南開大學
Nankai University

计算机网络

利用流式套接字编写聊天程序

姓名：李子恒

学号：2312114

专业：密码科学与技术

2025 年 11 月 17 日

Abstract

本实验设计并实现了一个基于 TCP 协议的多人聊天程序，采用客户端-服务器架构，支持多用户实时消息交流。程序使用 C++ 语言编写，具有跨平台兼容性，可在 Windows 和 Linux 系统上运行。服务器端采用多线程技术处理并发连接，客户端支持消息发送和接收功能。本报告详细阐述了系统的设计原理、协议规范、核心代码实现、运行方法以及开发过程中遇到的技术挑战与解决方案。通过本次实验，深入理解了网络编程的核心概念和技术，掌握了套接字编程、多线程并发控制和跨平台开发方法。

目 录

1 引言

1.1 实验目的

掌握 TCP/IP 网络编程的基本原理，学会流式套接字的创建、连接、数据传输及关闭等核心操作，深入理解客户端-服务器架构的设计与实现方法。通过完成一个具备基本功能的多人聊天系统，学生将有效培养网络应用程序的调试与排错能力。

1.2 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 系统架构与协议设计

2.1 系统架构

本聊天系统采用如图一所示的典型客户端-服务器架构，使用 TCP 协议保证数据传输的可靠性。系统由两个主要部分组成：

1. **服务器端**：负责监听客户端连接请求, 转发用户消息
2. **客户端**：负责与用户交互，发送用户输入的消息，接收并显示来自服务器的消息

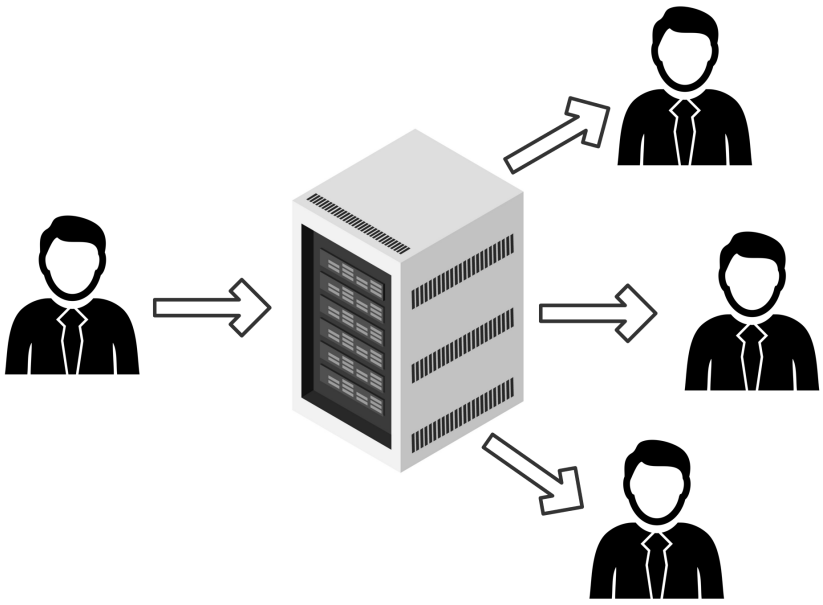

图 1: 系统架构示意图

2.2 消息类型

本聊天系统设计了五种不同类型的消息，用于实现各种功能：

表 1: 消息类型定义

类型码	消息类型	功能描述
0	登录消息	客户端向服务器发送登录请求
1	登出消息	客户端向服务器发送登出请求
2	广播消息	用户向所有在线用户发送消息
3	私聊消息	用户向特定用户发送私密消息
4	系统消息	服务器向客户端发送系统通知

2.3 协议语法与语义

2.3.1 基本格式设计

本系统采用简单直观的文本协议，使用竖线（|）作为字段分隔符。消息格式如下：

type|username|message

该设计格式兼具简洁性、可扩展性与可读性。其结构简单明了，易于解析与实现，同时也预留了良好的扩展能力，能便捷地增添新的消息类型；此外，清晰的格式也为调试和问题定位提供了极大便利。其中：

- **type**：消息类型代码（0-4）
- **username**：用户名或系统标识
- **message**：消息内容，不同类型的消息有不同的格式

2.3.2 消息格式语义

根据消息类型码，我们自然可以定义这五类消息的具体格式与语义：

1. **登录消息**：格式为”0|username|”，客户端发送此消息以加入聊天室
   - 字段含义：消息类型 0，用户名，空消息体
   - 处理逻辑：服务器验证用户名，将用户加入在线列表，广播系统通知
2. **登出消息**：格式为”1|username|”，客户端发送此消息以离开聊天室
   - 字段含义：消息类型 1，用户名，空消息体
   - 处理逻辑：服务器将用户从在线列表移除，广播系统通知，关闭连接
3. **广播消息**：格式为”2|username|content”，向所有用户发送公共消息
   - 字段含义：消息类型 2，发送者用户名，消息内容
   - 处理逻辑：服务器转发给除发送者外的所有在线用户
4. **私聊消息**：格式为”3|sender|target:content”，向特定用户发送私密消息
   - 字段含义：消息类型 3，发送者用户名，目标用户: 消息内容
   - 处理逻辑：服务器解析目标用户，单独转发给指定用户，并发送确认消息
5. **系统消息**：格式为”4|System|content”，服务器发送的系统通知
   - 字段含义：消息类型 4，系统标识，通知内容
   - 处理逻辑：客户端以特殊格式显示系统消息

2.4 协议时序

这个聊天系统的基本工作流程如下：当一个客户端成功连接到服务器后，它会先发送一条登录消息完成身份注册；随即，服务器会向所有已在线用户广播一条该用户加入的通知。在连接期间，用户既可以发送广播消息给所有人，也可以指定接收者发送私聊消息，而服务器则负责精准地识别并转发这些消息。当用户决定退出时，客户端会发送一条登出消息，服务器在收到后，会再次通知其他用户该成员已离开，随后才关闭与该客户端的连接，典型的通信时序如图 2 所示。

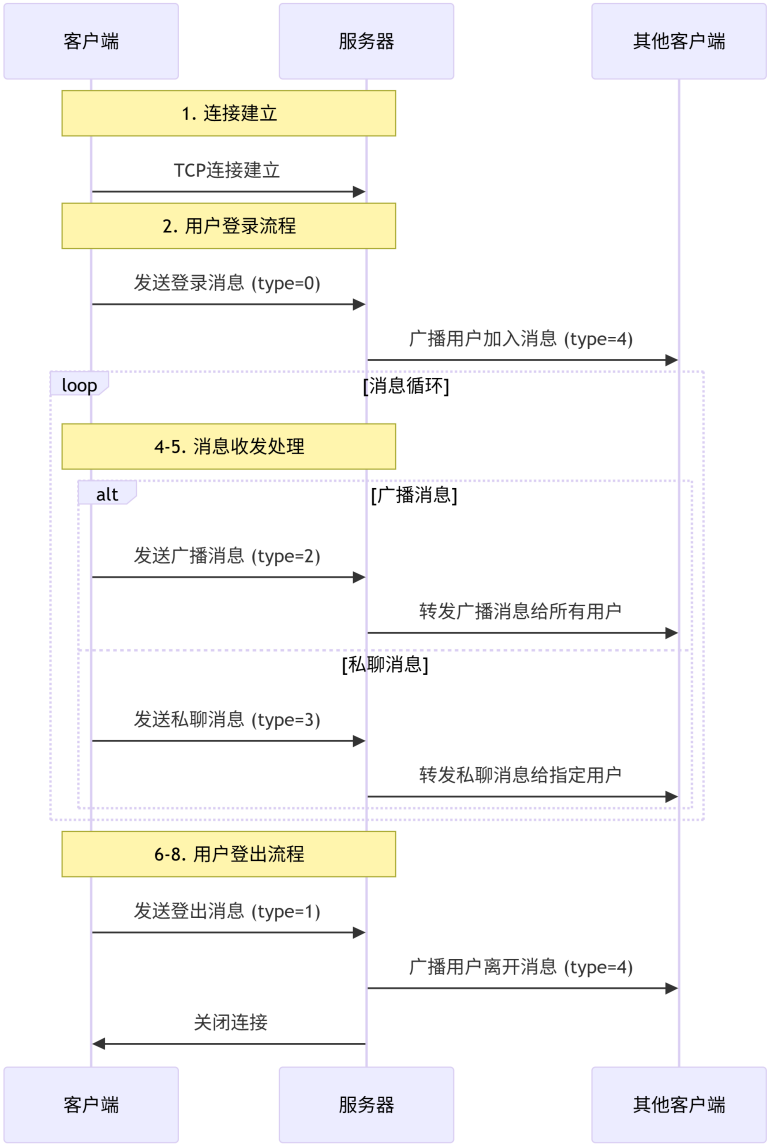

图 2: 通信时序

2.5 协议创新性

本协议设计在传统聊天协议的基础上进行了以下创新：

1. **直观的私聊机制**：使用”@ 用户名: 消息” 格式，简化了用户操作
2. **双向确认机制**：私聊消息发送后，服务器会向发送者返回确认消息
3. **错误处理完善**：包含超时机制和异常连接处理

3 核心代码实现

3.1 服务器端实现

3.1.1 数据结构设计

服务器端使用以下核心数据结构来管理客户端连接和状态：

Listing 1: 客户端信息结构

```
// 客户端信息结构
typedef struct {
    SocketType socket;          // 客户端套接字
    char username[50];          // 客户端用户名
    struct sockaddr_in address; // 客户端网络地址
} ClientInfo;

// 全局变量
std::vector<ClientInfo*> clients; // 存储所有连接的客户端
CRITICAL_SECTION clientsMutex;    // Windows平台的互斥锁，保证线程安全
bool serverRunning;               // 服务器运行状态标志
```

数据结构分析：

- **ClientInfo 结构体**：封装了单个客户端的完整信息，包括通信套接字、用户名和网络地址，便于统一管理
- **clients 向量**：动态管理所有在线客户端，支持添加、删除和遍历操作
- **clientsMutex 互斥锁**：在多线程环境下保护共享资源 clients 向量，避免数据竞争
- **serverRunning 标志**：控制服务器主循环，用于优雅关闭服务器

3.1.2 跨平台兼容设计

系统采用条件编译实现了 Windows 和 Linux 平台的兼容性：

Listing 2: 跨平台兼容性定义

```
#ifdef _WIN32
    #include <WinSock2.h>
    #include <WS2tcpip.h>
    #include <process.h> // _beginthreadex
    #include <windows.h> // SetConsoleOutputCP, SetConsoleCP
    #pragma comment(lib, "ws2_32.lib")
    typedef SOCKET SocketType;
    #define INVALID_SOCKET_VALUE INVALID_SOCKET
    #define CLOSE_SOCKET(s) closesocket(s)
    typedef unsigned (__stdcall *ThreadFunction)(void*);
    #define CREATE_THREAD(func, arg) _beginthreadex(NULL, 0, (ThreadFunction)func, arg,
        0, NULL)
#else
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>
```

```
    #include <unistd.h>
    #include <fcntl.h>
    #include <errno.h>
    #include <pthread.h>
    typedef int SocketType;
    #define INVALID_SOCKET_VALUE -1
    #define CLOSE_SOCKET(s) close(s)
    #define CREATE_THREAD(func, arg) pthread_create(NULL, NULL, func, arg)
#endif
```

设计分析：

- 通过类型别名（如 SocketType）抽象平台差异，统一代码风格
- 使用宏定义封装不同平台的 API 调用，简化代码复杂度
- 对线程创建、套接字操作等平台相关功能进行封装，提供一致的接口

3.1.3 核心函数功能分析

1. 初始化函数 该函数的主要作用是为整个多线程 Socket 聊天程序完成底层环境的准备工作。在 Windows 系统中，它首先调用 WSAStartup 初始化 Winsock 库，并指定使用 2.2 版本的 Socket 接口，这是后续所有网络通信功能能够正常运行的前提。接着，函数通过设置控制台输入输出编码为 UTF-8，以确保程序在发送与接收包含中文的聊天内容时能够正确显示，不会出现乱码问题。此外，函数还负责创建跨线程共享资源时所需的同步机制：在 Windows 平台下初始化临界区（CRITICAL_SECTION），在 Linux/Unix 系统中初始化 POSIX 互斥锁，以确保多个线程访问客户端列表等公共数据结构时不会发生竞争或数据损坏。通过这些初始化步骤，该函数为后续网络连接、消息收发和多线程调度提供了稳定、可靠的运行环境，并以布尔返回值指示初始化是否成功，以便主程序根据结果进行错误处理或继续执行。

Listing 3: 初始化 Winsock

```
bool initializeWinsock() {
#ifdef _WIN32
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        printError("WSAStartup failed");
        return false;
    }

    // 设置控制台为UTF-8编码，支持中文显示
    SetConsoleOutputCP(CP_UTF8);
    SetConsoleCP(CP_UTF8);

    InitializeCriticalSection(&clientsMutex);
#else
    pthread_mutex_init(&clientsMutex, NULL);
#endif
    return true;
}
```

2. 消息发送函数 这两个函数共同构成了服务器端消息发送与广播的核心机制。sendToClient 函数对底层的 send 系统调用进行了封装，屏蔽了 Windows 与 Linux 在错误码与返回值处理上的差异，确保在不

同平台上都能稳定向指定客户端发送文本消息；同时，它通过返回布尔值清晰指示发送操作是否成功，以便上层逻辑进行相应处理。broadcastMessage 则用于实现服务器的群发功能，它在进入遍历客户端列表前先加锁，以保护 clients 向量在多线程环境下的访问安全，避免多个线程同时增删客户端导致的数据竞争。在广播过程中，函数会跳过消息发送者本身，从而避免消息被回显给原用户；其余所有在线客户端都会收到完整消息，实现群聊的基本能力。通过两者的配合，服务器不仅能够稳定处理单点消息发送，还能够在高并发环境下安全、可靠地完成广播通信。

Listing 4: 发送消息给客户端

```
bool sendToClient(const ClientInfo* client, const char* message) {
#ifdef _WIN32
    if (send(client->socket, message, strlen(message), 0) == SOCKET_ERROR) {
        printError("Send failed");
        return false;
    }
#else
    if (send(client->socket, message, strlen(message), 0) == -1) {
        printError("Send failed");
        return false;
    }
#endif
    return true;
}
```

Listing 5: 广播消息函数

```
void broadcastMessage(const char* message, const char* senderUsername) {
    lockMutex(); // 加锁保护共享资源
    for (size_t i = 0; i < clients.size(); ++i) {
        if (strcmp(clients[i]->username, senderUsername) != 0) {
            sendToClient(clients[i], message);
        }
    }
    unlockMutex(); // 释放锁
}
```

3. 私聊消息处理 该函数负责处理客户端之间的私聊请求，其核心逻辑包括私聊信息的解析、目标用户的定位以及最终的结果反馈。函数首先依据约定的消息格式”目标用户名: 消息内容”对输入字符串进行解析，提取出目标用户的名称以及实际需要发送的消息内容。随后进入受互斥锁保护的临界区，对当前在线的客户端列表进行遍历查找：若找到匹配的目标用户，则构造带有私聊标识的消息格式并将内容发送给对方，并标记发送成功；若未找到，则认为该用户不存在或不在线。紧接着，函数再次在锁的保护下向消息发送者回传确认信息，无论私聊是否成功，都以系统消息的形式明确告知发送者结果，从而保证用户端能够获得清晰的状态反馈。通过这种设计，函数不仅确保了私聊消息在多线程环境下的安全传递，而且完善了错误提示和结果反馈机制，使私聊功能更加可靠、可用性更强。

Listing 6: 私聊消息处理函数

```
void handlePrivateMessage(const char* message, const char* senderUsername) {
    // 解析目标用户名，格式为"targetUsername:messageContent"
    const char* colonPos = strchr(message, ':');
```

```
    if (colonPos != NULL) {
        char targetUsername[50];
        strncpy(targetUsername, message, colonPos - message);
        targetUsername[colonPos - message] = '\0';

        const char* actualMessage = colonPos + 1;
        bool userFound = false;

        // 查找并发送给目标用户
        lockMutex();
        for (size_t i = 0; i < clients.size(); ++i) {
            if (strcmp(clients[i]->username, targetUsername) == 0) {
                char privateMsg[1024];
                sprintf(privateMsg, "3|%s|%s", senderUsername, actualMessage);
                sendToClient(clients[i], privateMsg);
                userFound = true;
                break;
            }
        }
        unlockMutex();

        // 向发送者发送确认消息
        lockMutex();
        for (size_t i = 0; i < clients.size(); ++i) {
            if (strcmp(clients[i]->username, senderUsername) == 0) {
                char confirmMsg[1024];
                if (userFound) {
                    sprintf(confirmMsg, "4|System|私聊消息已发送给 %s", targetUsername);
                } else {
                    sprintf(confirmMsg, "4|System|用户 %s 不存在或不在线",
                        targetUsername);
                }
                sendToClient(clients[i], confirmMsg);
                break;
            }
        }
        unlockMutex();
    }
}
```

功能分析：

- 解析消息格式，提取目标用户名和实际消息内容
- 遍历客户端列表查找目标用户，如存在则发送私聊消息
- 无论目标用户是否存在，都向发送者发送状态确认消息
- 使用互斥锁保护对 clients 向量的并发访问
- 实现了私聊消息的可靠传递和状态反馈机制

4. 客户端处理线程 该线程函数专门用于处理单个客户端的所有通信任务，是服务器并发架构的核心组成部分。服务器为每一个成功连接的客户端创建一个独立线程，使多个客户端能够同时与服务器交互而互不干扰。在该线程内部，以循环方式持续监听来自该客户端的消息，并根据 recv 的返回值对各种异常情况进行判断，包括网络错误、非阻塞模式下暂时无数据可读以及客户端主动断开连接等，从而保持通信逻辑的稳定性与健壮性。成功接收到数据后，线程会对消息进行协议解析，并根据消息类型调用对应的处理函数，实现私聊、广播、系统命令等不同功能的分发处理。在检测到客户端断开或发生不可恢复错误后，线程会进入资源清理阶段：在互斥锁保护下将该客户端从全局列表中安全移除，并关闭对应的套接字，释放动态分配的结构体内存，避免资源泄漏。通过这种设计，服务器不仅能够高效地管理多客户端通信，还能够借助非阻塞模式提升系统整体的响应速度和并发能力，使聊天系统运行更加稳定可靠。

Listing 7: 客户端处理线程

```
unsigned __stdcall handleClientThread(void* arg) {
    ClientInfo* client = (ClientInfo*)arg;
    char buffer[1024];
    bool connected = true;

    while (connected && serverRunning) {
        // 接收消息
        int bytesRead = recv(client->socket, buffer, sizeof(buffer) - 1, 0);

        // 错误处理和连接断开检测
        #ifdef _WIN32
        if (bytesRead == SOCKET_ERROR) {
            int errorCode = WSAGetLastError();
            if (errorCode == WSAEWOULDBLOCK) {
                // 非阻塞模式下没有数据可读，继续尝试
                Sleep(10);
                continue;
            } else {
                printError("Recv failed");
                connected = false;
                break;
            }
        } else if (bytesRead == 0) {
            printf("%s 断开连接\n", client->username);
            connected = false;
            break;
        }
        #else
        // Linux平台错误处理...
        #endif

        // 确保消息以null结尾
        buffer[bytesRead] = '\0';

        // 解析协议并处理不同类型的消息
        // ...
    }

    // 清理资源：移除客户端并关闭连接
```

```
    lockMutex();
    for (size_t i = 0; i < clients.size(); ++i) {
        if (strcmp(clients[i]->username, client->username) == 0) {
            clients.erase(clients.begin() + i);
            break;
        }
    }
    unlockMutex();

    CLOSE_SOCKET(client->socket);
    delete client;

    return 0;
}
```

3.2 客户端实现

3.2.1 核心功能模块

1. 非阻塞连接实现 该部分代码通过将客户端套接字配置为非阻塞模式，实现了更高效、更具用户体验的网络连接方式。

Listing 8: 客户端非阻塞连接

```
// 设置套接字为非阻塞模式
u_long mode = 1; // 非阻塞模式
ioctlsocket(clientSocket, FIONBIO, &mode);

// 连接服务器
if (connect(clientSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) ==
    SOCKET_ERROR) {
    int errorCode = WSAGetLastError();
    if (errorCode != WSAEWOULDBLOCK && errorCode != WSAEINPROGRESS) {
        printError("Connect failed");
        // 错误处理...
    }

    // 使用select等待连接完成，设置超时
    fd_set writeSet;
    FD_ZERO(&writeSet);
    FD_SET(clientSocket, &writeSet);

    struct timeval timeout;
    timeout.tv_sec = 5;
    timeout.tv_usec = 0;

    if (select(0, NULL, &writeSet, NULL, &timeout) <= 0) {
        printError("Connect timeout");
        // 超时处理...
    }

    // 检查连接是否真正成功
```

```
    int optval;
    int optlen = sizeof(optval);
    if (getsockopt(clientSocket, SOL_SOCKET, SO_ERROR, (char*)&optval, &optlen) < 0 ||
        optval != 0) {
        printError("Connect failed");
        // 错误处理...
    }
}
```

首先，套接字被设置为非阻塞，使得调用 connect 时不会阻塞主线程，从而避免界面卡顿或程序无响应的问题。在非阻塞模式下，connect 可能立即返回特定错误码（如 WSAEWOULDBLOCK 或 WSAEINPROGRESS），因此代码借助 select 函数对套接字的可写事件进行监控，从而实现带有 5 秒超时的连接等待机制。这不仅提升了程序的健壮性，还有效防止了由于网络不通或服务器未响应而导致的无限等待。连接尝试完成后，通过 getsockopt 查询 SO_ERROR 选项，进一步确认连接是否真正建立，避免出现“看似成功但实际失败”的情况。整体而言，这种基于非阻塞和 select 超时机制的连接方式能够显著提升客户端在弱网环境下的反应速度与错误恢复能力，相较传统阻塞式连接更加灵活可靠。

2. 消息接收线程 该线程专门负责从服务器端持续接收消息，是客户端保持实时通信能力的关键组成部分。通过将接收逻辑放入独立线程，程序能够在不阻塞用户输入的前提下同时进行消息监听与界面交互，从而提升整体的流畅性。在循环读取过程中，线程会根据 recv 的返回值判断当前网络状态：例如在非阻塞模式下，若暂时无数据可读则短暂休眠以降低 CPU 占用；若检测到连接被重置、中断或服务器主动关闭，则会向用户输出相应提示并安全退出线程，保证程序的稳定性。

成功读取到数据后，线程根据预定义的通信协议格式解析消息，通过分隔符提取消息类型、发送者名称以及具体内容，并按照广播消息、私聊消息或系统提示等不同类型采用不同的输出格式，使聊天内容更加清晰可读。同时，消息的显示通过线程安全的 printMessage 进行，避免多线程输出造成的混乱。整体而言，该线程实现了客户端的实时消息接收、稳定的异常处理与规范化显示，是客户端模块中保持通信顺畅和界面友好性的重要机制。

Listing 9: 客户端消息接收线程

```
unsigned __stdcall receiveMessagesThread(void* arg) {
    char buffer[1024];

    while (clientRunning) {
        // 接收消息
        int bytesRead = recv(clientSocket, buffer, sizeof(buffer) - 1, 0);

        // 处理各种接收情况
        #ifdef _WIN32
        if (bytesRead == SOCKET_ERROR) {
            int errorCode = WSAGetLastError();
            if (errorCode == WSAEWOULDBLOCK) {
                Sleep(10);
                continue;
            } else if (errorCode == WSAECONNABORTED || errorCode == WSAECONNRESET) {
                printMessage("连接已断开\n");
                clientRunning = false;
                break;
            } else {
                printError("Recv failed");
```

```
                clientRunning = false;
                break;
            }
        } else if (bytesRead == 0) {
            printMessage("服务器已关闭连接\n");
            clientRunning = false;
            break;
        }
        #endif

        // 确保消息以null结尾
        buffer[bytesRead] = '\0';

        // 解析协议并显示不同类型的消息
        char* firstDelim = strchr(buffer, '|');
        if (firstDelim != NULL) {
            int messageType = atoi(buffer);
            char* secondDelim = strchr(firstDelim + 1, '|');

            if (secondDelim != NULL) {
                *firstDelim = '\0';
                char* senderName = firstDelim + 1;
                *secondDelim = '\0';
                char* content = secondDelim + 1;

                // 根据消息类型显示不同格式
                switch (messageType) {
                    case 2: // 广播消息
                        printMessage("%s: %s\n", senderName, content);
                        break;
                    case 3: // 私聊消息
                        printMessage("[私聊] %s: %s\n", senderName, content);
                        break;
                    case 4: // 系统消息
                        printMessage("[系统] %s\n", content);
                        break;
                }

                // 恢复分隔符
                *firstDelim = '|';
                *secondDelim = '|';
            }
        }
    }
}
```

功能分析：

- 创建独立线程专门接收服务器消息，避免阻塞用户输入
- 处理各种网络异常情况（连接断开、重置等）
- 实现协议解析，根据消息类型以不同格式显示

- 使用线程安全的 printMessage 函数输出消息，避免显示混乱
- 非阻塞模式下适当休眠，减少 CPU 占用

3. 用户输入处理 该部分作为客户端的输入处理核心逻辑，通过循环持续接收用户在终端中输入的内容，并根据不同的输入格式对其分类处理，从而实现多种交互模式。首先，程序会检测是否输入了退出命令 /quit，若匹配则向服务器发送登出协议消息，并终止客户端运行，使用户能够安全退出会话。对于以 @ 开头的输入，程序将其视为私聊指令，通过检查是否包含合法的 @ 用户名: 消息格式来判断命令有效性；格式正确时，会将消息按照私聊协议编码发送给服务器，否则向用户输出友好的提示信息，帮助其纠正输入。对于不属于命令的普通文本，系统默认将其视为广播消息，按统一格式打包后发送到服务器交由其转发。整个过程保证了客户端操作的简洁性与可用性，使用户能够清晰地执行退出、私聊和群聊等不同操作，同时也增强了聊天系统的交互体验与输入容错能力。

Listing 10: 用户输入处理

```
while (clientRunning) {
    char message[1024];
    if (fgets(message, sizeof(message), stdin) == NULL) {
        break; // 处理输入错误
    }
    message[strcspn(message, "\n")] = '\0';  // 移除换行符

    // 命令处理逻辑
    if (strcmp(message, "/quit") == 0) {
        // 退出命令处理
        char logoutMsg[1024];
        sprintf(logoutMsg, "1|%s|", username);
        sendToServer(logoutMsg);

        clientRunning = false;
        break;
    } else if (message[0] == '@') {
        // 私聊命令处理
        char* colonPos = strchr(message, ':');
        if (colonPos != NULL && colonPos > message + 1) {
            // 格式检查通过，发送私聊消息
            char privateMsg[1024];
            sprintf(privateMsg, "3|%s|%s", username, message + 1); // 去掉@符号
            sendToServer(privateMsg);
        } else {
            printMessage("私聊格式: @用户名:消息\n");
        }
    } else {
        // 广播消息处理
        char broadcastMsg[1024];
        sprintf(broadcastMsg, "2|%s|%s", username, message);
        sendToServer(broadcastMsg);
    }
}
```

功能分析：

- 主循环处理用户输入，支持三种操作模式
- 识别退出命令（/quit），发送登出消息并关闭连接
- 识别私聊命令（@ 用户名: 消息），格式检查后发送私聊消息
- 普通消息作为广播消息发送给所有用户
- 提供友好的错误提示，帮助用户正确使用命令

3.2.2 线程安全设计

客户端实现了线程安全的输出机制，避免多线程并发输出导致的显示混乱：

Listing 11: 线程安全的消息输出

```
// 输出线程安全的消息
void printMessage(const char* format, ...) {
    va_list args;
    va_start(args, format);

#ifdef _WIN32
    EnterCriticalSection(&coutMutex);
    vprintf(format, args);
    LeaveCriticalSection(&coutMutex);
#else
    pthread_mutex_lock(&coutMutex);
    vprintf(format, args);
    pthread_mutex_unlock(&coutMutex);
#endif

    va_end(args);
}
```

该输出函数通过使用线程安全机制确保客户端在多线程环境下的输出内容始终保持整洁、有序。函数采用变参形式，使其能够处理任意格式化字符串，提高了输出接口的灵活性。在输出之前，程序会针对当前平台选择适当的互斥锁实现：Windows 环境下使用 CRITICAL_SECTION，Linux/Unix 系统则使用 POSIX 互斥锁 pthread_mutex，从而保证跨平台一致性。在加锁后进行 vprintf 输出操作，可以有效避免主输入线程和消息接收线程同时写入终端导致的内容交错或乱码问题，确保每条消息都按照正确的格式一次性输出到屏幕。通过这套设计，客户端成功实现了稳定可靠的线程安全输出机制，为终端聊天界面提供了清晰、可读的用户体验。

4 程序运行测试

4.1 运行展示

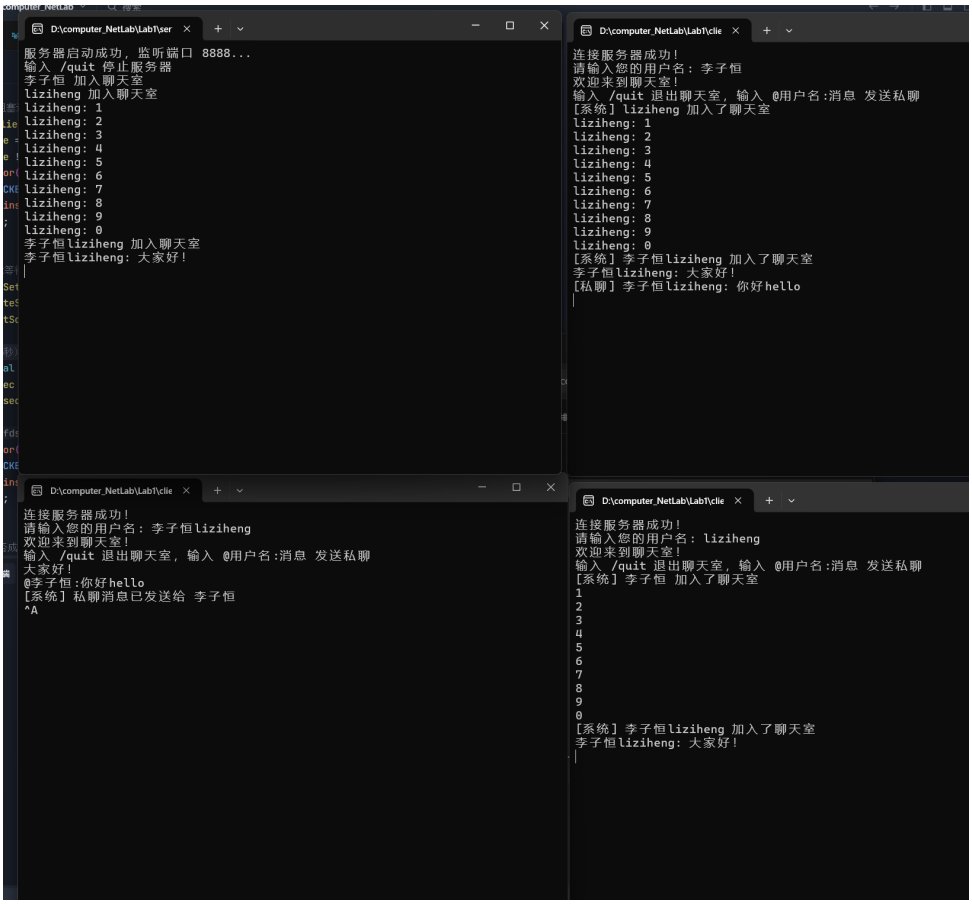

图 3: 运行展示

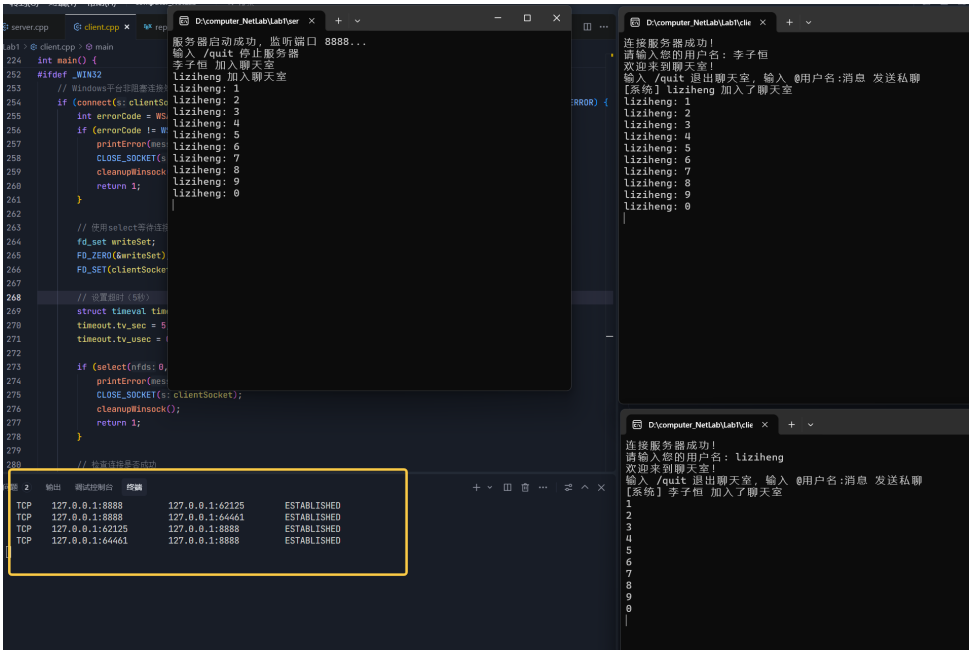

图 4: 丢包情况

4.2 丢包情况

如图 4，在 PowerShell 中运行以下命令监控 TCP 连接状态：

```
# 持续监控端口8888的连接状态
while(1) { netstat -n -p tcp | findstr "8888"; sleep 1; cls }
```

通过观察以下指标来判断可能的丢包：

- 大量的 TCP 连接建立和关闭
- 连接状态异常

5 实验总结与体会

5.1 实验过程中想到且尚未解决的问题

本次实验中其实可以改进的地方有很多，比如，结合我的专业学习（我是密码科学与技术专业的）通过网络我们可以传输秘密消息，在密码学相关课程的实验中，我实现了有关双方或多方秘密交互通信的协议，但是却无法很好地将本次实验与密码协议相互融合或者说结合。而这种扩展能力恰恰就是网络课程或者说计算机网络学习过程中需要培养的核心素养之一。毕竟我们所学习的知识最终实践出来才有意义。

其次，我没有很好地实现交互逻辑，例如如何从服务器端主动“踢出”某用户。虽然我实现了私聊的功能，但实现的很不完善。例如我在测试时就发现当用户 @ 自己时，现有逻辑会出错，发送错误信息。

5.2 实验体会

本次实验成功实现了基于流式套接字的多人聊天程序，涵盖了网络编程的核心概念和技术。通过项目的开发，我深入理解了 TCP/IP 协议的工作原理、套接字编程模型、多线程并发控制和跨平台开发技术。

主要收获包括：

1. 掌握了基于 TCP 协议的网络通信实现方法
2. 理解了服务器-客户端架构的设计与实现
3. 学会了使用多线程技术处理并发连接
4. 实现了跨平台兼容性设计
5. 设计并实现了简单高效的应用层协议
6. 培养了问题分析和解决能力

此次实验也让我深刻认识到网络编程的复杂性和挑战性，以及理论与实践相结合的重要性。在实际编程过程中，需要综合考虑性能、安全性、可靠性和用户体验等多个因素，才能开发出高质量的网络应用程序。