南開大學
Nankai University

计算机学院和网络空间安全学院

计算机网络实验报告

实验 3-3：拥塞控制算法实现

姓名：魏伯繁

学号：2011395

专业：信息安全

2022 年 12 月 23 日

目录

1 实验要求及实验目的

1.1 实验要求

在已经完成的可靠传输 GBN 算法的基础上，选择实现一种拥塞控制算法，也可以是改进的算法，完成给定测试文件的传输。

可以在 reno 算法的基础上自行设计协议或实现其他拥塞控制算法；

给出实现的拥塞控制算法的原理说明；

有必要日志输出（须显示窗口大小改变情况）。

1.2 实验目的

熟悉掌握理论课讲授的 RENO 算法，进一步加强对可靠数据传输以及拥塞控制算法的理解，通过动手编程提升网络编程能力

2 理论背景

2.1 GBN 传输

本次实验的基础是在实验 3-2 中实现的 3-2，具体的理论依据可以在实验 3-2 中查看，为了保证实验的完整性简单介绍 GBN 的实现机理

GBN 的特点如下：（1）允许发送端发出 N 个未得到确认的分组（2）分组首部中增加 k 位的序列号，序列号空间为 [0, 2k-1]（3）采用累积确认，只确认连续正确接收分组的最大序列号（4）可能接收到重复的 ACK，所以需要在发送端设置定时器，定时器超时时，重传所有未确认的分组

示意图如下图所示：

图 2.1: GBN 示意图

2.2 拥塞控制

让我们回顾在网络中数据包的传输过程，当报文分组到达时，如果出口链路忙，报文分组需要在路由器缓存中排队等待，引入排队时延；如果缓冲填满，报文分组会被丢弃；过长的排队时延和丢弃会对网络性能会产生较大的影响。

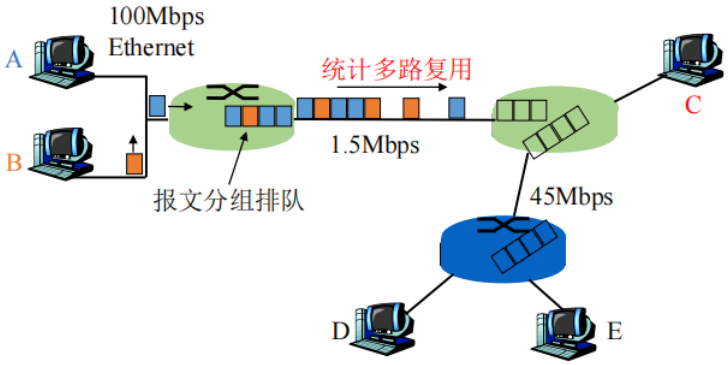

图 2.2: 数据包拥塞示意图

所以说，网络拥塞是指主机发送的数据过多或过快，造成网络中的路由器（或其他设备）无法及时处理，从而引入时延或丢弃。在存储转发交换所采用的统计多路复用机制中，拥塞不可避免。但是统计多路复用可以有效利用链路带宽资源，在所有的拥塞中，轻度拥塞可以接受，中度或严重拥塞需要避免。

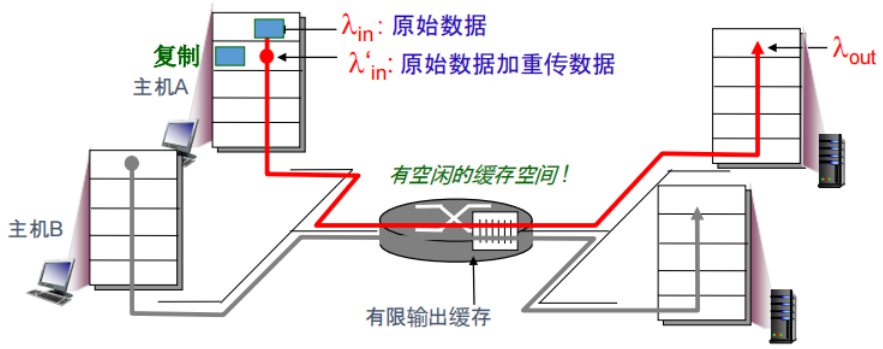

图 2.3: 路由器拥塞示意图

2.3 拥塞控制方法

端到端拥塞控制

网络中无明确的反馈

端系统通过观察丢失、延迟推断是否发生拥塞

网络辅助的拥塞控制

路由器提供到端系统的反馈

例如：可以使用 1 位指示拥塞（如 X.25, ATM）

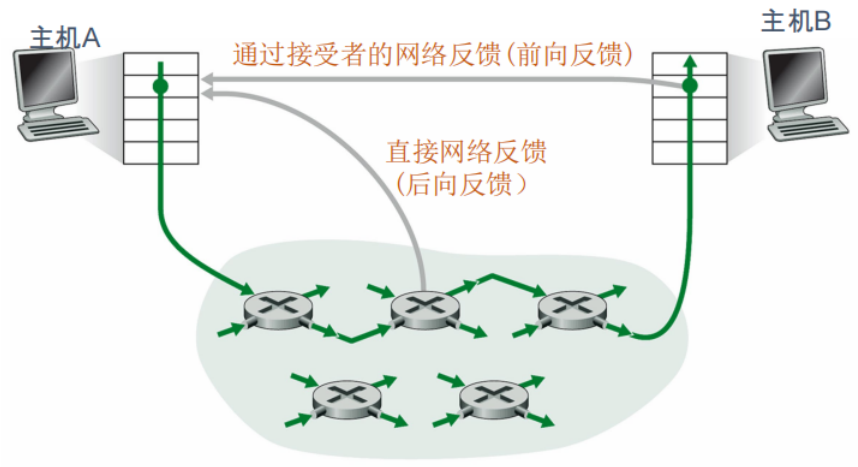

图 2.4: 网络辅助的拥塞控制

2.4 TCP 拥塞控制

经过前面的介绍，我们已经可以意识到，拥塞控制的核心就是：既不造成网络严重拥塞，又能更快地传输数据。

所以一种比较直观的方法就是带宽探测，当接收到 ACK 提高传输速率，发生丢失事件降低传输速率。如果正确收到 ACK 返回：说明网络并未拥塞，可以继续提高发送速率，如果发生丢失事件：假设所有丢失是由于拥塞造成的，可以适当的降低发送速率。

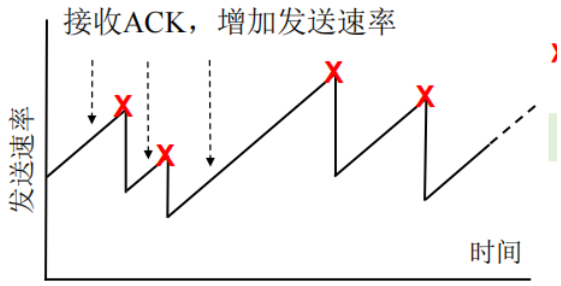

图 2.5: 带宽探测示意图

于是根据上面的设计我们可以很自然的给出一个基于窗口的拥塞控制算法：RENO 算法

2.5 RENO 算法

采用基于窗口的方法，通过拥塞窗口的增大或减小控制发送速率

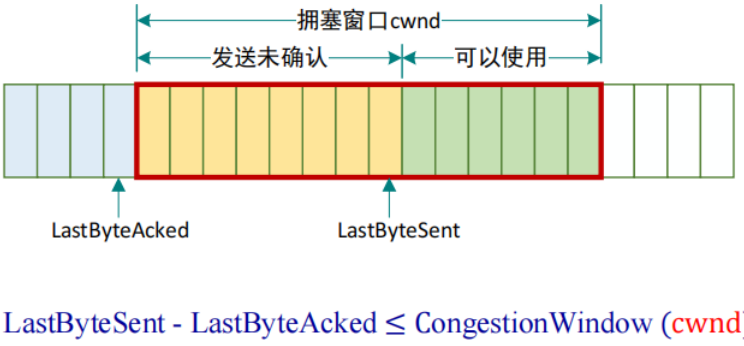

图 2.6: 基于窗口的拥塞控制算法

在这里，我们用 cwnd 来代表实际窗口的大小

在这里我们需要明确：实际发送窗口取决于接收通告窗口和拥塞控制窗口中较小值，但是在这里我们默认流量控制的接收通告窗口始终是较大的那一个，于是我们只考虑考虑拥塞控制算法。

在 RENO 算法中总共存在三个阶段，下面我们分别来介绍这三个阶段及其相互转换

2.5.1 慢启动阶段

当主机刚刚开始传输的时候会进入慢启动阶段，此时初始的窗口为 1 倍的 MSS，MSS 是我们固定的提前设定好的值，经过每个 RTT，如果我们正确收到了 ACK，那么此时 cwnd 翻倍，也就是成一个指数增长。在这个阶段的特点是：初始值小，增长速度快

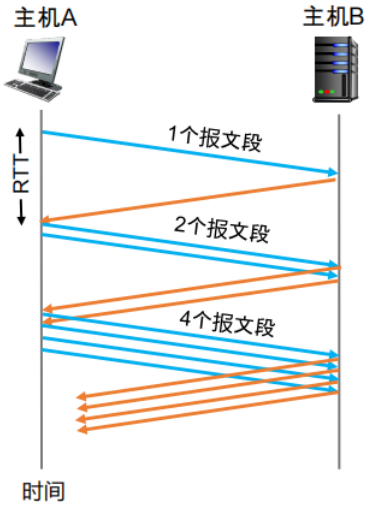

图 2.7: 慢启动示意图

2.5.2 拥塞避免阶段

当慢启动阶段进行了一段时间后，当 cwnd 到达阈值 ssthresh 时，会进入拥塞避免阶段，这个时候我们每接受到一个争取的 ACK 时 cwnd 呈线性增长。这个阶段的特点是：cwnd 的值普遍较大，也即是可以具有很多发出未确认的数据包，但是窗口增长的速度放缓。

具体的窗口大小变化公式为

$$cwnd = cwnd + MSS * \frac{MSS}{cwnd}$$

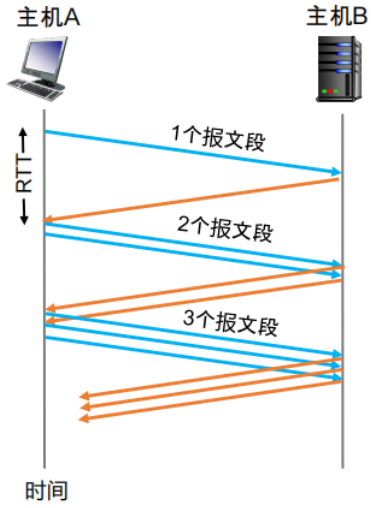

图 2.8: 拥塞避免示意图

2.5.3 丢失检测

在 RENO 算法中一共有两种丢失检测方式：
第一种方法为超时检测丢失，此时的参数变化为：

$$ssthresh = \frac{cwnd}{2} cwnd = 1(MSS)$$

在经历了超时检测后，无论此时处在哪一个状态，都将自动进入慢启动状态
第二种方法为三次重复 ACK 检测丢失算法，此时的参数变化为：

$$ssthresh = \frac{cwnd}{2} cwnd = ssthresh + 3(MSS)$$

当通过三次重复 ACK 检测后，会进入快速重传阶段，而后如果重传迅速解决丢包问题则会回到拥塞避免阶段。这个时候不回到一开始的慢启动阶段是因为，如果还能够收到 ACK 则说明网络仍可以交付一些报文段也就是拥塞不严重。

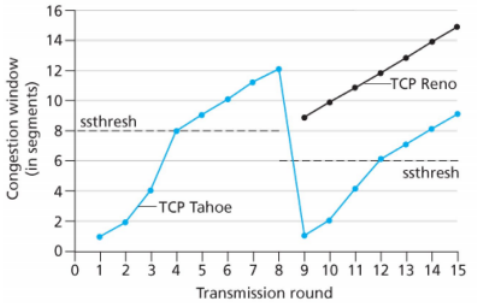

图 2.9: RENO 算法窗口变化示意图

2.5.4 状态转换

下图很好的介绍了 RENO 算法中的状态转换过程：当窗口的大小到达阈值时由慢启动阶段进入拥塞避免阶段；当收到三次重复的 ACK 时进入快速恢复阶段快速重传接收端需要的数据包；当主机识别超时时无条件进入慢启动阶段。

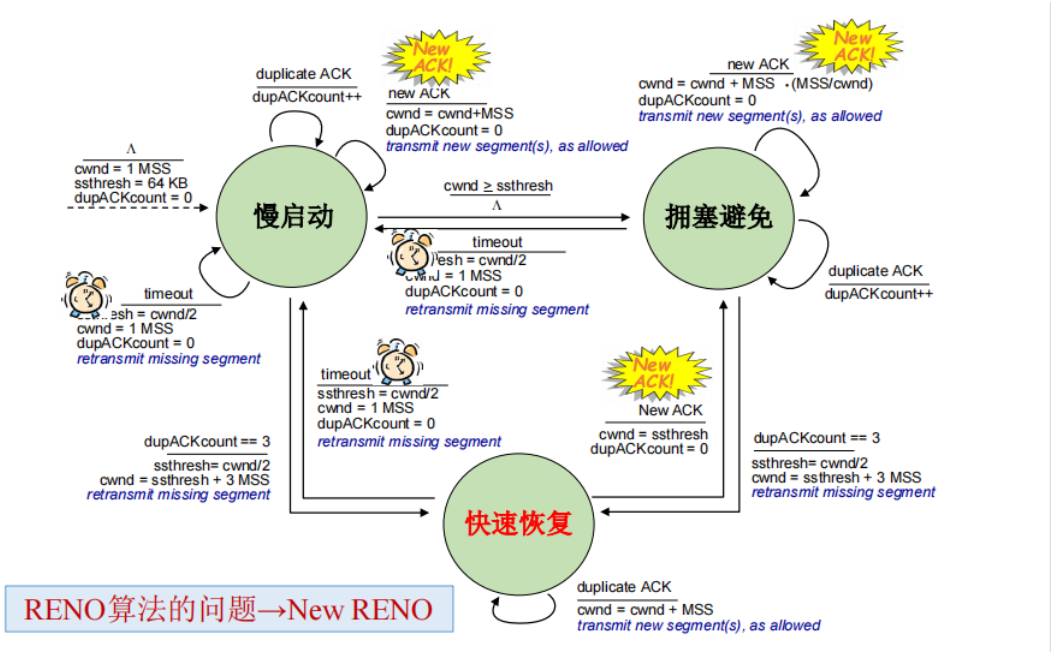

图 2.10: RENO 算法状态机

3 代码实现

3.1 主体介绍

首先，本次实现的思路主要为多线程实现数据包的传输工作，一个为收线程负责接受对方发送的数据包并做出反应，另一个数据包为发线程，负责向对方发送数据包。

而这两个线程通常会维护一些全局变量，比如窗口大小、现在接受的数据包序列号等等，为了避免线程冲突，我使用了互斥量 mutex 来解决线程之间的冲突问题。当线程访问全局变量时需要加锁，待全局变量访问结束后解锁。

其中一个比较明显的例子如下面的代码所展示的一样，其中 mtx 为声明的互斥量。

```
    mtx.lock();
    if (WINDOWNow < WINDOWStart) { WINDOWNow = WINDOWStart; }
    mtx.unlock();
```

3.2 客户端实现

3.2.1 接收端实现

接受端的主要工作是根据服务端发送的数据包的 ACK 调整状态与下一个要发送的数据包的位置，通过对全局变量 WINDOWNow 的调整，当发送线程访问时就可以发送对应的数据包了。

下面的代码展示了接收端的一段对于错误的 ACK 的处理函数，概括来说，我们根据错误的类型分成两类处理，一种是与由于多线程导致的，并没有发生丢包，只是参数没有正确的同步，这个时候我们同步参数就可以了，第二种可能是丢包了，这个时候我们就重塑环境，并且记录重复的 ACK 次数。

```

if (header.ack != WINDOWStart) {
sizeof(header) + MAX_DATA_LENGTH);
//不相等，其实一个很好的策略就是立刻重塑环境
if (header.ack > WINDOWStart) {
    //相当于你后面的我都收到了，那这个时候立刻收拾起始
    //但是这个时候不用充值 duplicateACK，因为如果发生这个错误是因为多线程导致的
    //实际上服务端已经收到包了，所以我只需要把现在要发的改成服务端想要的就行了
    mtx.lock();
    WINDOWStart = header.ack+1;
    WINDOWNow = WINDOWStart;
    WINDOWSIZE = 1;
    mtx.unlock();
}
else {
    mtx.lock();
    //这个才是真正的丢包了
    for (int i = header.ack + 1; i <= WINDOWNow;i++) {
        ifRecv[i] = false;
    }
    //现在要传输的置位
    WINDOWStart = header.ack + 1;
    WINDOWNow = WINDOWStart;
    WINDOWSIZE = 1;
    //正式进入 stage3
    if (++dupACKcount > 3) {
        dupACKcount = 3;
        ssthresh = WINDOWSIZE / 2;
```

```
29          if (ssthresh < 2) { ssthresh = 2; }
30          WINDOWSIZE = ssthresh + (3 * MSS / MAX_DATA_LENGTH);
31      }
32      cache.insert(header.ack);
33      mtx.unlock();
34      }
35  }
```

下面这段代码片段实现的是状态的转换，当接收到正确的 ACK 时对参数做出调整或者对当前的状态做出改变。具体的计算公式在前面展示的状态迁移图中已经给的非常详细了

```

2   mtx.lock();
3   ifRecv[header.ack] = true;
4   cout << "[RECV] 成功接受" << header.ack << " 号数据包的 ACK" << endl;
5   mtx.unlock();
6   //从快速重传恢复过来
7   if (stage == 3) {
8        mtx.lock();
9        dupACKcount = 0;
10      WINDOWSIZE = ssthresh;//从现在开始就还是进入拥塞避免阶段
11      stage = 2;
12      cout << "[stage] 进入拥塞避免阶段" << endl;
13      mtx.unlock();
14      continue;
15  }
16  if (stage == 1) {
17      //相当于翻倍
18      mtx.lock();
19      WINDOWSIZE = (2 * WINDOWSIZE * MSS) / MAX_DATA_LENGTH;
20      if (WINDOWSIZE >= ssthresh) {
21          WINDOWSIZE = ssthresh;
22          stage = 2;
23          cout << "[stage] 进入拥塞避免阶段" << endl;
24      }
25      if (WINDOWSIZE >= MAX_WINDOWSIZE) { WINDOWSIZE = MAX_WINDOWSIZE; }
26      mtx.unlock();
27      continue;
28  }
29  if (stage == 2) {
30      //相当于加一
31      mtx.lock();
```

```
32      WINDOWSIZE = (1 + WINDOWSIZE) * MSS / MAX_DATA_LENGTH;
33      if (WINDOWSIZE >= MAX_WINDOWSIZE) { WINDOWSIZE = MAX_WINDOWSIZE; }
34      //在这补偿一下吧，要不一直减太难受了
35      ssthresh++;
36      mtx.unlock();
37  }
```

3.2.2 发送端实现

发送端的实现其实主要就是三个部分：第一个部分是检查现在是否处于快速重传阶段，如果是那么就传那个接收端想收到的数据包；第二部分就是如果是正常的慢启动或者拥塞避免，那么看看现在的窗口大小还能不能发送数据包了，如果可以就发包；第三个阶段就是超时阶段，只要超时了立刻进入慢启动阶段。由于发送数据包的地方比较冗长而且大同小异，所以在这里就不同意展示了，只展示当现在是快速重传阶段的数据包发送代码。

```
1
2   if (stage == 3) { //快速重传的处理
3       //只要你没收到，我就一直传
4       //相当于这里就不是 WINDOWNow 决定的了
5       mtx.lock();
6       header.seq = WINDOWStart;
7       memset(sendBuffer, 0, sizeof(header) + MAX_DATA_LENGTH);
8       memcpy(sendBuffer, &header, sizeof(header));
9       int ml;
10      if ((header.seq + 1) * MAX_DATA_LENGTH > messagelength) {
11           ml = messagelength - (header.seq) * MAX_DATA_LENGTH;
12      }
13      else {
14          ml = MAX_DATA_LENGTH;
15      }
16      header.length = ml;
17      memcpy(sendBuffer + sizeof(header), message+((header.seq)*MAX_DATA_LENGTH), ml);
18      header.checksum = calcksum((u_short*)sendBuffer, sizeof(header) + MAX_DATA_LENGTH);
19       memcpy(sendBuffer, &header, sizeof(header));
20      //因为是重发 没必要更新两个 map 也不用更新 messagepointer
21      sendto(client, sendBuffer, sizeof(header) + MAX_DATA_LENGTH, 0, (sockaddr*)&router_addr, ri
22      WINDOWNow = WINDOWStart;
23      cout << "[send] 快速重传阶段重传"<<header.seq<<" 号数据包" << endl;
24      mtx.unlock();
25      continue;
26      }
27      //从状态三退出的后遗症
```

```
28      mtx.lock();
29      if (WINDOWNow < WINDOWStart) { WINDOWNow = WINDOWStart; }
30      mtx.unlock();
31      //看能不能发
32      //加锁？
33      //这里是 WINDOWNow 决定的地方
34      mtx.lock();
35      if (CanSend(WINDOWStart, WINDOWNow)) {//如果可以发送的话
36          //计算发送的长度
37          if (WINDOWNow > messagelength / MAX_DATA_LENGTH) {
38              if (ifRecv[messagelength / MAX_DATA_LENGTH] == true) {
39                  //发送数据包，省略了
40              }
41          }
42          else {
43              //发送数据包，省略了
44          }
45      }
46      //看能不能调整窗口
47      //加锁？
48      mtx.unlock();
49      mtx.lock();
50      if (ifRecv.find(WINDOWStart) != ifRecv.end()) {
51          if (ifRecv[WINDOWStart] == true) {
52              ifRecv[WINDOWStart] = false;
53              if (WINDOWStart != SEQSIZE) { WINDOWStart++; }
54              else { WINDOWStart = 1; }
55              c = clock();
56              int windowEnd = WINDOWStart + WINDOWSIZE;
57              cout << "[WINDOWS] 现在的窗口大小为： " << WINDOWStart << " - " << windowEnd << endl
58          }
59          //从状态三退出的后遗症
60          if (WINDOWNow < WINDOWStart) { WINDOWNow = WINDOWStart; }
61      }
62      mtx.unlock();
63      //超时 进入慢启动状态
64      if (clock() - c > MAX_TIME) {
65          mtx.lock();
66          WINDOWNow = WINDOWStart+1;//在这里调整一下
67          WINDOWSIZE = (1 * MSS) / MAX_DATA_LENGTH;
68          ssthresh /= 2;
69          if (ssthresh < 1) { ssthresh = 1; }
```

```
            stage = 1;
            //在这里要完成重传我觉得
            //加锁？
            int ml;
            if ((WINDOWStart + 1) * MAX_DATA_LENGTH > messagelength) {
                ml = messagelength - (WINDOWStart)*MAX_DATA_LENGTH;
            }
           //发送数据包，省略了
            mtx.unlock();
        }
    }
    return 1;
```

3.3 服务端实现

3.3.1 接收端实现

接收端的任务主要是获取客户端数据包的定义，把正确的数据包所携带的数据留下，把不正确的数据包丢弃并且同志发送端收到了不正确的数据包，通过发送端发送正确的想要发送的数据包。截取的部分核心代码如下所示：

```

    mtx.lock();
    //成功接收了数据包，是需要的数据包
    if (header.seq == SEQWanted&&!canSend) {
        cout << " 成功接收" << header.seq << " 号数据包" << endl;
        canSend = true;
        memcpy(message + nowpointer, recvbuffer + sizeof(header), header.length);
        nowpointer += header.length;
        mtx.unlock();
    }
    else {
        //可能是因为线程没同步正确
        //也有可能就是传乱了
        mtx.unlock();
        }
    }
        mtx.lock();
        //长时间没有收到客户端的信息那就直接退出
        if (clock() - c > MAX_TIME&&SEQWanted>0) {
            cout << "[exit] 开启自动退出机制" << endl;
            messagepointer = nowpointer - 1;
            canExit = true;
```

```
            canLoad = true;
            mtx.unlock();
            return 1;
        }
        mtx.unlock();
    }
```

3.3.2 发送端实现

发送端具体的功能就是向客户端发送指定的 ACK，而能否发送 ACK 则由接收线程通过对全局变量 canSend 的操作来决定，如果该变量为 true 则可以发送，反之则不能发送

```

    if (canSend) {
            header.ack = SEQWanted;
            SEQWanted++;
            if (SEQWanted > SEQSIZE) { SEQWanted = 1; }
            canSend = false;
            header.checksum = calcksum((u_short*)&header, sizeof(header));
            memcpy(sendbuffer, &header, sizeof(header));
            sendto(server, sendbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen);
            cout << " 成功发送" << header.ack << " 号 ACK" << endl;
            c = clock();
            //if 解锁
            mtx.unlock();
            continue;
        }
    else {
        //如果长时间没有收到自己想要的数据包
        //就把想要的数据包发过去
        //因为多线程有的时候可能有点乱
         //else 解锁
         mtx.unlock();
        //加锁
        mtx.lock();
        if (clock() - c > MAX_TIME&&SEQWanted>0) {
            header.ack = SEQWanted - 1;
            header.checksum = calcksum((u_short*)&header, sizeof(header));
            memcpy(sendbuffer, &header, sizeof(header));
            sendto(server, sendbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen);
            cout << " 成功发送" << header.ack << " 号 ACK" << endl;
            c = clock();
```

```
31            //if 解锁
32            mtx.unlock();
33            continue;
34            }
35            //else 解锁
36            mtx.unlock();
37        }
38
```

4 总结

通过本次实验，我对 RENO 算法以及拥塞控制的流程、具体状态的迁移都有了一个系统全面的认识，并通过多线程编程进一步学习了 mutex 的用法，并且对需要控制的变量在计算机网络算法中的作用有了更加深刻的认识。