计网Lab3-3 基于 UDP 服务设计可靠传输协议

姓名：刘玉菡

学号：2213244

专业：物联网工程

一、实验内容

在不可靠UDP通信基础上，实现了具有面向连接特性和可靠数据传输保障的自定义协议。该协议在设计上借鉴了TCP的基本思想，包括连接建立与释放的握手过程、在实验3_2的基础上实现一种拥塞控制算法，以及日志监控等要素。

二、协议设计

本实验采用UDP作为传输层协议，利用自定义数据包格式、序列号管理、校验和和确认机制实现面向连接的可靠数据传输。具体协议设计如下：

1. 数据包格式（网络层的数据包结构）

1. **以太网头部 (14 字节)**：用于指定网络中的源和目标设备的MAC地址。
2. **IP 头部 (20 字节)**：在网络层用来路由数据包的 IP 地址和相关信息。
3. **UDP 头部 (8 字节)**：传输层的协议头，主要负责在 IP 层和应用层之间传递数据。
4. **消息结构 (Message Structure)**：发送的数据部分，是在 UDP 负载中传输的数据，包括：`Seq` (4 字节)、`Ack` (4 字节)、`Flag` (2 字节)、`Length` (2 字节)、`Checksum` (2 字节)、`Data` (1024 字节)

   消息结构的大小为：**14 字节** (头部) + **1024 字节** (数据部分) = **1038 字节**

   下表展示了数据包（Message消息结构）的字段分布及其含义：

字段名	类型	长度（位）	含义
Seq	uint32_t	32	序列号，标识数据包的序列顺序，用于定位数据在传输中的位置。
Ack	uint32_t	32	确认号，指示已正确接收的最后一个数据包的下一个期望序列号。
Flag	uint16_t	16	标志位，指示数据包类型（SYN、ACK、FIN、DATA、FILENAME、CLOSE）。
Length	uint16_t	16	有效数据长度（Data字段中的字节数）。
Checksum	uint16_t	16	校验和，用于检验数据包头部及数据区的完整性。
Data	char[]	可变（定长 BUFFER_SIZE）	数据载荷，可能是文件名或文件数据内容，根据Flag类型而定。

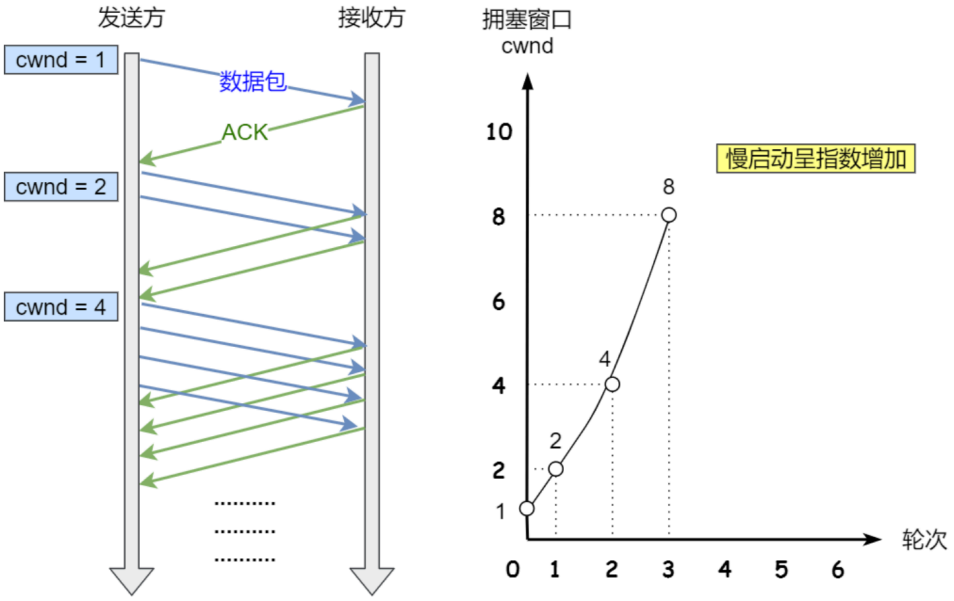
2. 基于拥塞控制算法的流量控制机制

拥塞控制算法用于动态调整发送数据的速率，以避免网络拥塞。该算法通过维护两个关键参数和四个阶段来实现：

- **拥塞窗口（cwnd）**：控制可以发送的未确认数据包的数量。
- **慢启动阈值（ssthresh）**：当拥塞窗口达到该阈值时，算法从慢启动阶段切换到拥塞避免阶段。
- 四个阶段：**慢启动（Slow Start）**、**拥塞避免（Congestion Avoidance）**、**快速重传（Fast Retransmit）** 和 **快速恢复（Fast Recovery）**。

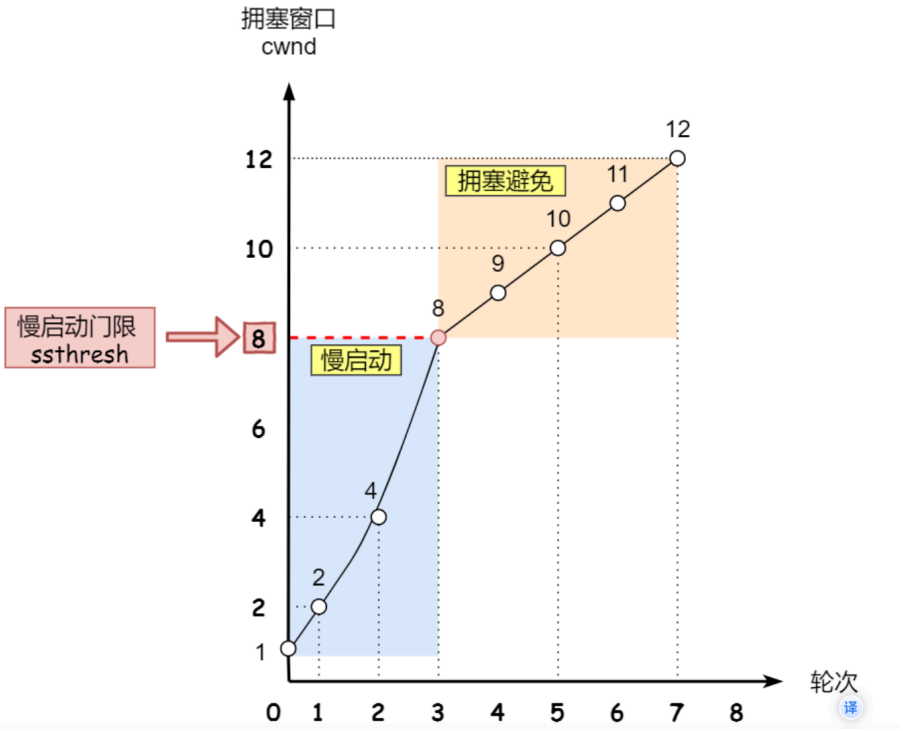
2.1 慢启动（Slow Start）

慢启动阶段的目标是快速增加拥塞窗口（cwnd），就是一点一点的提高发送数据包的量，以探索网络的可用带宽。在这个阶段，cwnd 每收到一个新的 ACK，就增加一个 MSS（最大报文段大小），即 cwnd 指数级增长，直到 cwnd 达到慢启动阈值（ssthresh）。



2.2 拥塞避免（Congestion Avoidance）

当拥塞窗口（cwnd）达到慢启动阈值（ssthresh）时，算法进入拥塞避免阶段。在这个阶段，cwnd 的增长速率变为线性增长，即每收到一个新的 ACK，`cwnd` 增加 `1 / cwnd`，从而避免过快增长导致网络拥塞。



所以，拥塞避免算法就是将原本慢启动算法的指数增长变成了线性增长，还是增长阶段，但是增长速度缓慢了一些。在一直增长之后，网络会慢慢进入了拥塞的状况，于是就会出现丢包现象，这时就需要对丢失的数据包进行重传。

当网络出现拥塞，也就是会发生数据包重传，重传机制主要有两种：

- 超时重传
- 快速重传

2.3 快速重传（Fast Retransmit）

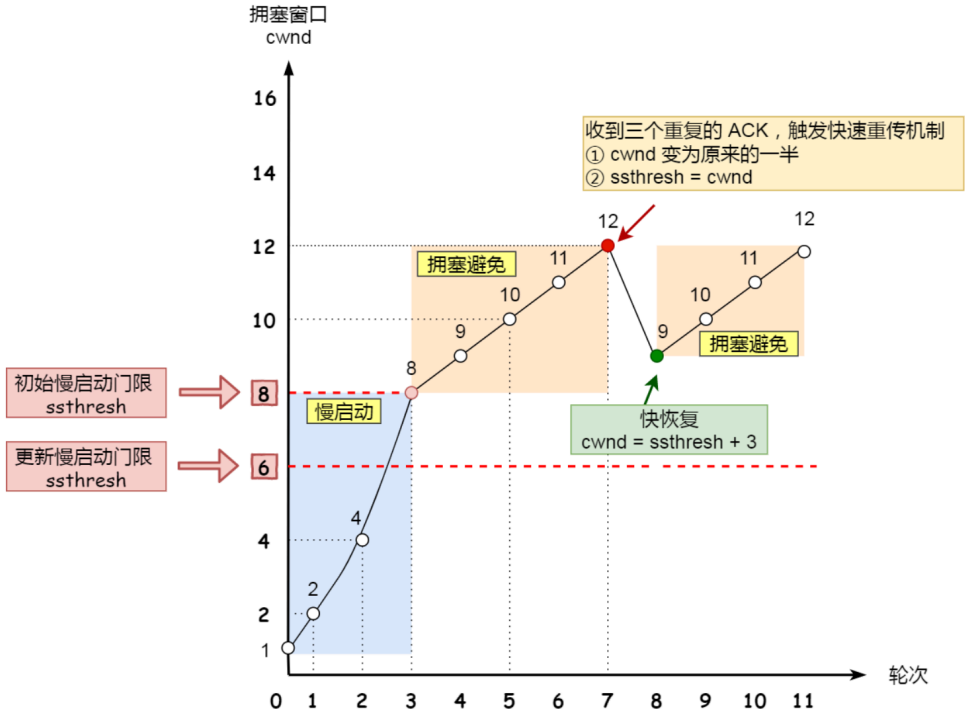
快速重传机制用于在检测到丢包时立即重传丢失的数据包，而无需等待重传定时器超时。具体来说，当接收方发现丢了一个中间包的时候，发送三次前一个包的ACK，于是发送端就会快速地重传，不必等待超时再重传。TCP 认为这种情况不严重，因为大部分没丢，只丢了一小部分，则 ssthresh 和 cwnd 变化为：cwnd = cwnd/2 ，也就是设置为原来的一半，ssthresh = cwnd ;

然后进入快速恢复算法。

2.4 快速恢复（Fast Recovery）

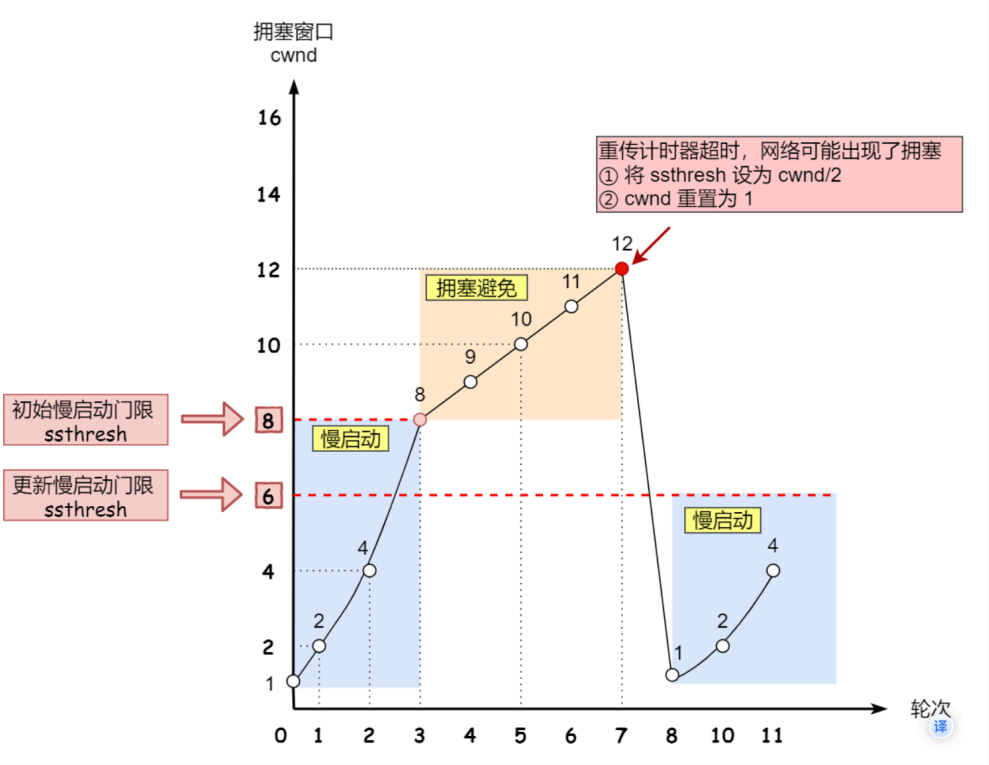
快速重传和快速恢复算法一般同时使用，快速恢复算法认为，还能收到 3 个重复 ACK 说明网络不那么糟糕，所以没必要像面对超时的反应那么强烈。前面所说，进入快速恢复之前， cwnd 和 ssthresh 已被更新了：cwnd = cwnd/2 ， ssthresh = cwnd ; 然后，进入快速恢复算法：拥塞窗口 cwnd = ssthresh + 3 （ 3 指确认有 3 个数据包被收到了）； 重传丢失的数据包；如果再收到重复的 ACK，那么 cwnd 增加 1。

如果收到了新数据的 ACK ， 就把 cwnd 设置为第一步中的 ssthresh 的值，因为该 ACK 确认了新的数据，说明从 duplicated ACK 时的数据都已收到，该恢复过程已经结束，可以回到恢复之前的状态了，也即再次进入拥塞避免状态。



2.5 超时重传

当发送的数据包在预定的超时时间内未收到ACK时，发送端认为该数据包丢失，触发重传。超时重传通常会触发更严格的拥塞控制措施，如将拥塞窗口（cwnd）重置为初始值（1），重新开始慢启动，以避免进一步的网络拥塞。



快速重传和超时重传的区别：

特性	快速重传（Fast Retransmit）	超时重传（Timeout Retransmit）
触发条件	收到三个连续的重复ACK	数据包在超时时间内未收到ACK
重传的数据包	仅重传丢失的特定数据包（基于重复ACK的序列号）	仅重传超时的数据包（根据定时器的序列号）

特性	快速重传（Fast Retransmit）	超时重传（Timeout Retransmit）
拥塞控制调整	将 `ssthresh` 设为 `cwnd / 2`，`cwnd` **设为** `ssthresh + 3`	将 `ssthresh` 设为 `cwnd / 2`，`cwnd` **重置为1.0**
恢复阶段	**进入快速恢复阶段**，逐步恢复 `cwnd`	重新**进入慢启动阶段**
发送窗口控制	基于快速恢复阶段的 `cwnd` 允许发送更多的数据包	基于 `cwnd` 重置为1.0，发送速率降低
适用场景	**数据包丢失但ACK仍在正常到达**（还是通路）	**数据包或ACK均可能丢失，或网络状况严重拥塞**

- **快速重传**和**超时重传**都采用了**选择性重传**的方式，即仅重传丢失的特定数据包，而不是整个发送窗口，减少不必要的重传，提升了传输性能。
- **快速重传**通过检测重复ACK，能够更快地响应数据包丢失，减少传输延迟，并通过进入快速恢复阶段逐步恢复发送速率。
- **超时重传**作为一种备用机制，确保即使在ACK丢失或乱序的情况下，数据包也能被可靠地传输，同时通过重置拥塞窗口来防止网络进一步拥塞。
- 通过使用 `sendBuffer` 记录已发送但未确认的数据包，并在检测到丢包时，仅重传相关的数据包，同时调整拥塞控制参数，确保传输的高效性和可靠性。

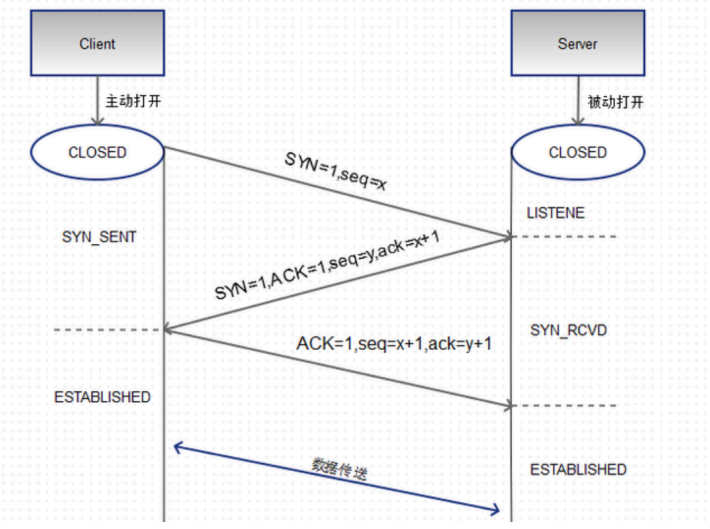
2.6 滑动窗口

滑动窗口初始化在完成三次握手建立连接后进行，此时 `baseSeq` 和 `nextSeq` 被设置为初始的序列号，用于跟踪未确认的数据包。拥塞窗口 `cwnd` 被初始化为1.0，表示最初只能发送一个数据包。**发送消息**的过程中，程序持续检查当前发送缓冲区 `sendBuffer` 中未确认的数据包数量是否小于 `cwnd`。如果是，则从文件中读取数据，创建数据包并发送，同时将这些数据包记录在 `sendBuffer` 中。每发送一个数据包，`nextSeq` 递增以准备下一个序列号。当接收到ACK时，程序根据ACK号滑动窗口，移除已确认的数据包，并根据拥塞控制算法动态调整 `cwnd`，以决定下一轮可以发送的数据包数量。

3. 连接管理

实验协议模拟TCP的三次握手和四次挥手过程：

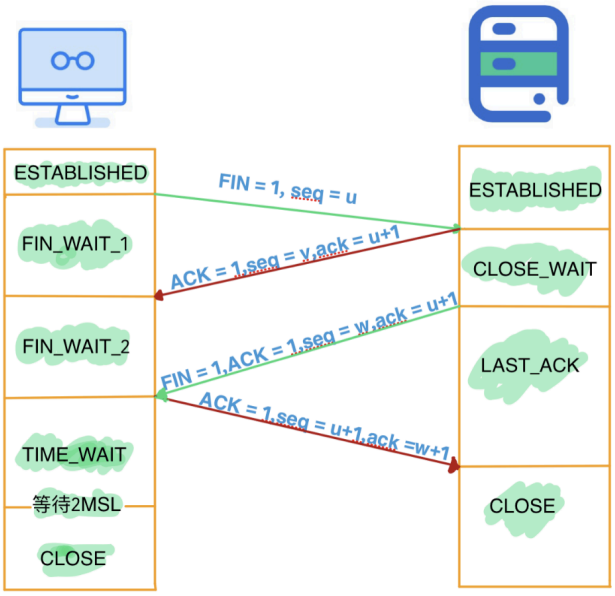
- **三次握手**：



1. **SYN**：客户端向服务器发送SYN包，要求建立连接。
2. **SYN-ACK**：服务器收到SYN包后，返回SYN-ACK包，表示同意建立连接。

3. **ACK**：客户端收到SYN-ACK包后，返回ACK包，完成连接的建立。

- **四次挥手**：



1. **FIN**：发送端发送FIN包，表示连接断开。
2. **ACK**：接收方返回ACK包，确认断开请求。
3. **FIN**：另一方也发送FIN包，表明同意断开连接。
4. **ACK**：最后一方返回ACK包，连接断开。

4. 差错检验

通过校验和机制（Checksum）保证数据的完整性：

1. 发送端在数据包发送前计算数据包内容的校验和，并将其写入Checksum字段。
2. 接收端在接收到数据包后重新计算校验和，若匹配则继续处理，否则丢弃该数据包。

校验和使用16位的“补码和”算法，遍历数据包的所有字节，对其进行累加并保留16位结果。

5. 日志输出

包括但不限于以下内容：

- 接收端时间戳
- 数据包的序列号（Seq）
- 数据包的确认号（Ack）
- 校验和（Checksum）
- 发送端窗口大小（cwnd）及发送状态
- 接收端的窗口大小，接收端期望序列号
- 数据包的标志位（Flag）
- 数据包的传输时间
- 吞吐率：文件传输速度，单位为KB/s

6. 错误处理

程序包括基本的错误处理机制。若在传输过程中出现错误（如超时重传次数超过限制、校验和错误等），程序将输出相应的错误信息，并终止当前操作。

三、代码实现部分

由于发送端和接收端的结构相似，接收端的实现部分省略，以下内容主要聚焦于发送端的实现。

1. 初始化和套接字创建

在程序开始时，初始化 Winsock 库以便使用网络功能，并创建一个 UDP 套接字。

- `initWinsock` 函数通过调用 `WSAStartup` 初始化 Windows 套接字 API。
- `createSocket` 函数创建一个 UDP 套接字用于数据报传输。

```
void initWinsock() {
    WSADATA wsaData;
    int result = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (result != 0) {
        cerr << "WSAStartup failed: " << result << endl;
        exit(1);
    }
}

SOCKET createSocket() {
    SOCKET sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sock == INVALID_SOCKET) {
        cerr << "Failed to create socket: " << WSAGetLastError() << endl;
        WSACleanup();
        exit(1);
    }
    return sock;
}
```

2. 消息结构体定义

为了组织数据包的结构，我们定义了一个 `Message` 结构体，包含了序列号（`Seq`）、确认号（`Ack`）、标志位（`Flag`）、数据长度（`Length`）、校验和（`Checksum`）和数据（`Data`）。该结构体的定义如下：

```
struct Message {
    uint32_t Seq;
    uint32_t Ack;
    uint16_t Flag;
    uint16_t Length;
    uint16_t Checksum;
    char Data[BUFFER_SIZE];
};
```

```
enum Flag {
    SYN = 1,
    ACK = 2,
    FIN = 4,
    DATA = 8,
    FILENAME = 16,
    CLOSE = 32
};
```

3.校验和计算

计算校验和的算法：16 位补码和

1. 将整个数据包（包括头部和数据部分）按 16 位（2 字节）为一组进行加和。
2. 如果加和结果超出了 16 位（即和大于 65535），则将溢出的部分加回到和的低位部分。
3. 最后，取和的反码（即所有位反转），得到校验和。

发送端会计算数据包的校验和，并将其写入消息结构体的 `Checksum` 字段。

```
uint16_t calculateChecksum(const char* data, int length) {
    uint32_t sum = 0;
    const uint16_t* ptr = reinterpret_cast<const uint16_t*>(data);

    while (length > 1) {
        sum += *ptr++;
        length -= 2;
    }

    if (length > 0) {
        uint16_t last_byte = 0;
        *reinterpret_cast<uint8_t*>(&last_byte) = *reinterpret_cast<const
uint8_t*>(ptr);
        sum += last_byte;
    }

    // 将 32 位的 sum 转换为 16 位
    while (sum >> 16) {
        sum = (sum & 0xFFFF) + (sum >> 16);
    }

    return static_cast<uint16_t>(~sum);
}
```

4. 发送消息

`sendMessage` 函数首先计算消息的校验和，将校验和字段置零，然后填充完整数据包，通过 `sendto` 函数将数据包发送给目标地址。发送完后，它会在控制台输出当前发送的数据包的信息。

```
void sendMessage(SOCKET sock, const sockaddr_in& destAddr, Message& msg) {
    msg.Checksum = 0;
    msg.Checksum = calculateChecksum(reinterpret_cast<char*>(&msg), sizeof(msg));
```

```
    int sendResult = sendto(sock, reinterpret_cast<char*>(&msg), sizeof(msg), 0,
        reinterpret_cast<const sockaddr*>(&destAddr), sizeof(destAddr));
    if (sendResult == SOCKET_ERROR) {
        cerr << "消息发送失败，错误码: " << WSAGetLastError() << endl;
        closesocket(sock);
        WSACleanup();
        exit(1);
    }

    cout << "发送数据包 - Seq:" << msg.Seq << " ,Ack: " << msg.Ack
        << " ,Checksum: " << msg.Checksum << " ,Flags: " << msg.Flag << endl;
}
```

4.1 慢启动（Slow Start）

```
if (cwnd < ssthresh) {
    cwnd += 1.0;
    cout << "慢启动： cwnd 增至 " << cwnd << endl;
}
```

- **判断阶段**：当`cwnd`小于`ssthresh`时，处于慢启动阶段。
- **增加拥塞窗口**：每收到一个新的ACK，`cwnd`增加1.0，实现指数级增长。
- **日志输出**：打印当前`cwnd`的值，便于监控。

假设初始`cwnd = 1`：

- 发送1个数据包，收到ACK后，`cwnd`增加到2。
- 发送2个数据包，收到ACK后，`cwnd`增加到3。
- 依此类推，`cwnd`呈指数级增长，直到达到`ssthresh`。

4.2 拥塞避免（Congestion Avoidance）

```
else {
    cwnd += 1.0 / cwnd;
    cout << "拥塞避免： cwnd 增至 " << cwnd << endl;
}
```

假设`ssthresh = 16`：

- 当`cwnd = 16`，收到一个ACK后，`cwnd`增加`1/16 = 0.0625`，变为`16.0625`。
- 再收到一个ACK，`cwnd`增加`1/16.0625 ≈ 0.0623`，变为`16.1248`。
- 依此类推，`cwnd`以线性方式缓慢增长。

4.3 快速重传（Fast Retransmit）

```
if (dupACKcount == 3 && !fastRecovery) {
    // 快速重传
    auto it = sendBuffer.find(ackMsg.Ack);
    if (it != sendBuffer.end()) {
        cout << "快速重传： 序列号 " << ackMsg.Ack << " 重传" << endl;
        sendMessage(it->second.msg);
```

```
        it->second.sentTime = chrono::steady_clock::now();
    }
    ssthresh = cwnd / 2;
    if (ssthresh < 1.0) ssthresh = 1.0;
    cwnd = ssthresh + 3;
    fastRecovery = true;
    cout << "快速重传： 进入快速恢复阶段, ssthresh = " << ssthresh << ", cwnd = " <<
cwnd << endl;
}
```

4.4 快速恢复（Fast Recovery）

在快速恢复阶段增加`cwnd`：

```
else if (dupACKcount > 3 && fastRecovery) {
    cwnd += 1.0;
    cout << "快速恢复：重复ACK增大，cwnd = " << cwnd << endl;
}
```

每收到一个重复ACK，`cwnd`增加1.0。

退出快速恢复阶段：

```
if (ackMsg.Flag & ACK) {
                if (ackMsg.Ack > lastAck) {
                    // 新的ACK
                    dupACKcount = 0;
                    if (fastRecovery) {
                        cout << "快速恢复： 收到新ACK，退出快速恢复阶段" << endl;
                        fastRecovery = false;
                        cwnd = ssthresh;
                    }
                    else {
                        // 慢启动或拥塞避免
                        if (cwnd < ssthresh) {
                            cwnd += 1.0;
                            cout << "慢启动： cwnd 增至 " << cwnd << endl;
                        }
                        else {
                            cwnd += 1.0 / cwnd;
                            cout << "拥塞避免： cwnd 增至 " << cwnd << endl;
                        }
                    }

                    // 移除已确认的包
                    for (uint32_t seq = baseSeq; seq < ackMsg.Ack; ++seq) {
                        sendBuffer.erase(seq);
                    }
                    baseSeq = ackMsg.Ack;
                    lastAck = ackMsg.Ack;
                }
```

4.5 超时处理

```
void adjustForTimeout(uint32_t seq) {
    cout << "超时！ 序列号 " << seq << " 超时，重传..." << endl;
    // 超时调整拥塞控制
    ssthresh = cwnd / 2;
    if (ssthresh < 1.0) ssthresh = 1.0;
    cwnd = 1.0;
    dupACKcount = 0;
    fastRecovery = false;
    cout << "超时处理： ssthresh = " << ssthresh << ", cwnd 重置为 " << cwnd <<
endl;
}
```

窗口滑动部分:

拥塞窗口 (`cwnd`) 控制了可以同时发送的未确认数据包的数量。

发送缓冲区 (`sendBuffer`) 存储了所有已发送但未确认的数据包，确保可以根据ACK来滑动窗口并进行必要的重传。

序列号管理 (`baseSeq` 和 `nextSeq`) 跟踪窗口的起始和结束位置，确保数据包的有序传输和确认。

发送窗口控制条件 `(sendBuffer.size() < (size_t)ceil(cwnd))` 确保发送端不会超过当前拥塞窗口允许的未确认数据包数量，从而有效地管理窗口的滑动。

数据包的发送:

```
while (connected && !fileSent && (sendBuffer.size() < (size_t)ceil(cwnd))) {
    // 读取文件数据并发送
    // 添加到 sendBuffer
    sendMessage(dataMsg);
    nextSeq++;
    totalBytesSent += bytesRead;
}
```

接收ACK并滑动窗口:

```
if (ackMsg.Ack > lastAck) {
    // 更新 cwnd
    // 移除已确认的数据包
    for (uint32_t seq = baseSeq; seq < ackMsg.Ack; ++seq) {
        sendBuffer.erase(seq);
    }
    baseSeq = ackMsg.Ack;
    lastAck = ackMsg.Ack;
}
```

5.超时检测并重传超时的数据包

```
void checkTimeouts() {
    auto currentTime = chrono::steady_clock::now();
    for (auto& [seq, pkt] : sendBuffer) {
        auto duration = chrono::duration_cast<chrono::milliseconds>(currentTime -
pkt.sentTime).count();
        if (duration > TIMEOUT_MS) {
            if (pkt.retransmissions < MAX_RETRANSMISSIONS) {
                adjustForTimeout(seq);
                sendMessage(pkt.msg);
                pkt.sentTime = chrono::steady_clock::now();
                pkt.retransmissions++;
            }
            else {
                cerr << "序列号 " << seq << " 达到最大重传次数，传输失败。" << endl;
                connected = false;
                break;
            }
        }
    }
}
```

6. 建立连接（三次握手）

（1）**发送 SYN 请求**：客户端发送带有 `SYN` 标志的数据包，表示请求建立连接。此时 `Seq` 设置为 1，`Flag` 设置为 `SYN`。

（2）**接收 SYN-ACK 响应**：服务器收到请求后，返回一个带有 `SYN` 和 `ACK` 标志的数据包，`Seq` 和 `Ack` 分别为 1 和 2。

（3）**发送 ACK 确认**：客户端收到 `SYN-ACK` 后，发送带有 `ACK` 标志的数据包，确认连接已建立。此时 `Seq` 和 `Ack` 分别为 2 和 2。

以下是发送端的实现：

```
// 开始连接建立（三次握手）
Message msg = {};
msg.Seq = 1;  // 初始序列号
msg.Flag = SYN;

cout << "尝试连接服务器..." << endl;
sendMessage(clientSocket, destAddr, msg);

// 等待服务器的 SYN-ACK
sockaddr_in from;
int fromSize = sizeof(from);
int bytesReceived = recvfrom(clientSocket, reinterpret_cast<char*>(&msg),
sizeof(msg), 0,
    reinterpret_cast<sockaddr*>(&from), &fromSize);
if (bytesReceived > 0 && (msg.Flag & (SYN | ACK))) {
    // 验证校验和
    uint16_t receivedChecksum = msg.Checksum;
```

```
    msg.Checksum = 0;
    uint16_t calculatedChecksum = calculateChecksum(reinterpret_cast<char*>
(&msg), sizeof(msg));
    if (receivedChecksum != calculatedChecksum) {
        cerr << "收到的SYN-ACK校验和不匹配。" << endl;
        closesocket(clientSocket);
        WSACleanup();
        return 1;
    }

    cout << "收到 SYN-ACK，发送 ACK..." << endl;
    msg.Flag = ACK;
    sendMessage(clientSocket, destAddr, msg);
    cout << "三次握手成功。" << endl;
}
else {
    cerr << "连接建立过程中出错。" << endl;
    closesocket(clientSocket);
    WSACleanup();
    return 1;
}
```

7. 连接关闭（四次挥手）

（1）**发送 FIN 包**：客户端发送带有 `FIN` 标志的数据包，请求关闭连接。此时 `Seq` 设置为当前的序列号，`Flag` 设置为 `FIN`。

（2）**接收 FIN-ACK 包**：服务器接收到 `FIN` 包后，发送一个带有 `FIN` 和 `ACK` 标志的数据包，确认关闭连接。

（3）**服务器发送 FIN 包**：服务器发送带有 `FIN` 标志的数据包，表示服务器也希望关闭连接。

（4）**客户端接收并确认**：客户端收到服务器的 `FIN` 包后，发送一个带有 `ACK` 标志的数据包，确认连接关闭。

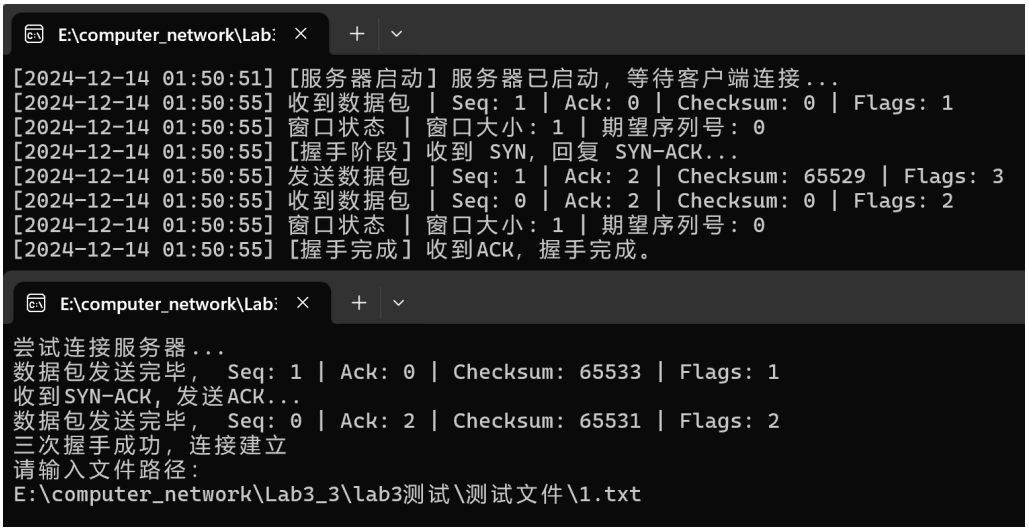
```
        else if (choice == 2) {
            // 发送断开连接的 CLOSE 消息
            cout << "正在关闭连接..." << endl;
            msg.Flag = CLOSE;
            msg.Seq = 0;
            msg.Ack = 0;
            msg.Length = 0;

            sendMessage(clientSocket, destAddr, msg);

            // 接收服务器的 ACK 确认
            int bytesReceived = recvfrom(clientSocket, reinterpret_cast<char*>
(&msg), sizeof(msg), 0,
                reinterpret_cast<sockaddr*>(&from), &fromSize);

            if (bytesReceived > 0) {
                // 验证校验和
                uint16_t receivedChecksum = msg.Checksum;
                msg.Checksum = 0;
```

```
                uint16_t calculatedChecksum =
calculateChecksum(reinterpret_cast<char*>(&msg), sizeof(msg));
                if (receivedChecksum != calculatedChecksum) {
                    cerr << "收到的消息校验和不匹配，丢弃数据。" << endl;
                }
                else if (msg.Flag & ACK) {
                    cout << "收到服务器的 ACK 确认。" << endl;

                    // 等待服务器发送 FIN
                    bytesReceived = recvfrom(clientSocket,
reinterpret_cast<char*>(&msg), sizeof(msg), 0,
                        reinterpret_cast<sockaddr*>(&from), &fromSize);

                    if (bytesReceived > 0) {
                        // 验证校验和
                        receivedChecksum = msg.Checksum;
                        msg.Checksum = 0;
                        calculatedChecksum =
calculateChecksum(reinterpret_cast<char*>(&msg), sizeof(msg));
                        if (receivedChecksum != calculatedChecksum) {
                            cerr << "收到的消息校验和不匹配，丢弃数据。" << endl;
                        }
                        else if (msg.Flag & FIN) {
                            cout << "收到服务器的 FIN，发送 ACK 确认..." << endl;

                            // 发送 ACK 确认
                            Message ackMsg = {};
                            ackMsg.Seq = 0;
                            ackMsg.Ack = msg.Seq + 1;
                            ackMsg.Flag = ACK;

                            sendMessage(clientSocket, serverAddr, ackMsg);

                            cout << "四次挥手完成，连接已关闭。" << endl;
                            connected = false;  // 退出循环，结束程序
                        }
                        else {
                            cerr << "未收到服务器的 FIN 消息。" << endl;
                        }
                    }
                    else {
                        cerr << "未收到服务器的 FIN 消息。" << endl;
                    }
                }
                else {
                    cerr << "未收到服务器的 ACK 确认。" << endl;
                }
            }
            else {
                cerr << "未收到服务器的响应。" << endl;
            }
        }
```

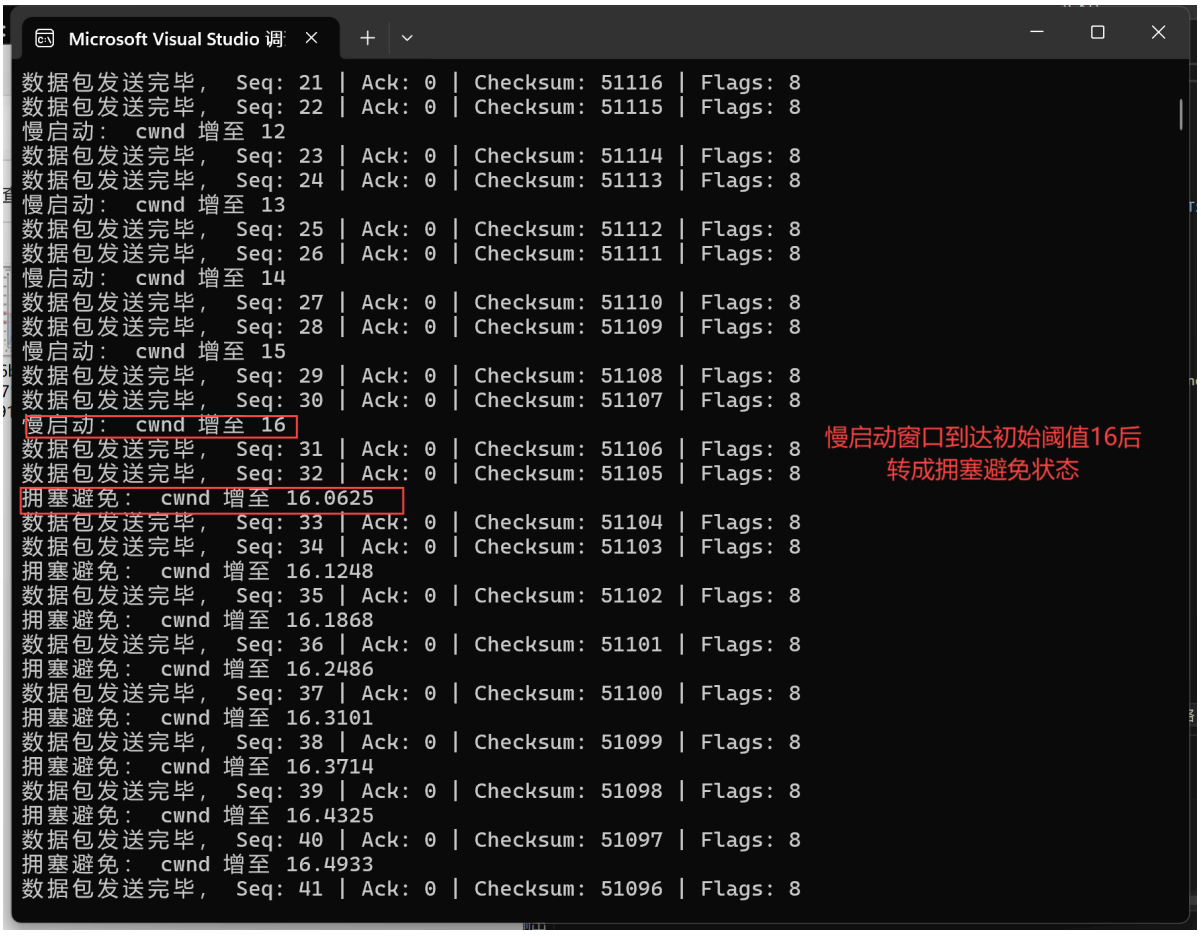
四、传输测试

首先设置好路由器信息，设置好发送端与接收端信息，开始传输文件，此处为不丢包和不延时的正常传输。

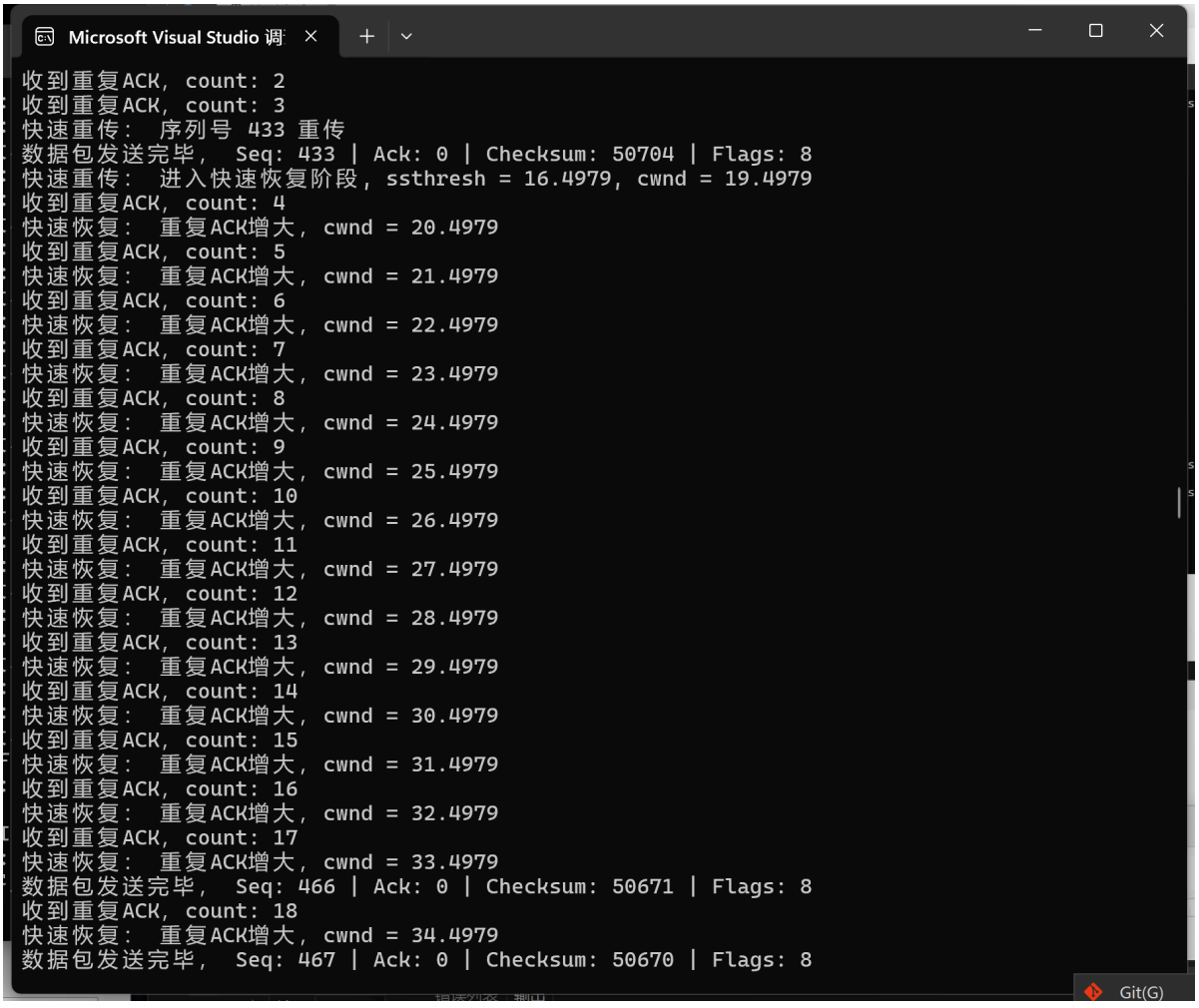
然后打开接受端，在无发送端开机时，接收端显示等待连接。然后打开发送端，自动与接收端进行三次握手，然后按照发送端提示输入文件路径，以传输大小为1.57 MB (1,655,808 字节）的1.txt文本文件为例。



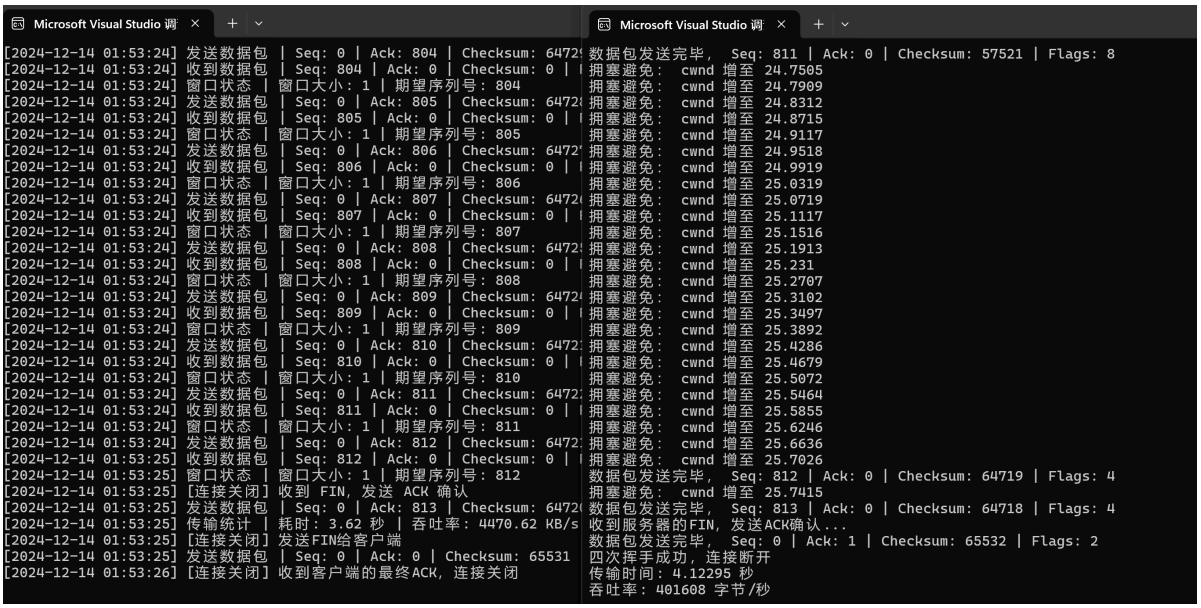
按回车，传输时可以看到具体的数据包传输信息，接收端窗口大小为1。



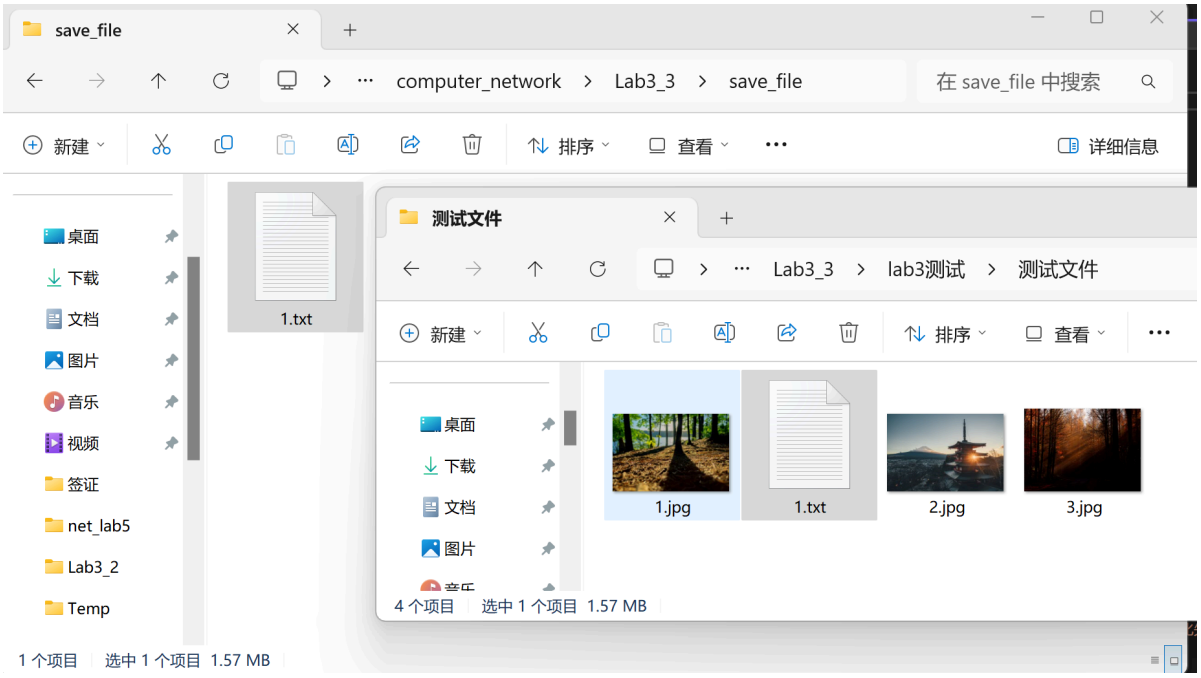
收到重复ACK后状态变为快速重传，再变为快速恢复，直到收到新的ACK。



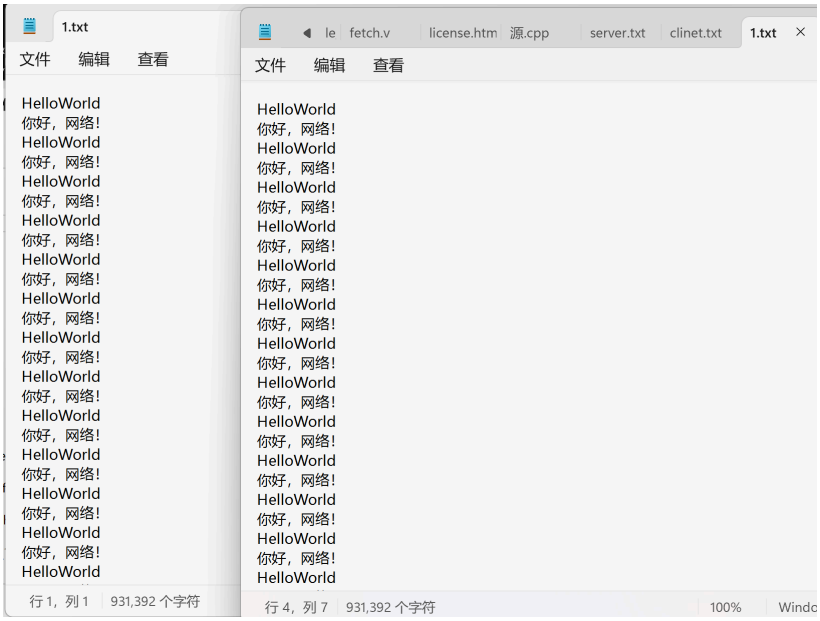
传输完成后显示文件发送成功，传输时间为3.62秒，吞吐率为4470.62KB/S。



在接收端保存的路径下，可以看到的确收到了大小完整、属性相同的文本文件。



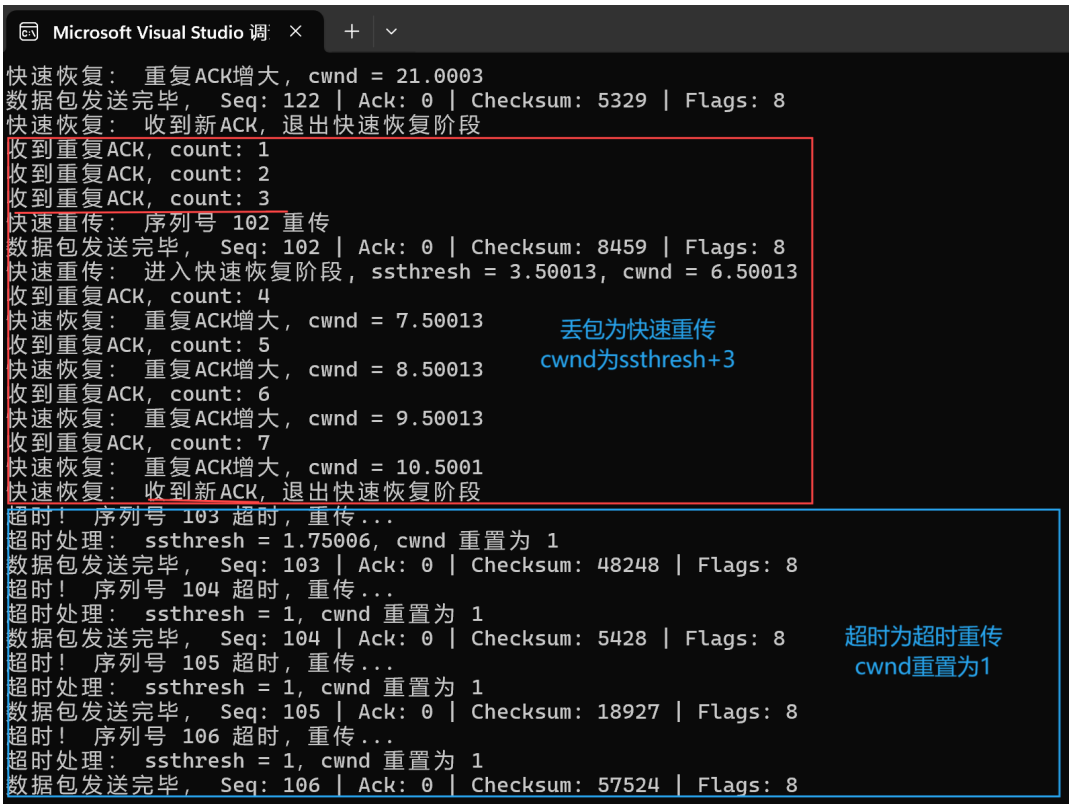
检查文本内容可以正常输出中英文。



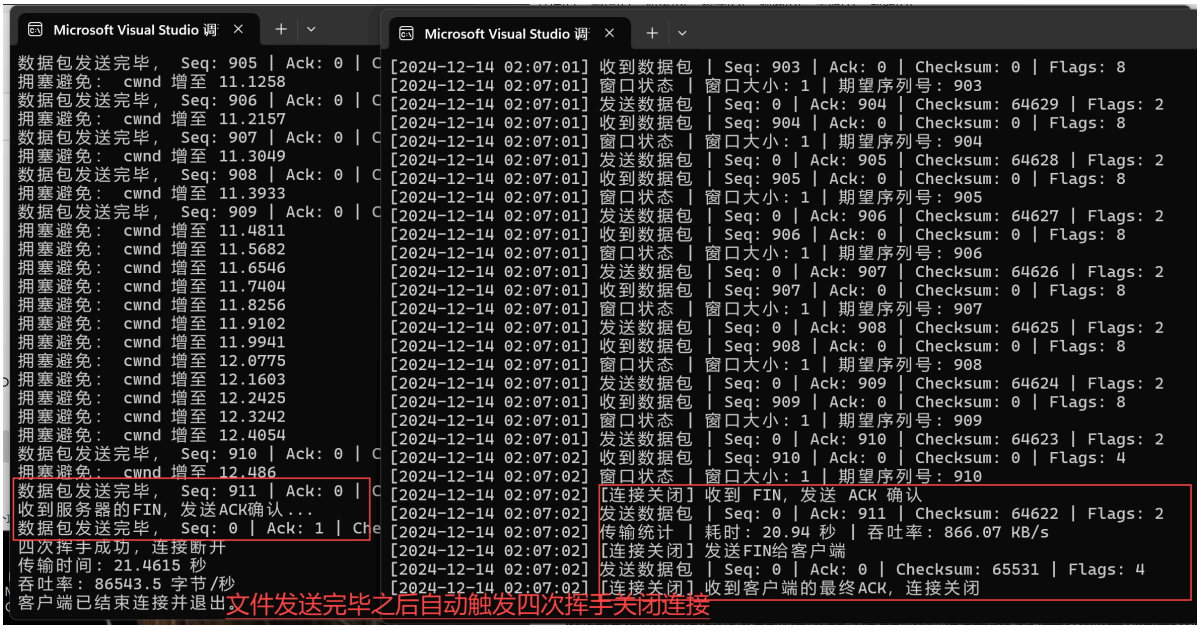
将路由器丢包率改为3%，延时改为3ms，重新进行测试。

以传输大小为1.77 MB的1.jpg文本文件为例。

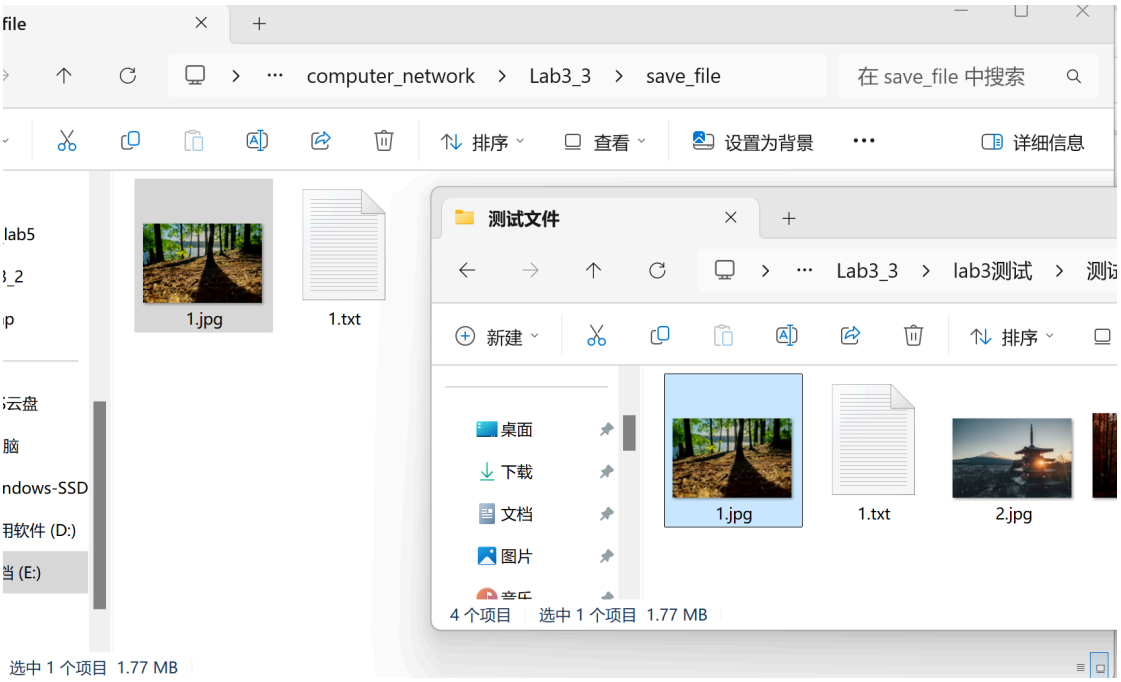
由于设置了丢包和延时，会分别进行快速重传和超时重传



20.94s后该1.jpg文本文档传输完毕，吞吐率为866.07kb/s.



检查图片传输正常清晰，且大小属性一致。



一般文件传输完之后自动触发四次挥手断开连接，如上上图。

五、问题反思

问题：如果cwnd是8.125，那是最多发8个数据包吗？

`cwnd` 的值是 `8.125` 表示在理想情况下，**可以发送8个完整的数据包**，并且理论上还有空间发送第9个数据包的部分数据（即 `0.125` 的数据包）。是因为拥塞窗口（`cwnd`）是浮动的，在计算 `sendBuffer.size() < ceil(cwnd)` 时，`cwnd` 的值会被向上取整为 `9`。但实际发送时，我的代码并没有考虑"部分数据包"的发送机制。即使 `cwnd = 8.125`，发送端仍然会发送8个完整数据包，而不会尝试发送第9个部分数据包。

感想：

在这次实验中，我深入理解了滑动窗口机制和拥塞控制算法如何配合起来进行流量控制。通过编写和调试协议的核心部分，我不仅掌握了如何利用拥塞窗口（`cwnd`）和慢启动、拥塞避免、快速重传等策略来提高网络传输效率，还对数据包的发送、确认和重传机制有了更为直观的理解。

目前每个数据包都包含完整的头部和数据，未来可以考虑根据需要优化数据包大小和传输效率。比如可以为每个数据包引入更简洁的控制信息，或根据实际需要调整每个数据包的大小，进一步提升吞吐率。