南開大學

计算机网络实验报告

Lab1

学院：网络空间安全学院

专业：信息安全、法学

学号：2113203

姓名：付政烨

班级：信安法班

摘要

本实验专注于流式套接字在多线程同步问题中的应用和一个基于 Socket 通信的聊天程序的设计与实现。在多线程同步问题的部分，我们探讨了如何在多线程环境下确保资源的正确访问和使用。为确保数据的完整性和一致性，实验中提出了一系列的优化方法。聊天程序基于 TCP 协议进行通讯，保证了数据的可靠性和顺序性。此程序实现了一个多客户端-服务器模型，其中服务器采用多线程（或可以选择多进程）来同时处理来自多个客户端的请求。根据提供的图像结果，当三个客户端同时发送消息时，服务器可以按照客户端的编号顺序（接收并正确显示消息。此外，服务器端引入了一个关键的功能，当操作者在服务器中输入 Ctrl+Z 命令时，服务器会立即终止，从而断开所有与之连接的客户端。尽管如此，与聊天交互相关的历史记录仍被保存，允许用户之后查阅。

关键字：socket；　流式套接字；　多线程

目录

一、 协议设计

(一) 协议基本属性

- 基于基本的 Socket 函数来实现。

```
#include<WinSock2.h>
#pragma comment(lib,"ws2_32.lib")
```

- 使用 TCP/IP 协议，采用流式套接字（SOCK_STREAM）进行通信。

```
serverSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

- 为了并发处理多个客户端，程序采用多线程方式。

```
CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE)SendToServer, NULL, NULL, NULL);
CreateThread(NULL, 0, InputThread, NULL, 0, NULL);
CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE)HandleClientMessages, (LPVOID)
    clientCount, NULL, NULL);
```

(二) 消息类型

- 连接消息：当客户端首次连接到服务器时发送。
- 文本消息：普通的聊天消息。
- 系统消息：例如，用户加入或离开聊天的通知。
- 断开消息：客户端即将断开连接时发送。

(三) 语法

每条消息都采用 UTF-8 编码。每条消息都有一个固定的最大长度（60 字节）。
例如：[2023-10-20 14:20:15] 用户 [1]: 你好！

(四) 语义

- 连接消息：用户与服务器连接成功。格式为：用户 [ID] 已加入聊天。
- 文本消息：用户之间的聊天消息。格式为：[时间戳] 用户 [用户 ID]: 消息内容。
- 断开消息：用户离开聊天室。格式为：用户 [用户 ID] 已离开聊天。
- 系统消息：服务器与客户端初始化信息、异常处理信息。

(五) 时序

- 客户端尝试连接到服务器。
- 服务器接受连接，并为该客户端分配一个唯一的用户 ID。
- 客户端可以开始发送文本消息。

- 当服务器接收到客户端的文本消息时，它将该消息添加时间戳和用户 ID，并广播给所有在线客户端。
- 客户端接收到消息后，将其显示在图形窗口中。
- 当客户端决定断开连接时，它发送一个断开消息。
- 服务器接收到断开消息后，通知其他客户端，然后关闭与该客户端的连接。

(六) 创新

- 时间戳的引入：在每条消息前加入时间戳，使得聊天更为直观。
- 用户 ID 的引入：为每个连接的客户端分配一个唯一的 ID，以便在聊天中识别不同的用户。
- 系统消息：使得聊天室的用户可以被通知有关其他用户的活动，例如用户加入或离开聊天。

二、 各模块功能

(一) 服务器

1. 初始化与常量定义

在此部分，定义了一些关键的全局常量和变量，为服务器的运行提供必要的参数。

```
    const int MAX_CLIENTS = 1024;              // 最大客户端数量
    const int BUFFER_SIZE = 60;                // 发送和接收消息的缓冲区大小
    const int FORMATTEDMSG_SIZE = 80;          // 格式化后的消息大小（包括用户ID、时
        间戳等信息）
    const char* SERVER_IP = "127.0.0.1";       // 服务器的IP地址（本机）
    const int SERVER_PORT = 9527;              // 服务器监听的端口号

    SOCKET clientSockets[MAX_CLIENTS];         // 存储客户端套接字的数组
    int clientCount = 0;                       // 当前已连接的客户端数量
    bool isRunning = true;                     // 控制服务器主循环
```

2. 客户端消息处理

这是核心功能之一。当客户端连接并发送消息时，这个函数负责处理。

```
void HandleClientMessages(int clientId) {
    int receivedBytes;
    char buffer[BUFFER_SIZE];
    char formattedMsg[FORMATTEDMSG_SIZE];

    snprintf(formattedMsg, FORMATTEDMSG_SIZE, "用户[%d] 已加入聊天。", clientId);
    for (int i = 0; i < clientCount; i++) {
        if (i != clientId) {
            send(clientSockets[i], formattedMsg, strlen(formattedMsg), NULL);
        }
    }

    while (isRunning) {
        receivedBytes = recv(clientSockets[clientId], buffer, BUFFER_SIZE - 1, NULL
            );
```

```
        if (receivedBytes > 0) {
            buffer[receivedBytes] = 0;

            auto current_time = chrono::system_clock::now();
            time_t tt = chrono::system_clock::to_time_t(current_time);
            struct tm* ptm = localtime(&tt);
            char timeString[32];
            strftime(timeString, sizeof(timeString), "%Y-%m-%d %H:%M:%S", ptm);

            memset(formattedMsg, 0, FORMATTEDMSG_SIZE);
            snprintf(formattedMsg, FORMATTEDMSG_SIZE, "[%s] 用户[%d]: %s",
                timeString, clientId, buffer);

            for (int i = 0; i < clientCount; i++) {
                send(clientSockets[i], formattedMsg, strlen(formattedMsg), NULL);
            }
        } else if (receivedBytes == 0 || receivedBytes == SOCKET_ERROR) {
            snprintf(formattedMsg, FORMATTEDMSG_SIZE, "用户[%d] 已离开聊天",
                clientId);
            for (int i = 0; i < clientCount; i++) {
                if (i != clientId) {
                    send(clientSockets[i], formattedMsg, strlen(formattedMsg), NULL
                        );
                }
            }
            cout << formattedMsg << endl;
            break;
        }
    }
}
```

- 函数首先向所有其他已连接的客户端发送新用户加入的通知。
- 然后，它在一个循环中不断地接收新消息并将它们广播给所有其他客户端。
- 如果客户端断开连接或发生通信错误，它会通知其他所有客户端这一事件。

3. 输入捕获线程

服务器需要一种方法来响应管理员的命令，特别是在要求终止时。为此，我们创建了一个单独的线程来捕获用户输入。

```
DWORD WINAPI InputThread(LPVOID param) {
    char input;
    while (true) {
        cin >> input;
        if (cin.eof()) {
            isRunning = false;
            cout << "服务端已终止" << endl;
            break;
        }
    }
    return 0;
}
```

- 当用户按下 Ctrl+Z 时，这个线程将 isRunning 设置为 false，导致服务器终止。

4. 主函数

1）初始化 WinSock

```
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
    {
        cout << "初始化WinSock失败！错误码：" << WSAGetLastError() << endl;
        return -1;
    }
    cout << "初始化WinSock成功！\n";
```

- 这部分的代码是开始使用 WinSock 库之前的必要步骤。WSAStartup 函数初始化 Windows Sockets API，必须在使用其他 WinSock 函数之前调用它。

2）创建服务器套接字

```
    SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (serverSocket == SOCKET_ERROR) {
        cout << "创建Socket失败！错误码：" << WSAGetLastError() << endl;
        WSACleanup();
        return -1;
    }
    cout << "创建Socket成功！\n";
```

- 使用 socket 函数创建了一个 TCP 套接字。如果创建失败，程序会打印错误信息并退出。

3）绑定套接字

```
    SOCKADDR_IN serverAddr = { 0 };
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.S_un.S_addr = inet_addr(SERVER_IP);
    serverAddr.sin_port = htons(SERVER_PORT);

    int Flag = bind(serverSocket, (sockaddr*)&serverAddr, sizeof serverAddr);
    if (Flag == -1) {
        cout << "绑定失败！错误码：" << WSAGetLastError() << endl;
        closesocket(serverSocket);
        WSACleanup();
        return -1;
    }
    cout << "绑定成功！\n";
```

- 这里定义了服务器的 IP 和端口，并将它们绑定到我们之前创建的套接字。这样，服务器就可以在指定的地址和端口上监听客户端的连接。

4）开始监听

```
    Flag = listen(serverSocket, 10);
    if (Flag == -1) {
        cout << "监听失败！错误码：" << WSAGetLastError() << endl;
        closesocket(serverSocket);
        WSACleanup();
        return -1;
    }
    cout << "开始监听客户端连接...（按下Ctrl+Z退出程序）\n";
```

- 使用 listen 函数开始监听客户端的连接请求。10 是待处理的最大连接请求的数量。

5）创建用户输入线程

```
CreateThread(NULL, 0, InputThread, NULL, 0, NULL);
```

- 为了能够在服务器运行时捕获用户输入（终止命令），这里创建了一个新线程来执行 InputThread 函数。
- 接受并处理客户端连接

6）接受并处理客户端连接

```
while (isRunning && clientCount < MAX_CLIENTS) {
    SOCKADDR_IN clientAddress = { 0 };
    int len = sizeof(clientAddress);
    clientSockets[clientCount] = accept(serverSocket, (sockaddr*)&
        clientAddress, &len);

    if (clientSockets[clientCount] == SOCKET_ERROR) {
        cout << "接受客户端连接失败！错误码：" << WSAGetLastError() << endl;
        continue;
    }

    char* clientIP = inet_ntoa(clientAddress.sin_addr);
    cout << "接受到来自 " << clientIP << " 的连接请求" << endl;
    cout << "用户[" << clientCount << "] 已加入聊天\n";

    CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE)HandleClientMessages, (
        LPVOID)clientCount, NULL, NULL);

    clientCount++;
}
```

- 在这个循环中，服务器持续检查新的客户端连接请求。当有客户端尝试连接时，服务器会接受这个连接，并为该客户端创建一个新的处理线程。

7）关闭所有连接并清理

```
for (int i = 0; i < clientCount; i++) {
    closesocket(clientSockets[i]);
}
closesocket(serverSocket);
WSACleanup();
```

- 当服务器终止或达到其最大客户端连接限制时，它会关闭所有客户端和服务器套接字，并使用 WSACleanup 函数清理 WinSock。

(二) 客户端

1. 初始化与常量定义

```
const char* SERVER_IP = "127.0.0.1";  // 服务器的IP地址。
const int SERVER_PORT = 9527;         // 服务器端口号。
const int BUFFER_SIZE = 60;           // 接收和发送数据的缓冲区大小。

SOCKET serverSocket;                  // SOCKET用于与服务器建立和维护连接。
bool isRunning = true;                // 控制程序的运行状态
```

2. SendToServer 函数

```
void SendToServer() {
    char buffer[BUFFER_SIZE];
    while (isRunning)
    {
        cout << ">> ";
        cin >> buffer;
        if (cin.eof()) {
            isRunning = false;
            cout << "程序已终止." << endl;
            break;
        }
        send(serverSocket, buffer, strlen(buffer), 0);
    }
}
```

- 这是一个循环函数，它持续获取用户的输入并将其发送到服务器。

3. 主函数

1）初始化聊天室界面

```
initgraph(300, 400, SHOWCONSOLE);
int displayPosition = 0;
HWND hWnd = GetHWnd();
SetWindowText(hWnd, TEXT("chat"));
```

2）初始化 WinSock

```
WSADATA wsaData;
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
{
    cout << "初始化WinSock失败！ 错误码： " << WSAGetLastError() << endl;
    return -1;
}
cout << "初始化WinSock成功！ \n";
```

- 在 Windows 上使用 sockets 之前，需要使用 WSAStartup 进行网络库的初始化。这里选择了 2.2 版本的 Winsock。

3）创建客户端套接字

```
SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (serverSocket == SOCKET_ERROR) {
    cout << "创建Socket失败！ 错误码： " << WSAGetLastError() << endl;
```

```
        WSACleanup();
        return -1;
    }
    cout << "创建Socket成功！\n";
```

- AF_INET: 表示这是一个 IPv4 地址类型的 socket。
- SOCK_STREAM: 表示这是一个流式 socket，通常用于 TCP。
- IPPROTO_TCP: 表示使用 TCP 协议。

4）指定服务器地址

```
    SOCKADDR_IN serverAddr = { 0 };
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.S_un.S_addr = inet_addr(SERVER_IP);
    serverAddr.sin_port = htons(SERVER_PORT);
```

- 这部分代码设置了服务器的 IP 地址和端口号，准备与其建立连接。

5）连接到服务器

```
     int Flag = connect(serverSocket, (sockaddr*)&serverAddress, sizeof
         serverAddress);
     if (Flag == -1) {
         cout << "连接服务器失败！错误码：" << GetLastError() << endl;
         closesocket(serverSocket);
         WSACleanup();
         return -1;
     }
     cout << "连接服务器成功！（按下Ctrl+Z退出程序）\n";
```

- 使用 connect 函数尝试连接到服务器。如果连接失败，程序会打印一个错误消息并退出。

6）创建发送数据的线程

```
    CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE)SendToServer, NULL, NULL, NULL
        );
```

- 使用 CreateThread 函数创建一个新线程，允许用户同时发送和接收消息。

7）接收数据

```
    char recvBuffer[BUFFER_SIZE];
    while (isRunning)
    {
        Flag = recv(serverSocket, recvBuffer, BUFFER_SIZE - 1, NULL);
        if (Flag > 0) {
            recvBuffer[Flag] = 0;
            outtextxy(1, displayPosition * 20, recvBuffer);
            displayPosition++;
        }
        else if (Flag == 0 || Flag == SOCKET_ERROR) {
            cout << "与服务器的连接已断开." << endl;
            isRunning = false;
```

```
            break;
        }
    }
```

- 主线程进入循环，持续从服务器接收数据。收到的数据会被显示在图形界面上。

8）结束程序

```
    closesocket(serverSocket);
    WSACleanup();
```

- 在接收数据的过程中，如果从服务器收到的数据量为 0（通常表示服务器关闭了连接）或发生错误，程序会停止运行，并关闭与服务器的连接。
- 最后，WSACleanup 被调用以清理网络库。

三、 程序界面展示及运行说明

(一) 聊天室界面

聊天室界面显示了一个服务器与三个客户端之间的简单通讯过程。每个客户端都连接到服务器，发送消息，然后得到服务器的反馈。服务器窗口和聊天窗口都为我们提供了这个过程的详细信息。

图 1

(二) 用户离开聊天室（退出方式 1）

当用户关闭客户端窗口时，意味着用户离开聊天。在服务器窗口中，显示：“用户 [2] 已离开聊天”，同时，在聊天窗口中，也显示了一个与此相关的信息，这意味着编号为 2 的客户端已经断开了与聊天室的连接。这里提供了一个简单的反馈机制，使得其他在聊天室中的用户和服务器都能知道哪个客户端已经离开。

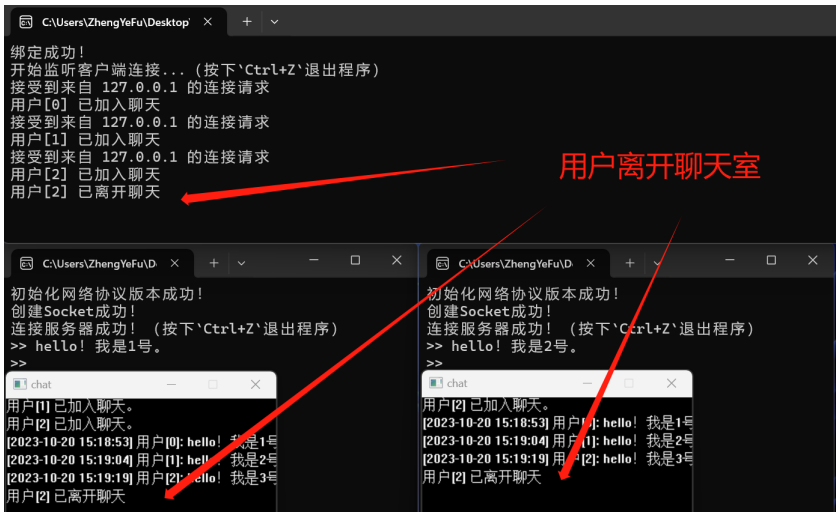

图 2

(三) 客户端程序终止（退出方式 2）

在用户界面中，当用户按下 ‘Ctrl+Z‘组合键时，会导致对话的终止。此后，输入窗口将不再接受任何新的输入数据。尽管如此，用户仍然可以浏览与之前的聊天交互相关的历史记录。

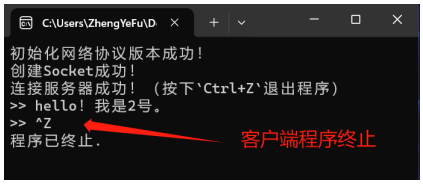

图 3

(四) 服务器关闭（退出方式 3）

在服务器环境中，当输入 ‘Ctrl+Z‘指令时，该服务器实例将被终止，导致所有与之相关的连接被断开，进而使得聊天程序停止运行。尽管如此，与聊天交互相关的历史纪录仍然保留并可供查阅。

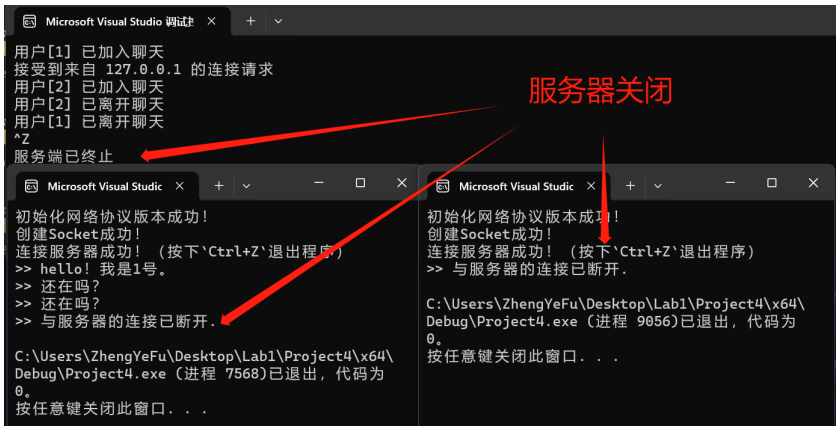

图 4

四、 实验过程中遇到的问题及分析

(一) 多线程处理问题

- 每当有一个新的客户端连接到服务器时，服务器都会创建一个新的线程来处理来自该客户端的消息。

```
CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE)HandleClientMessages, (LPVOID)
    clientCount, NULL, NULL);
```

- 这种多线程方法允许服务器同时与多个客户端交互，而不会由于单个客户端的阻塞或延迟而影响其他客户端的体验。

(二) 退出机制问题

- 服务器有一个单独的线程（通过 InputThread 函数实现）来检测服务器管理员的输入。

```
CreateThread(NULL, 0, InputThread, NULL, 0, NULL);
```

- 当管理员在控制台按下 Ctrl+Z 时，这个线程会设置全局变量 isRunning 为 false。

```
if (cin.eof()) {
    isRunning = false;
    cout << "服务端已终止" << endl;
    break;
}
```

- isRunning 变量用于控制服务器的主监听循环。

```
while (isRunning && clientCount < MAX_CLIENTS) {
    // ...
}
```

(三) 向所有客户端发送信息问题

- 当服务器从一个客户端接收到消息时，它会在 HandleClientMessages 函数中格式化该消息，加上时间戳和发送客户端的 ID。

```
sprintf(formattedMsg, FORMATTEDMSG_SIZE, "[%s] 用户[%d]: %s", timeString,
    clientId, buffer);
```

- 服务器随后遍历 clientSockets 数组，使用 send 函数将格式化的消息发送给每一个已连接的客户端。

```
for (int i = 0; i < clientCount; i++) {
    send(clientSockets[i], formattedMsg, strlen(formattedMsg), NULL);
}
```

（四） 线程同步问题

问题描述

- 当多个客户端同时发送信息给服务器时，可能会发生线程同步问题。比如，当两个或更多线程同时访问和修改 clientCount 或者 clientSockets 数组时，可能会导致数据不一致的情况。

优化思路

1. 使用互斥锁：互斥锁是最常用的线程同步机制之一。当一个线程拥有互斥锁时，其他线程不能拥有它，从而确保同一时间只有一个线程可以访问或修改受锁保护的资源。对于上述的 clientCount 和 clientSockets 数组，可以定义一个互斥锁。每次需要修改或访问这些资源时，首先锁定该互斥锁，然后进行操作，操作完成后再解锁。

2. 使用临界区：临界区是 Windows 特定的线程同步机制，与互斥锁类似，但更轻量级，并且只能在单一进程的线程之间使用。使用临界区可以为受其保护的资源提供一个锁。与互斥锁相比，临界区在同一进程内的线程同步时通常更快。

3. 避免死锁：死锁是当两个或更多的线程都在等待另一个线程释放资源，从而无法继续执行的情况。如果程序使用多个锁，那么需要确保所有线程总是以相同的顺序请求和释放锁，这是避免死锁的一种常见策略。

互斥锁实现

1. 定义一个全局互斥锁

```
        SOCKET serverSocket;
```

2. 初始化互斥锁

```
        SOCKET serverSocket;
```

3. 互斥锁的获取与释放

```
        while (isRunning && clientCount < MAX_CLIENTS) {
            SOCKADDR_IN clientAddress = { 0 };
            int len = sizeof(clientAddress);
            SOCKET acceptedSocket = accept(serverSocket, (sockaddr*)&clientAddress, &
                len);

            if (acceptedSocket == SOCKET_ERROR) {
                cout << "接受客户端连接失败！错误码：" << WSAGetLastError() << endl;
                continue;
            }

            WaitForSingleObject(clientMutex, INFINITE); // 获取互斥锁

            clientSockets[clientCount] = acceptedSocket;
            char* clientIP = inet_ntoa(clientAddress.sin_addr);
            cout << "接受到来自 " << clientIP << " 的连接请求" << endl;
            cout << "用户[" << clientCount << "] 已加入聊天\n";
            clientCount++;

```

```
19              ReleaseMutex(clientMutex); // 释放当前线程拥有的互斥锁
20
21              CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE)HandleClientMessages, (
                    LPVOID)(clientCount - 1), NULL, NULL);
22          }
```

参考文献