计算机网络大作业报告

学号：20020006107 姓名：王义钧 专业：通信工程 年级：2020

1. 结合代码和 LOG 文件分析针对每个项目举例说明解决效果。（17 分）

（1）RDT2.0

假设信道可能出现比特差错，但是信道不会丢包。

在 RDT1.0 的基础上，采用自动重传请求（ARQ）协议。ARQ 协议需要三种协议功能来处理存在的比特差错的情况。

1、差错检验：实现校验和 checksum，实现对数据的校验：

```
public static short computeChkSum(TCP_PACKET tcpPack) {
    int checkSum = 0;

    //构造校验字符段
    String seq = "" + tcpPack.getTcpH().getTh_seq();//确认字段
    String ack = "" + tcpPack.getTcpH().getTh_ack();//ack字段
    String sum = "" + tcpPack.getTcpH().getTh_sum();//校验和字段
    int tcp_Data[] = tcpPack.getTcpS().getData();//数据字段

    String _sum = "" + seq + ack;
    for(int i = 0; i < tcp_Data.length; i++){
        _sum += tcp_Data[i];
    }

    //用Adler32校验
    byte[] nxt_cheksum = _sum.getBytes();
    Checksum adler = new Adler32();
    adler.reset();
    adler.update(nxt_cheksum, 0, nxt_cheksum.length);
    long ans = adler.getValue();
    checkSum = (short) ans;

    //checkSum = (checkSum & 0xffff) + (checkSum >> 16);

    return (short) checkSum;
}
```

计算方法是分别取出数据包首部的 seq、ack 字段和数据字段，利用字符串类型_sum 存储这三项之和，再调用 java.util 包中的 adler32 校验函数。我看大多数参考资料采用的是 crc32 校验方法，但 adler32 的运算速度要更快，但安全性可能有问题。其使用方法也与 crc32 大致相同，先构造校验字段，利用 update 上传，计算后得到的值通过 getValue 获取，注意的是这里要转换成 short 类型，因为需要取 16 位。

2、接收方（receiver）反馈

对于接收到的每一个包，计算其校验和。若校验和与数据包首部的校验和匹配，则返回一个 **ack** 值为本次接收到的包的 seq 值的包，并新建一个数据包，计算校验和，将该数据包返回给接收方（ACK），将本次接收到的包插入 data 队列准备交付；若校验和不匹配，则新建一个返回值 **ack** 值为-1 的包，表示这是一个 **NACK**。

```
//检查校验码，生成ACK
if(CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
    //生成ACK报文段（设置确认号）
    tcpH.setTh_ack(recvPack.getTcpH().getTh_seq());
    ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());
    tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));
    //回复ACK报文段
    reply(ackPack);

    //将接收到的正确有序的数据插入data队列，准备交付
    //RDT2.0 只有返回值为ACK时才会插入队列
    if(recvPack.getTcpH().getTh_seq()!=sequence){
        dataQueue.add(recvPack.getTcpS().getData());
        sequence=recvPack.getTcpH().getTh_seq();
    }else{
        System.out.println("收到重复包，序列号为："+sequence);
    }
```

```
}else{
    System.out.println("Recieve Computed: "+CheckSum.computeChkSum(recvPack));
    System.out.println("Recieved Packet"+recvPack.getTcpH().getTh_sum());
    System.out.println("Problem: Packet Number: "+recvPack.getTcpH().getTh_seq()+" + InnerSeq:
    tcpH.setTh_ack(-1);
    ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());
    tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));
    //回复ACK报文段
    reply(ackPack);
}
```

同时还要注意设置 reply 函数中的错误控制标识为 1。

3、发送方（sender）重传

发送方每发送出一个数据包，循环检查队列中是否有新收到的 ACK。在 waitACK 函数中，若新收到的 ACK 等于刚刚发送包的 seq，则结束检索，开始发送下一个数据包；若不等，则重发刚才的数据包，并继续进行 waitACK。

```
//循环检查ackQueue
//循环检查确认号对列中是否有新收到的ACK
if(!ackQueue.isEmpty()){
    // ACK
    int currentAck=ackQueue.poll();
    System.out.println("CurrentAck: "+currentAck);
    if (currentAck == tcpPack.getTcpH().getTh_seq()){
        // 新收到的ACK等于发送数据包的seq 代表ACK
        System.out.println("Clear: "+tcpPack.getTcpH().getTh_seq());
        flag = 1;           // ACK 则继续发生
        //break;
    }else{
        //NACK
        System.out.println("Retransmit: "+tcpPack.getTcpH().getTh_seq());
        udt_send(tcpPack);  //检测到NACK 进行数据包重传
        flag = 0;           // 状态保持为0  waitACK
    }
}
```

同时还要注意设置 udt_send 函数中的错误控制标识为 1。

验证 LOG 文件：

```
42      2022-12-09 12:01:12:757 CST DATA_seq: 3801       ACKed
43      2022-12-09 12:01:12:773 CST DATA_seq: 3901  WRONG   NO_ACK
44      2022-12-09 12:01:12:775 CST *Re: DATA_seq: 3901     ACKed
45      2022-12-09 12:01:12:789 CST DATA_seq: 4001       ACKed
46      2022-12-09 12:01:12:804 CST DATA_seq: 4101  WRONG   NO_ACK
47      2022-12-09 12:01:12:807 CST *Re: DATA_seq: 4101     ACKed
48      2022-12-09 12:01:12:819 CST DATA_seq: 4201       ACKed
```

发送方发送 seq=3901 和 4101 时均出现了 WRONG，代表数据包本身出错。于是接收方发送了一个 ack=-1（NACK）的包（如下图所示），所以紧接着发送方重发了这个数据包。

```
1064    2022-12-09 12:01:12:774 CST ACK_ack: -1
1065    2022-12-09 12:01:12:775 CST ACK_ack: 3901
1066    2022-12-09 12:01:12:790 CST ACK_ack: 4001
1067    2022-12-09 12:01:12:806 CST ACK_ack: -1
```

（2）RDT2.1

解决 RDT2.0 的冗余分组问题：修复了 RDT2.0 的缺点，即无法对 ACK 和 NAK 的出错进行处理的问题。在 rdt2.0 的基础之上，发送方在打包数据包时添加了 0 或者 1 编号，同样 ACK、NAK 字段上也添加了 0，1 字段，表示 0、1 号字段的确认或者否定。发送方就有了 2 种状态：发送 0 号数据包，1 号数据包，接收方也有了 2 种状态：等待 0 号数据包和等待 1 号数据包。

1、发送方修改

修改 recv 函数，对发送端发送而来的 ack 包进行校验。当发送方收到错误的 ack 包时，则认为接收方没有正确收到该包，则进行重传；若未出错，则将收到的 ack 包加入 ackQueue 队列。

```
if(CheckSum.computeChkSum(recvPack)==recvPack.getTcpH().getTh_sum()){
    System.out.println("Receive ACK Number： "+ recvPack.getTcpH().getTh_ack());
    ackQueue.add(recvPack.getTcpH().getTh_ack());

}else{
    System.out.println("Receive WRONG ACK Number： ");
    ackQueue.add(-1);
}
System.out.println();
//处理ACK报文
//while(flag == 0)
waitACK();
```

2、接收方修改

修改 rdt_rev 函数，接收方与 RDT2.0 类似，若计算校验和正确，则接收方正确接收一个包后，发送 ACK；在 ACK 包中，要将当前 seq 与上一个 seq 做对比，因此可以防止重复发包的问题。

```
//将接收到的正确有序的数据插入data队列，准备交付
if(recvPack.getTcpH().getTh_seq() != sequence){
    //RDT2.1 只有返回值为ACK时才会插入队列
    dataQueue.add(recvPack.getTcpS().getData());
    sequence = recvPack.getTcpH().getTh_seq();
}else{
    System.out.println("收到重复包，序列号为：" + sequence);
}
```

验证 LOG 文件：

```
20    2022-12-09 12:01:12:417 CST DATA_seq: 1701      NO_ACK
21    2022-12-09 12:01:12:418 CST *Re: DATA_seq: 1701     ACKed
1041    2022-12-09 12:01:12:418 CST ACK_ack: -1195300016    WRONG
```

可以看到，接收方在收到一个 seq=1701 的包后，回复了一个错误的 ACK 包。而发送方检测到了错误的 ACK 包，于是重发该数据包，第二次则被正确接收。相比于 RDT2.0，**RDT2.1 具有了对 ACK 和 NAK 的校验能力**。

（3）RDT2.2

在 RDT2.1 的基础之上做了小小的改善，摒弃了 NAK，只需采用 ACK。

1、发送方修改

修改 waitACK 函数，只有新收到的 ACK 包的确认字段等于发送数据包的 seq 字段时，才会接受确认为 ACK；否则，则重发错误的 ACK 序列号。（与 RDT2.1 的改动类似）

```
if (currentAck == tcpPack.getTcpH().getTh_seq()){
    // 新收到的ACK等于发送数据包的seq 代表ACK
    System.out.println("Clear: "+tcpPack.getTcpH().getTh_seq());
    flag = 1;           // ACK 则继续发生
    //break;
}else{
    //NACK
    System.out.println("Retransmit: "+tcpPack.getTcpH().getTh_seq());
    udt_send(tcpPack);  //检测到NACK 进行数据包重传
    flag = 0;           // 状态保持为0  waitACK
}
```

2、接收方修改

修改 rdt_recv 函数，使得当数据包出错时，即校验位出错，回复包的 ack=-1 改为 ack=上一次确认分组的 seq。

验证 LOG 文件：

```
2022-12-09 21:30:47:220 CST DATA_seq: 22301 WRONG   NO_ACK
2022-12-09 21:30:47:221 CST *Re: DATA_seq: 22301        ACKed
```

```
2022-12-09 21:30:47:205 CST ACK_ack: 22201
2022-12-09 21:30:47:221 CST ACK_ack: 22201
2022-12-09 21:30:47:222 CST ACK_ack: 22301
```

该发送方发送了一个错误的数据包，其 seq=22301，故接收方应发送一个 NACK 的包。但是在 RDT2.2 中，我们需要把 NACK 换成对上一个包的 ACK，即 ACK=22201，这样发送方看到后，就知道自己发送的数据包出错了，便进行数据重传。

需要注意的是，以上 WRONG NO_ACK 代表着数据包出错，而 NO_ACK 代表着返回的 ACK 出错。

（4）RDT3.0

在前面 RTD2.0 系列中，我对发送方接收到错误、重复的数据包进行重传，而对数据丢失情况没有做处理。rdt3.0 在 rdt2.2 的基础之上处理了数据包丢失的情况，增加了计时器的机制，如果在 RTT 时间段内，发送方没有接收到反馈信息，那么发送方默认数据包已经丢失了，会自动重传。

实现停止等待：设置计时器和重传任务，建立 UDT_Timer 对象与 UDT_RetransTask 对象，参数为客户端对象 client 与 tcp 分组 tcpPacks，以固定周期 3s 进行重传任务。

1、发送方修改

首先在 UDT_Sender 中生成 UDT_Timer 对象 timer。修改 rdt_sender 函数，加入计时器并设置重传任务。

```
public class TCP_Sender extends TCP_Sender_ADT {

    private TCP_PACKET tcpPack; //待发送的TCP数据报
    private volatile int flag = 0;

    /*构造函数*/
    public TCP_Sender() {
        super();     //调用超类构造函数
        super.initTCP_Sender(this);      //初始化TCP发送端
    }

    // RDT3.0
    UDT_Timer timer;
```

```
// RDT3.0
timer = new UDT_Timer();     // 设置计时器
UDT_RetransTask reTran = new UDT_RetransTask(client, tcpPack);  // 设置重传任务

// 每隔3s执行一次重传，直到收到ACK为止
timer.schedule(reTran, 3000, 3000);
```

每隔 3s 执行一次重传，直到收到 ACK 为止。

修改 waitACK 函数：增加条件，如果收到正确的 ACK，则关闭计时器。

```
if (currentAck == tcpPack.getTcpH().getTh_seq()){
    // 新收到的ACK等于发送数据包的seq 代表ACK
    System.out.println("Clear: "+tcpPack.getTcpH().getTh_seq());
    // 停止等待时需要关闭计时器
    System.out.println("关闭计时器");
    timer.cancel();
    flag = 1;            // ACK 则继续发生
    //break;
```

同时，需要修改 eflag 位为 4，表示数据传输过程出现出错/丢包。

2、接收方修改

首先修改 rdt_recv 函数，对于计算校验和错误的数据包，什么也不做，等待发送方超过 RTT 时间段内自动重传。

```
}else{
    System.out.println("Recieve Computed: "+CheckSum.computeChkSum(recvPack)
    System.out.println("Recieved Packet"+recvPack.getTcpH().getTh_sum());
    System.out.println("Problem: Packet Number: "+recvPack.getTcpH().getTh_
    tcpH.setTh_ack(sequence);
    ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());
    tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));
    //回复ACK报文段
    reply(ackPack);
}
```

同时，需要修改 eflag 位为 7，表示数据传输过程出现出错/丢包/延迟。

```
public void reply(TCP_PACKET replyPack) {
    //设置错误控制标志
    tcpH.setTh_eflag((byte) 4); //eFlag=0, 信道无错误

    //发送数据报
    client.send(replyPack);
}
```

验证 LOG 文件：

```
106    2022-12-13 20:28:57:032 CST DATA_seq: 10201 LOSS    NO_ACK
107    2022-12-13 20:29:00:047 CST *Re: DATA_seq: 10201        ACKed
```

如上图所示，发送方发送的 seq=10201 的数据包发生了丢失，在时间超过 3s 后，发送方没有接收到 ACK，于是重新发了 seq=10201 的数据包。实现了 RDT3.0 的停止等待协议。

（5）RDT4.0 流水线协议 RDT4.1 Go-Back-N

在 RDT3.0 的基础上，采用**流水线协议**，允许发送方在未得到对方确认的情况下一次发送多个分组。在 GBN 协议中，需要用到发送缓冲区和超时重传任务。

1、发送方修改

首先，在 rdt_send 函数中，要判断发送窗口是否已满，需要调用在 SenderWindow 中的 isFull()函数，如果窗口已满，则置 flag=0，并向控制台输出"window is full!!!"，同时在 flag=0 的情况下无限执行 while 循环，从而阻塞应用层调用。

当窗口有空余时，将当前的 packet 通过调用 SenderWindow 的 putPacket 函数放到窗口中，将放包这一过程放入 try-catch 过程，为了保证每一个数据包成功放入窗口。

```
// 窗口中的包都发完了，空转
if (this.windows.isFull()) {
    this.flag = 0;
    System.out.println("********************");
    System.out.println("window is full!!!");
    System.out.println("********************");
}
//等待ACK报文
while (this.flag == 0);

try {
    this.windows.putPacket(this.tcpPack.clone());
} catch (CloneNotSupportedException e) {
    e.printStackTrace();
}

//发送TCP数据报
udt_send(tcpPack);
```

这里需要注意的是，要将最开始的标志位 **flag=0 置为 1**，表示最开始是有窗口存在的。

同时，设置发送方累计确认：修改 recv 函数，判断两种情况：

先检查数据包是否出错，在之前的 RDT3.0 中，如果接收到正确的 ACK，则将其放到 ack 队列中，如果 ACK 出错，则 ack 队列-1。而在 RDT

①但如果 ACK 出错，这时如果后续的包能被正确接收并且发送方可以收到 ACK，那么顺序传输即可；

②如果数据包本身就出错了，即校验和错误，这时需要触发超时重传任务 **My_UDT_RetransTask。**

所以正常来讲，正确的数据包首先要调用 SenderWindow 中的 rcv 函数，更新滑动窗口左沿与数量，再检查窗口是否已满。

```
if(CheckSum.computeChkSum(recvPack)==recvPack.getTcpH().getTh_sum()){
    // 收到正确的ACK
    System.out.println("Receive ACK Number： "+ recvPack.getTcpH().getTh_ack());
    this.windows.rcv((recvPack.getTcpH().getTh_ack() - 1) / 100);
    if (!windows.isFull()) {
        this.flag = 1;
    }
}
```

同时，需要修改 eflag 位为 7，表示数据传输过程出现出错/丢包/延迟。

```
//设置错误控制标志
//0.信道无差错
//1.只出错
//2.只丢包
//3.只延迟
//4.出错 / 丢包
//5.出错 / 延迟
//6.丢包 / 延迟
//7.出错 / 丢包 / 延迟
tcpH.setTh_eflag((byte) 7);
```

2、发送方窗口建立

首先定义发送窗口左沿、下一个窗口的序列号和窗口大小。使用 nextseq 跟踪窗口左沿。

```
public class SendWindow {
    public int base = 0;//窗口左沿
    public int nextseq = 0;//指向下一个发送
    public int size=16;//窗口大小
```

用数组建立存储数据包的窗口序列：

```
private TCP_PACKET[] packets = new TCP_PACKET[size];  // 存储窗口内的包
```

仿照 RDT3.0，建立计时器和超时重传任务：

```
private UDT_Timer timer;
private My_UDT_RetransTask task;
```

isFull()函数判断窗口已满：

```
// 判断窗口是否已满
public boolean isFull() {
    //return this.cwnd <= this.packets.size();
    return this.size <= this.nextseq;
}
```

Size 为窗口大小（16），nextseq 代表下一个包放入窗口的位置。

putPacket()函数使用一个计时器设置窗口放包。

```
/*向窗口中加入包*/
public void putPacket(TCP_PACKET packet) {
    packets[nextseq] = packet;       // 在窗口的插入位置放入包
    if (nextseq == 0) {              // 如果在窗口左沿，则要开启计时器
        timer = new UDT_Timer();     // 设置计时器
        My_UDT_RetransTask task = new My_UDT_RetransTask(client, packets);  // 设置重传任务

        // 每隔3s执行一次重传，直到收到ACK为止
        timer.schedule(task, 3000, 3000);
    }

    nextseq++;  // 更新窗口的插入位置

}
```

每一次在队列中的 nextseq 位置放入当前数据包，并判断如果是新窗口（窗口左沿），则开启重传计时。

rcv 函数用于更新滑动窗口：

```
(base <= CurrentAck && CurrentAck < base + size) {  // 如果收到的ACK在窗口范围内
for (int i = 0; CurrentAck - base + 1 + i < size; i++) {  // 将窗口中位于确认的包之后的包整体移动到窗口左沿
    packets[i] = packets[CurrentAck - base + 1 + i];
    packets[CurrentAck - base + 1 + i] = null;
}

nextseq -=CurrentAck - base + 1;  // 更新nextIndex
base = CurrentAck + 1;  // 更新窗口左沿指示的seq

timer.cancel();  // 停止计时器

if (nextseq != 0) {  // 窗口中仍有包，则重开计时器
    timer = new UDT_Timer();    // 设置计时器
    My_UDT_RetransTask task = new My_UDT_RetransTask(client, packets);  // 设置重传任务

    // 每隔3s执行一次重传，直到收到ACK为止
    timer.schedule(task, 3000, 3000);
}
```

如果当前收到的 ACK 满足≥窗口左沿，且小于窗口大小，那么整体移动窗口到当前 ACK 之后，并更新下一个窗口左沿右移的距离为当前减去当前左沿的宽度，同时需要再检测窗口中是否有包，如果没有包则停止发送，此时 nextseq=0，代表**下一个包放入的位置为窗口左沿**，如果还存在包则需要开启计时器判断是否需要超时重传。

3、超时重传任务建立

继承 TimerTask 类编写 My_UDT_RetransTask：

```
public class My_UDT_RetransTask extends TimerTask{
        // 构造超时传送数据包
        private Client senderClient;
        public int size=20;//窗口大小
//          private TCP_PACKET reTransPacket;
        private TCP_PACKET[] packets;  // 维护窗口内包的数组

        public My_UDT_RetransTask(Client client, TCP_PACKET packet[]){
            super();
            this.senderClient = client;
            this.packets = packet;
        }

        public void run() {
            System.out.println("超时重发包");
            for (int i = 0; i < packets.length; i ++ )
            {
                if (packets[i] == null) {  // 如果没有包则跳出循环
                    break;
                } else {  // 逐一递交各个包
                    senderClient.send(packets[i]);
                }
            }
        }
}
```

主要编写 run 函数，一次性重传窗口内所有的数据包。

4、接收方修改

修改 rdt_recv 函数，首先建立期待值 seq 和当前 seq 的记录：

```
private int expectedSequence = 0;  // 用于记录期望收到的seq
```

```
int currentACK = (recvPack.getTcpH().getTh_seq() - 1) / 100;  // 当前包的seq
```

如果校验和计算错误，则只需等待超时重传即可；如果计算正确，首先要比较当前包的 seq 与期待的 seq，如果正确，则参考 RDT2.0 的数据交付方式，回复 ACK 报文，并将数据加入到 data 队列中，同时给期望值 seq+1；如果不相等，则对已确认数据包中 seq 最大的包（即 expectSequence - 1 号包）回复。

```
public void rdt_recv(TCP_PACKET recvPack) {
    //检查校验码，生成ACK
    if(CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
        int currentACK = (recvPack.getTcpH().getTh_seq() - 1) / 100;  // 当前包的seq
        if (expectedSequence == currentACK) {  // 当前收到的包就是期望的包
            //生成ACK报文段（设置确认号）
            tcpH.setTh_ack(recvPack.getTcpH().getTh_seq());  // 设置确认号为收到的TCP分组的seq
            ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());  // 新建一个TCP分组（ACK）
            tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));  // 设置ACK的校验位

            reply(ackPack);  // 回复ACK报文段

            // 将接收到的正确有序的数据插入 data 队列，准备交付
            dataQueue.add(recvPack.getTcpS().getData());

            expectedSequence += 1;  // 更新期望收到的包的seq
        } else {  // 收到失序的包，返回已确认的最大序号分组的确认
            //生成ACK报文段（设置确认号）
            tcpH.setTh_ack((expectedSequence - 1) * 100 + 1);  // 设置确认号为已确认的最大序号
            ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());  // 新建一个TCP分组（ACK）
            tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));  // 设置ACK的校验位

            reply(ackPack);  // 回复ACK报文段
        }
```

交付数据时，积累 20 个数据包交付一次，注意的是这里的 20 不是窗口大小，**在超时重传时需要的参数是窗口大小 16**。

```
//交付数据（每20组数据交付一次）
if(dataQueue.size() == 20)
    deliver_data();
```

同时，需要修改 eflag 位为 7，表示数据传输过程出现出错/丢包/延迟。

```
//设置错误控制标志
tcpH.setTh_eflag((byte) 7); //eFlag=0，信道无错误
```

验证 LOG 文件：

```
2022-12-27 20:31:02:295 CST DATA_seq: 23401 DELAY   NO_ACK
2022-12-27 20:31:02:310 CST DATA_seq: 23501     NO_ACK
2022-12-27 20:31:02:325 CST DATA_seq: 23601     NO_ACK
2022-12-27 20:31:02:340 CST DATA_seq: 23701     NO_ACK
2022-12-27 20:31:02:356 CST DATA_seq: 23801     NO_ACK
2022-12-27 20:31:02:371 CST DATA_seq: 23901     NO_ACK
2022-12-27 20:31:02:387 CST DATA_seq: 24001     NO_ACK
2022-12-27 20:31:02:402 CST DATA_seq: 24101     NO_ACK
2022-12-27 20:31:02:417 CST DATA_seq: 24201     NO_ACK
2022-12-27 20:31:02:434 CST DATA_seq: 24301     NO_ACK
2022-12-27 20:31:02:449 CST DATA_seq: 24401     NO_ACK
2022-12-27 20:31:02:464 CST DATA_seq: 24501     NO_ACK
2022-12-27 20:31:02:480 CST DATA_seq: 24601     NO_ACK
2022-12-27 20:31:02:495 CST DATA_seq: 24701     NO_ACK
2022-12-27 20:31:02:510 CST DATA_seq: 24801     NO_ACK
2022-12-27 20:31:02:527 CST DATA_seq: 24901     NO_ACK
```

如图所示，当发送 seq=23401 的包时，出现了延迟，导致在这个窗口内的 16 个数据包都没有正确收到 ACK。

```
1445    2022-12-27 20:31:02:311 CST ACK_ack: 23301
1446    2022-12-27 20:31:02:325 CST ACK_ack: 23301
1447    2022-12-27 20:31:02:341 CST ACK_ack: 23301
1448    2022-12-27 20:31:02:357 CST ACK_ack: 23301
1449    2022-12-27 20:31:02:372 CST ACK_ack: 23301
1450    2022-12-27 20:31:02:388 CST ACK_ack: 23301
1451    2022-12-27 20:31:02:403 CST ACK_ack: 23301
1452    2022-12-27 20:31:02:419 CST ACK_ack: 23301
1453    2022-12-27 20:31:02:434 CST ACK_ack: 23301
1454    2022-12-27 20:31:02:450 CST ACK_ack: 23301
1455    2022-12-27 20:31:02:464 CST ACK_ack: 23301
1456    2022-12-27 20:31:02:480 CST ACK_ack: 23301
1457    2022-12-27 20:31:02:496 CST ACK_ack: 23301
1458    2022-12-27 20:31:02:511 CST ACK_ack: 23301
1459    2022-12-27 20:31:02:527 CST ACK_ack: 23301
```

此时，接收方在对已知的最大序号 seq=23301 重复确认。

```
284    2022-12-27 20:31:02:527 CST DATA_seq: 24901     NO_ACK
285    2022-12-27 20:31:05:301 CST *Re: DATA_seq: 23401        ACKed
286    2022-12-27 20:31:05:301 CST *Re: DATA_seq: 23501        ACKed
287    2022-12-27 20:31:05:301 CST *Re: DATA_seq: 23601        ACKed
288    2022-12-27 20:31:05:302 CST *Re: DATA_seq: 23701        ACKed
289    2022-12-27 20:31:05:302 CST *Re: DATA_seq: 23801        ACKed
290    2022-12-27 20:31:05:302 CST *Re: DATA_seq: 23901        ACKed
291    2022-12-27 20:31:05:302 CST *Re: DATA_seq: 24001        ACKed
292    2022-12-27 20:31:05:302 CST *Re: DATA_seq: 24101        ACKed
293    2022-12-27 20:31:05:302 CST *Re: DATA_seq: 24201        ACKed
294    2022-12-27 20:31:05:302 CST *Re: DATA_seq: 24301        ACKed
295    2022-12-27 20:31:05:302 CST *Re: DATA_seq: 24401        ACKed
296    2022-12-27 20:31:05:302 CST *Re: DATA_seq: 24501        ACKed
297    2022-12-27 20:31:05:302 CST *Re: DATA_seq: 24601        ACKed
298    2022-12-27 20:31:05:302 CST *Re: DATA_seq: 24701        ACKed
299    2022-12-27 20:31:05:302 CST *Re: DATA_seq: 24801        ACKed
300    2022-12-27 20:31:05:305 CST *Re: DATA_seq: 24901        ACKed
301    2022-12-27 20:31:05:305 CST DATA_seq: 25001     ACKed
```

3s 的时间到了，接收方的计时器到时间后，发送方窗口判断窗口内还存在数据包，调用超时重传任务，一次性发送所有数据包。并且全部被正确接收。**实现了 GBN 的累计确认**

（6）RDT4.2 Selective-Repeat

在 GBN 协议中，对序号为 n 的分组的确认采用**累积确认**的方式，这表明接收方在接收到序号为 n 的分组确认后，表示在序号为 n 以前包括 n 的所有分组已经正常发送到接收方。但是在传输过程中，因为分组可能丢失、出错导致窗口错误，比如接收方缓冲区大小为 3 情况下，在发送方发送分组的过程，2 号分组丢失，1 号分组确认收到后，接收方窗口不会向前滑动，因为 2 号分组的窗口是空的，而再接下来发送方在等待 2 号分组的确认信息是等不到的，所以最后触发**超时重传**。

这也是 GBN 与 SR 协议不同的地方，如果一个序号为 n 的分组被正确接收到，并且按序（即上次交付给上层的数据的序号为 n - 1），则接收方为分组 n 发送一个 ACK，并将数据部分交付给上层。其它情况所有情况（例如序号没有按序）都会丢失该分组，为了解决 GBN 协议中出些单个分组的差错时就可能会引起大量的重传的情况。**SR 协议只会重传那些发送方认为在接收方出现差错的分组**，避免了不必要的重传，极大的提升了效率。所以需要建立**接收缓冲区**。

1、发送方窗口修改

①从上层收到数据：与 GBN 相似，SR 发送方首先会检查下一个用于该组的序号是否在滑动窗口内，如果在，直接打包。如果不在，**要么缓存要么将数据返回给上层，过一段时间后再传输**；

②超时：超时就会重传对于的分组。（这里主要因为是要单独传送分组，所以**每个分组都拥有一个自己的逻辑定时器**）

③收到 ACK：收到的 ACK 可能的位置有以下两种情况：

分组序号在窗口内，SR 发送方将被确认的分组标记为已接收

分组序号等于 base，则**窗口基序号移动到最小序号的未确认分组处**。如果移动过程中有序号落在了窗口内的未发送分组，则发送这些分组

对 GBN 的滑动窗口作出修改，对于接受的 ACK，如果在发送方窗口内，则停止对应位置的计时器并删除（置为 NULL），若再次查找到当前计时器为空，代表着之前某个组发生了错误没有收到，当前的组为重传后的组，可以直接跳过。

```
if (base <= CurrentAck && CurrentAck < base + size) {  // 如果收到的ACK在窗口范围内
    if (timers[CurrentAck - base] == null) {  // 表示接收到重复ACK，什么也不做
        return;
    }
```

如果位于窗口左沿的 ACK 到达，无论是发生错误的还是又重传接受的，都需要进行窗口的移动，建立窗口左沿的位置标识 **leftMoveIndex**，标识最小还未 ACK 的 seq 号

```
int leftMoveIndex = 0;  // 窗口左沿应该移动到的位置：最小未ACK的分组
```

窗口左沿一直移动到 nextseq 的位置

```
while (leftMoveIndex + 1 <= nextseq && timers[leftMoveIndex] == null) {
    leftMoveIndex ++;
}
```

将窗口内的包左移，相当于窗口右移（移动的是 packets 数组，所以每次都需要清空之前的数组位置，添加新的数据包，相当于移动）

```
for (int i = 0; leftMoveIndex + i < size; i++) {  // 将窗口内的包左移
    packets[i] = packets[leftMoveIndex + i];
    timers[i] = timers[leftMoveIndex + i];
}

for (int i = size - (leftMoveIndex); i < size; i++) {  // 清空已左移的包原来所在位置处的包和
    packets[i] = null;
    timers[i] = null;
}
```

更新窗口左沿的位置和下一个插入包的位置：

```
base += leftMoveIndex;  // 移动窗口左沿至leftMoveIndex处
nextseq -= leftMoveIndex;  // 移动下一个插入包的位置
```

同时注意，SR 中为每个组添加计时器，故应用数组设置计时器组。修改 putPacket()函数，让其为 packet 数组每个位置分配 timers 数组。

```
public void putPacket(TCP_PACKET packet) {
    packets[nextseq] = packet;      // 在窗口的插入位置放入包
    timers[nextseq] = new UDT_Timer();  // 为新放入窗口内的包增加计时器
    UDT_RetransTask task = new UDT_RetransTask(client, packet); // 设置重传任务
    timers[nextseq].schedule(task, 3000, 3000); // 每隔3s执行一次重传，直到收到ACK为止
    nextseq++;  // 更新窗口的插入位置
```

注意的是：修改这里的重传任务，只需要重传当前的数据包即可：

```
public void run() {
    System.out.println("超时重发包");
    senderClient.send(packets);
}
```

2、接收方修改

修改 rdt_recv 函数，修改有序和无序的区分，只要接收到的 ACK 是正确的且满足在接收窗口的范围内，则可以进行回复。

```
public void rdt_recv(TCP_PACKET recvPack) {
    //检查校验码，生成ACK
    if(CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
        int currentACK = -1;  // 当前包的seq
        try {
            currentACK = this.window.rcv(recvPack.clone());
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }

        if (currentACK != -1) {
            //生成ACK报文段（设置确认号）
            tcpH.setTh_ack(currentACK * 100 + 1);  // 设置确认号为收到的TCP分组的seq
            ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());  // 新建一个TCP分组（ACK）
            tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));  // 设置ACK的校验位

            reply(ackPack);  // 回复ACK报文段
        }
    }
}
```

3、接收方窗口建立

具体参考 SenderWindow 和 TCP_Receiver，定义如下参数：

```
private Client client;
private int size = 16;
private int base = 0;
private TCP_PACKET[] packets = new TCP_PACKET[size];
private int counts = 0;
public int expextedseq=0;//期待的序号
//public LinkedList<TCP_PACKET>packets = new LinkedList<TCP_PACKET>();//缓存列表
 Queue<int[]> dataQueue = new LinkedList<int[]>();//提交数据

public RcvWindow(Client client){
    this.client=client;
}
```

在 SR 的接受方窗口中，要根据窗口范围判断两种情况：

序号在【rcv_base ，rcv_base + N - 1】内的分组被正确接收。正常情况下，落在接收窗口内，将会**返回 ACK**，如果分组以前没有收到过，缓存该分组。**如果序号等于接收窗口的基序号**，那么会向上交付当前缓存的分组序号，然后接收窗口向前滑动。

序号在【rcv_base - N，rcv_base - 1】内的分组被收到（表面这个分组已经被接收过了），那么会产生一个 ACK 返回。其原因我查找的是，接收方和发送方的窗口并不总是一致的，可能在某一次发送中，发送方重传了一个分组落在接收窗口的前面而非里面，如果不发送 ACK 给接收方，那么发送方的窗口可能会无法向前移动。

只要接收到 ACK，无论是失序还是正序，都要返回 ACK 值给接收方，生成回复报文。

```
public int rcv(TCP_PACKET rcvpacket) {
    //如果大于等于期待的seq，则将数据包缓存。
    int CurrentAck=(rcvpacket.getTcpH().getTh_seq()-1)/100;

    if (CurrentAck < base) {  // [rcvbase-N, rcvbase-1]
        if (base - size <= CurrentAck && CurrentAck <= base - 1) {
            return CurrentAck;  // 对于失序分组也要返回ACK
        }
    } else if (CurrentAck <= base + size - 1) {  // [rcvbase-N, rcvbase+N-1]
        packets[CurrentAck - base] = rcvpacket;  // 对于正确分组，加入窗口中.

        if (CurrentAck == base) {  // 接受到的分组位于窗口左沿

            // 滑动窗口
            slid();

            // 交付数据
            if (dataQueue.size() >= 20 || base == 1000) {
                deliver_data();
            }

        }

        return CurrentAck;  // 返回ACK
    }

    return -1;  // 错误返回-1
}
```

如果序号=base，接收窗口要进行滑动，编写 Slid()函数：
建立窗口左沿的位置标识 **leftMoveIndex**，标识最小还未 ACK 的 seq 号

```
int leftMoveIndex = 0;  // 用于记录窗口左移到的位置：最小未收到数据包处
while (leftMoveIndex <= size - 1 && packets[leftMoveIndex] != null) {
    leftMoveIndex ++;
}
```

将已收到的分组加入缓冲队列，等待交付，并右移窗口，将包置空，窗口左沿加上移动的距离：

```
for (int i = 0; i < leftMoveIndex; i++) {  // 将已接收到的分组加入交付队列
    dataQueue.add(packets[i].getTcpS().getData());
}

for (int i = 0; leftMoveIndex + i < size; i++) {  // 剩余位置的包左移
    packets[i] = packets[leftMoveIndex + i];
}

for (int i = size - (leftMoveIndex); i < size; i++) {  // 将左移的包原来位置处置空
    packets[i] = null;
}

base += leftMoveIndex;  // 移动窗口左沿
```

添加 deliver_data()函数，参考原 TCP_Receiver 函数：

```
public void deliver_data() {
            //检查dataQueue，将数据写入文件
            File fw = new File("recvData.txt");
            BufferedWriter writer;

            try {
                writer = new BufferedWriter(new FileWriter(fw, true));

                //循环检查data队列中是否有新交付数据
                while(!dataQueue.isEmpty()) {
                    int[] data = dataQueue.poll();

                    //将数据写入文件
                    for(int i = 0; i < data.length; i++) {
                        writer.write(data[i] + "\n");
                    }

                    writer.flush();      //清空输出缓存
                }
                writer.close();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

}
```

验证 LOG 文件：

```
2022-12-29 18:30:39:512 CST DATA_seq: 1001      ACKed
2022-12-29 18:30:39:528 CST DATA_seq: 1101      NO_ACK
2022-12-29 18:30:39:543 CST DATA_seq: 1201      ACKed
```

Seq=1101 的数据包，ACK 出错

```
2022-12-29 18:30:39:513 CST ACK_ack: 1001
2022-12-29 18:30:39:530 CST ACK_ack: 1101   LOSS
```

发送方没有接收到 1101 的数据包，显示丢失

```
2022-12-29 18:30:39:762 CST DATA_seq: 2601      ACKed
2022-12-29 18:30:42:540 CST *Re: DATA_seq: 1101     ACKed
2022-12-29 18:30:42:542 CST DATA_seq: 2701      ACKed
```

3s 后，计时器到时，只单独重传 1101 数据包。**实现 SR 功能的失序缓存**。

且 GBN 的实现过程中，仅使用 1 个超时计时器完成了超时重传，**实现了 TCP 的基本功能**。

（7）RDT5.0 Tahoe

TCP 最早的版本称之为 Tahoe。TCP Tahoe 主要有三个机制去控制数据流和拥塞窗口: slow start (慢开始), congestion avoidance (拥塞避免), and fast retransmit(快重传)。

1、发送方窗口修改

采用数据结构 **HashMap**，其思路来自于 C++的 Map，构造一个散列表 packets，用于缓存序号与数据包的键值对应关系。同时定义 MapHead，指向链表的头部。

```
private Map<Integer,TCP_PACKET> packets =  new LinkedHashMap<Integer, TCP_PACKET>();//储存数据包
private int mapHead=0;
```

修改 putPacket 函数：

先计算当前接受到数据包的 ACK，若加入包后不存在计时器，即 packet 为空，则开启计时器并设置重传任务，利用 put 方法加入键值对（CurrentSeq, packet）

```
int CurrentSeq=(packet.getTcpH().getTh_seq()-1)/100;
this.packets.put(CurrentSeq, packet);

if(this.timer==null) {
    this.timer = new UDT_Timer();    // 设置计时器
    this.task = new My_UDT_RetransTask(client, this);    // 设置重传任务
    this.timer.schedule(task, 3000,3000);
}
```

快重传：维护一个 depseqcount，如果收到重复 ACK，则 depseqcount++，如果 depseqcount = 4（初值为 1），代表连续收到 3 个重复 ACK，执行快重传。因为 3 个重复 ACK 为对上一个的重复，重传的包时下一个，所以要利用 get 方法获取 cerrentACK+1 的数据包，并重启计时器。

```
// 快重传
TCP_PACKET packet=this.packets.get(CurrentAck+1);
// 连续收到3个对上一个包的ACK 获取下一个应该重传的包的ACK
if(packet!=null){
    System.out.println("连续收到3个ACK");
    System.out.println("执行快重传");
    System.out.println();

    this.client.send(packet);

    if(this.timer!=null){
        this.timer.cancel();
    }
    this.timer = new UDT_Timer();
    this.task = new My_UDT_RetransTask(client, this);
    this.timer.schedule(task, 3000,3000);
}
// 执行快恢复
fastRecovery();
```

快恢复：建立快恢复 FastRecovery 函数。

在 Tahoe 版本，其实不需要快恢复函数，因为这是与 Reno 版本的区别。为了方便我在这里先写好快恢复。

快恢复为连续收到 3 个重复 ACK 后，要进行快重传，同时，发送方知道现在只是丢失了个别的报文段，于是调整门限 ssthresh = cwnd / 2，同时设置 cwnd = 1，并开始执行拥塞避免阶段。

首先判断 ssthresh 不得小于 2，其次再执行除 2 操作。

```
private void fastRecovery() {
    System.out.println("********************");
    System.out.println("快恢复阶段");
    System.out.println("********************");

    this.ssthresh = this.cwnd/2;
    if(this.ssthresh < 2)
        // ssthresh 不得小于2
        this.ssthresh = 2;
    // Tahoe cwnd 变为 1
    this.cwnd = 1;
    //this.cwnd = this.ssthresh;

    System.out.println("cwnd is :"+this.cwnd);
    System.out.println("ssthresh is :"+this.ssthresh);
}
```

慢开始与拥塞避免：

如果没有收到重复 ACK，由于是累计确认，故可以对发送窗口中当前 ACK 之前的分组进行删除，并将链表的首部置为下一个（当前的已被收到确认即可删除），并清空计时器。如果窗口中还有包，则要重启计时器并设置重传任务。用 dupseq 记录当前的 ACK 号，以便下一个 ACK 到来与上一

个比较，并将 dupseqcount 重置为 1。

```
// 收到新的ACK 清除前面的包
for(int i=this.mapHead;i<=CurrentAck;i++){
    this.packets.remove(i);
}

// 链表的起始位置为当前的下一个
this.mapHead=CurrentAck+1;

// 清空计时器
if(this.timer!=null){
    this.timer.cancel();
}

// 如果窗口中仍有分组 重开计时器
if(this.packets.size()!=0){
    this.timer= new UDT_Timer();
    this.task = new My_UDT_RetransTask(client, this);
    this.timer.schedule(task, 3000,3000);
}

this.dupseq = CurrentAck;    // dupseq记录当前ACK 在下一个包到来后 相当于之前的ACK
this.dupseqcount=1;          // 重置为1
```

慢开始：如果 cwnd < ssthresh，则启动慢开始，每一次接收到 ACK，就使 cwnd ++。

```
// 慢开始
if(this.cwnd < this.ssthresh){
    this.cwnd++;     // 每收到一个ACK cwnd++
```

拥塞避免：如果 cwnd ≥ ssthresh，则启动拥塞避免。设置拥塞避免计数器 count，使 count++。cwnd = cwnd + count / cwnd。每次对 count++，当 count == cwnd，即一轮后，cwnd++，并清空 count。

```
}else{
    // 拥塞避免
    this.count++;
    if(count>=this.cwnd){
        this.count=this.count-this.cwnd;
        this.cwnd++;
    }
}
```

2、重传任务修改

无论是超时，还是 3 个重复 ACK，都要进行重传，重传要遵循“**乘法减小**”。要把门限值 ssthresh 设置为当前拥塞窗口 cwnd 的一半，并减少 cwnd = 1。与快恢复的过程类似。

```
public void multiDecrease() {
    System.out.println("********************");
    System.out.println("乘法减小");
    System.out.println("********************");
    System.out.println("cwnd is :"+this.cwnd);
    System.out.println("ssthresh is :"+this.ssthresh);

    this.ssthresh=this.cwnd/2;
    if(this.ssthresh<2)
        this.ssthresh=2;
    this.cwnd=1;

    System.out.println("cwnd is :"+this.cwnd);
    System.out.println("ssthresh is :"+this.ssthresh);
}
```

乘法减少后，要重传数据包：首先要清空计时器，从链表头指针 mapHead 开始，搜寻当前拥塞窗口的所有数据包，利用 map 结构的“由键找值”，将数据包依次发送出去，如果发送完还存在数据包，那么要重新设置计时器。

```
public void retrand() {
    System.out.println("********************");
    System.out.println("开始重传");
    System.out.println("********************");

    this.timer.cancel();
    for(int i=this.mapHead,t=0;t<this.packets.size();t++,i++){
        TCP_PACKET packet=this.packets.get(i);
        if(packet!=null){
            System.out.println("retrand:   "+(packet.getTcpH().getTh_seq()-1)/100);
            this.client.send(packet);
        }
    }
    if(this.packets.size()!=0){
        this.timer= new UDT_Timer();
        this.task = new My_UDT_RetransTask(client, this);
        this.timer.schedule(task, 3000,3000);
    }
}
```

3、接收方窗口修改

接收方窗口要将失序变成有序，所以每收到一个包，就要根据他的 ACK 号，插入对应位置。故我们可以维护一个存储数据包的数据结构：链表。建立 LinkList。

```
public LinkedList<TCP_PACKET> packets = new LinkedList<TCP_PACKET>();//缓存列表
```

修改 rcv 函数，同样需要维护一个变量记录当前的 ACK 值：currentACK，并建立 index 作为指针，正常情况下，均是 currentACK ≥ expextedseq，即当前接受到的理应等于我所期待的值，那么每一次 index 均为 0，加入数据包在链表的头部插入，这样 getFirst()也是读取链表的头，读取完加入 data 队列后，同时通过 poll()方法，从链表中删除，这样，链表的长度始终为 0；如果发生了超时，或者收到重复 3 个 ACK，那么 packets 便要缓冲这 3 个数据包，同时链表的长度增加，但是 expextedseq 不会增加，需要等到重传后才会增加，这个我会在后面结合 LOG 文件介绍。

```
public int rcv(TCP_PACKET rcvpacket) {
    //如果大于等于期待的seq，则将数据包缓存。
    int CurrentAck = (rcvpacket.getTcpH().getTh_seq()-1)/100;
    if(CurrentAck>=this.expextedseq){
        //找到合适的位置存放数据包
        int index=0;
        while(index < this.packets.size() && CurrentAck > (this.packets.get(index).getTcpH().getTh_s
            index++;
        }
        if(index==this.packets.size()||CurrentAck!=(this.packets.get(index).getTcpH().getTh_seq()-1)
            this.packets.add(index,rcvpacket);
        }
    }

    //滑动窗口
    this.slid();
    System.out.println("this.expextedseq:"+this.expextedseq);
    return this.expextedseq-1;
}
```

修改 slid()滑动窗口函数：正常情况为收到一个包，加入链表，从链表删除，加入 data 队列；非正常情况则需要缓冲未发送的包，到时候一次全部发送。

```
public void slid() {
    //第一个数据包收到，就滑动
    while(!this.packets.isEmpty()&&(this.packets.getFirst().getTcpH().getTh_seq()-1)/100==this.expex
        this.dataQueue.add(this.packets.poll().getTcpS().getData());
        this.expextedseq++;
    }
    //累积到20个包或者到发送结束，向上提交数据
    if(this.dataQueue.size()>=20||this.expextedseq==1000){
        this.deliver_data();
    }

}
```

验证 LOG 文件：

慢开始与拥塞避免：

```
***** Change List Show *****
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 19, 19, 1, 2, 3,
[16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16,
```

为解释方便，我分别打印了 cwnd 和 ssthresh 的变化过程。

慢开始阶段，cwnd 从 1 开始，每收到 1 个 ACK，cwnd++，达到 ssthresh=16 后，转为拥塞避免阶段，每收到 16 个 ACK 才会使 cwnd++。当出现超时重传时，令 cwnd 降为 1。

快恢复：当收到连续 3 个重复 ACK 时，执行快恢复，将 cwnd 由 21 降到 1。且 ssthresh=cwnd/2=10（ssthresh 为 int 型，取整）

```
********************
cwnd is: 21
ssthresh is: 16

连续收到3个ACK
执行快重传

********************
快恢复阶段
********************
cwnd:21--->1
ssthresh is :10
```

快重传：

发送 12401 数据包出错，接收方没有收到 ACK，于是发送方发送了 12501,12601,12701 三个包，此时接收方均返回 12301。发送方收到 3 个重复 ACK，重发 12401 包。

```
-> 2022-12-31 11:22:25:897 CST
** TCP_Receiver
   Receive packet from: [192.168.6.1:9001
   Packet data: 132857 132859 132863 1328
   PACKET_TYPE: DATA_SEQ_12401
-> 2022-12-31 11:22:25:908 CST
** TCP_Receiver
   Receive packet from: [192.168.6.1:9001
   Packet data: 134059 134077 134081 1340
   PACKET_TYPE: DATA_SEQ_12501
CurrentACK:125
Index:0
size:0
size:125
expextedseq:124
-> 2022-12-31 11:22:25:909 CST
** TCP_Sender
   Receive packet from: [192.168.6.1:9002
   Packet data:
   PACKET_TYPE: ACK_12301
Receive ACK Number: 12301
```

```
********************
快恢复阶段
********************
cwnd:21--->1
ssthresh is :10
-> 2022-12-31 11:22:25:930 CST
** TCP_Receiver
   Receive packet from: [192.168.
   Packet data: 132857 132859 132
   PACKET_TYPE: DATA_SEQ_12401
CurrentACK:124
Index:0
size:3
size:124
```

至此，基本实现了 TCP Tahoe 和 Reno 的基本功能，且在 SR 的基础上，去掉了计时器组，发送方始终使用一个超时计时器。

（8）RDT5.1 Reno

基于 Tahoe 版本，需要修改的部分为快恢复部分，在连续收到 3 个重复 ACK 时，Reno 需要将 cwnd = ssthresh。

```
private void fastRecovery() {
    System.out.println("********************");
    System.out.println("快恢复阶段");
    System.out.println("********************");

    this.ssthresh = this.cwnd/2;
    if(this.ssthresh < 2)
        // ssthresh 不得小于2
        this.ssthresh = 2;
    // Tahoe cwnd 变为 1
    System.out.println("cwnd:"+this.cwnd+"--->1");
    //this.cwnd = 1;
    this.cwnd = this.ssthresh;


    System.out.println("ssthresh is :"+this.ssthresh);
}
```

验证 LOG 文件

①出错

```
2022-12-31 11:52:38:066 CST DATA_seq: 18401 WRONG   NO_ACK
2022-12-31 11:52:38:076 CST DATA_seq: 18501     NO_ACK
2022-12-31 11:52:38:088 CST DATA_seq: 18601     NO_ACK
2022-12-31 11:52:38:098 CST DATA_seq: 18701     ACKed
2022-12-31 11:52:38:099 CST *Re: DATA_seq: 18401        NO_ACK
```

如图，seq=18401 的数据包出错，此时接收方对数据包 18401，18501，18601 的回应 ACK 均为 18301。这时需要用到**缓存失序分组**：

```
while(index < this.packets.size() && CurrentAck > (this.packets.get(index).getTcpH().getTh_seq()-1),
    index++;
}

System.out.println("CurrentACK:"+CurrentAck);
System.out.println("Index:"+index);
System.out.println("size:"+this.packets.size());


if(index==this.packets.size()||CurrentAck!=(this.packets.get(index).getTcpH().getTh_seq()-1)/100){
    this.packets.add(index,rcvpacket);
    System.out.println("size:"+(this.packets.get(index).getTcpH().getTh_seq()-1)/100);
}
```

如果是有序分组，index=0，如果数据包出错，像 18401 出错，那么接收方只能缓存发送过来的 18501,18601。当 18401 重发被接收后，index 置为 0，将 18401 重新添加到 data 队列的头部，并一次全部发送出去，清空 data 队列。注意的是，因为 18401 没有接受到，接收窗口的期待序列（expextedseq）一直为 18401，故其返回对上一个已确认的最小的 ACK=18301。直到这三个数据包发送完，expextedseq 置为 18801。**一开始在这里会有疑问，为什么是+4 不是+3，不是收到 3 个重复 ACK 吗？**其实还有一个重传的 ACK，这也体现了什么是“快”重传。这里我下面会说道。

```
2022-12-31 11:52:38:056 CST ACK_ack: 18301
2022-12-31 11:52:38:077 CST ACK_ack: 18301
2022-12-31 11:52:38:088 CST ACK_ack: 18301
2022-12-31 11:52:38:099 CST ACK_ack: 18301
2022-12-31 11:52:38:100 CST ACK_ack: 18701
2022-12-31 11:52:38:110 CST ACK_ack: 18801
```

查看 LOG 记录，我们会发现对 18701 产生了 ACK，而对重发的 18401 没有 ACK。注意时间，这里接收方做出的 ACK，不是收到 18701 分组产生的，其实还是在回复 18301，同时缓存了 18701 分组，并立即执行快重传 18401。

快重传后，接收方已经缓存了 18501,18601,18701，此时最大的 seq=18701，故返回的 ACK 也为 18701。这也是为什么 18701 有 ACK，而 18401 没有 ACK 的原因。因为重传的 18401 已经加入到上面的缓存队列，缓存队列每发送一个，expextedseq++,最终的 expextedseq=18801，即返回 expextedseq-1=18701 的 ACK。

②ACK 出错

在 LOG 文件里，我们会看到其中偶尔会出现 NO_ACK，这个我在 RDT2.0 中说过，代表 ACK 出错。

```
2022-12-31 11:52:39:024 CST DATA_seq: 27501     ACKed
2022-12-31 11:52:39:035 CST DATA_seq: 27601     NO_ACK
2022-12-31 11:52:39:045 CST DATA_seq: 27701     ACKed
```

```
2022-12-31 11:52:39:024 CST ACK_ack: 27501
2022-12-31 11:52:39:035 CST ACK_ack: -1522835042    WRONG
2022-12-31 11:52:39:045 CST ACK_ack: 27701
```

在 TCP_Sender 的 recv 函数中，没有对校验和失败的数据包进行处理。即虽然发送方没有收到正确的 ACK，但也没有做出回应。因为使用**累计确认**的原因，显然发送方已经收到了更高序号的分组 ACK，则可以认为这个 ACK 出错的分组已经被接收到了。

③丢失

同理出错的逻辑，接受到 3 个重复 ACK 执行快重传。具体过程我在①中已经阐释。

```
2022-12-31 11:52:36:199 CST DATA_seq: 5401  LOSS    NO_ACK
2022-12-31 11:52:36:215 CST DATA_seq: 5501      NO_ACK
2022-12-31 11:52:36:231 CST DATA_seq: 5601      NO_ACK
2022-12-31 11:52:36:247 CST DATA_seq: 5701      ACKed
2022-12-31 11:52:36:248 CST *Re: DATA_seq: 5401     NO_ACK
```

```
2022-12-31 11:52:36:168 CST ACK_ack: 5201
2022-12-31 11:52:36:185 CST ACK_ack: 5301
2022-12-31 11:52:36:215 CST ACK_ack: 5301
2022-12-31 11:52:36:231 CST ACK_ack: 5301
2022-12-31 11:52:36:247 CST ACK_ack: 5301
2022-12-31 11:52:36:248 CST ACK_ack: 5701
```

Seq=5401 数据包丢失，则连续发送三个对 5301 的 ACK，最后重发 5401，整体进行累计确认，发送对 5701 的 ACK。

④发送方接收 ACK 丢失

同理②，采用累计确认，只需对最高 ACK 确认即可。

```
2022-12-31 11:52:42:962 CST DATA_seq: 63901      ACKed
2022-12-31 11:52:42:973 CST DATA_seq: 64001      NO_ACK
2022-12-31 11:52:42:984 CST DATA_seq: 64101      ACKed
```

```
2022-12-31 11:52:42:963 CST ACK_ack: 63901
2022-12-31 11:52:42:974 CST ACK_ack: 64001  LOSS
2022-12-31 11:52:42:985 CST ACK_ack: 64101
```

⑤延迟

```
2022-12-31 11:52:39:403 CST DATA_seq: 31101 DELAY   NO_ACK
2022-12-31 11:52:39:413 CST DATA_seq: 31201      NO_ACK
2022-12-31 11:52:39:423 CST DATA_seq: 31301      NO_ACK
2022-12-31 11:52:39:433 CST DATA_seq: 31401      ACKed
2022-12-31 11:52:39:434 CST *Re: DATA_seq: 31101         NO_ACK
```

```
2022-12-31 11:52:39:383 CST ACK_ack: 30901
2022-12-31 11:52:39:393 CST ACK_ack: 31001
2022-12-31 11:52:39:414 CST ACK_ack: 31001
2022-12-31 11:52:39:424 CST ACK_ack: 31001
2022-12-31 11:52:39:434 CST ACK_ack: 31001
2022-12-31 11:52:39:434 CST ACK_ack: 31401
```

延迟的情况略有不同，一般延迟的分组没有及时得到 ACK，所以没有算进重复 ACK 里，但观察 LOG 文件，我们可以在最后看到对延迟的回复。

```
CLIENT HOST TOTAL    SUC_RATIO    NORMAL  WRONG    LOSS     DELAY
192.168.6.1:9001     1008     96.03%  1000     3    3    2
```

```
2022-12-31 11:54:48:503 CST ACK_ack: 99801
2022-12-31 11:54:48:514 CST ACK_ack: 99901
2022-12-31 11:55:09:319 CST ACK_ack: 99901
2022-12-31 11:55:13:698 CST ACK_ack: 99901
```

在以上过程中，总共有 2 组延迟，因而在所有数据包（99901）发送完，转而对延迟数据包进行确认。由于已接收完全部分组，故回复对已确认最大分组序号（99901）的 ACK。

⑥ACK 延迟

```
2022-12-31 11:54:44:720 CST DATA_seq: 63401      ACKed
2022-12-31 11:54:44:730 CST DATA_seq: 63501      NO_ACK
2022-12-31 11:54:44:740 CST DATA_seq: 63601      ACKed
```

```
2022-12-31 11:54:44:720 CST ACK_ack: 63401
2022-12-31 11:54:44:731 CST ACK_ack: 63501  DELAY
2022-12-31 11:54:44:741 CST ACK_ack: 63601
```

窗口变化情况：

```
***** Change List Show *****
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 22, 22,
11, 12, 13, 14, 15, 16, 17, 17, 17,
8, 9, 10, 11, 12, 12, 12,
6, 7, 8, 9, 9, 9,
4,
2, 2, 2, 2, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 14, 14,
7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 28, 28,
14, 15, 16, 17, 18, 18, 18,
9, 10, 11, 12, 12, 12,
6, 7, 8, 9, 10, 11, 12, 12, 12, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

[16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16,
11, 11, 11, 11, 11, 11, 11, 11, 11,
8, 8, 8, 8, 8, 8, 8,
6, 6, 6, 6, 6, 6,
4,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
14, 14, 14, 14, 14, 14, 14,
9, 9, 9, 9, 9, 9,
6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6]
***** Show End *****
```

为了区分，每一次快重传后的 cwnd 变化均按行区分，可以看出，ssthresh 的大小与每一行第一个 cwnd 相同。

慢开始：

```
********************
cwnd is: 2
ssthresh is: 16

********慢开始********
cwnd:2--->3
```

拥塞避免：

使用计时器，当 ACK 达到 cwnd 个数时，cwnd++。

```
********************
cwnd is: 18
ssthresh is: 16

*******拥塞避免*********
cwnd:18--->19
```

快重传快恢复：

```
********************
cwnd is: 20
ssthresh is: 16

连续收到3个ACK
执行快重传

********************
快恢复阶段
********************
cwnd:20--->10
ssthresh is :10
```

乘法减小：

因 3 重复 ACK 的快重传时间远短于超时计时器相应，所以缩短超时计时器，我们可以看到重传过程中乘法减小过程：

```
********************
乘法减小
********************
cwnd is :8
ssthresh is :8
cwnd:8--->1
ssthresh is :4
```

2. 未完全完成的项目，说明完成中遇到的关键困难，以及可能的解决方式。（2 分）

RDT1.0
RDT2.0 RDT2.1 RDT2.2
RDT3.0
RDT4.0 流水线协议 RDT4.1 Go-Back-N RDT4.2 SR
RDT5.0 Tahoe RDT5.1 Reno
所有版本均已完成

3. 说明在实验过程中采用迭代开发的优点或问题。(优点或问题合理：1 分)

优点：

①难度中上，整体由简单到复杂。RDT2.0 到 3.0 的迭代过程比较容易，通过查找资料或翻阅课本，主要围绕校验和过程构造发送方与接收方之间的通信过程；从 RDT4.0 开始需要用到滑动窗口，需要考察我们的编程能力和对 TCP 的理解能力。

②激发学生的探索兴趣。每实现一个版本，都会产生巨大的成就感。在做完前面几个版本后，我也有了动力去完成后面的版本。在 RDT4.0 实现流水线协议和缓冲区那里我卡了很长时间，同时因为感染了新冠阳性一直没时间完成，身体恢复后硬着头皮继续做，最终也是凭着兴趣坚持到底完成了这次实验。

③代码复用性好。每一个版本大多沿用前面版本的代码，根据前面，这样的迭代既符合历史发展规律，即开发过程弃用 Tahoe 到 Reno，又满足面向对象编程的代码重用性思想，同时减轻我们的编程压力。

4. 总结完成大作业过程中已经解决的主要问题和自己采取的相应解决方法(1 分)

问题①：SR 中接收方窗口左沿范围问题。

一开始，我默认接收到的 ACK 都落在【rcv_base ，rcv_base + N - 1】内。因为按照正常情况，发送方和接收方的窗口范围基本上是一致的，只有不同时间段上出现差异。但在测试过程中，偶尔会出现发送方卡死的情况，一开始我没太注意，直到在 CSDN 上搜索 GBN 和 SR 的区别时，看到了 SR 中接收方窗口的两种情况。

如果序号在【rcv_base - N，rcv_base - 1】内的分组被收到，表面上这个分组已经被接收过了，但实际上可能出现差错导致接收方窗口右移完，这个包才发送过来，那么会产生一个 ACK 返回。其原因我查找的是，接收方和发送方的窗口

并不总是一致的，可能在某一次发送中，发送方重传了一个分组落在接收窗口的前面而非里面，如果不发送 ACK 给接收方，那么发送方的窗口可能会无法向前移动。

解决方法：重新更新 CurrentAck 的判断条件，让其满足在【rcv_base - N，rcv_base - 1】内。

问题②：如何定位重传数据包？

TCP 对单一计时器的实现，是只重传第一个未应答包。一开始我打算建立索引，哈希成 Map 键值对。但对于 SR 来讲，窗口大小是固定的，但对于后面的可变窗口来说并不可靠。这里我采用统一的思路，即缓冲区只存放失序的包，有序的包只要接收到即“丢出”缓冲区。故这里也想到采用数据结构中的链表，为什么不选择栈？是因为我需要把无序包变成有序包，而栈先进先出的顺序，导致重传包必定排在栈顶，依然是无序的。故重传数据包，只需要重传所有在 packets 中的数据包即可，所有已经 ACK 的包均“丢出”packets，这样即可解决索引构造的难题。

解决方法：构造窗口项的 clone()方法与从链表移除方法 poll()。

问题③：拥塞避免的实现方法。

一开始我对 cwnd 的增加挺迷惑的。如果说慢开始，每收到 1 个 ACK 则窗口+1。那拥塞避免期间该怎么处理？按理来说是一个 RTT（轮次）+1，但对于轮次又该怎么定义？我查找到拥塞避免算法为 cwnd = cwnd + 1 / cwnd，然后按照这个方法写一下，发现不行。然后看到我一直加的是 1 / cwnd，这样永远也加不到 2 倍 cwnd。

解决方法：采用拥塞避免计时器 count，在拥塞避免阶段每收到正确的 ACK，则使 count++。这样如果 cwnd=16，那么需要 16 个 ACK，使 cwnd+1。

5. 对于实验系统提出问题或建议(1 分)

①建议采用可视化 GUI，LOG 文件看起来有点冗杂，并且某些地方的 ACK 很难看出来。同时控制台信息太多导致很难找到需要的信息，需要一次一次只运行一点来找。

②虽然某些打包好的 jar 包并不需要修改，但是在 eclipse 编译器里打不开，在 Idea 里能打开但没法运行，虽然不影响实验过程，但多少有点麻烦。

③实验还是有难度的，虽然假期时间比较充裕，但还是卡着 DDL 写完，建议降低下难度。