计算机网络实验报告

Lab3-3 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)

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**的报文格式进行的了设计，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
`src_port`	4	源端口号
`dest_port`	4	目标端口号
`seq`	4	序列号，用于顺序控制
`ack`	4	确认号，用于确认接收数据
`length`	4	数据包长度
`check`	2	校验和
`flag`	2	标志位，控制协议状态
`data`	1024	数据部分

```
|0                        15|16                             31|
+-----------------------------------------------------------+
|                   src_port (4 bytes)                      |
+-----------------------------------------------------------+
|                   dest_port (4 bytes)                     |
+-----------------------------------------------------------+
|                     seq (4 bytes)                         |
+-----------------------------------------------------------+
```

```
|                      ack (4 bytes)                            |
+---------------------------------------------------------------+
|                     length (4 bytes)                          |
+---------------------------------------------------------------+
|      flag (2 bytes)          |        check (2 bytes)         |
+---------------------------------------------------------------+
|                       data (1024 bytes)                       |
+---------------------------------------------------------------+


|0         |1         |2         |3         |4                15|
+---------------------------------------------------------------+
| FLAG_FIN | FLAG_CFH | FLAG_ACK | FLAG_SYN |      Reserved     |
+---------------------------------------------------------------+
```

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**

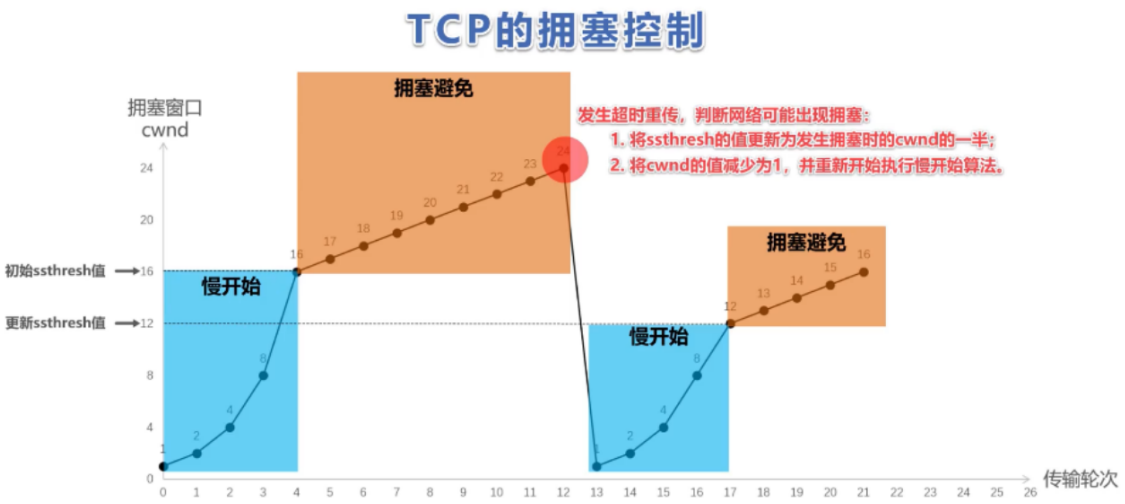
  实验设计的**校验和机制**的核心思想是通过累加数据字段的值来形成一个**校验和**，并在传输数据时携带该校验和。当接收端收到数据时，重新计算数据的校验和，并与收到的校验和对比：

  - 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。

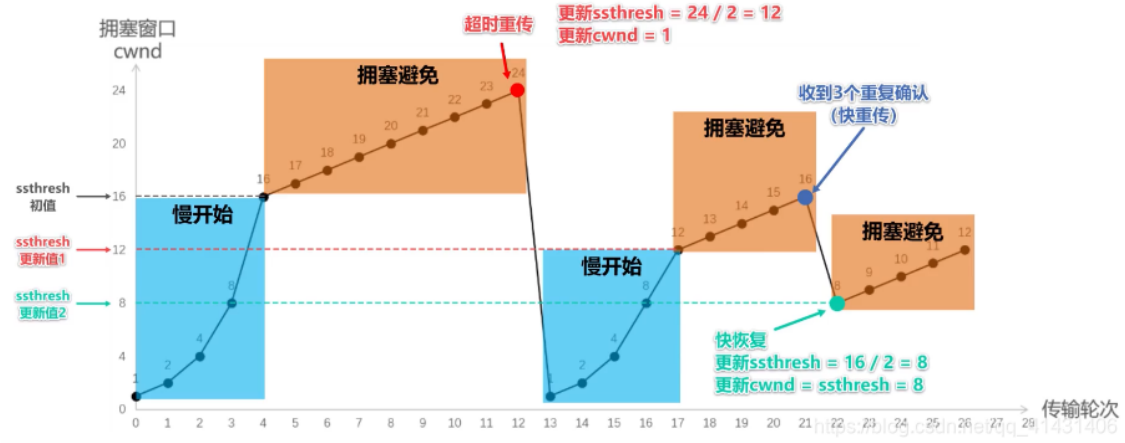
- **拥塞控制机制——RENO算法**

1. 发送方维护一个叫做拥塞窗口cwnd的状态变量，其值取决于网络的拥塞程度，并且动态变化。
   - 拥塞窗口cwnd的维护原则:只要网络没有出现拥塞，拥塞窗口就再增大一些;但只要网络出现拥塞拥塞窗口就减少一些。
   - 判断出现网络拥塞的依据:没有按时收到应当到达的确认报文(即发生重传)

2. 发送方将拥塞窗口作为发送窗口，即swnd=cwnd。
3. 维护一个慢开始门限ssthresh状态变量:
    - 当cwnd<ssthresh时，使用慢开始算法;
    - 当cwnd>ssthresh时，停止使用慢开始算法而改用拥塞避免算法;
    - 当cwnd=ssthresh时，既可使用慢开始算法，也可使用拥塞避免算法。



- **“慢开始”**是指一开始向网络注入的报文段少，并不是指拥塞窗口cwnd增长速度慢;
- **“拥塞避免”**并非指完全能够避免拥塞，而是指在拥塞避免阶段将拥塞窗口控制为按线性规律增长，使网纱比较不容易出现拥塞。



- 发送方一旦收到3个重复确认，就知道现在只是丢失了个别的报文段。于是不启动慢开始算法，而执行快恢复算法;
    - 发送方将慢开始门限ssthresh值和拥塞窗口cwnd值调整为当前窗口的一半;开始执行拥塞避免算法。
    - 也有的快恢复实现是把快恢复开始时的拥塞窗口cwnd值再增大一些，即等于新的ssthresh +3.
        - 既然发送方收到3个重复的确认，就表明有3个数据报文段已经离开了网络;
        - 这3个报文段不再消耗网络资源而是停留在接收方的接收缓存中;
        - 可见现在网络中不是堆积了报文段而是减少了3个报文段。因此可以适当把拥塞窗口扩大些。

- **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 <cstdint>
#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 `Calculate_Checksum()` 的结果判断包是否完整。

```
bool UDP_Packet::CheckValid() const {
    return (check & 0xFFFF) == Calculate_Checksum();
}
```

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 = WSAStartup(MAKEWORD(2, 2), &wsaData);
        if (result != 0) {
            cerr << "[错误] WSAStartup 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

在客户端部分实现了 TCP Reno 拥塞控制算法的基本思路，具体在文件传输过程中对网络的拥塞状况进行监测与调节。TCP Reno 主要通过三个阶段来调整拥塞窗口（`cwnd`），分别是：慢启动（`Slow Start`）、拥塞避免（`Congestion Avoidance`）和快速恢复（`Fast Recovery`）。我们可以根据代码分析其各个阶段的实现和具体操作。

1. 拥塞控制状态定义

```
enum CongestionState {
    SLOW_START,           // 慢启动阶段
    CONGESTION_AVOIDANCE, // 拥塞避免阶段
    FAST_RECOVERY         // 快速恢复阶段
};
```

这里定义了三种拥塞控制状态，分别是：

- **慢启动**：在该阶段，`cwnd`从一个较小的初始值开始增大，通常是 1 MSS (最大报文段)。
- **拥塞避免**：当`cwnd`达到`ssthresh`时，进入此阶段，`cwnd`以较小的增速增长，通常每经过一个往返时延 (RTT) 增加 1 MSS。
- **快速恢复**：当发生丢包时，通过快速重传和快速恢复来减小`cwnd`，避免过度回退。

2. 拥塞控制状态管理

根据`cwnd`的值及收到的 ACK 消息，程序会动态调整当前的拥塞控制状态。以下是每个状态的处理函数：

慢启动阶段（`enterSlowStart`）

```
void enterSlowStart() {
    cwnd = INIT_CWND;             // 初始化窗口大小
    ssthresh = INIT_SSTHRESH;     // 设置慢启动阈值
    congestion_state = SLOW_START;  // 当前状态设置为慢启动
    // 输出状态变化日志
}
```

在慢启动阶段，`cwnd`会逐步增加，直到`cwnd`大于或等于`ssthresh`（慢启动阈值）。如果窗口过大，将转到拥塞避免阶段。

拥塞避免阶段（`enterCongestionAvoidance`）

```
void enterCongestionAvoidance() {
    congestion_state = CONGESTION_AVOIDANCE;
    // 输出状态变化日志
}
```

在拥塞避免阶段，`cwnd` 增长较慢，每经过一个 RTT 增加 1 MSS。当发生丢包（检测到重复 ACK 或超时）时，将进入快速恢复阶段。

快速恢复阶段（`enterFastRecovery`）

```
void enterFastRecovery() {
    ssthresh = max(cwnd / 2, (double)INIT_CWND); // 更新慢启动阈值
    cwnd = ssthresh + 3; // 增加 cwnd
    congestion_state = FAST_RECOVERY;
    // 输出状态变化日志
}
```

在快速恢复阶段，`cwnd` 会减小到 `ssthresh` 的一半，并且增加一些额外的报文段数量。此阶段旨在快速恢复连接，但不会让 `cwnd` 回退得过多。

3. 主线程发送消息

主线程中调用了 `Send_Message` 来发送文件数据：

```
bool Send_Message(string file_path) {
    // 打开文件，获取文件信息
    // 创建数据包缓冲区并初始化
    thread ackThread([this]() { this->Thread_Ack(); });  // 启动一个独立的线程来接收
ACK
    while (!Over) {
        // 根据拥塞窗口大小控制发送
        int effective_window = min((int)cwnd, MAX_CWND);  // 当前有效的窗口大小
        if (Next_Seq < Base_Seq + effective_window && Next_Seq <= Msg_Num + 1) {
            // 在有效窗口内发送数据
            if (Next_Seq == 1) {
                sendFileHeader(data_msg.get(), file_name);
            }
            else {
                sendFileData(data_msg.get(), file, Next_Seq, last_length);
            }
            Next_Seq++; // 增加序列号
        }

        // 流量控制和拥塞控制
        if (Next_Seq - Base_Seq > effective_window * 0.8) {
            this_thread::sleep_for(chrono::milliseconds(10)); // 适当休眠，避免过度
发送
        }
    }
    // 输出传输状态
    ackThread.join();
    return true;
}
```

在发送消息时，程序首先通过文件路径获取文件的大小和分块数量，然后根据拥塞窗口（`cwnd`）的大小来控制发送速率。如果当前序列号小于 `Base_Seq + cwnd`，则发送数据包。这里实现了拥塞控制的动态调整，在不同的阶段通过 `cwnd` 控制发送窗口。

4. 子线程处理 ACK 消息

当子线程收到有效的 ACK 消息时，会根据 ACK 更新窗口：

- 如果 ACK 与上次 ACK 相同，则说明发生了丢包，进入快速恢复。
- 否则，更新 `Base_Seq`，并根据 `cwnd` 状态调整拥塞窗口。

```
if (ack_msg.ack == last_ack) {
    handleDuplicateAck();
} else {
    duplicate_ack_count = 0;
    adjustcwnd();  // 调整拥塞窗口
    last_ack = ack_msg.ack;
}
```

ACK 子线程（`Thread_Ack`）负责接收接收端的 ACK 消息，adjustcwnd()根据收到的 ACK 更新拥塞窗口：

```
void adjustcwnd() {
    switch (congestion_state) {
        case SLOW_START:
            cwnd = 2 * cwnd;  // 慢启动阶段，cwnd 增长为原来的两倍
            if (cwnd >= ssthresh) {
                enterCongestionAvoidance(); // 进入拥塞避免阶段
            }
            break;
        case CONGESTION_AVOIDANCE:
            cwnd += 1.0; // 拥塞避免阶段，每经过一个 RTT 增加 1 MSS
            break;
        case FAST_RECOVERY:
            cwnd += 1.0;
            congestion_state = CONGESTION_AVOIDANCE;
            break;
    }
    cwnd = min(cwnd, (double)MAX_CWND); // 限制最大窗口大小
}
```

根据当前的拥塞控制状态，`cwnd` 可能会增加、减小或者保持不变。如果收到重复的 ACK（丢包的标志），会进入快速恢复阶段，并设置 `Resend` 为 `true`，要求重新发送丢失的数据包。

```
void handleDuplicateAck() {
    duplicate_ack_count++;
    if (duplicate_ack_count == 3) {
        enterFastRecovery(); // 收到 3 次重复 ACK，进入快速恢复
        Resend = true; // 标记重新发送丢包
    }
}
```

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

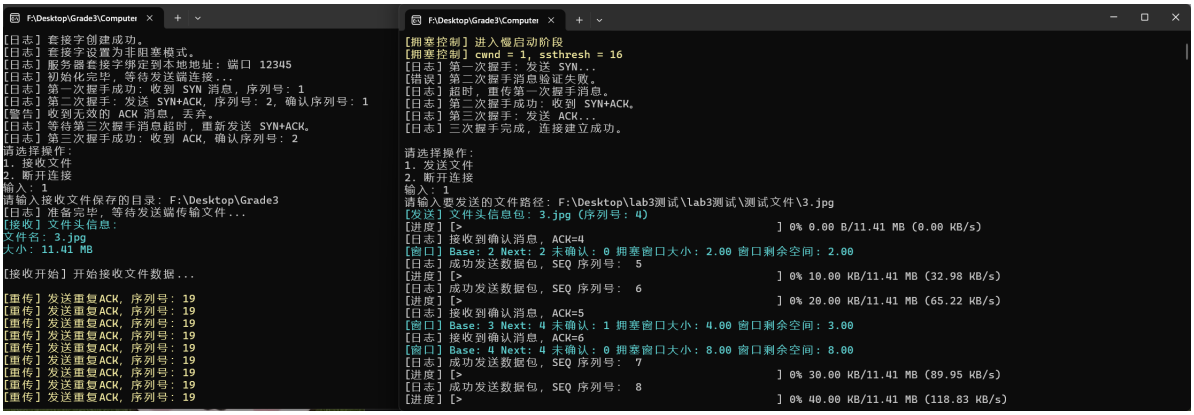
- 调用 `recvfrom` 接收客户端发送的 **ACK** 消息。
- 验证消息合法性：
  - 是否设置了 **ACK** 标志位。
  - 确认号是否匹配服务端发送的 **FIN** 消息序列号。
- 如果在超时时间内未收到 ACK，重传 **FIN** 消息。
- 重新计算校验和，确保重传消息完整。

断开连接代码部分与客户端高度重合，此处不做展示。

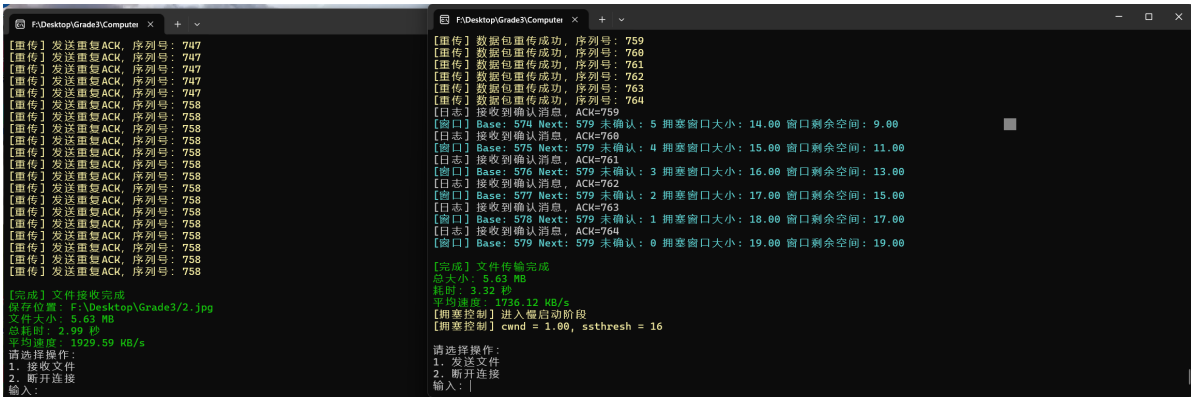
程序效果

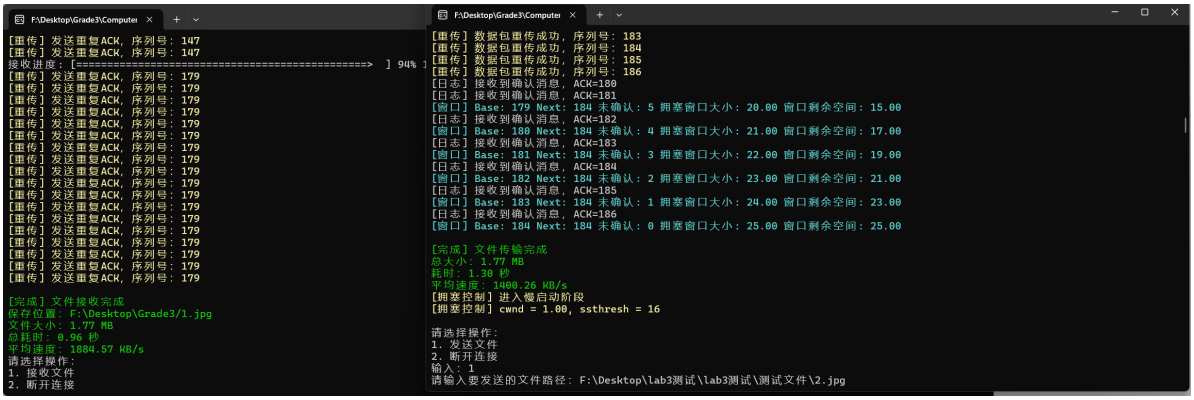
由于在测试过程中路由程序存在一定bug，因此本次选择使用本地直接连接来测试。

首先打开发送端和接收端程序，可以看到发送端一开始处于慢启动阶段：

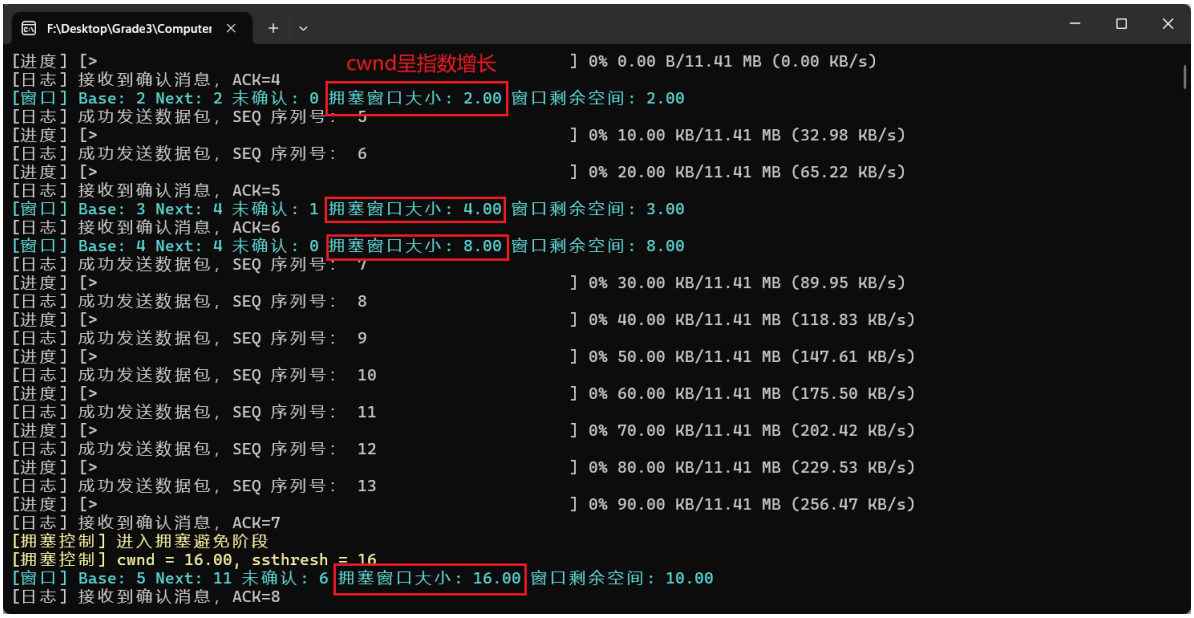


我们最终对几个测试文件进行了测试，可以发现最终成功传输并可以正常打开，最终的传输速率与固定大小的滑动窗口算法相比有了很大提升：

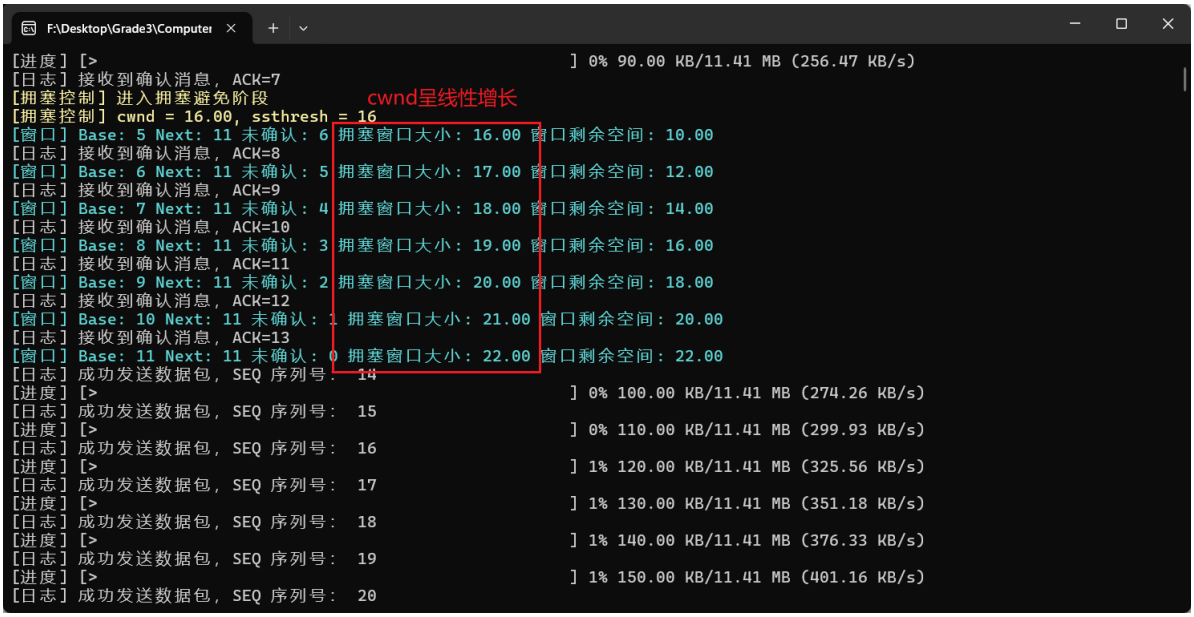




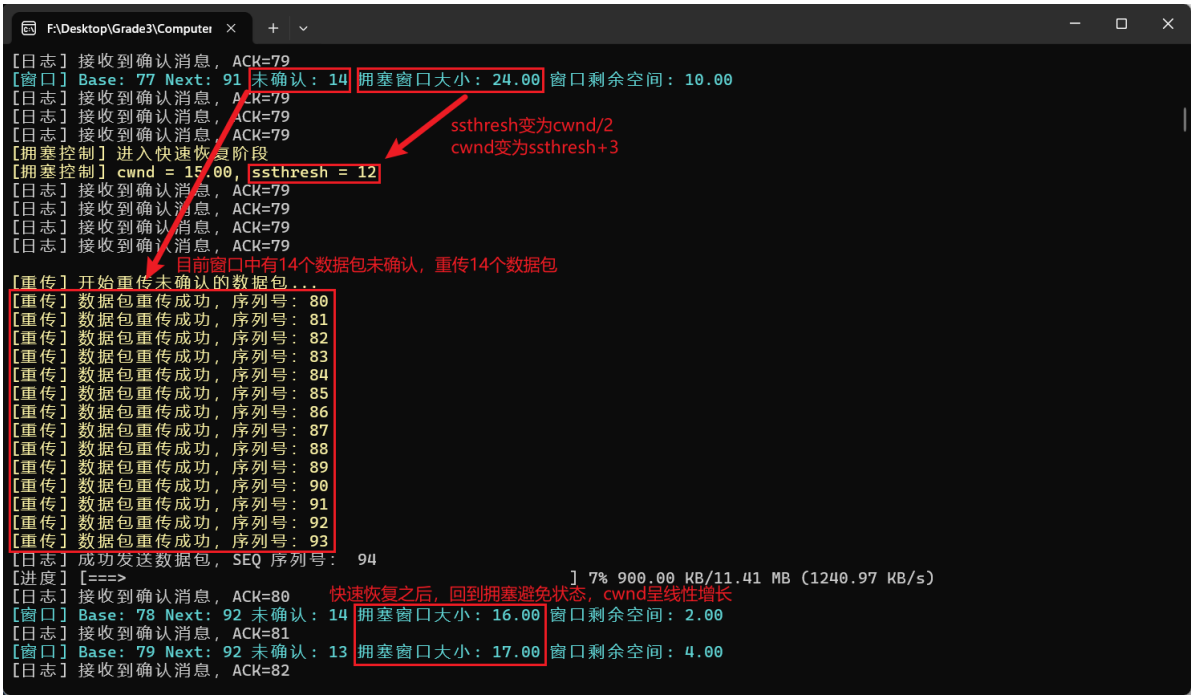
接下来我们查看发送端的窗口变化过程来更好的理解RENO算法。刚开始时发送端处于慢启动阶段，cwnd呈指数增长：



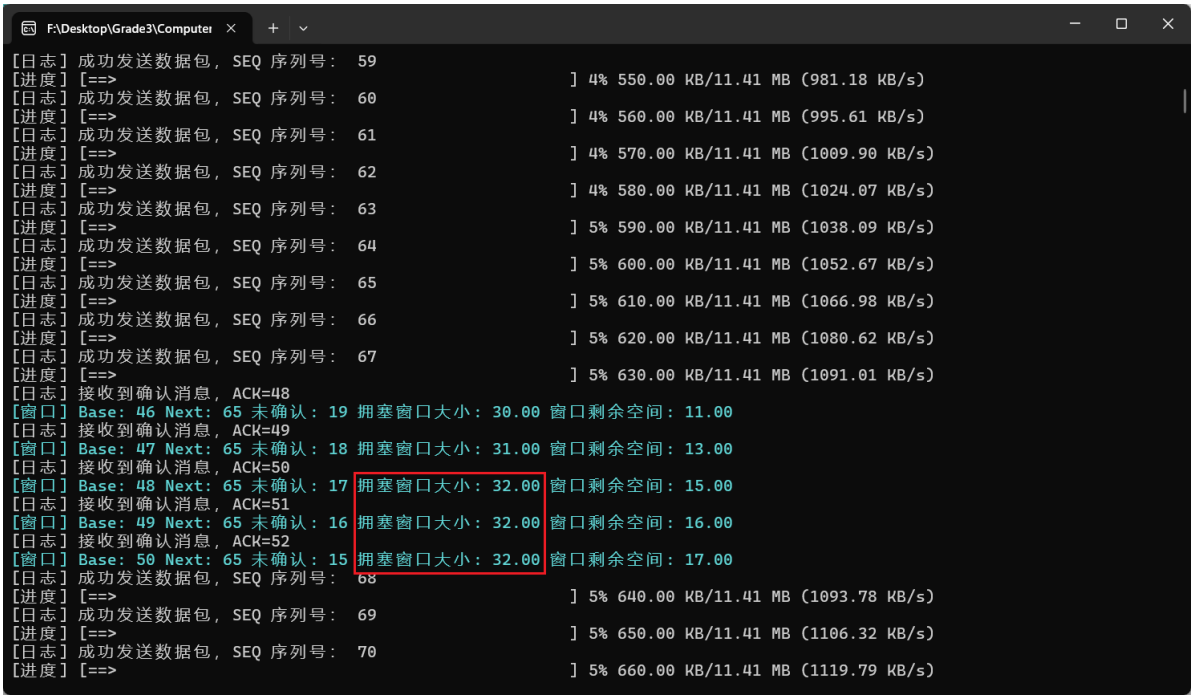
当cwnd达到慢启动阈值ssthresh之后，cwnd呈线性增长：



当发送端连续收到三个相同的确认号，证明接收端发生了乱序，需要进行重传数据包，此时发送端进入快速恢复阶段，ssthresh变为cwnd/2，cwnd变为ssthresh+3，快速恢复之后再次进入拥塞避免阶段，cwnd呈线性增长阶段：



我们还设置了cwnd的最大值，防止窗口无限增大，当达到这个最大值时，窗口大小会保持不变，直到下次状态切换：



至此我们将RENO算法所有的状态切换过程进行了讲解。

总结与反思

RENO算法的性能分析?

RTT 分析：

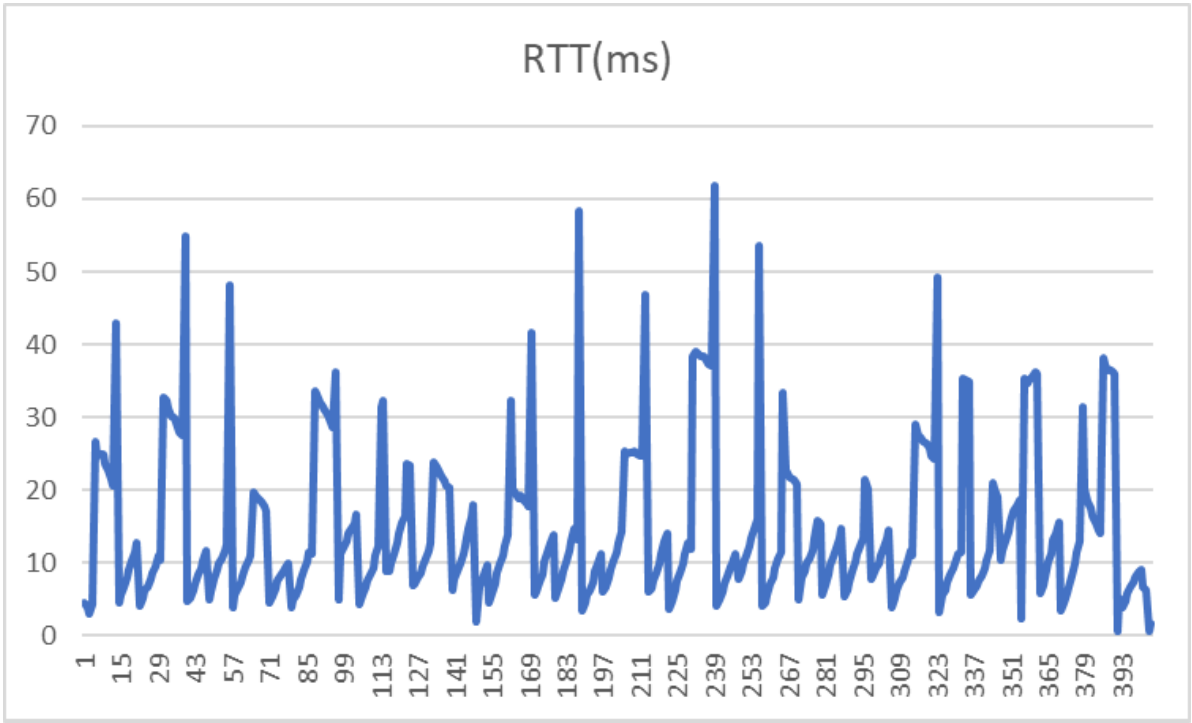
1. RTT 波动范围主要在 5-60ms 之间，平均值大约在 20ms 左右
2. 存在多个明显的峰值（约 60ms），这些可能是由网络拥塞导致的延迟增加
3. RTT 曲线呈现出锯齿状模式，这是 TCP RENO 算法的典型特征，反映了其拥塞控制机制的工作过程

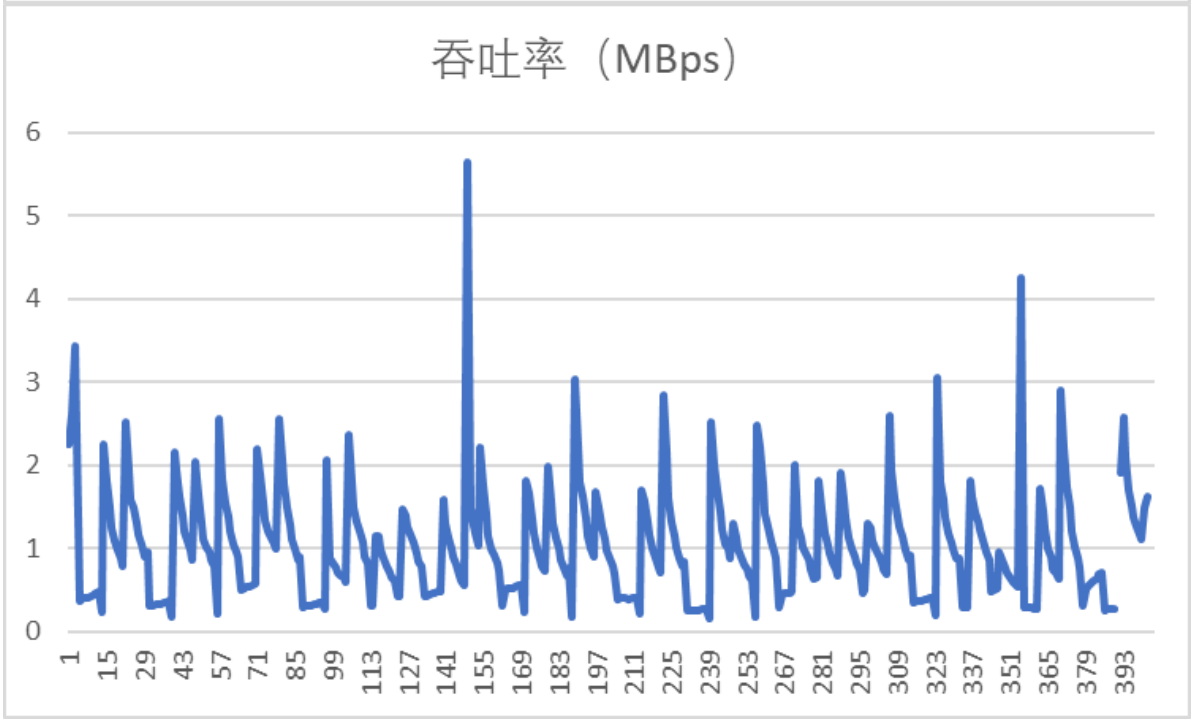
吞吐率分析：

1. 吞吐率大多在 0.5-3 Mbps 之间波动，最高达到约 5.5 Mbps
2. 吞吐率曲线也呈现出典型的锯齿形状，这与 RENO 的"加法增加，乘法减少"(AIMD)特性一致
3. 吞吐率的变化与 RTT 的变化存在明显的负相关性，当 RTT 出现峰值时，吞吐率往往会出现谷值

算法行为评估：

1. 拥塞响应：从图表可以看出，算法对网络拥塞的响应较为敏感，当检测到拥塞（RTT 增加）时，会适当降低吞吐率
2. 带宽利用：算法能够在网络条件较好时（RTT 较低）提高吞吐率，显示出对带宽的有效利用
3. 稳定性：虽然存在波动，但整体上维持在一个相对稳定的范围内，表明算法具有一定的稳定性





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。