南开大学
Nankai University

南　　开　　大　　学

网 络 空 间 安 全 学 院

计算机网络实验报告

UDP 可靠传输 Part3

2014074 费泽锟

年级：2020 级

专业：信息安全

指导教师：徐敬东、张建忠

2022 年 12 月 10 日

摘要

在 UDP 实现可靠传输的 Part3 的设计中，主要是针对传输时的拥塞问题来进行设计的。在传输过程之中，如果仅仅使用滑动窗口协议，例如 GBN 算法和 SR 算法，一直保持一个恒定的窗口大小的话，如果实际情况下出现了拥塞问题，就很容易导致数据分组的丢失而导致大量数据分组的重传。

为了解决这个问题，在 Part3 的设计之中，主要实现了基于 New Reno 的拥塞控制算法。因为在 Part2 的部分之中，实现了完整的 GBN 算法和较为简略的 SR 算法，所以在 Part3 的设计之中，使用的超时重传策略是基于 GBN 算法的。

需要注意的是 Reno 与 New Reno 算法其实是针对 TCP 设计的，而 TCP 的滑动窗口协议与 GBN 算法和 SR 算法又有所不同，所以为了使 New Reno 算法在 GBN 上能够实现，我们还需要自主设计算法中的一部分，来使拥塞控制算法与 GBN 策略相匹配。

关键字：UDP 可靠传输、Reno、New Reno、超时重传

目录

一、 Part3 设计

(一) New Reno 在 GBN 上的实现

在 Part2 中的滑动窗口算法的实现中，实现了完整的 GBN 算法，实现了只基于一个计时器，超时重传策略与 GBN 算法相同的简易 SR 算法，因为考虑到 GBN 算法实现的更加完善，所以在 Part3 的拥塞控制实现过程之中是基于 GBN 的框架来进行设计的。

首先在开始我们的设计之前，我们首先需要知道一点点背景知识（因为真的很容易被混淆...）

（以下介绍摘自 https://zhuanlan.zhihu.com/p/126312611）

TCP:

1. TCP 使用累计应答的方式。这一点与 GBN 类似。
2. TCP 在接收端会设置缓存，来缓存正确接收但是失序的分组，这点与 SR 类似。（实际上 TCP RFC 并没有对接收端要怎样处理失序到达的分组提出要求，但是在接收端设置缓存是实践中大家都采用的方法）
3. TCP 使用快速重传机制：如果收到对于一个特定报文段的 3 个冗余 ACK，则在超时事件发生前就会对该报文段进行重传，这大大节约了时间。
4. 注意：TCP 中的 ACK 是指接收端希望从发送端收到的下一字节的序号。例如发送端发送了编号为 0-5 的字节，这时接收端成功接收后就会发送 ACK 为 6。

而对于 Reno 算法而言:

Reno 算法和 New Reno 算法均是针对的 TCP 协议中的拥塞问题而进行设计的，但是对于 TCP 我们能够看到，它兼具了 GBN 的特点也就是说重传重复的 ACK 序列号（等待确认的 ACK 序列号），也兼具了 SR 算法的特点那就是拥有缓冲区，如果乱序到达了在窗口内也能够进行确认。

具体的 TCP 设计实现流程可见（https://zhuanlan.zhihu.com/p/144273871）

所以，如果单纯地将 Reno 的状态机应用在 GBN 的滑动窗口设计或是 SR 的滑动窗口设计之中，都是会出现一定的问题的，具体的自主设计与修改将在后续继续介绍。

我们接下来看看经典的 Reno 拥塞控制算法到底做了件什么事儿。

下图展示的是 Reno 算法的有限状态机（FSM）:

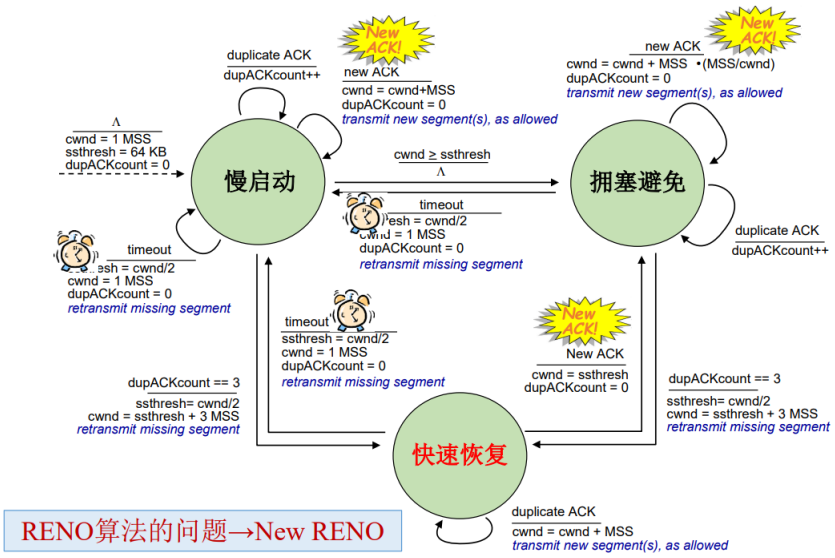

图 1: Reno 算法有限状态机

我们可以看到具体的 Reno 算法的有限状态机转换的过程，首先是慢启动阶段，这时的拥塞窗口为 1，我们需要注意的是每次的发送过程之中，发送端的窗口大小除了要考虑拥塞窗口的大小之外，还要考虑接收端的消息接收窗口，正常来说是需要接收端对其自身的接收窗口大小进行通告的，因为在实验过程之中实现的是文件的传输，而在我的 Part1 的设计之中已经将文件接受的缓冲区调整至 2560 个页的大小，所以这时我们可以直接在发送端设置好接收窗口的大小为 2560，每当成功接收新的 ACK 后递减即可。(这一策略显然在多路复用的情形下是不合适的)

在慢启动阶段，每次接收到新的 ACK 响应消息，那么拥塞窗口 cwnd 就会递增 +1，也就是说在一个 RTT 后会翻倍，当慢启动阶段的 cwnd 大小增大到大于 ssthresh 阈值时，就会进入拥塞避免阶段（其实也会有例外，例如在慢启动阶段接收到三次 ACK 消息，或者更加极端的是在慢启动阶段窗口尚未达到 3 的大小时的情形），特殊情况将在第三部分进行分析。

在拥塞避免阶段，每隔一个 RTT 时间 cwnd 递增 +1，这里的一个 RTT 时间有待商榷，在课堂讲授的内容之中，出现了两种线性增长的方式，如下展示:

拥塞避免算法

```
/* slowstart is over        */
/* cwnd >= ssthresh     */
Until (loss event) {
  every w segments ACKed:
     cwnd ++
  }
ssthresh = cwnd /2
If (loss detected by timeout) {
    cwnd = 1
    perform slowstart }
If (loss detected by triple
            duplicate ACK)
    cwnd = ssthresh + 3
```

图 2: 线性递增方式 1

在这段伪代码之中，我们能看到这里是每隔 w 个数据分组，cwnd 拥塞窗口的大小递增 +1，而在流程之中还给出了另一种方案:

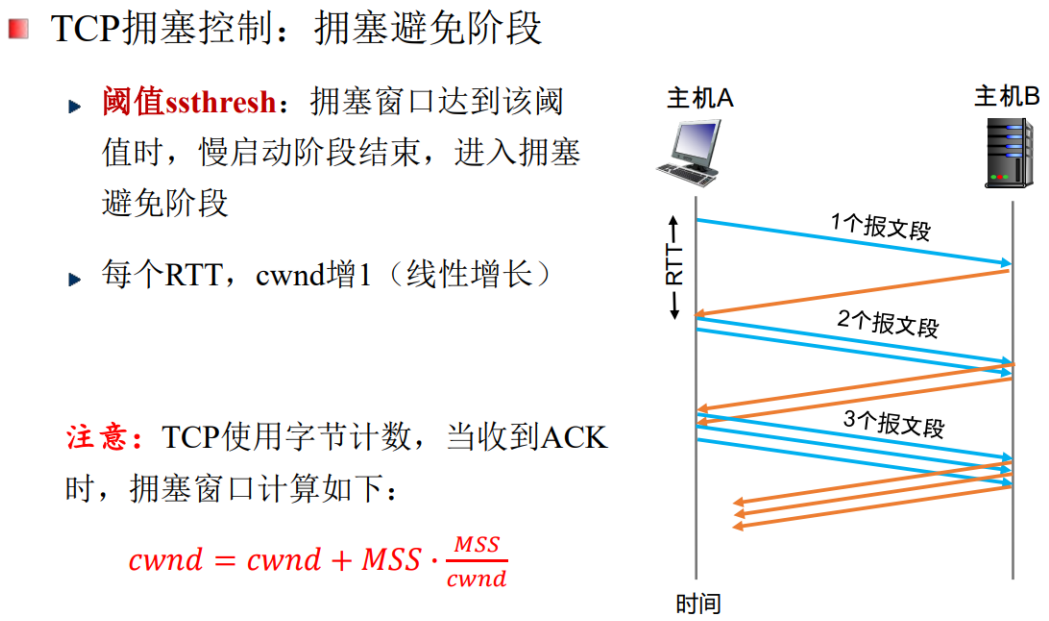

图 3: 线性递增方式 2

在第二种描述之中，每隔 cwnd 个数据分组，cwnd 拥塞窗口的大小递增 +1，其实这两种

方案在实现的过程之中都是可以的，但是在本机实验的过程之中，使用第二种方案会有更好的性能。

在拥塞避免阶段过程之中，如果出现了连续的 3 个重复的 ACK 消息就会进入快速恢复阶段，这时阈值减半，cwnd 重设，再退出快速恢复阶段后再重新设置为阈值，这时就会出现一个很复杂的问题，在 TCP 实现的过程之中对于正确到达的但是乱序的数据分组，会使用缓冲区将其进行存储，但是在 GBN 算法之中，所有没有按照正确顺序到达的数据分组都会被丢弃。

快速重传阶段的目的就是通过连续的三次 ACK 响应消息，来判断出现了拥塞情况和丢包的事件，来提前进行重传的操作避免继续线性递增发送更多的数据分组，但是 TCP 之中因为乱序的数据分组已经进行了缓存，所以在重传的过程之中只需要重传很少一部分数据分组即可。但是 GBN 会重传设计的消息队列之中的所有的数据分组，这显然可能继续增大拥塞的情形，而且在快速恢复阶段 cwnd 的值会随着重复的 ACK 消息不断的递增 +1，这很有可能使得 cwnd 窗口增大的速度甚至和慢启动阶段差不多。

而且对于 GBN 来说一定会连续失去数据分组，所以一定要实现的是 New Reno 算法来避免状态机在拥塞避免阶段和快速恢复阶段连续转换，而导致 cwnd 和阈值不断的减半减小。New Reno 算法的思想就是在进入快速恢复阶段时，会记录未确认的数据分组的最大序列号，直到到该序列号的所有数据分组都被确认后才转换到拥塞避免阶段。

所以在拥塞控制算法的实现过程之中，改编了一下 New Reno 算法，具体的源代码如下：

自主设计的 New Reno 算法

```
// 循环之中也接收ACK消息，可封装
if (recvfrom(SendSocket, RecvBuf, UDP_LEN, 0, (sockaddr*)&RecvAddr, &
    RecvAddrSize) > 0) {
    memcpy(&Recv_udp, RecvBuf, UDP_LEN);

    // 没有校验和的错误
    if (Recv_udp.udp_header.Flag == ACK && checksum((uint16_t*)&Recv_udp,
        UDP_LEN) == 0) {
        switch (RenoState) {
        // 慢启动
        case 0:
            cout << "***** 慢启动阶段 *****" << endl;
            cout << "Send Window Size: " << N << endl;
            cout << "base: " << base << endl;
            cout << "nextseqnum " << next_seqnum << endl;
            // 丢弃重复响应的ACK，取模防止回环问题
            if ((base % DEFAULT_SEQNUM) == Recv_udp.udp_header.SEQ) {
                if (cwnd < ssthresh) {
                    cwnd++;
                    N = min(cwnd, recv_window); // 需要更新发送窗口大小
                    cout << "*** 慢启动阶段接收到新ACK，cwnd++ ***" << endl;
                }
                else {
                    RenoState = 1;
                    cout << "*** 慢启动阶段结束，进入拥塞避免阶段 ***" <<
                        endl;
                }

```

```
26                  dupACKcount = 0; // 接收新的ACK, dup=0
27                   base = base + 1; // 确认一个移动一个位置
28                  cout << "Send has been confirmed! Flag:" << Recv_udp.
                        udp_header.Flag;
29                  cout << " STREAM_SEQ:" << Recv_udp.udp_header.STREAM_SEQ << "
                         SEQ:" << Recv_udp.udp_header.SEQ << endl;
30                  recv_window--;
31                  N = min(cwnd, recv_window);
32                  ACK_index++;
33                  message_queue.pop();
34                  start = clock();
35              }
36              // 重复的ACK
37              else {
38                  dupACKcount++;
39                  cout << "Repetitive ACK! Flag:" << Recv_udp.udp_header.Flag;
40                  cout << " STREAM_SEQ:" << Recv_udp.udp_header.STREAM_SEQ << "
                         SEQ:" << Recv_udp.udp_header.SEQ << endl;
41                  cout << "dupACKcount: " << dupACKcount << endl;
42              }
43              // 重复三次ACK
44              if (dupACKcount == 3) {
45                  // 检测到丢包, 开始提前重传
46                  cout << "*** ACK重复三次, 开始重传 ***" << endl;
47                  start = clock();
48                  for (int i = 0; i < message_queue.size(); i++) {
49                      send_packet_GBN(message_queue.front(), SendSocket,
                            RecvAddr);
50                      print_Send_information(message_queue.front(), "ReSend");
51                       message_queue.push(message_queue.front());
52                      message_queue.pop();
53                      // Sleep(10);
54                  }
55                  // 快速恢复阶段
56                  ssthresh = cwnd / 2;
57                  cwnd = ssthresh + 3;
58                  RenoState = 2;
59                  recover = message_queue.back().udp_header.SEQ;
60                  dupACKcount = 0;
61                   N = min(cwnd, recv_window); // 需要更新发送窗口大小
62              }
63              break;
64
65          // 拥塞避免阶段
66          case 1:
67              cout << "***** 拥塞避免阶段 *****" << endl;
68              cout << "Send Window Size: " << N << endl;
69              // cout << "RTT_ACK: " << RTT_ACK << endl;
```

```
70              cout << "base: " << base << endl;
71              cout << "nextseqnum " << next_seqnum << endl;
72              // 丢弃重复响应的ACK, 取模防止回环问题
73              if ((base % DEFAULT_SEQNUM) == Recv_udp.udp_header.SEQ) {
74                  RTT_ACK++;
75                  // 这里设置cwnd或者给定值应该都可以, 但是cwnd快
76                  if (RTT_ACK == cwnd) {
77                      cwnd++;
78                      N = min(cwnd, recv_window); // 需要更新发送窗口大小
79                      RTT_ACK = 0;
80                      cout << "*** 拥塞避免阶段, 线性递增+1 ***" << endl;
81                  }
82
83                  dupACKcount = 0; // 接收新的ACK, dup=0
84                  base = base + 1; // 确认一个移动一个位置
85                  cout << "Send has been confirmed! Flag:" << Recv_udp.
                        udp_header.Flag;
86                  cout << " STREAM_SEQ:" << Recv_udp.udp_header.STREAM_SEQ << "
                         SEQ:" << Recv_udp.udp_header.SEQ << endl;
87                  recv_window--;
88                  N = min(cwnd, recv_window);
89                  ACK_index++;
90                  message_queue.pop();
91                  start = clock();
92              }
93              // 重复的ACK
94              else {
95                  dupACKcount++;
96                  RTT_ACK = 0; // 保证RTT_ACK值的正确性
97                  cout << "Repetitive ACK! Flag:" << Recv_udp.udp_header.Flag;
98                  cout << " STREAM_SEQ:" << Recv_udp.udp_header.STREAM_SEQ << "
                         SEQ:" << Recv_udp.udp_header.SEQ << endl;
99                  cout << "dupACKcount: " << dupACKcount << endl;
100             }
101             if (dupACKcount == 3) {
102                 // 检测到丢包, 开始提前重
103                 cout << "*** ACK重复三次, 开始重传 ***" << endl;
104                 start = clock();
105                 for (int i = 0; i < message_queue.size(); i++) {
106                     send_packet_GBN(message_queue.front(), SendSocket,
                            RecvAddr);
107                     print_Send_information(message_queue.front(), "ReSend");
108                     message_queue.push(message_queue.front());
109                     message_queue.pop();
110                 }
111                 ssthresh = cwnd / 2;
112                 cwnd = ssthresh + 3;
113                 RenoState = 2; // 快速恢复阶段
```

```
114                 recover = message_queue.back().udp_header.SEQ; //设置New Reno
                        的覆盖
115                 RTT_ACK = 0;
116                 dupACKcount = 0;
117                 N = min(cwnd, recv_window); // 需要更新发送窗口大小
118             }
119             break;
120
121         // 快速恢复阶段
122         case 2:
123             cout << "***** 快速恢复阶段 *****" << endl;
124             cout << "Send Window Size: " << N << endl;
125             cout << "base: " << base << endl;
126             cout << "nextseqnum " << next_seqnum << endl;
127             // 所以要保留之前传输的最大序列号
128             // uint16_t recover = next_seqnum - 1;
129             if ((base % DEFAULT_SEQNUM) == Recv_udp.udp_header.SEQ) {
130                 // 新的ACK, New Reno会去判断是否发送的消息都已经被确认
131                 if (Recv_udp.udp_header.SEQ < recover) {
132                     RenoState = 2; // 维持快速恢复
133                 }
134                 else {
135                     RenoState = 1; // 进入拥塞避免阶段
136                     cwnd = ssthresh;
137                     N = min(cwnd, recv_window); // 需要更新发送窗口大小
138                      // dupACKcount = 0;
139                 }
140
141                 dupACKcount = 0;
142                 base = base + 1; // 确认一个移动一个位置
143                 cwnd++; // 成功接收就是网络状态好呗
144                 cout << "Send has been confirmed! Flag:" << Recv_udp.
                        udp_header.Flag;
145                 cout << " STREAM_SEQ:" << Recv_udp.udp_header.STREAM_SEQ << "
                         SEQ:" << Recv_udp.udp_header.SEQ << endl;
146                 recv_window--;
147                 N = min(cwnd, recv_window);
148                  ACK_index++;
149                  message_queue.pop();
150                   start = clock();
151             }
152              else {
153                 // 要是这块儿卡住了, 要是这块儿卡住了就返回慢启动吧(不行)
154                 // 要是不返回慢启动的话, 会导致快速恢复阶段时间过长, 阻塞窗口
                        过大
155                 dupACKcount++;
156                 /*
157                 // 这儿真的有待思考
```

```
158                 if (dupACKcount <= 3)
159                     cwnd++;
160                 else;
161                 */
162                 N = min(cwnd, recv_window); // 需要更新发送窗口大小
163                 cout << "*** 快速恢复阶段, cwnd: " << cwnd << endl;
164                 cout << "Repetitive ACK! Flag:" << Recv_udp.udp_header.Flag;
165                 cout << " STREAM_SEQ:" << Recv_udp.udp_header.STREAM_SEQ << "
                        SEQ:" << Recv_udp.udp_header.SEQ << endl;
166             }
167             // 重复六次ACK, 宽容一点
168             if (dupACKcount == 6) {
169                 // 检测到快速恢复阶段还在丢包, 那么直接进入慢启动
170                 cout << "*** ACK重复六次, 进入慢启动 ***" << endl;
171                 ssthresh = cwnd / 2; // 阈值
172                 cwnd = 1;
173                 dupACKcount = 0; // 重新计数
174                 N = min(cwnd, recv_window);
175                 RenoState = 0; // 慢启动阶段
176                 cout << "*** 重新进入慢启动阶段 窗口大小重设为" << N << " ***
                        " << endl;
177                 start = clock();
178
179                 for (int i = 0; i < message_queue.size(); i++) {
180                     send_packet_GBN(message_queue.front(), SendSocket,
                            RecvAddr);
181                     print_Send_information(message_queue.front(), "ReSend");
182                     message_queue.push(message_queue.front());
183                     message_queue.pop();
184                     // Sleep(10);
185                 }
186             }
187             break;
188
189         default:
190             cout << "Error RenoState!!!" << endl;
191             break;
192         }
193     }
194     else;
195 }
```

这里使用了 switch 结构来完成具体的状态机的转换过程，只不过为了使 New Reno 算法适配所有数据分组重传的策略，在自主设计之中改变了快速恢复阶段的策略，也就是在每次出现重复的 ACK 消息后不使 cwnd 拥塞窗口递增 +1，因为这样会导致 cwnd 快速增大（因为如果出现，重传数据包又丢失的情形，那么就会卡住，而如果像原来的 New Reno 中的一个一个重传处理则会效率很低，也有可能会卡住），而在 New Reno 的快速恢复阶段过程之中，每确认一个新的 ACK，cwnd 递增 +1（这里启发式的认为，如果有丢失的数据分组确认了，那么网络状况变

好)。

具体的该部分代码如下，实际的实验过程之中也展示了，这样的策略会有不错的性能（在 10% 的丢包率下，能在 3-4 秒内完成 1.jpg，大小为 1.8M 的传输）

自主设计的 New Reno 算法

```
// 快速恢复阶段
case 2:
    cout << "***** 快速恢复阶段 *****" << endl;
    cout << "Send Window Size: " << N << endl;
    cout << "base: " << base << endl;
    cout << "nextseqnum " << next_seqnum << endl;
    // 所以要保留之前传输的最大序列号
    // uint16_t recover = next_seqnum - 1;
    if ((base % DEFAULT_SEQNUM) == Recv_udp.udp_header.SEQ) {
        // 新的ACK, New Reno会去判断是否发送的消息都已经被确认
        if (Recv_udp.udp_header.SEQ < recover) {
            RenoState = 2; // 维持快速恢复
        }
        else {
            RenoState = 1; // 进入拥塞避免阶段
            cwnd = ssthresh;
            N = min(cwnd, recv_window); // 需要更新发送窗口大小
            // dupACKcount = 0;
        }

        dupACKcount = 0;
        base = base + 1; // 确认一个移动一个位置
        cwnd++; // 成功接收就是网络状态好呗
        cout << "Send has been confirmed! Flag:" << Recv_udp.udp_header.Flag;
        cout << " STREAM_SEQ:" << Recv_udp.udp_header.STREAM_SEQ << " SEQ:"
            << Recv_udp.udp_header.SEQ << endl;
        recv_window--;
        N = min(cwnd, recv_window);
        ACK_index++;
        message_queue.pop();
        start = clock();
    }
    else {
        // 要是这块儿卡住了，要是这块儿卡住了就返回慢启动吧(不行)
        // 要是不返回慢启动的话，会导致快速恢复阶段时间过长，阻塞窗口过大
        dupACKcount++;
        /*
        // 这儿真的有待思考
        if (dupACKcount <= 3)
            cwnd++;
        else;
        */
        N = min(cwnd, recv_window); // 需要更新发送窗口大小
```

```
43          cout << "*** 快速恢复阶段, cwnd: " << cwnd << endl;
44          cout << "Repetitive ACK! Flag:" << Recv_udp.udp_header.Flag;
45          cout << " STREAM_SEQ:" << Recv_udp.udp_header.STREAM_SEQ << " SEQ:"
                << Recv_udp.udp_header.SEQ << endl;
46      }
47      // 重复六次ACK, 宽容一点
48      if (dupACKcount == 6) {
49          // 检测到快速恢复阶段还在丢包, 那么直接进入慢启动
50          cout << "*** ACK重复六次, 进入慢启动 ***" << endl;
51          ssthresh = cwnd / 2; // 阈值
52          cwnd = 1;
53          dupACKcount = 0; // 重新计数
54          N = min(cwnd, recv_window);
55          RenoState = 0; // 慢启动阶段
56          cout << "*** 重新进入慢启动阶段 窗口大小重设为" << N << " ***" <<
                endl;
57          start = clock();
58
59          for (int i = 0; i < message_queue.size(); i++) {
60              send_packet_GBN(message_queue.front(), SendSocket, RecvAddr);
61              print_Send_information(message_queue.front(), "ReSend");
62              message_queue.push(message_queue.front());
63              message_queue.pop();
64          }
65      }
66      break;
```

在这一部分之中还会针对出现 GBN 重传策略下的快速恢复阶段又出现了丢包情况的处理，这一部分的讲解会在第三部分之中。

(二) New Reno 在 SR 上的思考

我们既然已经知道了，TCP 的实现上兼具着重复 ACK 的特征和缓冲区的特征。那么对于 SR 算法而言，如果简单的应用 New Reno 算法的状态机肯定也是会有点问题的。

问题主要集中在，实现的 SR 算法之中，每次传递回来的 ACK 都是已经确认的 ACK 数据分组序列号，那么按道理来说是不会传输重复的 ACK 的，那么对于 SR 算法如何进入快速重传阶段呢。

以下给出的是 SR 算法实现流程：

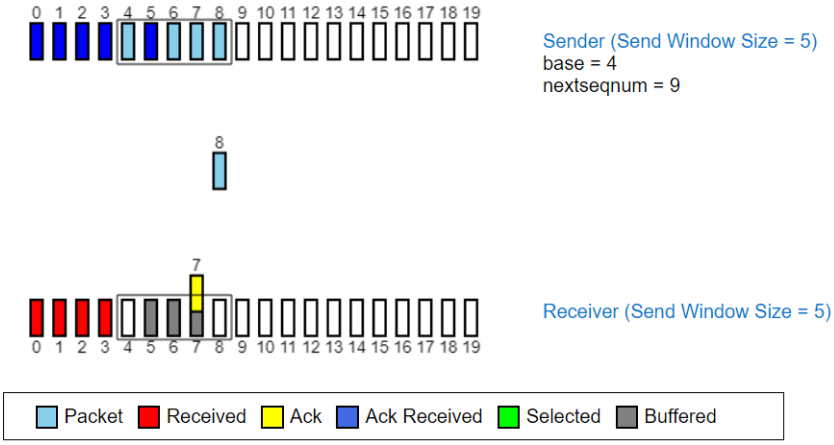

图 4: SR 算法实现流程

其实对于 TCP 而言，是用 3 个重复的 ACK 序列号来表征丢包事件的产生，那么我们可以简单地在 SR 算法之中使用 3 个失序的（也就是不是 base 序列号相应的）消息，来表征数据分组的丢失。在检测到丢失之后，我们只需要使用将已经确认的最大序列号存储下来，再将 base-> 最大序列号之间的所有未确认的分则进行确认即可。

其实，分析到这里我们已经发现了，如果想要像 TCP 一样实现 New Reno 算法，我们需要在 GBN 算法之中添加缓冲区确认机制，并且能够同时存在两个字段，分别来做确认数据分组和传输 base 值的功能。如果是基于 SR 算法的修改，我们就需要在传输回确认的数据分组同时传输卡住的 base 值，来造成 3 次重复 ACK 的跳转动作。

这些显然都是复杂的，需要重新设计 udp 报文等内容，因为时间原因这里就没有重构整体框架结构了。

(三) Part3 中实现的自主设计分析

在上文之中，我们已经讨论过了如果单纯地将 New Reno 算法的状态机在 GBN 算法或者是 SR 算法之上进行实现的话，会出现一定的问题，所以在实现 New Reno 算法的过程之中，进行了适当的自主设计，使其适配于完整的 GBN 算法。

首先，需要进行自主设计的第一点内容就是，在于快速恢复阶段的数据分组重传问题，该问题已经在上文进行了分析，因为对于 GBN 算法而言一旦数据分组丢失，那么窗口内的所有数据分组都会丢失，按照 New Reno 算法的要求需要重传所有的数据分组，实际上重传所有的数据分组，再将 cwnd 每次接收重复 ACK 后递增 +1，会加剧拥塞现象，因为这里我们已经知道了后续的所有数据分组都会被丢失。并且，如果在重传的过程之中，按照一个一个数据分组的重传，效率会很低，不同于 TCP 的缓冲机制，后续的数据分组我们已经清楚地知道丢失了，再按照三次重复 ACK 的策略显然是不合适的。所以这里的设计在接收重复的 ACK 时不再递增 +1，而是在快速恢复阶段每次接收一个新的 ACK 响应消息后递增 +1。

其次，出现的问题是在实验的过程之中，因为设置了 10% 的随机丢包概率，而导致的问题。这个问题就是，如果在重传窗口内的所有数据分组的过程之中，又出现了丢包的情况，无论是再次采用三次重复 ACK 来进行判断是否丢失，还是有序地重传所有数据分组，这时前序的重传确认的数据分组就会导致 cwnd 拥塞窗口的递增，那么这时就会继续发送数据包，而这些新发的数据包将继续丢包，这样的话就会一直循环导致进入慢启动阶段，这样显然是不合理的，在这里对其进行了调整。

具体的源代码如下:

自主设计的 New Reno 算法

```
// 重复六次ACK, 宽容一点
if (dupACKcount == 6) {
    // 检测到快速恢复阶段还在丢包, 那么直接进入慢启动
    cout << "*** ACK重复六次, 进入慢启动 ***" << endl;
    ssthresh = cwnd / 2; // 阈值
    cwnd = 1;
    dupACKcount = 0; // 重新计数
    N = min(cwnd, recv_window);
    RenoState = 0; // 慢启动阶段
    cout << "*** 重新进入慢启动阶段 窗口大小重设为" << N << " ***" << endl;
    start = clock();

    for (int i = 0; i < message_queue.size(); i++) {
        send_packet_GBN(message_queue.front(), SendSocket, RecvAddr);
        print_Send_information(message_queue.front(), "ReSend");
        message_queue.push(message_queue.front());
        message_queue.pop();
    }
}
```

也就是说在这里, 如果在重传的阶段之中又出现了丢包的情况, 那么就会导致 dupACKcount 不断递增, 如果 dupACKcount 的值大于了 6, 那么就会启发式的认为重传阶段又出现了丢包的现象, 所以直接进入慢启动阶段进行探测网络的行为, 这样的话那么经历慢启动阶段和快速恢复阶段的几次迭代后, 就可以正常把未确认的数据分组中的内容处理掉了。(因为此时的 cwnd 都会维持较小的值, 不会继续发送新的数据分组)

这里也会出现一个问题, 那就是如果进行有序的数据分组重传的话, 有可能会出现前一个阶段的重复 ACK 再后一个阶段才接收到, 这时其实修改框架为携带缓冲区的 ACK 确认就很少出现这种问题了, 相当于将其改为很像是停等机制的操作, 这样是最合理的。也可以选择添加阶段的标志位, 来进行判断是否丢失响应的 ACK。在本次设计之中采用了最简单的策略, 也就是延时来丢弃响应的 ACK, 这里也尝试了停等发送但是效率没有延时策略高。

最后一个问题也是在实验的过程之中发现的, 就是无论是在慢启动阶段还是在拥塞避免阶段等, 如果出现 cwnd 拥塞窗口的大小尚未达到 3 的时候出现了丢包的情况, 那么其实这种情况 Reno 算法就已经解决了。因为发送窗口的大小不足 3, 所以重复的 ACK 计数将无法达到 3, 这样就不会进入快速恢复阶段, 只能等到时钟超时重新进入慢启动阶段, 这里虽然也能优化, 但是为了维持原本的 New Reno 的模样就不进行更多其他的处理了。

补充部分:

后续时间充足, 又重构了一下 UDP 可靠传输 3-2 部分中的 GBN 实现, 就是将原来的单线程设计修改为了多线程设计。在上一次的实验报告之中已经详细阐述了单线程和多线程的区别, 这里就简单地讲解一下。

如果是在一个 while 循环之中实现的发送数据分组、接收 ACK 响应以及计时器操作的话, 其实就是将原本异步的操作进行了同步的处理, 滑动窗口有空余位置时就发送, 如果 socket 接收缓冲区之中有数据分组就接收, 如果超时就重传。单线程会将原本异步的事件变成了同步处理, 所以这显然可能因为程序的运行而导致一定的延迟, 这与实际情形时不符合的。

于是在时间富余的情况下，修改为了多线程实现，具体实现的源代码先给出再讲解其中需要注意的地方：

发送时 Send 线程函数

```
mutex slock;

DWORD WINAPI Send(LPVOID lparam_send) {
    // 先传入参数
    // cout << "send" << endl;
    Send_params* S_params = (Send_params*)lparam_send;
    string filename = S_params->filename;
    SOCKET SendSocket = S_params->SendSocket;
    sockaddr_in RecvAddr = S_params->RecvAddr;

    int RecvAddrSize = sizeof(RecvAddr);

    ifstream fin(filename.c_str(), ifstream::binary);
    fin.seekg(0, std::ifstream::end);
    long size = fin.tellg();
    file_size = size;
    fin.seekg(0);

    char* binary_file_buf = new char[size];
    cout << " ** 文件大小: " << size << " bytes" << endl;
    fin.read(&binary_file_buf[0], size);
    fin.close();

    HEADER udp_header(filename.length(), 0, START, stream_seq_order, 0);
    my_udp udp_packets(udp_header, filename.c_str());
    uint16_t check = checksum((uint16_t*)&udp_packets, UDP_LEN); // 计算校验和
    udp_packets.udp_header.cksum = check;
    // packet_num = size / DEFAULT_BUFLEN + 1;
    cout << " ** 文件名校验和: " << check << endl;
    cout << " ** 发送数据包的数量: " << packet_num << endl;
    cout << " ** Windows窗口大小: " << N << endl;

    // 正常发送第一个文件名数据包
    send_packet(udp_packets, SendSocket, RecvAddr);

    // char* RecvBuf = new char[UDP_LEN];
    // my_udp Recv_udp;
    start = clock();

    // 处理SEQ回环，mod运算，商和余数
    // uint16_t quotient = 0;
    uint16_t remainder = 0;
    while (ACK_index < packet_num) {
```

```
44          // slock.lock();
45          if (next_seqnum < base + N && next_seqnum <= packet_num) {
46              // quotient = next_seqnum / DEFAULT_SEQNUM;
47              slock.lock();
48              remainder = next_seqnum % DEFAULT_SEQNUM;
49              if (next_seqnum == packet_num) {
50                  udp_header.set_value(size - (next_seqnum - 1) *
                        DEFAULT_BUFLEN, 0, OVER, stream_seq_order, remainder);
51                  udp_packets.set_value(udp_header, binary_file_buf + (
                        next_seqnum - 1) * DEFAULT_BUFLEN, size - (next_seqnum -
                        1) * DEFAULT_BUFLEN);
52                  check = checksum((uint16_t*)&udp_packets, UDP_LEN);
53                  udp_packets.udp_header.cksum = check;
54
55                  // cout << "send window size: " << N << endl;
56                  send_packet_GBN(udp_packets, SendSocket, RecvAddr);
57                  print_Send_information(udp_packets, "Send");
58                  message_queue.push(udp_packets);
59                  next_seqnum++;
60              }
61              else {
62                  udp_header.set_value(DEFAULT_BUFLEN, 0, 0, stream_seq_order,
                        remainder);
63                  udp_packets.set_value(udp_header, binary_file_buf + (
                        next_seqnum - 1) * DEFAULT_BUFLEN, DEFAULT_BUFLEN);
64                  check = checksum((uint16_t*)&udp_packets, UDP_LEN);
65                  udp_packets.udp_header.cksum = check;
66
67                  // cout << "send window size: " << N << endl;
68                  send_packet_GBN(udp_packets, SendSocket, RecvAddr);
69                  print_Send_information(udp_packets, "Send");
70                  message_queue.push(udp_packets);
71                  next_seqnum++;
72              }
73              slock.unlock();
74          }
75
76          // 使用同一个socket进行发送
77          if (clock() - start > MAX_TIME) {
78              cout << "*** TIME OUT! ReSend Message *** " << endl;
79              ssthresh = cwnd / 2; // 阈值
80              cwnd = 1;
81              dupACKcount = 0; // 重新计数
82              N = min(cwnd, recv_window);
83              RenoState = 0; // 慢启动阶段
84              cout << "*** 进入慢启动阶段 窗口大小重设为" << N << " ***" <<
                    endl;
85              start = clock();
```

```
86
87              // 依旧是超时重传，全部重发 !!修改一下策略
88              // uint16_t front_seq = base;
89              slock.lock();
90              for (int i = 0; i < message_queue.size(); i++) {
91                  send_packet_GBN(message_queue.front(), SendSocket, RecvAddr);
92                  print_Send_information(message_queue.front(), "ReSend");
93                  message_queue.push(message_queue.front());
94                  message_queue.pop();
95                  int j = 0;
96                  while (j < 50) {
97                      j++;
98                  }
99              }
100             slock.unlock();
101         }
102         // slock.unlock();
103     }
104     delete[] binary_file_buf;
105     return 0;
106 }
```

就以上述的发送部分的线程函数为例，我们首先需要注意的是开启一个线程需要传入适当的参数，这里的参数类型是 void* 类型的，需要进行转换，同时因为我们可能需要传入许多的参数，这时可以设置一个 struct 结构体来进行参数的传入，具体示例如下：

发送线程参数结构体

```
1  struct Send_params {
2      string filename;
3      SOCKET SendSocket;
4      sockaddr_in RecvAddr;
5      Send_params(string f, SOCKET s, sockaddr_in r) {
6          filename = f;
7          SendSocket = s;
8          RecvAddr = r;
9      }
10 };
```

我们已经将需要的参数成功传入到线程了，但是有一件我们在 debug 的过程之中不得不进行处理的事儿，那就是在发送端和接收端我们都需要操作许多的全局变量，包括需要发送的数据分组数目也被重设为全局变量来保证线程循环能够成功退出。

这时我们就会遇到数据库和 OS 之中的经典问题，那就是同时读写条件竞争以及原子操作的问题，遇到这种问题，我们别无选择，只能选择进行上锁操作，这里我们需要利用 mutex 库，设置一个用于给原子操作进行上锁的互斥量。

我们其实不需要对全局变量来进行 lock guard 的操作，我们其实只需要对多线程之中的原子操作进行上锁就可以了，例如发送数据分组过程之中的发送一个数据分组或者是接收一次响应 ACK 消息。

在实现多线程的过程之中，仍会出现一个问题，那就是在发送线程和接收线程的过程之中，

如果使用一个 socket 会出现 abort 系统中断的问题，也就是说如果发送线程使用了一个 while 循环占据了 socket，那么接收线程的 while 循环就不能占据这个线程了，所以这里在每次线程开始时需要新开一个 socket 使用两个 socket 来完成多线程的应用。使用代码如下：

线程新创建 socket

```
Send_params* S_params = (Send_params*)lparam_send;
string filename = S_params->filename;
SOCKET SendSocket = S_params->SendSocket;
sockaddr_in RecvAddr = S_params->RecvAddr;
```

这里本来想使用三个线程：发送线程、接收线程以及计时器线程，但是计时器线程也涉及发送操作，多线程的经典问题就是竞争问题，如果这里为计时器也设置一个线程实验中会出现同时向接收端 socket 发送数据分组的情形就会导致程序崩溃，所以这里将计时器线程与发送线程进行了合并，统一使用一块 socket 内存进行发送。

至此，完整的拥塞设计就已经完成了。

二、 可靠 UDP 传输流程展示

在 3-3 的 UDP 传输流程测试之中，因为 router 无法使用的原因，这里进行了两部分的测试，一种是基于一定概率的随机丢包测试（与 3-2 相同），其实对于随机丢掉的数据分组可以视为因为拥塞而导致丢弃的数据分组，这里首先对这一部分进行展示讲解。

随机丢包测试：这次的展示部分好像没办法像之前那样易懂，其中有一些状态转换部分还是需要进行削微的分析的。在本次实验展示之中，设置的随机丢包概率为 10%，这显然是一个很大的概率了，根据 google 得到的结果实际网络情况之中，正常的丢包率最大值也就为 4% 左右，在我的设计之中考虑了高丢包率的情形：

首先就是正常的三次握手连接，四次挥手断开连接，command 命令过程，这里就不过多展示了基本流程同 3-1 和 3-2，这里给出一张示意图：

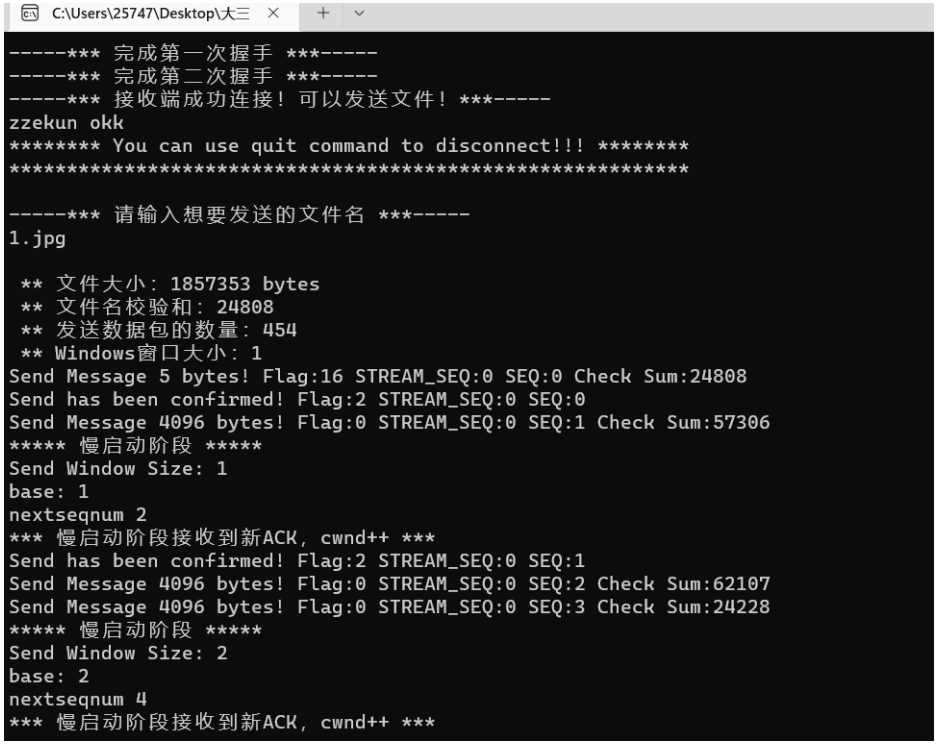

图 5: 基本流程展示

好的我们接下来就可以进行状态转换分析了，因为是随机丢包测试，出现了很多很多的问题需要一一解决，首先，很不幸的就是在随机丢包测试的过程之中，第三个数据分组就被丢失了（但是这其中也能看到 cwnd 指数增加的策略）：

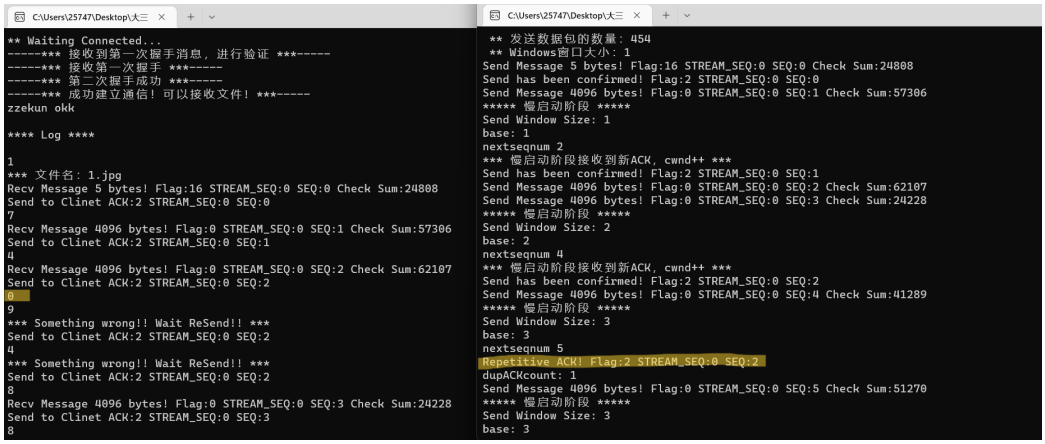

图 6: 第三个数据分组丢失

可见第三个数据分组的随机数为 0，正好是 0-9 中的不处理的那个数，这时就会除法丢包情形，这个时候就会出现一个问题，那就是重复的 ACK 响应一定到达不了 3，那么这时只能靠超时来让我们重新进入慢启动阶段了...

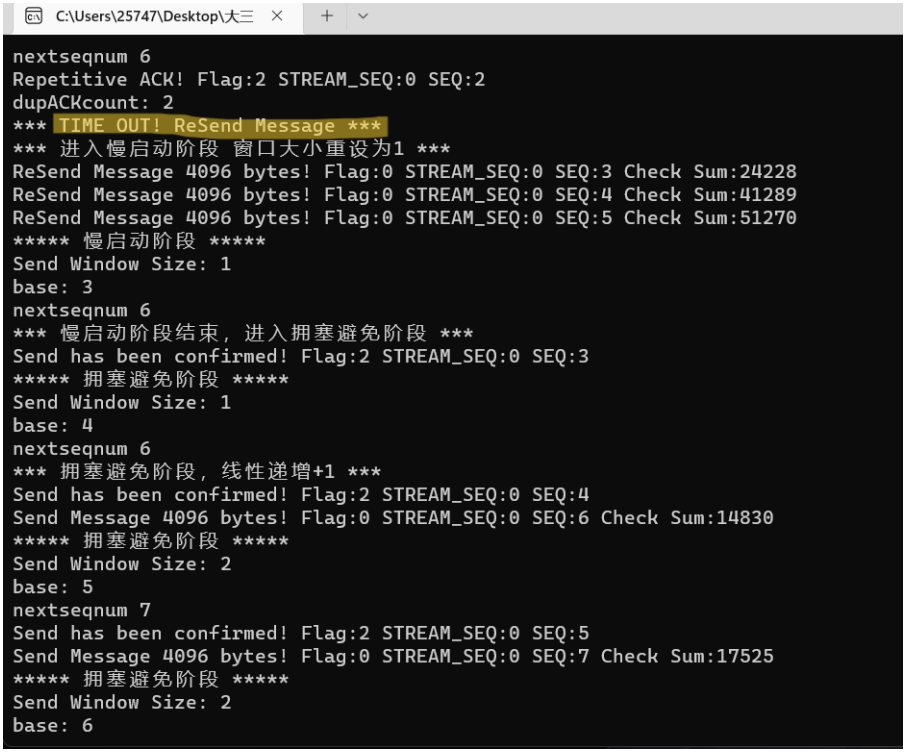

图 7: Time Out 进入慢启动阶段

经历了艰难坎坷的开始传输阶段，逐渐变得好起来了，这时就会根据阈值转换状态为拥塞避免阶段，并且在拥塞避免阶段可以看到，cwnd 窗口根据第一种策略进行变化的情形（这里使用第一种策略主要是因为第一种策略的效率更高）

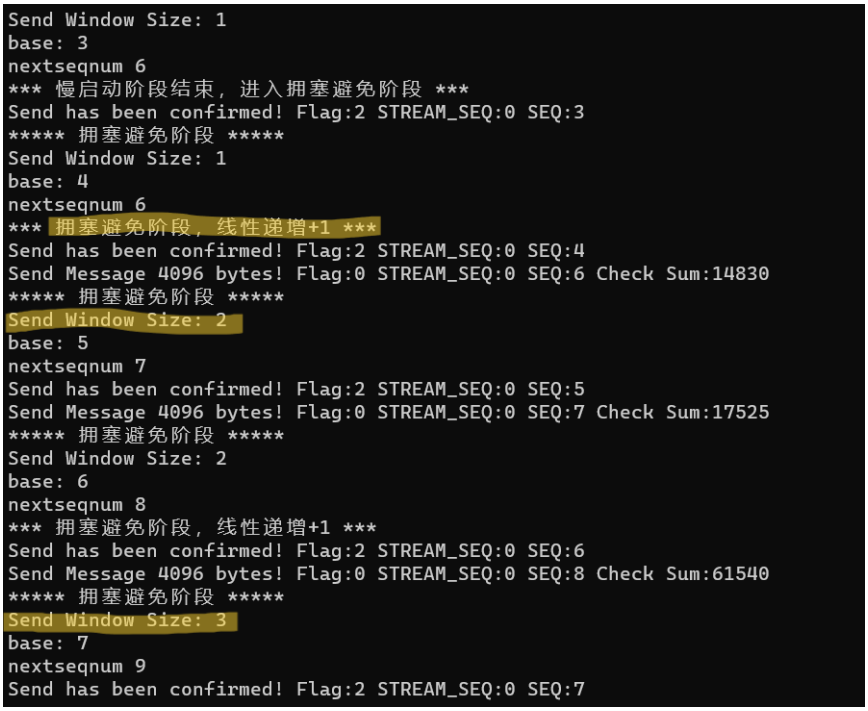

图 8: 进入拥塞避免阶段，cwnd 线性递增

但是更不幸的是，在序列号为 59 的数据分组这里我们遇到了致命的问题，那就是序列号为 59 的数据分组被丢弃了之后，接收到 3 次重复 ACK 进入了快速恢复阶段，但是正好重传的 59 号数据分组也被丢弃了，如果这里不进行处理会快进到卡住或循环多次的情形，这就出现了需要自定义修改算法的情形：

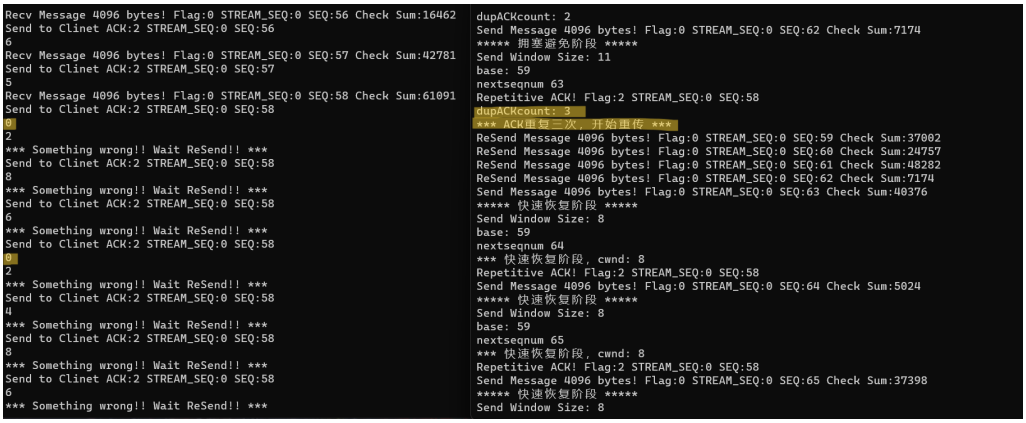

图 9: 不幸的情形，59 号数据分组丢失后重传也丢失了

可以看到，这里接收到了 3 次重复 ACK 已经进入了快速恢复阶段，这时就能看到我们的策略展现用处的地方了，如果又接收到了 3 次重复 ACK 就会提前预测重传的数据分组出现了丢失的情形，从而进入提前重传进入慢启动阶段：

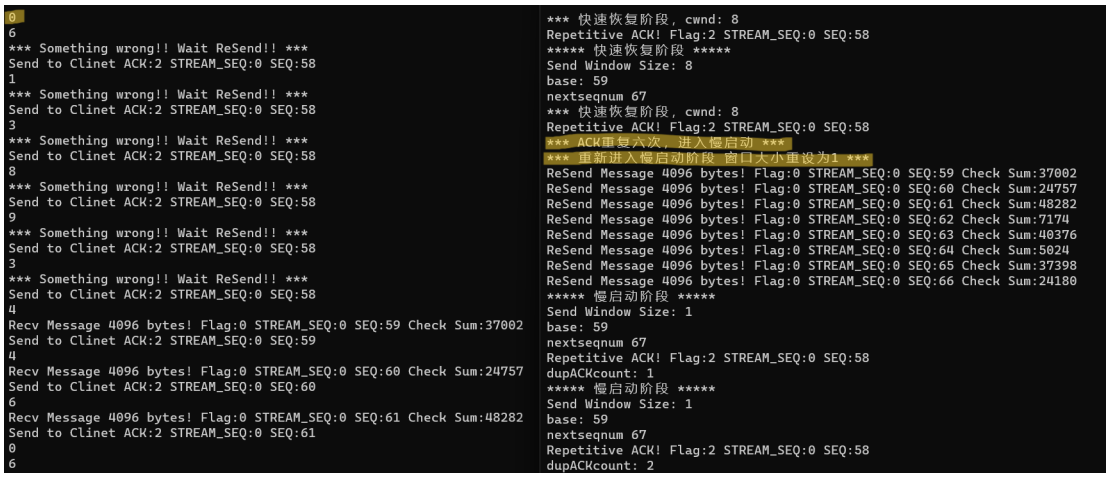

图 10: 6 次重复 ACK，提前进入慢启动阶段

我们可以看到，传输过程提前进入了慢启动阶段，这时因为在快速恢复阶段的策略被我进行了修改自定义，这时只有成功确认了新的数据分组才会 cwnd+1，如果快速恢复阶段接收到重复的 ACK 那么 cwnd 不会递增 +1，这时 cwnd 变化不会增加很多，就能够给重传数据分预留出充足的处理时间，直到处理结束。

```
***** 慢启动阶段 *****
Send Window Size: 1
base: 62
nextseqnum 68
*** 慢启动阶段接收到新ACK, cwnd++ ***
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:62
***** 慢启动阶段 *****
Send Window Size: 2
base: 63
nextseqnum 68
*** 慢启动阶段接收到新ACK, cwnd++ ***
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:63
***** 慢启动阶段 *****
Send Window Size: 3
base: 64
nextseqnum 68
*** 慢启动阶段结束, 进入拥塞避免阶段 ***
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:64
***** 拥塞避免阶段 *****
Send Window Size: 3
base: 65
nextseqnum 68
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:65
Send Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:68 Check Sum:41845
***** 拥塞避免阶段 *****
Send Window Size: 3
base: 66
nextseqnum 69
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:66
Send Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:69 Check Sum:57953
```

图 11: 慢启动阶段处理完成，重新进入拥塞避免阶段

通过修改一部分的 New Reno 算法就可以完整地完成文件传输流程了，最终看看吞吐率和传输时延：

```
nextseqnum 455
Send has been confirmed! Flag:2 STREAM_SEQ:0 SEQ:454
----- ***对方已成功接收文件！***-----

**传输文件时间为：5s
**吞吐率为:371471 bytes/s

------*** 请输入想要发送的文件名 ***------
```

图 12: 自定义 New Reno 的效率

为什么需要自定义 New Reno 中的部分策略，首先是需要解决重传又丢包的情形，其次是需要提升效率，最后就是尽量避免通过超时重传解决问题的次数。可见实现的改进 New Reno 算法是 3-2 之中滑动窗口的 2-3 倍（滑动窗口大小为 5），通过观察拥塞控制算法传输过程之中窗口普遍不高于 12，所以拥塞控制算法通过试探，在网络情况好的情形下加大发送力度，效率是有不错的提升的，能够在 4-5 秒就完成 1.8M 大小图片的传输。

除了随机丢包测试之外，这里为了实现真实的延时丢包测试，在接收端还实现了一个拥有缓冲区的简易 router 转发线程，将数据分组先存入 router 线程中的缓冲区中，通过 Sleep 函数在接收线程之中延时处理，这样处理速率不同，超出缓冲区的数据分组就会被丢弃，就能够实现延时丢包测试了，这里就不再赘述了，一下给出延时测试源代码：

简易 router 转发线程

```
1 DWORD WINAPI recv_router(LPVOID lparam_router) {
2     int iResult = 0;
3     bool flag = true;
4     router* temp_router = (router*)lparam_router;
```

```
5
6       while (flag) {
7           char* RecvBuf = new char[UDP_LEN]();
8           my_udp temp;
9           iResult = recvfrom(temp_router->RecvSocket, RecvBuf, UDP_LEN, 0, (
                SOCKADDR*)&(temp_router->SenderAddr), &(temp_router->
                SenderAddrSize));
10          if (iResult == SOCKET_ERROR) {
11              cout << "Recvfrom failed with error: " << WSAGetLastError() <<
                    endl;
12          }
13          else if (router_buffer.size() <= MAX_ROUTER) {
14              memcpy(&temp, RecvBuf, UDP_LEN);
15
16              // 新获取的数据分组入队
17              router_buffer.push(temp);
18              // if (temp.udp_header.Flag == OVER)
19                  // return 1;
20          }
21          else;
22
23          if (router_buffer.empty()) {
24              flag = false;
25          }
26          delete[] RecvBuf;
27      }
28
29      return 1;
30  }
```

这里接收端也是多线程的设计，如果我们使用该线程进行延时丢包测试，也能够看到策略的成功处理，展示图如下：

```
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:183 Check Sum:16110
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:184 Check Sum:1775
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:185 Check Sum:23032
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:186 Check Sum:48526
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:187 Check Sum:14033
***** 慢启动阶段 *****
Send Window Size: 1
base: 170
nextseqnum 188
Repetitive ACK! Flag:2 STREAM_SEQ:0 SEQ:169
dupACKcount: 1
***** 慢启动阶段 *****
Send Window Size: 1
base: 170
nextseqnum 188
Repetitive ACK! Flag:2 STREAM_SEQ:0 SEQ:169
dupACKcount: 2
***** 慢启动阶段 *****
Send Window Size: 1
base: 170
nextseqnum 188
Repetitive ACK! Flag:2 STREAM_SEQ:0 SEQ:169
dupACKcount: 3
*** ACK重复三次，开始重传 ***
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:170 Check Sum:25844
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:171 Check Sum:47828
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:172 Check Sum:36584
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:173 Check Sum:58479
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:174 Check Sum:13201
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:0 SEQ:175 Check Sum:35787
```

图 13: 延时丢包测试时的状态转换

最终也能够成功地传输文件，但是因为接收端的接收线程和 router 线程之间有 Sleep 函数人为的延迟，所以这里与 3-2 中的滑动窗口的效率没有办法直接地进行对比。

三、 总结

在本次实验之中，深入地了解了 TCP 的拥塞控制算法包括 Tahoe 算法、Reno 算法以及 New Reno 算法。在实现过程之中，对于多线程以及消息队列的相关知识进行了深入了解。由于 TCP 的机制，我们需要对拥塞控制算法进行一定的改编与优化，在实现完整的设计后可以发现在基于探测思想的拥塞控制算法，在 10% 丢包率的情形下是 3-2 效率的两倍多，有了不错的优化效果。

这里给出的是个人的 Github 仓库链接：

Computer Network

参考文献

02-计算机网络-第二章-2022（第二部分）.pdf
03-计算机网络-第三章-2022-marked.pdf
https://zhuanlan.zhihu.com/p/126312611
https://zhuanlan.zhihu.com/p/144273871