实验3：基于UDP服务设计可靠传输协议并编程实现

2011269 王楠舟

实验3-3：在实验3-2的基础上，选择实现一种拥塞控制算法，也可以是改进的算法，完成给定测试文件的传输。

程序功能介绍

1. 在实验3-2的基础上，实现**RENO拥塞控制算法**；
2. 发送端使用三线程，分别负责报文的发送、接收和超时信号设置。

协议设计

数据报文结构

<table>
<tr><th>0~7</th><th>8~15</th><th>16~23</th><th>24~31</th></tr>
<tr><td colspan="4">SEQ</td></tr>
<tr><td colspan="4">ACK</td></tr>
<tr><td colspan="2">CheckSum</td><td colspan="2">BufferSize</td></tr>
<tr><td>Flag</td><td>WindowSize</td><td colspan="2"></td></tr>
<tr><td colspan="4">DATA</td></tr>
</table>

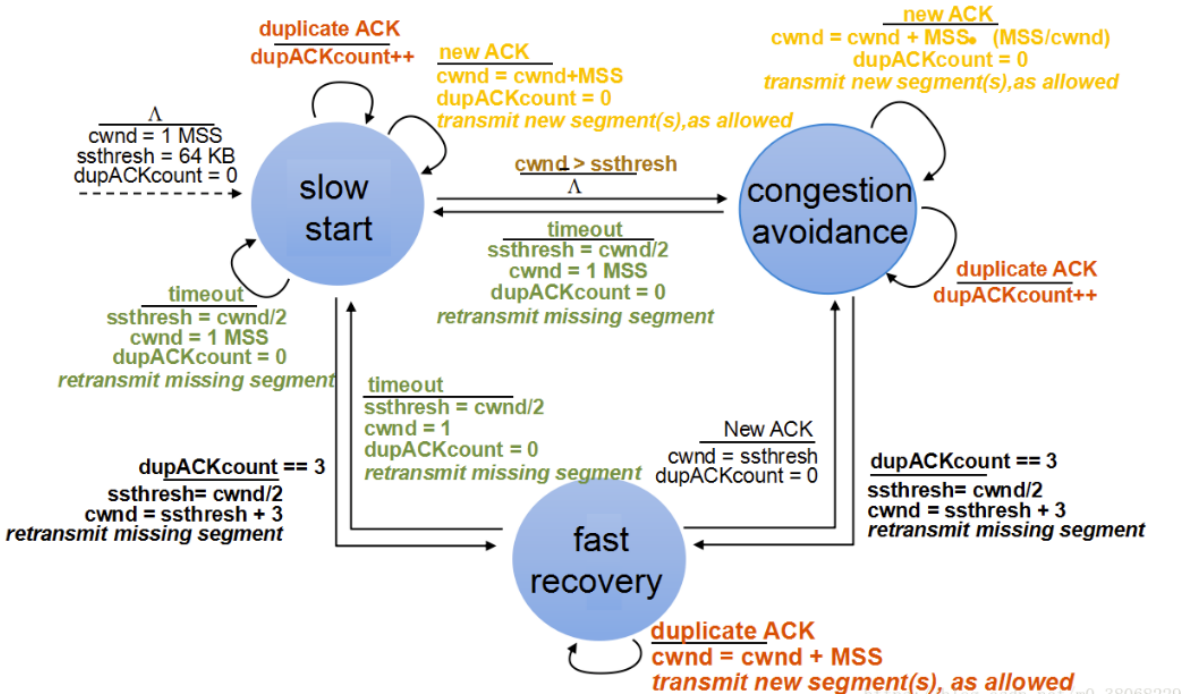
差错检测--计算校验和；

面向连接的数据传输：建立连接与断开连接；

传输协议：使用基于滑动窗口的流量控制机制的rdt3.0；

累积确认。

拥塞控制算法的实现：



初始发送端处于**慢启动**阶段，窗口大小设置为`cwnd = 1MSS`，`ssthresh = 32MSS`。

- 如果收到新的ACK，窗口大小增大1MSS。即在慢启动阶段每过一个RTT，cwnd翻倍，窗口大小呈指数增长；
- 如果收到被冗余的ACK报文，重复收到三次则认为出现了丢包，发送端将丢失报文重传，进入**快速恢复**阶段；
- 如果`cwnd > ssthresh`，发送端进入**拥塞避免**阶段；
- 如果超时没有收到新的ACK，重新进入**慢启动**阶段；

进入**拥塞避免**阶段后：

- 每次收到新的ACK，`cwnd = cwnd + MSS*(MSS/cwnd)`。即在拥塞避免阶段，每过一个RTT，cwnd加1，窗口大小线性增长；
- 如果收到被冗余的ACK报文，重复收到三次则认为出现了丢包，发送端将丢失报文重传，进入**快速恢复**阶段；
- 如果超时没有收到新的ACK，说明拥塞十分严重，`cwnd = 1MSS, ssthresh = cwnd.2`，发送端进入**慢启动阶段**；

进入**快速恢复**阶段后：

- 每次冗余的ACK，`cwnd += 1MSS`；
- 知道收到新的ACK，`cwnd = ssthresh`，将窗口大小恢复成阈值，然后进入**拥塞避免**阶段；
- 如果超时没有收到新的ACK，说明拥塞十分严重，`cwnd = 1MSS, ssthresh = cwnd.2`，发送端进入**慢启动阶段**；

算法代码实现

```
static u_long cwnd = MSS;
static u_long ssthresh = 10 * MSS;
static int dupACKCount = 0;

//sender缓冲区
static u_long lastSendByte = 0, lastAckByte = 0;
static Packet sendPkts[20]{};
```

对RENO算法的一些宏定义：

```
enum {
    START_UP, AVOID, RECOVERY
};

//Client端初始化的阶段
static int RENO_STAGE = START_UP;
```

算法代码的实现主要是在`Client`端的接收线程上修改：

线程函数：

```
DWORD WINAPI ACKHandler(LPVOID param) {
    SOCKET *clientSock = (SOCKET *) param;
    char recvBuffer[sizeof(Packet)];
    Packet recvPacket;

    while (true) {
        if (recvfrom(*clientSock, recvBuffer, sizeof(Packet), 0, (SOCKADDR *)
&addrSrv, &addrLen) > 0) {
            memcpy(&recvPacket, recvBuffer, sizeof(Packet));
            ......
```

如果接收到的ACK报文无误，且`base < (recvPacket.head.ack + 1)`是一个新的ACK，`Client`端首先进行窗口的滑动，然后根据目前所处在的RENO阶段来做处理：

- 如果处于**慢启动**阶段，`cwnd += d * MSS`，并判断cwnd是否超过阈值，如果超过则将RENO状态更新至**阻塞避免**阶段；
- 如果处于**阻塞避免**阶段，则`cwnd += d * MSS * MSS / cwnd`；
- 如果处于**快速恢复**阶段，`cwnd = ssthresh`，并将RENO状态更新进入**阻塞避免**阶段。

```
    if (CheckPacketSum((u_short *) &recvPacket, sizeof(Packet)) == 0 &&
recvPacket.head.flag & ACK) {
                mutexLock.lock();
                if (base < (recvPacket.head.ack + 1)) {
                    int d = recvPacket.head.ack + 1 - base;
                    //move the windows:
                    for (int i = 0; i < d; i++) {
                        lastAckByte += sendPkts[i].head.bufSize;
                    }
                    for (int i = 0; i < (int) waitingNum(nextSeqNum) - d; i++) {
                        sendPkts[i] = sendPkts[i + d];
                    }

                    switch (RENO_STAGE) {
                        case START_UP:
                            cwnd += d * MSS;
                            dupACKCount = 0;
                            if (cwnd >= ssthresh)
                                RENO_STAGE = AVOID;
                            break;
                        case AVOID:
                            cwnd += d * MSS * MSS / cwnd;
```

```
                        dupACKCount = 0;
                        break;
                    case RECOVERY:
                        cwnd = ssthresh;
                        dupACKCount = 0;
                        RENO_STAGE = AVOID;
                        break;
                }
                window = min(cwnd, windowSize);
                base = (recvPacket.head.ack + 1) % MAX_SEQ;
            }
```

如果收到的ACK报文是冗余的，则进入`else`分支：`dupACKCount++`，并且如果处于`START_UP || AVOID`阶段，就会重传报文，并进入`RECOVERY`**快速恢复**阶段；如果处于`RECOVERY`阶段，则`cwnd += MSS`，增大窗口。

```
            else {
                //duplicate ACK
                dupACKCount++;
                if (RENO_STAGE == START_UP || RENO_STAGE == AVOID) {
                    if (dupACKCount == 3) {
                        ssthresh = cwnd / 2;
                        cwnd = ssthresh + 3 * MSS;
                        RENO_STAGE = RECOVERY;

                        //retransmit missing segment
                        fastResend = true;
                    }
                } else {
                    cwnd += MSS;
                }

            }
            mutexLock.unlock();
```

在发送线程：每次循环首先判断是否需要重传缓冲区：

```
void sendFSM(u_long len, char *fileBuffer, SOCKET &socket, SOCKADDR_IN &addr) {

    int packetDataLen;

    char *data_buffer = new char[sizeof(Packet)], *pkt_buffer = new
char[sizeof(Packet)];
    nextSeqNum = base;
    cout << "本次文件数据长度为" << len << "Bytes" << endl;

    HANDLE ackhandler = CreateThread(nullptr, 0, ACKHandler, LPVOID(&socket), 0,
nullptr);
    while (true) {

        if (lastAckByte == len) {
            //结束发送
            CloseHandle(ackhandler);
            ......
```

```
        }

        if (fastResend)
            goto GBN;
            ....


        GBN:
        mutexLock.lock();
        resendPacketNum = nextSeqNum - 1;
        for (int i = 0; i < nextSeqNum - base; i++) {
            memcpy(pkt_buffer, &sendPkts[i], sizeof(Packet));
            sendto(socket, pkt_buffer, sizeof(Packet), 0, (SOCKADDR *) &addr,
addrLen);
        }
        fastResend = false;
        mutexLock.unlock();
        start = clock();
        stopTimer = false;
```

如果是正常的发送窗口：计算出每次发送报文的数据段长度`packetDataLen`，取值为`min(MSS,窗口剩余大小,文件剩余大小)`，虽然保存到`sendPkts`缓冲区中。

```
mutexLock.lock();
window = min(cwnd, windowSize);
if ((lastSendByte < lastAckByte + window) && (lastSendByte < len)) {
    packetDataLen = min(lastAckByte + window - lastSendByte, MSS);
    packetDataLen = min(packetDataLen, len - lastSendByte);
    memcpy(data_buffer, fileBuffer + lastSendByte, packetDataLen);

    sendPkts[nextSeqNum - base] = makePacket(nextSeqNum, data_buffer,
packetDataLen);
    memcpy(pkt_buffer, &sendPkts[nextSeqNum - base], sizeof(Packet));

    sendto(socket, pkt_buffer, sizeof(Packet), 0, (SOCKADDR *) &addr, addrLen);

    if (base == nextSeqNum) {
        start = clock();
        stopTimer = false;
    }
    nextSeqNum = (nextSeqNum + 1) % MAX_SEQ;
    lastSendByte += packetDataLen;
}
mutexLock.unlock();
```

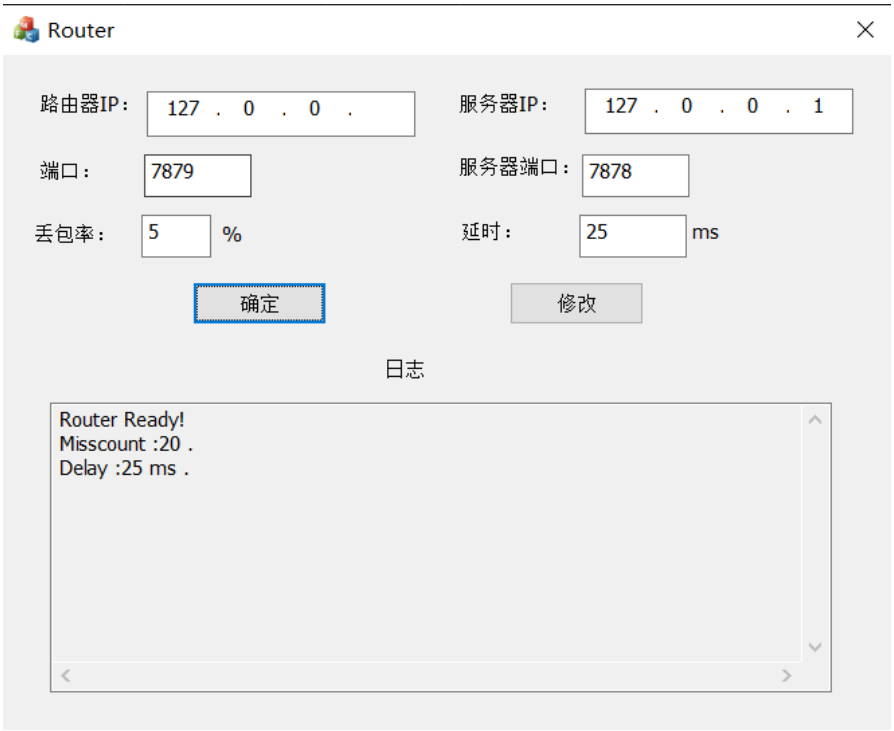
如果当前窗口超时：发送端将缓冲区中内容全部重传一次，然后进入`START_UP`阶段。

```
time_out:
if (!stopTimer && clock() - start >= MAX_TIME) {
    mutexLock.lock();
    ssthresh = cwnd / 2;
    cwnd = MSS;
    dupACKCount = 0;
    RENO_STAGE = START_UP;
```

```
    cout << "[time out!]resend" << endl;
    for (int i = 0; i < nextSeqNum - base; i++) {
        memcpy(pkt_buffer, &sendPkts[i], sizeof(Packet));
        sendto(socket, pkt_buffer, sizeof(Packet), 0, (SOCKADDR *) &addr,
addrLen);
    }
    mutexLock.unlock();
    start = clock();
    stopTimer = false;
}
continue;
```

实验结果展示

启动`Router`程序，设置如下:




```
//client端设置
#define PORT 7879

//server端设置
#define PORT 7878
#define ADDRSRV "127.0.0.1"
```

三次握手测试:

```
[NOT CONNECTED]请输入聊天服务器的地址
127.0.0.1
[SYN:1  ACK:0   FIN:0   END:0]SEQ:0     ACK:0   LEN:0
[SYN_SEND]第一次握手成功
[SYN:1  ACK:1   FIN:0   END:0]SEQ:0     ACK:0   LEN:0
[ACK_RECV]第二次握手成功
[SYN:0  ACK:1   FIN:0   END:0]SEQ:0     ACK:0   LEN:0
[ACK_SEND]三次握手成功
[CONNECTED]成功与服务器建立连接，准备发送数据
[SYSTEM]请输入需要传输的文件名
F:\Computer_network\Computer_Network\Lab3\Lab3_3\workfile3_1\1.jpg
```

```
F:\Computer_network\Computer_Network\Lab3\Lab3_3\cmake-build-debug\server.exe
[SYN:1  ACK:0   FIN:0   END:0]SEQ:0     ACK:0   LEN:0
[SYN_RECV]第一次握手成功
[SYN:1  ACK:1   FIN:0   END:0]SEQ:0     ACK:0   LEN:0
[SYN_ACK_SEND]第二次握手成功
[SYN:0  ACK:1   FIN:0   END:0]SEQ:0     ACK:0   LEN:0
[ACK_RECV]第三次握手成功
[CONNECTED]与用户端成功建立连接，准备接收文件
```

文件传输过程：

```
1857353
[SYSTEM]开始传输
本次文件数据长度为1857353Bytes
[START_UP]cwnd:16384    window:16384    ssthresh:81920
[lastACKByte:8192       lastSendByte:8192       lastWritenByte:24576]
[START_UP]cwnd:24576    window:24576    ssthresh:81920
[lastACKByte:16384      lastSendByte:24576      lastWritenByte:40960]
[START_UP]cwnd:32768    window:32768    ssthresh:81920
[lastACKByte:24576      lastSendByte:40960      lastWritenByte:57344]
[START_UP]cwnd:40960    window:40960    ssthresh:81920
[lastACKByte:32768      lastSendByte:57344      lastWritenByte:73728]
[START_UP]cwnd:49152    window:49152    ssthresh:81920
[lastACKByte:40960      lastSendByte:73728      lastWritenByte:90112]
[START_UP]cwnd:57344    window:57344    ssthresh:81920
[lastACKByte:49152      lastSendByte:73728      lastWritenByte:106496]
[START_UP]cwnd:65536    window:65536    ssthresh:81920
[lastACKByte:57344      lastSendByte:106496     lastWritenByte:122880]
[START_UP]cwnd:73728    window:73728    ssthresh:81920
[lastACKByte:65536      lastSendByte:106496     lastWritenByte:139264]
[AVOID]cwnd:81920       window:81920    ssthresh:81920
[lastACKByte:73728      lastSendByte:139264     lastWritenByte:155648]
[AVOID]cwnd:82739       window:82739    ssthresh:81920
[lastACKByte:81920      lastSendByte:155648     lastWritenByte:164659]
[AVOID]cwnd:83550       window:83550    ssthresh:81920
[lastACKByte:90112      lastSendByte:164659     lastWritenByte:173662]
[AVOID]cwnd:84353       window:84353    ssthresh:81920
[lastACKByte:98304      lastSendByte:173662     lastWritenByte:182657]
```

Lab3_3中的日志输出内容主要是窗口大小的变化以及窗口的滑动。可以发现，在`START_UP`阶段，cwnd的值迅速增大，当`cwnd>=ssthresh`后进入了`AVOID`阶段，cwnd增速减缓。

```
[lastACKByte:122880     lastSendByte:200624     lastWritenByte:209596]
[AVOID]cwnd:87489       window:87489    ssthresh:81920
[lastACKByte:131072     lastSendByte:209596     lastWritenByte:218561]
[AVOID]cwnd:87489       window:87489    ssthresh:81920
[lastACKByte:131072     lastSendByte:218561     lastWritenByte:218561]
[AVOID]cwnd:87489       window:87489    ssthresh:81920
[lastACKByte:131072     lastSendByte:218561     lastWritenByte:218561]
ACK DUP 3 times!Retransmit the missing packet
[RECOVERY]cwnd:68320    window:87489    ssthresh:43744
[lastACKByte:131072     lastSendByte:218561     lastWritenByte:218561]
[RECOVERY]cwnd:76512    window:68320    ssthresh:43744
[lastACKByte:131072     lastSendByte:218561     lastWritenByte:199392]
[RECOVERY]cwnd:84704    window:76512    ssthresh:43744
[lastACKByte:131072     lastSendByte:218561     lastWritenByte:207584]
[RECOVERY]cwnd:92896    window:84704    ssthresh:43744
[lastACKByte:131072     lastSendByte:218561     lastWritenByte:215776]
[RECOVERY]cwnd:101088   window:92896    ssthresh:43744
[lastACKByte:131072     lastSendByte:223968     lastWritenByte:223968]
[RECOVERY]cwnd:109280   window:101088   ssthresh:43744
[lastACKByte:131072     lastSendByte:232160     lastWritenByte:232160]
[AVOID]cwnd:43744       window:43744    ssthresh:43744
[lastACKByte:139264     lastSendByte:240352     lastWritenByte:183008]
[AVOID]cwnd:45278       window:45278    ssthresh:43744
[lastACKByte:147456     lastSendByte:240352     lastWritenByte:192734]
```

当在`AVOID`阶段，`Duplicate ACK == 3`开始快速重传，发送端进入`RECOVERY`阶段，在`RECOVERY`阶段每收到一个冗余的ACK值`cwnd += MSS`，直到收到一个`New ACK`，发送端进入`AVOID`阶段。`cwnd,ssthresh`的值的变化都体现在上述过程中。

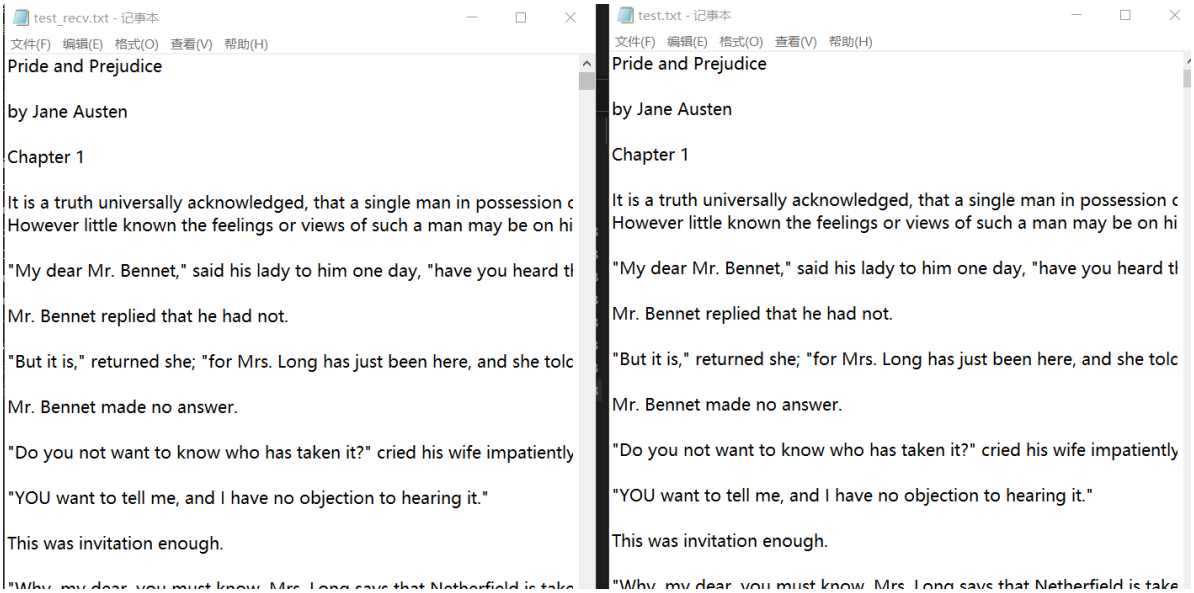
```
[window move]base:11 nextSeq:16 endWindow:27
[SYN:0  ACK:0   FIN:0   END:0]SEQ:16    ACK:0   LEN:8192
[SYN:0  ACK:0   FIN:0   END:0]SEQ:17    ACK:0   LEN:8192
[SYN:0  ACK:0   FIN:0   END:0]SEQ:18    ACK:0   LEN:8192
[SYN:0  ACK:0   FIN:0   END:0]SEQ:19    ACK:0   LEN:8192
[SYN:0  ACK:0   FIN:0   END:0]SEQ:20    ACK:0   LEN:8192
[SYN:0  ACK:0   FIN:0   END:0]SEQ:21    ACK:0   LEN:8192
[SYN:0  ACK:0   FIN:0   END:0]SEQ:22    ACK:0   LEN:8192
[SYN:0  ACK:0   FIN:0   END:0]SEQ:23    ACK:0   LEN:8192
[SYN:0  ACK:0   FIN:0   END:0]SEQ:24    ACK:0   LEN:8192
[SYN:0  ACK:0   FIN:0   END:0]SEQ:25    ACK:0   LEN:8192
[SYN:0  ACK:0   FIN:0   END:0]SEQ:26    ACK:0   LEN:8192
[time out!]resend begin
```

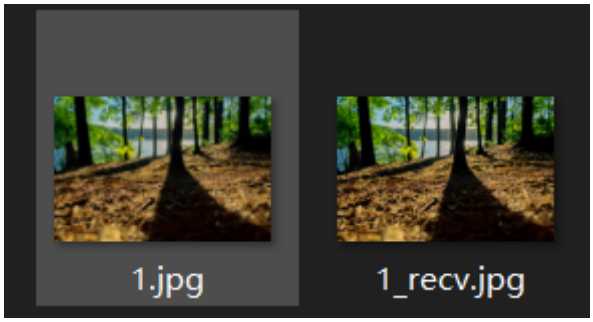
对应接收端点的状态：

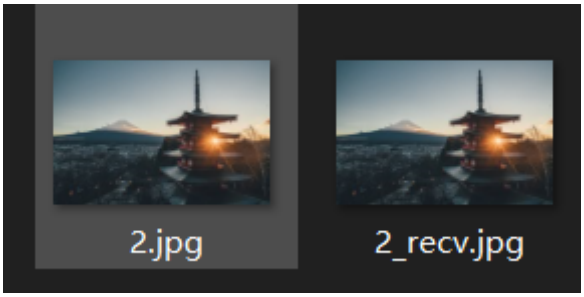
```
[ACK_RECV]第三次握手成功
[CONNECTED]与用户端成功建立连接，准备接收文件
[SYS]wait head:16
[SYS]recv head:17
[SYS]wait head:16
[SYS]recv head:18
[SYS]wait head:16
[SYS]recv head:21
[SYS]wait head:16
[SYS]recv head:22
[SYS]wait head:16
[SYS]recv head:23
[SYS]wait head:16
[SYS]recv head:24
[SYS]wait head:16
[SYS]recv head:29
[SYS]wait head:16
```

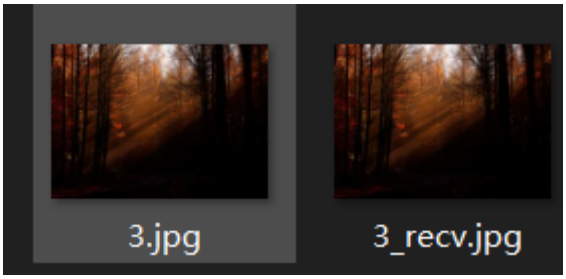
序列号为`16`报文可能在传输过程发生丢包，没有按序到达，所以接收端始终在等候，抛弃其他错序到达的报文。而发送端在接收到三次重复的`ACK = 15`就触发了快速重传进入`RECOVERY`快速恢复阶段，而不是等待`TIME_OUT`发生。最后等到重传时发送端传输过来的`16`号报文，接收端应答`ACK=16`，发送端恢复到`AVOID`阶段。

传输结果对比：









可见无论哪种类型的文件，传输前后都是一致的，验证了传输的可靠性。

GitHub仓库

仓库链接