

计算机网络实验报告

Lab3-2 基于UDP服务设计可靠传输协议并编程实现

网络空间安全学院 物联网工程专业 2212039 田晋宇

[jassary08/Computer_Network \(github.com\)](https://github.com/jassary08/Computer_Network)

实验要求

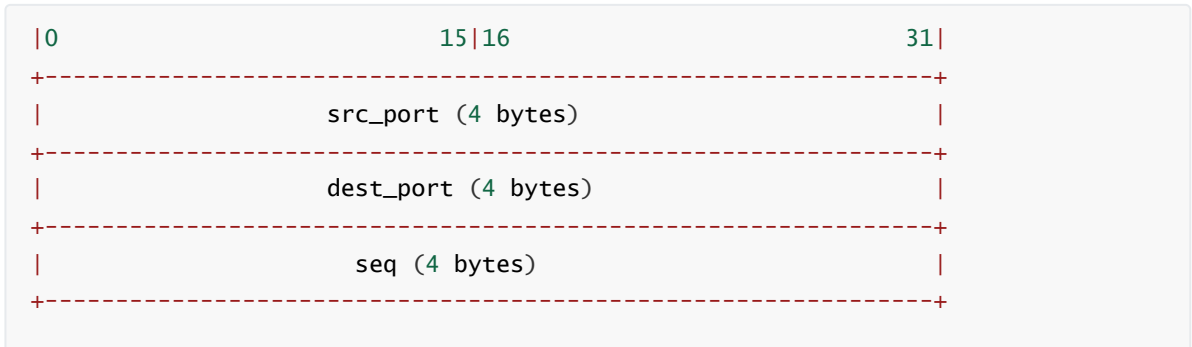
利用数据报套接字在用户空间实现面向连接的可靠数据传输，功能包括：建立连接、差错检测、接收确认、超时重传等。采用基于滑动窗口的流量控制机制，接收窗口大小为 1，发送窗口大小大于 1，支持累积确认，完成给定测试文件的传输。

一. 设计思想

1.协议设计

本次实验我仿照了TCP协议对**UDP 协议**的报文格式进行了设计，以支持可靠传输和连接管理。报文头部包含多个字段：`src_port` 和 `dest_port` 用于标识通信的源端口和目标端口，`seq` 和 `ack` 分别表示序列号和确认号，用于控制数据的发送和接收顺序，确保可靠性。`length` 表示整个数据报的长度，`check` 用于存储校验和，验证数据完整性。`flag` 字段定义了一组标志位，用于连接建立（SYN）、终止（FIN）、确认（ACK）以及自定义功能（CFH）。`reserved` 提供了扩展空间。数据部分最大支持 1024 字节，用于传输实际的应用数据。

字段	大小 (字节)	描述
<code>src_port</code>	4	源端口号
<code>dest_port</code>	4	目标端口号
<code>seq</code>	4	序列号，用于顺序控制
<code>ack</code>	4	确认号，用于确认接收数据
<code>length</code>	4	数据包长度
<code>check</code>	2	校验和
<code>flag</code>	2	标志位，控制协议状态
<code>data</code>	1024	数据部分





2.消息传输机制

本实验中使用的**ACK** 的值直接**等于**发送数据包的**序列号**，使用这种非累加的确认机制，更适合 UDP 等无连接协议，能够明确地告诉发送端接收端收到了哪些包，提高了可靠性，适合对传输完整性要求较高的实验和场景（如文件传输、实时通信等）。

• 三次握手——建立连接

改进后的协议建立连接的流程如下：

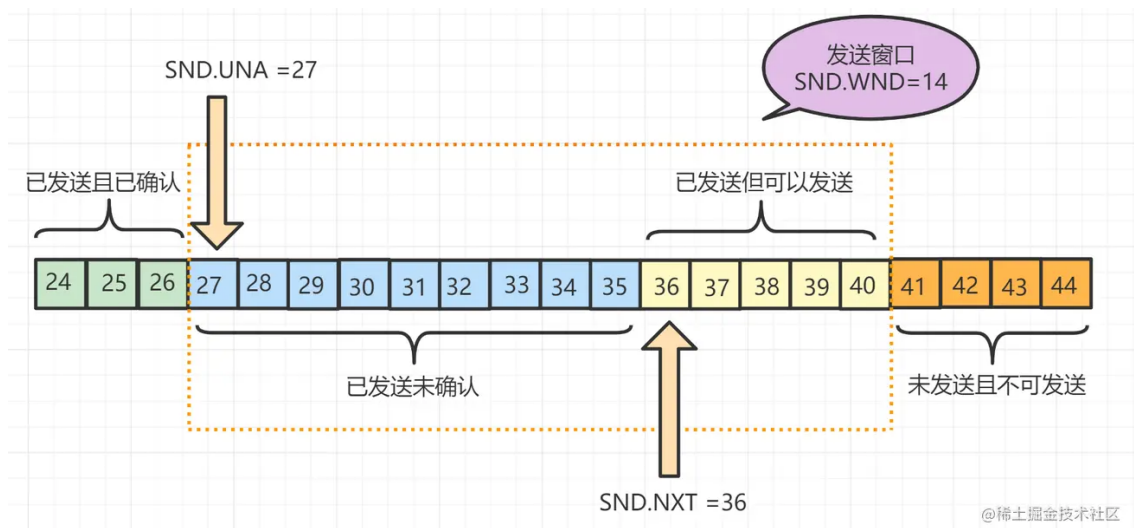
步骤	发送方	接收方
第一次握手	发送 SYN, Seq = X	
第二次握手		接收并发送 SYN, ACK = X, Seq = Y
第三次握手	接收并发送 ACK = Y	

• 差错检查机制——校验和

实验设计的**校验和机制**的核心思想是通过累加数据字段的值来形成一个**校验和**，并在传输数据时携带该校验和。当接收端收到数据时，重新计算数据的校验和，并与收到的校验和对比：

- 如果两者一致，说明数据未被破坏。
- 如果不一致，说明数据在传输过程中发生了错误。

• 流量控制机制——滑动窗口机制



本实验使用的可靠性传输机制是**GoBackN滑动窗口**，这是一种在网络数据传输中实现流量控制的机制。它通过限制发送方在接收到确认前发送数据的最大量,来避免网络拥塞。窗口大小会根据网络状况动态调整,因此被称为"滑动"窗口。

滑动窗口维护了三个关键变量:

1. **窗口大小(Windows_Size):** 表示在收到确认前,发送方最多可以发送的数据量。这个值初始为1,之后会根据接收方的反馈动态调整。
2. **窗口base(Base_Seq):** 指向当前窗口的第一个分组的序列号。只有收到了base对应分组的ACK确认,窗口才会向前滑动。
3. **下一个待发送的序列号(Next_Seq):** 它等于base加上当前已发送但未确认的分组数。

发送方在给定时刻,只能发送序列号在 **[base, base+Windows_Size)** 范围内的分组。当收到了**base**对应的**ACK**时,窗口就会向前滑动,**base**增加1。这样,就允许发送更多的新数据。

如果在一定时间内没收到**ACK**,发送方会重发窗口内未确认的分组。如果收到了重复的**ACK**,发送方也会重发对应的分组。这提供了一定程度的可靠传输。

同时,通过动态调整**Windows_Size**的大小,滑动窗口可以很好地适应网络状况。当网络状况良好时,窗口会增大,允许更多的数据同时发送。当网络拥塞时,窗口会减小,限制发送速率。

• **重传机制——超时重传和快速重传机制**

快速重传机制是基于接收方返回的重复ACK来进行数据包重传的方法。它不需要等待超时时间就能触发重传，从而能更快地恢复因丢包导致的传输中断。在实现中，发送方会维护一个重复ACK的计数器，当收到三个相同序号的重复ACK时，就认为该序号之后的数据包可能已经丢失，此时会立即重传从丢失包开始的所有未确认数据包。这种机制的主要优势在于能够快速响应网络中的丢包情况，大大减少了等待超时的时间浪费，提高了网络的吞吐量和传输效率。

超时重传机制是通过设置超时计时器来监控数据包传输，当在规定时间内未收到确认就进行重传的基本可靠传输保障机制。它需要合理设置超时阈值(RTO)，如果设置太短会导致不必要的重传，设置太长则会影响传输效率。一般来说，超时时间会基于网络的RTT(往返时间)来设置。在具体实现时，发送方会记录每个数据包的发送时间，并周期性检查是否超时，一旦发现超时就会重新发送相应的数据包，同时可能触发拥塞控制机制。

这两种重传机制在实际应用中往往是**配合使用**的。快速重传机制响应更快，不需要等待超时周期就能发现并处理丢包情况，特别适合网络质量较好的场景。而超时重传机制虽然反应较慢，但是作为最后的保障手段必不可少，可以处理那些无法通过快速重传机制检测到的丢包情况。在资源消耗方面，快速重传需要维护计数器但网络开销较小，超时重传则需要定时器机制但实现相对简单。

• **四次挥手——断开连接**

四次挥手的过程与三次握手类似，发送的确认号为对方发来的序列号。

步骤	发送方	接收方
第一次挥手	发送 FIN, Seq = U	
第二次挥手		接收并发送 ACK, Ack = U
第三次挥手		发送 FIN, Seq = V
第四次挥手	接收并发送 ACK, Ack = V	

二. 代码实现

1. 协议设计及宏定义

我们在头文件 `udp_packet.h` 中定义了UDP数据包的结构体，以及一些**宏定义常量和函数**。在 `udp_packet.cpp` 中给出了宏定义函数的具体实现。

具体各成员变量和功能在上一章节已经阐述，此处不再过多赘述。

```
#ifndef UDP_PACKET_H
#define UDP_PACKET_H

#include <iostream>
#include <bitset>
#include <stdint>
#include <cstring>
#include <winsock2.h>
#include <fstream>
#include <thread>
#include <ws2tcpip.h>
#include <chrono>
using namespace std;

#define MAX_DATA_SIZE 1024 // 数据部分的最大大小

#define TIMEOUT 1000 // 超时时间（毫秒）

#define CLIENT_PORT 54321 // 客户端端口
#define ROUTER_PORT 12345 // 目标路由端口
#define CLIENT_IP "127.0.0.1" // 目标路由 IP 地址
#define ROUTER_IP "127.0.0.1" // 目标路由 IP 地址

// UDP 数据报结构
struct UDP_Packet {
    uint32_t src_port; // 源端口
    uint32_t dest_port; // 目标端口
    uint32_t seq; // 序列号
    uint32_t ack; // 确认号
    uint32_t length; // 数据长度（包括头部和数据）
    uint16_t flag; // 标志位
    uint16_t check; // 校验和
    char data[MAX_DATA_SIZE]; // 数据部分

    // 标志位掩码
    static constexpr uint16_t FLAG_FIN = 0x8000; // FIN 位
    static constexpr uint16_t FLAG_CFH = 0x4000; // CFH 位
    static constexpr uint16_t FLAG_ACK = 0x2000; // ACK 位
    static constexpr uint16_t FLAG_SYN = 0x1000; // SYN 位

    UDP_Packet() : src_port(0), dest_port(0), seq(0), ack(0), length(0),
flag(0), check(0) {
        memset(data, 0, MAX_DATA_SIZE); // 将数据部分初始化为 0
    }

    // 设置标志位
```

```

void Set_CFH();
bool Is_CFH() const;

void Set_ACK();
bool Is_ACK() const;

void Set_SYN();
bool Is_SYN() const;

void Set_FIN();
bool Is_FIN() const;

// 校验和计算
uint16_t Calculate_Checksum() const;

// 校验和验证
bool CheckValid() const;

// 打印消息
void Print_Message() const;
};

#endif

```

在 `udp_packet.cpp` 中为 `CFH`、`ACK`、`SYN` 和 `FIN` 四种标志位提供了设置和检查方法，使用按位或操作 (`|=`) 来设置特定标志位，按位与操作 (`&`) 检查某一位是否被设置。

```

// 以ACK位的设置和检查为例
void UDP_Packet::Set_ACK() {
    flag |= FLAG_ACK;
}

bool UDP_Packet::Is_ACK() const {
    return (flag & FLAG_ACK) != 0;
}

```

`Calculate_Checksum` 方法负责对包头和数据部分计算校验和，用于保证数据完整性：

- 在计算校验和前，验证 `this` 和 `data` 指针，避免空指针导致的错误。
- 将包头字段逐一累加到 `sum`。
- 按双字节读取数据部分，每次读取两个字节形成一个 16 位的 `word`，并累加。
- 将 32 位的累加和中的高 16 位进位加回到低 16 位，直到 `sum` 只剩下 16 位。
- 返回 `sum` 的按位取反值作为校验和。

```

uint16_t UDP_Packet::Calculate_Checksum() const {
    // 验证 this 和 data 的有效性
    if (this == nullptr) {
        cerr << "[错误] this 指针为空，无法计算校验和。" << endl;
        return 0;
    }
    if (data == nullptr) {
        cerr << "[错误] 数据指针无效，无法计算校验和。" << endl;
        return 0;
    }
}

```

```

uint32_t sum = 0;

// 累加 UDP 头部
sum += src_port;
sum += dest_port;
sum += (seq >> 16) & 0xFFFF;
sum += seq & 0xFFFF;
sum += (ack >> 16) & 0xFFFF;
sum += ack & 0xFFFF;
sum += length;

// 累加数据部分, 确保范围合法
for (size_t i = 0; i < MAX_DATA_SIZE - 1 && i + 1 < length; i += 2) {
    uint16_t word = (data[i] << 8) | (data[i + 1] & 0xFF);
    sum += word;
}

// 将进位加回低 16 位
while (sum >> 16) {
    sum = (sum & 0xFFFF) + (sum >> 16);
}

return ~sum & 0xFFFF;
}

```

`checkValid` 方法负责验证接收包的校验和是否正确,通过比较包内的 `check` 值和 `calculateChecksum()` 的结果判断包是否完整。

```

bool UDP_Packet::checkValid() const {
    return (check & 0xFFFF) == calculateChecksum();
}

```

2. 套接字初始化

发送端和接收端的部分我们分别在 `client_send.cpp` 和 `server_receive.cpp` 中实现。我通过 `UDPClient` 类和 `UDPServer` 类进行封装,类内实现了**初始化**, **建立连接**, **传输文件**和**断开连接**等功能。

以**发送端**为例, `SOCKET clientSocket` 用于存储客户端的 UDP 套接字。 `sockaddr_in clientAddr` 保存客户端的地址信息,包括 IP 和端口。保存目标路由的地址信息。 `uint32_t seq` 客户端的当前序列号,用于标记数据包的顺序,在发送每个数据包时递增,确保可靠性。

```

class UDPClient {
private:
    SOCKET clientSocket;           // 客户端套接字
    sockaddr_in clientAddr;        // 客户端地址
    sockaddr_in routerAddr;        // 目标路由地址
    uint32_t seq;                  // 客户端当前序列号

public:
    UDPClient() : clientSocket(INVALID_SOCKET), seq(0) {}
    bool init() {}
}

```

```

bool connect() {}
bool Send_Message(const string& file_path) {}
bool Disconnect() {}
~UDPClient() {}
};

```

在**初始化**客户端UDP套接字的过程中，首先初始化**winsock库**，并检查版本是否与指定版本匹配，接着初始化**UDP套接字**，使用IPv4地址族，并将套接字设置为**非阻塞模式**，接着将**客户端地址**绑定在套接字上，最后配置好**目标路由地址**，初始化的过程完成。

```

bool init() {
    // 初始化 winsock
    WSADATA wsaData;
    int result = WSASStartup(MAKEWORD(2, 2), &wsaData);
    if (result != 0) {
        cerr << "[错误] WSASStartup 失败，错误代码: " << result << endl;
        return false;
    }

    // 检查版本是否匹配
    if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2) {
        cerr << "[错误] 不支持的 winSock 版本。" << endl;
        WSACleanup();
        return false;
    }

    // 创建 UDP 套接字
    clientSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if (clientSocket == INVALID_SOCKET) {
        cerr << "[错误] 套接字创建失败，错误代码: " << WSAGetLastError() << endl;
        WSACleanup();
        return false;
    }

    cout << "[日志] 套接字创建成功。" << endl;

    // 设置非阻塞模式
    u_long mode = 1;
    if (ioctlsocket(clientSocket, FIONBIO, &mode) != 0) {
        cerr << "[错误] 设置非阻塞模式失败，错误代码: " << WSAGetLastError() <<
endl;

        closesocket(clientSocket);
        WSACleanup();
        return false;
    }

    cout << "[日志] 套接字设置为非阻塞模式。" << endl;

    // 配置客户端地址
    memset(&clientAddr, 0, sizeof(clientAddr));
    clientAddr.sin_family = AF_INET;
    clientAddr.sin_port = htons(CLIENT_PORT);
    inet_pton(AF_INET, CLIENT_IP, &clientAddr.sin_addr);

    // 绑定客户端地址到套接字

```

```

        if (bind(clientSocket, (sockaddr*)&clientAddr, sizeof(clientAddr)) ==
SOCKET_ERROR) {
            cerr << "[错误] 套接字绑定失败, 错误代码: " << WSAGetLastError() << endl;
            closesocket(clientSocket);
            WSACleanup();
            return false;
        }

        cout << "[日志] 套接字绑定到本地地址: 端口 " << CLIENT_PORT << endl;

        // 配置目标路由地址
        memset(&routerAddr, 0, sizeof(routerAddr));
        routerAddr.sin_family = AF_INET;
        routerAddr.sin_port = htons(ROUTER_PORT);
        inet_pton(AF_INET, ROUTER_IP, &routerAddr.sin_addr);

        return true;
    }

```

3. 三次握手——建立连接

发送端

`connect()` 方法实现了通过 UDP 协议的三次握手过程，此处仿照 TCP 连接的建立。三次握手流程确保了发送方和接收方的同步，保证双方连接建立成功。同样以发送端为例：

1. 第一次握手

- 初始化一个 `UDP_Packet`，设置以下字段：
 - 源端口和目标端口。
 - 设置 `SYN` 标志位，表示这是一个连接请求包。
 - 设置序列号 `seq`，并计算校验和。
- 发送 `SYN` 数据包。
- 记录发送时间 `msg1_Send_Time`，用于后续超时重传判断。

2. 第二次握手

- 循环等待接收服务端返回的 `SYN + ACK` 包：
 - 验证标志位、校验和和确认号是否正确。
 - 如果验证通过，退出循环，表示第二次握手成功。
- 如果超时未收到正确的响应包：
 - 重新发送第一次握手的 `SYN` 包，并更新发送时间。

3. 第三次握手

- 初始化一个新的 `UDP_Packet`：
 - 设置序列号为 `seq + 1`。
 - 确认号 `ack` 设置为服务端的序列号 `con_msg[1].seq`。
 - 设置 `ACK` 标志位，表示确认服务端的同步。
 - 发送第三次握手的 `ACK` 包。

```

bool connect() {

```



```

UDP_Packet con_msg[3]; // 三次握手消息

// 第一次握手
con_msg[0] = {}; // 清空结构体
con_msg[0].src_port = CLIENT_PORT;
con_msg[0].dest_port = ROUTER_PORT;
con_msg[0].Set_SYN(); // 设置 SYN 标志位
con_msg[0].seq = ++seq; // 设置序列号
con_msg[0].check = con_msg[0].Calculate_Checksum(); // 计算校验和
auto msg1_Send_Time = chrono::steady_clock::now(); // 记录发送时间

cout << "[日志] 第一次握手: 发送 SYN..." << endl;
if (sendto(clientSocket, (char*)&con_msg[0], sizeof(con_msg[0]), 0,
    (sockaddr*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR) {
    cerr << "[错误] 第一次握手消息发送失败。" << endl;
    return false;
}

// 第二次握手
socklen_t addr_len = sizeof(routerAddr);
while (true) {
    // 接收 SYN+ACK 消息
    if (recvfrom(clientSocket, (char*)&con_msg[1], sizeof(con_msg[1]),
0,
        (sockaddr*)&routerAddr, &addr_len) > 0) {
        if (con_msg[1].Is_ACK() && con_msg[1].Is_SYN() &&
con_msg[1].CheckValid() &&
            con_msg[1].ack == con_msg[0].seq) {
            cout << "[日志] 第二次握手成功: 收到 SYN+ACK。" << endl;
            break;
        }
        else {
            cerr << "[错误] 第二次握手消息验证失败。" << endl;
        }
    }

    // 超时重传第一次握手消息
    auto now = chrono::steady_clock::now();
    if (chrono::duration_cast<chrono::milliseconds>(now -
msg1_Send_Time).count() > TIMEOUT) {
        cout << "[日志] 超时, 重传第一次握手消息。" << endl;
        con_msg[0].check = con_msg[0].Calculate_Checksum(); // 重新计算校验
和
        if (sendto(clientSocket, (char*)&con_msg[0], sizeof(con_msg[0]),
0,
            (sockaddr*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR)
{
            cerr << "[错误] 重传失败。" << endl;
            return false;
        }
        msg1_Send_Time = now; // 更新发送时间
    }
}
seq = con_msg[1].seq;
// 第三次握手

```

```

con_msg[2] = {}; // 清空结构体
con_msg[2].src_port = CLIENT_PORT;
con_msg[2].dest_port = ROUTER_PORT;
con_msg[2].seq = seq + 1; // 设置序列号
con_msg[2].ack = con_msg[1].seq; // 设置确认号
con_msg[2].Set_ACK(); // 设置 ACK 标志位
con_msg[2].check = con_msg[2].Calculate_Checksum(); // 计算校验和
cout << "[日志] 第三次握手: 发送 ACK..." << endl;
if (sendto(clientSocket, (char*)&con_msg[2], sizeof(con_msg[2]), 0,
(sockaddr*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR) {
    cerr << "[错误] 第三次握手消息发送失败。" << endl;
    return false;
}

cout << "[日志] 三次握手完成, 连接建立成功。" << endl;
return true;
}

```

服务器端

服务器端建立连接的过程与客户端相对应，首先接收来自客户端的 SYN 消息，接收后对消息进行校验，发送 SYN + ACK 消息，最后等待接收客户端的第三次握手消息。

代码逻辑与客户端相似，此处不再展示。

4. 发送数据包

客户端

在本次实验中，由于滑动窗口的实现较为复杂，因此我将发送数据包所用的功能用函数封装起来，最终在 Send_Message 函数中调用，大大提高了代码的可读性。

为了实现多线程的滑动窗口算法，增加了一系列必要的变量：

```

//多线程变量定义
atomic_int Base_Seq(1);
atomic_int Next_Seq(1);
atomic_int Header_Seq(0);
atomic_int Count(0);
atomic_bool Resend(false);
atomic_bool Over(false);
mutex mtx;

```

- **Base_Seq**:表示滑动窗口的基序列号，初始值为1。原子操作确保在多线程环境下对该变量的访问是线程安全的。
- **Next_Seq**:表示下一个要发送的消息的序列号，初始值为1。
- **Header_Seq**:表示消息头中的序列号，初始值为0。
- **Count**:用于计数某些事件或操作的次数，初始值为0。
- **Resend**:表示是否需要重发消息，初始值为false。
- **Over**:表示通信是否结束，初始值为false。
- **mtx**:互斥量，用于在多线程环境下对共享资源进行同步访问。互斥量可以防止多个线程同时访问临界区，避免竞态条件的发生。

1. 数据包发送

客户端首先通过 `sendFileHeader` 方法发送文件头信息，该信息通常包括文件名、文件大小、以及其他必要的文件元数据。之后，客户端通过 `sendFileData` 方法分割文件数据并发送，每个数据包都会被标记一个序列号。

- **sendFileHeader 方法：**该方法将文件的基本信息封装为数据包，发送给服务器。文件头通常包括文件的名称、大小、以及文件类型等。

```
bool sendFileHeader(UDP_Packet* data_msg, const string& file_name) {
    strcpy_s(data_msg[0].data, file_name.c_str());
    data_msg[0].data[strlen(data_msg[0].data)] = '\0';
    data_msg[0].length = file_length;
    data_msg[0].seq = ++seq;
    data_msg[0].Set_CFH();
    data_msg[0].src_port = CLIENT_PORT;
    data_msg[0].dest_port = ROUTER_PORT;
    data_msg[0].check = data_msg[0].Calculate_Checksum();

    SetConsoleTextAttribute(hConsole, 11); // 浅蓝色
    cout << "[发送] 文件头信息包: " << file_name
         << " (序列号: " << data_msg[0].seq << ")" << endl;
    SetConsoleTextAttribute(hConsole, 7);

    if (sendto(clientSocket, (char*)&data_msg[0], sizeof(data_msg[0]),
0,
        (SOCKADDR*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR) {
        SetConsoleTextAttribute(hConsole, 12); // 红色
        cerr << "[错误] 文件头发送失败, 错误码: " << WSAGetLastError() <<
endl;

        SetConsoleTextAttribute(hConsole, 7);
        return false;
    }
    return true;
}
```

- **sendFileData 方法：**该方法将文件数据拆分成固定大小的块，并通过 UDP 包逐个发送。每个数据包都带有一个序列号，用于标识数据包的顺序。

```
bool sendFileData(UDP_Packet* data_msg, ifstream& file, int next_seq,
int last_length) {
    // 读取文件数据
    if (next_seq == Msg_Num && last_length) {
        file.read(data_msg[next_seq - 1].data, last_length);
        data_msg[next_seq - 1].length = last_length;
    }
    else {
        file.read(data_msg[next_seq - 1].data, MAX_DATA_SIZE);
        data_msg[next_seq - 1].length = MAX_DATA_SIZE;
    }

    // 设置数据包属性
    data_msg[next_seq - 1].seq = ++seq;
    data_msg[next_seq - 1].src_port = CLIENT_PORT;
    data_msg[next_seq - 1].dest_port = ROUTER_PORT;
}
```

```

        data_msg[next_seq - 1].check = data_msg[next_seq -
1].Calculate_Checksum();

        if (sendto(clientSocket, (char*)&data_msg[next_seq - 1],
sizeof(data_msg[next_seq - 1]), 0,
(SOCKADDR*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR) {
            SetConsoleTextAttribute(hConsole, 12);
            cerr << "[错误] 数据包发送失败, 序列号: " << data_msg[next_seq -
1].seq
                << ", 错误码: " << WSAGetLastError() << endl;
            SetConsoleTextAttribute(hConsole, 7);
            return false;
        }
        return true;
    }
}

```

- **handleResend方法**: 该方法处理那些通过快速重传机制需要重传的数据包

```

void handleResend(UDP_Packet* data_msg) {
    SetConsoleTextAttribute(hConsole, 14);
    cout << "\n[重传] 开始重传未确认的数据包..." << endl;
    SetConsoleTextAttribute(hConsole, 7);

    for (int i = 0; i < Next_Seq - Base_Seq; i++) {
        lock_guard<mutex> lock(mtx);
        int resend_seq = Base_Seq + i - 1;
        data_msg[resend_seq].check =
data_msg[resend_seq].Calculate_Checksum();

        if (sendto(clientSocket, (char*)&data_msg[resend_seq],
sizeof(data_msg[resend_seq]), 0,
(SOCKADDR*)&routerAddr, sizeof(routerAddr)) != SOCKET_ERROR)
        {
            SetConsoleTextAttribute(hConsole, 14);
            cout << "[重传] 数据包重传成功, 序列号: " << resend_seq +
Header_Seq + 1 << endl;
            SetConsoleTextAttribute(hConsole, 7);
        }
    }
    Resend = false;
}

```

2. 传输进度与统计

- **传输进度**: 客户端会打印一个传输进度条来显示当前文件传输的进度。

```

void printTransferProgress(uint64_t transferred, uint64_t total,
chrono::steady_clock::time_point start_time) {
    auto now = chrono::steady_clock::now();
    double elapsed = chrono::duration<double>(now - start_time).count();
    double speed = (transferred - total) / elapsed / 1024; // KB/s
    int percentage = (int)((transferred - total) * 100.0 / total);

    // 进度条宽度
    const int bar_width = 50;
}

```

```

int filled = bar_width * percentage / 100;

cout << "\r[进度] [";
for (int i = 0; i < bar_width; ++i) {
    if (i < filled) cout << "=";
    else if (i == filled) cout << ">";
    else cout << " ";
}
cout << "]" << percentage << "% "
    << formatFileSize(transferred - total) << "/" <<
formatFileSize(total)
    << " (" << fixed << setprecision(2) << speed << " KB/s)    " <<
flush;
cout << endl;
}

```

- **传输统计：**在文件传输完成后，输出统计数据，如传输的总时间、吞吐率等。

```

void Thread_Ack() {
    int Err_ack_Num = 0;
    int resend_threshold = 3; // 设定重复确认的重发阈值
    int resend_counter = 0; // 用于统计连续的相同 ACK

    while (true) {
        UDP_Packet ack_msg;

        // 接收ACK消息
        if (recvfrom(clientSocket, (char*)&ack_msg, sizeof(ack_msg), 0,
(SOCKADDR*)&routerAddr, &addr_len)) {
            // 确保接收到的包是有效的 ACK
            if (ack_msg.Is_ACK() && ack_msg.CheckValid()) {
                lock_guard<mutex> lock(mtx); // 加锁保护临界区

                cout << "[日志] 接收到确认消息, ACK 序列号: " << ack_msg.ack
<< endl;

                // 处理 ACK 序列号更新
                if (ack_msg.ack >= Base_Seq + Header_Seq) {
                    Base_Seq = ack_msg.ack - Header_Seq + 1;
                }

                // 检查是否完成所有消息传输
                if (ack_msg.ack - Header_Seq == Msg_Num + 1) {
                    Over = true;
                    return; // 完成传输, 退出线程
                }

                // 错误 ACK 重发控制
                if (Err_ack_Num != ack_msg.ack) {
                    Err_ack_Num = ack_msg.ack;
                    resend_counter = 0; // 重设计数器
                }
                else {
                    resend_counter++;
                    if (resend_counter >= resend_threshold) {

```

```
Resend = true; // 达到重发阈值，设置重发标志  
resend_counter = 0; // 重设计数器  
  
    }  
  
    }  
  
    }  
  
}
```

3. 子线程：ACK 消息处理

实现了一个滑动窗口机制中接收消息的**线程函数** `Thread_Ack()`，它的主要功能是接收来自发送端的确认消息 (ACK)，并根据接收到的 ACK 更新滑动窗口的状态。

1. 初始化变量:

这些变量用于实现快速重传机制

- o `Err_ack_Num`：用于记录上一次收到的错误 ACK 序列号。
- o `resend_threshoId`：设定重复确认的重发阈值，默认为 3。
- o `resend_counter`：用于统计连续收到相同 ACK 的次数。

2. 对接收到的消息进行检查，确保是**有效的 ACK 消息**。

3. 如果接收到有效的 ACK 消息, 进入**临界区** (使用互斥锁 mtx 进行保护):

- 输出接收到的 ACK 序列号的日志信息。
- 更新滑动窗口的基序列号 `Base_Seq`：如果接收到的 ACK 序列号大于等于 `Base_Seq + Header_Seq`，则将 `Base_Seq` 更新为 `ack_msg.ack - Header_Seq + 1`。
- 检查是否完成了所有消息的传输：如果接收到的 ACK 序列号减去 `Header_Seq` 等于消息总数 `Msg_Num + 1`，则说明传输完成，将 `over` 标志设置为 `true`，并退出线程函数。

4. 错误 ACK 重发控制:

- 如果接收到的 ACK 序列号与上一次收到的错误 ACK 序列号 `Err_ack_Num` 不同，则更新 `Err_ack_Num` 为当前接收到的 ACK 序列号，并将 `resend_counter` 重置为 0。
- 如果接收到的 ACK 序列号与 `Err_ack_Num` 相同，则增加 `resend_counter` 的计数。如果 `resend_counter` 达到重发阈值 `resend_threshold`，则将 `Resend` 标志设置为 `true`，表示需要重发消息，并将 `resend_counter` 重置为 0。

通过这个线程函数，接收端可以接收来自发送端的 ACK 消息，并根据接收到的 ACK 更新滑动窗口的状态。同时，它还实现了错误 ACK 的重发控制机制，当连续收到一定次数的相同 ACK 时，会触发消息的重发，以确保可靠的数据传输。

```
void Thread_Ack() {
    int Err_ack_Num = 0;
    int resend_threshold = 3; // 设定重复确认的重发阈值
    int resend_counter = 0; // 用于统计连续的相同 ACK
    while (true) {
        UDP_Packet ack_msg;
        // 接收ACK消息
        if (recvfrom(clientSocket, (char*)&ack_msg, sizeof(ack_msg), 0,
            (SOCKADDR*)&routerAddr, &addr_len)) {
            // 确保接收到的包是有效的 ACK
            if (ack_msg.Is_ACK() && ack_msg.CheckValid()) {
                lock_guard<mutex> lock(mtx); // 加锁保护临界区
                cout << "[日志] 接收到确认消息, ACK 序列号: " << ack_msg.ack <<
end]:
```



```

        if (Resend) {
            handleResend(data_msg.get());
            continue;
        }

        // 正常发送处理
        if (Next_Seq < Base_Seq + windows_Size && Next_Seq <= Msg_Num + 1) {
            lock_guard<mutex> lock(mtx);

            if (Next_Seq == 1) {
                // 发送文件头
                if (!sendFileHeader(data_msg.get(), file_name)) {
                    return false;
                }
            }
            else {
                // 发送文件数据
                if (!sendFileData(data_msg.get(), file, Next_Seq,
last_length)) {
                    return false;
                }
            }
            printWindowStatus();
            // 更新进度和速率
            total_sent_bytes += data_msg[Next_Seq - 1].length;
            printTransferProgress(total_sent_bytes, file_length,
start_time);

            Next_Seq++;
        }

        // 流控制：当窗口接近满时适当延迟
        if (Next_Seq - Base_Seq > windows_Size * 0.8) {
            this_thread::sleep_for(chrono::milliseconds(10));
        }
    }
}

```

服务器端

服务器端同样封装了函数，让主循环看着更加简洁明了。

`Receive_Message()` 函数首先进行必要的**初始化**工作，包括设置序列号、准备文件名缓冲区，以及建立接收过程所需的各项跟踪变量。

1. 接收文件头

第一个关键步骤是通过 `receiveFileHeader()` 接收文件头信息。文件头包含了文件名和大小等元数据，这些信息对于管理后续的文件传输过程至关重要。如果文件头接收失败，函数会立即返回 `false`，体现了快速失败的设计原则。采用循环等待的方式接收文件头信息，在接收过程中实现了完善的错误检测和超时重传机制。

```

bool receiveFileHeader(char* file_name, UDP_Packet& rec_msg, int&
waiting_Seq, socklen_t routerAddrLen) {
    auto start_time = chrono::steady_clock::now();

```



```

while (true) {
    if (recvfrom(serverSocket, (char*)&rec_msg, sizeof(rec_msg), 0,
        (SOCKADDR*)&routerAddress, &routerAddrLen) > 0) {

        if (rec_msg.Is_CFH() && rec_msg.CheckValid() && rec_msg.seq ==
waiting_Seq) {

            file_length = rec_msg.length;
            strcpy_s(file_name, MAX_DATA_SIZE, rec_msg.data);

            SetConsoleTextAttribute(hConsole, 11);
            cout << "[接收] 文件头信息: "
                << "\n文件名: " << file_name
                << "\n大小: " << formatFileSize(file_length) << endl;
            SetConsoleTextAttribute(hConsole, 7);

            // 发送确认
            UDP_Packet ack_packet;
            ack_packet.ack = rec_msg.seq;
            ack_packet.Set_ACK();
            ack_packet.check = ack_packet.Calculate_Checksum();

            if (sendto(serverSocket, (char*)&ack_packet,
sizeof(ack_packet), 0,
                (SOCKADDR*)&routerAddress, routerAddrLen) > 0) {
                waiting_Seq++;
                return true;
            }
        }
        else if (rec_msg.Is_CFH() && rec_msg.CheckValid()) {
            sendDuplicateAck(waiting_Seq - 1);
        }
    }

    // 超时检查
    if (chrono::duration_cast<chrono::milliseconds>(
        chrono::steady_clock::now() - start_time).count() > TIMEOUT) {
        SetConsoleTextAttribute(hConsole, 12);
        cout << "[超时] 等待文件头超时, 请求重传" << endl;
        SetConsoleTextAttribute(hConsole, 7);
        sendDuplicateAck(waiting_Seq - 1);
        start_time = chrono::steady_clock::now();
    }
}
}

```

2. 文件写入缓冲区

在成功接收文件头后，函数通过初始化一个具有 1MB 缓冲区的 **BufferedFileWriter** 来准备文件写入操作。这种缓冲写入的方式通过减少实际的磁盘写入频率，显著优化了 I/O 性能。

```

class BufferedFilewriter {
private:
    ofstream file;
    vector<char> buffer;

```

```

    size_t current_pos;

public:
    BufferedFileWriter(const string& filename, size_t buffer_size)
        : buffer(buffer_size), current_pos(0) {
        file.open(filename, ios::binary);
    }

    void write(const char* data, size_t length) {
        while (length > 0) {
            size_t space = buffer.size() - current_pos;
            size_t to_write = min(space, length);

            memcpy(&buffer[current_pos], data, to_write);
            current_pos += to_write;
            data += to_write;
            length -= to_write;

            if (current_pos == buffer.size()) {
                flush();
            }
        }
    }

    void flush() {
        if (current_pos > 0) {
            file.write(buffer.data(), current_pos);
            current_pos = 0;
        }
    }

    ~BufferedFileWriter() {
        flush();
        file.close();
    }
};

```

3. 接收数据包

函数的核心是一个持续运行直到接收完所有预期数据的 while 循环。这个循环实现了几个关键功能：

1. **数据包接收**：使用 receivePacketWithTimeout 实现可靠的数据包接收，包含超时处理机制。
2. **进度监控**：通过每 100 毫秒更新一次显示的方式，在保持用户及时了解传输状态的同时，避免了过于频繁的控制台更新。
3. **断点管理**：每接收 5MB 数据保存一次断点信息，为可能的传输中断提供恢复机制。

```

// 主接收循环
while (total_received_bytes < file_length) {
    UDP_Packet packet;
    auto receive_result = receivePacketWithTimeout(packet, routerAddrLen,
    TIMEOUT);

    if (handleReceivedPacket(packet, waiting_seq, filewriter,
    total_received_bytes)) {
        // 更新进度显示
    }
}

```

```

        auto current_time = chrono::steady_clock::now();
        if (chrono::duration_cast<chrono::milliseconds>(current_time -
last_progress_update).count() >= 100) {
            printReceiveProgress(total_received_bytes, file_length,
start_time);
            last_progress_update = current_time;
        }
    }

    // 检查是否需要保存断点续传信息
    if (total_received_bytes % (5 * 1024 * 1024) == 0) { // 每5MB保存一次
        saveCheckpoint(filePath, total_received_bytes);
    }
}

```

在确认包发送成功后，函数执行三个重要操作：

1. 使用 **BufferedFileWriter** 将数据写入文件
2. 更新已接收数据的总量 (**total_received**)
3. 递增期望序列号 (**Waiting_Seq**)

错误处理机制 当接收到的数据包校验和正确但序列号不匹配时，函数通过 `sendDuplicateAck` 发送重复确认包，刺激发送端重新发送当前窗口中的全部数据包，获取正确序列号的数据包。这种机制有效处理了数据包乱序和丢失的情况。

```

bool handleReceivedPacket(const UDP_Packet& packet, int& waiting_seq,
    BufferedFileWriter& writer, uint64_t& total_received) {

    if (packet.CheckValid() && packet.seq == waiting_seq) {
        // 发送确认
        UDP_Packet ack_packet;
        ack_packet.ack = packet.seq;
        ack_packet.Set_ACK();
        ack_packet.check = ack_packet.Calculate_Checksum();

        if (sendto(serverSocket, (char*)&ack_packet, sizeof(ack_packet), 0,
            (SOCKADDR*)&routerAddress, sizeof(routerAddress)) > 0) {

            // 写入数据
            writer.write(packet.data, packet.length);
            total_received += packet.length;
            waiting_seq++;
            return true;
        }
    }
    else if (packet.CheckValid()) {
        sendDuplicateAck(waiting_seq - 1);
    }
    return false;
}

```

5.四次挥手——断开连接

客户端

`disconnect()` 方法实现了基于四次挥手机制的连接断开流程。通过确保双方的 FIN 和 ACK 消息正确收发，达到可靠断开连接的目的。

1. 第一次挥手：发送 FIN 消息

- 通过 `Set_FIN` 设置 FIN 标志位，表示开始断开连接。
- 设置源端口和目标端口，计算校验和，并通过 `sendto` 发送消息。
- 如果超时未收到 ACK，重传 FIN 消息。

2. 第二次挥手：接收 ACK 消息

- 通过 `recvfrom` 接收 ACK 消息。
- 验证消息合法性：
 - 是否设置了 ACK 标志位。
 - ACK 是否对应第一次挥手的序列号。
 - 校验和是否正确。
- 超时重传第一次挥手的 FIN 消息。

3. 第三次挥手：接收 FIN 消息

- 通过 `recvfrom` 接收服务端的 FIN 消息。
- 验证消息合法性：
 - 是否设置了 FIN 标志位。
 - 校验和是否正确。
- 如果超时未收到 FIN，断开连接失败。

4. 第四次挥手：发送 ACK 消息

- 设置 ACK 标志位，确认服务端的 FIN 消息。
- 通过 `sendto` 发送 ACK，标志连接已完全断开。

```
bool Disconnect() {
    UDP_Packet wavehand_packets[4]; // 定义四次挥手消息数组
    socklen_t addr_len = sizeof(routerAddr);
    auto start_time = chrono::steady_clock::now();

    // 初始化挥手消息数组
    memset(wavehand_packets, 0, sizeof(wavehand_packets)); // 清零消息结构体数组

    // 第一次挥手：发送 FIN 消息
    wavehand_packets[0].src_port = CLIENT_PORT;
    wavehand_packets[0].dest_port = ROUTER_PORT;
    wavehand_packets[0].Set_FIN();
    wavehand_packets[0].seq = ++seq;
    wavehand_packets[0].check = wavehand_packets[0].Calculate_Checksum();
    cout << "[日志] 第一次挥手：发送 FIN 消息，序列号：" <<
    wavehand_packets[0].seq << endl;
    if (sendto(clientSocket, (char*)&wavehand_packets[0],
    sizeof(wavehand_packets[0]), 0,
    (sockaddr*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR) {
```

```

cerr << "[错误] FIN 消息发送失败，错误代码: " << WSAGetLastError() <<
endl;

    return false;
}

while (true) {
    // 第二次挥手：等待 ACK 消息
    if (recvfrom(clientSocket, (char*)&wavehand_packets[1],
sizeof(wavehand_packets[1]), 0,
(sockaddr*)&routerAddr, &addr_len) > 0) {
        if (wavehand_packets[1].Is_ACK() &&
            wavehand_packets[1].ack == wavehand_packets[0].seq &&
            wavehand_packets[1].CheckValid()) {
            cout << "[日志] 收到第二次挥手消息 (ACK)，确认序列号: " <<
wavehand_packets[1].ack << endl;
            break;
        }
        else {
            cerr << "[警告] 收到无效的 ACK 消息，丢弃。" << endl;
        }
    }

    // 超时重传第一次挥手消息
    auto now = chrono::steady_clock::now();
    if (chrono::duration_cast<chrono::milliseconds>(now -
start_time).count() > TIMEOUT) {
        cout << "[日志] FIN 消息超时，重新发送。" << endl;
        wavehand_packets[0].check =
wavehand_packets[0].Calculate_Checksum(); // 重算校验和
        if (sendto(clientSocket, (char*)&wavehand_packets[0],
sizeof(wavehand_packets[0]), 0,
(sockaddr*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR)
{
            cerr << "[错误] 重传失败。" << endl;
            return false;
        }
        start_time = now; // 更新计时
    }
}

// 第三次挥手：接收 FIN 消息
start_time = chrono::steady_clock::now();
while (true) {
    if (recvfrom(clientSocket, (char*)&wavehand_packets[2],
sizeof(wavehand_packets[2]), 0,
(sockaddr*)&routerAddr, &addr_len) > 0) {
        cout << wavehand_packets[2].Is_FIN() <<
wavehand_packets[2].CheckValid();
        if (wavehand_packets[2].Is_FIN() &&
            wavehand_packets[2].CheckValid()) {
            cout << "[日志] 收到第三次挥手消息 (FIN)，序列号: " <<
wavehand_packets[2].seq << endl;
            break;
        }
        else {
            wavehand_packets[2].Print_Message();

```

```

        cerr << "[警告] 收到无效的 FIN 消息，丢弃。" << endl;
    }
}

// 超时处理
auto now = chrono::steady_clock::now();
if (chrono::duration_cast<chrono::milliseconds>(now -
start_time).count() > TIMEOUT) {
    cerr << "[日志] 等待 FIN 超时，断开连接失败。" << endl;
    return false;
}
}
seq = wavehand_packets[2].seq;
// 第四次挥手：发送 ACK 消息
wavehand_packets[3].src_port = CLIENT_PORT;
wavehand_packets[3].dest_port = ROUTER_PORT;
wavehand_packets[3].Set_ACK();
wavehand_packets[3].ack = wavehand_packets[2].seq;
wavehand_packets[3].seq = ++seq;
wavehand_packets[3].check = wavehand_packets[3].Calculate_Checksum();
if (sendto(clientSocket, (char*)&wavehand_packets[3],
sizeof(wavehand_packets[3]), 0,
(sockaddr*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR) {
    cerr << "[错误] 第四次挥手消息发送失败，错误代码：" << WSAGetLastError() <<
endl;
    return false;
}
cout << "[日志] 第四次挥手：发送 ACK 消息，确认序列号：" <<
wavehand_packets[3].ack << endl;

// 等待 2 * TIMEOUT 时间以确保消息完成
cout << "[日志] 等待 2 * TIMEOUT 确保连接断开..." << endl;
this_thread::sleep_for(chrono::milliseconds(2 * TIMEOUT));
return true;
}

```

服务器端

服务器端实现断开连接的操作与客户端相对应：

1. 第一次挥手：接收 FIN 消息

- 调用 `recvfrom` 函数接收客户端的 **FIN** 消息。
- 使用 `Is_FIN` 和 `CheckValid` 验证消息的合法性，确保收到的是有效的 **FIN** 消息。

2. 第二次挥手：发送 ACK 消息

- 构造 **ACK** 消息，通过 `Set_ACK` 设置 **ACK** 标志位。
- 使用 `Calculate_Checksum` 计算校验和，确保消息完整性。
- 调用 `sendto` 函数发送 **ACK** 消息给客户端。

3. 第三次挥手：发送 FIN 消息

- 构造 **FIN** 消息，使用 `Set_FIN` 和 `Set_ACK` 同时设置 **FIN** 和 **ACK** 标志位。
- 再次通过 `Calculate_Checksum` 计算校验和，调用 `sendto` 函数发送消息。

4. 第四次挥手：接收 ACK 消息

- 调用 `recvfrom` 接收客户端发送的 **ACK** 消息。
- 验证消息合法性：
 - 是否设置了 **ACK** 标志位。
 - 确认号是否匹配服务端发送的 **FIN** 消息序列号。
- 如果在超时时间内未收到 ACK，重传 **FIN** 消息。
- 重新计算校验和，确保重传消息完整。

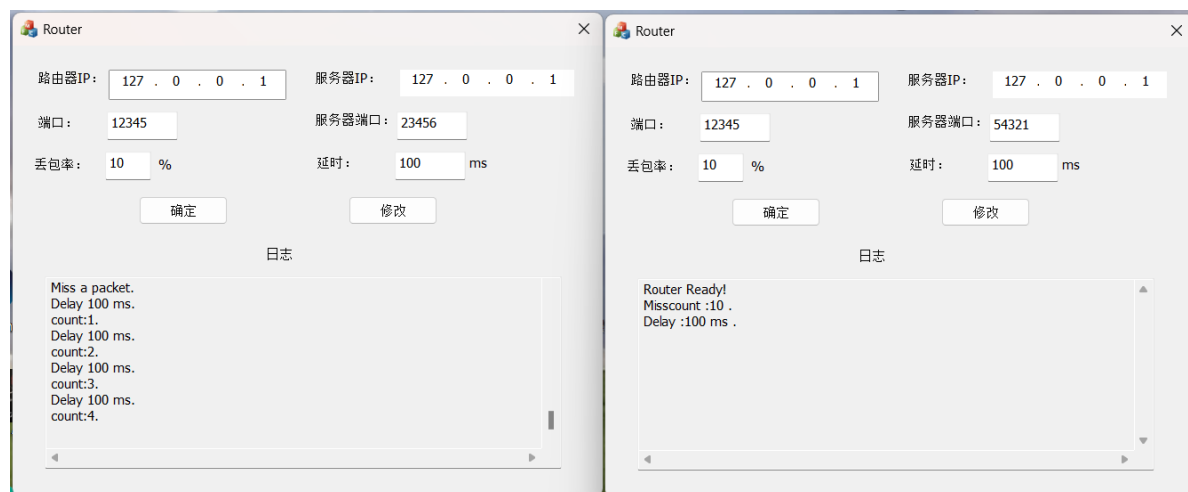
断开连接代码部分与客户端高度重合，此处不做展示。

程序效果

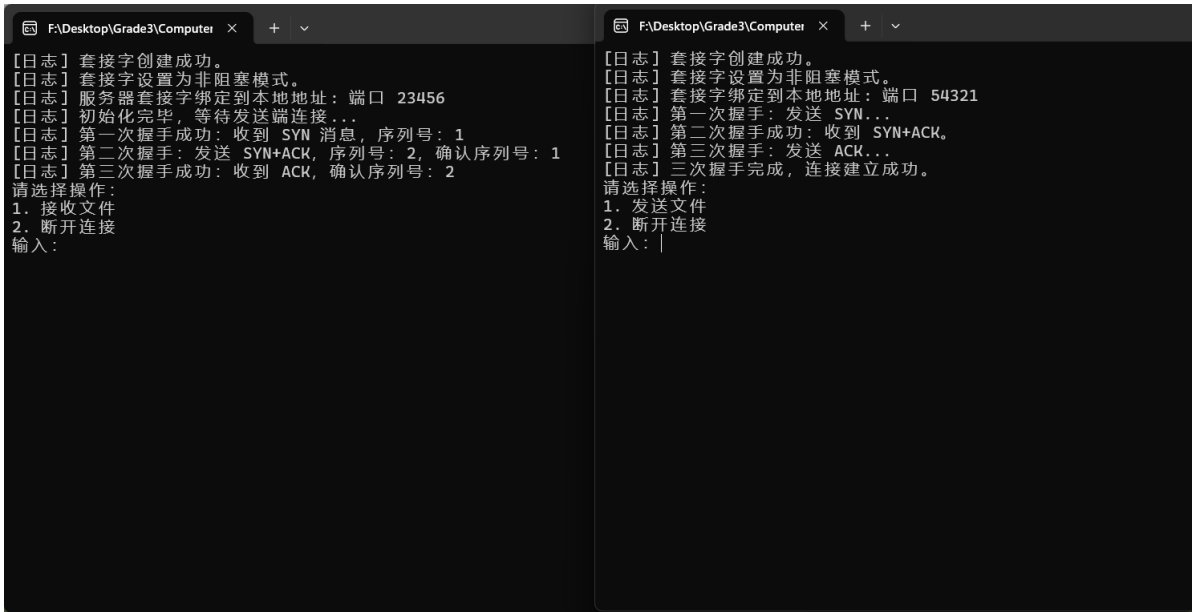
为了方便对程序的运行效果进行测试，客户端与服务器端之间的交互要通过一个路由转发，基本原理如图所示：



首先我们对路由程序进行设置，路由IP地址为127.0.0.1，端口号为12345，服务器端端口号为23456，客户端端口号为54321：



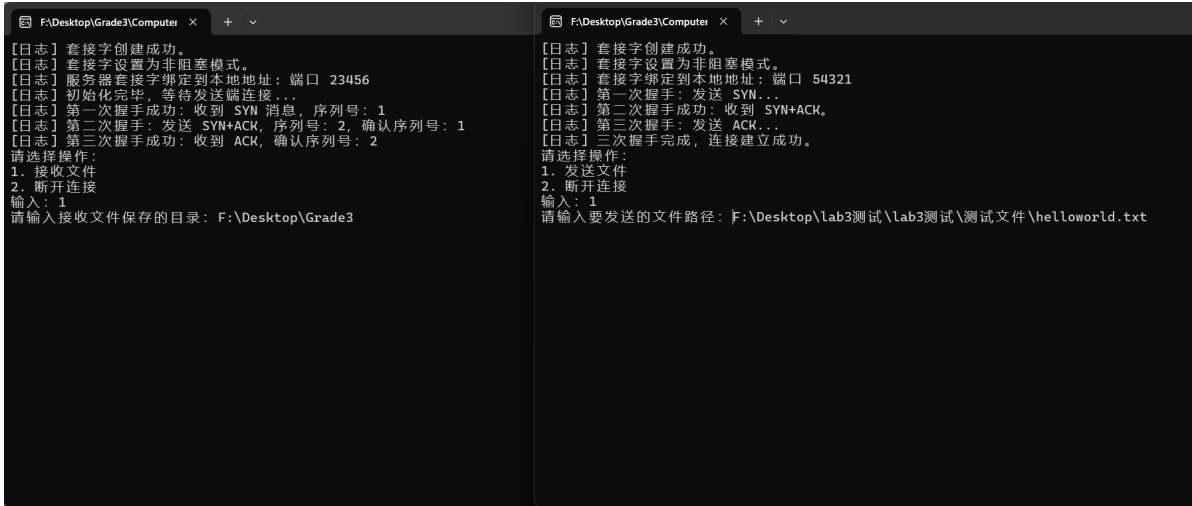
我们分别打开客户端和服务端程序，双方通过路由转发完成三次握手，同时给用户进行下一步操作的选择，输入1为发送或接收文件，输入2为断开连接：



在wireshark中我们捕获到了本地回环中的udp数据包，由于中间通过了一次路由转发，所以三次握手的过程一共收到了捕获到了六个数据包：

正在捕获 Adapter for loopback traffic capture							
文件(E) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)							
udp							
No.	Time	Source	Destination	Protocol	Length	Info	
133	20.924102	127.0.0.1	127.0.0.1	UDP	1080	54321 → 12345	Len=1048
136	21.046014	127.0.0.1	127.0.0.1	UDP	1080	12345 → 23456	Len=1048
137	21.047299	127.0.0.1	127.0.0.1	UDP	1080	23456 → 12345	Len=1048
138	21.062252	127.0.0.1	127.0.0.1	UDP	1080	12345 → 54321	Len=1048
139	21.062592	127.0.0.1	127.0.0.1	UDP	1080	54321 → 12345	Len=1048
140	21.182840	127.0.0.1	127.0.0.1	UDP	1080	12345 → 23456	Len=1048

接下来我们在客户端和服务端选择发送和接收文件功能，选择指定文件路径：



文件传输完毕后我们可以看到文件成功传输完毕，并将文件传输过程的总耗时和吞吐率打印在终端里：

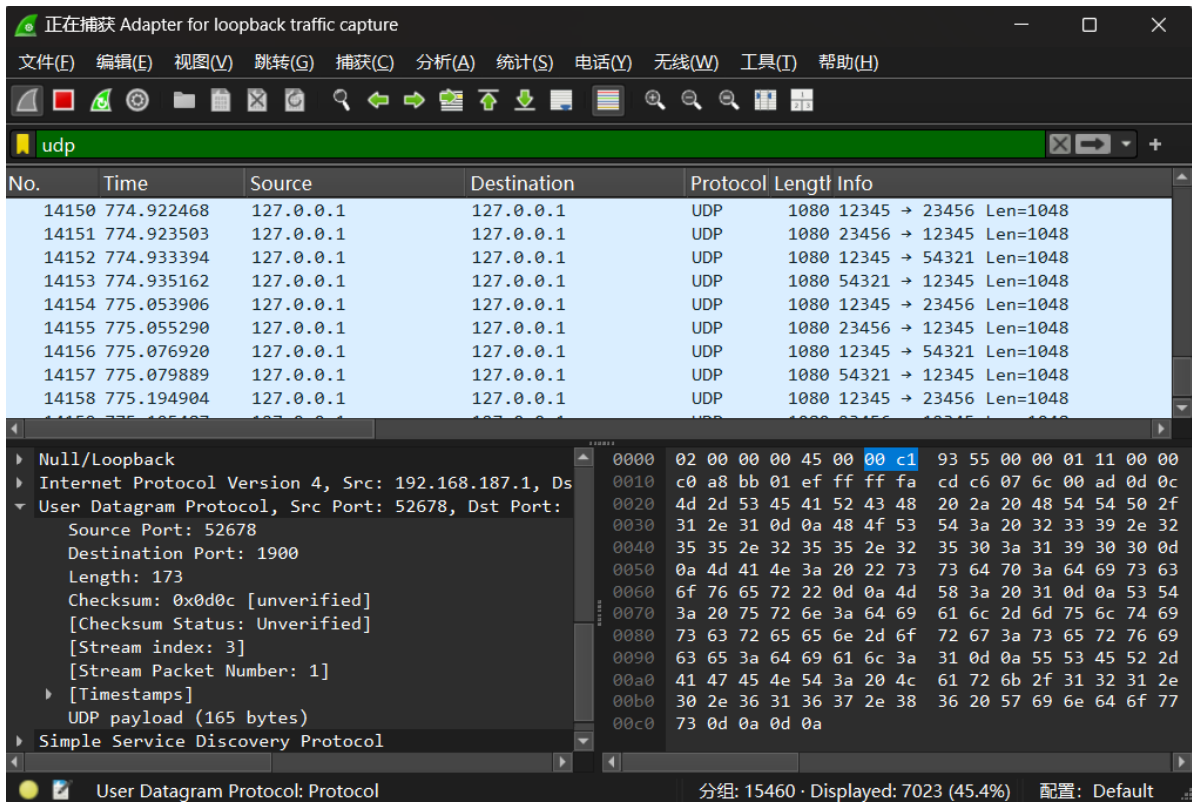

```
C:\Users\田晋宇\source\repos X + v
[重传] 发送重复ACK, 序列号: 136
接收进度: [=====>] 73% 1.30 MB/1.77 MB (240.89 KB/s)
接收进度: [=====>] 75% 1.34 MB/1.77 MB (242.69 KB/s)
接收进度: [=====>] 77% 1.37 MB/1.77 MB (242.63 KB/s)
接收进度: [=====>] 79% 1.41 MB/1.77 MB (244.29 KB/s)
接收进度: [=====>] 81% 1.45 MB/1.77 MB (245.87 KB/s)
接收进度: [=====>] 83% 1.48 MB/1.77 MB (247.46 KB/s)
接收进度: [=====>] 86% 1.52 MB/1.77 MB (248.48 KB/s)
接收进度: [=====>] 87% 1.55 MB/1.77 MB (249.25 KB/s)
[重传] 发送重复ACK, 序列号: 164
[重传] 发送重复ACK, 序列号: 164
[重传] 发送重复ACK, 序列号: 164
[重传] 发送重复ACK, 序列号: 164
接收进度: [=====>] 88% 1.57 MB/1.77 MB (243.06 KB/s)
接收进度: [=====>] 90% 1.61 MB/1.77 MB (244.47 KB/s)
接收进度: [=====>] 92% 1.64 MB/1.77 MB (245.00 KB/s)
接收进度: [=====>] 94% 1.67 MB/1.77 MB (245.55 KB/s)
接收进度: [=====>] 96% 1.71 MB/1.77 MB (246.88 KB/s)
接收进度: [=====>] 98% 1.74 MB/1.77 MB (247.19 KB/s)
接收进度: [=====] 100% 1.77 MB/1.77 MB (247.67 KB/s)

[完成] 文件接收完成
保存位置: F:\Desktop\Grade3\1.jpg
文件大小: 1.77 MB
总耗时: 7.33 秒
平均速度: 247.61 KB/s
请选择操作:
1. 接收文件
2. 断开连接
输入:

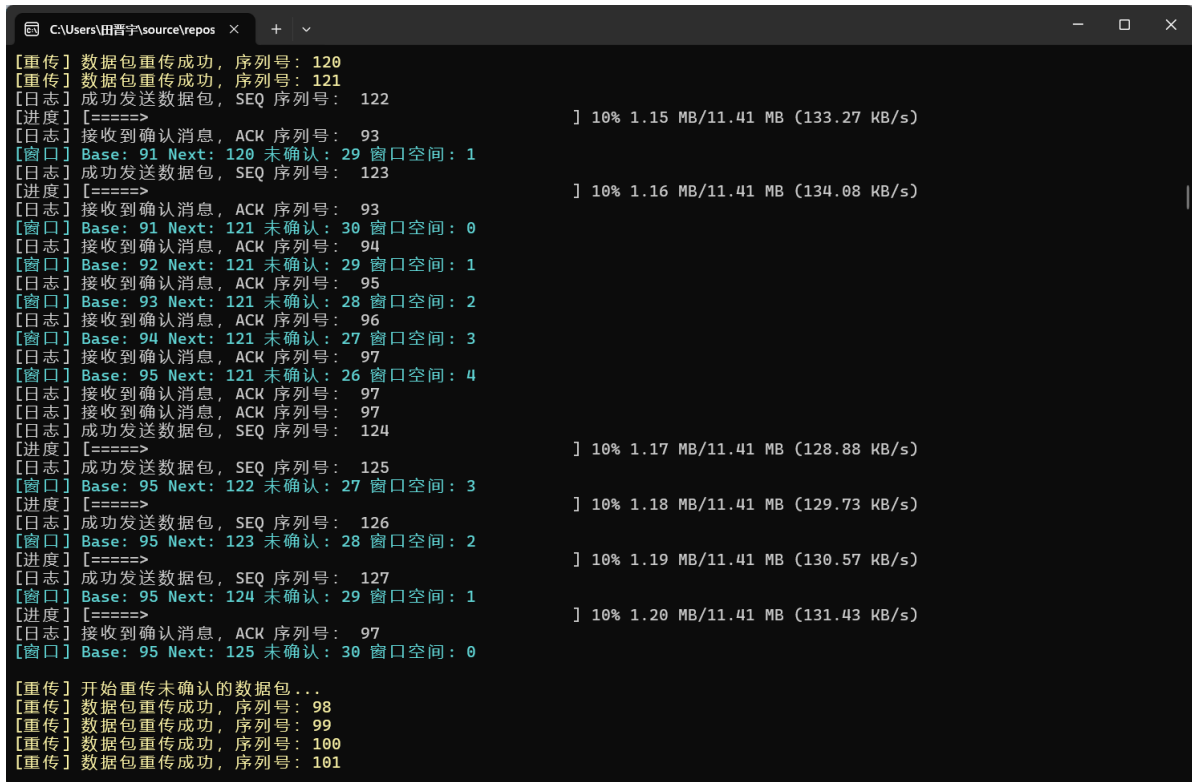
C:\Users\田晋宇\source\repos X + v
[进度] [=====>] 97% 1.73 MB/1.77 MB (247.97 KB/s)
[日志] 接收到确认消息, ACK 序列号: 177
[日志] 接收到确认消息, ACK 序列号: 178
[日志] 接收到确认消息, ACK 序列号: 179
[窗口] Base: 177 Next: 179 未确认: 2 窗口空间: 3
[进度] [=====>] 98% 1.74 MB/1.77 MB (248.28 KB/s)
[窗口] Base: 177 Next: 180 未确认: 3 窗口空间: 2
[进度] [=====>] 98% 1.75 MB/1.77 MB (249.52 KB/s)
[窗口] Base: 177 Next: 181 未确认: 4 窗口空间: 1
[进度] [=====>] 99% 1.76 MB/1.77 MB (250.83 KB/s)
[日志] 接收到确认消息, ACK 序列号: 180
[窗口] Base: 178 Next: 182 未确认: 4 窗口空间: 1
[进度] [=====>] 99% 1.76 MB/1.77 MB (250.09 KB/s)
[日志] 接收到确认消息, ACK 序列号: 181
[窗口] Base: 179 Next: 183 未确认: 4 窗口空间: 1
[进度] [=====] 100% 1.77 MB/1.77 MB (248.29 KB/s)
[日志] 接收到确认消息, ACK 序列号: 182
[日志] 接收到确认消息, ACK 序列号: 183
[日志] 接收到确认消息, ACK 序列号: 184
[日志] 接收到确认消息, ACK 序列号: 185
[日志] 接收到确认消息, ACK 序列号: 186

[完成] 文件传输完成
总大小: 1.77 MB
耗时: 7.36 秒
平均速度: 246.56 KB/s
请选择操作:
1. 发送文件
2. 断开连接
输入: |
```

wireshark中捕获到的文件传输数据包:



为了深入理解滑动窗口和重传机制的原理，我们将滑动窗口的大小调整为20，再来看终端的调试信息。可以看到发送端在接收消息的同时可以发送数据包，而并非之前的串行发送方式，当接收到确认数据包时，Base指针向前移动，当发送数据包时，Next指针向前移动；确认消息时当接收端接收到了乱序的数据包，如下图中显示，接收端本应该期待接收98号数据包，但由于路由程序的丢包，接收到了99号数据包，发送端由于未收到98数据包的确认号，因此窗口基指针不会向前移，此时接收端重复发送97号的确认号，激发发送端的快速重传机制，发送当前窗口中的所有数据包，此时接收端又重新收到了丢失的165号数据包：



```
C:\Users\田晋宇\source\repos x + v

接收进度: [====>] ] 7% 870.00 KB/11.41 MB (106.67 KB/s)
[重传] 发送重复ACK, 序列号: 91
[重传] 发送重复ACK, 序列号: 91
[重传] 发送重复ACK, 序列号: 91
[重传] 发送重复ACK, 序列号: 91
[重传] 发送重复ACK, 序列号: 91
[重传] 发送重复ACK, 序列号: 91
[重传] 发送重复ACK, 序列号: 91
[重传] 发送重复ACK, 序列号: 91
接收进度: [====>] ] 7% 880.00 KB/11.41 MB (101.73 KB/s)
[重传] 发送重复ACK, 序列号: 93
[重传] 发送重复ACK, 序列号: 93
接收进度: [====>] ] 7% 900.00 KB/11.41 MB (100.51 KB/s)
接收进度: [====>] ] 7% 930.00 KB/11.41 MB (102.28 KB/s)
[重传] 发送重复ACK, 序列号: 97
[重传] 发送重复ACK, 序列号: 97
[重传] 发送重复ACK, 序列号: 97
[重传] 发送重复ACK, 序列号: 97
[重传] 发送重复ACK, 序列号: 97
接收进度: [====>] ] 8% 940.00 KB/11.41 MB (98.67 KB/s)
接收进度: [====>] ] 8% 970.00 KB/11.41 MB (100.34 KB/s)
[重传] 发送重复ACK, 序列号: 102
[重传] 发送重复ACK, 序列号: 102
[重传] 发送重复ACK, 序列号: 102
[重传] 发送重复ACK, 序列号: 102
[重传] 发送重复ACK, 序列号: 102
接收进度: [====>] ] 8% 990.00 KB/11.41 MB (98.38 KB/s)
接收进度: [====>] ] 8% 1010.00 KB/11.41 MB (99.20 KB/s)
接收进度: [====>] ] 8% 1.02 MB/11.41 MB (101.07 KB/s)
```

其他几个测试样里的测试结果如下，传输完毕后均可以正常打开：

```
C:\Users\田晋宇\source\repos x + v
[接收进度: [=====] ] 92% 5.22 MB/5.63 MB (188.86 KB/s)
[接收进度: [=====] ] 93% 5.25 MB/5.63 MB (188.83 KB/s)
[接收进度: [=====] ] 93% 5.27 MB/5.63 MB (188.69 KB/s)
[接收进度: [=====] ] 94% 5.29 MB/5.63 MB (188.73 KB/s)
[接收进度: [=====] ] 94% 5.31 MB/5.63 MB (188.76 KB/s)
[接收进度: [=====] ] 94% 5.34 MB/5.63 MB (188.79 KB/s)
[接收进度: [=====] ] 95% 5.37 MB/5.63 MB (188.86 KB/s)
[接收进度: [=====] ] 96% 5.40 MB/5.63 MB (189.09 KB/s)
[接收进度: [=====] ] 96% 5.43 MB/5.63 MB (189.40 KB/s)
[接收进度: [=====] ] 97% 5.46 MB/5.63 MB (189.72 KB/s)
[接收进度: [=====] ] 97% 5.49 MB/5.63 MB (189.95 KB/s)
[重传] 发送重复ACK, 序列号: 566
[重传] 发送重复ACK, 序列号: 566
[重传] 发送重复ACK, 序列号: 566
[重传] 发送重复ACK, 序列号: 566
接收进度: [=====] ] 97% 5.50 MB/5.63 MB (188.67 KB/s)
接收进度: [=====] ] 98% 5.53 MB/5.63 MB (188.83 KB/s)
接收进度: [=====] ] 98% 5.56 MB/5.63 MB (188.95 KB/s)
接收进度: [=====] ] 99% 5.59 MB/5.63 MB (189.17 KB/s)
接收进度: [=====] ] 99% 5.62 MB/5.63 MB (189.46 KB/s)
[完成] 文件接收完成
保存位置: F:\Desktop\Grade3\2.jpg
文件大小: 5.63 MB
总耗时: 30.41 秒
平均速度: 189.41 KB/s
请选择操作:
1. 接收文件
2. 断开连接
输入: |

C:\Users\田晋宇\source\repos x + v
[进度: [=====] ] 99% 5.59 MB/5.63 MB (190.09 KB/s)
[日志] 接收到确认消息, ACK 序列号: 572
[日志] 接收到确认消息, ACK 序列号: 573
[日志] 接收到确认消息, ACK 序列号: 574
[日志] 接收到确认消息, ACK 序列号: 575
[日志] 接收到确认消息, ACK 序列号: 576
[窗口] Base: 574 Next: 574 未确认: 0 窗口空间: 5
[进度: [=====] ] 99% 5.60 MB/5.63 MB (189.12 KB/s)
[窗口] Base: 574 Next: 575 未确认: 1 窗口空间: 4
[进度: [=====] ] 99% 5.61 MB/5.63 MB (189.44 KB/s)
[窗口] Base: 574 Next: 576 未确认: 2 窗口空间: 3
[进度: [=====] ] 99% 5.62 MB/5.63 MB (189.76 KB/s)
[窗口] Base: 574 Next: 577 未确认: 3 窗口空间: 2
[进度: [=====] ] 99% 5.62 MB/5.63 MB (189.76 KB/s)
[窗口] Base: 574 Next: 578 未确认: 4 窗口空间: 1
[进度: [=====] ] 100% 5.63 MB/5.63 MB (190.07 KB/s)
[日志] 接收到确认消息, ACK 序列号: 577
[日志] 接收到确认消息, ACK 序列号: 578
[日志] 接收到确认消息, ACK 序列号: 579
[日志] 接收到确认消息, ACK 序列号: 580
[日志] 接收到确认消息, ACK 序列号: 581
[完成] 文件传输完成
总大小: 5.63 MB
耗时: 30.41 秒
平均速度: 188.97 KB/s
请选择操作:
1. 发送文件
2. 断开连接
输入: |

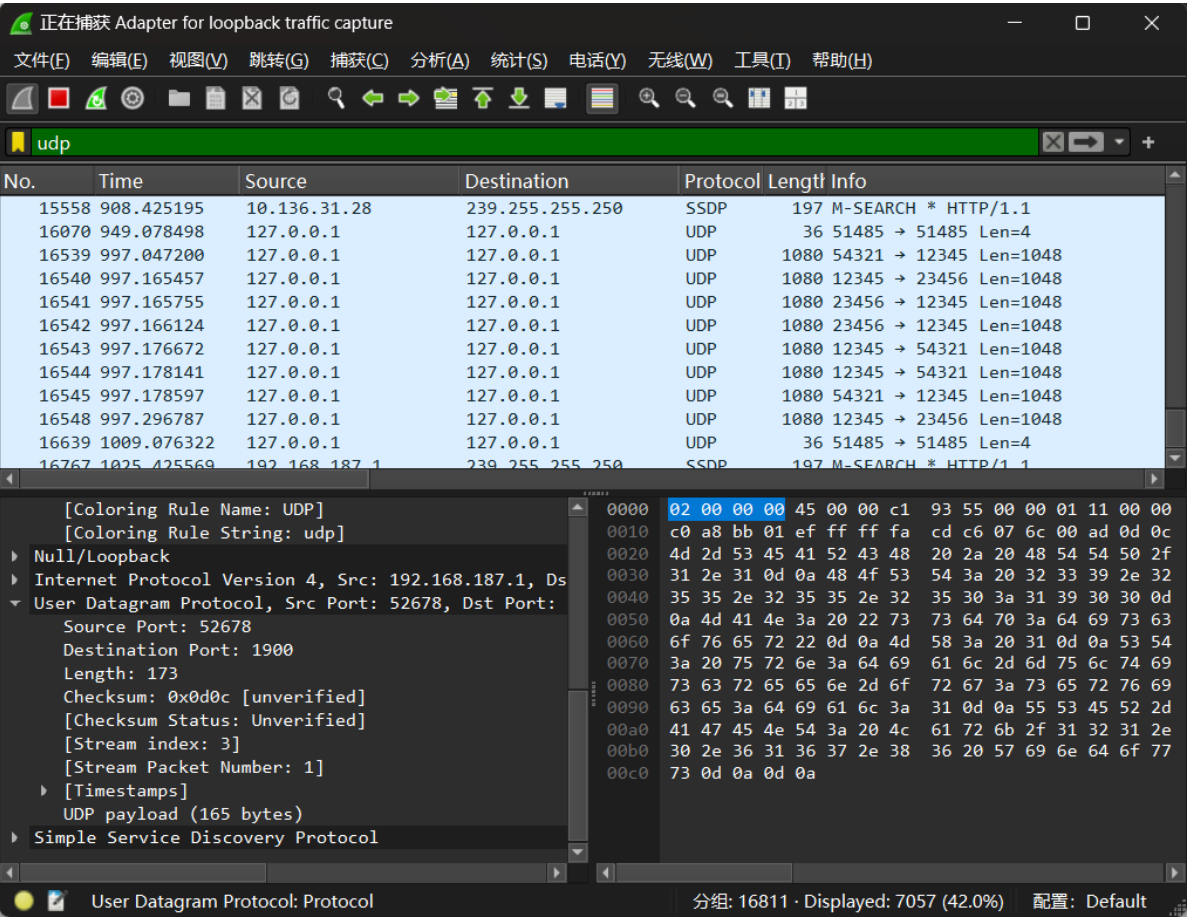
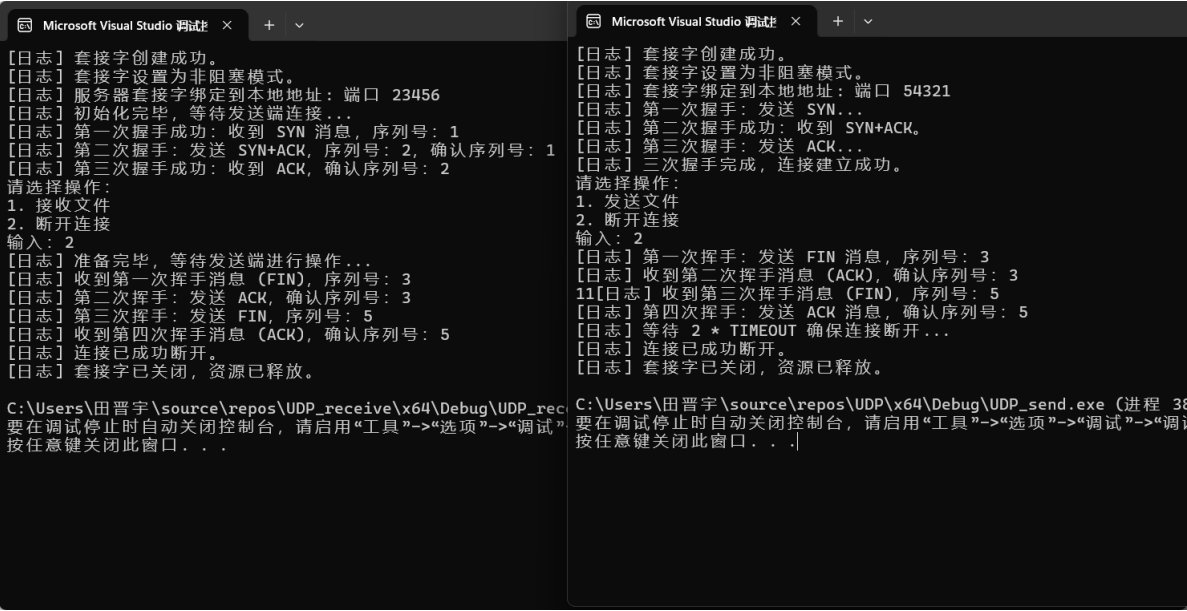
C:\Users\田晋宇\source\repos x + v
[重传] 发送重复ACK, 序列号: 1134
[重传] 发送重复ACK, 序列号: 1134
[重传] 发送重复ACK, 序列号: 1134
接收进度: [=====] ] 96% 11.04 MB/11.41 MB (234.38 KB/s)
接收进度: [=====] ] 97% 11.08 MB/11.41 MB (234.68 KB/s)
接收进度: [=====] ] 97% 11.11 MB/11.41 MB (234.62 KB/s)
接收进度: [=====] ] 97% 11.14 MB/11.41 MB (234.58 KB/s)
接收进度: [=====] ] 97% 11.17 MB/11.41 MB (234.65 KB/s)
接收进度: [=====] ] 98% 11.20 MB/11.41 MB (234.72 KB/s)
接收进度: [=====] ] 98% 11.23 MB/11.41 MB (234.86 KB/s)
接收进度: [=====] ] 98% 11.26 MB/11.41 MB (234.98 KB/s)
接收进度: [=====] ] 98% 11.30 MB/11.41 MB (235.19 KB/s)
[重传] 发送重复ACK, 序列号: 1162
[重传] 发送重复ACK, 序列号: 1162
[重传] 发送重复ACK, 序列号: 1162
[重传] 发送重复ACK, 序列号: 1162
接收进度: [=====] ] 99% 11.32 MB/11.41 MB (234.36 KB/s)
接收进度: [=====] ] 99% 11.36 MB/11.41 MB (234.59 KB/s)
接收进度: [=====] ] 99% 11.39 MB/11.41 MB (234.63 KB/s)
接收进度: [=====] ] 100% 11.41 MB/11.41 MB (234.57 KB/s)
[完成] 文件接收完成
保存位置: F:\Desktop\Grade3\3.jpg
文件大小: 11.41 MB
总耗时: 49.23 秒
平均速度: 234.56 KB/s
请选择操作:
1. 接收文件
2. 断开连接
输入: |

C:\Users\田晋宇\source\repos x + v
[日志] 接收到确认消息, ACK 序列号: 1165
[日志] 接收到确认消息, ACK 序列号: 1166
[日志] 接收到确认消息, ACK 序列号: 1167
[窗口] Base: 1165 Next: 1165 未确认: 0 窗口空间: 5
[进度: [=====] ] 99% 11.37 MB/11.41 MB (227.15 KB/s)
[窗口] Base: 1165 Next: 1166 未确认: 1 窗口空间: 4
[进度: [=====] ] 99% 11.38 MB/11.41 MB (227.34 KB/s)
[窗口] Base: 1165 Next: 1167 未确认: 2 窗口空间: 3
[进度: [=====] ] 99% 11.39 MB/11.41 MB (227.53 KB/s)
[窗口] Base: 1165 Next: 1168 未确认: 3 窗口空间: 2
[进度: [=====] ] 99% 11.40 MB/11.41 MB (227.71 KB/s)
[窗口] Base: 1165 Next: 1169 未确认: 4 窗口空间: 1
[进度: [=====] ] 99% 11.40 MB/11.41 MB (227.87 KB/s)
[日志] 接收到确认消息, ACK 序列号: 1168
[日志] 接收到确认消息, ACK 序列号: 1169
[日志] 接收到确认消息, ACK 序列号: 1170
[日志] 接收到确认消息, ACK 序列号: 1171
[日志] 接收到确认消息, ACK 序列号: 1172
[窗口] Base: 1170 Next: 1170 未确认: 0 窗口空间: 5
[进度: [=====] ] 100% 11.41 MB/11.41 MB (227.20 KB/s)
[日志] 接收到确认消息, ACK 序列号: 1173
[完成] 文件传输完成
总大小: 11.41 MB
耗时: 51.40 秒
平均速度: 227.02 KB/s
请选择操作:
1. 发送文件
2. 断开连接
输入: |

C:\Users\田晋宇\source\repos x + v
[重传] 发送重复ACK, 序列号: 129
[重传] 发送重复ACK, 序列号: 129
[重传] 发送重复ACK, 序列号: 129
[重传] 发送重复ACK, 序列号: 129
接收进度: [=====] ] 77% 1.23 MB/1.58 MB (216.8 KB/s)
接收进度: [=====] ] 79% 1.26 MB/1.58 MB (217.4 KB/s)
接收进度: [=====] ] 81% 1.29 MB/1.58 MB (219.1 KB/s)
接收进度: [=====] ] 83% 1.32 MB/1.58 MB (220.6 KB/s)
接收进度: [=====] ] 85% 1.35 MB/1.58 MB (222.0 KB/s)
接收进度: [=====] ] 87% 1.38 MB/1.58 MB (223.4 KB/s)
接收进度: [=====] ] 89% 1.42 MB/1.58 MB (224.8 KB/s)
接收进度: [=====] ] 91% 1.45 MB/1.58 MB (226.2 KB/s)
接收进度: [=====] ] 93% 1.47 MB/1.58 MB (227.2 KB/s)
[重传] 发送重复ACK, 序列号: 157
[重传] 发送重复ACK, 序列号: 157
[重传] 发送重复ACK, 序列号: 157
[重传] 发送重复ACK, 序列号: 157
接收进度: [=====] ] 95% 1.50 MB/1.58 MB (215.4 KB/s)
接收进度: [=====] ] 97% 1.52 MB/1.58 MB (216.4 KB/s)
接收进度: [=====] ] 98% 1.56 MB/1.58 MB (216.4 KB/s)
[完成] 文件接收完成
保存位置: F:\Desktop\Grade3\helloworld.txt
文件大小: 1.58 MB
总耗时: 7.49 秒
平均速度: 215.91 KB/s
请选择操作:
1. 接收文件
2. 断开连接
输入: |

C:\Users\田晋宇\source\repos x + v
[重传] 数据包重传成功, 序列号: 159
[重传] 数据包重传成功, 序列号: 160
[重传] 数据包重传成功, 序列号: 161
[重传] 数据包重传成功, 序列号: 162
[日志] 接收到确认消息, ACK 序列号: 158
[日志] 接收到确认消息, ACK 序列号: 159
[日志] 接收到确认消息, ACK 序列号: 160
[日志] 接收到确认消息, ACK 序列号: 161
[日志] 接收到确认消息, ACK 序列号: 162
[窗口] Base: 160 Next: 160 未确认: 0 窗口空间: 5
[进度: [=====] ] 98% 1.55 MB/1.58 MB (216.90 KB/s)
[窗口] Base: 160 Next: 161 未确认: 1 窗口空间: 4
[进度: [=====] ] 98% 1.56 MB/1.58 MB (218.13 KB/s)
[窗口] Base: 160 Next: 162 未确认: 2 窗口空间: 3
[进度: [=====] ] 99% 1.57 MB/1.58 MB (219.02 KB/s)
[窗口] Base: 160 Next: 163 未确认: 3 窗口空间: 2
[进度: [=====] ] 100% 1.58 MB/1.58 MB (220.31 KB/s)
[日志] 接收到确认消息, ACK 序列号: 163
[日志] 接收到确认消息, ACK 序列号: 164
[日志] 接收到确认消息, ACK 序列号: 165
[日志] 接收到确认消息, ACK 序列号: 166
[完成] 文件传输完成
总大小: 1.58 MB
耗时: 7.53 秒
平均速度: 214.84 KB/s
请选择操作:
1. 发送文件
2. 断开连接
输入: |
```

接下来分别在两个窗口中选择断开连接功能，并在wireshark中捕获到了四次挥手的8个数据包：



总结与反思

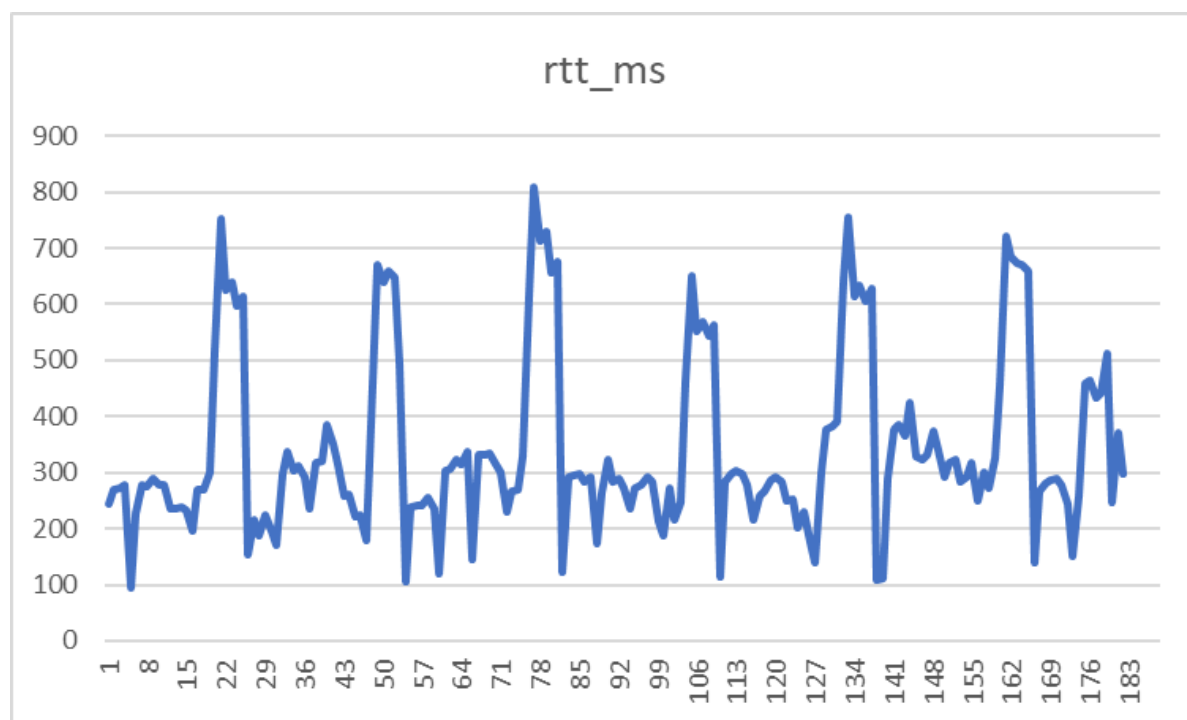
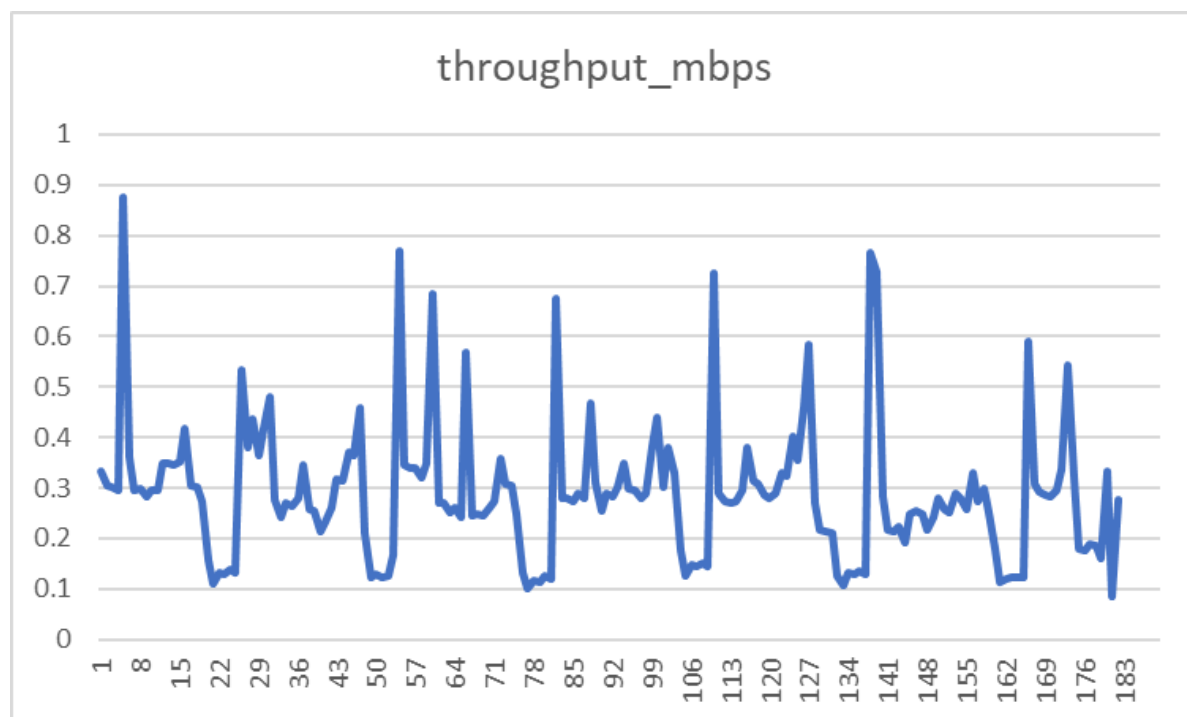
为何使用快速重传机制？

快速重传机制通过接收方主动发送重复ACK来触发数据包重传，而不是等待发送方的超时计时器触发，这种方式显著减少了数据包丢失后的恢复时间，提高了网络带宽利用效率，同时也增强了传输的稳定性和实时性能，是可靠传输协议中不可或缺的组成部分，尤其在高速网络环境和大文件传输场景中发挥着重要作用。

滑动窗口的性能分析？

往返时延分析： 这次传输的RTT表现出较大的波动性。平均RTT约为330毫秒，最短RTT为93.5毫秒（第9号数据包），最长RTT为808.2毫秒（第81号数据包）。在传输过程中，我观察到以下特征：传输开始阶段（序号4-20），RTT相对稳定，大多保持在230-280毫秒之间。随后在序号24-29的传输中出现了一次明显的RTT增长，上升到500-750毫秒范围。类似的RTT峰值还出现在序号80-85和序号164-169之间，这些时段的RTT都超过了600毫秒。这种周期性的RTT波动可能表明网络出现了短暂的拥塞。

吞吐量分析： 实时吞吐量同样表现出显著的波动。平均吞吐率约为0.3 Mbps，最高达到59.7 Mbps（文件头传输），最低降至0.084 Mbps（最后一个数据包）。吞吐率表现出以下特点：文件传输开始时（序号4）达到最高吞吐率，这是由于文件头信息包含了较大的数据量（1.8MB）。在常规数据传输中，吞吐率主要集中在0.2-0.4 Mbps之间。当RTT较高的时段，吞吐率会相应降低到0.1-0.15 Mbps。部分数据包（如序号9、58、86、114和142）出现了较高的吞吐率（0.7-0.8 Mbps），这些可能是得益于临时的网络状况改善。



这次传输展现了典型的网络传输特征，即周期性的性能波动。RTT和吞吐率的变化呈现明显的负相关关系，当RTT增加时，吞吐率随之下降。传输过程中出现的周期性性能波动可能与网络拥塞控制机制有关，下一步可以通过调整发送窗口大小和重传策略来优化传输性能。