

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

本实验在不可靠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3_1的基础上增加了基于流水线的滑动窗口传输机制、累积确认的Go-Back-N策略，以及超时重传与日志监控等要素。

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）

- 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。
- 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。
- 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。
- 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. 基于滑动窗口的流量控制机制

本协议采用滑动窗口机制实现流量控制，结合Go-Back-N（GBN）累积确认策略，确保数据的可靠传输与高效利用网络带宽。

2.1 窗口大小（Window Size）

在本实验中，我把窗口大小（`WINDOW_SIZE`）设置为固定值20。窗口大小决定了发送端在未收到确认（ACK）之前，能够连续发送的数据包数量。具体控制方式如下：

- 发送条件：**发送端在任意时刻仅当 `nextSeq < base + WINDOW_SIZE` 时，才允许发送新的数据包。
 - `base`：当前窗口的基序列号，表示最早未被确认的数据包的序列号。
 - `nextSeq`：下一个待发送的数据包的序列号。
- 窗口滑动：**当接收端发送ACK确认后，发送端将 `base` 移动至新的序列号，从而释放窗口空间，允许发送更多的数据包。确保发送端不会以过快的速率发送数据，有效控制了数据流量。

2.2 支持累积确认的Go-Back-N协议

本协议采用Go-Back-N（GBN）机制来实现可靠的数据传输和错误恢复。当某个数据包或其ACK丢失时，接收端会持续发送相同的ACK，指示发送端重传从丢失点开始的所有未确认的数据包。具体特点如下：

- 累积确认：**
 - 接收端通过返回 `Ack = expectedSeq` 来累积确认所有序列号小于 `expectedSeq` 的分组已被正确接收。
 - 若中间有分组丢失，接收端不会对之后的分组单独确认，而是继续发送对同一 `expectedSeq` 的ACK，阻止ACK的前移，迫使发送端进行重传。
- 重传机制：**
 - 在主循环的每次迭代结束时，发送端会遍历 `sendBuffer`，检查每个未确认的数据包是否超时，如果超时且未达到最大重传次数，则重传该数据包。
 - 当某个数据包的重传次数达到 `MAX_RETRANSMISSIONS` 时，表示该数据包仍未得到确认，并且多次尝试重传也没有成功。这时，程序会打印错误信息，表示传输失败，通过设置 `connected = false` 来中断文件传输过程，结束连接，并且清空发送缓冲区，以避免继续在不可靠的网络环境中尝试传输更多的数据。

2.3 工作流程示意

以下表格和示意图展示了在 `WINDOW_SIZE = 4` 的情况下，滑动窗口和GBN协议的工作流程（实际代码中窗口大小为20）。

时间点	发送的分组序号	窗口状态 (base - nextSeq)	接收端ACK情况	窗口滑动情况
初始	无	base=0, nextSeq=0 (空窗口)	无	无

时间点	发送的分组序号	窗口状态 (base - nextSeq)	接收端ACK情况	窗口滑动情况
发送	0,1,2,3	base=0, nextSeq=4 (窗口满)	等待确认	无
接收端ACK=4	无	base=0 → base=4 (已确认0~3)	累积确认至包序列号3 下一个期望=4	窗口向前滑动
再发送	4,5,6,7	base=4, nextSeq=8	等待确认	窗口内新数据

流水线协议及多序列号机制示意图



说明：

1. 初始状态：
- 发送端发送序号0、1、2、3的数据包，窗口状态为 `base=0, nextSeq=4`，窗口已满。

接收端期望序列号为0。
2. 正常确认：
- 接收端按序收到序号0和1的数据包，分别发送 `ACK=1` 和 `ACK=2`。

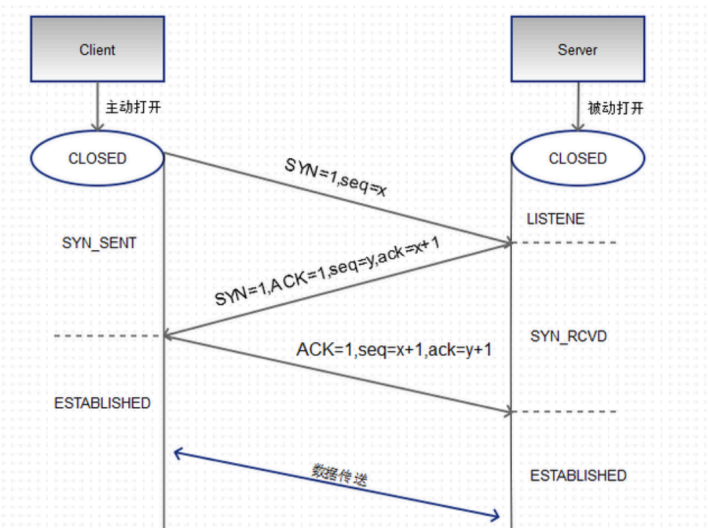
发送端收到ACK后，`base` 移动到相应的序列号，窗口滑动，允许发送新的数据包。
3. 数据包丢失：
- 假设序号2的数据包丢失，接收端无法收到序号2，仍然期望序列号为2，继续发送 `ACK=2`。

发送端在超时后，检测到ACK未更新，重传序号2和3的数据包。
4. 重传成功：
- 接收端收到重传的序号2和3的数据包，依次发送 `ACK=3` 和 `ACK=4`，窗口再次滑动。

3. 连接管理

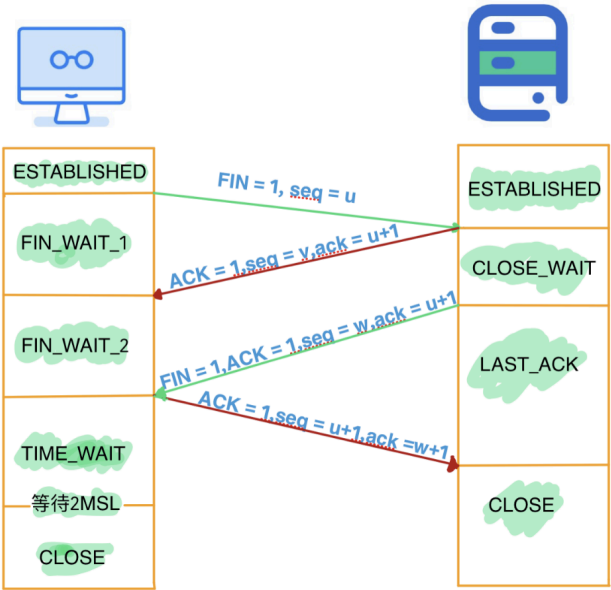
实验协议模拟TCP的三次握手和四次挥手过程：

• 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)
- 发送端和接收端的窗口大小, 接收端期望序列号
- 数据包的标志位 (Flag)
- 数据包的传输时间
- 吞吐量: 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;
}

```

窗口滑动部分代码:

```

while (!transmissionComplete || !sendBuffer.empty()) {
    // 填充滑动窗口
    while ((nextSeq < base + WINDOW_SIZE) && !fileStream.eof() &&
!transmissionComplete) {
        Message dataMsg = {};
        fileStream.read(dataMsg.Data, BUFFER_SIZE);
        size_t bytesRead = fileStream.gcount();

        if (bytesRead > 0) {
            dataMsg.Seq = nextSeq;
            dataMsg.Ack = 0;
            dataMsg.Flag = DATA;
            dataMsg.Length = static_cast<uint16_t>(bytesRead);

```

```

        Packet dataPkt;
        dataPkt.msg = dataMsg;
        dataPkt.sentTime = chrono::steady_clock::now();
        dataPkt.retransmissions = 0;

        sendBuffer[nextSeq] = dataPkt;
        sendMessage(clientSocket, destAddr,
sendBuffer[nextSeq].msg);
        nextSeq++;
        totalBytesSent += bytesRead;
    }

    if (fileStream.eof()) {
        transmissionComplete = true;
    }
}

// 设置超时时间
fd_set readfds;
FD_ZERO(&readfds);
FD_SET(clientSocket, &readfds);
timeval timeout;
timeout.tv_sec = 0;
timeout.tv_usec = 100 * 1000; // 100ms

int selectResult = select(0, &readfds, NULL, NULL, &timeout);
if (selectResult > 0 && FD_ISSET(clientSocket, &readfds)) {
    // 接收ACK
    Message ackMsg = {};
    int ackBytes = recvfrom(clientSocket, reinterpret_cast<char*>
(&ackMsg), sizeof(ackMsg), 0,
        reinterpret_cast<sockaddr*>(&from), &fromSize);
    if (ackBytes > 0) {
        uint16_t recvChecksum = ackMsg.Checksum;
        ackMsg.Checksum = 0;
        uint16_t calcChecksum =
calculateChecksum(reinterpret_cast<char*>(&ackMsg), sizeof(ackMsg));
        if (recvChecksum != calcChecksum) {
            cerr << "收到的ACK校验和不匹配, 丢弃数据包" << endl;
            continue;
        }

        if (ackMsg.Flag & ACK) {
            cout << "Ack: " << ackMsg.Ack << endl;
            if (ackMsg.Ack > base) {
                // 移除已确认的包
                for (uint32_t seq = base; seq < ackMsg.Ack;
++seq) {
                    sendBuffer.erase(seq);
                }
                base = ackMsg.Ack;
            }
        }
    }
}
}

```


累计确认部分代码：通过接收方发送的ACK消息，实现对多个数据包的累计确认。即一个ACK值代表接收方已经成功接收了所有序列号小于该ACK值的数据包。

```
if (ackMsg.Flag & ACK) {
    cout << "Ack: " << ackMsg.Ack << endl;
    if (ackMsg.Ack > base) {
        // 移除已确认的包
        for (uint32_t seq = base; seq < ackMsg.Ack; ++seq) {
            sendBuffer.erase(seq);
        }
        base = ackMsg.Ack;
    }
}
```

5.超时重传

遍历发送缓冲区 `sendBuffer` 中的所有未确认数据包，检查每个数据包是否已经超时。如果某个数据包的发送时间超过了预设的超时时间 `TIMEOUT_MS`（1000毫秒），且其重传次数未达到最大限制 `MAX_RETRANSMISSIONS`（5次），则触发重传。

```
// 检查超时并重传
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) {
            cout << "序列 " << seq << " 超时，正在重传..." << endl;
            sendMessage(clientSocket, destAddr, pkt.msg);
            pkt.sentTime = chrono::steady_clock::now();
            pkt.retransmissions++;
        }
        else {
            cerr << "序列 " << seq << " 达到最大重传次数，传输失败。" << endl;
            filestream.close();
            // 清空发送缓冲区并关闭连接
            sendBuffer.clear();
            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;

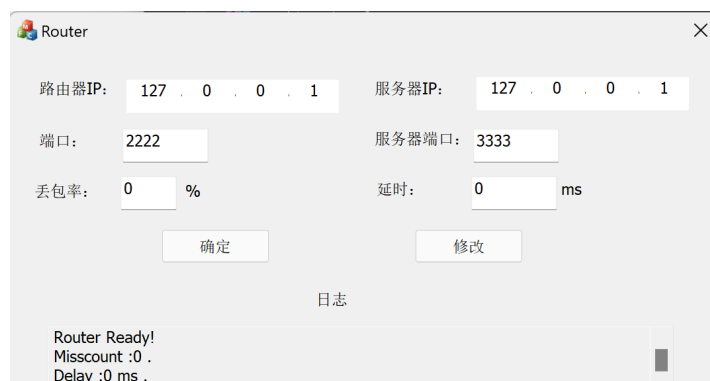
sendMessage(clientSocket, serverAddr, ackMsg);

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;
}
}
}

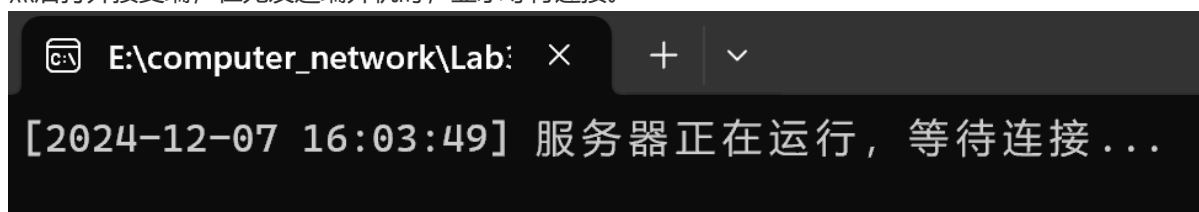
```

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

```
E:\computer_network\Lab: × + ∨

[2024-12-07 16:03:49] 服务器正在运行，等待连接...
[2024-12-07 16:04:26] 收到数据包 | Seq: 1 | Ack: 0 | Checksum: 0 | Flags: 1
[2024-12-07 16:04:26] 窗口状态 | 窗口大小: 20 | 期望序列号: 0
[2024-12-07 16:04:26] 收到 SYN, 正在回复 SYN-ACK...
[2024-12-07 16:04:26] 发送数据包 | Seq: 0 | Ack: 2 | Checksum: 65530 | Flags: 3
[2024-12-07 16:04:26] 三次握手成功，等待客户端操作。
[2024-12-07 16:04:26] 收到数据包 | Seq: 0 | Ack: 2 | Checksum: 0 | Flags: 2
[2024-12-07 16:04:26] 窗口状态 | 窗口大小: 20 | 期望序列号: 0
[2024-12-07 16:04:26] 收到ACK | Seq: 0 | Ack: 2 | Checksum: 0 | Flags: 2

E:\computer_network\Lab: × + ∨

尝试连接服务器...
发送数据包 - Seq:1 ,Ack: 0 ,Checksum: 65533 ,Flags: 1
收到SYN-ACK, 发送ACK...
发送数据包 - Seq:0 ,Ack: 2 ,Checksum: 65531 ,Flags: 2
三次握手成功
请选择操作: 1 - 发送文件, 2 - 断开连接
1
请输入文件路径:
E:\computer_network\Lab3_2\lab3测试\测试文件\1.txt
```

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2.76449秒，吞吐率为584.918KB/S。

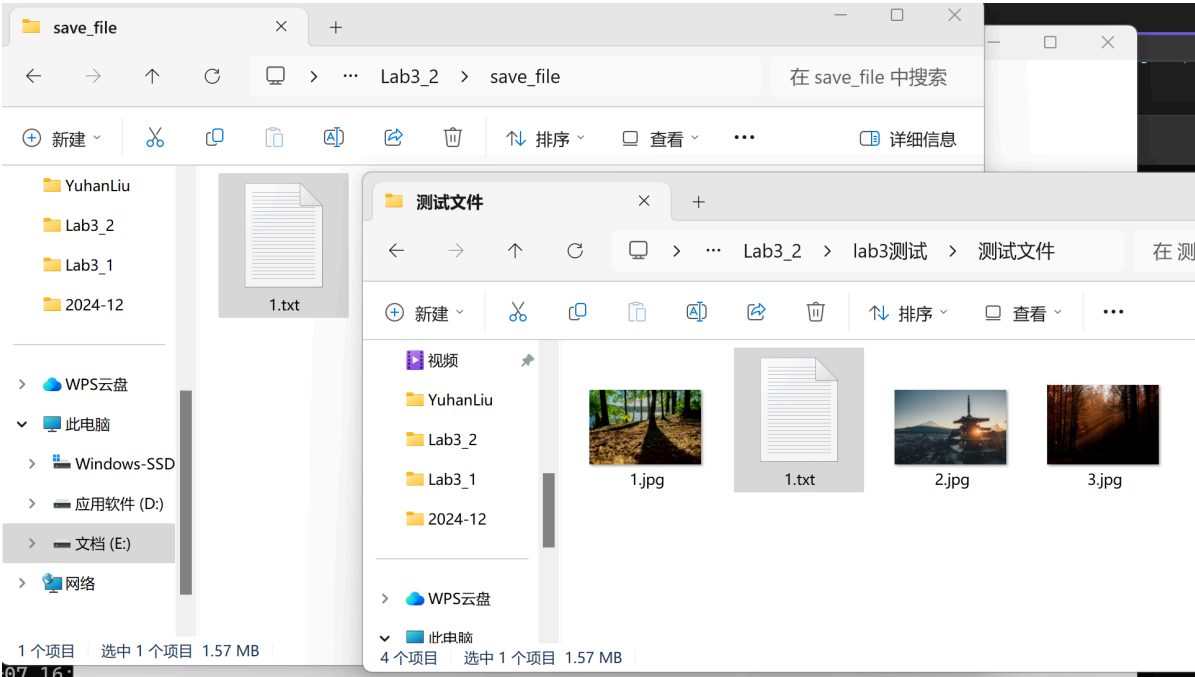
```
E:\computer_network\Lab: × + ∨

[2024-12-07 16:05:26] 窗口状态 | 窗口大小: 20 | 期望序列号: 1607
[2024-12-07 16:05:26] 发送数据包 | Seq: 0 | Ack: 1608 | Checksum: 63925 | Flags: 2
[2024-12-07 16:05:26] 收到数据包 | Seq: 1608 | Ack: 0 | Checksum: 0 | Flags: 8
[2024-12-07 16:05:26] 窗口状态 | 窗口大小: 20 | 期望序列号: 1608
[2024-12-07 16:05:26] 发送数据包 | Seq: 0 | Ack: 1609 | Checksum: 63925 | Flags: 2
[2024-12-07 16:05:26] 收到数据包 | Seq: 1609 | Ack: 0 | Checksum: 0 | Flags: 8
[2024-12-07 16:05:26] 窗口状态 | 窗口大小: 20 | 期望序列号: 1609
[2024-12-07 16:05:26] 发送数据包 | Seq: 0 | Ack: 1610 | Checksum: 63925 | Flags: 2
[2024-12-07 16:05:26] 收到数据包 | Seq: 1610 | Ack: 0 | Checksum: 0 | Flags: 8
[2024-12-07 16:05:26] 窗口状态 | 窗口大小: 20 | 期望序列号: 1610
[2024-12-07 16:05:26] 发送数据包 | Seq: 0 | Ack: 1611 | Checksum: 63925 | Flags: 2
[2024-12-07 16:05:26] 收到数据包 | Seq: 1611 | Ack: 0 | Checksum: 0 | Flags: 8
[2024-12-07 16:05:26] 窗口状态 | 窗口大小: 20 | 期望序列号: 1611
[2024-12-07 16:05:26] 发送数据包 | Seq: 0 | Ack: 1612 | Checksum: 63925 | Flags: 2
[2024-12-07 16:05:26] 收到数据包 | Seq: 1612 | Ack: 0 | Checksum: 0 | Flags: 8
[2024-12-07 16:05:26] 窗口状态 | 窗口大小: 20 | 期望序列号: 1612
[2024-12-07 16:05:26] 发送数据包 | Seq: 0 | Ack: 1613 | Checksum: 63925 | Flags: 2
[2024-12-07 16:05:26] 收到数据包 | Seq: 1613 | Ack: 0 | Checksum: 0 | Flags: 8
[2024-12-07 16:05:26] 窗口状态 | 窗口大小: 20 | 期望序列号: 1613
[2024-12-07 16:05:26] 发送数据包 | Seq: 0 | Ack: 1614 | Checksum: 63925 | Flags: 2
[2024-12-07 16:05:26] 收到数据包 | Seq: 1614 | Ack: 0 | Checksum: 0 | Flags: 8
[2024-12-07 16:05:26] 窗口状态 | 窗口大小: 20 | 期望序列号: 1614
[2024-12-07 16:05:26] 发送数据包 | Seq: 0 | Ack: 1615 | Checksum: 63925 | Flags: 2
[2024-12-07 16:05:26] 收到数据包 | Seq: 1615 | Ack: 0 | Checksum: 0 | Flags: 8
[2024-12-07 16:05:26] 窗口状态 | 窗口大小: 20 | 期望序列号: 1615
[2024-12-07 16:05:26] 发送数据包 | Seq: 0 | Ack: 1616 | Checksum: 63925 | Flags: 2
[2024-12-07 16:05:26] 收到数据包 | Seq: 1616 | Ack: 0 | Checksum: 0 | Flags: 8
[2024-12-07 16:05:26] 窗口状态 | 窗口大小: 20 | 期望序列号: 1616
[2024-12-07 16:05:26] 发送数据包 | Seq: 0 | Ack: 1617 | Checksum: 63925 | Flags: 2
[2024-12-07 16:05:26] 收到数据包 | Seq: 1617 | Ack: 0 | Checksum: 0 | Flags: 8
[2024-12-07 16:05:26] 窗口状态 | 窗口大小: 20 | 期望序列号: 1617
[2024-12-07 16:05:26] 发送数据包 | Seq: 0 | Ack: 1618 | Checksum: 63925 | Flags: 2
[2024-12-07 16:05:26] 收到数据包 | Seq: 1618 | Ack: 0 | Checksum: 0 | Flags: 8
[2024-12-07 16:05:26] 窗口状态 | 窗口大小: 20 | 期望序列号: 1618
[2024-12-07 16:05:26] 收到 FIN, 发送 ACK确认...
[2024-12-07 16:05:26] 发送数据包 | Seq: 0 | Ack: 1619 | Checksum: 63925 | Flags: 2
[2024-12-07 16:05:26] 文件接收完毕，传输时间: 2.76 秒吞吐率: 584.918 KB/s
2024-12-07 16:05:26等待下一个文件传输...

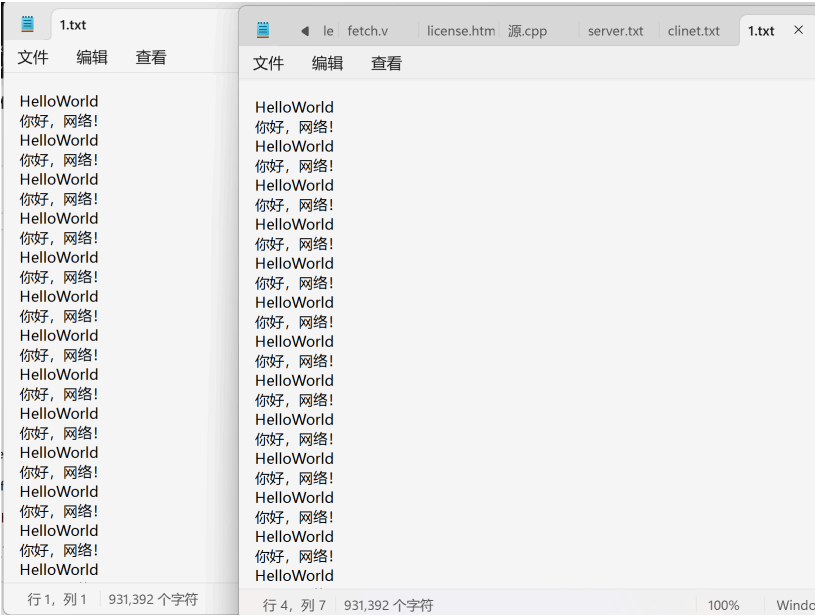
Microsoft Visual Studio 调试器 × + ∨

发送数据包 - Seq:1614 ,Ack: 0 ,Checksum: 56718 ,Flags: 8
Ack: 1596
发送数据包 - Seq:1615 ,Ack: 0 ,Checksum: 56717 ,Flags: 8
Ack: 1597
发送数据包 - Seq:1616 ,Ack: 0 ,Checksum: 56716 ,Flags: 8
Ack: 1598
发送数据包 - Seq:1617 ,Ack: 0 ,Checksum: 56715 ,Flags: 8
Ack: 1599
Ack: 1600
Ack: 1601
Ack: 1602
Ack: 1603
Ack: 1604
Ack: 1605
Ack: 1606
Ack: 1607
Ack: 1608
Ack: 1609
Ack: 1610
Ack: 1611
Ack: 1612
Ack: 1613
Ack: 1614
Ack: 1615
Ack: 1616
Ack: 1617
Ack: 1618
发送数据包 - Seq:1618 ,Ack: 0 ,Checksum: 63913 ,Flags: 4
文件发送成功。
传输时间: 2.76449 秒
吞吐率: 584.918 KB/s
收到ACK, 四次挥手成功，连接断开。
```

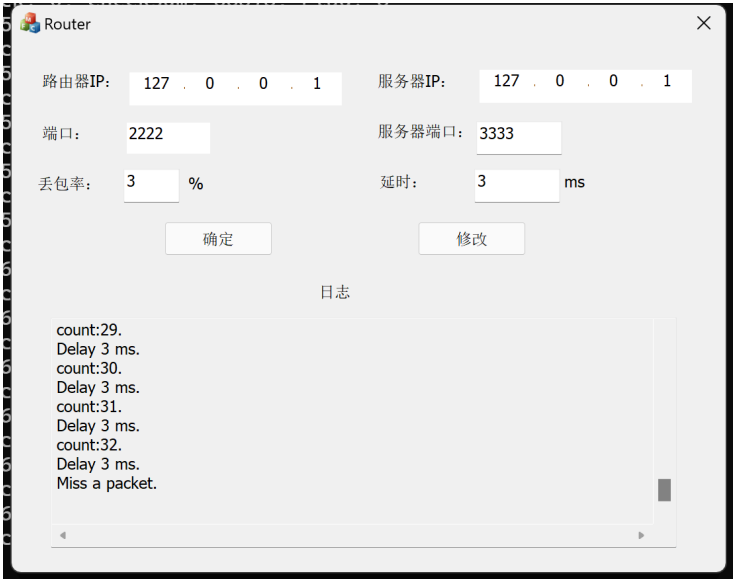
在接收端保存的路径下，可以看到的确收到了大小完整、属性相同的文本文件。



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。

```

收到数据包 | Seq: 0 | Ack: 2 | Checksum: 0 | Flags: 2
窗口状态 | 窗口大小: 20 | 期望序列号: 0
收到ACK | Seq: 0 | Ack: 2 | Checksum: 0 | Flags: 2
收到数据包 | Seq: 0 | Ack: 0 | Checksum: 0 | Flags: 16
窗口状态 | 窗口大小: 20 | 期望序列号: 0
接收到文件名: 1.jpg
发送数据包 | Seq: 0 | Ack: 1 | Checksum: 65532 | Flags: 2
收到数据包 | Seq: 1 | Ack: 0 | Checksum: 0 | Flags: 8
窗口状态 | 窗口大小: 20 | 期望序列号: 1
发送数据包 | Seq: 0 | Ack: 2 | Checksum: 65531 | Flags: 2
收到数据包 | Seq: 2 | Ack: 0 | Checksum: 0 | Flags: 8
窗口状态 | 窗口大小: 20 | 期望序列号: 2
发送数据包 | Seq: 0 | Ack: 3 | Checksum: 65530 | Flags: 2
收到数据包 | Seq: 3 | Ack: 0 | Checksum: 0 | Flags: 8
窗口状态 | 窗口大小: 20 | 期望序列号: 3
发送数据包 | Seq: 0 | Ack: 4 | Checksum: 65529 | Flags: 2
收到数据包 | Seq: 4 | Ack: 0 | Checksum: 0 | Flags: 8
窗口状态 | 窗口大小: 20 | 期望序列号: 4
发送数据包 | Seq: 0 | Ack: 5 | Checksum: 65528 | Flags: 2
收到数据包 | Seq: 5 | Ack: 0 | Checksum: 0 | Flags: 8
窗口状态 | 窗口大小: 20 | 期望序列号: 5
发送数据包 | Seq: 0 | Ack: 6 | Checksum: 65527 | Flags: 2
收到数据包 | Seq: 6 | Ack: 0 | Checksum: 0 | Flags: 8
窗口状态 | 窗口大小: 20 | 期望序列号: 6

请输入文件路径:
E:\computer_network\Lab3_2\lab3测试\测试文件\1.jpg
发送数据包 - Seq:0 ,Ack: 0 ,Checksum: 24551 ,Flags: 16
发送数据包 - Seq:1 ,Ack: 0 ,Checksum: 42741 ,Flags: 8
发送数据包 - Seq:2 ,Ack: 0 ,Checksum: 4429 ,Flags: 8
发送数据包 - Seq:3 ,Ack: 0 ,Checksum: 55579 ,Flags: 8
发送数据包 - Seq:4 ,Ack: 0 ,Checksum: 20051 ,Flags: 8
发送数据包 - Seq:5 ,Ack: 0 ,Checksum: 1670 ,Flags: 8
发送数据包 - Seq:6 ,Ack: 0 ,Checksum: 46558 ,Flags: 8
发送数据包 - Seq:7 ,Ack: 0 ,Checksum: 33960 ,Flags: 8
发送数据包 - Seq:8 ,Ack: 0 ,Checksum: 45398 ,Flags: 8
发送数据包 - Seq:9 ,Ack: 0 ,Checksum: 28797 ,Flags: 8
发送数据包 - Seq:10 ,Ack: 0 ,Checksum: 30736 ,Flags: 8
发送数据包 - Seq:11 ,Ack: 0 ,Checksum: 58318 ,Flags: 8
发送数据包 - Seq:12 ,Ack: 0 ,Checksum: 37376 ,Flags: 8
发送数据包 - Seq:13 ,Ack: 0 ,Checksum: 18140 ,Flags: 8
发送数据包 - Seq:14 ,Ack: 0 ,Checksum: 26589 ,Flags: 8
发送数据包 - Seq:15 ,Ack: 0 ,Checksum: 29423 ,Flags: 8
发送数据包 - Seq:16 ,Ack: 0 ,Checksum: 32586 ,Flags: 8
发送数据包 - Seq:17 ,Ack: 0 ,Checksum: 40938 ,Flags: 8
发送数据包 - Seq:18 ,Ack: 0 ,Checksum: 63444 ,Flags: 8
发送数据包 - Seq:19 ,Ack: 0 ,Checksum: 21016 ,Flags: 8
Ack: 1
发送数据包 - Seq:20 ,Ack: 0 ,Checksum: 56841 ,Flags: 8

```

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，

```

6 Ack: 1788
6 发送数据包 - Seq:1807 ,Ack: 0 ,Checksum: 30792 ,Flags: 8
6 Ack: 1789
6 发送数据包 - Seq:1808 ,Ack: 0 ,Checksum: 25851 ,Flags: 8
6 Ack: 1790
6 发送数据包 - Seq:1809 ,Ack: 0 ,Checksum: 648 ,Flags: 8
6 Ack: 1791
6 发送数据包 - Seq:1810 ,Ack: 0 ,Checksum: 57954 ,Flags: 8
6 Ack: 1791
6 Ack: 1791
6 Ack: 1791
6 Ack: 1791
6 Ack: 1791
6 Ack: 1791
6 序列 1791 超时，正在重传...
6 发送数据包 - Seq:1791 ,Ack: 0 ,Checksum: 11031 ,Flags: 8
6 序列 1792 超时，正在重传...
6 发送数据包 - Seq:1792 ,Ack: 0 ,Checksum: 32259 ,Flags: 8
6 序列 1793 超时，正在重传...
6 发送数据包 - Seq:1793 ,Ack: 0 ,Checksum: 25160 ,Flags: 8
6 序列 1794 超时，正在重传...

```

接收端收到来自发送端连续相同的ACK，
说明1791数据包丢失

于是触发重传机制，重新发送序号为1791
及当前窗口内所有其他未确认的数据包

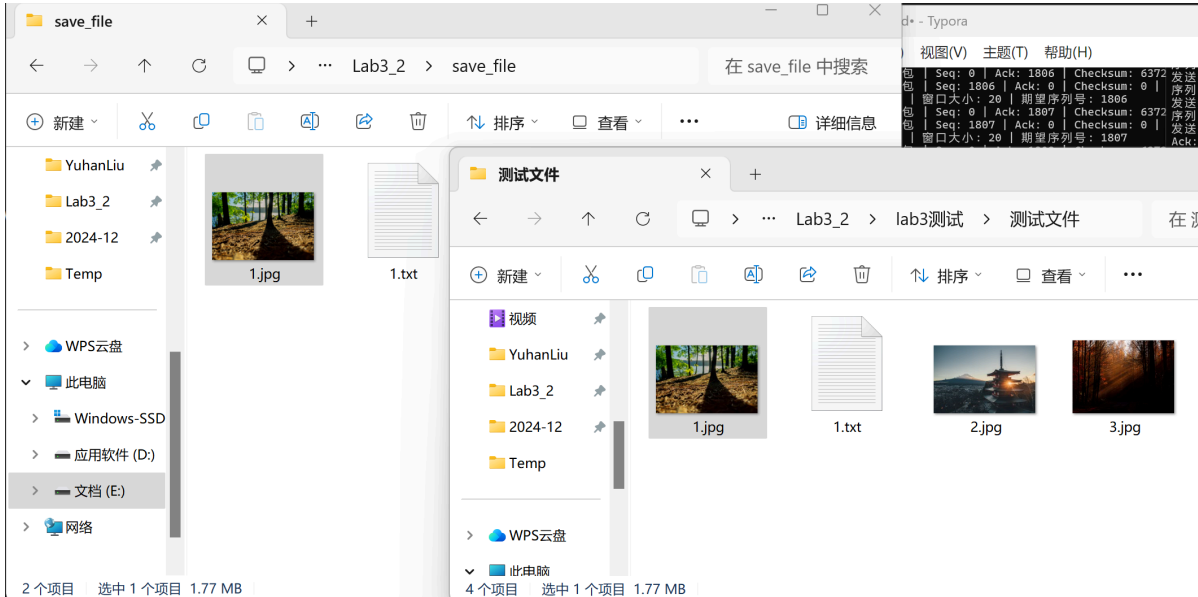
120.975s后该1.jpg文本档传输完毕，吞吐率为14.9934kb/s.


```
E:\computer_network\Lab3_2 > netstat -n -o | findstr 192.168.1.100
[2024-12-07 16:11:06] 发送数据包 | Seq: 0 | Ack: 1804 | Checksum: 6372
[2024-12-07 16:11:06] 收到数据包 | Seq: 1804 | Ack: 0 | Checksum: 0 |
[2024-12-07 16:11:06] 窗口状态 | 窗口大小: 20 | 期望序列号: 1804
[2024-12-07 16:11:06] 发送数据包 | Seq: 0 | Ack: 1805 | Checksum: 6372
[2024-12-07 16:11:06] 收到数据包 | Seq: 1805 | Ack: 0 | Checksum: 0 |
[2024-12-07 16:11:06] 窗口状态 | 窗口大小: 20 | 期望序列号: 1805
[2024-12-07 16:11:06] 发送数据包 | Seq: 0 | Ack: 1806 | Checksum: 6372
[2024-12-07 16:11:06] 收到数据包 | Seq: 1806 | Ack: 0 | Checksum: 0 |
[2024-12-07 16:11:06] 窗口状态 | 窗口大小: 20 | 期望序列号: 1806
[2024-12-07 16:11:06] 发送数据包 | Seq: 0 | Ack: 1807 | Checksum: 6372
[2024-12-07 16:11:06] 收到数据包 | Seq: 1807 | Ack: 0 | Checksum: 0 |
[2024-12-07 16:11:06] 窗口状态 | 窗口大小: 20 | 期望序列号: 1807
[2024-12-07 16:11:06] 发送数据包 | Seq: 0 | Ack: 1808 | Checksum: 6372
[2024-12-07 16:11:06] 收到数据包 | Seq: 1808 | Ack: 0 | Checksum: 0 |
[2024-12-07 16:11:06] 窗口状态 | 窗口大小: 20 | 期望序列号: 1808
[2024-12-07 16:11:06] 发送数据包 | Seq: 0 | Ack: 1809 | Checksum: 6372
[2024-12-07 16:11:06] 收到数据包 | Seq: 1809 | Ack: 0 | Checksum: 0 |
[2024-12-07 16:11:06] 窗口状态 | 窗口大小: 20 | 期望序列号: 1809
[2024-12-07 16:11:06] 发送数据包 | Seq: 0 | Ack: 1810 | Checksum: 6372
[2024-12-07 16:11:06] 收到数据包 | Seq: 1810 | Ack: 0 | Checksum: 0 |
[2024-12-07 16:11:06] 窗口状态 | 窗口大小: 20 | 期望序列号: 1810
[2024-12-07 16:11:06] 发送数据包 | Seq: 0 | Ack: 1811 | Checksum: 6372
[2024-12-07 16:11:06] 收到数据包 | Seq: 1811 | Ack: 0 | Checksum: 0 |
[2024-12-07 16:11:06] 窗口状态 | 窗口大小: 20 | 期望序列号: 1811
[2024-12-07 16:11:06] 发送数据包 | Seq: 0 | Ack: 1812 | Checksum: 6372
[2024-12-07 16:11:06] 收到数据包 | Seq: 1812 | Ack: 0 | Checksum: 0 |
[2024-12-07 16:11:06] 窗口状态 | 窗口大小: 20 | 期望序列号: 1812
[2024-12-07 16:11:06] 发送数据包 | Seq: 0 | Ack: 1813 | Checksum: 6372
[2024-12-07 16:11:06] 收到数据包 | Seq: 1813 | Ack: 0 | Checksum: 0 |
[2024-12-07 16:11:06] 窗口状态 | 窗口大小: 20 | 期望序列号: 1813
[2024-12-07 16:11:06] 发送数据包 | Seq: 0 | Ack: 1814 | Checksum: 6371
[2024-12-07 16:11:06] 收到数据包 | Seq: 1814 | Ack: 0 | Checksum: 0 |
[2024-12-07 16:11:06] 窗口状态 | 窗口大小: 20 | 期望序列号: 1814
[2024-12-07 16:11:06] 发送数据包 | Seq: 0 | Ack: 1815 | Checksum: 6371
[2024-12-07 16:11:06] 收到数据包 | Seq: 1815 | Ack: 0 | Checksum: 0 |
[2024-12-07 16:11:06] 窗口状态 | 窗口大小: 20 | 期望序列号: 1815
[2024-12-07 16:11:07] 收到 FIN, 发送 ACK 确认...
[2024-12-07 16:11:07] 发送数据包 | Seq: 0 | Ack: 1816 | Checksum: 63717 | Flags: 2 |

Microsoft Visual Studio 调试器
发送数据包 - Seq:1806, Ack: 0, Checksum: 39093, Flags: 8
序列 1807 超时, 正在重传...
发送数据包 - Seq:1807, Ack: 0, Checksum: 30792, Flags: 8
序列 1808 超时, 正在重传...
发送数据包 - Seq:1808, Ack: 0, Checksum: 25851, Flags: 8
序列 1809 超时, 正在重传...
发送数据包 - Seq:1809, Ack: 0, Checksum: 648, Flags: 8
序列 1810 超时, 正在重传...
发送数据包 - Seq:1810, Ack: 0, Checksum: 57954, Flags: 8
Ack: 1804
Ack: 1805
Ack: 1806
Ack: 1807
Ack: 1808
Ack: 1809
Ack: 1810
序列 1811 超时, 正在重传...
发送数据包 - Seq:1811, Ack: 0, Checksum: 55340, Flags: 8
Ack: 1812
序列 1812 超时, 正在重传...
发送数据包 - Seq:1812, Ack: 0, Checksum: 36515, Flags: 8
序列 1813 超时, 正在重传...
发送数据包 - Seq:1813, Ack: 0, Checksum: 13419, Flags: 8
序列 1814 超时, 正在重传...
发送数据包 - Seq:1814, Ack: 0, Checksum: 44086, Flags: 8
Ack: 1813
Ack: 1814
Ack: 1815
发送数据包 - Seq:1815, Ack: 0, Checksum: 63716, Flags: 4
文件发送成功。
传输时间: 120.975 秒
吞吐量: 14.9934 KB/s
收到 ACK, 四次挥手成功, 连接断开。

fin
Flags: 4
```

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，也可以在发送端输入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。


```
[2024-12-07 16:28:28] 收到数据包 | Seq: 1 | Ack: 0 | Checksum: 0 | Flags: 1
[2024-12-07 16:28:28] 窗口状态 | 窗口大小: 20 | 期望序列号: 0
[2024-12-07 16:28:28] 收到 SYN, 正在回复 SYN-ACK...
[2024-12-07 16:28:28] 发送数据包 | Seq: 0 | Ack: 2 | Checksum: 65530 | Flags: 3
[2024-12-07 16:28:28] 三次握手成功, 等待客户端操作。
[2024-12-07 16:28:28] 收到数据包 | Seq: 0 | Ack: 2 | Checksum: 0 | Flags: 2
[2024-12-07 16:28:28] 窗口状态 | 窗口大小: 20 | 期望序列号: 0
[2024-12-07 16:28:28] 收到ACK | Seq: 0 | Ack: 2 | Checksum: 0 | Flags: 2
[2024-12-07 16:28:32] 收到数据包 | Seq: 0 | Ack: 0 | Checksum: 0 | Flags: 32
[2024-12-07 16:28:32] 窗口状态 | 窗口大小: 20 | 期望序列号: 0
2024-12-07 16:28:32收到客户端的关闭请求, 发送 ACK...
[2024-12-07 16:28:32] 发送数据包 | Seq: 0 | Ack: 1 | Checksum: 65532 | Flags: 2
[2024-12-07 16:28:32] 发送数据包 | Seq: 0 | Ack: 0 | Checksum: 65531 | Flags: 4
2024-12-07 16:28:32收到客户端的 ACK, 连接已关闭。
```

```
Microsoft Visual Studio 调 × + ∨

尝试连接服务器...
发送数据包 - Seq:1 ,Ack: 0 ,Checksum: 65533 ,Flags: 1
收到SYN-ACK, 发送ACK...
发送数据包 - Seq:0 ,Ack: 2 ,Checksum: 65531 ,Flags: 2
三次握手成功
请选择操作: 1 - 发送文件, 2 - 断开连接
2
正在关闭连接...
发送数据包 - Seq:0 ,Ack: 0 ,Checksum: 65503 ,Flags: 32
收到服务器的ACK确认
收到服务器的 FIN, 发送 ACK 确认...
发送数据包 - Seq:0 ,Ack: 1 ,Checksum: 65532 ,Flags: 2
四次挥手成功, 连接断开
请按任意键继续...

E:\computer_network\Lab3_2\clinet\x64\Debug\clinet.exe (进程 23744)已退出, 代码为 0
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”
```

五、问题反思

1. 如果发送端收到了ack=2和ack=4, 但是没有收到ack=3, 应该怎么处理?

Ack=4 的到来表示序列号0到3的数据包全部被成功接收。虽然没有显式地从接收端获得 Ack=3, 但累积确认确保了 Ack=4 即包含了对2号和3号包的确认。所以只要看到 Ack 大于 base, 就会移除对应范围内的已确认包。(答辩的时候我好像说的有些问题, 在此进行更正, 希望助教老师理解🙏)

2. 超时重传的时间检测机制

每当发送一个数据包(包括文件名、数据包等), 程序会将该数据包的相关信息(如序列号、发送时间、重传次数)存储在发送缓冲区 sendBuffer 中, 同时记录下发送该数据包的时间点 sentTime 到 packet 结构体中。在主循环中, 遍历发送缓冲区中的所有未确认数据包, 计算自发送以来的时间差来判断是否超时来决定是否调用重传机制。

```
struct Packet {
    Message msg;
    chrono::steady_clock::time_point sentTime;
    int retransmissions;
};
```

3. 阻塞与非阻塞

阻塞模式 (Blocking Mode) : 在阻塞模式下, 当程序执行一个I/O操作(如 `recvfrom`)时, 如果数据尚未准备好, 程序会在该操作上等待, 暂停执行, 直到数据到达或操作完成。

- **优点**: 实现简单, 代码逻辑直观。
- **缺点**: 如果等待时间过长, 程序可能会无响应, 特别是在需要同时处理多个I/O操作时。

非阻塞模式 (Non-Blocking Mode)：在非阻塞模式下，I/O操作会立即返回，无论数据是否准备好。如果数据尚未到达，操作会返回一个错误或特定的状态，程序可以继续执行其他任务。

- **优点：**程序不会因为单个I/O操作而停滞，适合需要同时处理多个I/O操作的场景。
- **缺点：**实现复杂，需要额外的逻辑来管理I/O状态和数据处理

在我的原始代码中，通过 `select()` 函数，尽管套接字本身是阻塞的，但代码能够在短暂的时间内检查数据是否可用，从而实现类似非阻塞的行为，使程序在等待数据的同时能够执行其他任务（如检测超时并重传）。相比于将套接字设置为完全非阻塞模式，使用 `select()` 函数更容易管理多个I/O操作，同时保持代码的可读性和维护性。

后续为了体验非阻塞模式，我将套接字进行了如下修改，这样所有的I/O操作（如 `recvfrom`）不论数据是否可用都会立即返回了。

```
u_long mode = 1; // 1 为非阻塞模式
ioctlsocket(clientSocket, FIONBIO, &mode);
```

4. 滑动窗口的调整对吞吐率和传输速度的影响

窗口过小：当滑动窗口较小时，发送端只能发送有限数量的数据包，这会导致发送端频繁等待确认（ACK）。每次收到确认后，发送端才会继续发送新的数据包，造成频繁的停顿，降低了传输速度。吞吐率会受到严重影响，因为发送端的空闲时间较多，网络带宽没有被充分利用。

窗口过大：窗口过大会导致接收端的缓冲区可能出现溢出，增加了丢包的风险，并且发送端可能在某些情况下需要等待更多的ACK确认，浪费网络资源。由于接收端需要处理更多的数据包，处理能力可能成为瓶颈，导致系统延迟增大，吞吐率的提升反而有限。

反思

这次实验加深了我对可靠传输协议工作原理的理解，尤其是滑动窗口、超时重传和累积确认等机制的具体应用。实验的最终结果表明，协议能够在不同的网络条件下稳定工作，特别是在丢包和延迟较高的情况下，超时重传和滑动窗口的结合使得数据传输得以完成。通过实际测试，我还发现了一些细节上的问题，因此在实际应用中，可能需要根据具体的网络环境调整协议的各个参数，来平衡效率和稳定性。

未来，我打算进一步优化重传机制，减少因超时重传带来的性能下降。例如，可以通过引入更智能的拥塞控制算法，动态调整窗口大小，从而提高协议的适应性和吞吐率。此外，在面对高丢包率的网络环境时，可以考虑结合其他的错误恢复机制（如FEC、ARQ等），进一步提升数据传输的可靠性和效率。