计网Lab3-1 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

本次实验旨在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，实验包括以下主要功能：

1. **建立连接**：实现连接的建立机制，类似于 TCP 协议中的三次握手。
2. **差错检测**：仿照 UDP 的校验和机制，确保数据传输的可靠性。
3. **接收确认**：每个数据包的接收都需要通过确认报文 ACK 来反馈。
4. **超时重传**：为了应对数据包丢失和失序问题，在发送方实现超时重传机制。
5. **流量控制**：采用停等机制来进行流量控制，保证数据包按顺序可靠地传输。

通过实现上述功能，确保给定测试文件能够可靠传输，并且能够应对丢包、失序和延时等网络异常情况。

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)

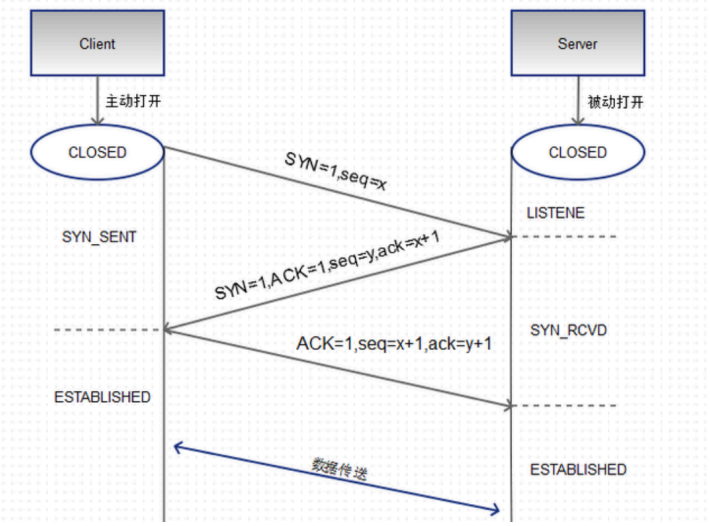
   消息结构的大小为：**14 字节** (头部) + **1024 字节** (数据部分) = **1038 字节**

部分	大小 (字节)	内容
以太网头部	14	目标 MAC 地址 (6 字节) + 源 MAC 地址 (6 字节) + 类型 (2 字节)
IP 头部	20	包括源地址、目标地址、TTL 等
UDP 头部	8	源端口、目标端口、数据长度和校验和
消息结构 (Message Structure)	1038	`Message` 结构体，包含序列号、确认号等

2. 连接管理

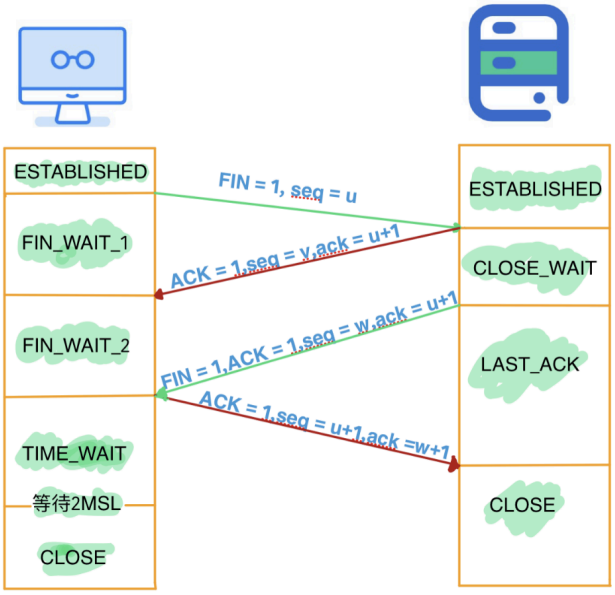
实验协议模拟TCP的三次握手和四次挥手过程：

- **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。

3. 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。

4. 数据传输

数据传输采用停等机制：

1. 发送端每发送一个数据包后，必须等待接收端的ACK确认。如果在指定时间内未收到ACK，则重新发送该数据包。
2. 接收端在收到数据包后，验证其校验和是否正确，并返回ACK确认数据包的接收情况。

为了防止丢包和超时问题，发送端采用重传机制，如果发送的包未能在规定时间内收到确认，发送端会重传数据包，最多尝试5次。

5. 文件传输过程

1. **文件名传输**：首先，发送端传输文件名给接收端。文件名传输采用数据包格式中的DATA字段，发送文件名作为数据内容。
2. **文件内容传输**：发送端通过数据包逐块传输文件内容，每块的大小为1024字节。每发送一块数据，发送端都会等待接收端的ACK确认，确认后再发送下一块。
3. **文件传输完成**：文件传输完成后，发送端发送FIN包，表示数据传输结束。

6. 超时重传

在文件传输过程中，若发送端未在设定的超时时间（我设置为1秒）内接收到接收端的ACK，则会触发重传机制。每个数据包的重传次数限制为5次，若5次重传均未成功，发送端将终止传输，直接退出。

7. 日志输出

包括但不限于以下内容：

- 数据包的序列号（Seq）
- 数据包的确认号（Ack）
- 校验和（Checksum）
- 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

8. 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];
};
```

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; // 在计算前将校验和字段设为 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 << "发送失败，错误码: " << WSAGetLastError() << endl;
        closesocket(sock);
        WSACleanup();
        exit(1);
    }
```

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

5.超时重传

`sendWithTimeout` 函数使用 `select` 函数来检测是否有接收到的 ACK。如果接收超时（`select` 返回 0），则重传数据包。如果接收到正确的 ACK（校验和匹配，`Flag` 包含 ACK 且确认号等于 `Seq+1`），则认为数据传输成功。在主函数中我将超时时间设置为1秒，最大重传次数设置成5次。

```
int sendWithTimeout(SOCKET sock, const sockaddr_in& destAddr, Message& msg,
uint32_t expectedAck, int timeoutMs, int maxRetransmissions) {
    int retransmissions = 0;
    fd_set readfds;
    timeval timeout;
    sockaddr_in from;
    int fromSize = sizeof(from);
    Message recvMsg;

    while (retransmissions < maxRetransmissions) {
        // 发送消息
        sendMessage(sock, destAddr, msg);

        // 设置超时时间
        FD_ZERO(&readfds);
        FD_SET(sock, &readfds);
        timeout.tv_sec = timeoutMs / 1000;
        timeout.tv_usec = (timeoutMs % 1000) * 1000;

        // 等待ACK
        int selectResult = select(0, &readfds, NULL, NULL, &timeout);
        if (selectResult > 0) {
            // 接收ACK
            int bytesReceived = recvfrom(sock, reinterpret_cast<char*>(&recvMsg),
sizeof(recvMsg), 0,
                reinterpret_cast<sockaddr*>(&from), &fromSize);

            if (bytesReceived > 0) {
                // 验证校验和
                uint16_t receivedChecksum = recvMsg.Checksum;
                recvMsg.Checksum = 0;
                uint16_t calculatedChecksum =
calculateChecksum(reinterpret_cast<char*>(&recvMsg), sizeof(recvMsg));
                if (receivedChecksum != calculatedChecksum) {
                    cerr << "收到的ACK校验和不匹配，丢弃数据。" << endl;
                    continue;
                }

                if ((recvMsg.Flag & ACK) && recvMsg.Ack == msg.Seq + 1) {
                    // 收到正确的ACK

                    // 日志输出
```

```
                    cout << "收到ACK - Seq: " << recvMsg.Seq << ", Ack: " <<
recvMsg.Ack
                         << ", Checksum: " << receivedChecksum << endl;

                    return 0;
                }
            }
        }
        else if (selectResult == 0) {
            // 超时，重传
            retransmissions++;
            cout << "超时，正在重传数据包 (" << retransmissions << "/" <<
maxRetransmissions << ")" << endl;
        }
        else {
            // 发生错误
            cerr << "Select函数出错: " << WSAGetLastError() << endl;
            return -1;
        }
    }

    // 超过最大重传次数
    cerr << "超过最大重传次数，未收到ACK确认。" << endl;
    return -1;
}
```

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。

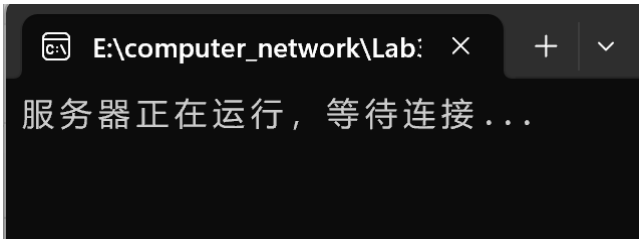
```
        else if (choice == 2) {
            // 发送断开连接的 CLOSE 消息
            cout << "正在关闭连接..." << endl;
            msg.Flag = CLOSE;
            msg.Seq = 0;
            msg.Ack = 0;
            msg.Length = 0;

            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

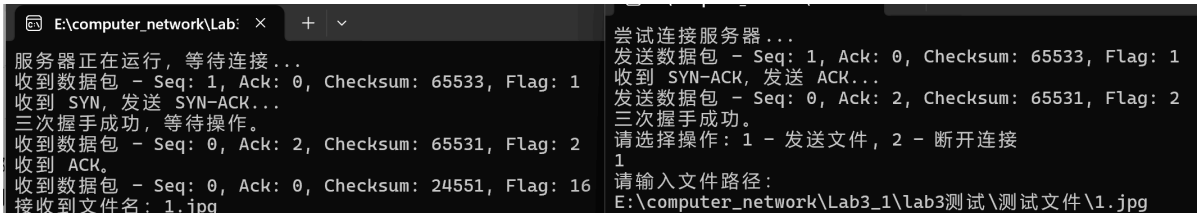
首先设置好路由器信息，设置好发生端与接收端信息，开始传输文件，此处为不丢包和不延时的正常传输。

然后打开接受端，在无发送端开机时，显示等待连接。

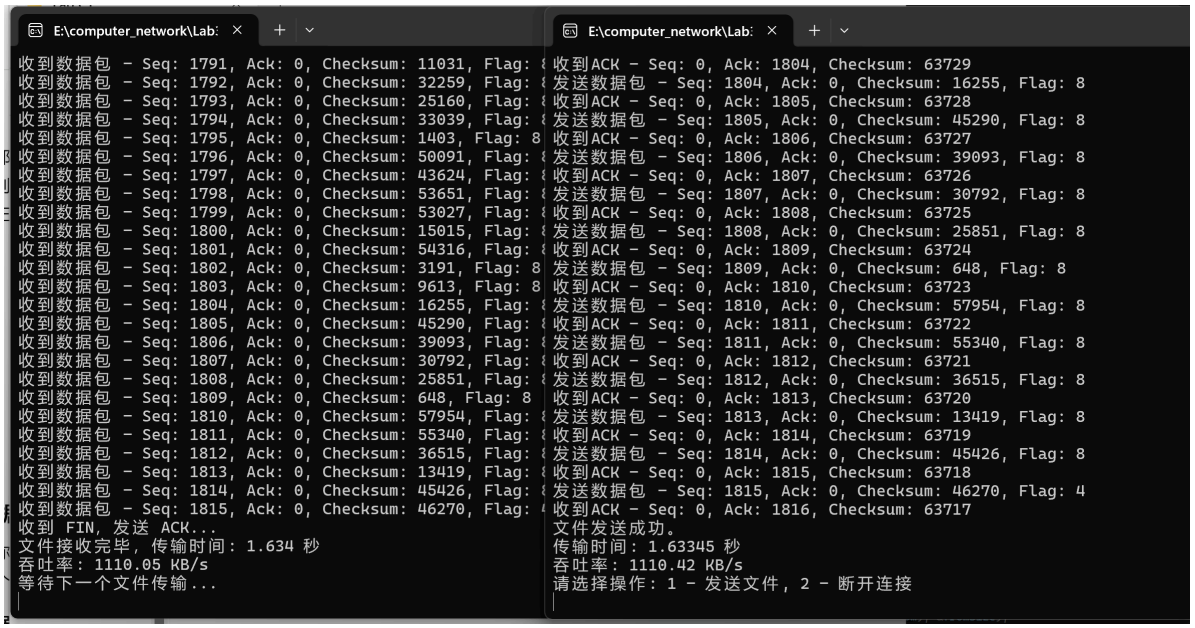


打开发送端，自动与接收端进行三次握手，然后输入1选择发送文件，

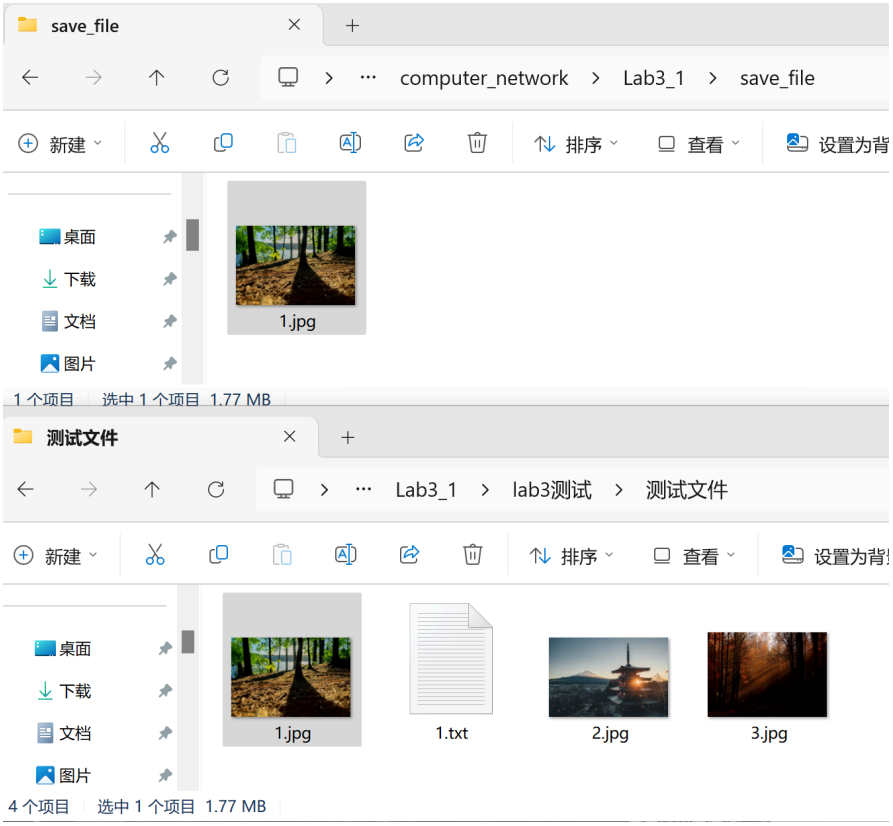
接着按照发送端提示输入文件路径，这里以1,857,353 字节的1.jpg为例



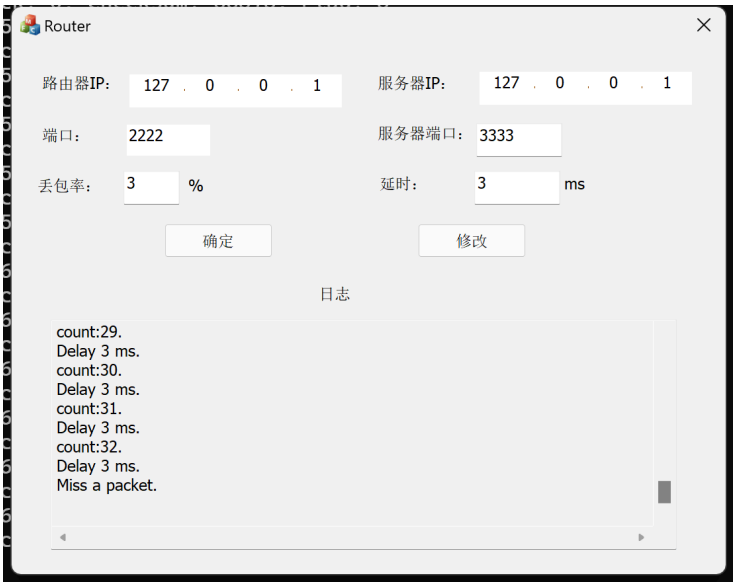
按回车，传输时可以看到具体的数据包传输信息，传输完成后显示文件发送成功，传输时间为1.63345秒，吞吐率为1110.42KB/S。



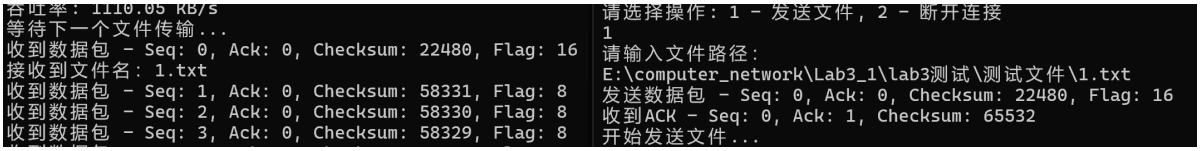
在接收端保存的路径下，可以看到的确收到了大小完整、属性相同的图片文件。



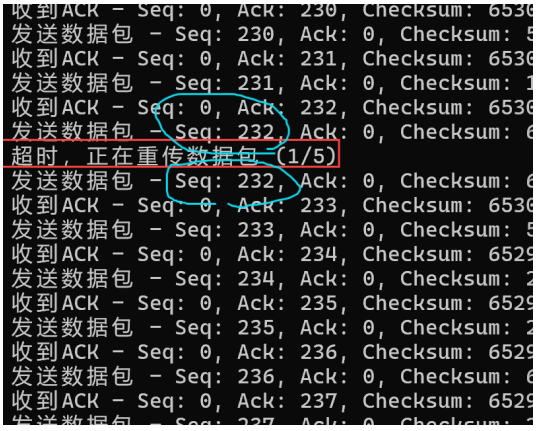
将路由器丢包率改为3%，延时改为3ms，重新进行测试。



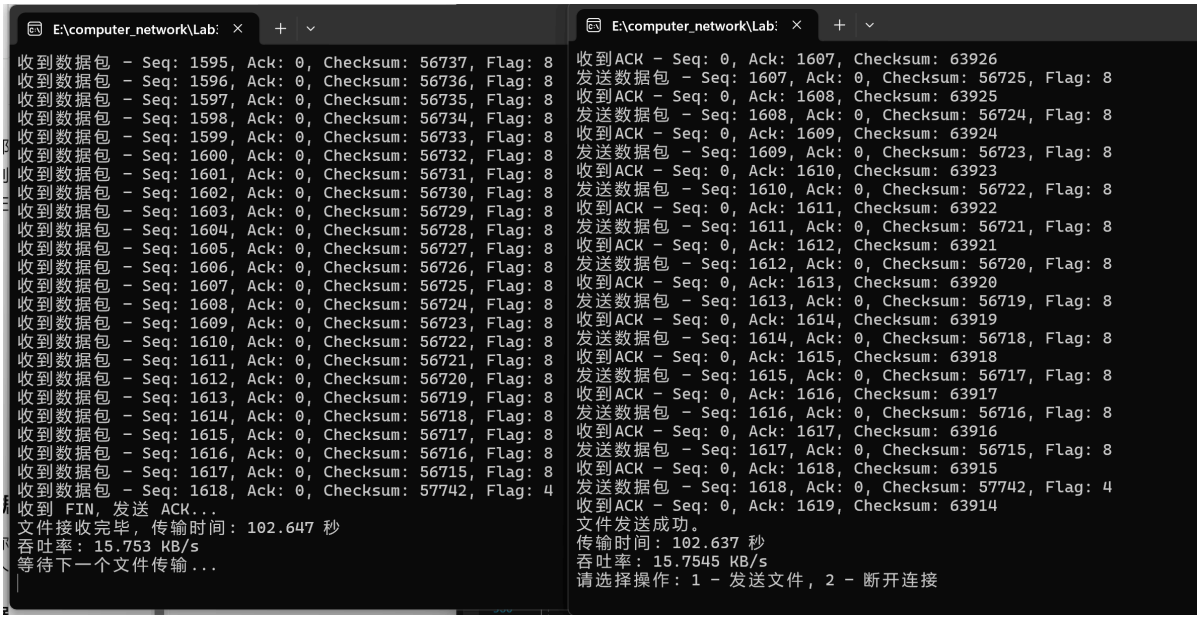
以传输大小为1.57 MB (1,655,808 字节)的1.txt文本文件为例。



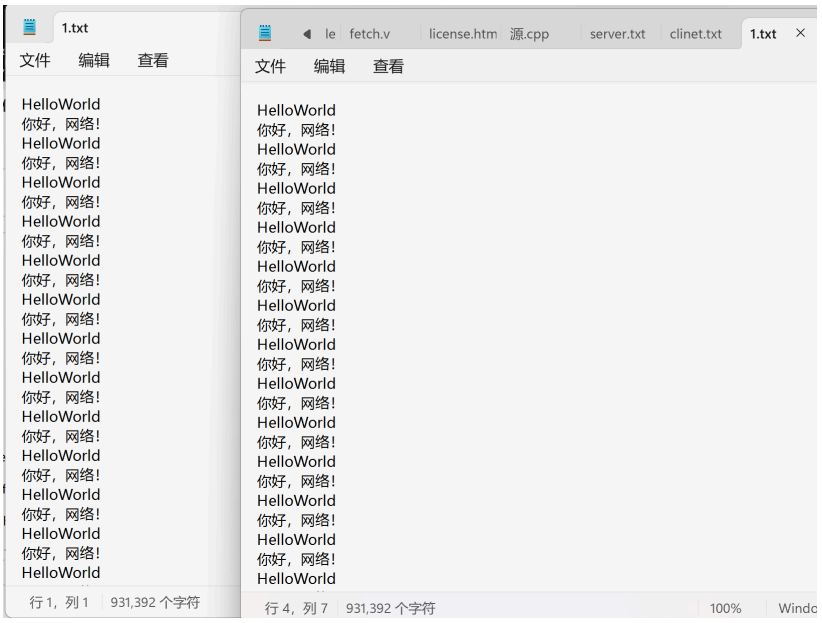
由于设置了丢包和延时，所以会把丢失的数据包进行重新发送，对应丢包信息如上上图中所示，

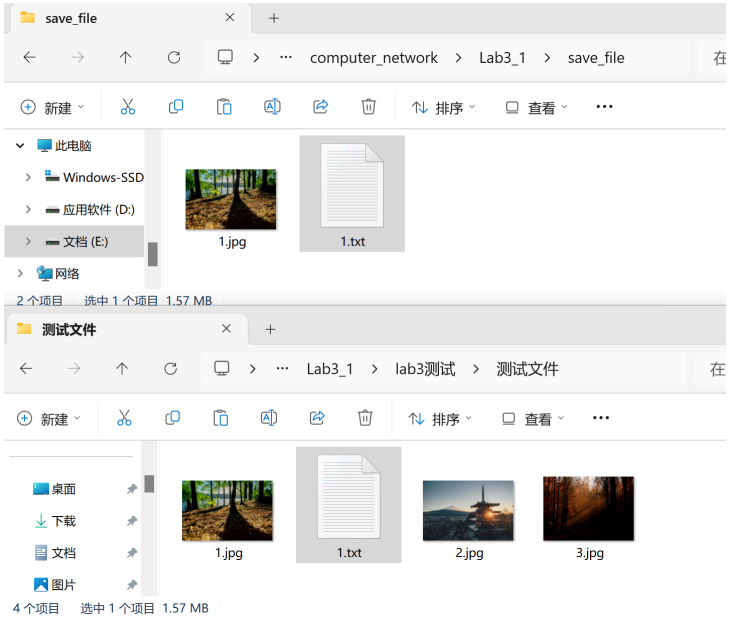


102.647s后该1.txt文本文档传输完毕，

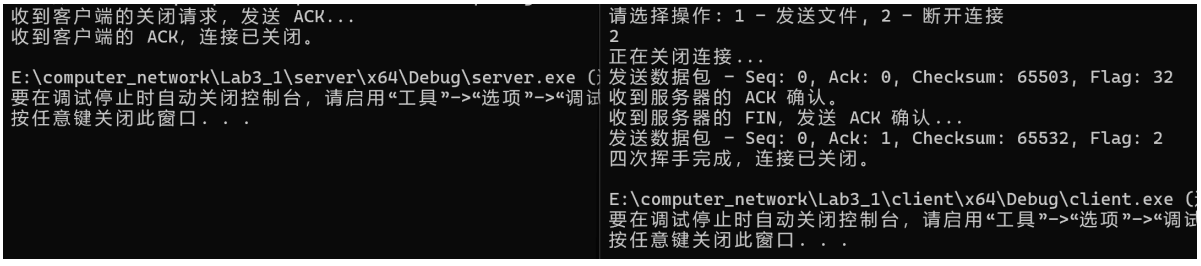


检查文本内容可以正常输出中英文，且大小属性一致，





在发送端输入2请求断开连接，发送端向接收端发送fin关闭请求，然后经过四次挥手后成功关闭。



五、问题反思

1. 校验和是怎么计算应用的

校验和通过将数据包分割为 16 位块进行加和，处理溢出，并取反，最终得到一个 16 位的校验和。这个校验和在发送数据时计算并插入到数据包中，接收方通过相同的算法验证数据包的完整性。

2. 发送端接收到ack之后要检查什么

如果 `Ack` 包含的确认号等于自己发送的下一个期望序列号，说明数据包已被接收端成功接收并确认，发送端可以继续发送下一个数据包。如果 `Ack` 包中的确认号不匹配，发送端会 **丢弃接收端发来的这个 ACK 包**，然后重新发送刚刚的数据包，因为可能是该包丢失或出现了网络延迟导致的 ACK 包丢失。

3. 流量控制的实现方法是什么?

本实验中通过**停等协议（Stop-and-Wait）**实现流量控制。即发送方每发送一个数据包后，都必须等待接收端确认（ACK）才能继续发送下一个数据包。如果没有收到确认，发送端会重发数据包，直到收到确认或达到最大重传次数。

4. 在传输过程中如何处理丢包、延时和失序问题?

丢包和失序问题通过重传机制解决。当发送端未收到 ACK 或 ACK 丢失、确认号错误时，会重传数据包；接收端通过序列号来确保数据包的顺序。如果数据包丢失，发送端会尝试多次重发，直到收到确认。延时问题也能通过设定合理的超时机制来应对，发送端会根据超时重传包。

反思

在这次实验中，我深入理解了如何在 UDP 协议上实现可靠传输，通过自定义协议增强其可靠性。实验过程中，我学习到了如何设计自定义的协议头结构，以及如何模拟 TCP 的连接管理机制，特别是在三次握手和四次挥手方面。尽管成功实现了重传和校验机制，但也意识到，在实际网络环境中，UDP 的可靠性可能会受到更多因素的影响，如网络拥塞、路由问题等，这些问题在实验中没有考虑到。

未来可以进一步改进该协议，如增加更复杂的流量控制算法，优化 ACK 机制，或增加更多的错误恢复手段，以提高协议的鲁棒性和性能。此外，我还想在接收端实现数据包排序，确保数据包的顺序，比如若数据包丢失，发送端会尝试多次重发，直到收到确认。延时问题也能通过设定合理的超时机制来应对，发送端会根据超时重传数据包。希望以后能实现我的这些设想。