

计算机网络大作业报告

学号：20090022058 姓名：朱甲文 专业：机械设计制造及其自动化 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）
说明：目标版本号及得分对照
5.1 (Reno): 17
5.0 (Tahoe): 15
4.0 SR/GB/TCP: 12
3.0: 9
2.2: 6
2.0: 3
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）
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)
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)
5. 对于实验系统提出问题或建议(1 分)

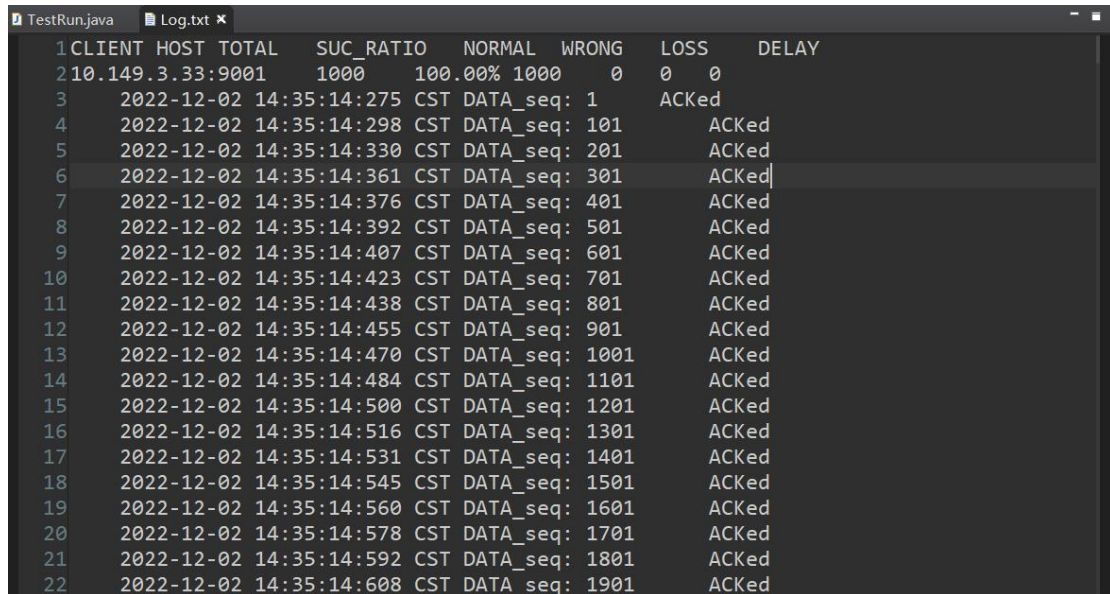
目录

一、结合代码和 LOG 文件分析针对每个项目举例说明解决效果	2
1、RDT1.0	2
2、RDT2.0	2
3、RDT2.1	4
4、RDT2.2	5
5、RDT3.0	6
6、RDT4.0 Secletive Response	8
7、TCP Tahoe&Reno	11
二、未完全完成的项目，完成中遇到的关键困难，以及可能的解决方式	19
三、说明在实验过程中采用迭代开发的优点或问题	19
四、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	19
五、对于实验系统提出问题或建议	20

一、结合代码和 LOG 文件分析针对每个项目举例说明解决效果

1、RDT1.0

RDT1.0 是在可靠信道上进行可靠的数据传输。可以看到，数据在可靠信道上传输的成功率是 100%。



1	CLIENT	HOST	TOTAL	SUC_RATIO	NORMAL	WRONG	LOSS	DELAY
2	10.149.3.33	9001	1000	100.00%	1000	0	0	0
3	2022-12-02	14:35:14:275	CST	DATA_seq: 1			ACKed	
4	2022-12-02	14:35:14:298	CST	DATA_seq: 101			ACKed	
5	2022-12-02	14:35:14:330	CST	DATA_seq: 201			ACKed	
6	2022-12-02	14:35:14:361	CST	DATA_seq: 301			ACKed	
7	2022-12-02	14:35:14:376	CST	DATA_seq: 401			ACKed	
8	2022-12-02	14:35:14:392	CST	DATA_seq: 501			ACKed	
9	2022-12-02	14:35:14:407	CST	DATA_seq: 601			ACKed	
10	2022-12-02	14:35:14:423	CST	DATA_seq: 701			ACKed	
11	2022-12-02	14:35:14:438	CST	DATA_seq: 801			ACKed	
12	2022-12-02	14:35:14:455	CST	DATA_seq: 901			ACKed	
13	2022-12-02	14:35:14:470	CST	DATA_seq: 1001			ACKed	
14	2022-12-02	14:35:14:484	CST	DATA_seq: 1101			ACKed	
15	2022-12-02	14:35:14:500	CST	DATA_seq: 1201			ACKed	
16	2022-12-02	14:35:14:516	CST	DATA_seq: 1301			ACKed	
17	2022-12-02	14:35:14:531	CST	DATA_seq: 1401			ACKed	
18	2022-12-02	14:35:14:545	CST	DATA_seq: 1501			ACKed	
19	2022-12-02	14:35:14:560	CST	DATA_seq: 1601			ACKed	
20	2022-12-02	14:35:14:578	CST	DATA_seq: 1701			ACKed	
21	2022-12-02	14:35:14:592	CST	DATA_seq: 1801			ACKed	
22	2022-12-02	14:35:14:608	CST	DATA_seq: 1901			ACKed	

2、RDT2.0

RDT2.0 假设底层信道传输过程中，个别数据包的某些字节可能发生位错，所以我们要利用校验和算法检查数据包的正确性。所以我们要先完成 CheckSum 类中的 ComputeChkSum()方法，我们使用 CRC32 来计算校验码。

```
8 public class CheckSum {
9
10     /*计算TCP报文段校验和：只需校验TCP首部中的seq、ack和sum，以及TCP数据字段*/
11     public static short computeChkSum(TCP_PACKET tcpPack) {
12         int checksum = 0;
13         CRC32 crc32 = new CRC32();
14         // 计算seq、ack和sum部分(sum初始化为0不用计算);
15         crc32.update(tcpPack.getTcpH().getTh_seq());
16         crc32.update(tcpPack.getTcpH().getTh_ack());
17         // TCP数据部分
18         for(int i = 0; i < tcpPack.getTcpS().getData().length; i++) {
19             crc32.update(tcpPack.getTcpS().getData()[i]);
20         }
21         checksum = (int) crc32.getValue();
22         return (short) checksum;
23     }
24 }
25 }
```

向该函数传入 TCP 包，拿出头部并获取其中的 seq 和 ack 字段，再和数据字段一同计算校验和，通过 getValue()方法得到 checksum 并返回。

发送方：首先将 TCP_Sender 中的 udt_eflag 置为 1，也就是“只出错”。

```

44• @Override
45 //不可靠发送：将打包好的TCP数据报通过不可靠传输信道发送；仅需修改错误标志
46 public void udt_send(TCP_PACKET stcpPack) {
47     //设置错误控制标志, eflag==1, “只出错”
48     tcpH.setTh_eflag((byte)1);
49     //System.out.println("to send: "+stcpPack.getTcpH().getTh_seq());
50     //发送数据报
51     client.send(stcpPack);
52 }

```

在 waitACK 状态,检查接收方发来的确认号队列,判定它是 ACK 还是 NACK。如果是 ACK,则切换状态等待应用层调用;如果是 NACK,则继续等待 ACK 并且重新发送。

```

54• @Override
55 //需要修改
56 public void waitACK() {
57     //循环检查ackQueue
58     //循环检查确认号队列中是否有新收到的ACK
59     if(!ackQueue.isEmpty()){
60         int currentAck=ackQueue.poll();
61         // System.out.println("CurrentAck: "+currentAck);
62         if (currentAck == -1) {
63             System.out.println("Retransmit: " + this.tcpPack.getTcpH().getTh_ack(
64             udt_send(this.tcpPack);
65             this.flag = 0;
66         } else {
67             System.out.println("Clear: " + currentAck);
68             this.flag = 1;
69         }
70     }
71 }
72

```

接收方：首先将 TCP_Receiver 中的 reply()中的 eflag 置为 1。

```

87• @Override
88 //回复ACK报文段
89 public void reply(TCP_PACKET replyPack) {
90     //设置错误控制标志, eflag==1, “只出错”
91     tcpH.setTh_eflag((byte)1); //eFlag=0, 信道无错误
92
93     //发送数据报
94     client.send(replyPack);
95 }
96
97 }

```

接受方接受到一个包时,首先将这个包送入 ComputeChkSum()中计算校验和,并与所收到的包的头部中的校验和进行比较,如果相等,则证明 TCP 分组正确,没有出错,此时发送 ACK 给发送方;如果不相等,则表明 TCP 分组出错,此时发送 NACK 给发送方。

Log 分析

可以看出,本次传输共发生了 12 次位错误。

	CLIENT	HOST	TOTAL	SUC_RATIO	NORMAL	WRONG	LOSS	DELAY
2	192.168.3.123	9001	1012	98.02%	1000	12	0	0
3	2022-12-13	14:03:27:417	CST	DATA_seq: 1		ACKed		
4	2022-12-13	14:03:27:459	CST	DATA_seq: 101		ACKed		
5	2022-12-13	14:03:27:493	CST	DATA_seq: 201		ACKed		

如下图，发送方发送了一个 DATA_seq 为 8901 的包并出现了 wrong，我们接着查看接收方的 log，发现接收方回复了一个 ACK_ack 为-1 的包，此时发送方收到了这个 ack 为-1 的包，于是进行了重传。

90	2022-12-13 14:03:29:192	CST DATA_seq: 8701	ACKed
91	2022-12-13 14:03:29:207	CST DATA_seq: 8801	ACKed
92	2022-12-13 14:03:29:223	CST DATA_seq: 8901	WRONG NO_ACK
93	2022-12-13 14:03:29:225	CST *Re: DATA_seq: 8901	ACKed
94	2022-12-13 14:03:29:241	CST DATA_seq: 9001	ACKed
95	2022-12-13 14:03:29:256	CST DATA_seq: 9101	ACKed
96	2022-12-13 14:03:29:272	CST DATA_seq: 9201	ACKed
97	2022-12-13 14:03:29:287	CST DATA_seq: 9301	ACKed
98	2022-12-13 14:03:29:301	CST DATA_seq: 9401	ACKed
99	2022-12-13 14:03:29:316	CST DATA_seq: 9501	ACKed

1103	2022-12-13 14:03:29:179	CST ACK_ack: 8601	
1104	2022-12-13 14:03:29:193	CST ACK_ack: 8701	
1105	2022-12-13 14:03:29:209	CST ACK_ack: 8801	
1106	2022-12-13 14:03:29:224	CST ACK_ack: -1	
1107	2022-12-13 14:03:29:226	CST ACK_ack: 8901	
1108	2022-12-13 14:03:29:242	CST ACK_ack: 9001	
1109	2022-12-13 14:03:29:257	CST ACK_ack: 9101	
1110	2022-12-13 14:03:29:273	CST ACK_ack: 9201	
1111	2022-12-13 14:03:29:288	CST ACK_ack: 9301	
1112	2022-12-13 14:03:29:303	CST ACK_ack: 9401	

但是，RDT2.0 无法处理出错的 ACK/NACK，如下图所示。发送方发送了一个 DATA_seq 为 11801 的包，而接收方回复了一个错误的 ACK，可以看出发送方在收到接收方错误的 ACK 后并没有重传这个包。

117	2022-12-13 14:03:29:592	CST DATA_seq: 11301	ACKed
118	2022-12-13 14:03:29:608	CST DATA_seq: 11401	ACKed
119	2022-12-13 14:03:29:626	CST DATA_seq: 11501	ACKed
120	2022-12-13 14:03:29:642	CST DATA_seq: 11601	ACKed
121	2022-12-13 14:03:29:670	CST DATA_seq: 11701	ACKed
122	2022-12-13 14:03:29:686	CST DATA_seq: 11801	NO_ACK
123	2022-12-13 14:03:29:701	CST DATA_seq: 11901	ACKed
124	2022-12-13 14:03:29:716	CST DATA_seq: 12001	ACKed
125	2022-12-13 14:03:29:733	CST DATA_seq: 12101	ACKed
126	2022-12-13 14:03:29:746	CST DATA_seq: 12201	ACKed

1133	2022-12-13 14:03:29:627	CST ACK_ack: 11501	
1134	2022-12-13 14:03:29:644	CST ACK_ack: 11601	
1135	2022-12-13 14:03:29:671	CST ACK_ack: 11701	
1136	2022-12-13 14:03:29:687	CST ACK_ack: -1406942697	WRONG
1137	2022-12-13 14:03:29:703	CST ACK_ack: 11901	
1138	2022-12-13 14:03:29:717	CST ACK_ack: 12001	
1139	2022-12-13 14:03:29:734	CST ACK_ack: 12101	
1140	2022-12-13 14:03:29:748	CST ACK_ack: 12201	

3、RDT2.1

在 RDT2.1 中，我们对 RDT2.0 的缺陷进行处理，即无法处理 ACK/NACK 出错的情况。

发送方：修改 recv 函数，将收到的回复包的 ack 加入 ackQueue，如果出错，则在 ackQueue 中加上-1。


```

73  @Override
74  //接收到ACK报文: 检查校验和, 将确认号插入ack队列;NACK的确认号为-1; 不需要修改
75  public void recv(TCP_PACKET recvPack) {
76      if(CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
77          System.out.println("Receive ACK Number: " + recvPack.getTcpH().getTh_ack());
78          ackQueue.add(recvPack.getTcpH().getTh_ack()); //未出错, 将收到回复包的ack加入队列
79          System.out.println();
80      } else {
81          System.out.println("Receive corrupt ACK: " + recvPack.getTcpH().getTh_ack());
82          this.ackQueue.add(-1); //出错, 在ack队列加上-1
83          System.out.println();
84      }
85  }

```

接收方: 修改 rdt_recv 函数, 在计算校验和正确后, 计算当前收到的包的 seq, 接收方需要记录上次接收的包的 seq 值, 若与本次接收的相同, 则不能将它插入 data 队列。

```

38
39      if(recvPack.getTcpH().getTh_seq() != sequence){
40          //将接收到的正确有序的数据插入data队列, 准备交付
41          dataQueue.add(recvPack.getTcps().getData());
42          sequence=recvPack.getTcpH().getTh_seq();
43          //sequence++;
44      }
45
46      }else{
47          System.out.println("Recieve Computed: " + CheckSum.computeChkSum(recvPack));
48          System.out.println("Recieved Packet" + recvPack.getTcpH().getTh_sum());
49          System.out.println("Problem: Packet Number: " + recvPack.getTcpH().getTh_seq() + " + Inr
50          tcpH.setTh_ack(-1);
51          ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());
52          tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));
53          //回复ACK报文段
54          reply(ackPack);
55      }

```

Log 分析

如下图可以看出, 在传输 DATA_seq 为 101 的包时, 接收方回复了一个错误的 ACK, 发送方收到这个错误的 ACK 之后进行了重传, 这个重传的包被接收方收到后, 接收方回复了正确的 ACK, 说明 RDT2.1 具备了检查 ACK/NACK 出错的能力, 并能够在出错时对这个包进行重传。

	CLIENT	HOST	TOTAL	SUC_RATIO	NORMAL	WRONG	LOSS	DELAY
1	192.168.3.123	9001	1016	98.43%	1006	10	0	0
3	2022-12-13	17:55:42:195	CST DATA seq: 1				ACKed	
4	2022-12-13	17:55:42:233	CST DATA_seq: 101				NO_ACK	
5	2022-12-13	17:55:42:236	CST *Re: DATA_seq: 101				ACKed	
6	2022-12-13	17:55:42:267	CST DATA_seq: 201				ACKed	
7	2022-12-13	17:55:42:296	CST DATA_seq: 301				ACKed	

	CLIENT	HOST	TOTAL	SUC_RATIO	NORMAL	WRONG	LOSS	DELAY
1020	192.168.3.123	9002	1016	99.31%	1009	7	0	0
1021	2022-12-13	17:55:42:204	CST ACK ack: 1					
1022	2022-12-13	17:55:42:235	CST ACK_ack: -1351667078				WRONG	
1023	2022-12-13	17:55:42:238	CST ACK_ack: 101					
1024	2022-12-13	17:55:42:269	CST ACK_ack: 201					
1025	2022-12-13	17:55:42:298	CST ACK_ack: 301					

4、RDT2.2

RDT2.2 和 RDT2.1 的功能相同, 但是仅使用 ACK。接收方正确接收一个包后, 发送 ACK, 在 ACK 包中, 接收方必须通过信号指明是对哪个数据包的确认。

重复的 ACK 包对发送方来说，和收到 NACK 的效果一样。

发送端：修改 waitACK 函数，使其接收到错误的 ACK 后进行重发。

```
56 public void waitACK() {
57     //循环检查ackQueue
58     //循环检查确认号对列中是否有新收到的ACK
59     if(!ackQueue.isEmpty()){
60         int currentAck=ackQueue.poll();
61         // System.out.println("CurrentAck: "+currentAck);
62         if (currentAck != tcpPack.getTcpH().getTh_seq()) {
63             System.out.println("Retransmit: " + this.tcpPack.getTcpH().getTh_ack());
64             udt_send(this.tcpPack);
65             this.flag = 0;
```

接收端：修改 rdt_recv 函数，记录上次接收的包的 seq 值,若与本次接收的相同,则不能将它插入 data 队列，发送方就会重发数据包。

```
39         if(recvPack.getTcpH().getTh_seq()!=sequence){
40             //将接收到的正确有序的数据插入data队列，准备交付
41             dataQueue.add(recvPack.getTcpS().getData());
42             sequence=recvPack.getTcpH().getTh_seq();
43             //sequence++;
44         }
45     }else{
46         System.out.println("Recieve Computed: "+Checksum.computeChkSum(recvPack));
47         System.out.println("Recieved Packet"+recvPack.getTcpH().getTh_sum());
48         System.out.println("Problem: Packet Number: "+recvPack.getTcpH().getTh_seq()+" + Inr
49         tcpH.setTh_ack(-1);
50         ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());
51         tcpH.setTh_sum(Checksum.computeChkSum(ackPack));
52         //回复ACK报文段
53         reply(ackPack);
54     }
```

LOG 分析

如下图，发送方发送了一个 DATA_seq 为 12301 的包，但是发生了 wrong 并且是 NO_ACK，也就是没有收到正确的 ACK，而接收方对 DATA_seq 为 12301 的包出错后，回复了错误的 ACK，此时发送方便会进行重发，于是接收方收到了正确的 DATA_seq 为 12301 的包，并回复了正确的 ACK。

```
125 2022-12-14 17:52:14:341 CST DATA_seq: 12201 ACKed
126 2022-12-14 17:52:14:355 CST DATA_seq: 12301 WRONG NO_ACK
127 2022-12-14 17:52:14:359 CST *Re: DATA_seq: 12301 ACKed
128 2022-12-14 17:52:14:387 CST DATA_seq: 12401 ACKed
129 2022-12-14 17:52:14:403 CST DATA_seq: 12501 ACKed

950 2022-12-14 17:52:14:310 CST ACK_ack: 12001
951 2022-12-14 17:52:14:326 CST ACK_ack: 12101
952 2022-12-14 17:52:14:342 CST ACK_ack: 12201
953 2022-12-14 17:52:14:357 CST ACK_ack: -1
954 2022-12-14 17:52:14:360 CST ACK_ack: 12301
955 2022-12-14 17:52:14:388 CST ACK_ack: 12401
956 2022-12-14 17:52:14:404 CST ACK_ack: 12501
```

5、RDT3.0

RDT3.0 假设底层信道可能丢包，这时需要设置让发送方等待一段时间，当某些数据或 ACK 丢失时，也就是发送方在这段等待的时间内没有收到 ACK，就进行重传。如果 ACK 仅仅被延迟而没有丢失，这时重传会导致包重复，我们可

以通过 seq 进行控制，也就是接收方必须指明被 ACK 分组所确认的 seq。

发送端：修改 rdt_send 函数，通过实例化一个 UDT_Timer 对象来作为一个计时器，再设置一个 UDT_RetransTask 对象，作为重传任务，之后我们将计时器加入重传任务，也就是如果超过我们设定的时间发送方还没有收到回复，就进行重传。具体的实现方式如下：

```
19 UDT_Timer timer;
20 /*构造函数*/
21 public TCP_Sender() {
22     super(); //调用超类构造函数
23     super.initTCP_Sender(this); //初始化TCP发送端
24 }
25
26 class My_UDT_RetransTask extends TimerTask {
27     private Client senderClient;
28     private TCP_PACKET reTransPacket;
29
30     public My_UDT_RetransTask(Client client, TCP_PACKET packet){
31         this.senderClient = client;
32         this.reTransPacket = packet;
33     }
34
35     @Override
36     public void run() {
37         System.out.println("Timeout and retransmit packets!");
38         this.senderClient.send(this.reTransPacket);
39     }
40 }
41
42 //设置计时器和重传任务
43 timer = new UDT_Timer();
44 UDT_RetransTask reTrans = new UDT_RetransTask(client, tcpPack);
45 //设置等待时间为2秒
46 timer.schedule(reTrans, 2000, 2000);
47 //等待ACK报文
48 waitACK();
49 //while (flag==0);
50 }
```

在这里，我们设置等待时间为 2 秒。同时，我们还要修改 waitACK 函数，如果收到正确的 ACK，就终止计时器。如果收到错误的 ACK，那么等待重传即可。具体的实现方式如下：

```
89 } else {
90     System.out.println("Clear: " + currentAck);
91     timer.cancel(); //收到正确的ACK，终止计时器
92     this.flag = 1;
93 }
94 }
```

我们还需要修改 udt_send 函数，将错误控制标志 eflag 修改为 4，也就是“出错/丢包”的状态。

```
69 public void udt_send(TCP_PACKET stcpPack) {
70     //设置错误控制标志，eflag==4，“出错/丢包”
71     tcpH.setTh_eflag((byte)4);
72     //System.out.println("to send: "+stcpPack.getTcpH().getTh_seq());
73     //发送数据报
74     client.send(stcpPack);
75 }
76 }
```

接收端：只需要修改 reply 函数，将错误控制标志 eflag 修改为 4。

```
96 public void reply(TCP_PACKET replyPack) {
97     //设置错误控制标志, eflag==4, “出错/丢包”
98     tcpH.setTh_eflag((byte)4);
99
100     //发送数据报
101     client.send(replyPack);
102 }
```

Log 分析

可以看到,发送端发送了一个 DATA_seq 为 7801 的包但是并没有收到 ACK, 而该包是丢失的状态, 在等待 2 秒后, 发送端重传了该数据包, 接收端也回复了正确的 ACK。

```
81 2022-12-14 17:58:26:011 CST DATA_seq: 7601 ACKed
82 2022-12-14 17:58:26:027 CST DATA_seq: 7701 ACKed
83 2022-12-14 17:58:26:042 CST DATA_seq: 7801 NO_ACK
84 2022-12-14 17:58:28:045 CST *Re: DATA_seq: 7801 ACKed
85 2022-12-14 17:58:28:064 CST DATA_seq: 7901 ACKed
86 2022-12-14 17:58:28:078 CST DATA_seq: 8001 ACKed

1049 2022-12-14 17:58:26:019 CST ACK_ack: 7801
1050 2022-12-14 17:58:26:042 CST ACK_ack: 7801 LOSS
1051 2022-12-14 17:58:28:046 CST *Re: ACK_ack: 7801
1052 2022-12-14 17:58:28:065 CST ACK_ack: 7901
1053 2022-12-14 17:58:28:079 CST ACK_ack: 8001
```

6、RDT4.0 Seclctive Response

用来表达窗口的数据结构是数组, 具体实现窗口的代码写在 Sender_Window.java 和 Receiver_Window.java 里。

在发送端中实现窗口满阻塞应用层的调用。具体实现方法: 在 TCP_Sender.java 的 rdt_send()函数中增加以下代码, 用来判断窗口是否已满。如果已满, 那么让 flag=0, 并且执行 while 循环, 从而实现窗口满阻塞应用层的调用。

```
42 //判断窗口是否已满
43 if (window.isFull()) {
44     System.out.println();
45     System.out.println("Sliding Window Full");
46     System.out.println();
47
48     flag = 0;
49 }
50 while (flag == 0) ;
```

窗口在确认后如何移动: 对于窗口外的确认不做响应。对于窗口内的 ACK, 我们要将其在窗口的对应的位置处的计时器停止并删除, 具体的实现方法如下: 在 Sender_Window.java 中的 receiveACK()函数中书写如下代码, 从而实现计时器的终止和删除。


```

40     if (this.base <= currentSequence && currentSequence < this.base + this.size) {
41         if (this.timers[currentSequence - this.base] == null) {
42             return;
43         }
44
45         this.timers[currentSequence - this.base].cancel(); //终止计时器
46         this.timers[currentSequence - this.base] = null; //删除计时器

```

如果接收到的 ACK 位于窗口左沿，这时候要进行窗口移动，具体的实现方法如下：在 Sender_Window.java 中的 receiveACK()函数中书写如下代码，首先计算窗口左沿应该移动到的位置，然后将窗口内的包移动，移动后清空已经左移的包原来所在位置处的包和计时器，最后更新窗口左沿的值和下一个包的插入位置。

```

48     if (currentSequence == this.base) {
49         int maxACKedIndex = 0;
50         while (maxACKedIndex + 1 < this.nextIndex
51             && this.timers[maxACKedIndex + 1] == null) {
52             maxACKedIndex++;
53         }
54
55         //将窗口的包左移
56         for (int i = 0; maxACKedIndex + 1 + i < this.size; i++) {
57             this.packets[i] = this.packets[maxACKedIndex + 1 + i];
58             this.timers[i] = this.timers[maxACKedIndex + 1 + i];
59         }
60
61         //清空原位置处的包和计时器
62         for (int i = this.size - (maxACKedIndex + 1); i < this.size; i++) {
63             this.packets[i] = null;
64             this.timers[i] = null;
65         }
66
67         this.base += maxACKedIndex + 1;
68         this.nextIndex -= maxACKedIndex + 1;
69     }

```

确认后的数据包是否去除：是。具体过程如下：接收方要维护一个接收窗口，使用数组来缓存正确接收的包，对于失序数组，要回复 ACK，对于正确序号的数组，要将其加入到缓存窗口中，如果这个位置正好是窗口左沿的话，还要进行窗口滑动，最后交付数据。具体的代码如下：

```

31     if (currentSequence < this.base) {
32         // ACK [base - size, base - 1]
33         int left = this.base - this.size;
34         int right = this.base - 1;
35         if (left <= 0) {
36             left = 1;
37         }
38         if (left <= currentSequence && currentSequence <= right) {
39             return currentSequence;
40         }
41     } else if (this.base <= currentSequence && currentSequence < this.base + this.size) {
42         this.packets[currentSequence - this.base] = packet;
43
44         if (currentSequence == this.base) {
45             this.slid();
46         }
47
48         return currentSequence;
49     }

```

计时器数组如何声明和去除：在 Sender_Window.java 中进行声明，并在

putPacket()函数中使用，之后在 receiveACK()函数中清除，具体的实现如下：

```
20 private TCP_PACKET[] packets = new TCP_PACKET[this.size];
21 private UDT_Timer[] timers = new UDT_Timer[this.size];

31 public void putPacket(TCP_PACKET packet) {
32     this.packets[this.nextIndex] = packet;
33     this.timers[this.nextIndex] = new UDT_Timer();
34     this.timers[this.nextIndex].schedule(new UDT_RetransTask(this.client, packet), 3000, 3000);
35
36     this.nextIndex++;
37 }

61 //清空原位置处的包和计时器
62 for (int i = this.size - (maxACKedIndex + 1); i < this.size; i++) {
63     this.packets[i] = null;
64     this.timers[i] = null;
65 }
```

累计确认后的窗口如何移动：首先要计算窗口左沿应该移动到的位置，也就是最小未收到数据包处，之后将已接收到的分组加入到交付队列中，将剩余位置的包左移，再将左移的包的原来的位置处置空，最后移动窗口左沿。具体代码的实现在 Receiver_Window.java 中的 slid()函数中实现，如下：

```
54 private void slid() {
55     int maxIndex = 0;
56     while (maxIndex + 1 < this.size
57         && this.packets[maxIndex + 1] != null) {
58         maxIndex++;
59     }

60     for (int i = 0; i < maxIndex + 1; i++) {
61         this.dataQueue.add(this.packets[i].getTcpS().getData());
62     }

63     for (int i = 0; maxIndex + 1 + i < this.size; i++) {
64         this.packets[i] = this.packets[maxIndex + 1 + i];
65     }

66     for (int i = this.size - (maxIndex + 1); i < this.size; i++) {
67         this.packets[i] = null;
68     }

69     this.base += maxIndex + 1;
70 }
```

在传输之前，我们要将错误控制标志 eflag 修改为 7，也就是“出错/丢包/延迟”的状态，如下。

```
72 @Override
73 //不可靠发送：将打包好的TCP数据报通过不可靠传输信道发送；仅需修改错误标志
74 public void udt_send(TCP_PACKET stcpPack) {
75     //设置错误控制标志，eflag==7，“出错/丢包/延迟”
76     tcpH.setTh_eflag((byte)7);
77     //System.out.println("to send: "+stcpPack.getTcpH().getTh_seq());
78     //发送数据报
79     client.send(stcpPack);
80 }
```

```

113 • @Override
114 //回复ACK报文段
115 public void reply(TCP PACKET replyPack) {
116     //设置错误控制标志, eflag==7, “出错/丢包/延迟”
117     tcpH.setTh_eflag((byte)7);
118
119     //发送数据报
120     client.send(replyPack);
121 }

```

Log 分析

对于丢包的情况, seq 为 32101 的包出现了 LOSS, 发送方没有收到该包的 ACK, 由于我们设定的计时器等待时间为 3000ms, 所以 3000ms 之后, 发送方重新发送了这个包并收到了正确的 ACK。

```

328 2023-01-02 16:21:37:350 CST DATA seq: 32001 ACKed
329 2023-01-02 16:21:37:359 CST DATA seq: 32101 LOSS NO_ACK
330 2023-01-02 16:21:37:370 CST DATA seq: 32201 ACKed

344 2023-01-02 16:21:37:517 CST DATA seq: 33601 ACKed
345 2023-01-02 16:21:40:359 CST *Re: DATA seq: 32101 ACKed
346 2023-01-02 16:21:40:362 CST DATA seq: 33701 ACKed

1367 2023-01-02 16:21:37:519 CST ACK ack: 33601
1368 2023-01-02 16:21:40:362 CST ACK ack: 32101
1369 2023-01-02 16:21:40:363 CST ACK ack: 33701

```

对于出错的情况, seq 为 37801 的包出现了 WRONG, 发送方没有收到该包的 ACK, 所以 3000ms 之后, 发送方重新发送了这个包并收到了正确的 ACK。

```

386 2023-01-02 16:21:40:783 CST DATA seq: 37701 ACKed
387 2023-01-02 16:21:40:794 CST DATA seq: 37801 WRONG NO_ACK
388 2023-01-02 16:21:40:804 CST DATA seq: 37901 ACKed

402 2023-01-02 16:21:40:960 CST DATA seq: 39301 ACKed
403 2023-01-02 16:21:43:795 CST *Re: DATA seq: 37801 ACKed
404 2023-01-02 16:21:43:796 CST DATA seq: 39401 ACKed

1424 2023-01-02 16:21:40:960 CST ACK ack: 39301
1425 2023-01-02 16:21:43:795 CST ACK ack: 37801
1426 2023-01-02 16:21:43:796 CST ACK ack: 39401

```

对于延迟的情况, seq 为 76501 的包出现了 DELAY, 发送方没有收到该包的 ACK, 所以 3000ms 之后, 发送方重新发送了这个包并收到了正确的 ACK。

```

781 2023-01-02 16:22:08:074 CST DATA seq: 76401 ACKed
782 2023-01-02 16:22:08:090 CST DATA seq: 76501 DELAY NO_ACK
783 2023-01-02 16:22:08:105 CST DATA seq: 76601 ACKed

797 2023-01-02 16:22:08:254 CST DATA seq: 78001 ACKed
798 2023-01-02 16:22:11:102 CST *Re: DATA seq: 76501 ACKed
799 2023-01-02 16:22:11:104 CST DATA seq: 78101 ACKed

1813 2023-01-02 16:22:08:255 CST ACK ack: 78001
1814 2023-01-02 16:22:11:104 CST ACK ack: 76501
1815 2023-01-02 16:22:11:105 CST ACK ack: 78101

```

7、TCP Tahoe&Reno

Tahoe 是 TCP 的最早版本, 其主要有三个算法去控制数据流和拥塞窗口, 分别是慢启动、拥塞避免和快重传。而 Reno 除了包含 Tahoe 的三个算法, 还多了一个 Fast Recovery (快速恢复) 算法。当收到三个重复的 ACK 或是超过了 RTO 时间且尚未收到某个数据包的 ACK, Reno 就会认为丢包了, 并认定网络中发生

了拥塞。Reno 会把当前的 ssthresh 的值设置为当前 cwnd 的一半，但是并不会回到 slow start 阶段，而是将 cwnd 设置为（更新后的）ssthresh+3MSS，之后 cwnd 呈线性增长。

在实验代码中，由于 Reno 版本仅比 Tahoe 多了快速恢复算法，其他均类似，所以我们在 Sender_Window.java 中定义了变量 RENO_FLAG，当该变量的值为 0 时，代码为 Tahoe 版本，当变量的值为 1 时，代码为 Reno 版本。该功能的实现写在 Sender_Window.java 的 receiveAck()函数中，通过一个 if else 分支结构来实现，如下：

```
120 // 快恢复
121 if (RENO_FLAG == 1) {
122     // TCP Reno版本
123     System.out.println("***** Fast Recovery *****");
124     if (cwnd / 2 < 2) {
125         System.out.println("ssthresh: " + ssthresh + " ---> 2");
126         ssthresh = 2;
127     } else {
128         System.out.println("ssthresh: " + ssthresh + " ---> " + cwnd / 2);
129         ssthresh = cwnd / 2;
130     }
131     System.out.println("cwnd: " + cwnd + " ---> " + ssthresh);
132     cwnd = ssthresh;
133     CongestionAvoidanceCount = 0;
134 } else {
135     // TCP Tahoe版本
136     if (cwnd / 2 < 2) {
137         System.out.println("ssthresh: " + ssthresh + " ---> 2");
138         ssthresh = 2;
139     }
```

在该版本中，我们设接收方的窗口大小为无限大。

在该版本中，发送方的滑动窗口编写在 Sender_Window.java 中，慢开始、拥塞避免、快重传和快恢复均在此代码中实现，对于该代码的介绍在 P14。

超时重传的内容编写在 TCP_Retry.java 中。

发送端 TCP_Sender.java 和接收端 TCP_Receiver.java 的具体介绍如下。

窗口满阻塞应用层调用的实现：与 Selective Response 版本类似，在发送端中实现窗口满阻塞应用层的调用。具体实现方法：在 TCP_Sender.java 的 rdt_send() 函数中增加以下代码，用来判断窗口是否已满。如果已满，那么让 flag=0，并且执行 while 循环，从而实现窗口满阻塞应用层的调用。其中，putPacket()函数的作用是处理发送方的滑动窗口。


```

46      //判断窗口是否已满
47      if (window.isFull()) {
48          System.out.println();
49          System.out.println("Sliding Window Full");
50          System.out.println();
51
52          flag = 0;
53      }
54      while (flag == 0) ;
55
56      try {
57          window.putPacket(tcpPack.clone());
58      } catch (CloneNotSupportedException e) {
59          e.printStackTrace();
60      }

```

在接收端 TCP_Receiver.java 中，我们使用 Hashtable 来缓存失序的分组，并且定义一个私有变量 expectedSequence 来表示我们期望收到的包的 seq，这样如果我们对收到的分组计算校验和，如果正确并且收到的 seq 就是我们期待的 expectedSequence，我们就将接收到的正确的包插入准备交付的 data 序列，并且随着正确的包的接收将 expectedSequence 进行自加 1 操作。具体的代码实现如下：

```

20      private int expectedSequence = 0; // 期望收到的seq
21      private Hashtable<Integer, TCP_PACKET> storagePackets = new Hashtable<>(); // 用于缓存失序分组
22
23      //接收数据报：检查校验和，设置回复的ACK报文段
24      public void rdt_rcv(TCP_PACKET rcvPack) {
25
26          //检查校验码，生成ACK
27          if(checkSum.computeChkSum(rcvPack) == rcvPack.getTcpH().getTh_sum()) {
28              int currentSequence = (rcvPack.getTcpH().getTh_seq() - 1) / 100; // 当前包的seq
29              if (expectedSequence == currentSequence) { // 收到的seq=expectedSequence
30
31                  // 将接收到的正确的包插入 data 队列，准备交付
32                  dataQueue.add(rcvPack.getTcpS().getData());
33                  expectedSequence += 1 ;
34              }
35          }
36      }

```

接下来，我们判断用来缓存失序包的哈希表中是否有数据，也就是从 expectedSequence 开始进行循环加 1 来检查，如果有数据就需要插入 data 队列中，同时在哈希表中删除交付的数据，具体的实现如下：

```

42      // 处理缓存数据
43      for (int i = expectedSequence; ; i++) {
44          if (storagePackets.containsKey(i)) {
45              dataQueue.add(storagePackets.get(i).getTcpS().getData());
46              expectedSequence += 1;
47              storagePackets.remove(i);
48          } else {
49              break;
50          }
51      }

```

之后交付数据即可，如下：

```

53      //交付数据（每20组数据交付一次）
54      if(dataQueue.size() >= 20)
55          deliver_data();

```

如果出现 seq 不等于 expectedSequence 的情况，则进行缓存操作。如果要缓

存的分组的 seq 小于 expectedSequence, 此时该 seq 已经交付, 因此无需缓存, 如果要缓存的分组的 seq 大于 expectedSequence, 则将其缓存, 具体的实现如下:

```
57     } else {
58
59         // 缓存失序分组
60         if (!storagePackets.containsKey(currentSequence) && currentSequence > expectedSequence) {
61             storagePackets.put(currentSequence, recvPack);
62         }
63     }
```

最后回复 ACK 即可。

```
93     //生成ACK报文段 (设置确认号)
94     tcpH.setTh_ack((expectedSequence - 1) * 100 + 1);
95     ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());
96     tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));
97     reply(ackPack); // 回复ACK报文段
98     System.out.println();
```

在 Sender_Window.java 中实现了慢开始、拥塞避免、快重传和快恢复。

首先我们定义了拥塞窗口 cwnd 和慢开始的门限 ssthresh 两个变量, 初始值分别是 1 和 16。定义了变量 CongestionAvoidanceCount, 记录进入拥塞避免状态收到的 ACK 的数量, 初始值是 0。定义了变量 lastACKSequence 和 lastACKSequenceCount, 分别记录上一次收到的 ACK 的 seq 和重复 ACK 的数量, 初始值分别是 -1 和 0, 这两个变量主要用于快重传的条件判断。定义了一个哈希表, 用来表示窗口, key 是分组序号 seq, value 是 TCP 的分组类型。最后我们定义一个计时器, 如下:

```
20     public int cwnd = 1;
21     private volatile int ssthresh = 16;
22     private int CongestionAvoidanceCount = 0; // 进入拥塞避免状态时收到的ACK数
23     private int lastACKSequence = -1; // 上一次收到的ACK的包的seq
24     private int lastACKSequenceCount = 0; // 收到重复ACK的次数
25     private Hashtable<Integer, TCP_PACKET> packets = new Hashtable<>();
26     private Timer timer; // 计时器
```

我们通过 isFull() 函数来判断发送窗口是否已满, 也就是当 cwnd 与 packets 相等时, 发送窗口已满。

```
70     public boolean isFull() {
71         return this.cwnd <= this.packets.size();
72     }
```

接下来, 我们编写了一个 putPacket() 函数, 其作用是将分组放入滑动窗口中。首先要计算当前的 seq, 如果加入的分组在窗口左沿, 也就是当 packet 为空时, 开启计时器, 我们设定的时间为 3000ms。之后在 packets 中加入这个分组。

```

74 public void putPacket(TCP_PACKET packet) {
75     int currentSequence = (packet.getTcpH().getTh_seq() - 1) / 100;
76     if (packets.isEmpty()) {
77         // 窗口左沿则开始计时器
78         timer = new Timer();
79         timer.schedule(new RetransmitTask(client, this), 3000, 3000);
80     }
81     packets.put(currentSequence, packet);

```

我们还编写了 receiveACK() 函数，判断是否有重复的 ACK，进一步判断是否是三个重复的 ACK，如果是的话启动快重传，重传的分组是上一个 ACK 对应的 seq 值加 1，并且重启计时器，具体的实现如下：

```

96 // 收到重复ACK
97 if (currentSequence == lastACKSequence) {
98     lastACKSequenceCount++;
99     if (lastACKSequenceCount == 4) { // 三个重复ACK，执行快重传
100         if (packets.containsKey(currentSequence + 1)) {
101             // 快重传
102             System.out.println("***** Fast Retransmit *****");
103             TCP_PACKET packet = packets.get(currentSequence + 1);
104             client.send(packet);
105             timer.cancel();
106             timer = new Timer();
107             timer.schedule(new TCP_Retry(client, this), 3000, 3000);
108         }

```

接下来是快恢复阶段，将 ssthresh 变为 cwnd 的 1/2，cwnd 变为 ssthresh，如果 cwnd 大于等于 ssthresh，则进入拥塞避免阶段。对于 Tahoe 版本，这里 cwnd 不是变为 ssthresh，而是置为 1。

```

110 // 快恢复
111 if (RENO_FLAG == 1) {
112     /*TCP_Reno版本*/
113     System.out.println("***** Fast Recovery *****");
114     if (cwnd / 2 < 2) {
115         System.out.println("ssthresh: " + ssthresh + " ---> 2");
116         ssthresh = 2; // ssthresh 不得小于2
117     } else {
118         System.out.println("ssthresh: " + ssthresh + " ---> " + cwnd / 2);
119         ssthresh = cwnd / 2; // 慢开始门限变为 cwnd
120     }
121     System.out.println("cwnd: " + cwnd + " ---> " + ssthresh);
122     cwnd = ssthresh; // cwnd 置为ssthresh
123     CongestionAvoidanceCount = 0;
124 } else {
125     /*TCP_Tahoe版本*/
126     if (cwnd / 2 < 2) {
127         System.out.println("ssthresh: " + ssthresh + " ---> 2");
128         ssthresh = 2; // ssthresh 不得小于2
129     } else {
130         System.out.println("ssthresh: " + ssthresh + " ---> " + cwnd / 2);
131         ssthresh = cwnd / 2; // 慢开始门限变为 cwnd 的一半
132     }
133     System.out.println("cwnd: " + cwnd + " ---> 1");
134     cwnd = 1; // cwnd 置为1
135     CongestionAvoidanceCount = 0;
136 }

```

接下来是清空拥塞避免的计时器。如果收到新的 ACK，那么直接清空计时

器，还要对当前的 seq 之前的分组进行删除，并且更新 lastACKSequence 为当前的 ACK 对应的 seq 值，并且更新 lastACKSequenceCount 为 1。

```
143 // 清空计时器
144 timer.cancel();
145
146 // 收到新的ACK
147 for (int i = lastACKSequence + 1; i <= currentSequence; i++) { // 清除前面的包
148     packets.remove(i);
149 }
150
151 lastACKSequence = currentSequence; // 重置lastACKSequence为当前收到ACK的包的seq
152 lastACKSequenceCount = 1; // 重置lastACKCount为1
```

如果窗口中仍然有分组，那么需要重新开启计时器，并且更新窗口的大小 cwnd 和慢开始的门限 ssthresh，如果 cwnd 小于 ssthresh 的话，那么说明这时处于慢开始阶段，每收到一个 ACK，窗口的大小 cwnd 就要加 1。如果 cwnd 大于等于 ssthresh 的话，说明此时处于拥塞避免阶段，计数器 CongestionAvoidanceCount 要加 1，如果计数器的值等于窗口大小，此时要使 cwnd 加 1 并清空计数器，如下：

```
154 if (!packets.isEmpty()) {
155     timer = new Timer();
156     timer.schedule(new TCP_Retry(client, this), 3000, 3000);
157 }
158
159 if (cwnd < ssthresh) {
160     // 慢启动
161     System.out.println("Slow Start");
162     System.out.println("cwnd: " + cwnd + " ---> " + (cwnd + 1));
163     cwnd++;
164     appendChange(currentSequence);
165 } else {
166     // 拥塞避免
167     CongestionAvoidanceCount++;
168     System.out.println("Congestion Avoidance");
169     System.out.println("cwnd: " + cwnd + " NO. " + CongestionAvoidanceCount);
170     if (CongestionAvoidanceCount == cwnd) {
171         CongestionAvoidanceCount = 0; // 重置计数器
172         System.out.println("cwnd: " + cwnd + " ---> " + (cwnd + 1));
173         cwnd++;
174         appendChange(currentSequence);
175     }
176 }
```

在 TCP_Retry.java 中实现了超时重传。首先要清空拥塞避免的计时器，然后立刻重传超时的分组。在超时重传窗口中，ssthresh 变为 cwnd 的一半，cwnd 置 1。


```

23 // 清空拥塞避免计数器
24 window.setCongestionAvoidanceCount(0);
25
26 // 超时重传
27 System.out.println("***** Timeout Retransmit *****");
28 if (window.getCwnd() / 2 < 2) {
29     System.out.println("ssthresh: " + window.getSsthresh() + " ---> 2");
30     window.setSsthresh(2);
31 } else {
32     System.out.println("ssthresh: " + window.getSsthresh() + " ---> " + window.getCwnd() / 2);
33     window.setSsthresh(window.getCwnd() / 2);
34 }
35 System.out.println("cwnd: " + window.getCwnd() + " ---> 1");
36 window.setCwnd(1);
37
38 window.appendChange(window.getLastACKSequence());

```

Log 分析

对于出错的情况，对于 seq 为 31101 的分组出现了 WRONG，于是接收方回复了已确认序号的最大的 ACK，也就是 31001，同时，发送方发送的 seq 为 31201 同样是 NO_ACK 的状态，因为接收方回复的也是已确认序号的最大的 ACK，也就是 31001，于是接收方同样缓存了 31201，但是继续往下可以发现，发送方发送的 seq 为 31301 的分组收到了 ACK，但是接收方回复的不是 31301，而同样是最大 ACK 的 seq，也就是 31001，这是因为发送方连续收到三个重复的 ACK 之后，执行了快重传，立刻重传了分组 31101，重传之后还要取消并且重开计时器，

```

316 2023-01-03 20:51:58:960 CST DATA_seq: 31001 ACKed
317 2023-01-03 20:51:58:975 CST DATA_seq: 31101 WRONG NO_ACK
318 2023-01-03 20:51:58:985 CST DATA_seq: 31201 NO_ACK
319 2023-01-03 20:51:58:997 CST DATA_seq: 31301 ACKed
320 2023-01-03 20:51:58:999 CST *Re: DATA_seq: 31101 NO_ACK
321 2023-01-03 20:51:59:010 CST DATA_seq: 31401 ACKed
322 2023-01-03 20:51:59:022 CST DATA_seq: 31501 ACKed

```

```

1325 2023-01-03 20:51:58:950 CST ACK_ack: 30901
1326 2023-01-03 20:51:58:961 CST ACK_ack: 31001
1327 2023-01-03 20:51:58:978 CST ACK_ack: 31001
1328 2023-01-03 20:51:58:987 CST ACK_ack: 31001
1329 2023-01-03 20:51:58:998 CST ACK_ack: 31001
1330 2023-01-03 20:51:59:001 CST ACK_ack: 31301

```

我们继续观察，接收方收到了这个重传的分组，同时缓存了失序分组 31201 和 31301，所以分组 31301 才会收到 ACK。

接下来，我们考虑 ACK 出错的情况。可以看到，seq 为 55201 的分组的 ACK 出错了，发送方对于 seq 为 55201 分组是 NO_ACK 状态，但是并未采取重传等措施，这是因为发送方已经接受到了后面的 seq 的 ACK，于是发送方认为该分组已经被接收端正确的接收。所以发送方对于出错的 ACK 不做任何处理。

```

1569 2023-01-03 20:52:01:760 CST ACK_ack: 55101
1570 2023-01-03 20:52:01:771 CST ACK_ack: -787971096 WRONG
1571 2023-01-03 20:52:01:783 CST ACK_ack: 55301

```

```

560 2023-01-03 20:52:01:760 CST DATA seq: 55101 ACKed
561 2023-01-03 20:52:01:770 CST DATA seq: 55201 NO_ACK
562 2023-01-03 20:52:01:782 CST DATA seq: 55301 ACKed

```

我们考虑丢包的情况,可以看到,seq 为 14001 的分组发生了 LOSS,由于 LOSS 的分组并没有 ACK,于是接下来发送方要连续发送三个分组之后才会启动快重传。在接收方我们也可以看出,接收方缓存了 14101 和 14201,重传之后,14001,14101 和 14201 都成功到达了接收方。

```

143 2023-01-03 20:51:56:840 CST DATA seq: 13901 ACKed
144 2023-01-03 20:51:56:850 CST DATA seq: 14001 LOSS NO_ACK
145 2023-01-03 20:51:56:861 CST DATA seq: 14101 NO_ACK
146 2023-01-03 20:51:56:874 CST DATA seq: 14201 NO_ACK
147 2023-01-03 20:51:56:885 CST DATA seq: 14301 ACKed
148 2023-01-03 20:51:56:889 CST *Re: DATA seq: 14001 NO_ACK
149 2023-01-03 20:51:56:897 CST DATA seq: 14401 ACKed

1154 2023-01-03 20:51:56:830 CST ACK ack: 13801
1155 2023-01-03 20:51:56:847 CST ACK ack: 13901
1156 2023-01-03 20:51:56:863 CST ACK ack: 13901
1157 2023-01-03 20:51:56:875 CST ACK ack: 13901
1158 2023-01-03 20:51:56:887 CST ACK ack: 13901
1159 2023-01-03 20:51:56:890 CST ACK ack: 14301

```

接下来,我们考虑接收方发送 ACK 丢失的情况。可以看到,ACK 为 18901 发生了 LOSS。发送方对于 seq 为 18901 分组是 NO_ACK 状态,但是并未采取重传等措施,这是因为发送方已经接受到了后面的 seq 的 ACK,于是发送方认为该分组已经被接收端正确的接收。

```

1204 2023-01-03 20:51:57:445 CST ACK ack: 18801
1205 2023-01-03 20:51:57:456 CST ACK ack: 18901 LOSS
1206 2023-01-03 20:51:57:468 CST ACK ack: 19001

194 2023-01-03 20:51:57:442 CST DATA seq: 18801 ACKed
195 2023-01-03 20:51:57:455 CST DATA seq: 18901 NO_ACK
196 2023-01-03 20:51:57:467 CST DATA seq: 19001 ACKed

```

我们考虑延迟的情况,可以看到,seq 为 10501 的分组发生了 DELAY,由于 DELAY 的分组并没有及时得到 ACK,于是接下来发送方要连续发送三个分组之后才会启动快重传,但是 DELAY 的分组最终会到达接收方。在接收方我们也可以看出,接收方缓存了 10601 和 10701,重传之后,10501,10601 和 10701 都成功到达了接收方。

```

107 2023-01-03 20:51:56:403 CST DATA seq: 10401 ACKed
108 2023-01-03 20:51:56:419 CST DATA seq: 10501 DELAY NO_ACK
109 2023-01-03 20:51:56:433 CST DATA seq: 10601 NO_ACK
110 2023-01-03 20:51:56:444 CST DATA seq: 10701 NO_ACK
111 2023-01-03 20:51:56:456 CST DATA seq: 10801 ACKed
112 2023-01-03 20:51:56:459 CST *Re: DATA seq: 10501 NO_ACK
113 2023-01-03 20:51:56:468 CST DATA seq: 10901 ACKed

1119 2023-01-03 20:51:56:388 CST ACK ack: 10301
1120 2023-01-03 20:51:56:404 CST ACK ack: 10401
1121 2023-01-03 20:51:56:434 CST ACK ack: 10401
1122 2023-01-03 20:51:56:446 CST ACK ack: 10401
1123 2023-01-03 20:51:56:457 CST ACK ack: 10401
1124 2023-01-03 20:51:56:460 CST ACK ack: 10801

```

接下来，我们考虑接收方发送 ACK 延迟的情况。可以看到，ACK 为 48201 发生了 DELAY。发送方对于 seq 为 48201 分组是 NO_ACK 状态，但是并未采取重传等措施，这可能是因为 ACK 到达发送方之前 TCP 连接已经释放。

1498	2023-01-03 20:52:00:965 CST ACK ack: 48101
1499	2023-01-03 20:52:00:980 CST ACK ack: 48201 DELAY
1500	2023-01-03 20:52:00:992 CST ACK ack: 48301
488	2023-01-03 20:52:00:965 CST DATA seq: 48101 ACKed
489	2023-01-03 20:52:00:979 CST DATA seq: 48201 NO ACK
490	2023-01-03 20:52:00:990 CST DATA seq: 48301 ACKed

二、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。

RDT 2.0、RDT 2.2、RDT 3.0、RDT 4.0 SR、RDT 5.0 (Tahoe)、RDT 5.1 (Reno) 版本均已经完成，除了前三个版本外，其他版本在完成时或多或少遇到了一些困难，不过均已经通过自己查资料、和同学交流方法等途径解决，最终完成了 Reno 版本。

三、说明在实验过程中采用迭代开发的优点或问题。

我认为迭代开发的优点是显然的，迭代开发从易到难，可以兼顾不同层次的学生，最简单的 RDT 2.0 版本，其核心仅需要完善校验和函数，这对于刚接触计算机网络这门课程的我还是比较友好的，上手较为容易。不过随后的几个版本难度都在逐渐增大，尤其到了拥塞控制阶段，需要我们不断的去翻阅课本，回顾课堂知识，这样可以加深我们对于课堂知识的理解。

至于迭代开发容易出现的问题，我也深有体会。比如我在做 RDT 3.0 版本时，意外的发现自己之前书写的一个函数有问题，这下我不仅需要重新改动代码，还要把之前已经使用过这部分代码的 RDT 2.2 版本也重新推翻，重新再进行分析等。所以迭代开发要求我们更加细心，每完成一个版本都尽量不要留下错误，以免后面填坑更加麻烦。

四、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

在我实现低版本的过程中并未遇到太大的问题，主要问题是在 Reno 版本的实现中，如何实现拥塞避免。我采用的方法是使用拥塞避免计数器，也就是在拥塞避免阶段收到正确的 ACK 就让计数器加 1，这样收到 16 个 ACK 之后就可以

让窗口的大小直接变成 17，但是退出拥塞避免时一定要清空计时器。

五、对于实验系统提出问题或建议

说一点自己的在实验过程中深有体会的建议（可能无关紧要），就是我在进行 Log 文件分析时，鼠标上下不断滑动，有一种“乱花渐欲迷人眼”的感觉。所以我认为如果 Log 文件中发送方和接收方分列左右两栏可能看起来会更加的简便。当然，上下布局的话并不会有什么大的问题，也只是一点无关紧要的小建议，这个 TCP 实验的意义远大于我在实验前想象到的，这个实验使我的课堂理论知识更加巩固，对知识的理解也更加深刻，在这里要说一句老师辛苦了，助教学长学姐辛苦了！