg = ss.str();
    systemMessage.time = getCurrentTime();
    string systemMsgString = msgToString(systemMessage);

    for (const auto& client : clientSockets) {
        send(client.first, systemMsgString.c_str(), systemMsgString.size(), 0);
    }
}
```

实现:构建系统消息 `SYSTEM_MSG`，通知所有客户端当前的在线人数。遍历 `clientSockets`，通过 `send()` 向每个客户端发送该消息。

7. `serveraccept(LPVOID IpParameter)`

服务器处理客户端通信的**主要线程**，确保客户端的**消息能够及时处理和转发。**

功能:服务器接收客户端发送的消息，并根据消息类型进行处理（聊天消息、下线消息等）。

```
// 接收消息线程
DWORD WINAPI serveraccept(LPVOID IpParameter) {
    SOCKET s_accept = *(SOCKET*)IpParameter;
    char recvBuf[1000];
    memset(recvBuf, 0, sizeof(recvBuf));
    int recvLen = 0;
    bool clientOnline = true;  // 维护当前客户端的在线状态
    bool clientOffline = false;  // 用于追踪客户端是否暂时离线

    while (clientOnline) {
        recvLen = recv(s_accept, recvBuf, sizeof(recvBuf), 0);

        if (recvLen == SOCKET_ERROR && WSAGetLastError() == WSAEWOULDBLOCK) {
            // 处理非阻塞错误或暂时不可读情况
            continue;
        }
        else if (recvLen <= 0) {
            if (clientOffline) {
                // 如果客户端是处于 OFFLINE 状态，可能发生超时或未响应，处理为断开
                cout << "【系统消息】: 客户端已超时断开连接!!" << endl;
                handleClientDisconnect(s_accept);
                clientOnline = false;
            }
            else {
                // 客户端没有发送离线消息，直接断开
                cout << "【系统消息】: 客户端断开连接或连接失败!!" << endl;
                handleClientDisconnect(s_accept);
                clientOnline = false;
            }
        }
        else {
```

```
            // 正常处理收到的消息
            string s(recvBuf, recvLen);  // 确保处理实际收到的字节
            message r = stringToMsg(s);

            if (r.type == CHAT) {
                cout << r.time << "【" << r.name << "】:" << r.msg << endl;
                // 广播消息给所有客户端
                lock_guard<mutex> lock(clientSocketsMutex);
                for (const auto& client : clientSockets) {
                    send(client.first, recvBuf, recvLen, 0);
                }
            }
            else if (r.type == OFFLINE) {
                // 客户端发送了 OFFLINE 消息，暂时离线
                cout << "【系统消息】: 客户端【" << r.name << "】暂时离线" << endl;
                clientOffline = true;  // 标记客户端为离线
            }
            else if (r.type == EXIT) {
                // 客户端请求退出，处理退出
                cout << "【系统消息】: 客户端【" << r.name << "】已退出" << endl;
                handleClientDisconnect(s_accept);  // 完全处理客户端断开连接
                clientOnline = false;  // 退出该客户端的循环
            }
        }
        memset(recvBuf, 0, sizeof(recvBuf));  // 清空缓冲区
    }
    closesocket(s_accept);  // 关闭该客户端的socket
    return 0;
}
```

实现:线程循环等待客户端消息，使用 `recv()` 函数接收消息并根据不同的 `type` 执行不同的操作，如广播聊天消息或处理客户端断线。当接收到 `CHAT` 类型消息时，将消息广播给其他客户端；当接收到 `EXIT` 类型消息时，断开连接（同时更新人数）。

8. `clientaccept(LPVOID IpParameter)`

该线程确保客户端能够持续接收服务器的消息，并实时更新用户界面。

功能:客户端从服务器接收消息，并根据消息类型显示聊天内容或处理系统消息。

```
// 负责接收消息的线程
DWORD WINAPI clientaccept(LPVOID IpParameter) {
    SOCKET sockConn = *(SOCKET*)IpParameter;
    char recvBuf[1000];  // 接收消息的缓冲区
    memset(recvBuf, 0, sizeof(recvBuf));
    int recvLen = 0;

    // 当客户端还在线时，持续接收消息
    while (flag) {
        recvLen = recv(sockConn, recvBuf, 1000, 0);  // 最多接收1000字节的数据
        if (recvLen <= 0) {
            if (offlineMode) {
```

```
                // 如果处于离线状态，继续尝试重连
                cout << "-----------------------------------------------------" <<
endl;
                cout << "【系统消息】:与服务器断开连接，尝试重连..." << endl;
                reconnect(sockConn, *(SOCKADDR_IN*)IpParameter);  // 调用自动重连
            }
            else {
                cout << "-----------------------------------------------------" <<
endl;
                cout << "【系统消息】:与服务器断开连接!" << endl;
                closesocket(sockConn);
                flag = 0;  // 设置为退出状态
                break;
            }
        }
        else {
            string s = recvBuf;
            message r = stringToMsg(s);

            // 输出消息类型以便调试
            cout << "【调试】: 收到消息类型: " << r.type << endl;

            switch (r.type) {
            case CHAT:
                cout << r.time << "【" << r.name << "】:" << r.msg << endl;
                break;
            case OFFLINE:
                cout << "-----------------------------------------------------" <<
endl;
                cout << "【系统消息】:服务端已离线! 摁ENTER 关闭连接!" << endl;
                flag = 0;
                break;
            case EXIT:
                cout << "-----------------------------------------------------" <<
endl;
                cout << "【系统消息】:服务端已下线! 摁ENTER 关闭连接!" << endl;
                flag = 0;
                break;
            case SYSTEM_MSG:
                // 处理系统消息，显示当前在线人数
                cout << "【系统消息】: " << r.msg << endl;
                break;
            }
        }
        memset(recvBuf, 0, sizeof(recvBuf));  // 清空接收缓冲区
    }
    return 0;
}
```

实现:客户端使用 `recv()` 函数接收服务器的消息，并根据 `type` 类型显示聊天消息或系统通知。

9. `serversend(LPVOID IpParameter)`

该线程负责将服务器端输入的消息发送给所有客户端，从而实现服务器与客户端的双向通信。

功能:服务器端负责发送消息给所有已连接的客户端，通常是管理员通过服务器端输入并广播给所有用户。

```
// 发送消息线程
DWORD WINAPI serversend(LPVOID IpParameter) {
    SOCKET s_accept = *(SOCKET*)IpParameter;
    bool clientOnline = true;  // 独立控制每个客户端的在线状态

    while (clientOnline) {
        char send_buf[1000];
        memset(send_buf, 0, sizeof(send_buf));
        message m;
        m.name = "管理员";

        char in[1000];
        cin.getline(in, 1000);
        m.msg = in;
        m.time = getCurrentTime();
        if (m.msg == "off" || m.msg == "OFF") {
            m.type = OFFLINE;
            clientOnline = false;  // 仅当前客户端设置为离线
        }
        else if (m.msg == "exit" || m.msg == "EXIT") {
            m.type = EXIT;
            clientOnline = false;  // 仅当前客户端退出
        }
        else {
            m.type = CHAT;
        }
        strcpy_s(send_buf, msgToString(m).c_str());
        strcpy_s(send_buf, msgToString(m).c_str());

        lock_guard<mutex> lock(clientSocketsMutex);
        for (const auto& client : clientSockets) {
            send(client.first, send_buf, sizeof(send_buf), 0);
        }

        cout << m.time << "【" << m.name << "】:" << m.msg << endl;
        memset(send_buf, 0, sizeof(send_buf));

        if (m.msg == "off" || m.msg == "OFF") {
            cout << "【系统消息】:您选择了OFF,连接离线!" << endl;
            break;
        }
        else if (m.msg == "EXIT" || m.msg == "exit") {
            cout << "【系统消息】:您选择了EXIT，连接关闭!" << endl;
            break;
        }
    }
```

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

实现:通过输入获取管理员的消息，将其封装成 `message` 结构体后，使用 `send()` 函数广播给所有在线客户端。

10. `reconnect(SOCKET& s_server, SOCKADDR_IN& server_addr)`

提供了**自动重连**的逻辑，确保客户端在网络波动或断开连接时能够自动恢复连接。

功能:客户端在与服务器断开连接后，尝试进行自动重连，确保聊天功能的连续性。

```
void reconnect(SOCKET& s_server, SOCKADDR_IN& server_addr) {
    while (true) {
        cout << "【系统消息】:尝试重新连接..." << endl;
        s_server = socket(AF_INET, SOCK_STREAM, 0);
        if (connect(s_server, (SOCKADDR*)&server_addr, sizeof(SOCKADDR)) ==
SOCKET_ERROR) {
            cout << "【系统消息】:重连失败，5秒后重试..." << endl;
            Sleep(5000);  // 每5秒重试一次
        } else {
            cout << "【系统消息】:重连成功！" << endl;// 发送userID给服务器，告知这是重连
            send(s_server, userID.c_str(), userID.size(), 0);  // 重新发送昵称
            offlineMode = false;  //  重新上线，取消离线状态，恢复在线状态
            break;
        }
    }
}
```

实现:通过循环反复尝试重新连接服务器，每次连接失败后等待一段时间，直到成功为止。重连成功后重新发送 `userID` 给服务器，以恢复在线状态。

11. `getCurrentTime()`

生成时间戳，确保每条消息都带有发送的时间信息，方便用户查看消息的发送顺序。

功能:获取当前系统时间并将其格式化为字符串，显示为消息的时间戳。

```
string getCurrentTime() {
    time_t now = time(0);
    tm local_time;
    localtime_s(&local_time, &now);
    char timeString[100];
    strftime(timeString, sizeof(timeString), "%Y-%m-%d %H:%M:%S", &local_time);
    return string(timeString);
}
```

实现:使用 `time()` 函数获取当前时间，使用 `strftime()` 将时间格式化为字符串。

通过以上关键函数，聊天程序实现了完整的客户端-服务器通信，包括:

- **初始化和连接**：`initwsa()`、`connect()` 和 `accept()` 完成了网络连接的初始化和建立。
- **消息处理**：`msgToString()` 和 `stringToMsg()` 实现了消息的序列化与反序列化，确保消息能够在客户端和服务器之间传递。
- **线程处理**：`serveraccept()` 和 `clientaccept()` 分别在服务器和客户端中处理消息的接收。
- **广播与通知**：`broadcastOnlineCount()` 广播在线人数，`serversend()` 广播服务器端消息。

这些函数确保了程序的完整性和并发性，支持多人聊天、系统通知、离线重连等功能。

4.3 server模块-main

1. **显示启动信息**:输出启动界面提示信息，标明服务器正在启动。

```
cout << "-----------------------------------------------------" << endl;
cout << "||                   Mini-We-Chat                  ||" << endl;
cout << "-----------------------------------------------------" << endl;
```

2. **初始化网络环境**:调用 `initwsa()` 函数初始化Winsock环境，确保Socket编程的库准备就绪。如果初始化失败，则直接退出程序。

```
if (!initwsa()) {
    return 0;}
```

3. **设置服务器地址和端口**:配置 `SOCKADDR_IN server_addr`，将其设置为使用 `AF_INET` 地址族、任意IP地址（`INADDR_ANY`）和端口号5010。

```
server_addr.sin_family = AF_INET;
server_addr.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
server_addr.sin_port = htons(5010);
```

4. **创建服务器Socket**:使用 `socket()` 创建一个流式Socket（TCP）。如果失败，程序会退出。

```
s_server = socket(AF_INET, SOCK_STREAM, 0);
```

5. **绑定Socket到服务器地址**:使用 `bind()` 函数将服务器的Socket与指定的IP地址和端口绑定。如果绑定失败，程序会退出并清理网络环境。

```
if (bind(s_server, (SOCKADDR*)&server_addr, sizeof(server_addr)) == SOCKET_ERROR) {
    cout << "||【系统消息】: 服务器绑定失败!!" << endl;
    WSACleanup();
    return 0;
}
```

6. **开始监听连接**:使用 `listen()` 函数使服务器开始监听来自客户端的连接请求。

```
if (listen(s_server, SOMAXCONN) < 0) {
    cout << "||【系统消息】: 设置监听失败!!" << endl;
    closesocket(s_server);
    WSACleanup();
    return 0;
}
```

7. **接受客户端连接**:服务器**进入一个循环**，等待客户端连接。每当有客户端连接时，使用 `accept()` 函数接受连接，获取客户端的Socket。接收到客户端的昵称后，将其注册到服务器中。

```
s_accept = accept(s_server, (SOCKADDR*)&accept_addr, &len);

char nickname[256];
int nickname_len = recv(s_accept, nickname, sizeof(nickname) - 1, 0);
if (nickname_len > 0) {
    nickname[nickname_len] = '\0';
    string userNickname = nickname;
    handleClientConnect(s_accept, userNickname);
}
```

8. **创建消息处理线程**:为每个客户端创建两个线程：一个用于处理消息的接收（`serveraccept`）。另一个用于处理服务器端消息的发送（管理员输入）（`serversend`）。

```
CloseHandle(CreateThread(NULL, 0, serversend, (LPVOID)&s_accept, 0, 0));
CloseHandle(CreateThread(NULL, 0, serveraccept, (LPVOID)&s_accept, 0, 0));
```

9. **清理资源**:程序终止时，通过 `WSACleanup()` 函数清理网络资源。

4.3 client模块-main

1. **显示启动信息，输入用户昵称和服务器IP地址**:提示用户输入昵称（`userID`）和服务器的IP地址（`ipaddress`），为后续与服务器通信做准备。
2. **初始化网络环境，设置服务器地址。**
3. **创建客户端Socket并连接服务器**:使用 `socket()` 创建客户端的Socket，随后使用 `connect()` 与服务器建立TCP连接。如果连接失败，程序尝试重连。

```
s_server = socket(AF_INET, SOCK_STREAM, 0);
if (connect(s_server, (SOCKADDR*)&server_addr, sizeof(SOCKADDR)) == SOCKET_ERROR) {
    cout << "【系统消息】:服务器连接失败!!" << endl;
    closesocket(s_server);
    reconnect(s_server, server_addr);  // 尝试重连
}
```

4. **发送用户昵称给服务器**:连接成功后，客户端会将用户输入的昵称 `userID` 发送给服务器，完成登录注册。

5. **创建消息处理线程**:创建接收消息线程 `clientaccept`，用于持续接收服务器发送的消息，并在主线程中持续处理用户输入消息并发送到服务器。

```
CloseHandle(CreateThread(NULL, 0, clientaccept, (LPVOID)&s_server, 0, 0));
```

6. **消息发送循环**:主线程中进入一个循环，用户输入消息后发送给服务器。如果用户输入 `off` 或 `exit`，则进入离线模式或退出程序。

```
while (flag) {
    char in[1000];
    cin.getline(in, 1000);
    m.msg = in;
    m.time = getCurrentTime();

    if (m.msg == "off" || m.msg == "OFF") {
        m.type = OFFLINE;
        offlineMode = true;// 设置客户端进入离线模式
        send(s_server, msgToString(m).c_str(), msgToString(m).size(), 0);
        cout << "【系统消息】: 您选择了OFF,连接离线!" << endl;
    } else if (m.msg == "exit" || m.msg == "EXIT") {
        m.type = EXIT;
        flag = 0;// 设置退出标志
        send(s_server, msgToString(m).c_str(), msgToString(m).size(), 0);
        cout << "【系统消息】: 您选择了EXIT，连接关闭! " << endl;
        break;
    } else {
        m.type = CHAT;
        send(s_server, msgToString(m).c_str(), msgToString(m).size(), 0);
    }
}
```

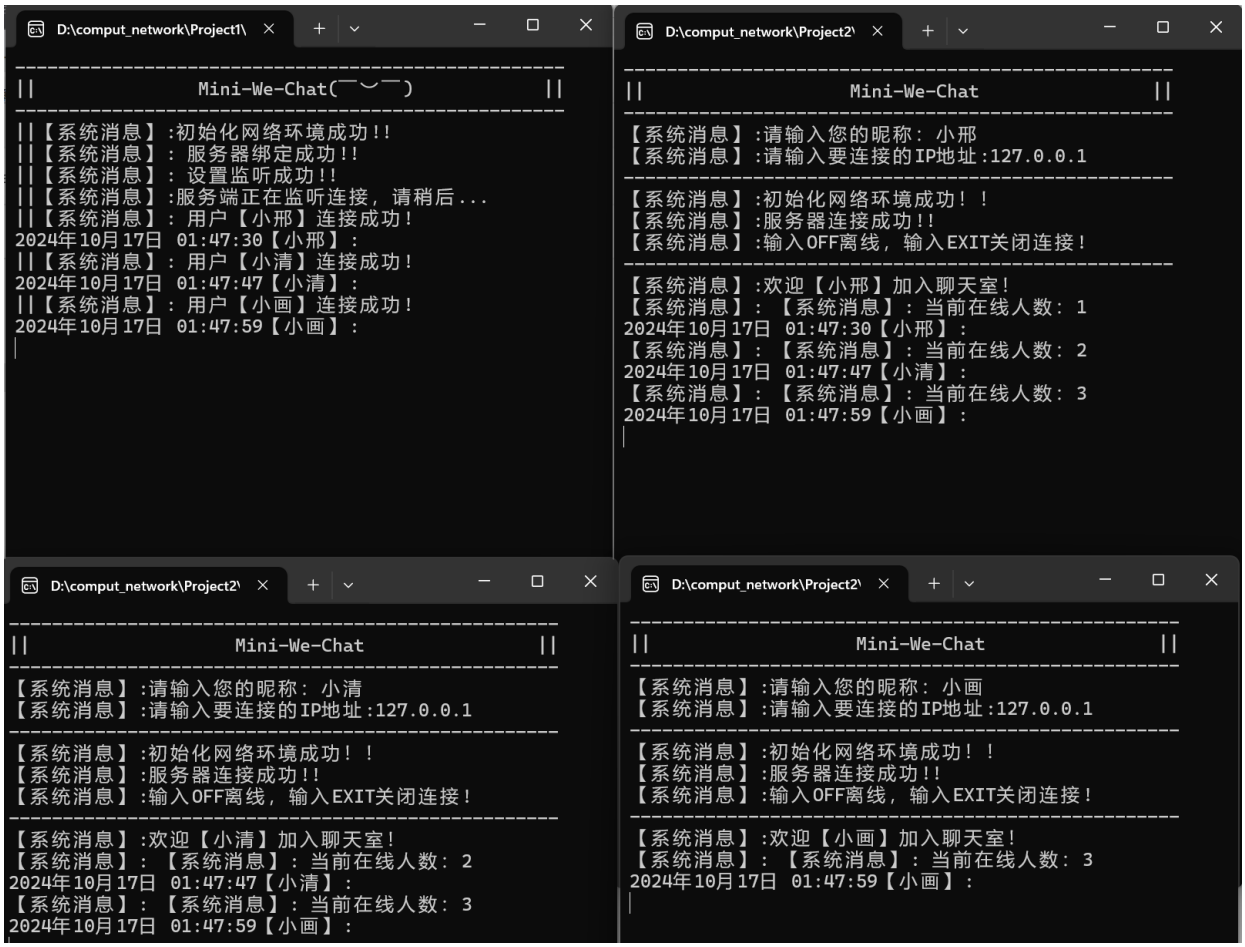
7. **清理资源**:在退出循环后，关闭客户端的Socket，并调用 `WSACleanup()` 清理网络资源。

```
closesocket(s_server);
WSACleanup();
return 0;
```

五、实验结果

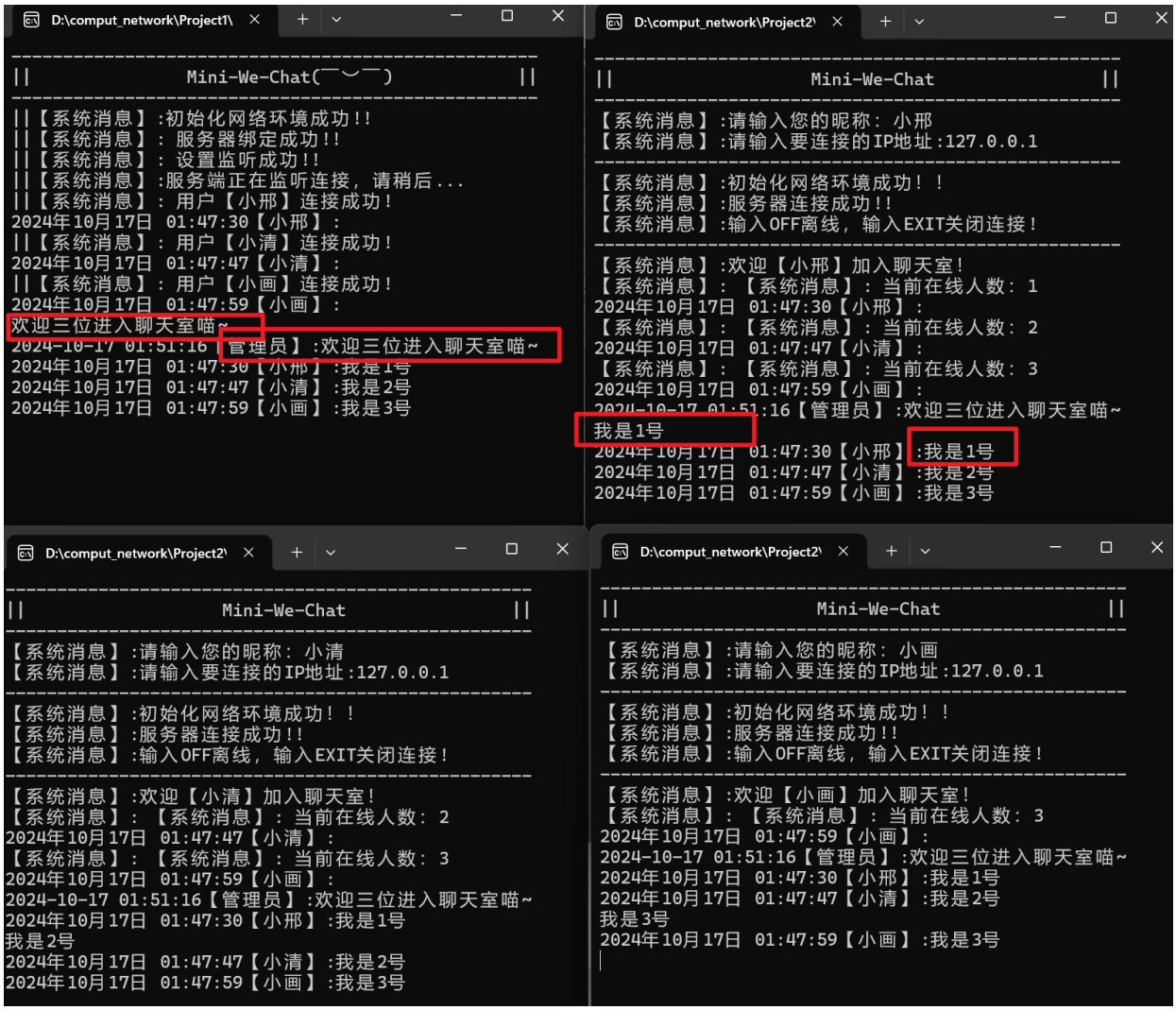
5.1 server-client 通信

首先将服务器端与客户端打开，服务器端等待连接，客户端需 要输入自己的昵称和连接的 IP 地址，连接成功后客户端 1，客户端 2，客户端3和服务器端都可互相通信：



注：左上为server端，右上为客户端1，左下2，右下3，随着用户加入不断更新人数，并发出系统通知。

5.2 多人聊天

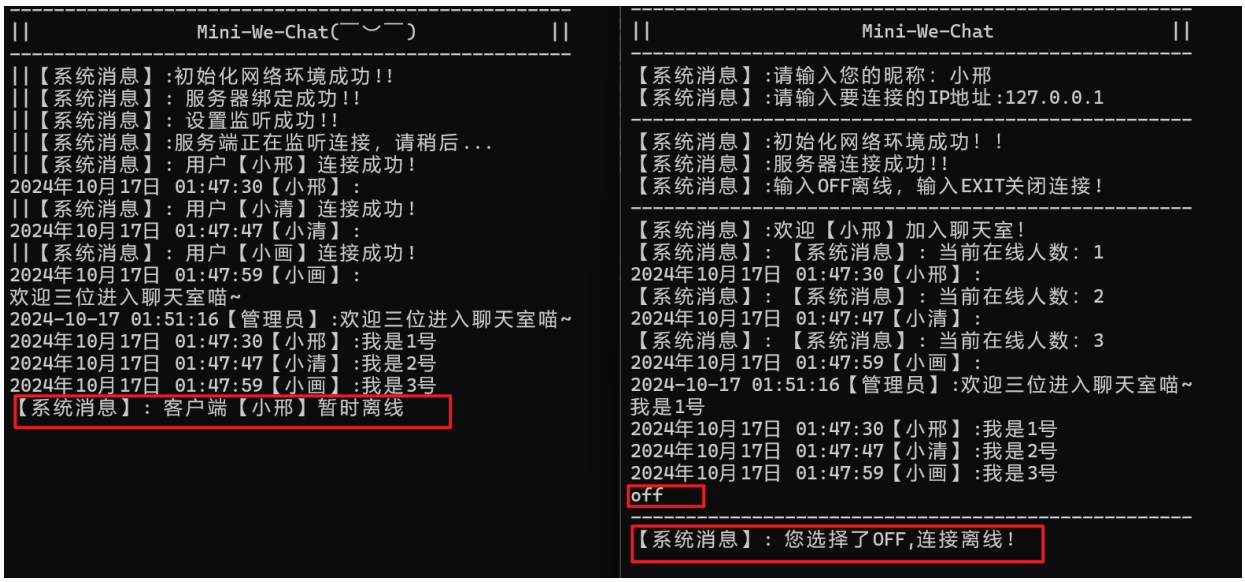


从左上至右下，依次发送消息，服务器端显示【管理员】，客户端由用户昵称显示。用户可以看到自己发送消息并且被成功转发的过程（如图）

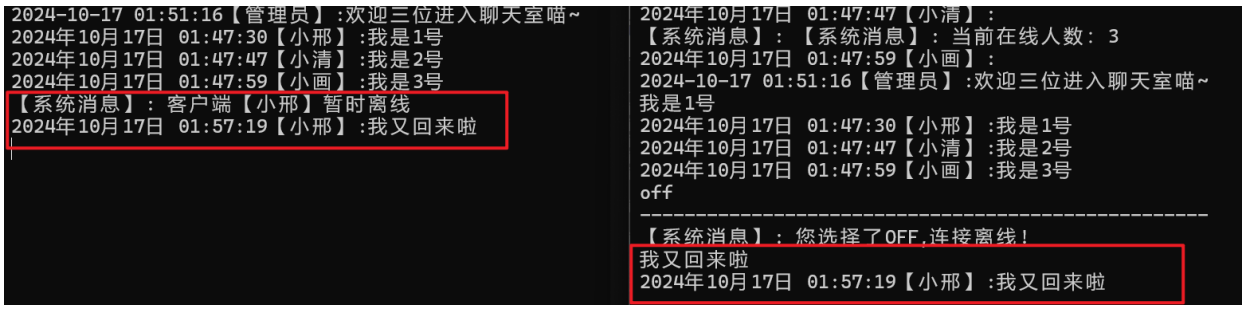
5.3 client端off/exit

由2号进行off操作，3号进行exit操作，观察server端和4号窗口：

5.3.1客户端off离线:

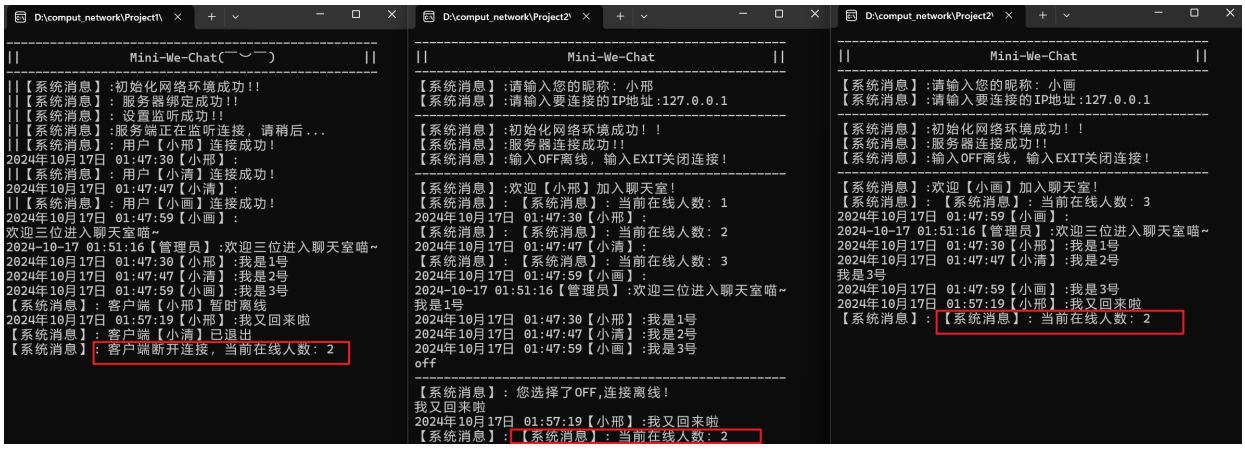


客户端**off离线**总人数不减少，可以通过重新发送消息上线



5.3.2 客户端exit退出

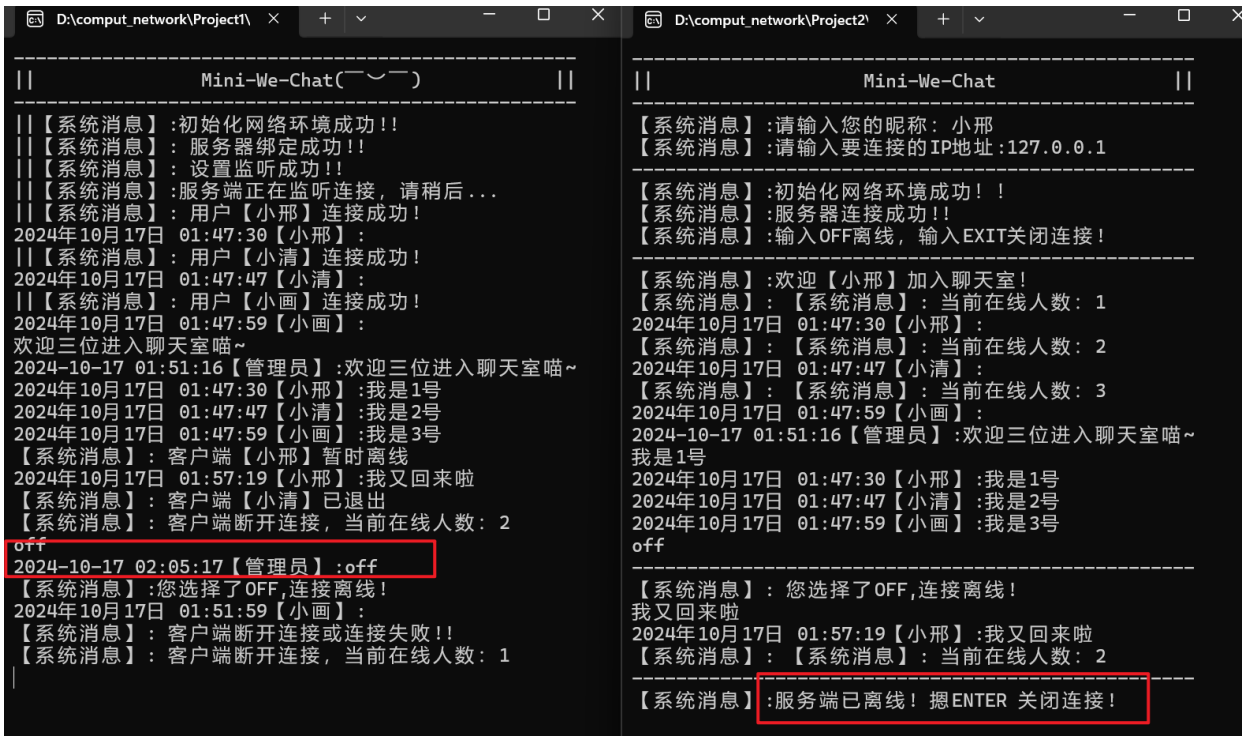
3号窗口按exit退出，人数播报-1，显示退出昵称。



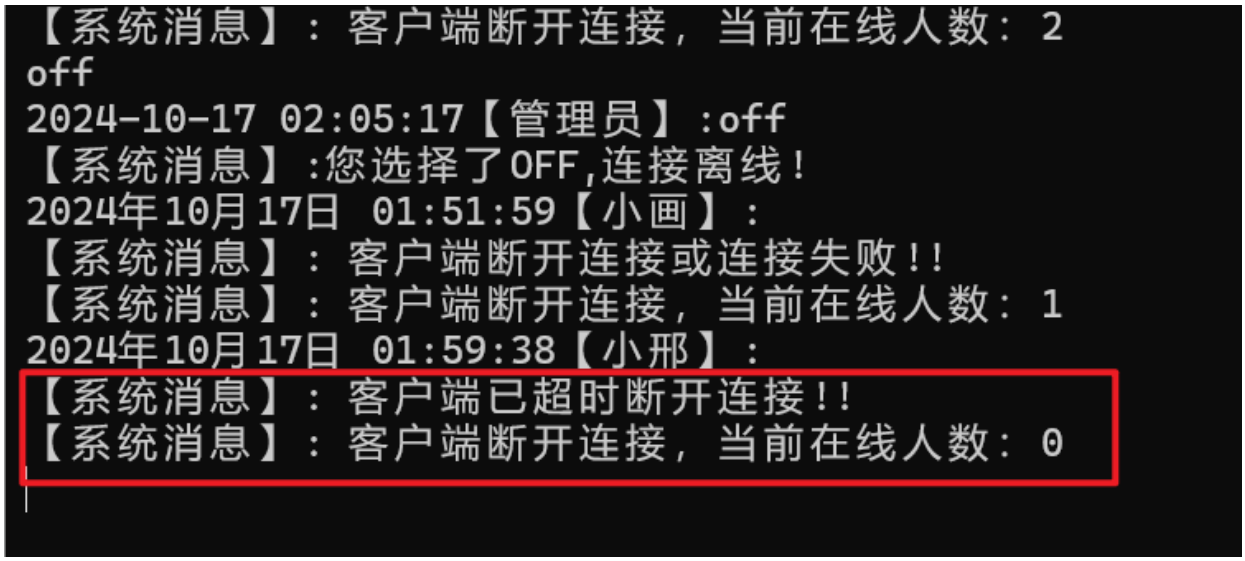
5.4 server端off/exit

5.4.1 命令行操作off/exit

客户端会相应提示“服务端已离线! 摁ENTER 关闭连接!”；“服务端已下线! 摁ENTER 关闭连接!”



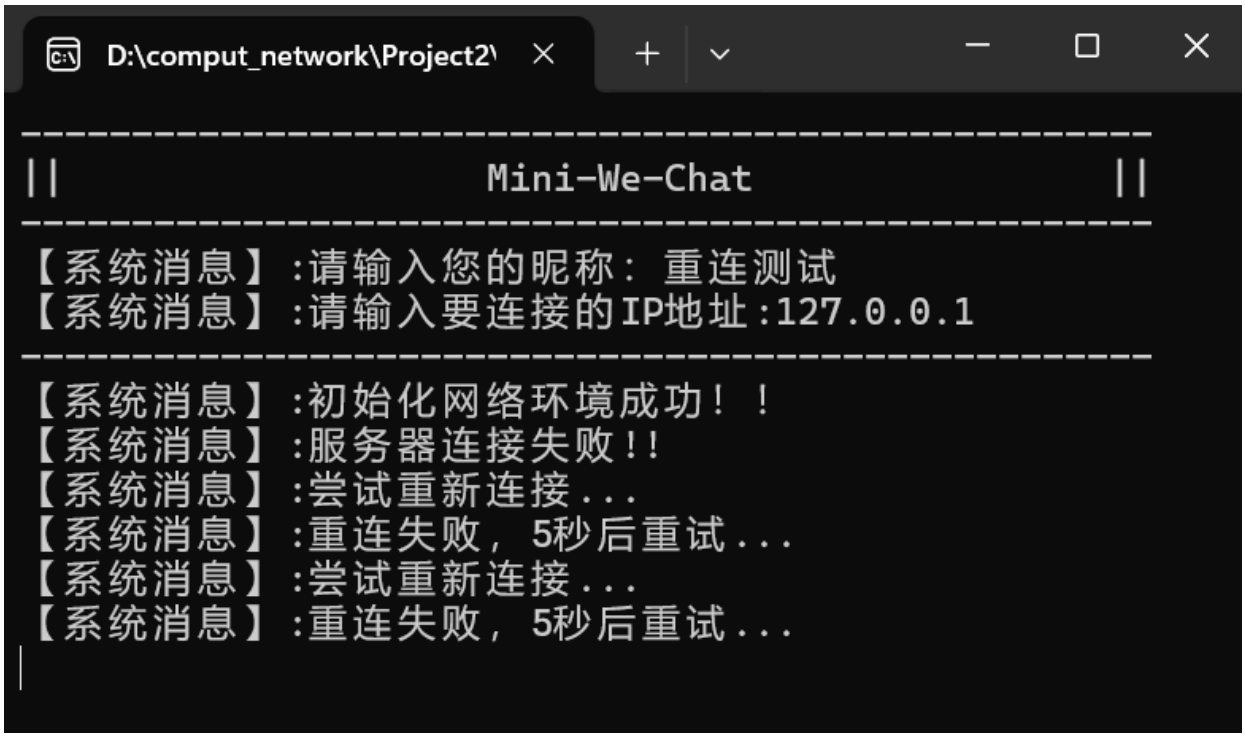
4号按下enter键，窗口关闭，更新现在客户端数量为1（2号窗口）



窗口2长时间未按照提示按下enter会自动断开连接，之后关闭窗口更新人数。

5.4.1 直接关闭server端，测试client端重连机制

首先再开启一个客户端测试（刚刚所有的客户端在测试中已经关闭），我们先不打开服务端，可以观察到客户端每5秒发送一次重连请求：



打开服务器，开始监听连接，即可重连成功

六、实验总结

在本次实验中，主要任务是编写一个基于 Socket 的多人聊天程序，过程中遇到了许多挑战，同时也收获了宝贵的经验和知识。

实验过程：

实验的目标是实现一个服务器与多个客户端通信的聊天程序，支持多人同时在线，能够发送和接收消息，并具备自动重连功能。服务器需要处理多个客户端的连接，广播消息给所有在线用户，客户端则负责发送消息给服务器，并接收其他用户的消息。

实验过程分为以下几个阶段：

1. **Socket编程的学习和应用**：首先学习了基本的 Socket 编程，了解如何在 C++ 中通过 `Winsock` 进行 TCP 连接，并处理客户端与服务器之间的通信。
2. **多人聊天功能的实现**：利用多线程在服务器端同时处理多个客户端的连接，每个客户端都有一个独立的线程处理消息发送和接收。
3. **自动重连功能的添加**：为了解决客户端断开连接后无法恢复的问题，尝试加入了自动重连的功能，让客户端在服务器重启后能够自动恢复连接。

遇到的困难：

我在实验过程中遇到了许多困难：

1. **自动重连问题**： 在实现自动重连功能时，最初的设计并未成功。客户端无法在连接断开后自动重连，原因是 `socket` 在断开后未能正确更新，导致重连操作没有成功。通过多次调试，最终确定了需要通过引用传递 `socket`，并确保每次重连时创建一个新的 `socket` 实例，这才解决了问题。
2. **多线程的管理**： 在服务器端，每个客户端都由一个独立的线程来处理。当服务器断开连接或某个客户端异常退出时，如何安全地结束线程、释放资源成为一个难题。最初的实现没有正确地等待线程结束，导致程序在退出时出现异常。后来通过使用 `WaitForSingleObject` 等线程管理函数解决了这个问题。
3. **消息同步和互斥**： 在多线程环境下，需要确保多个客户端发送消息时不会互相干扰。为了解决这个问题，使用了 `mutex` 来同步对共享资源（如在线用户列表）的访问。但在早期版本中，由于锁机制设计不当，程序有时会出现死锁或性能问题。通过合理地设置锁的粒度和使用范围，确保了程序的稳定性。

4. **调试跨平台问题**： 由于实验环境主要是在 Windows 上进行，但考虑到跨平台的兼容性，部分代码需要作适配。在 `Winsock` 和 POSIX `sockets` 之间的一些差异导致了不少问题，尤其是在 socket 的关闭处理上。经过学习和查阅资料，理解了不同平台上的 socket 行为差异，确保程序的可移植性。

学到的知识：

1. **深入理解 Socket 编程**：通过实验，深入学习了如何在 C++ 中使用 `Winsock` 进行网络编程，包括创建连接、发送和接收消息、处理断开连接等操作。
2. **多线程编程与同步**：在服务器端的实现中，使用多线程并发处理多个客户端的连接，并通过 `mutex` 和线程同步技术，确保程序在多线程环境下稳定运行。
3. **自动重连机制的实现**：在网络通信中，客户端与服务器可能随时断开，自动重连机制的设计让我深入理解了如何在断线重连的场景下保证 socket 的正常使用和资源管理。
4. **错误处理与调试能力**：实验中遇到了多次异常情况，通过反复调试和查找资料，提升了处理网络异常、线程管理等复杂问题的能力。

实验心得与思考：

这次实验不仅让我对网络编程有了更深入的理解，还让我体会到了在实际开发中，问题往往比预期复杂得多。在实验中，我学会了如何系统性地分析和解决问题，从最初的简单实现，到逐步加入复杂的功能（如自动重连），再到优化和调试，这个过程让我认识到编程不仅仅是写代码，更是如何设计出健壮的、可维护的系统。

实验还让我意识到，多线程编程和网络编程涉及到的并发和同步问题非常复杂，稍有不慎就会导致程序的不稳定性或性能下降。为了编写出高效、稳定的代码，需要对程序的逻辑和资源管理有清晰的理解，这也是我在实验中学到的最重要的经验之一。

总的来说，这次实验让我不仅掌握了 Socket 编程的知识，还锻炼了调试能力、问题分析能力和系统设计能力。