计算机网络大作业报告

学号：19020011038 姓名：岳宇轩 专业：慧与 年级：2019

评分标准

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。如果某个功能完成有错误，按照对应功能分值扣分，比如学生自称做到了 3.0，但发现校验码函数不正确，扣掉校验码的三分。

   评分主要检查点：

   (1) 2.0：有没有正确完成校验码函数 P3

   (2) 2.2：有没有正确完成出错处理的控制逻辑。 P7

   (3) 3.0：有没有正确完成丢包处理的控制逻辑。 P9

   (4) GB/SR/TCP：有没有正确完成滑动窗口的管理；GB 关注累积确认 P11；SR 关注接收方有没有进行失序报文缓存 P16；TCP 同时关注累积确认、失序缓存以及发送方是否仅使用 1 个超时计时器。。

   (5) TCP Tahoe：检查发送窗口有没有按照慢开始 P23、拥塞避免增长 P23，超时重发时有没有降到 1 P22；同时关注接收方发出的确认 P20 是否正确以及发送方是否仅使用 1 个超时计时器 P23。

   3）TCP Reno：检查有没有实现快重传 P23、快恢复 P23。

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：关键困难合理：1 分；解决方式可行：1 分) P38

3.说明在实验过程中采用迭代开发的优点或缺点。(优点或缺点合理：1 分) P38

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) P39

5.对于实验系统提出问题或建议(1 分) P41

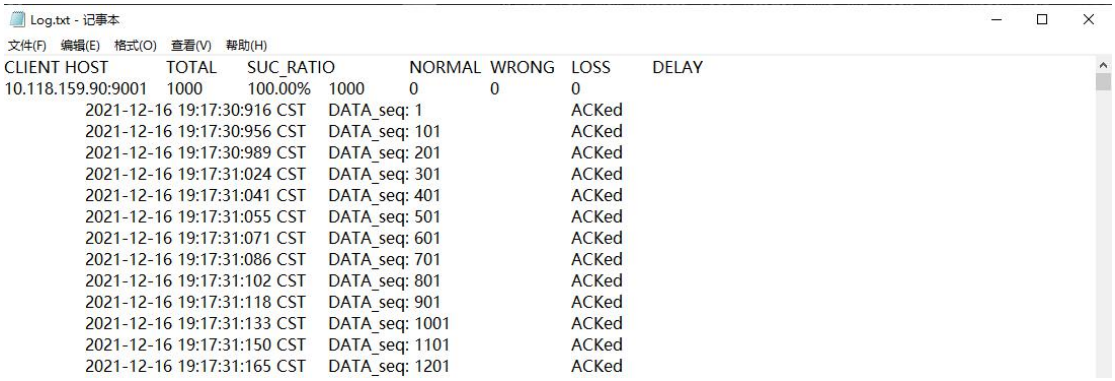
TCP Reno 在 P19

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）

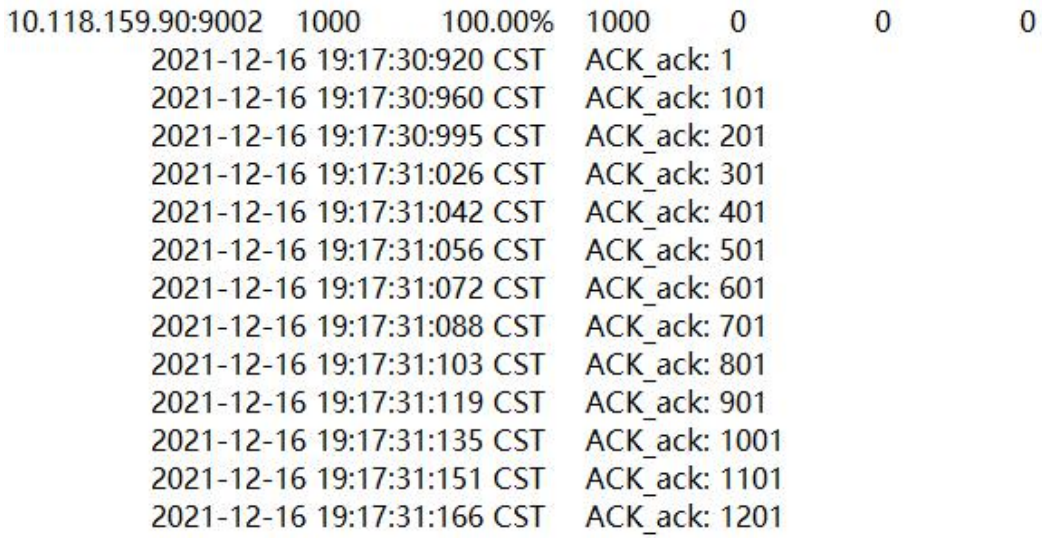
RDT 1.0

信道是可靠信道

发送方：



接收方：



在可靠信道上的传输成功率为 100%

RDT 2.0

• 接收方如何产生确认或代表否认的序号：
确认：ack 为接收到的包的 seq
否认：ack 为-1

• 接收方收到重复的包如何处理：
不做处理

• 发送方收到错误/重复确认：
错误：重传
重复：不做处理

过程：

信道问题只有出错

1. 思路是利用校验和判断 TCP 报文段是否出错，故首先修改 computeChkSum 函数，使用 CRC32 来计算校验码。

```
public static short computeChkSum(TCP_PACKET tcpPack) {
    CRC32 crc32 = new CRC32();
    TCP_HEADER header = tcpPack.getTcpH();  // 获取原TCP报文头部
    crc32.update(header.getTh_seq());  // 添加seq进行校验
    crc32.update(header.getTh_ack());  // 添加ack进行校验

    // 添加 TCP 数据字段进行校验
    for (int i = 0; i < tcpPack.getTcpS().getData().length; i ++ )
    {
        crc32.update(tcpPack.getTcpS().getData()[i]);
    }

    // 获取校验码，并返回
    return (short) crc32.getValue();
}
```

参数为一个 TCP 包，首先拿出它的头部，获取其中的 seq 字段和 ack 字段，再加上 TCP 包的数据字段，一同计算校验和，校验和可以通过 CRC32 的 getValue 方法返回。

发送方：

①首先将 udt_send 的 eflag 置为 1

②在发送方的 waitACK 状态，如果收到一个接收方发来的确认，就看一下它是 ACK 还是 NACK。

如果是 NACK 则继续等待 ACK，并且重新发包。

```
if (currentAck == -1){  // NACK
    System.out.println("Retransmit: "+tcpPack.getTcpH().getTh_seq());
    udt_send(tcpPack);  // 重新发包
    flag = 0; // 仍然是waitACK状态
```

如果是 ACK 则切换状态为等待应用层调用；

```
}else{  // ACK
    System.out.println("Clear: "+tcpPack.getTcpH().getTh_seq());
    flag = 1; // 切换为等待应用层调用状态
    //break;
}
```

接收方：

①首先将 reply 函数中的 eflag 置为 1
②接受方接受到一个包时，首先讲这个包送入 computeChkSum 函数计算校验和，并与所收到的包的头部中的校验和进行比较：

如果相等，证明 TCP 分组正确，没有出错，则发送 ACK 给发送方；
- 设置确认号 ack 为确认的分组的 seq
- 新建一个 ACK 包
- 计算校验和
- 回复 ACK 报文段

```
public void rdt_recv(TCP_PACKET recvPack) {
    //检查校验码，生成ACK
    if(CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {  // 计算并比对校验和，如果相等：
        //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），发往发送方
        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());
        sequence++;
```

如果不相等，则表明 TCP 分组出错，发送 NACK 给发送方；
- 设置确认号 ack= - 1，表示这是一个 NACK
- 打一个 NACK 包
- 计算校验和
- 回复 NACK 报文段

```
}else{
    System.out.println("Recieve Computed: "+CheckSum.computeChkSum(recvPack));  // 显示对于收到的分组的校验和计算结果
    System.out.println("Recieved Packet"+recvPack.getTcpH().getTh_sum());  // 显示收到的分组中的校验和
    System.out.println("Problem: Packet Number: "+recvPack.getTcpH().getTh_seq()+" + InnerSeq:  "+sequence);
    tcpH.setTh_ack(-1); // -1表示这是一个NACK
    ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());  // 打一个NACK包
    tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));  // 计算校验和

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

结果：

```
2021-12-21 22:48:27:754 CST   DATA_seq: 2101                    ACKed
2021-12-21 22:48:27:769 CST   DATA_seq: 2201                    ACKed
2021-12-21 22:48:27:785 CST   DATA_seq: 2301                    ACKed
2021-12-21 22:48:27:801 CST   DATA_seq: 2401                    ACKed
2021-12-21 22:48:27:817 CST   DATA_seq: 2501                    ACKed
2021-12-21 22:48:27:832 CST   DATA_seq: 2601                    ACKed
2021-12-21 22:48:27:850 CST   DATA_seq: 2701                    ACKed
2021-12-21 22:48:27:864 CST   DATA_seq: 2801                    ACKed
2021-12-21 22:48:27:880 CST   DATA_seq: 2901                    ACKed
2021-12-21 22:48:27:896 CST   DATA_seq: 3001        WRONG       NO_ACK
2021-12-21 22:48:27:898 CST   *Re: DATA_seq: 3001               ACKed
2021-12-21 22:48:27:917 CST   DATA_seq: 3101                    ACKed
2021-12-21 22:48:27:944 CST   DATA_seq: 3201                    ACKed
2021-12-21 22:48:27:960 CST   DATA_seq: 3301                    ACKed
2021-12-21 22:48:27:974 CST   DATA_seq: 3401                    ACKed
2021-12-21 22:48:27:990 CST   DATA_seq: 3501                    ACKed
```

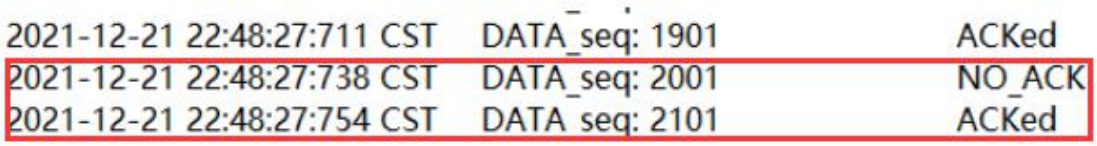
发送方发送 seq 为 3001 的包时，产生了一个 WRONG（如上图红框中第一行所示），于是接收方回复了一个 ack 为-1 的包（如下图红框所示），所以发送方重发了这个包（如上图红框中第二行所示）。

```
2021-12-21 22:48:27:755 CST   ACK_ack: 2101
2021-12-21 22:48:27:770 CST   ACK_ack: 2201
2021-12-21 22:48:27:786 CST   ACK_ack: 2301
2021-12-21 22:48:27:802 CST   ACK_ack: 2401
2021-12-21 22:48:27:818 CST   ACK_ack: 2501
2021-12-21 22:48:27:833 CST   ACK_ack: 2601
2021-12-21 22:48:27:851 CST   ACK_ack: 2701
2021-12-21 22:48:27:865 CST   ACK_ack: 2801
2021-12-21 22:48:27:881 CST   ACK_ack: 2901
2021-12-21 22:48:27:897 CST   ACK_ack: -1
2021-12-21 22:48:27:898 CST   ACK_ack: 3001
2021-12-21 22:48:27:920 CST   ACK_ack: 3101
2021-12-21 22:48:27:945 CST   ACK_ack: 3201
2021-12-21 22:48:27:960 CST   ACK_ack: 3301
2021-12-21 22:48:27:975 CST   ACK_ack: 3401
2021-12-21 22:48:27:991 CST   ACK_ack: 3501
```

但是，RDT 2.0 有一个缺陷，无法对 ACK 或 NACK 出错进行处理，比如：

```
2021-12-21 22:48:27:715 CST   ACK_ack: 1901
2021-12-21 22:48:27:739 CST   ACK_ack: -1845823979          WRONG
2021-12-21 22:48:27:755 CST   ACK_ack: 2101
```

接收方对 seq=2001 的包回复了一个错误的 ACK.



但发送方在发送 seq=2001 的报文之后，没有收到正确的 ACK，但也没有重传此包。以上问题将在 RDT 2.1 中解决。

RDT 2.1

• 接收方如何产生确认或代表否认的序号：
确认：ack 为接收到的包的 seq
否认：ack 为-1

• 接收方收到重复的包如何处理：
不做处理

• 发送方收到错误/重复确认：
错误：重传
重复：不做处理

由于 RDT 2.0 有一个致命的缺陷：无法处理 ACK 或 NACK 出错，故进行修改：

发送方：

修改 recv 函数，对收到的接收方的回复进行校验：
若未出错，则将收到的回复包的 ack 号加入 ackQueue；

```
if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {  // 检查校验和
    System.out.println("Receive ACK Number:  "+ recvPack.getTcpH().getTh_ack());
    ackQueue.add(recvPack.getTcpH().getTh_ack());  // 插入ack队列
    System.out.println();
```

若出错，则在 ackQueue 中加入-1；

```
} else {
    System.out.println("Receive corrupt ACK: " + recvPack.getTcpH().getTh_ack());
    this.ackQueue.add(-1);  // ACK或NACK出错，在ack队列插入-1
    System.out.println();
}
```

接收方：

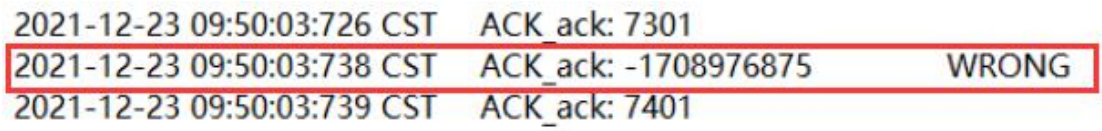
修改 rdt_recv 函数，修改如下：
①设置私有成员变量 last_sequence 用于记录上一个包的 seq

```
int last_sequence = -1; // 用于记录上一次收到包的序号
```

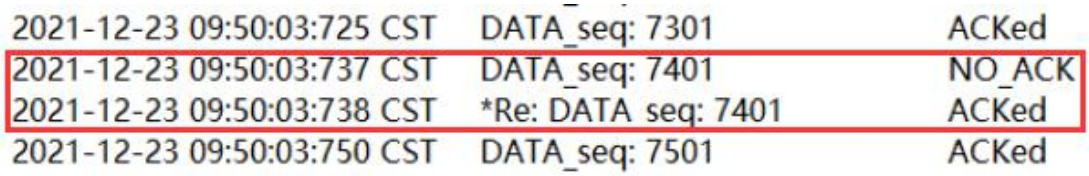
②如果计算校验和正确，则在回复 ack 包之后不是直接 diliver，而是要计算当前收到包的 seq，如果是重复的包（即计算结果和 last_sequence 值相同），则什么也不做；如果是新的包（计算结果和 last_sequence 值不同），则更新 last_sequence 为本次接受到的包的 seq，并将接受到的正确有序的数据插入 data 队列，准备交付。

```
int currentSequence = (recvPack.getTcpH().getTh_seq() - 1) / 100;  // 当前包的seq
if (currentSequence != this.last_sequence) {  // 收到的包有新的seq
    this.last_sequence = currentSequence;  // 更新上一次接受的seq为本次接受到的包的seq

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

实验结果：



通过上图可以看出，接收方在收到第一个 seq 为 7401 的包时，回复了一个出错的 ack(第二行）



发送方没有收到正确的 ACK（第二行），于是重发了 7401 号包（第三行），这个包被正确的收到了。相比于 RDT 2.0，RDT 2.1 具备了检查 ACK 或 NACK 是否出错的能力。

RDT 2.2

• 接收方如何产生确认或代表否认的序号：
确认：ack 为接收到的包的 seq
否认：ack 为上一次确认的包的 seq

• 接收方收到重复的包如何处理：
不做处理

• 发送方收到错误/重复确认：
重传

对于 RDT 2.1，可以仅使用 ACK，从而进行优化。在 RDT 2.1 的代码基础上进行以下修改

发送方：

修改 waitACK 函数，将重发条件改为接收到错误序号的 ACK（即接收到的包的 ack 不是刚刚发送的包的 seq）

```
71          if (currentAck != tcpPack.getTcpH().getTh_seq()){  // 接收到错误序号的ack
```

接收方：

修改 rdt_recv 函数，将检查校验和发现出错时，回复包时添加的 ack= -1 改为 ack=上一次确认的分组的序号。

```
50              tcpH.setTh_ack(last_sequence * 100 + 1); // 设置ack为上一个接收到的包的seq
```

结果：

```
2021-12-23 10:50:48:135 CST   DATA_seq: 10101              ACKed
2021-12-23 10:50:48:152 CST   DATA_seq: 10201              ACKed
2021-12-23 10:50:48:166 CST   DATA_seq: 10301   WRONG      NO_ACK
2021-12-23 10:50:48:171 CST   *Re: DATA_seq: 10301                    ACKed
2021-12-23 10:50:48:198 CST   DATA_seq: 10401              ACKed
2021-12-23 10:50:48:214 CST   DATA_seq: 10501              ACKed
2021-12-23 10:50:48:230 CST   DATA_seq: 10601              ACKed
```

发送方发送一个 seq=10301 的分组，但是没有收到正确的 ACK

```
2021-12-23 10:50:48:122 CST   ACK_ack: 10001
2021-12-23 10:50:48:137 CST   ACK_ack: 10101
2021-12-23 10:50:48:154 CST   ACK_ack: 10201
2021-12-23 10:50:48:169 CST   ACK_ack: 10201
2021-12-23 10:50:48:173 CST   ACK_ack: 10301
2021-12-23 10:50:48:201 CST   ACK_ack: 10401
2021-12-23 10:50:48:217 CST   ACK_ack: 10501
2021-12-23 10:50:48:232 CST   ACK_ack: 10601
```

发送方接收到 seq=10201 的包，检查正确，回复对 seq=10201 的包的确认；
但是 seq=10301 的包，检查错误，回复对上一个正确接收的包的确认，所以回复的是对 seq=10201 的包的确认；

```
2021-12-23 10:50:48:119 CST   DATA_seq: 10001              ACKed
2021-12-23 10:50:48:135 CST   DATA_seq: 10101              ACKed
2021-12-23 10:50:48:152 CST   DATA_seq: 10201              ACKed
2021-12-23 10:50:48:166 CST   DATA_seq: 10301   WRONG      NO_ACK
2021-12-23 10:50:48:171 CST   *Re: DATA_seq: 10301                    ACKed
2021-12-23 10:50:48:198 CST   DATA_seq: 10401              ACKed
2021-12-23 10:50:48:214 CST   DATA_seq: 10501              ACKed
```

发送方收到了重复的 ACK，所以进行重发。

RDT 3.0

• 停止等待如何实现：
通过设置计时器和重传任务实现，先实例化一个 UDT_Timer 对象，在设置一个 UDT_RetransTask 对象，参数为客户端对象 client 和 tcp 分组 tcpPack；将计时器加入重传任务中，设置开始时间为程序运行的 1000ms 的 delay，之后每隔 1000ms 的 period 就进行一次重传

• 超时重传出现位置和前后代码的关系：
出现位置应该是在应用层调用可靠传输函数 rdt_send 中，在生成 tcp 包之后（因为可能要重发分组，不能每次重新打），在调用不可靠发送 udt_send 之前

• 发送方收到重复的确认如何处理：
重传

• 接收方收到重复的包如何处理：
不做处理

在信道出现出错/丢包的情况下，在 RDT 2.2 的基础上进行以下修改：

发送方：

修改 rdt_send 函数，加入计时器并设置重传任务：

```
timer = new UDT_Timer();  // 设置计时器
task = new UDT_RetransTask(client, tcpPack);  // 设置重传任务
// 将计时器加入重传任务中，设置开始时间为1s之后，之后每隔1s执行一次
timer.schedule(task,  delay: 1000,  period: 1000);
```

以上代码使得程序运行的 1000ms 的 delay，之后每隔 1000ms 的 period 就进行一次重传。

修改 waitACK 函数：

```
if (currentAck == tcpPack.getTcpH().getTh_seq()) {  // 接收到正确的ACK
    System.out.println("Clear: "+tcpPack.getTcpH().getTh_seq());
    timer.cancel();  // 终止计时器
    flag = 1; // 切换为等待应用层调用状态
}
// 如果收到错误的ack，什么也不用做，因为计时器会定时自动重传
```

如果收到正确的 ACK，则终止计时器，切换为等待应用层调用状态；
如果收到错误的 ACK，则什么也不做，因为计时器到时间会自动重发；

修改 udt_send 函数：

```
tcpH.setTh_eflag((byte)4);
```

将 eflag 置为 4：出错/丢包

接收方：

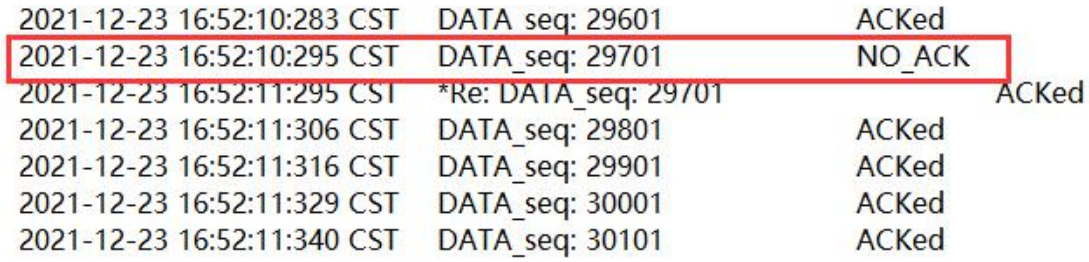
修改 rdt_recv 函数，对于计算校验和出错的包，什么也不做，因为发送方在定时内没有收到 ACK 会自动重传；

修改 reply 函数：

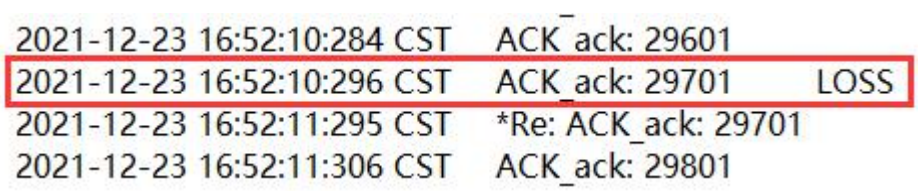
```
tcpH.setTh_eflag((byte)4);
```

置 eflag 为 4

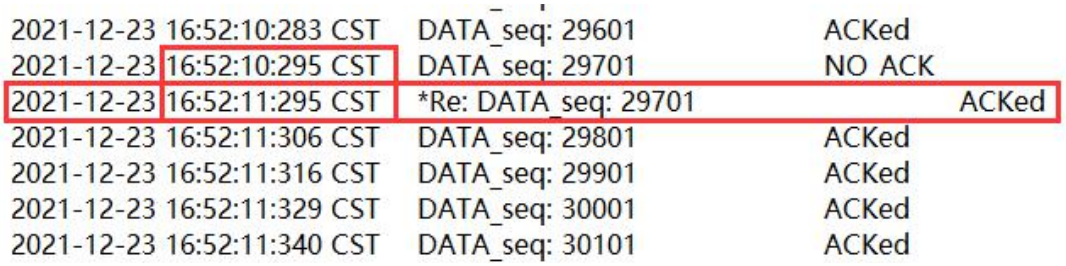
结果：



发送方发送了 seq=29701 的包，但是没有收到 ACK



发现原因是接收方发送的对 seq=29701 的包的 ACK 丢失了



1000ms 时间到了，发送方没有收到 ACK，于是重发了 seq=29701 的包（从上图中也可以看出两次发包的时间间隔是 1000ms）。

RDT 4.0 Go-Back-N

• 发送方或者接收方收到重复的包如何处理：
发送方：不做处理
接收方：对已确认的序号中最大的 ACK

• 用来表达窗口的数据结构是什么：
数组
具体见 SenderSlidingWindow.java

• 窗口满阻塞应用层调用如何实现：

```
// 判断发送窗口是否已满
if (this.window.isFull()) {
    System.out.println();
    System.out.println("Sliding Window is full");
    System.out.println();
    this.flag = 0;
}

//等待ACK报文
while (flag==0);
```

在 TCP_Sender 的 rdt_send 函数中编写以上内容：判断窗口是否已满（调用 SenderSlidingWindow 的 isFull 函数），如果已满，则置 flag 为 0，并且在 flag=0 时无限执行 while 循环，从而实现阻塞应用层调用。

• 仅使用 1 个计时器如何重传窗口内的所有包：
在 SenderSlidingWindow 中，将应用层传输的包放入一个窗口数组中

```
/*向窗口中加入包*/
public void putPacket(TCP_PACKET packet) {
    packets[nextIndex] = packet;  // 在窗口的插入位置放入包
    if (nextIndex == 0) {  // 如果在窗口左沿，则要开启计时器
        timer = new Timer();
        task = new TaskPacketsRetransmit(client, packets);
        timer.schedule(task,  delay: 1000,  period: 1000);
    }

    nextIndex++;  // 更新窗口的插入位置
}
```

编写类 TaskPacketsRetransmit，重写其中的 run 函数，一次性发送所有窗口内所有包

```java
@Override
public void run() {

    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]);
        }
    }
}
```

• 如何使用计时器跟踪窗口左沿:

在 SenderSlidingWindow 中有一个成员变量交 nextIndex，指向应用层下一个传输的包要存储在窗口中的位置。nextIndex 为 0 表示窗口左沿。

• 累积确认在接收方如何实现:

①维护一个期待的 seq 值

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

②修改 rdt_recv 函数：

2.1 如果计算校验和错误：什么也不做，等待发方重传；

2.2 如果计算校验和正确：首先计算当前收到的包的 seq

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

2.2.1 比较当前包的 seq 和期待的 seq，如果相等：

```java
if (expectedSequence == currentSequence) {  // 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），发往发送方
    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
```

设置确认号为收到的 TCP 分组的序号；

生成 ACK 并回复；

准备交付数据；

更新 expectSequence；

2.2.2 比较当前包的 seq 和期待的 seq，如果不相等：

```
} 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），发往发送方
    tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));  // 设置ACK的校验位

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

返回对已确认的包中 seq 最大的包（即第 expectSequence - 1 号包）的确认

③交付数据

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

• 在发送方如何应用累积确认：

修改 recv 函数：

```
public void recv(TCP_PACKET recvPack) {
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {  // 检查校验和
        System.out.println("Receive ACK Number:  "+ recvPack.getTcpH().getTh_ack());

        window.receiveACK( currentSequence: (recvPack.getTcpH().getTh_ack() - 1) / 100);
        if (!window.isFull()) {
            this.flag = 1;
        }
    }
}
```

①首先检查是否出错，如果出错什么也不管，因为即使对该包的 ACK 出错，这时有两种情况：

1. 这个包已经被接收方正确接收了，但是接收方发送的 ACK 在传输途中出错。这时后续的包如果被正确传送并且发送方可以收到 ACK，传输过程可以正常进行
2. 如果这个包本身就没有被正确接收，那么重传机制还会重传此包。

②如果没有出错，则调用 SenderSlidingWindow 的 recvPack 函数，更新滑动窗口

③检查串口是否满，如果没满则置 flag 为 1，表示允许应用层调用。

• 累积确认后的窗口如何移动：

①首先判断确认的包是否在窗口内，如果在：

②

```
for (int i = 0; currentSequence - 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[currentSequence - base + 1 + i];
    packets[currentSequence - base + 1 + i] = null;
}

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

timer.cancel();  // 停止计时器
```

将窗口中位于确认的包之后的包整体移动到窗口左沿；

更新窗口左沿为当前收到的确认 ack + 1，更新 nextIndex

• 计时器是否跟踪了移动后的窗口左沿：

```
if (nextIndex != 0) {  // 窗口中仍有包，则重开计时器
    timer = new Timer();
    task = new TaskPacketsRetransmit(client, packets);
    timer.schedule(task,  delay: 1000,  period: 1000);
}
```

如果 nextIndex 为 0，则表示下一个包放入窗口的位置是窗口左沿；
而如果 nextIndex 更新之后不为 0，表示窗口中还有包，因此要重开计时器；

• 对于窗口满和累积确认说明：

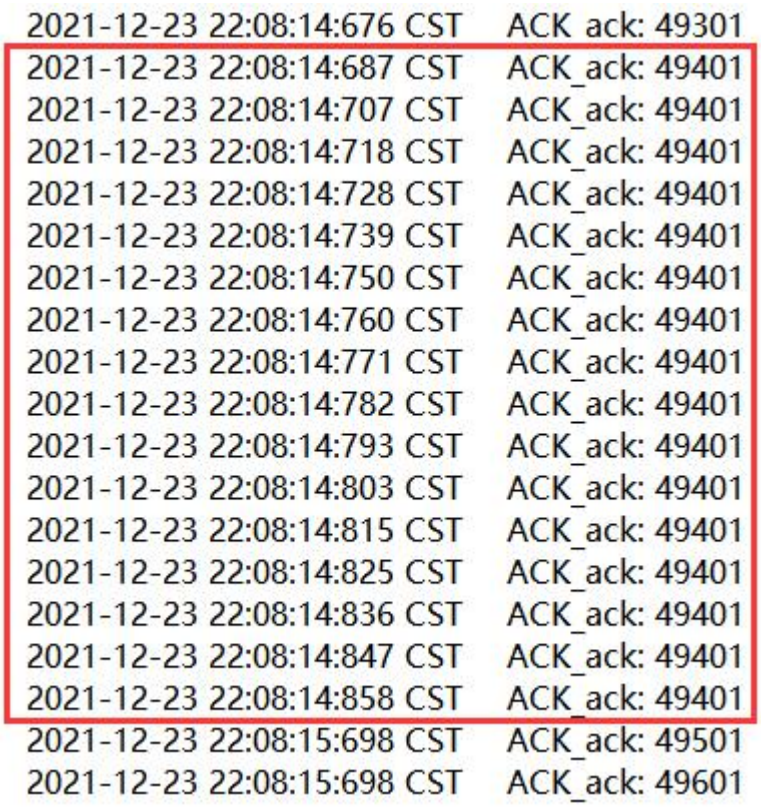
```
/*判断窗口是否已满*/
public boolean isFull() {
    return size <= nextIndex;
}
```

size 表示窗口大小（16）
nextIndex 表示下一个包放入窗口的位置

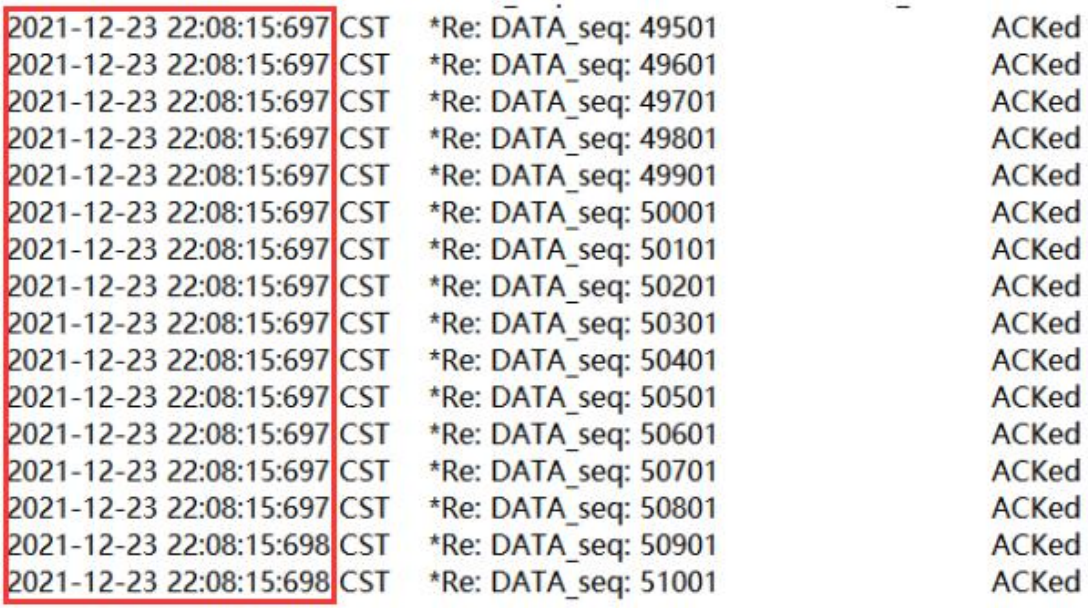
结果：

DATA_seq: 49501	DELAY	NO_ACK
DATA_seq: 49601		NO_ACK
DATA_seq: 49701		NO_ACK
DATA_seq: 49801		NO_ACK
DATA_seq: 49901		NO_ACK
DATA_seq: 50001		NO_ACK
DATA_seq: 50101		NO_ACK
DATA_seq: 50201		NO_ACK
DATA_seq: 50301		NO_ACK
DATA_seq: 50401		NO_ACK
DATA_seq: 50501		NO_ACK
DATA_seq: 50601		NO_ACK
DATA_seq: 50701		NO_ACK
DATA_seq: 50801		NO_ACK
DATA_seq: 50901		NO_ACK
DATA_seq: 51001		NO_ACK

由图可知，发送方发送 seq=49401 的包，这个包被确认了。随后发送一个 seq=49501 的包，delay 了，没有收到确认。
下面观察接收方的情况：



由图可知，接收方连续发送对最大的已确认序号（49401）的确认。



接收方的计时器时间到了之后，同时发送窗口内的所有包

```
2021-12-23 22:08:15:698 CST    ACK_ack: 49501
2021-12-23 22:08:15:698 CST    ACK_ack: 49601
2021-12-23 22:08:15:698 CST    ACK_ack: 49701
2021-12-23 22:08:15:698 CST    ACK_ack: 49801
2021-12-23 22:08:15:699 CST    ACK_ack: 49901
2021-12-23 22:08:15:700 CST    ACK_ack: 50001
2021-12-23 22:08:15:700 CST    ACK_ack: 50101
2021-12-23 22:08:15:701 CST    ACK_ack: 50201
2021-12-23 22:08:15:701 CST    ACK_ack: 50301
2021-12-23 22:08:15:702 CST    ACK_ack: 50401
2021-12-23 22:08:15:702 CST    ACK_ack: 50501
2021-12-23 22:08:15:702 CST    ACK_ack: 50601
2021-12-23 22:08:15:703 CST    ACK_ack: 50701
2021-12-23 22:08:15:703 CST    ACK_ack: 50801
2021-12-23 22:08:15:704 CST    ACK_ack: 50901
2021-12-23 22:08:15:704 CST    ACK_ack: 51001
```

接收方给出确认

RDT 4.0 Selective-Response

• 用来表达窗口的数据结构是什么：

数组

具体见 SenderSlidingWindow.java 和 ReceiverSlidingWindow.java

• 窗口满阻塞应用层调用如何实现：

```
// 判断发送窗口是否已满
if (this.window.isFull()) {
    System.out.println();
    System.out.println("Sliding Window is full");
    System.out.println();
    this.flag = 0;
}

//等待ACK报文
while (flag==0);
```

在 TCP_Sender 的 rdt_send 函数中编写以上内容：判断窗口是否已满（调用 SenderSlidingWindow 的 isFull 函数），如果已满，则置 flag 为 0，并且在 flag=0 时无限执行 while 循环，从而实现阻塞应用层调用。

• 窗口在确认后如何移动：

①对于窗口外的确认不做响应；

②对于窗口内的 ACK，首先将其在窗口对应位置处的计时器停止并删除（如果计时器已经是空表明这是一个重复的 ACK，不用理会）

```
if (timers[currentSequence - base] == null) {  // 表示接收到重复ACK，什么也不做
    return;
}

timers[currentSequence - base].cancel();  // 终止计时器
timers[currentSequence - base] = null;  // 删除计时器
```

③如果接收到的 ACK 位于窗口左沿，则要进行窗口移动：

3.1 计算窗口左沿应该移动到的位置：min noACKed

```
int leftMoveIndex = 0;  // 窗口左沿应该移动到的位置：最小未ACK的分组
while (leftMoveIndex + 1 <= nextIndex && timers[leftMoveIndex] == null) {
    leftMoveIndex ++;
}
```

3.2 窗口内的包移动（相当于窗口左移）

```
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];
}
```

3.3 清空已左移的包原来所在位置处的包和计时器

```
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计时器
    packets[i] = null;
    timers[i] = null;
}
```

3.4 更新窗口左沿值和下一个包的插入位置

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

• 计时器数组如何生成、如何去除：

声明：

```
private UDT_Timer[] timers = new UDT_Timer[size];  // 存储计时器
```

使用：

```
/*向窗口中加入包*/
public void putPacket(TCP_PACKET packet) {
    packets[nextIndex] = packet;  // 在窗口的插入位置放入包
    timers[nextIndex] = new UDT_Timer();  // 为新放入窗口内的包增加计时器
    timers[nextIndex].schedule(new UDT_RetransTask(client, packet), delay: 1000, period: 1000);
    nextIndex++;  // 更新窗口的插入位置
}
```

去除：

见上一个问题

• 收到确认的数据包的清除：

接收方维护一个接收窗口，使用数组缓存正确接收的包

①对于失序分组[rcvbase-N, rcvbase-1]，回复 ACK

②对于正确序号到达分组，加入缓存窗口中

```
packets[currentSequence - base] = packet;
```

③如果这个位置正好是窗口左沿，则要进行窗口滑动

④交付数据

接收方窗口滑动：

①首先计算窗口左沿应移动到的位置：最小未收到数据包处

```
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 ++;
}
```

②将已接收到的分组加入交付队列

```
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());
}
```

③剩余位置的包左移

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

④将左移的包原来位置处置空

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

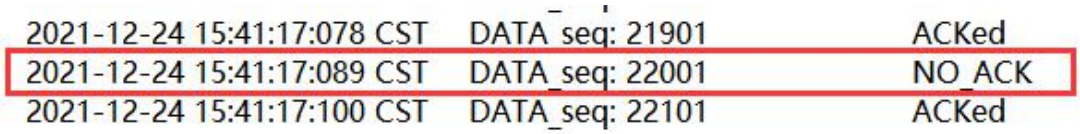
⑤移动窗口左沿

```
base += leftMoveIndex;  // 移动窗口左沿
```

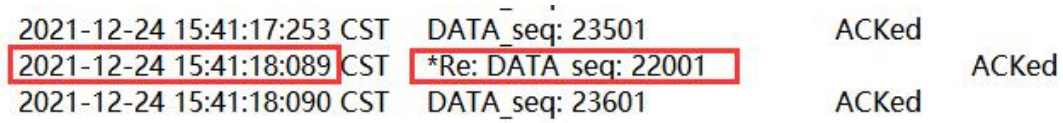
结果：

```
2021-12-24 15:41:17:079 CST    ACK_ack: 21901
2021-12-24 15:41:17:090 CST    ACK_ack: 22001        DELAY
2021-12-24 15:41:17:101 CST    ACK_ack: 22101
```

接收方延迟了对 seq=22001 号包的 ACK



发送方没有收到对 seq=22001 号包的 ACK



1000ms 之后，发送方重发了这个分组，被正确 ACK

TCP Tahoe

• 由于 Tahoe 版本和 Reno 版本相差不大，故省略对 Tahoe 版本的说明。
• 若要验证 TCP Tahoe 版本的功能，只需将 SenderSlidingWindow.java 文件中的 RENO_FLAG 变量置为 0 即可。

TCP Reno

• 对于 Reno 版本的说明，先从以下几个代码文件以及对应解决的问题开始。
• 默认接收方窗口无限大

①TCP_Sender
1.1 rdt_send 函数

```
// 判断发送窗口是否已满
if (window.isFull()) {
    System.out.println();
    System.out.println("Sliding Window is full");
    System.out.println();
    flag = 0;
}

//等待ACK报文
while (flag==0);

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

// 发送 TCP 数据报
udt_send(tcpPack);
```

阻塞应用层调用的方法与 GBN 和 SR 相同，若窗口满则置 flag 为 0。如果 flag==0 发送端则一直执行 while 循环，无法发送分组；只有当 flag==1 时才可发送分组。putPacket 为发送方滑动窗口的处理函数。

1.2 recv 函数

```
@Override
//接收到ACK报文：
public void recv(TCP_PACKET recvPack) {
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {  // 检查校验和
        System.out.println("Receive ACK Number:  "+ recvPack.getTcpH().getTh_ack());

        window.receiveACK( currentSequence: (recvPack.getTcpH().getTh_ack() - 1) / 100);
        if (!window.isFull()) {
            this.flag = 1;
        }
    }
}
```

当接收方收到一个 ACK 时，先检查它是否出错，若未出错则调用发送窗口的 reciveACK 函数进行进行接收方接收到 ACK 之后的处理，并继续判断窗口是否已满，若窗口未满则置 flag 为 1，表示允许接收应用层调用。

1.3 udt_send 函数

```
tcpH.setTh_eflag((byte)7);
//发送数据报
client.send(stcpPack);
```

置错误标志 eflag 为 7，发送分组

②TCP_Receiver

接收方维护一个期待收到的 seq，与 GBN 类似，一下用 expectedSequence 称呼此值

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

我使用 HashTable 缓存失序分组

```
private Hashtable<Integer, TCP_PACKET> storagePackets = new Hashtable<>(); // 用于缓存失序分组
```

2.1 reply 函数

```
tcpH.setTh_eflag((byte)7);
```

2.1 rdt_recv 函数

如果接收到的分组计算校验和正确且 seq 值与 expectedSequence 相同，则进行如下操作：

```
// 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 ;
```

• 将接受到的数据插入 data 队列准备交付

• 期待 seq+1

```
for (int i = expectedSequence; ; i ++ ) {
    if (storagePackets.containsKey(i)) {
        dataQueue.add(storagePackets.get(i).getTcpS().getData());
        expectedSequence += 1;
        storagePackets.remove(i);
    } else {
        break;
    }
}
```

• 从 expectedSequence 开始检测，序号每轮循环+1，判断缓存的 Hashtable 中是否有数据，如果有则将其添加至交付队列并将其从缓存队列删除；当然，每添加一个缓存的分组，expectedSequence 都要+1

```
//交付数据（每20组数据交付一次）
if(dataQueue.size() >= 20)
    deliver_data();
if(currentSequence >= 899 && currentSequence <= 999) {
    deliver_data();
}
```

•交付数据（此处我有一个疑惑，就是怎么交付最后几个分组。我用了一个不是很好的方法。之后会在困难问题中详述此问题及可能解决方法）

如果收到失序的分组，则进行缓存（此处失序指的是 seq != expectedSequence）

```
else {  // 收到失序的包，返回已确认的最大序号分组的确认

  // 缓存失序分组
  if (!storagePackets.containsKey(currentSequence) && currentSequence > expectedSequence) {
      storagePackets.put(currentSequence, recvPack);
  }
```

• 如果缓存的 Hashtable 中没有当前 seq，并且要缓存的分组的 seq 大于 expectSequence（因为比 expectedSequence 小的分组肯定已经交付，故无需缓存）,则将其加入缓存中。

```
//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序号最大的TCP分组的seq
ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());  // 新建一个TCP分组（ACK），发往发送方
tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));  // 设置ACK的校验位

reply(ackPack);  // 回复ACK报文段
```

最后回复 ACK，ack 值为 expectedSequence - 1（即最大已 ACK 的 seq），无需进行区分。

③RetransmitTask

该类用于编写超时重传的逻辑，在 run 函数中编写以下内容：

```
// 清空拥塞避免计数器
window.setCongestionAvoidanceCount(0);


// 立刻重传分组(窗口左沿)
if ( window.getPackets().containsKey(window.getLastACKSequence() + 1)) {
    client.send(window.getPackets().get(window.getLastACKSequence() + 1));
}
```

- 首先要清空拥塞避免的计时器（这是我踩过的一个坑，之后会在问题困哪部分详细说明）
- 立刻重传超时的分组，超时的分组的 seq 就是上一次 ACK 的分组的 seq + 1（因为是累计确认）

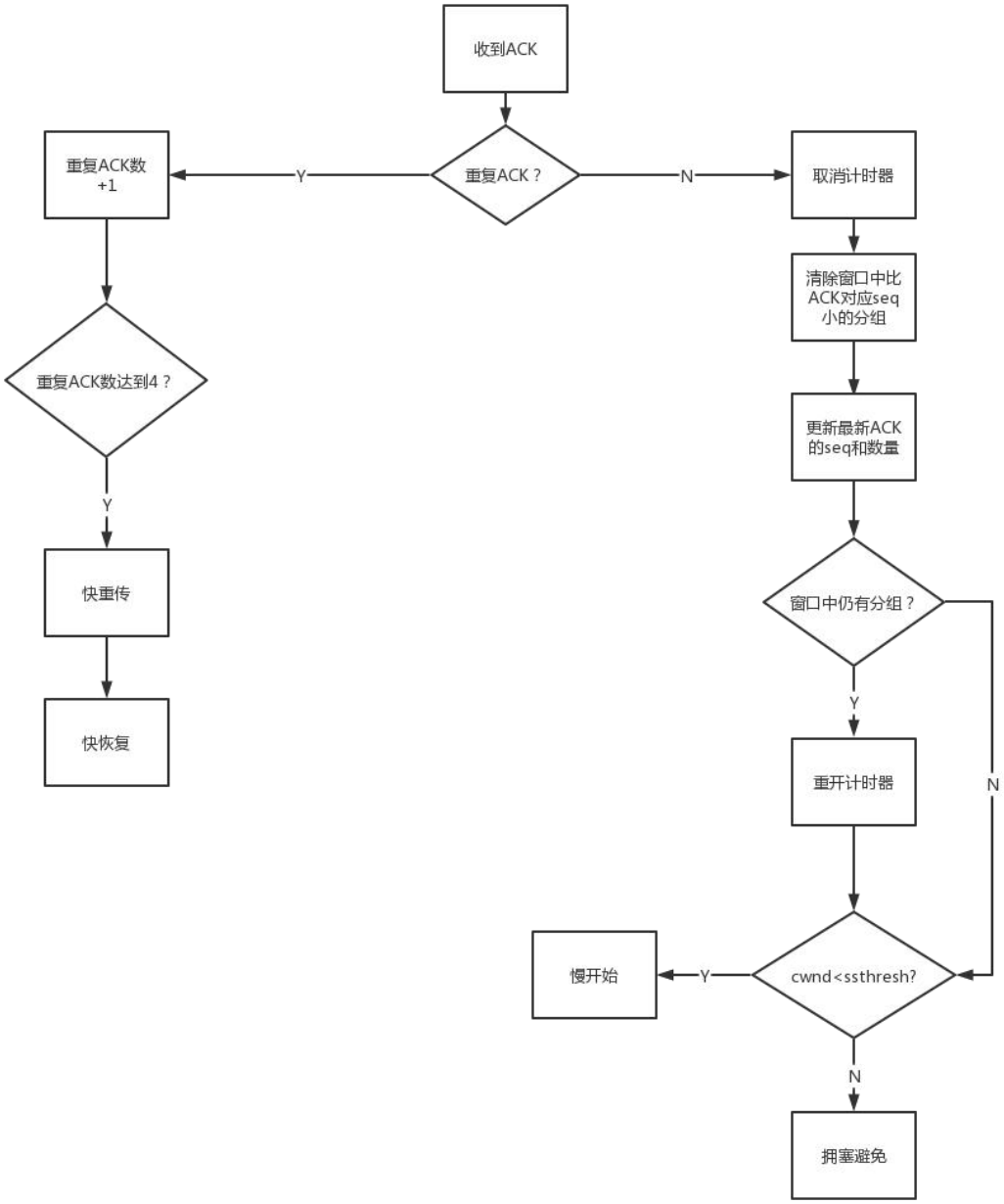
超时重传窗口的变化：

```
// 超时重传
System.out.println("***** Timeout Retransmit *****");
if (window.getCwnd() / 2 < 2) {
    System.out.println("ssthresh: " + window.getSsthresh() + " ---> 2");
    window.setSsthresh(2);  // ssthresh 不得小于2
} else {
    System.out.println("ssthresh: " + window.getSsthresh() + " ---> " + window.getCwnd() / 2);
    window.setSsthresh(window.getCwnd() / 2);  // 慢开始门限变为 cwnd 的一半
}
System.out.println("cwnd: " + window.getCwnd() + " ---> 1");
window.setCwnd(1);  // cwnd 置为1

window.appendChange(window.getLastACKSequence());
```

ssthresh 变为 cwnd 的一半（不能小于 2），cwnd 变为 1.　上图中最后一行是我用来输出 cwnd 和 ssthresh 变化情况的函数，与 TCP 的运行逻辑无关，只是用来向控制台输出信息。

④SenderSlidingWindow

该类用于编写发送方的滑动窗口，实现慢开始、拥塞避免、快重传、快恢复
代码流程如下：



详细解释如下：

1.首先定义 cwnd 和 ssthresh 的初始默认值分别为 1 和 16：

```
public int cwnd = 1;  // 拥塞窗口，初始值为1
private volatile int ssthresh = 16;  // 慢开始门限，初始值为16
```

2.定义 CongestionAvoidanceCount 变量，用于记录进入拥塞避免状态时累计收到的 ACK 数。注意在窗口大小发生变化从而退出或者重新进入拥塞避免状态时，该值要清零（这是我做的过程中踩到的坑，会在之后的遇到困难部分详细说明）

```
private int CongestionAvoidanceCount = 0;  // 记录进入拥塞避免状态时收到的ACK数
```

3.lastACKSequence 用于记录上一次收到的 ACK 的 seq，lastACKSequenceCount 用于记录重复 ACK 的数量。这两个变量主要用于快重传条件的判断。

```
private int lastACKSequence = -1;  // 上一次收到的ACK的包的seq
private int lastACKSequenceCount = 0;  // 收到重复ACK的次数
```

4.表示窗口的数据结构为 Hashtable，key 为整型（即分组序号 seq），value 为 TCP 分组类型。由此可以构建由 seq 到分组的映射）。以下说明中用 packets 表示发送方滑动窗口。

```
private Hashtable<Integer, TCP_PACKET> packets = new Hashtable<>();  // 存储发方窗口的包
```

5.使用一个计时器

```
private Timer timer;  // 计时器
```

6.判断发送窗口是否满的函数：

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

• 根据课本中的定义，发放窗口的大小与拥塞窗口大小一致，故当 cwnd == packets 中分组数量时，窗口满。

7.putPacket 函数：当 TCP_Sender 发送分组时调用，将分组放入滑动窗口中
其功能逻辑如下：

```
int currentSequence = ((packet.getTcpH().getTh_seq() - 1) / 100);
```

• 计算当前加入窗口分组的 ACK

```
if (packets.isEmpty()) {
    // 窗口左沿则开始计时器
    timer = new Timer();
    timer.schedule(new RetransmitTask(client,  window: this),  delay: 1000,  period: 1000);
}
```

• 加入的分组若在窗口左沿（即 packet 为空的情况），则开启计时器

```
packets.put(currentSequence, packet);
```

• 在 packets 中加入该分组。

8.receiveACK 函数
该函数在发送方每次收到正确 ACK 时调用，执行的操作有如下：

首先判断是否是重复 ACK，若重复 ACK，是否是三重复 ACK

```
// 收到重复ACK
if (currentSequence == lastACKSequence) {
    lastACKSequenceCount++;
    if (lastACKSequenceCount == 4) {  // 三个重复ACK，执行快重传
```

• 这里我遇到一个坑是 lastACKSequenceCount 要判是否==4 而不是==3 。 因为虽然是三重复 ACK 但是上一个正确接收的 ACK 也被计算在内，所以实际上是四个连续一样的 ACK 触发快重传。

```
if (packets.containsKey(currentSequence + 1)) {
    // 快重传
    System.out.println("***** Fast Retransmit *****");
    TCP_PACKET packet = packets.get(currentSequence + 1);
    client.send(packet);
    timer.cancel();
    timer = new Timer();
    timer.schedule(new RetransmitTask(client, window: this), delay: 1000, period: 1000);
}
```

• 快重传：重传的分组为上一个 ACK 对应的 seq 值 + 1
• 重启计时器

```
System.out.println("***** Fast Recovery *****");
if (cwnd / 2 < 2) {
    System.out.println("ssthresh: " + ssthresh + " ---> 2");
    ssthresh = 2;  // ssthresh 不得小于2
} else {
    System.out.println("ssthresh: " + ssthresh + " ---> " + cwnd / 2);
    ssthresh = cwnd / 2;  // 慢开始门限变为 cwnd

}
System.out.println("cwnd: " + cwnd + " ---> " + ssthresh);
cwnd = ssthresh;  // cwnd 置为ssthresh
CongestionAvoidanceCount = 0;
```

• 快恢复：ssthresh 变为 cwnd 一半（不能小于 2），cwnd 变为 ssthresh 的值。由于拥塞避免的条件是 cwnd>=ssthresh，故进入拥塞避免阶段。

• 清空拥塞避免计数器

※对于重复 ACK 但每到三次的情况，无需终止计时器。（这一点是我在某次 log 中遇到的，会在之后遇到问题困难部分详细说明）

对于新的 ACK：

```
// 清空计时器
timer.cancel();
```

• 首先清空计时器

```
// 收到新的ACK
for (int i = lastACKSequence + 1; i <= currentSequence; i++) {  // 清除前面的包
    packets.remove(i);
}
```

• 由于是累计确认，故可对发送窗口中当前 ACK 对应 seq 之前的分组进行删除。

```
lastACKSequence = currentSequence;  // 重置lastACKSequence为当前收到ACK的包的seq
lastACKSequenceCount = 1;  // 重置lastACKCount为1
```

• 更新最新 ACK 对应 seq 的值为当点 ACK 对应 seq
• 更新累计 ACK 的数量为 1

```
// 如果窗口中仍有分组，则重开计时器
if (!packets.isEmpty()) {
    timer = new Timer();
    timer.schedule(new RetransmitTask(client, window: this), delay: 1000, period: 1000);
}
```

• 如果窗口中仍有分组，则重开计时器。

```
if (cwnd < ssthresh) {
    // 慢启动
    System.out.println("***** Slow Start *****");
    System.out.println("cwnd: " + cwnd + " ---> " + (cwnd + 1));
    cwnd ++;  // 每收到一个ACK， cwnd加一
    appendChange(currentSequence);
```

• 更新窗口大小 cwnd 和慢开始门限 ssthresh：如果 cwnd<ssthresh，表明此时处于慢开始阶段，每收到一个 ACK, cwnd 就+1. appendChange 函数是我用来输出 cwnd 和 ssthresh 变化情况的函数，与 TCP 的运行逻辑无关，只是用来向控制台输出信息。

```
else {
    // 拥塞避免
    CongestionAvoidanceCount ++;
    System.out.println("***** Congestion Avoidance *****");
    System.out.println("cwnd: " + cwnd + " NO. " + CongestionAvoidanceCount);
    if (CongestionAvoidanceCount == cwnd) {  // 收到ACK数量达到 cwnd 大小时
        CongestionAvoidanceCount = 0;  // 重置计数器
        System.out.println("cwnd: " + cwnd + " ---> " + (cwnd + 1));
        cwnd ++;  // cwnd 加一
        appendChange(currentSequence);
    }
```

• 更新窗口大小 cwnd 和慢开始门限 ssthresh：如果 cwnd>=ssthresh，表明此时处于拥塞避免阶段，拥塞避免计数器+1。当拥塞避免计数器的值达到了窗口大小 cwnd 时，变可使 cwnd+1，此时也要清空计数器。（此处我有一个疑惑，就是有看到拥塞避免阶段受到 ACK 时 cwnd += 1/cwnd 的算法，但我不是很理解，写了一下发现也不行。故采用计数器技术的方法记录拥塞避免阶段累计收到的 ACK 数）

结果：

说明：

1. 代码逻辑的说明已在上面部分给出，在说明代码如何处理 Log 文件的情况时仅给出对应代码和简单的解释；

2. 对于出错情况下的解释较为详细；在丢失和延迟的情况下，需要解释的代码也大致相同，在区别之处也予以了解释。

3. TCP 拥塞控制的内容在 Log 文件中不方便看出来，逻辑流程图和代码解释也在 P23 页给出。我将程序运行过程中的每个分组传输结束之后所处的阶段以及是否发生快重传、快恢复输出到了控制台，可以在 out2 文件夹 console.txt 中查看。

4. 为了更加直观的显示 cwnd 和 ssthresh 的变化，我将 cwnd 发生变化时的 cwnd,ssthresh,和对应收到的 ack 均输出，在 P36 可以看到。

1.出错：

首先考虑发送的分组出错的情况

在发送方第一次收到对 6501 的 ACK 时，发送窗口类中维护的变量 lastACKSequence 更新为 6501，lastACKSequenceCount 为 1

```
lastACKSequence = currentSequence;  // 重置lastACKSequence为当前收到ACK的包的seq
lastACKSequenceCount = 1;  // 重置lastACKCount为1
```

```
2021-12-30 15:49:19:581 CST DATA_seq: 6501                ACKed
2021-12-30 15:49:19:597 CST DATA_seq: 6601    WRONG       NO_ACK
2021-12-30 15:49:19:612 CST DATA_seq: 6701                NO_ACK
2021-12-30 15:49:19:628 CST DATA_seq: 6801                ACKed
2021-12-30 15:49:19:629 CST *Re: DATA_seq: 6601                    NO_ACK
2021-12-30 15:49:19:643 CST DATA_seq: 6901                ACKed
```

发送方发送了一个错误的分组，序号是 6601.

```
2021-12-30 15:49:19:566 CST ACK_ack: 6401
2021-12-30 15:49:19:581 CST ACK_ack: 6501
2021-12-30 15:49:19:598 CST ACK_ack: 6501
2021-12-30 15:49:19:613 CST ACK_ack: 6501
2021-12-30 15:49:19:629 CST ACK_ack: 6501
2021-12-30 15:49:19:630 CST ACK_ack: 6801
2021-12-30 15:49:19:644 CST ACK_ack: 6901
2021-12-30 15:49:19:659 CST ACK_ack: 7001
```

接收方回复的是已确认的序号最大值（即 6501）对应的 ACK。

对应接收方代码如下：错误的分组，回复累计确认的最大 ACK，即期望 Seq 的前一个分组

```
@Override
//接收到数据报：检查校验和，设置回复的ACK报文段
public void rdt_recv(TCP_PACKET recvPack) {
    //检查校验码，生成ACK
    if(CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {...} else {

    }

    //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序号最大的TCP分组的seq
    ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());  // 新建一个TCP分组（ACK），发往发送方
    tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));  // 设置ACK的校验位

    reply(ackPack);  // 回复ACK报文段


    System.out.println();
```

发送方接收到了这个 ACK 之后，lastACKSequenceCount 变为 2

```
// 收到重复ACK
if (currentSequence == lastACKSequence) {
    lastACKSequenceCount++;
```

```
2021-12-30 15:49:19:581 CST DATA_seq: 6501                ACKed
2021-12-30 15:49:19:597 CST DATA_seq: 6601    WRONG       NO_ACK
2021-12-30 15:49:19:612 CST DATA_seq: 6701                NO_ACK
2021-12-30 15:49:19:628 CST DATA_seq: 6801                ACKed
2021-12-30 15:49:19:629 CST *Re: DATA_seq: 6601                     NO_ACK
2021-12-30 15:49:19:643 CST DATA_seq: 6901                ACKed
```

发送方发送的 seq=6701 同样没有收到 ACK

```
2021-12-30 15:49:19:566 CST ACK_ack: 6401
2021-12-30 15:49:19:581 CST ACK_ack: 6501
2021-12-30 15:49:19:598 CST ACK_ack: 6501
2021-12-30 15:49:19:613 CST ACK_ack: 6501
2021-12-30 15:49:19:629 CST ACK_ack: 6501
2021-12-30 15:49:19:630 CST ACK_ack: 6801
2021-12-30 15:49:19:644 CST ACK_ack: 6901
2021-12-30 15:49:19:659 CST ACK_ack: 7001
```

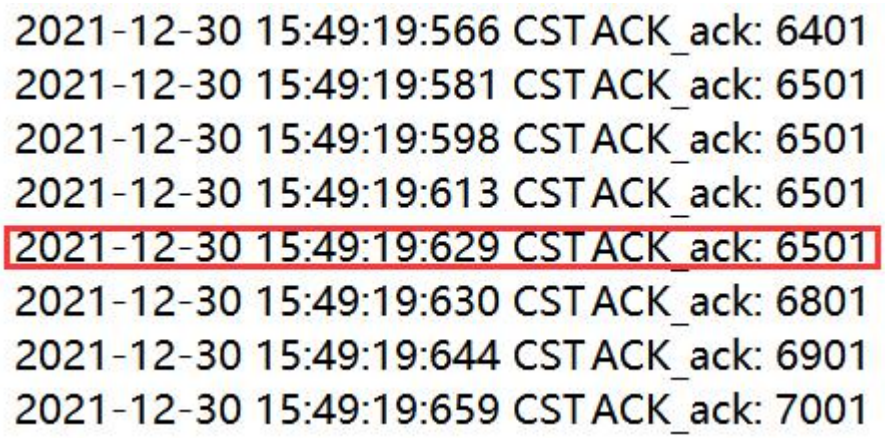
因为接收方回复的同样是最大确认 6501。接收方同样缓存这个分组 6701。

缓存失序分组的代码如下：如果缓存中没有这个分组并且这个分组的序号比当前期待的序号要大（小的话不用缓存了，因为比期望分组序号小的肯定已经正确接收了），则将其加入缓存中。

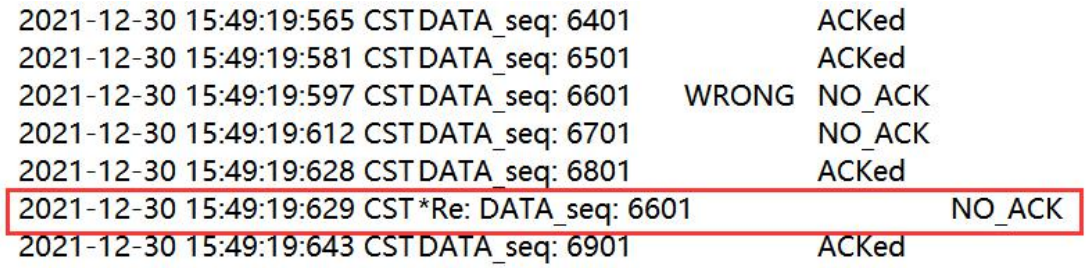
```
else {  // 收到失序的包，返回已确认的最大序号分组的确认

    // 缓存失序分组
    if (!storagePackets.containsKey(currentSequence) && currentSequence > expectedSequence) {
        storagePackets.put(currentSequence, recvPack);
    }
```

```
2021-12-30 15:49:19:565 CST DATA_seq: 6401                ACKed
2021-12-30 15:49:19:581 CST DATA_seq: 6501                ACKed
2021-12-30 15:49:19:597 CST DATA_seq: 6601    WRONG       NO_ACK
2021-12-30 15:49:19:612 CST DATA_seq: 6701                NO_ACK
2021-12-30 15:49:19:628 CST DATA_seq: 6801                ACKed
2021-12-30 15:49:19:629 CST *Re: DATA_seq: 6601                     NO_ACK
2021-12-30 15:49:19:643 CST DATA_seq: 6901                ACKed
```

发送方发送的 seq=6801 的分组得到了 ACK。但是要注意，这里 6801 的得到的 ACK 并不是接收方收到 6801 分组（即上图红框所示）后回复的 ACK，因为观察接收方：



对这个分组，接收方同样是回复了最大确认 6501，同样缓存这个分组 6801. 之所以 6801 显示 ACKed 是因为：



发送方连续收到三个重复 ACK(6501)之后，执行快重传，立刻重传了分组 6601。

检测三重复 ACK 部分代码如下：在发送方接受到对 6801 分组的 ACK 时，lastACKSequenceCount 已经加到了 4，这时候就要立刻重传分组。重传的分组是重复 ACK 对应的分组的下一个（因为累计确认）。此时还要取消并重开计时器。

```
/*接收到ACK*/
public void receiveACK(int currentSequence) {


    // 收到重复ACK
    if (currentSequence == lastACKSequence) {
        lastACKSequenceCount++;
        if (lastACKSequenceCount == 4) {  // 三个重复ACK，执行快重传
            if (packets.containsKey(currentSequence + 1)) {
                // 快重传
                System.out.println("***** Fast Retransmit *****");
                TCP_PACKET packet = packets.get(currentSequence + 1);
                client.send(packet);
                timer.cancel();
                timer = new Timer();
                timer.schedule(new RetransmitTask(client, window: this), delay: 1000, period: 1000);
            }
```

2021-12-30 15:49:19:566 CST ACK_ack: 6401
2021-12-30 15:49:19:581 CST ACK_ack: 6501
2021-12-30 15:49:19:598 CST ACK_ack: 6501
2021-12-30 15:49:19:613 CST ACK_ack: 6501
2021-12-30 15:49:19:629 CST ACK_ack: 6501
2021-12-30 15:49:19:630 CST ACK_ack: 6801
2021-12-30 15:49:19:644 CST ACK_ack: 6901
2021-12-30 15:49:19:659 CST ACK_ack: 7001

接收方收到了这个重传的分组，同时因为接收方已经缓存了失序的分组 6701 和 6801，故此时接收方已接收的最大 seq 为 6801，返回的 ACK 对应的 seq 也是 6801.这也就是为什么 6801 有 ACKed，而*Re:6601 没有 ACKed 的原因。

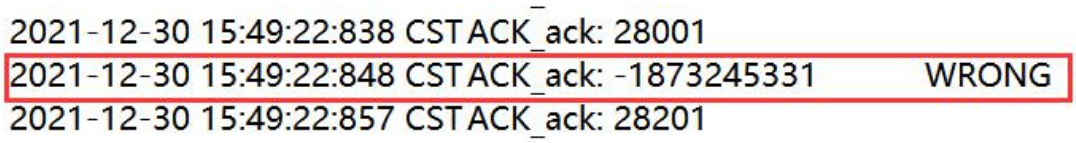
接收方在收到重发的 6601 分组之后进行如下的处理：计算校验和，正确；判断是否是期望接收的分组：是。于是：将这个 6601 分组加入交付队列，期望 Seq+1，并且要从缓存中将所有存在的按序且比 6601 大的序号的分组也加入到交付队列中来，并且将他们从缓存中清除，同时每从缓存中交付一个分组，期望 Seq 都要+1

```
if(CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {  // 计算并比对校验和，如果相等:
    int currentSequence = (recvPack.getTcpH().getTh_seq() - 1) / 100;  // 当前包的seq
    if (expectedSequence == currentSequence) {  // 当前收到的包就是期望的包

        // 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 ;

        // 处理缓存数据

        for (int i = expectedSequence; ; i ++ ) {
            if (storagePackets.containsKey(i)) {
                dataQueue.add(storagePackets.get(i).getTcpS().getData());
                expectedSequence += 1;
                storagePackets.remove(i);
            } else {
                break;
            }
        }
```

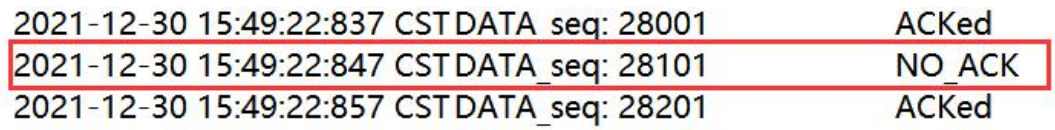
下面的图将更加清楚的说明对应关系：

上图中的红色箭头表示真实的接收方的分组和回复 ACK 之间的对应关系，蓝绿色的线是虚拟的，表示：接收方虽然收到重发的包 6601，并对其回复确认是 6801，但是这个 ACK 被接收方理解为是对发送的分组 6801 的 ACK 了。实际上对 6801 的 ACK 是 6501.

再考虑 ACK 出错的情况



接收方回复 28101 的 ACK 出错了



代码如下：

```
@Override
//接收到ACK报文：
public void recv(TCP_PACKET recvPack) {
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {...}
}
```

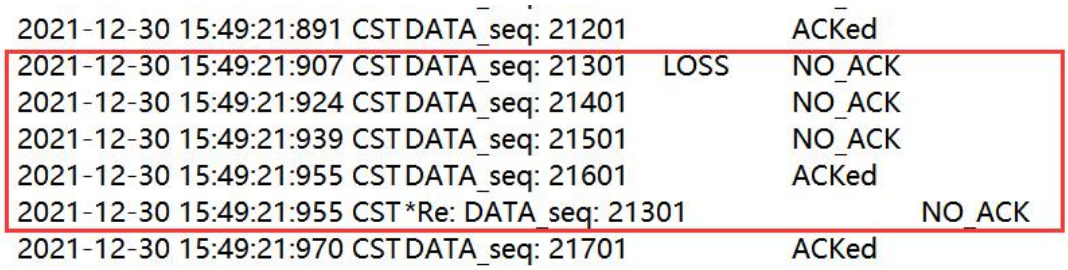
发送方没有收到正确 ACK。但是发送方也并没有执行其他操作。因为发送方虽然不知道错误是出在发送数据的分组还是回复的 ACK，但他已经接收到了对更高序号的分组 ACK，则可认为这个分组已经被接收端正确接收了（因为使用累计确认的缘故）。

总结 reno 对于出错的处理：
- 接收方对于错误的分组回复已确认最大序号分组 ACK
- 发送方对于错误 ACK 不进行处理

2 丢失：

首先考虑发送方发送的分组丢失的情况

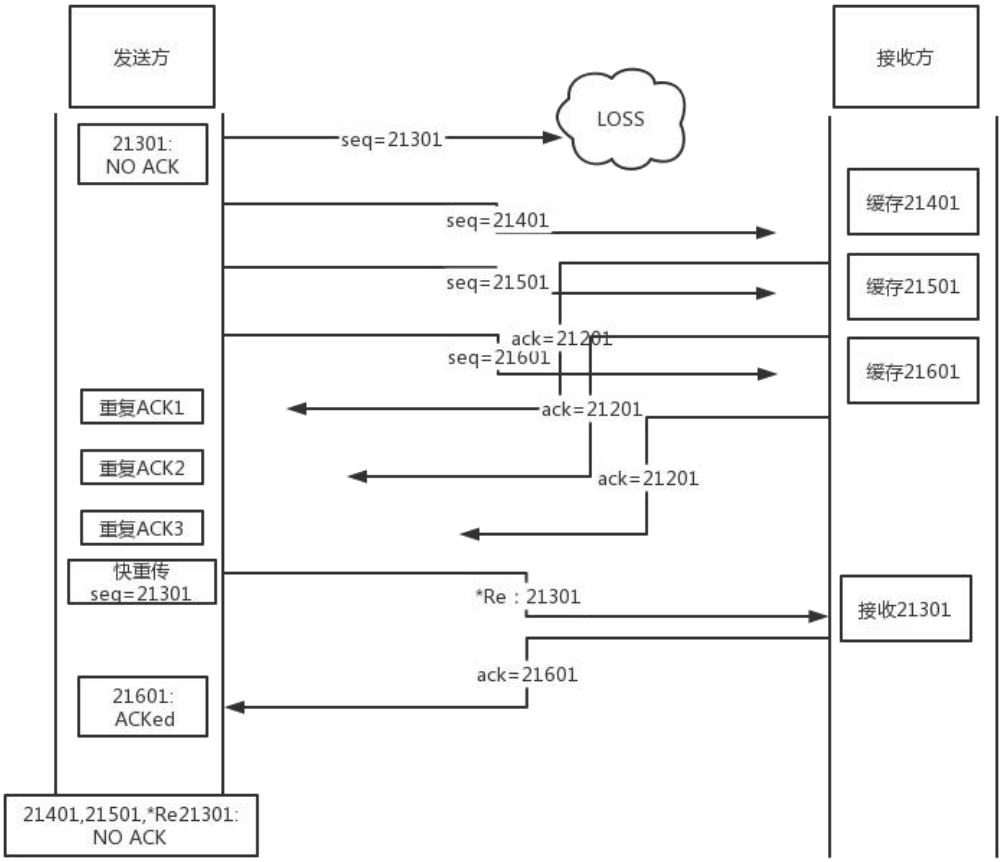
发送方



观察发送方，与出错情况下对比，可以看到是多了一行的，这是因为 LOSS 的分组没有 ACK，所以没有算进重复 ACK 里面，后续还要再发三个分组，得到三个 ACK。

接收方

```
2021-12-30 15:49:21:892 CST ACK_ack: 21201
2021-12-30 15:49:21:925 CST ACK_ack: 21201
2021-12-30 15:49:21:940 CST ACK_ack: 21201
2021-12-30 15:49:21:955 CST ACK_ack: 21201
2021-12-30 15:49:21:956 CST ACK_ack: 21601
2021-12-30 15:49:21:971 CST ACK_ack: 21701
```

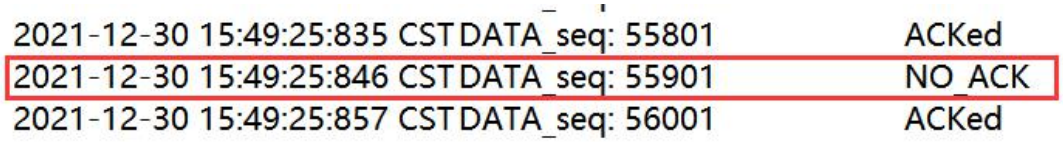


p.s.从上图应该可以很清楚的看出 TCP 处理分组丢失的逻辑了。上图中的箭头都应该是直到对应双方的边界的，但是那样我调不出一个比较清楚的图片。只要理解上图中 seq=21401,21501,21601 的分组和三个 ack=21201 的分组都已到达对应方即可。

再考虑接收方发送 ACK 丢失的情况

```
2021-12-30 15:49:25:836 CST ACK_ack: 55801
2021-12-30 15:49:25:848 CST ACK_ack: 55901      LOSS
2021-12-30 15:49:25:857 CST ACK_ack: 56001
```

接收方发送的 ACK=55901 丢失了

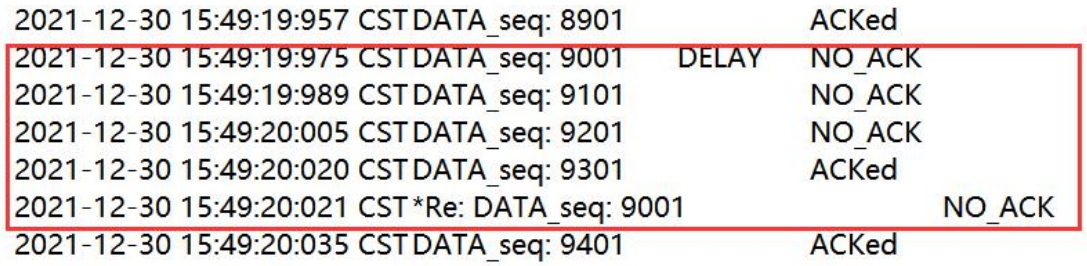


接收方没有收到对应的正确 ACK。但他已经接收到了对更高序号的分组 ACK，则可认为这个分组已经被接收端正确接收了（因为使用累计确认的缘故）。

3. 延迟：

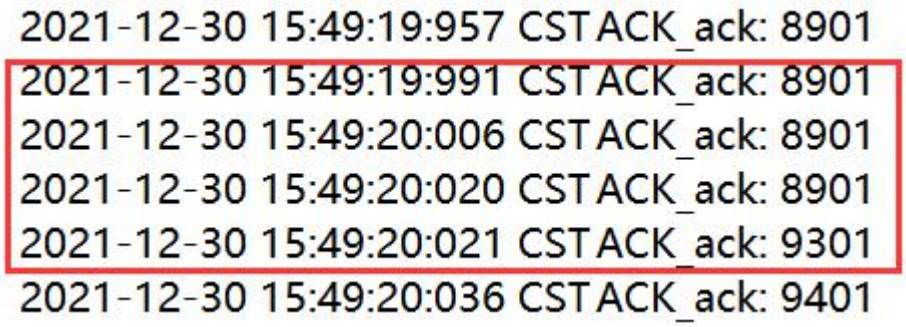
首先考虑发送方发送的分组延迟的情况

发送方

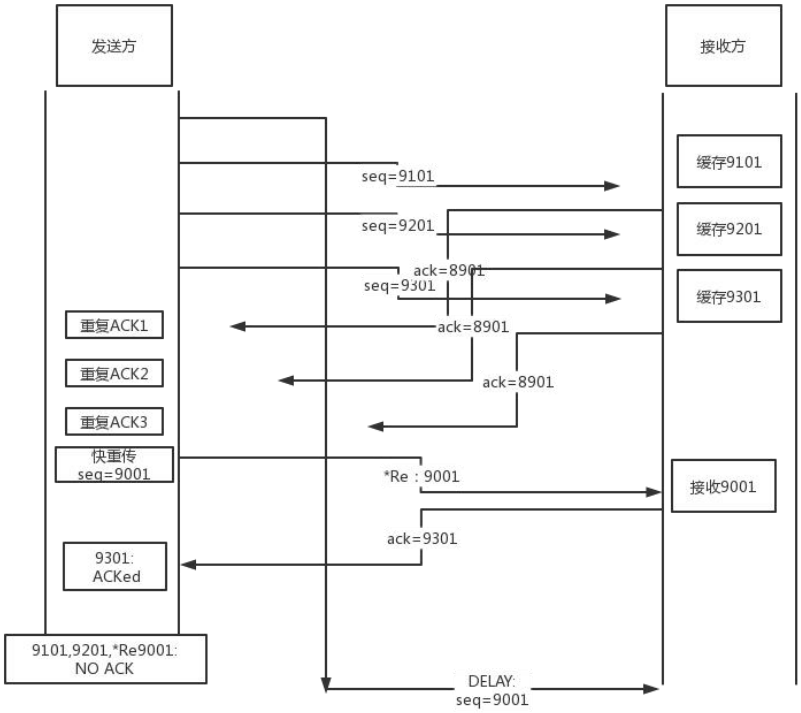


观察发送方，与出错情况下对比，可以看到是多了一行的，这是因为 DELAY 的分组没有及时得到 ACK，所以没有算进重复 ACK 里面，后续还要再发三个分组，得到三个 ACK。但是，与 LOSS 不同，DELAY 的分组最终会到达接收方，见后续分析。

接收方



解释过程图：



虽然 9001 DELAY 了，但是接收方也应该收到并回复一个 ACK。那么接收方回复的这个 ACK 应该是什么呢？继续看 Log 文件发现文件最底部有：

```
2021-12-30 15:49:30:543 CST ACK_ack: 99701
2021-12-30 15:49:30:553 CST ACK_ack: 99801
2021-12-30 15:49:30:564 CST ACK_ack: 99901
2021-12-30 15:49:50:451 CST ACK_ack: 99901
2021-12-30 15:49:59:980 CST ACK_ack: 99901
2021-12-30 15:50:02:438 CST ACK_ack: 99901
```

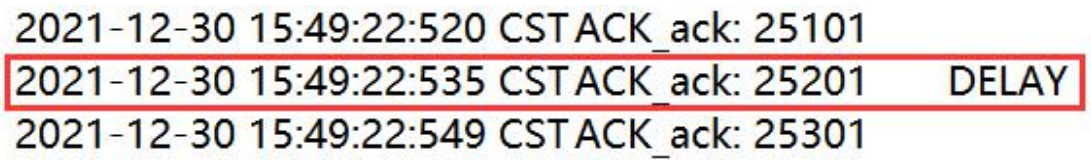
可以看到最后多了三个对 99901 的 ACK。

CLIENT HOST	TOTAL	SUC_RATIO		NORMAL	WRONG	LOSS	DELAY
192.168.50.248:9001		1008	96.63%	1000	4	1	3

再看 Log 文件的开头，发送方一共 DELAY 了三个分组的发送。故猜想 DELAY 的三个分组在 TCP 连接过程的最后传到了接收端。由于接收端目前已经接受了全部的分组，故回复对已确认最大分组序号（99901）的 ACK。

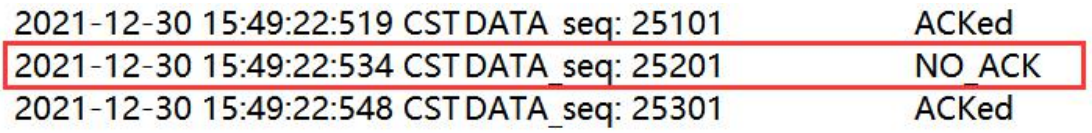
再考虑接收方发送 ACK 延迟的情况

接收方



可以见到发送的对 25201 的 ACK 延迟了

发送方



可以看到发送方发送的 25201 没有收到正确的 ACK。推测是因为在 ACK 到达发送方之前 TCP 连接已经释放。

窗口变化情况：

说明 1：.由于当时没有保存下实验报告分析的 Log 文件对应的那次程序运行的控制台输出，所以我又重新跑了一次。我把 Log 文件和 recData 文件放到 out1 文件夹中，把控制台输出到的 console.txt 和表示窗口大小变化的 cwnd&ssthresh_change.txt 文件放到 out2 文件夹中。注意！out1 和 out2 中的内容是两次不同运行的结果.
说明 2：控制台输出的最后有三个 1000 维的数组，分别表示 cwnd 发生变化时，cwnd，ssthresh 和 cuurentSequence（当前收到 ACK 对应的 seq）（此值已做 seq - 1 / 100 的处理）的值。最后有很多 0 是因为开的是 1000 的静态数组用于存储输出的变化值。

例如：

在 cwnd&ssthresh_change.txt 文件中可以看到如下内容：
cwnd:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 12, 13, 14, 15, 16, 17, 18, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 9, 10, 11, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 7, 8, 9, 10, 11, 12, 13, 14, 15, 7, 8, 9, 10, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 8, 9, 10, 5, 6, 7, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
标黄部分可以看到实现了快恢复（ssthresh 变为 cwnd 的一半，cwnd 又变为 ssthresh 的值，故相当于 cwnd 减半）

ssthresh：
16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 12, 12, 12, 12, 12, 12, 12, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 8, 8, 8, 5, 5, 5, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,

currentSequence:
0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 30, 47, 65, 84, 105, 126, 148, 171, 195, 195, 210,

223, 237, 252, 268, 287, 288, 300, 310, 321, 333, 346, 360, 375, 391, 408, 422, 434, 444, 445, 453, 459, 466, 474, 483, 493, 504, 516, 529, 543, 548, 558, 566, 575, 585, 597, 609, 622, 636, 641, 652, 660, 669, 676, 683, 689, 696, 704, 713, 723, 734, 746, 759, 773, 788, 801, 812, 821, 824, 831, 838, 841, 847, 851, 856, 862, 869, 877, 886, 896, 907, 919, 932, 946, 961, 977, 994,
(第一个 0 可以忽略，是为了便于程序编写添加的）

在 console.txt 中可以看到
慢开始：

```
***** Slow Start *****
cwnd: 2 ---> 3
```

每收到一个正确 ACK 窗口+1，每个轮次窗口翻倍。

拥塞避免：

```
***** Congestion Avoidance *****
cwnd: 16 NO. 2
```

使用计数器，当正确 ACK 数到达 cwnd 时窗口大小+1.

```
***** Congestion Avoidance *****
cwnd: 5 NO. 5
cwnd: 5 ---> 6
```

快重传&快恢复：

```
***** Fast Retransmit *****
***** Fast Recovery *****
ssthresh: 16 ---> 12
cwnd: 25 ---> 12
```

ssthresh 变为 cwnd 一半，cwnd 变为 ssthresh 值

超时重传：
其中，由于 3 重复 ACK 快重传的存在，多次运行程序，结果都是，一旦发包出现问题，3 重复 ACK 都比超时计时器响应更快。因此为了验证超时重传的正确性，先将三重复 ACK 部分的代码注释掉后观察控制台输出情况。

结果如下：

```
***** Timeout Retransmit *****
ssthresh: 2 ---> 5
cwnd: 10 ---> 1
```

ssthresh 变为 cwnd 一半，cwnd 减为 1

控制台输出见 out2 文件夹中的 timeout.txt

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）

RDT 1.0
RDT 2.0
RDT 2.1
RDT 2.2
RDT 3.0
Go-Back-N
Selective-Response
TCP Tahoe
TCP Reno
版本均已完成，程序代码和 Log 文件，recvData 文件见项目目录下的 edition 文件夹

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难度由易到难，最初的几个版本比较简单，非常适合学生入手去做，上手较快；最后几个版本也比较有挑战性，但在之前版本练习的基础上可以有较好的处理；
②容易激发学生的探索兴趣，我就是这样，在做完前几版本之后，对自己有了一定的（莫名的）自信，激发我去进行后面几个版本的编写。虽然在写后面几个版本时有好几次卡主很久（尤其是 GBN 和 SR 窗口数组的移动问题），但是在这种兴趣的激励下我还是坚持完成下来。
③从 WRONG->LOSS->DELAY 的顺序来看，各个版本侧重于解决不同的问题，迭代开发比较适合针对 TCP 传输过程中某一类问题进行重点分析。
④符合历史发展的认知。比如从 Tahoe 到 Reno 的转变，既是实验中迭代开发的顺序，也是历史中出现并应用的顺序；采用如此顺序的迭代开发更能带我们贴近真实。
⑤可以较好实现代码的重用。比如 Reno 可以直接照搬 GBN 的累计确认，只要再加上一些其他处理即可（比如缓存失序分组）。

缺点：
①由于代码重用，如果低版本出现错误，之后可能会一直出错。比如如果纠错的 RDT 2.0 的校验码函数都写错了，之后就算做到了 Reno 也是无济于事的。
②虽然各个版本都有侧重的问题，在完成高版本的同时却容易忘记低版本处理的问题，比如 Reno 是如何处理 WRONG 和 LOSS 的？这个我在开发的过程中也进行了思考和处理。

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)

p.s 该模块主要针对 reno 版本中的问题进行说明

问题①正确使用一个计数器的相关问题

问题说明：

首先说一下我原本 Reno 中计时器的使用是参照 GBN 的思想（因为 GBN 也是一个计时器），就是发送方每收到一个正确 ACK 就停止计时器，进行窗口移动之后看窗口内若有剩余分组则重开计时器。

这是我在一次 Log 时发现的问题。当时的 Log 我没有留存，但我清楚的记得 Log 是这样的：

.....

seq=1 的分组---->ACKed

seq=101 的分组---->DELAY　　NO_ACK

seq=201 的分组---->NO_ACK

.....

然后,控制台便输出 Sliding Window is full，这是在应用层调用时进行的 isFull 函数的输出，输出这句话表示发送窗口已满。然后程序就停止了，后续的分组也没有发出。

问题分析：

分析问题的原因，我认为是我 timer 的使用错误。我们都知道当收到三重复 ACK 要立刻重传并重开计时器，那么当收到重复 ACK 但重复 ACK 数量不足 3 时该怎么办？

我回看自己写的代码，发现我是每次收到正确 ACK（不论序号，只要校验和计算不出错）就停止计时器，重开计时器只有在三重复 ACK 和清楚窗口内分组后窗口内仍有分组的情况。那么，当收到重复 ACK 但重复 ACK 数量不足 3 时便不会重开计时器，这也是出现上述问题的原因。

seq=1 得到 ACK 之后 cwnd 变为 2，然后窗口内放入了 seq=101 和 201 两个包，但由于 101 被 DELAY 了，所以 101 和 201 都没有 ACK，但由于还没到三重复 ACK（因为只有两个重复 ACK），所以计时器也不会重开，没有重发 seq=101 的包，所以这两个包就一直占据的 cwnd 的两个位置，不会被清除掉，故 cwnd 一直是满的，应用层无法继续发送分组，整个程序就死了。

问题解决：

只要将停止计时器的位置放到收到正确且不重复 ACK 的位置就 OK 了，重复 ACK 是不需要停计时器的。

困难②：交付数据时最后几个分组的交付问题

问题说明：

在编写程序时，按照最开始几个版本是交付队列 dataQueue 中每 20 个分组交付一次，代码如下：

```
if(dataQueue.size() >= 20)
    deliver_data();
```

但在 reno 版本中，会出现最后的分组无法交付的情况，猜想原因是前面的交付中会出现这样的情况：在某次收到模 20 等于 0 的分组执行到 deliver_data 时，接收方就又收到一个分组并将其插入到交付队列中。在本程序中共有 1000 个分组，模 20 正好为 0.但若出现上述情况，则会使得接收方在 dataQueue 中插入 seq=99901 的分组之后，dataQueue 中的数量不足 20 个，从而无法交付最后的数据。

困难解决：

• 我使用了一种方法，就是判断接收的分组位置比较靠后时，就直接交付分组，代码如下：

```
if(currentSequence >= 899 && currentSequence <= 999) {
    deliver_data();
}
```

“比较靠后”指的是从 seq=89901 到 seq=99901 的分组。这种方法有一个先决条件，就是要知道此次 TCP 连接中发送发要发送的最大分组序号。这种方法不够好。

• 我认为更好地方法是在接收方设置一个计时任务，每隔一定时间执行一次 deliver_data；在 TCP 链接释放时停止并移除该计时器。

困难③拥塞避免阶段的实现方式问题

问题说明：

首先说明为什么要用拥塞避免计数器。我在学习的过程中看到有说拥塞避免阶段的 cwnd 扩张算法是这样的：没收到一个正确不重复 ACK 就使 cwnd += 1 / cwnd . 这是我不是很能理解的地方。就比如我现在 ssthresh 是 16，然后我慢开始 cwnd 加加加一直加到 16 了，好现在进入拥塞避免，16+= 1/16 , 那下一岂不是就要+ 1 / (16 + 1/16) ？那这样下去岂不是一直到不了 17？只有加 16 个 1/16 才到 17，但如果按照这种算法的话每次加完值都会比 1/16 小，加 16 次怎么也到不 17。

问题解决：

我猜想可能是计算机内部浮点数表示的问题？于是我照着算法写了，但是事实证明没有效果，行不通。于是我就采用了拥塞避免计时器。先说明一下拥塞避免计数器的用处，就是在拥塞避免阶段每收到正确新 ACK 就使计数器+1，当计数器值到了 cwnd 时就使 cwnd+1. 这样一来 16 个 ACK 之后，cwnd 就变成了 17 了。

困难④拥塞避免计数器的清除问题

问题说明：

在使用拥塞避免计数器时我也遇到了一个这样的问题：就是当进入拥塞避免之后，cwnd 和 ssthresh 的变化就会在 1 和 2 之间，就陷入无限反复的循环之中。

问题解决：

经过 debug 之后发现是因为没有清空拥塞避免计数器。当程序退出拥塞避免阶段时，应将拥塞避免计数器清空，否则下次进入拥塞避免阶段时，计数器的初始值就不一定是 0 了。现在看来这是一个比较低级的问题，但是当时属实困惑了许久，所以就在此记录。

5.对于实验系统提出问题或建议(1 分)

①关于 Log 文件的问题：比如在下面这张图中

• seq=21601 的分组收到的 ACK 实际上是 ack=21201，但是由于接收方对*Re:21301 的分组 ACK 了 ack=21601,所以 seq=21601 的这行也显示了 ACKed。

• 虽然这在实际当中没有问题，因为发送方完全可以认为自己发送的 seq=21601 分组收到了 ACK，但在实验过程中不是很便于同学理解。

建议：可以在实验之前说明一下这个 Log 文件的解读方式，或者告诉学生要去读控制台输出。

②发送数据包太快了，数量太多了，而且是一次性全部发完。

建议：可以设置成连续发送一段时间之后停一会儿，再继续接着发送。

③DELAY 出现的比较分散。当网络拥塞时可能在某一个区间的序号内的分组都 DELAY 了。

建议:可以设置连续或间隔较短便出现 DELAY。