南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机网络实验报告

利用流式套接字编写聊天程序

张得涵

年级：2023 级

专业：计算机科学与技术

指导教师：徐敬东，张建忠

2025 年 10 月 23 日

目录

一、 实验要求

- 给出聊天协议的完整说明;
- 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;
- 使用流式套接字，采用多线程（或多进程）方式完成程序;
- 程序应该有基本的对话界面，但可以不是图形界面。程序应该有正常的退出方式;
- 完成的程序应该支持多人聊天，支持英文和中文聊天;
- 编写的程序应该结构清晰，具有较好的可读性;
- 在实验中观察是否有数据包丢失，提交可执行文件、程序源码和实验报告。

二、 实验原理

(一) TCP/IP 协议

TCP/IP 协议是互联网的核心通信协议族，包含应用层、传输层、网络层和链路层。在本实验中，我们主要使用其传输层的 TCP（Transmission Control Protocol）协议。

TCP 是一种面向连接、可靠的、基于字节流的传输协议。它通过三次握手建立连接，确保数据按序、无差错地传输，并在通信结束后通过四次挥手释放连接。由于聊天程序要求消息不丢失、不乱序，因此选择 TCP 而非 UDP 作为传输层协议。

本实验通过 Socket 接口封装 TCP 通信细节，实现客户端与服务器之间的稳定数据传输。

(二) socket 编程

Socket（套接字）是应用层与传输层之间的一个抽象编程接口，屏蔽了底层复杂的网络协议细节。本实验采用客户端-服务器（C/S）模型进行通信:

- **服务器端**: 创建监听套接字，绑定 IP 地址和端口，等待客户端连接请求。每当有新客户端接入，为其创建独立线程处理通信。
- **客户端**: 主动向服务器发起连接请求，连接成功后可发送消息，同时开启接收线程监听服务器广播的消息。

通过 Socket 编程，程序可以像操作文件一样进行网络读写（使用 `send` 和 `recv` 函数），从而实现跨进程、跨主机的通信。

(三) Socket 常用函数

本实验中，客户端与服务器均基于 Winsock API 实现 TCP 通信。服务器采用多线程模型处理并发连接，客户端实现持续接收消息的异步机制。以下是本项目中实际使用的核心函数及其作用:

- `WSAStartup()` 和 `WSACleanup()`: 初始化和清理 Winsock 库。Windows 平台下所有套接字操作前必须调用 `WSAStartup`，程序结束前调用 `WSACleanup` 释放资源。

  - 服务端在 main() 中调用：WSAStartup(MAKEWORD(2,2), &wsa)
  - 客户端在连接前调用，程序结束前调用 WSACleanup()

- socket(AF_INET, SOCK_STREAM, 0)：创建一个 TCP 套接字。服务器和客户端均调用此函数创建通信端点。
  - 服务端用于创建监听套接字
  - 客户端用于创建连接套接字
  - 返回类型为 SOCKET（Windows 特有）

- bind()：将服务器的监听套接字绑定到指定 IP 和端口。服务端调用此函数使套接字监听本地地址。
  - 使用 sockaddr_in 结构设置地址信息
  - 本实验绑定到 INADDR_ANY:8080，表示监听所有网卡的 8080 端口

- listen(socket, 5)：将套接字设为监听模式，并设置连接请求队列长度为 5。服务端在 bind() 后调用。
  - 第二个参数指定最大等待连接数
  - 超出队列长度的连接请求将被拒绝

- accept()：阻塞等待客户端连接请求。每当有新客户端连接，accept() 返回一个新的套接字，用于与该客户端通信。
  - 服务端在主循环中调用：accept(server_sock, (SOCKADDR*)&client_addr, &len)
  - 返回的 client_sock 交给新线程处理，实现并发

- connect()：客户端调用此函数主动连接服务器。
  - 使用 sockaddr_in 指定服务器 IP 和端口
  - 成功后建立 TCP 连接，可进行数据收发

- send() 和 recv()：基于已建立的 TCP 连接发送和接收数据。
  - 服务端调用 send() 向所有客户端广播消息
  - 客户端调用 recv() 在独立线程中持续接收服务器消息
  - 本实验中，recv() 配合循环使用，实现异步接收

- closesocket()：关闭套接字，释放系统资源。服务器在客户端断开后调用，客户端在退出时调用。
  - 服务端在 handle_client 线程结束时调用
  - 客户端在用户退出时调用
  - 注意：close() 是 Unix/Linux 系统函数，Windows 使用 closesocket()

- CreateThread()（Windows API）：创建新线程处理客户端连接或异步接收消息。
  - 服务端为每个客户端创建一个线程执行 handle_client

  - 客户端创建线程执行 receive_thread，避免阻塞主循环

- CreateMutex() 和 WaitForSingleObject()/ReleaseMutex()：实现线程同步，保护共享资源（如客户端列表）。
  - 服务端使用互斥锁确保对 clients 和 usernames 的访问是线程安全的
  - 在修改共享数据前加锁，操作完成后解锁

这些函数构成了本实验 TCP 聊天系统的核心通信机制。通过合理组合 socket、bind、listen、accept 等函数，实现了服务器的并发处理能力；通过 CreateThread 和 recv 的配合，实现了客户端的异步消息接收，保证了用户体验的流畅性。

三、 协议设计

(一) 总体设计 (Message.cpp)

本实验实现了一个简单的应用层聊天协议。为了确保客户端和服务端之间数据交换的统一性、可读性和可扩展性，本协议设计采用了 JSON (JavaScript Object Notation) 格式作为应用层消息数据的封装方式。所有在网络中传输的消息，无论是登录请求、聊天内容还是系统通知，都统一封装在一个自定义的 Message 类对象中，然后在发送前序列化为 JSON 字符串进行传输。

(二) 协议报文 JSON 格式概述

本协议的核心是 JSON 报文格式，它定义了所有消息的统一数据结构。每个报文严格包含四个键值对，全部采用字符串类型。

报文在网络中传输时,以如下 JSON 格式的字符串形式呈现,并在代码中通过 Message::toString() 方法进行序列化生成:

Listing 1: 应用层报文 JSON 格式

```
{
    "type":"<消息类型>",
    "user":"<用户名>",
    "msg":"<消息内容>",
    "time":"<时间戳>"
}
```

- type：**消息类型**，用于指示接收方应如何解析和处理此消息。
- user：**发送方标识**，即该消息发送者的用户名或昵称。
- msg：**负载数据**，承载了聊天文本或系统通知的具体内容。
- time：**时间戳**，记录消息在发送端生成的准确时间（格式为 HH:MM:SS）。

这种基于 JSON 的设计使得协议具有良好的自描述性，易于理解和调试。

(三) 报文类型定义与功能

根据 type 字段的值，本协议定义了四种主要的消息类型，实现了程序的核心功能：

- **type = "login"：登录/连接请求。**
  - 由客户端在成功连接到服务器后，发送其用户名。
  - 服务器接收后，将该客户端加入在线列表。
- **type = "chat"：聊天消息。**
  - 由客户端发送给服务器，包含用户的聊天文本。
  - 服务器接收后，将此报文原样广播给所有在线的其他客户端。
- **type = "logout"：登出/断开连接。**
  - 由客户端主动输入 quit 命令时发送。
  - 服务器接收后，将其从在线列表中移除，并关闭其连接。
- **type = "system"：系统通知。**
  - 仅由服务器生成和发送，用于通知所有客户端当前状态。
  - 典型的消息内容包括："User joined."（用户加入）、"User left."（用户离开）。

(四) Message 类详解：序列化与反序列化

Message 类是 C++ 中对上述 JSON 报文结构的抽象封装，负责将 C++ 对象与网络传输所需的字符串格式进行互相转换。

1. Message.h 结构定义

Message 类采用 C++ 封装，将四个数据字段定义为私有成员，并通过公有的 Getter 方法（如 getType()）提供只读访问权限。核心的转换方法为：

Listing 2: Message.h 核心接口

```
class Message {
private:
    std::string type;
    std::string user;
    std::string msg;
    std::string time;
    // ... 私有工具函数 extract_value_robust ...

public:
    // 构造函数
    Message(const std::string& t, const std::string& u,
            const std::string& m, const std::string& tm);

    // 序列化：将对象转换为 JSON 字符串
    std::string toString() const;
```

```
16
17      // 反序列化：通过工厂公有静态方法将 JSON 字符串解析为 Message 对象
18      static Message from_json(const std::string& json);
19
20      // 通过Getter方法保护消息内容不会泄露
21  };
```

（五） Message 类详解：序列化与反序列化

由于 C++ 没有自带的 JSON 解析库，又不太想使用别人的库，于是我简单定义了一个用于解析并存储 JSON 格式文件的类。Message 类是对上述 JSON 报文结构的抽象封装，负责实现对象和字符串之间的互相转换。

1. Message.h 结构定义

Message 类采用 C++ 封装，将四个数据字段定义为私有成员，并通过公有的 Getter 方法提供只读访问权限以保护内部数据。核心的方法有：

Listing 3: Message.h 核心接口

```
1   class Message {
2   private:
3       std::string type;
4       // ... 其他私有成员 ...
5       static std::string extract_value_robust(const std::string& json, const
            std::string& key);
6
7   public:
8       // 构造函数
9       // ...
10      // 序列化：将对象转换为 JSON 字符串
11      std::string toString() const;
12
13      // 反序列化：将 JSON 字符串解析为 Message 对象
14      static Message from_json(const std::string& json);
15      // ... Getter 方法 ...
16  };
```

2. 序列化实现详解：报文构建逻辑

通过 toString() 方法将程序内部的 Message 对象转换为符合协议规范的 JSON 字符串，以便通过网络发送。该方法的实现逻辑是：

1. **动态创建缓冲区：**首先，在堆上动态分配一个固定大小的 char 缓冲区 (char* buffer)。这用于临时存储即将生成的 JSON 字符串。
2. **核心格式化操作：**使用 C 风格的格式化函数 snprintf 来完成 JSON 报文的构建。

3. **定义 JSON 模板：**该函数使用一个固定的格式化字符串作为模板，该模板严格遵循 JSON 结构：

```
"{\\\"type\\\":\\\"%s\\\",\\\"user\\\":\\\"%s\\\",\\\"msg\\\":\\\"%s\\\",\\\"
    time\\\":\\\"%s\\\"}"
```

   在这个模板中，`%s` 是 C 语言的占位符，用于插入 `Message` 对象的四个成员变量的值；而反斜杠 (\) 则用于转义双引号，确保生成的字符串中的键和值都被正确的双引号 (") 包裹。

4. **插入数据并生成报文：**将 `Message` 对象的四个 `std::string` 成员通过 `.c_str()` 方法，依次填入格式化字符串的 `%s` 占位符中。格式化后的完整 JSON 字符串被写入到 `buffer` 中。

5. **返回与清理：**最后，将 `buffer` 中的 C 风格字符串转换为 C++ 的 `std::string` 类型并返回，随后释放动态分配的 `buffer` 内存。

安全保护：使用 `snprintf`(而不是 `sprintf`)是为了在格式化时限制写入的字节数(即 `MAX_BUFFER_SIZE`)，这是一种防止缓冲区溢出 (`Buffer Overflow`) 的安全措施，确保了程序的稳健性。

3. 反序列化实现详解：手动解析逻辑

反序列化 (`from_json()`) 是将接收到的原始 JSON 字符串解析为可操作的 `Message` C++ 对象的过程。本实验采用了一种高效且针对本协议格式的手动字符串查找与提取方案，核心逻辑在私有方法 `extract_value_robust()` 中实现。

1. **查找模式构造：**对于需要提取的每个键 (例如 `"type"`)，我们构造一个精确的查找模式，如：`"key":"`。

2. **值起始定位：**使用 `json.find(pattern)` 找到模式的起始位置,然后将值的起始位置 (`value_start`) 定位在模式字符串之后。

3. **值结束定位：**从 `value_start` 开始，逐个字符向后查找。关键在于找到该值后面的第一个未被转义 (\) 的双引号 (")。这个引号即是该值在 JSON 格式中的结束标记。

   - 实现细节：代码通过一个 `bool in_escape` 标志来判断当前双引号是否被前一个反斜杠转义。只有当 `in_escape` 为 `false` 且当前字符为双引号时，才视为值的结束。

4. **子串提取：**确定了值的起始和结束位置后，利用 `std::string::substr()` 方法准确地提取出值内容，并将其赋给 `Message` 对象的对应成员。

四、 功能实现与代码分析

本实验基于 WinSock2 库，通过流式套接字 (SOCK_STREAM) 实现了服务器和客户端程序。所有核心功能均围绕多线程并发和自定义 JSON 协议的收发而展开。

(一) 服务器端 (server.cpp) 实现

服务器端负责监听连接、管理客户端列表、处理并发请求和广播消息。

1. 网络初始化与监听

服务器端首先在 main 函数中完成 Winsock 库的初始化和套接字的基本设置:

1. **初始化 Winsock:** 调用 WSAStartup(MAKEWORD(2, 2), &wsa) 确保正确加载 Winsock 库。
2. **创建套接字:** 使用 socket(AF_INET, SOCK_STREAM, 0) 创建 TCP 监听套接字 (server_sock)。
3. **绑定与监听:** 将套接字绑定到预设的 PORT (8080) 和 INADDR_ANY 地址。随后调用 listen(server_sock, 5) 将套接字置于监听状态，允许最多 5 个连接请求排队。
4. **线程同步资源:** 创建互斥锁 (HANDLE clients_mutex), 用于保护全局客户端列表 (clients 和 usernames 向量)，确保多线程环境下的数据安全。

2. 并发处理与客户端接入

服务器通过主循环实现对多客户端的并发支持:

- **主循环 (while(1)):** 服务器在循环中调用阻塞函数 accept()。
- **接受连接:** 一旦有客户端连接，accept() 返回一个新的通信套接字 (client_sock)。
- **创建线程:** 服务器立即调用 CreateThread(...)，为每个新连接分配一个独立的线程来执行 handle_client 函数。这实现了 I/O 并发，避免了某一客户端阻塞整个服务器的运行。

3. 客户端消息处理线程 (handle_client)

handle_client 是服务器的核心逻辑，负责单个客户端的整个生命周期管理:

1. **登录处理:**
   - 线程首先接收客户端的第一个 type="login" 报文，并使用 Message::from_json() 反序列化，提取 username。
   - 通过互斥锁 (WaitForSingleObject) 保护，将 client_sock 和 username 存入全局列表。
   - 构造 type="system" 消息，通过 broadcast() 广播该用户已加入的消息给所有在线用户。
2. **消息收发循环:** 线程进入主循环，持续调用 recv() 等待接收消息。
3. **消息转发与退出判断:**
   - 若接收到 type="chat" 消息，则调用 broadcast() 转发给所有客户端。

- 若接收到 type="logout" 消息或 recv() 返回 0（客户端断开），则进入退出处理流程。

4. **退出与清理:**线程在清理阶段,通过互斥锁从全局列表中移除该客户端信息,调用 closesocket() 关闭连接，并广播该用户已离开的系统通知。

4. 消息广播机制 (broadcast)

broadcast 函数实现了线程安全的群发功能:

- **互斥访问:** 在操作全局客户端列表前，使用 WaitForSingleObject 获取互斥锁，确保数据在多线程环境下的完整性。
- **循环发送:** 遍历 clients 向量中的所有套接字，调用 send() 将 JSON 格式的字符串消息发送给每个在线客户端。
- **释放锁:** 循环结束后，调用 ReleaseMutex 立即释放互斥锁。

(二) 客户端 (client.cpp) 实现

客户端采用多线程设计，以实现消息的 ** 异步收发 **，提升用户体验。

1. 连接与登录流程

1. **初始化与连接:** 初始化 Winsock，创建套接字，并调用 connect() 连接到服务器（默认为 127.0.0.1:8080）。
2. **发送登录报文:** 用户输入 username 后，客户端构造 type="login" 的 Message 对象，序列化后通过 send() 发送给服务器。

2. 异步接收线程 (receive_thread)

为保证用户在输入时能够同时接收到新消息，客户端创建了单独的接收线程:

- **接收循环:** 该线程进入无限循环，调用阻塞函数 recv() 等待数据。
- **消息处理:** 接收到数据后，立即调用 Message::from_json() 反序列化报文，并根据 type 字段格式化输出:
  - type="system": 显示 [SYSTEM] 提示信息。
  - type="chat": 显示 [<user>]: <msg> 聊天内容。
- **断线处理:** 若 recv() 返回 0 或小于 0 的值，线程打印服务器断线提示后退出，并通知主线程清理资源。

3. 主线程的消息发送与退出

客户端的主线程专注于用户交互和消息发送:

- **发送循环:** 主线程在循环中通过 std::getline(cin, input) 等待用户输入。
- **聊天消息:** 接收到普通输入时，构造 type="chat" 报文，序列化后发送给服务器。

- **正常退出机制：**当用户输入 "quit" 时，程序执行以下步骤：
  1. 构造 type="logout" 报文并发送给服务器。
  2. 退出主线程的发送循环。
  3. 调用 closesocket() 和 WSACleanup() 清理资源，程序终止。

五、 测试

(一) 编译指令

本项目包含三个核心源代码文件:server.cpp(服务器端)、client.cpp(客户端)和 Message.cpp/Message.h（自定义协议消息封装）。由于程序使用了 Windows API（如 winsock2.h 和 windows.h），在 MinGW 或 Cygwin 环境下使用 g++ 编译时，必须链接 Winsock 库 (-lws2_32)。

编译指令如下:

- **服务器端编译：**

Listing 4: 编译服务器端程序

```
g++ server.cpp Message.cpp -o server.exe -lws2_32
```

- **客户端编译：**

Listing 5: 编译客户端程序

```
g++ client.cpp Message.cpp -o client.exe -lws2_32
```

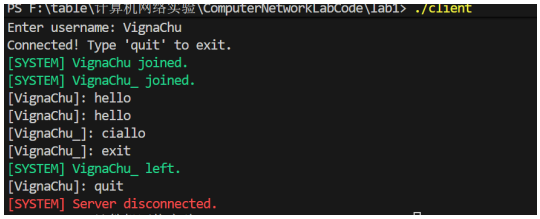
(二) 运行流程

程序采用经典的 C/S（客户端-服务器）模型，必须先启动服务器端程序，客户端才能连接。运行步骤如下:

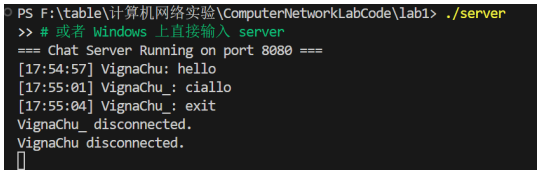
1. **启动服务器：**在命令行中运行 server.exe。服务器将启动监听预设端口（默认为 8080），并等待客户端连接。
2. **启动客户端（多实例）：**打开至少两个独立的命令行窗口，分别运行 client.exe。每个客户端启动后，程序会提示用户输入一个唯一的 username 进行登录。
3. **进行聊天测试：**任一客户端输入消息后，按回车键发送。消息将经由服务器接收、广播，并被所有在线的客户端异步接收并显示。聊天内容支持中英文输入。
4. **观察系统通知：**观察服务器端控制台，它会记录用户连接和断开的日志。同时，客户端会收到 [SYSTEM] 类型的通知，表明有用户加入或离开。
5. **正常退出：**客户端用户输入 quit 命令后，程序将构造 type="logout" 报文通知服务器，然后安全关闭套接字并退出。服务器端会收到通知，并将该用户从在线列表中移除。

(三) 运行效果展示

经过上述流程测试，系统运行稳定，支持多客户端并发连接与通信。



(a) 客户端 1 和客户端 2 的并发聊天界面



(b) 服务器端实时控制台日志

图 1: 多人聊天系统运行效果展示

如图 1 所示，系统通过 TCP 协议保证了消息的有序和可靠传输。客户端登录后，服务器端 (1b) 会记录用户的连接和断开信息，并将聊天消息广播给所有在线客户端 (1a)。

(四) 观察数据包的传递接受与丢失

在本次实验中，我通过 wireshark 工具实现对数据包的监听

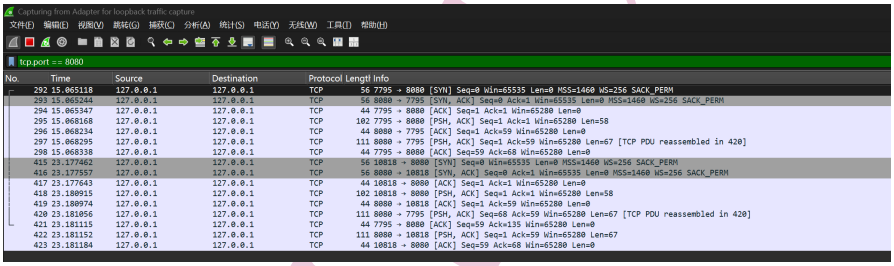


图 2: 通过 wireshark 进行监听

可以从图中看到 TCP 中的三次握手（这里以最上面三条为例）

表 1: TCP 三次握手过程

序号	方向（端口）	说明
292	7795 → 8080 [SYN]	客户端 A 发起连接请求
293	8080 → 7795 [SYN, ACK]	服务器响应，同意连接
294	7795 → 8080 [ACK]	客户端确认，连接建立完成

点进消息还可以看到具体传输的协议内容：

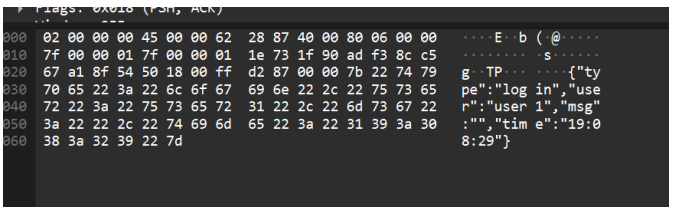


图 3: 查看传过去的 json 文件 (注：其中的空格只是为了对齐加上的，无实意)

可以看到成功将用户登录的 json 消息传了过去，包括登录时间，用户昵称等。

由于本机传输基本不会产生丢包，因此为了观察丢包现象，我使用 clumsy 工具人为的造成丢包。人为制造丢包，观察 TCP 是否能自动恢复

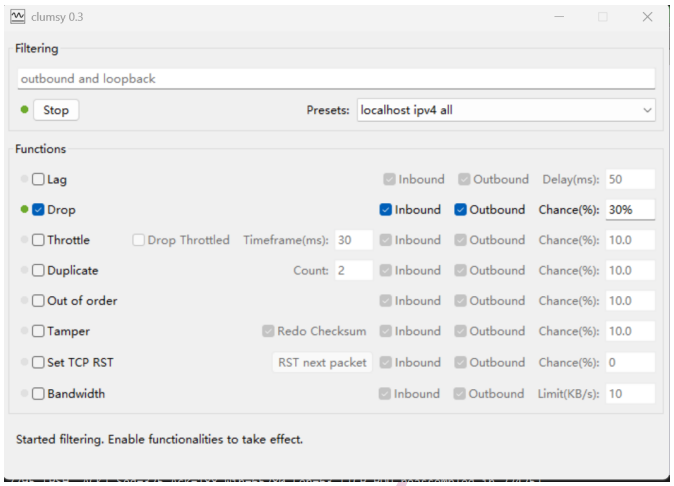


图 4: 使用 clumsy 制造丢包

这里我把丢包率调成了 50%, 并继续发消息，成功用 wireshark 工具捕捉到了丢包现象 图中

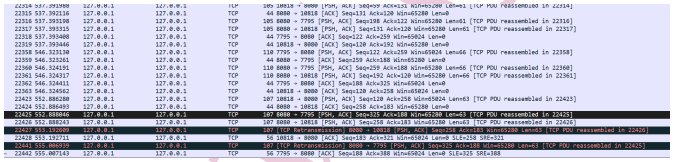


图 5: 成功捕捉到了丢包

几条黑色的就是丢包提醒，并在之后捕获到了客户端的重传现象。

此外用户在通讯中感受到了明显的卡顿

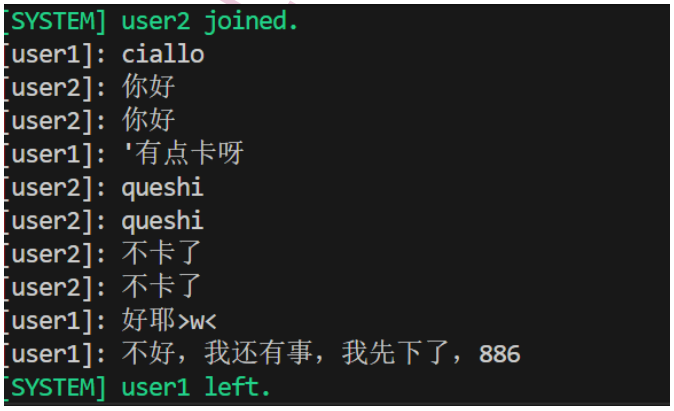


图 6: 产生了一些卡顿

六、 实验总结与心得体会

(一) 技术收获

本次计算机网络实验成功地利用 C++ 和 WinSock2 库实现了一个基于 TCP 的多用户聊天系统，在实践中对网络编程和并发处理有了深刻的认识：

1. **TCP/IP 基础实践：** 掌握了 `WSAStartup`、`socket`、`bind`、`listen`、`accept` 和 `connect` 等核心函数的使用流程，清晰地建立了服务器端“监听-接受-服务”的经典模型。
2. **多线程并发：** 服务器端采用 `CreateThread` 为每个客户端分配独立线程 (`handle_client`)，有效地实现了 I/O 并发。这确保了服务器在处理一个客户端的阻塞式 `recv()` 时，不会暂停对新连接的接受和对其他客户端的转发服务。
3. **应用层协议设计：** 深刻理解了在应用层设计协议的必要性。通过自定义 `Message` 类和基于 JSON 的报文格式，使得客户端和服务器之间的通信具有明确的结构和语义（`login`, `chat`, `system`），大大提高了系统的可读性和可扩展性。

(二) 调试难点与反思

在实验过程中，主要在以下两个方面耗费了大量的调试精力：

1. Winsock 头文件冲突问题

最初在引入 `<windows.h>` 和 `<winsock2.h>` 时，由于头文件包含顺序或宏定义缺失，导致了编译器报出数百个关于网络函数和结构体重定义的 ** 级联错误 (Cascading Errors)**。这一问题通过调整头文件顺序并将 `#define WIN32_LEAN_AND_MEAN` 等宏置于所有头文件之前而得以解决。这次调试让我认识到 Windows API 中头文件的依赖和优先级，必须严格遵守。

2. 多线程数据同步

服务器端对全局客户端列表 (`clients` 和 `usernames`) 的读写是多线程环境下的关键挑战。在 `handle_client` 线程移除用户、或 `broadcast` 函数遍历列表时，如果缺少同步机制，极易发生各种类型的冲突。通过引入 `HANDLE clients_mutex` 互斥锁，并在所有涉及列表操作的关键代码段使用 `WaitForSingleObject` 进行保护，最终保证了列表操作的原子性和线程安全性。从而保障服务器稳定运行。

(三) 未来展望与改进方向

虽然本次实验完成了基本的多人聊天功能，但仍有诸多方面可以进行改进：

1. **消息长度限制与分包：** 目前的消息收发是基于固定的 `BUFFER_SIZE`，适用于短消息。未来应实现完整的应用层分包和重组机制，以可靠传输长度可变的超长消息。
2. **错误处理细化：** 对 `send()` 和 `recv()` 返回的错误码处理可以更加精细，区分网络瞬时故障和永久断开，从而实现更优雅的错误恢复机制。

总而言之，本次实验是理论知识向实际编码转化的宝贵机会，加深了对 TCP 协议可靠性、并发控制和应用层协议设计的理解。

七、 代码仓库

本次实验的原码和报告均放到计算机网络实验课报告原码中。