实验 1：利用流式套接字编写聊天程序
计算机网络第一次实验报告

姓名：郭佳成　学号：2311990　专业：密码科学与技术

代码：https://github.com/Fighting05/ComputerNetwork

2025 年 10 月 24 日

1 实验要求

1. 设计聊天协议，并给出聊天协议的完整说明。
2. 利用 C 或 C++ 语言，使用基本的 Socket 函数进行程序编写，不允许使用 CSocket 等封装后的类。
3. 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。
4. 完成的程序应能支持英文和中文聊天。
5. 采用多线程，支持多人聊天。
6. 编写的程序应结构清晰，具有较好的可读性。
7. 在实验中观察是否有数据包的丢失，提交程序源码、可执行代码和实验报告。

2 实验环境

- **操作系统**：Windows 11
- **开发语言**：C++11
- **编译器**：MinGW-w64 GCC
- **网络库**：Winsock2
- **GUI 框架**：ImGui + GLFW + OpenGL3（图形界面版本）
- **服务器地址**：60.205.14.222:2059
- **本地测试**：127.0.0.1:1023

3 实验原理

3.1 TCP 流式套接字基础

TCP（Transmission Control Protocol）是一种面向连接、可靠的传输层协议。流式套接字（Stream Socket）基于 TCP 协议，提供以下特性：

3.2 Socket 编程模型

3.2.1 服务器端流程

服务器端的基本 Socket 编程流程包括以下几个步骤：首先调用 **socket()** 函数创建套接字，然后使用 **bind()** 函数绑定 IP 地址和端口号，接着调用 **listen()** 函数监听客户端连接请求，当有客户端连接时通过 **accept()** 函数接受客户端连接，之后就可以使用 **recv()/send()** 函数进行数据的接收和发送，最后在完成通信后调用 **closesocket()** 函数关闭套接字。

3.2.2 客户端流程

客户端的基本 Socket 编程流程包括以下几个步骤：首先调用 **socket()** 函数创建套接字，然后使用 **connect()** 函数连接服务器，接着就可以通过 **send()/recv()** 函数进行数据的发送和接收，最后在完成通信后调用 **closesocket()** 函数关闭套接字。

3.3 多线程并发处理

为了支持多人同时在线聊天，服务器需要为每个客户端创建独立的线程来处理消息收发。主要涉及：

- **线程创建：** 使用 `std::thread` 为每个客户端创建处理线程
- **线程同步：** 使用 `std::mutex` 保护共享资源（客户端列表）
- **线程分离：** 使用 `detach()` 让线程在后台独立运行

4 聊天协议设计

协议概述

本聊天系统采用基于文本的应用层协议，使用 TCP 作为传输层协议。协议设计简洁高效，易于实现和扩展。

连接阶段

客户端连接成功后的第一条消息为昵称注册：

```
格式：<nickname>\n
示例：张三\n
说明：昵称不能包含换行符，建议长度不超过20个字符
```

普通消息（广播）

客户端发送的普通消息会广播给所有在线用户：

```
客户端发送格式：<message>\n
示例：大家好！\n

服务器转发格式：[<nickname>] <message>
示例：[张三] 大家好！
```

私聊消息

使用特殊命令格式发送私聊消息：

```
客户端发送格式：/msg <target_nickname> <message>\n
示例：/msg 李四 你好\n

接收方收到格式：[私聊] <sender_nickname>: <message>
示例：[私聊] 张三：你好

发送方回显格式：<sender_nickname>:[私聊] TO<target_nickname>:<message
    >
示例：张三:[私聊] TO李四:你好
```

系统消息

系统自动生成的通知消息：

```
用户加入：[<nickname>] 加入了聊天室\n
用户离开：[<nickname>] 离开了聊天室\n
在线列表：当前在线用户：<user1>,<user2>,...\n
错误消息：用户 [<nickname>] 不在线\n
```

协议状态机

客户端状态转换：

1. 未连接 → 发送昵称 → 已连接
2. 已连接 → 发送消息/接收消息 → 已连接
3. 已连接 → 断开连接 → 未连接

服务器状态转换（针对每个客户端）：

1. 等待连接 → 接受连接 → 接收昵称
2. 接收昵称 → 注册成功 → 通信状态
3. 通信状态 → 接收/转发消息 → 通信状态
4. 通信状态 → 检测断开 → 清理资源

图 1: 协议状态机

5 程序设计与实现

5.1 系统架构

本聊天系统采用经典的客户端/服务器（C/S）架构，包含以下三个主要组件：

1. **服务器程序（server.cpp）**：负责管理客户端连接、消息转发和用户管理
2. **控制台客户端（client.cpp）**：提供命令行界面的聊天客户端
3. **图形界面客户端（client_gui.cpp）**：提供基于 ImGui 的图形用户界面

5.2 服务器端实现

5.2.1 核心数据结构

服务器使用以下数据结构管理客户端信息：

```
// 存储所有客户端的Socket
vector<SOCKET> clients;

// Socket到昵称的映射
```

```
unordered_map<SOCKET, string> clientNames;

// 昵称到Socket的映射（用于私聊）
unordered_map<string, SOCKET> nameToSocket;

// 互斥锁，保护共享数据结构
mutex clientsMutex;
```

5.2.2 主函数流程

服务器主函数的核心逻辑如下：

```
int main() {
    // 1. 初始化Winsock
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    // 2. 创建Socket
    SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);

    // 3. 绑定地址和端口
    sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(1023);
    addr.sin_addr.s_addr = INADDR_ANY;
    bind(serverSocket, (sockaddr*)&addr, sizeof(addr));

    // 4. 开始监听
    listen(serverSocket, 5);

    // 5. 循环接受客户端连接
    while (true) {
        SOCKET clientSocket = accept(serverSocket, nullptr, nullptr);

        // 接收并注册客户端昵称
        char buf[1024];
        int n = recv(clientSocket, buf, sizeof(buf)-1, 0);
        buf[n] = '\0';
        string nickname(buf);
        if (!nickname.empty() && nickname.back() == '\n') {
```

```
            nickname.pop_back();
        }

        // 保存客户端信息
        {
            lock_guard<mutex> lock(clientsMutex);
            clientNames[clientSocket] = nickname;
            nameToSocket[nickname] = clientSocket;
            clients.push_back(clientSocket);

            // 广播加入消息
            string welcome = "[" + nickname + "]␣加入了聊天室\n";
            for (SOCKET s : clients) {
                send(s, welcome.c_str(), welcome.size(), 0);
            }
        }

        // 为该客户端创建处理线程
        thread(handleClient, clientSocket).detach();
    }

    return 0;
}
```

5.2.3 客户端处理函数

每个客户端由独立线程处理，实现消息接收和转发：

```
void handleClient(SOCKET clientSocket) {
    char buf[1024];
    while (true) {
        int n = recv(clientSocket, buf, sizeof(buf)-1, 0);
        if (n <= 0) {
            // 客户端断开连接
            lock_guard<mutex> lock(clientsMutex);
            string name = clientNames[clientSocket];
            clientNames.erase(clientSocket);
            nameToSocket.erase(name);
            clients.erase(remove(clients.begin(), clients.end(),
                          clientSocket), clients.end());
```

```
13
14              // 广播离开消息
15              string goodbye = "[" + name + "]␣离开了聊天室\n";
16              for (SOCKET s : clients) {
17                  send(s, goodbye.c_str(), goodbye.size(), 0);
18              }
19              break;
20          }
21
22          buf[n] = '\0';
23          string msg(buf);
24
25          // 判断是否为私聊消息
26          if (msg.length() >= 5 && msg.substr(0, 5) == "/msg␣") {
27              // 处理私聊逻辑
28              string nickname = clientNames[clientSocket];
29              size_t firstSpace = msg.find('␣', 5);
30
31              if (firstSpace != string::npos) {
32                  string target = msg.substr(5, firstSpace - 5);
33                  string content = msg.substr(firstSpace + 1);
34                  string privateMsg = "[私聊]␣" + nickname + ":␣" +
                        content;
35
36                  lock_guard<mutex> lock(clientsMutex);
37                  if (nameToSocket.count(target)) {
38                      send(nameToSocket[target], privateMsg.c_str(),
39                           privateMsg.size(), 0);
40                  } else {
41                      string err = "用户␣[" + target + "]␣不在线\n";
42                      send(clientSocket, err.c_str(), err.size(), 0);
43                  }
44              }
45          } else {
46              // 广播消息
47              string nickname = clientNames[clientSocket];
48              string broadcastMsg = "[" + nickname + "]␣" + msg;
49
50              lock_guard<mutex> lock(clientsMutex);
51              for (SOCKET s : clients) {
```

```
                send(s, broadcastMsg.c_str(), broadcastMsg.size(), 0)
                    ;
            }
        }
    }
}
```

5.3 控制台客户端实现

5.3.1 消息接收线程

客户端使用独立线程接收服务器消息：

```
void rcvFromServer(SOCKET clientSocket) {
    while(true) {
        string rcvBuf(1024, '\0');
        int bytesReceived = recv(clientSocket, &rcvBuf[0],
                                 rcvBuf.size(), 0);
        if (bytesReceived <= 0) {
            cout << "接收失败或连接关闭" << endl;
            return;
        }
        rcvBuf.resize(bytesReceived);
        cout << rcvBuf << endl;
    }
}
```

5.3.2 消息发送线程

另一个独立线程负责发送用户输入的消息：

```
void sendServer(SOCKET clientSocket, const string& nickname) {
    while(true) {
        string buffer;
        cout << "请输入你要发送的消息（输入␣/quit␣退出）:␣" << endl;
        getline(cin, buffer);

        if (buffer == "/quit") {
            cout << "你已经退出聊天" << endl;
            closesocket(clientSocket);
            return;
```

```
        }

        buffer += "\n";
        send(clientSocket, buffer.c_str(), buffer.size(), 0);
    }
}
```

5.4 图形界面客户端实现

图形界面客户端基于 ImGui 框架实现，提供了更友好的用户体验。

5.4.1 全局数据结构

```
// 聊天记录
vector<string> g_chatHistory;

// 保护聊天记录的互斥锁
mutex g_chatMutex;

// 客户端Socket
SOCKET g_clientSocket = INVALID_SOCKET;

// 连接状态
bool g_connected = false;

// 用户昵称
string g_nickname;
```

5.4.2 连接对话框

程序启动时显示连接对话框，收集服务器信息和昵称：

```
// 界面状态变量
char nicknameInput[128] = "";
char serverIP[64] = "60.205.14.222";
int serverPort = 2059;
bool showConnectDialog = true;

// 在主循环中渲染连接对话框
if (showConnectDialog && !g_connected) {
```

```
    // 创建全屏背景
    ImGui::SetNextWindowPos(ImVec2(0, 0), ImGuiCond_Always);
    ImGui::SetNextWindowSize(ImVec2(window_width, window_height),
                             ImGuiCond_Always);

    ImGui::Begin("##ConnectionBackground", nullptr,
                 ImGuiWindowFlags_NoTitleBar |
                 ImGuiWindowFlags_NoResize);

    // 居中对话框
    ImVec2 dialogSize(1056, 768);
    ImVec2 dialogPos((window_width - dialogSize.x) * 0.5f,
                     (window_height - dialogSize.y) * 0.5f);
    ImGui::SetCursorPos(dialogPos);

    ImGui::BeginChild("ConnectionDialog", dialogSize, true);

    // 标题
    ImGui::Text("连接到服务器");

    // 输入框
    ImGui::InputText("##ServerIP", serverIP, sizeof(serverIP));
    ImGui::InputInt("##Port", &serverPort);
    ImGui::InputText("##Nickname", nicknameInput,
                     sizeof(nicknameInput));

    // 连接按钮
    if (ImGui::Button("连接", ImVec2(180, 45))) {
        if (strlen(nicknameInput) > 0) {
            if (connectToServer(serverIP, serverPort,
                                string(nicknameInput))) {
                showConnectDialog = false;
            }
        }
    }

    ImGui::EndChild();
    ImGui::End();
}
```

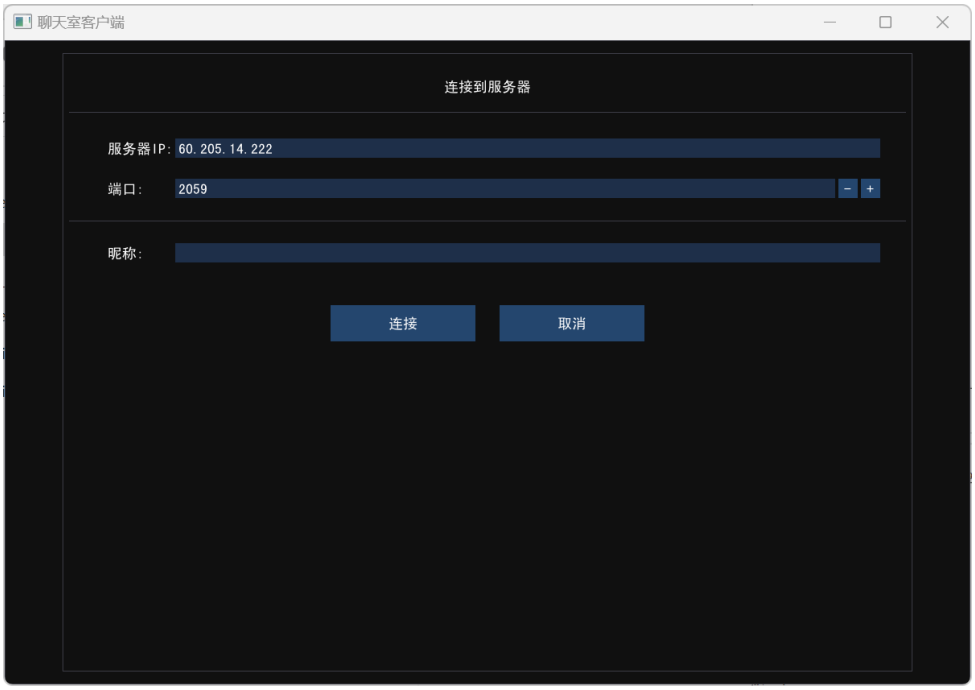

图 2: 连接对话框界面

5.4.3 主聊天窗口

连接成功后显示主聊天窗口：

```
if (g_connected || !showConnectDialog) {
    ImGui::SetNextWindowPos(ImVec2(0, 0), ImGuiCond_Always);
    ImGui::SetNextWindowSize(ImVec2(window_width, window_height),
                             ImGuiCond_Always);

    ImGui::Begin("聊天室", nullptr,
                 ImGuiWindowFlags_NoResize |
                 ImGuiWindowFlags_MenuBar);

    // 菜单栏
    if (ImGui::BeginMenuBar()) {
        if (ImGui::BeginMenu("连接")) {
            if (ImGui::MenuItem("重新连接", nullptr, false,
                                !g_connected)) {
                showConnectDialog = true;
            }
            if (ImGui::MenuItem("断开连接", nullptr, false,
                                g_connected)) {
                closesocket(g_clientSocket);
```

```
                g_connected = false;
            }
            ImGui::EndMenu();
        }
        ImGui::EndMenuBar();
    }

    // 状态栏
    if (g_connected) {
        ImGui::TextColored(ImVec4(0.0f, 1.0f, 0.0f, 1.0f),
                           "␣已连接");
    } else {
        ImGui::TextColored(ImVec4(1.0f, 0.0f, 0.0f, 1.0f),
                           "␣未连接");
    }

    // 聊天记录区域
    ImGui::BeginChild("ChatHistory", ImVec2(0, chatHeight), true);
    {
        lock_guard<mutex> lock(g_chatMutex);
        for (const auto& msg : g_chatHistory) {
            ImGui::TextWrapped("%s", msg.c_str());
        }
    }
    if (autoScroll) {
        ImGui::SetScrollHereY(1.0f);
    }
    ImGui::EndChild();

    // 输入区域
    ImGui::Text("输入消息:");
    bool enterPressed = ImGui::InputText("##MessageInput",
                        messageInput, sizeof(messageInput),
                        ImGuiInputTextFlags_EnterReturnsTrue);

    ImGui::SameLine();
    bool sendClicked = ImGui::Button("发送", ImVec2(100, 38));

    // 发送消息
    if ((enterPressed || sendClicked) && strlen(messageInput) > 0) {
```

```
60          if (g_connected) {
61              string msg(messageInput);
62              if (sendToServer(msg)) {
63                  messageInput[0] = '\0';
64              }
65          }
66      }
67
68      ImGui::End();
69  }
```

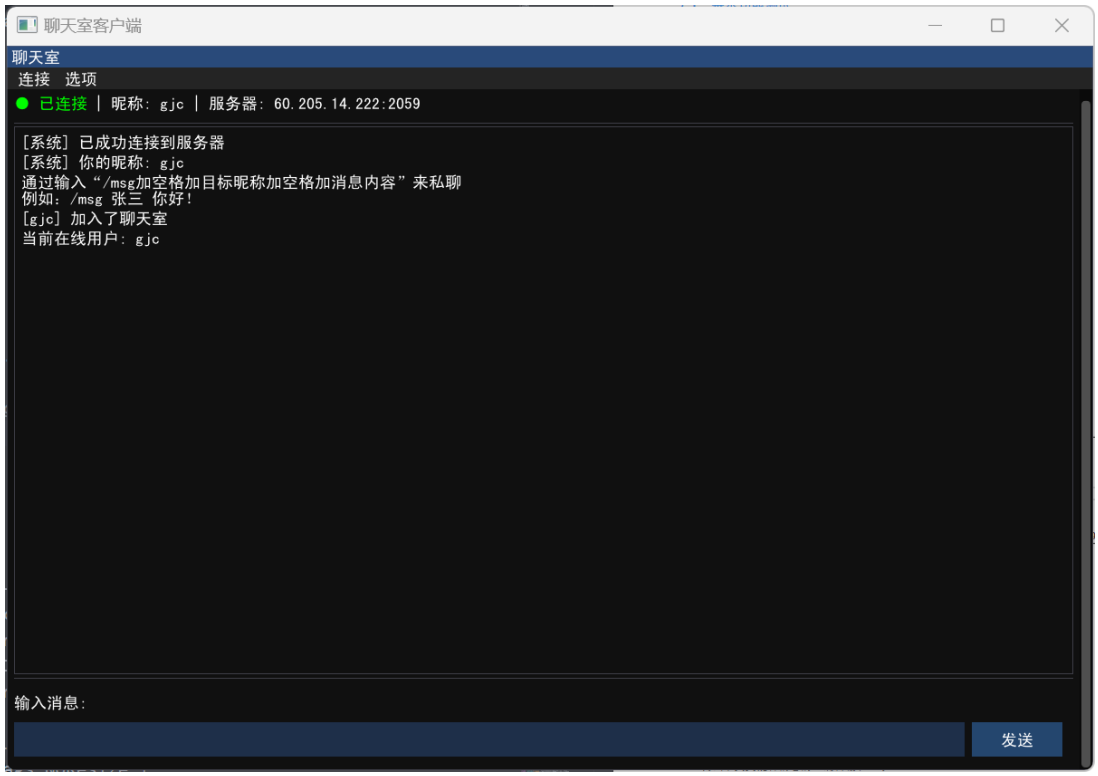

图 3: 主聊天窗口界面

6　编译与运行

6.1　运行步骤

1. **启动服务器：**在命令行运行 `server.exe`，服务器将在端口 1023 开始监听
2. **启动客户端：**运行 `client.exe` 或 `client_gui.exe`
3. **连接服务器：**输入服务器 IP、端口和昵称，点击连接
4. **开始聊天：**连接成功后即可发送消息

5. **私聊功能**：使用/msg 昵称消息内容格式发送私聊
6. **退出程序**：控制台版输入/quit，GUI 版点击菜单中的退出

7 功能测试

7.1 基本功能测试

7.1.1 连接功能

测试客户端能否成功连接到服务器：

- **预期结果**：客户端显示” 连接服务器成功”，服务器显示” 新客户端连接成功”
- **测试结果**：通过

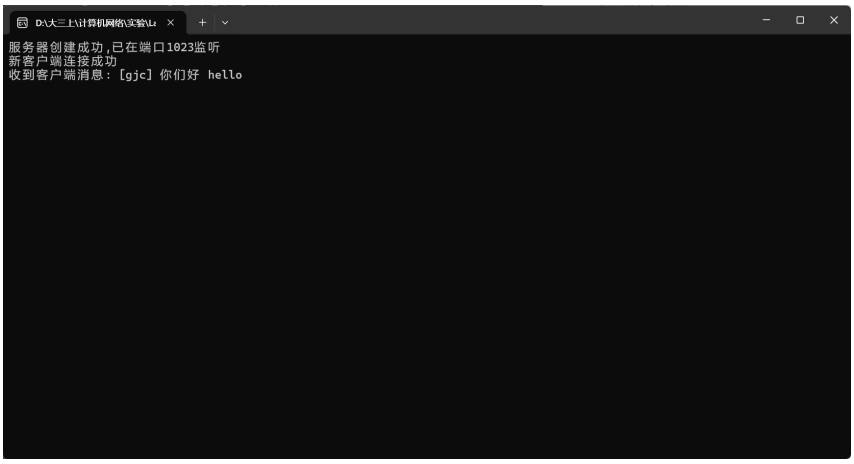

图 4: 连接功能测试

7.1.2 昵称注册

测试昵称是否正确注册和显示：

- **预期结果**：服务器和所有客户端显示”[昵称] 加入了聊天室”
- **测试结果**：通过

7.1.3 消息广播

测试普通消息的广播功能：

- **预期结果**：所有客户端都能收到格式为”[昵称] 消息内容” 的消息
- **测试结果**：通过

7.1.4 私聊功能

测试点对点私聊功能：

- **测试目的：**验证私聊消息的正确传递
- **测试方法：**使用/msg 目标昵称消息格式发送私聊
- **预期结果：**只有目标用户能收到私聊消息，格式为”[私聊] 发送者: 消息”
- **测试结果：** 通过

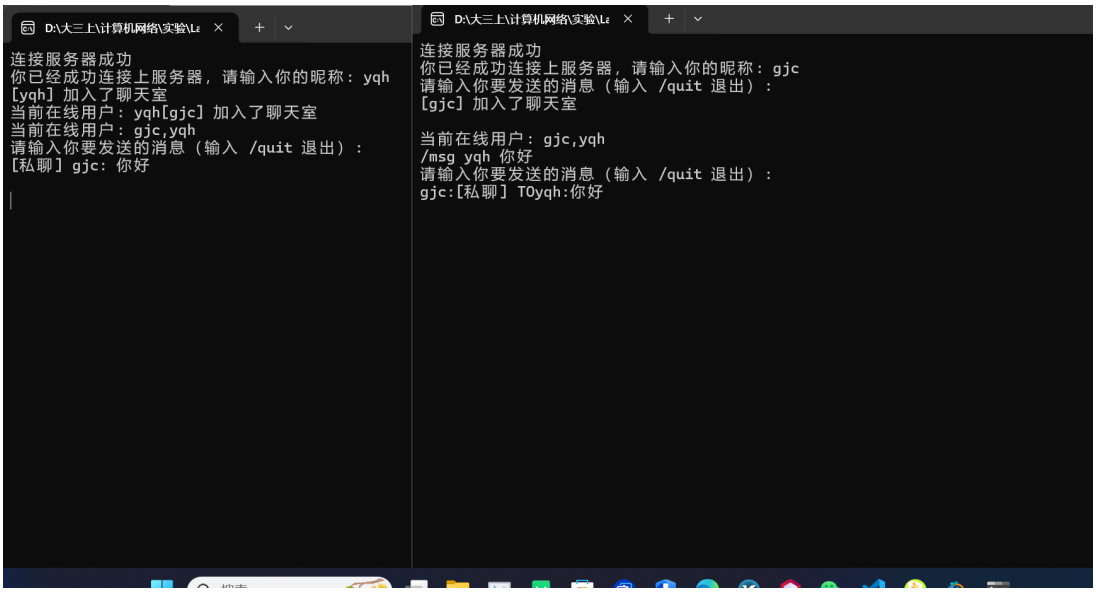

图 5: 私聊功能测试

7.2 异常情况测试

7.2.1 网络断开测试

测试客户端异常断开时的处理：

- **测试方法：**强制关闭客户端进程或断开网络连接
- **预期结果：**
  1. 服务器检测到连接断开
  2. 服务器清理该客户端的资源
  3. 向其他客户端广播”[昵称] 离开了聊天室”
  4. 不影响其他客户端的正常通信
- **测试结果：** 通过

通过在 `recv()` 函数检测返回值为 0 或负数来判断连接断开，并及时清理资源。

7.2.2 无效私聊目标测试

测试向不存在的用户发送私聊消息：

- **测试方法：**使用`/msg 不存在的昵称 消息`发送私聊
- **预期结果：**发送者收到”用户 [昵称] 不在线”的提示
- **测试结果：** 通过

7.2.3 重复昵称测试

由于当前实现未限制重复昵称，多个用户可以使用相同的昵称。这是系统的一个可改进点：

- **当前行为：**允许重复昵称，使用后连接的用户覆盖之前的映射
- **建议改进：**在用户注册时检查昵称是否已存在，拒绝重复昵称

8 数据包丢失分析

本实验要求观察程序运行过程中是否存在数据包丢失。由于聊天程序基于 TCP 流式套接字实现，而 TCP 协议本身具有可靠性保障机制（如确认应答、超时重传、顺序控制等），在正常网络环境下理论上不会发生底层数据包丢失。因此，测试的重点在于验证应用程序是否能正确发送和接收完整消息，避免因编码错误、缓冲区处理不当或粘包问题导致应用层“逻辑丢包”。

为直观验证消息传输的完整性，我们使用 Wireshark 在本地回环接口上对通信过程进行抓包分析。测试时，服务器运行于本地 1023 端口，客户端连接至 `127.0.0.1:1023`。在 Wireshark 中选择 Npcap Loopback Adapter 接口，并设置过滤条件 `tcp.port == 1023`，以仅捕获聊天程序的通信流量。

客户端发送英文消息 `hello` 后，Wireshark 成功捕获到对应的 TCP 数据包，如图 6 所示。该数据包源端口为客户端临时端口（如 3032），目的端口为 1023，TCP 标志位为 [PSH, ACK]，表明其携带有效应用层数据。在数据负载部分，十六进制内容显示为 `68 65 6c 6c 6f 0a`，恰好对应 ASCII 字符串 `hello`，总长度为 6 字节，与程序发送内容完全一致。同时，服务器与其他客户端均能正常显示该消息，未出现截断、乱码或缺失现象。

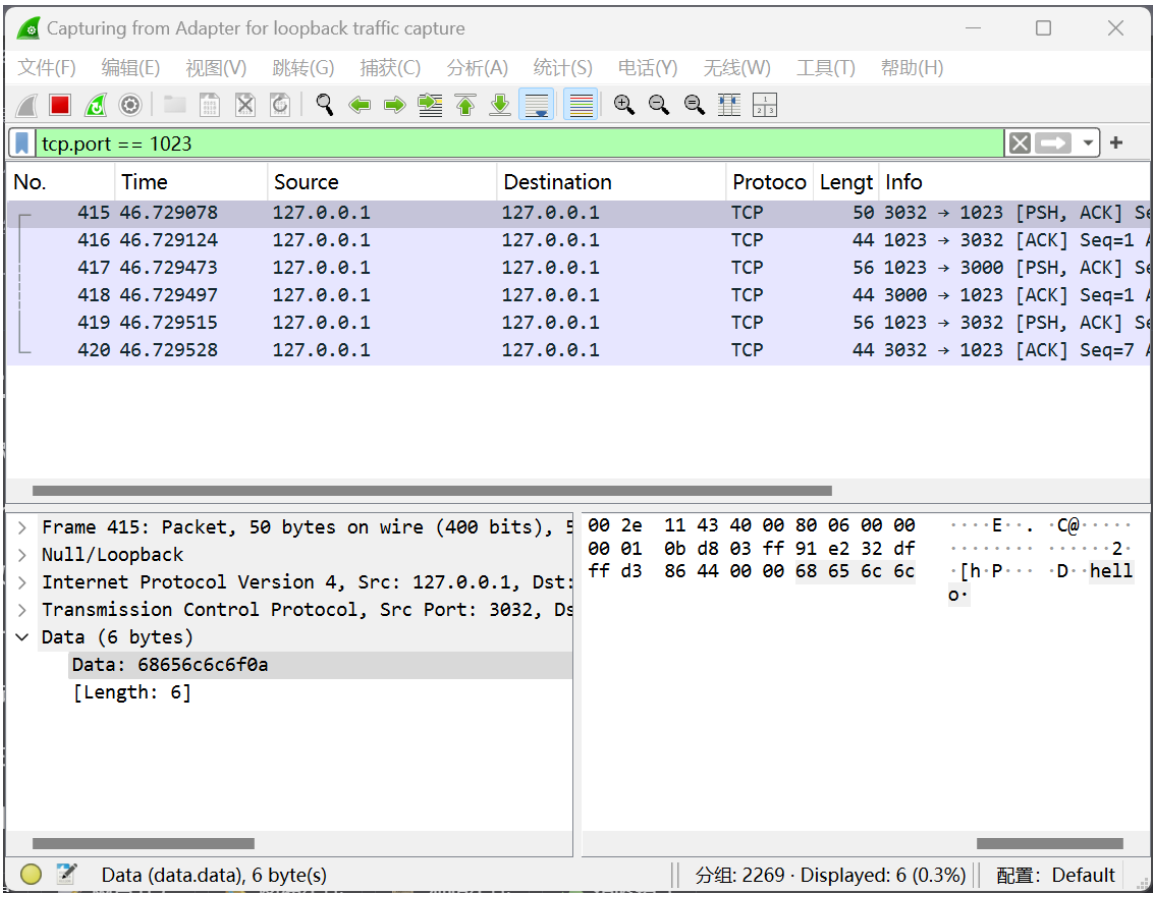

图 6: Wireshark 抓包结果：客户端发送”hello” 消息

上述结果表明，程序在发送消息时调用 `send()` 成功，TCP 协议栈完整传输了数据，接收端也能正确解析并输出。在整个测试过程中，未观察到任何数据包丢失或内容损坏的情况。尽管当前接收逻辑未显式处理粘包（依赖单次 `recv` 读取完整消息），但在低频、短消息场景下仍能正常工作。若未来扩展至高并发或长消息场景，建议引入基于换行符的消息帧解析机制以进一步提升鲁棒性。

综上，通过 Wireshark 抓包验证，本聊天程序在常规使用条件下能够可靠传输消息，满足实验对“观察数据包丢失”的要求。

9 总结

这次实验让我第一次完整地用原始 Socket API 实现了一个多人聊天程序。从服务器监听、客户端连接，到多线程处理消息、支持中英文，每一步都遇到了问题，也都在调试中解决了。虽然代码还有改进空间（比如粘包处理、昵称去重），但基本功能都跑通了，也能稳定聊天。整个过程加深了我对 TCP 流式套接字和多线程并发的理解，也让我意识到，写网络程序不能只依赖协议的“可靠性”，应用层的细节同样关键。这次实验收获很大，也为后续更复杂的网络编程打下了基础。

编译和运行说明

环境要求

- Windows 7 或更高版本
- MinGW-w64 GCC 编译器（支持 C++11）
- 对于 GUI 版本，需要 GLFW 和 OpenGL 支持

编译步骤

1. 编译服务器：

```
g++ -std=c++11 server.cpp -o server.exe -lws2_32 -static -static-
    libgcc -static-libstdc++
```

2. 编译控制台客户端：

```
g++ -std=c++11 client.cpp -o client.exe -lws2_32 -static -static-
    libgcc -static-libstdc++
```

3. 编译 GUI 客户端：

```
g++ -std=c++11 client_gui.cpp imgui/*.cpp imgui/backends/
    imgui_impl_glfw.cpp imgui/backends/imgui_impl_opengl3.cpp -o
    client_gui.exe -I./imgui -I./imgui/backends -lglfw3 -lopengl32 -
    lgdi32 -lws2_32 -static -static-libgcc -static-libstdc++
```

运行步骤

1. 首先运行 `server.exe` 启动服务器
2. 运行一个或多个 `client.exe` 或 `client_gui.exe` 启动客户端
3. 在客户端中输入服务器 IP、端口和昵称
4. 连接成功后即可开始聊天