



南開大學
Nankai University

计算机学院
计算机网络实验报告

UDP 可靠数据传输

姓名：赵一名

学号：2013922

专业：计算机科学与技术

2022 年 12 月 9 日

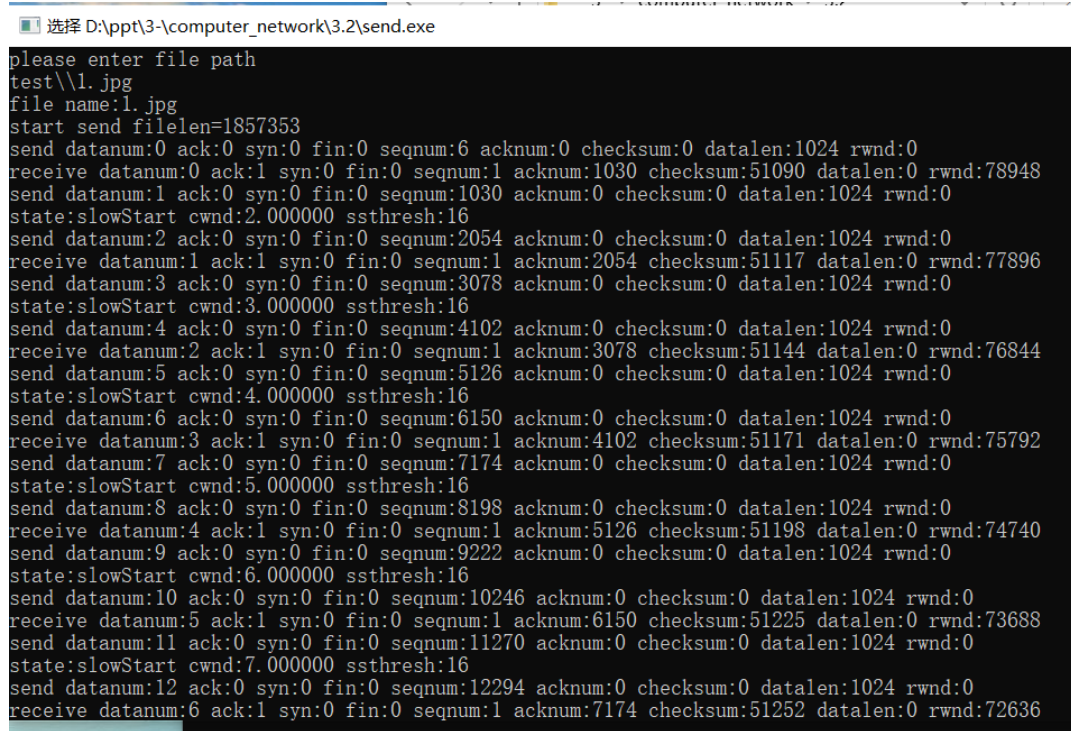
目录

1 概述	2
1.1 程序使用说明	4
2 原理介绍	4
2.1 差错重传机制的设计	4
2.2 动态流量控制的是实现	4
2.3 拥塞控制机制	5
3 具体实现	6
3.1 <i>reno</i> 算法实现	6
3.2 动态流量控制实现	8
3.3 差错重传机制实现	9
4 传输速率测量	9
5 实验中遇到的问题	11

1 概述

本次实验我在实验 3.1 的基础上进一步利用 *UDP* 实现了可靠数据传输，本程序实现了流量控制与 *reno* 算法实现的拥塞控制。

程序运行的最终效果图如下图所示：



```

选择 D:\ppt\3-\computer_network\3.2\send.exe
please enter file path
test\l.jpg
file name:l.jpg
start send filelen=1857353
send datanum:0 ack:0 syn:0 fin:0 seqnum:6 acknum:0 checksum:0 datalen:1024 rwnd:0
receive datanum:0 ack:1 syn:0 fin:0 seqnum:1 acknum:1030 checksum:51090 datalen:0 rwnd:78948
send datanum:1 ack:0 syn:0 fin:0 seqnum:1030 acknum:0 checksum:0 datalen:1024 rwnd:0
state:slowStart cwnd:2.000000 ssthresh:16
send datanum:2 ack:0 syn:0 fin:0 seqnum:2054 acknum:0 checksum:0 datalen:1024 rwnd:0
receive datanum:1 ack:1 syn:0 fin:0 seqnum:1 acknum:2054 checksum:51117 datalen:0 rwnd:77896
send datanum:3 ack:0 syn:0 fin:0 seqnum:3078 acknum:0 checksum:0 datalen:1024 rwnd:0
state:slowStart cwnd:3.000000 ssthresh:16
send datanum:4 ack:0 syn:0 fin:0 seqnum:4102 acknum:0 checksum:0 datalen:1024 rwnd:0
receive datanum:2 ack:1 syn:0 fin:0 seqnum:1 acknum:3078 checksum:51144 datalen:0 rwnd:76844
send datanum:5 ack:0 syn:0 fin:0 seqnum:5126 acknum:0 checksum:0 datalen:1024 rwnd:0
state:slowStart cwnd:4.000000 ssthresh:16
send datanum:6 ack:0 syn:0 fin:0 seqnum:6150 acknum:0 checksum:0 datalen:1024 rwnd:0
receive datanum:3 ack:1 syn:0 fin:0 seqnum:1 acknum:4102 checksum:51171 datalen:0 rwnd:75792
send datanum:7 ack:0 syn:0 fin:0 seqnum:7174 acknum:0 checksum:0 datalen:1024 rwnd:0
state:slowStart cwnd:5.000000 ssthresh:16
send datanum:8 ack:0 syn:0 fin:0 seqnum:8198 acknum:0 checksum:0 datalen:1024 rwnd:0
receive datanum:4 ack:1 syn:0 fin:0 seqnum:1 acknum:5126 checksum:51198 datalen:0 rwnd:74740
send datanum:9 ack:0 syn:0 fin:0 seqnum:9222 acknum:0 checksum:0 datalen:1024 rwnd:0
state:slowStart cwnd:6.000000 ssthresh:16
send datanum:10 ack:0 syn:0 fin:0 seqnum:10246 acknum:0 checksum:0 datalen:1024 rwnd:0
receive datanum:5 ack:1 syn:0 fin:0 seqnum:1 acknum:6150 checksum:51225 datalen:0 rwnd:73688
send datanum:11 ack:0 syn:0 fin:0 seqnum:11270 acknum:0 checksum:0 datalen:1024 rwnd:0
state:slowStart cwnd:7.000000 ssthresh:16
send datanum:12 ack:0 syn:0 fin:0 seqnum:12294 acknum:0 checksum:0 datalen:1024 rwnd:0
receive datanum:6 ack:1 syn:0 fin:0 seqnum:1 acknum:7174 checksum:51252 datalen:0 rwnd:72636
  
```

图 1.1: 发送端

未发生丢包时，程序可正常发送，当丢包事件触发计时器时，重发丢失的数据包，每次可能要进行状态迁移时输出当前状态，上图表示刚开始发送时程序处于慢启动状态。

```

D:\ppt\3-computer_network\3.2\receive.exe
datanum:492 ack:0 syn:0 fin:0 seqnum:503814 acknum:0 checksum:21747 datalen:1024 rwnd:0
datanum:493 ack:0 syn:0 fin:0 seqnum:504838 acknum:0 checksum:38953 datalen:1024 rwnd:0
datanum:494 ack:0 syn:0 fin:0 seqnum:505862 acknum:0 checksum:54855 datalen:1024 rwnd:0
datanum:495 ack:0 syn:0 fin:0 seqnum:506886 acknum:0 checksum:46510 datalen:1024 rwnd:0
datanum:496 ack:0 syn:0 fin:0 seqnum:507910 acknum:0 checksum:23023 datalen:1024 rwnd:0
datanum:497 ack:0 syn:0 fin:0 seqnum:508934 acknum:0 checksum:7905 datalen:1024 rwnd:0
datanum:498 ack:0 syn:0 fin:0 seqnum:509958 acknum:0 checksum:62616 datalen:1024 rwnd:0
datanum:499 ack:0 syn:0 fin:0 seqnum:510982 acknum:0 checksum:5151 datalen:1024 rwnd:0
unexpected datadatanum:501 ack:0 syn:0 fin:0 seqnum:513030 acknum:0 checksum:13782 datalen:1024 rwnd:0
unexpected datadatanum:502 ack:0 syn:0 fin:0 seqnum:514054 acknum:0 checksum:24344 datalen:1024 rwnd:0
unexpected datadatanum:503 ack:0 syn:0 fin:0 seqnum:515078 acknum:0 checksum:38566 datalen:1024 rwnd:0
unexpected datadatanum:504 ack:0 syn:0 fin:0 seqnum:516102 acknum:0 checksum:31827 datalen:1024 rwnd:0
unexpected datadatanum:505 ack:0 syn:0 fin:0 seqnum:517126 acknum:0 checksum:63903 datalen:1024 rwnd:0
unexpected datadatanum:506 ack:0 syn:0 fin:0 seqnum:518150 acknum:0 checksum:6448 datalen:1024 rwnd:0
unexpected datadatanum:507 ack:0 syn:0 fin:0 seqnum:519174 acknum:0 checksum:63247 datalen:1024 rwnd:0
datanum:500 ack:0 syn:0 fin:0 seqnum:512006 acknum:0 checksum:4784 datalen:1024 rwnd:0
duplicate data datanum:500 ack:0 syn:0 fin:0 seqnum:512006 acknum:0 checksum:4784 datalen:1024 rwnd:0
datanum:508 ack:0 syn:0 fin:0 seqnum:520198 acknum:0 checksum:2226 datalen:1024 rwnd:0
datanum:509 ack:0 syn:0 fin:0 seqnum:521222 acknum:0 checksum:2100 datalen:1024 rwnd:0
datanum:510 ack:0 syn:0 fin:0 seqnum:522246 acknum:0 checksum:41934 datalen:1024 rwnd:0
datanum:511 ack:0 syn:0 fin:0 seqnum:523270 acknum:0 checksum:49133 datalen:1024 rwnd:0
datanum:512 ack:0 syn:0 fin:0 seqnum:524294 acknum:0 checksum:26297 datalen:1024 rwnd:0
datanum:513 ack:0 syn:0 fin:0 seqnum:525318 acknum:0 checksum:54316 datalen:1024 rwnd:0
datanum:514 ack:0 syn:0 fin:0 seqnum:526342 acknum:0 checksum:1586 datalen:1024 rwnd:0
datanum:515 ack:0 syn:0 fin:0 seqnum:527366 acknum:0 checksum:31980 datalen:1024 rwnd:0
datanum:516 ack:0 syn:0 fin:0 seqnum:528390 acknum:0 checksum:59687 datalen:1024 rwnd:0
datanum:517 ack:0 syn:0 fin:0 seqnum:529414 acknum:0 checksum:56964 datalen:1024 rwnd:0
unexpected datadatanum:519 ack:0 syn:0 fin:0 seqnum:531462 acknum:0 checksum:46835 datalen:1024 rwnd:0
unexpected datadatanum:520 ack:0 syn:0 fin:0 seqnum:532486 acknum:0 checksum:43278 datalen:1024 rwnd:0
unexpected datadatanum:521 ack:0 syn:0 fin:0 seqnum:533510 acknum:0 checksum:10376 datalen:1024 rwnd:0

```

图 1.2: 接收端

接收端对于失序的数据包进行缓存，不进行丢弃。

一次文件发送完毕后会输出所传输的文件的大小，传输所需的时间与吞吐量等信息，并询问发送端是否继续发送下一个文件，如下图所示：

```

选择 D:\ppt\3-computer_network\3.2\send.exe
send datanum:1805 ack:0 syn:0 fin:0 seqnum:1848326 acknum:0 checksum:0 datalen:1024 rwnd:0
receive datanum:1803 ack:1 syn:0 fin:0 seqnum:1 acknum:1847302 checksum:39047 datalen:0 rwnd:77896
send datanum:1806 ack:0 syn:0 fin:0 seqnum:1849350 acknum:0 checksum:0 datalen:1024 rwnd:0
state:congestionAvoidance cwnd:4.339440 ssthresh:2
send datanum:1807 ack:0 syn:0 fin:0 seqnum:1850374 acknum:0 checksum:0 datalen:1024 rwnd:0
receive datanum:1804 ack:1 syn:0 fin:0 seqnum:1 acknum:1848326 checksum:39074 datalen:0 rwnd:76844
send datanum:1808 ack:0 syn:0 fin:0 seqnum:1851398 acknum:0 checksum:0 datalen:1024 rwnd:0
state:congestionAvoidance cwnd:4.569884 ssthresh:2
send datanum:1809 ack:0 syn:0 fin:0 seqnum:1852422 acknum:0 checksum:0 datalen:1024 rwnd:0
receive datanum:1805 ack:1 syn:0 fin:0 seqnum:1 acknum:1849350 checksum:39101 datalen:0 rwnd:75792
send datanum:1810 ack:0 syn:0 fin:0 seqnum:1853446 acknum:0 checksum:0 datalen:1024 rwnd:0
state:congestionAvoidance cwnd:4.788708 ssthresh:2
send datanum:1811 ack:0 syn:0 fin:0 seqnum:1854470 acknum:0 checksum:0 datalen:1024 rwnd:0
receive datanum:1806 ack:1 syn:0 fin:0 seqnum:1 acknum:1850374 checksum:39128 datalen:0 rwnd:74740
state:congestionAvoidance cwnd:4.997533 ssthresh:2
receive datanum:1807 ack:1 syn:0 fin:0 seqnum:1 acknum:1851398 checksum:39155 datalen:0 rwnd:73688
state:congestionAvoidance cwnd:5.197631 ssthresh:2
receive datanum:1808 ack:1 syn:0 fin:0 seqnum:1 acknum:1852422 checksum:39182 datalen:0 rwnd:72636
state:congestionAvoidance cwnd:5.390027 ssthresh:2
receive datanum:1809 ack:1 syn:0 fin:0 seqnum:1 acknum:1853446 checksum:31845 datalen:0 rwnd:78948
state:congestionAvoidance cwnd:5.575554 ssthresh:2
receive datanum:1810 ack:1 syn:0 fin:0 seqnum:1 acknum:1854470 checksum:31872 datalen:0 rwnd:77896
state:congestionAvoidance cwnd:5.754909 ssthresh:2
send datanum:1812 ack:0 syn:0 fin:0 seqnum:1855494 acknum:0 checksum:0 datalen:1024 rwnd:0
send datanum:1813 ack:0 syn:0 fin:0 seqnum:1856518 acknum:0 checksum:0 datalen:841 rwnd:0
receive datanum:1810 ack:1 syn:0 fin:0 seqnum:1 acknum:1854470 checksum:31872 datalen:0 rwnd:77896
file size:1857353 bytes
file transmission time:58466.000000 ms
throughput:0.242371 Mbit/s
send next picture?

```

图 1.3: 发送端

上图中也可以看到，发送完毕时程序处于拥塞避免状态。

1.1 程序使用说明

使用路由器小程序的各个参数如下图所示：

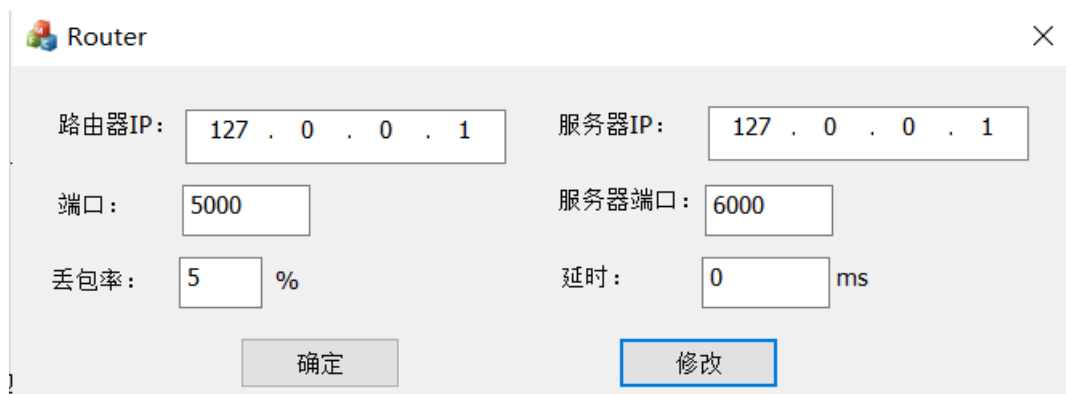


图 1.4: 路由器小程序

打开 *receive.exe* 与 *send.exe* 两个可执行程序后，在发送端中输入待发送的文件的路径，文件将会自动传输到接收端同级目录下的 *receive* 文件夹中。

2 原理介绍

2.1 差错重传机制的设计

在实验 3.1 中，我采用 *GBN* 协议来实现差错恢复机制，但在面对丢包问题 *GBN* 协议的效率会大打折扣，所以在本次实验中，我首先将差错重传机制改为了类似 *TCP* 所使用的差错重传机制，我将我实现的这个差错重传机制称为 *TCPP* 协议，具体内容如下：

TCPP 协议的确认仍然是累计式的，正确接收但失序的报文是会被接收端逐个确认的。*TCPP* 在发送方维护了已经发送但未被确认的字节的最小序号 *sendBase* 和下一个要发送的字节的序号 *nextSeqNum*，假设发送方要发送一组报文段 1, 2, 3, ..., *N*，但第 *n* ($n < N$) 个报文丢失，那么接收段将接收并确认所有序号小于 *n* 的报文段，对于 *n* + 1 到 *N* 之间所有的报文段，接收方将缓存这些报文段，但仅确认按序收到的最后一个报文的序号，也即 *n*。

2.2 动态流量控制的是实现

在实验 3.1 中滑动窗口的大小是固定的，发送端并不能根据接收端的反馈来动态调整发送窗口的大小，在本次实验中，我也仿照 *TCP* 流量控制机制实现了流量控制。

发送方始终维护一个接收窗口 *rwnd* 的变量来提供流量控制，也就是说，接收窗口用于给发送方一个指示——该接收方还有多少可用的缓存空间。

也即发送方始终维护如下不等式：

$$LastByteRcvd - LastByteRead \leq RcvBuffer$$

接收窗口的大小为：

$$rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]$$

其中 *LastByteRead* 在本程序中表示最后一个被写入磁盘中的数据段的序号。接收端也将 *rwnd* 封装进 *ack* 报文中，这样发送端就能获得相应的信息。

同时接收端维护如下不等式：

$$LastByteSent - LastByteAcked \leq rwnd$$

这样就可以满足流量控制要求了，同时，在本次实验中序列号的范围为 $[0 - \max \text{ int}]$ 远大于窗口大小的二倍，所以也不会出现二义性的问题。

2.3 拥塞控制机制

本实验中我实现了 *reno* 算法，*reno* 算法有如下三个阶段：

1. 慢启动：当 *cwnd* 的值小于 *ssthresh* 时，*TCP* 则处于慢启动阶段，每收到一个 *ACK*，*cwnd* 的值就会加 1。仔细分析，其实慢启动并不慢，经过一个 *RTT* 的时间，*cwnd* 的值就会变成原来的两倍，实为指数增长。

2. 拥塞避免：当 *cwnd* 的值超过 *ssthresh* 时，就会进入拥塞避免阶段，在该阶段下，*cwnd* 以线性方式增长，大约每经过一个 *RTT*，*cwnd* 的值就会加 1。

3. 快速恢复：当收到三个重复的 *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，但是并不会回到慢启动阶段，而是将 *cwnd* 设置为（更新后的）*ssthresh* + 3*MSS*，之后 *cwnd* 呈线性增长。

reno 算法的状态机如下图所示：

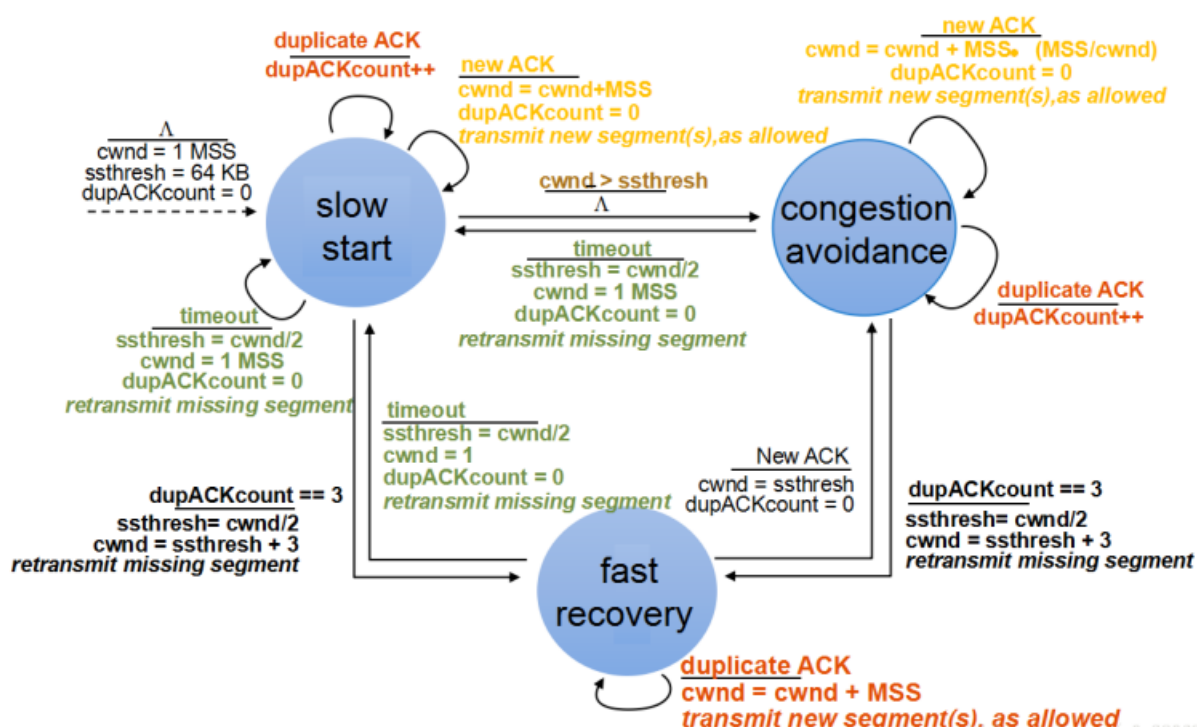


图 2.5: *reno* 算法

3 具体实现

3.1 *reno* 算法实现

实际中 *reno* 算法是很好实现的，只需按照其状态机编写其对应代码即可。根据状态机，主要有三种事件会导致状态迁移，对应代码如下：

```
switch(state)
{
    case slowStart:
        cwnd+=ackCount;
        if(cwnd>=ssthresh)
            state=fastRecovery;
        break;
    case congestionAvoidance:
        cwnd+=ackCount/cwnd;
        break;
    case fastRecovery:
        cwnd=ssthresh;
        state=congestionAvoidance;
        break;
    default:
        printf("wrong");
}
printstate();
duplicateAckCount=0;
```

图 3.6: 接收到正确 *ACK* 时的状态机

这里需要注意 *cwnd* 需设为 *float* 型变量，因为其在拥塞避免阶段增加的 *cwnd* 不是整数。


```
switch(state)
{
    case slowStart:
        ssthresh=int(cwnd)>>1;
        cwnd=ssthresh+3;
        state=fastRecovery;
        break;
    case congestionAvoidance:
        ssthresh=int(cwnd)>>1;
        cwnd=ssthresh+3;
        state=fastRecovery;
        break;
    case fastRecovery:
        break;
    default:
        printf("wrong");
}
```

图 3.7: 重复 ACK 数达到 3 时的状态机

重复 ACK 数达到 3 时都进入到快速恢复状态。

```
switch(state)
{
    case slowStart:
        ssthresh=int(cwnd)>>1;
        cwnd=1;
        break;
    case congestionAvoidance:
        ssthresh=int(cwnd)>>1;
        cwnd=1;
        state=slowStart;
        break;
    case fastRecovery:
        ssthresh=int(cwnd)>>1;
        cwnd=1;
        state=slowStart;
        break;
    default:
        printf("wrong");
}
```

图 3.8: 超时事件发生时的状态机

超时事件发生时都进入到慢启动状态。

3.2 动态流量控制实现

```
while(i<dataNum)
{
    int cwndSize=int(cwnd)*datagramLen;
    int temp=cwndSize<rwnd?cwndSize:rwnd;
    if(buffer[i].seqnum>=sendBase+temp)
    {
        Sleep(150);
        continue;
    }
    for(int j=0;j<cwnd&& i<dataNum;j++)
    {
        sd(i);
        printf("send ");
        showDataInfo(buffer[i++]);
    }
}
```

图 3.9: 发送端发送线程

在发送端发送时的发送窗口大小为 $\min(rwnd, cwnd)$ ，当满足发送窗口的条件时将 $cwnd$ 个数据包一并传输过去。

```
if(receiveData.seqnum==nextAck)
{
    memcpy(&buffer[nextReceiveNum],&receiveData,datagramLen);
    received[nextReceiveNum]=1;
    while(received[++nextReceiveNum]==1);
    datagram* d=&buffer[nextReceiveNum-1];
    nextAck=d->seqnum+d->dataLen;
    sendData.num=nextReceiveNum-1;
    sendData.rwnd=maxWindow-(nextReceiveNum-nextReadNum)*datagramLen;
    sendData.acknum=nextAck;sendData.ack=1;
    checkSum(sendData);
    if(receiveData.fin==1){
        sendData.fin=1;
        sd();
        break;
    }
    sd();
    showDataInfo(receiveData);
}
```

图 3.10: 接收端发送 ack 线程

如上图红线部分所示，接收方在每一个 ack 的数据段中都将其当前的 $rwnd$ 大小一并发送过去。

3.3 差错重传机制实现

2.1 节中提出的差错重传机制由如下代码实现：

```
if(receiveData.seqnum==nextAck)
{
    memcpy(&buffer[nextReceiveNum],&receiveData,datagramLen);
    received[nextReceiveNum]=1;
    while(received[++nextReceiveNum]==1);
    datagram* d=&buffer[nextReceiveNum-1];
    nextAck=d->seqnum+d->dataLen;
    sendData.num=nextReceiveNum-1;
    sendData.rwnd=maxWindow-(nextReceiveNum-nextReadNum)*datagramLen;
    sendData.acknum=nextAck;sendData.ack=1;
    checksum(sendData);
    if(receiveData.fin==1){
        sendData.fin=1;
        sd();
        break;
    }
    sd();
    showDataInfo(receiveData);
}
else if(receiveData.seqnum>nextAck)//&&receiveData.seqNum<nextReadNum+maxWindow
{
    received[receiveData.num]=1;
    memcpy(&buffer[receiveData.num],&receiveData,datagramLen);|
    sd();
    printf("unexpected data");
    showDataInfo(receiveData);
}
```

图 3.11: 接收端发送 *ack* 线程

当收到失序的数据段时缓存下来，当收到 *nextAck* 对应的包时则向右推动滑动窗口。

4 传输速率测量

为了在发送一个文件的过程中更好的体现拥塞控制的算法，我将 *MSS* 缩小为 1024byte (实验 3.1, 3.2 中为 4086byte)，这样在发送一个一个文件时需要发送的数据包的数量就是原来的四倍，在接下来的测试中，我将 3.1, 3.2 中的数据段大小都改为 1024byte 后进行测量。

首先测量传输速率随丢包率与路由器延时的变化情况进行了测量，并与实验 3.1 与 3.2 中设计的程序进行了对比，接下来的测试结果都是重复测量五次并取平均值的结果。

首先比较传输速率随丢包率的变化，在测量过程中路由器时延设为 0ms

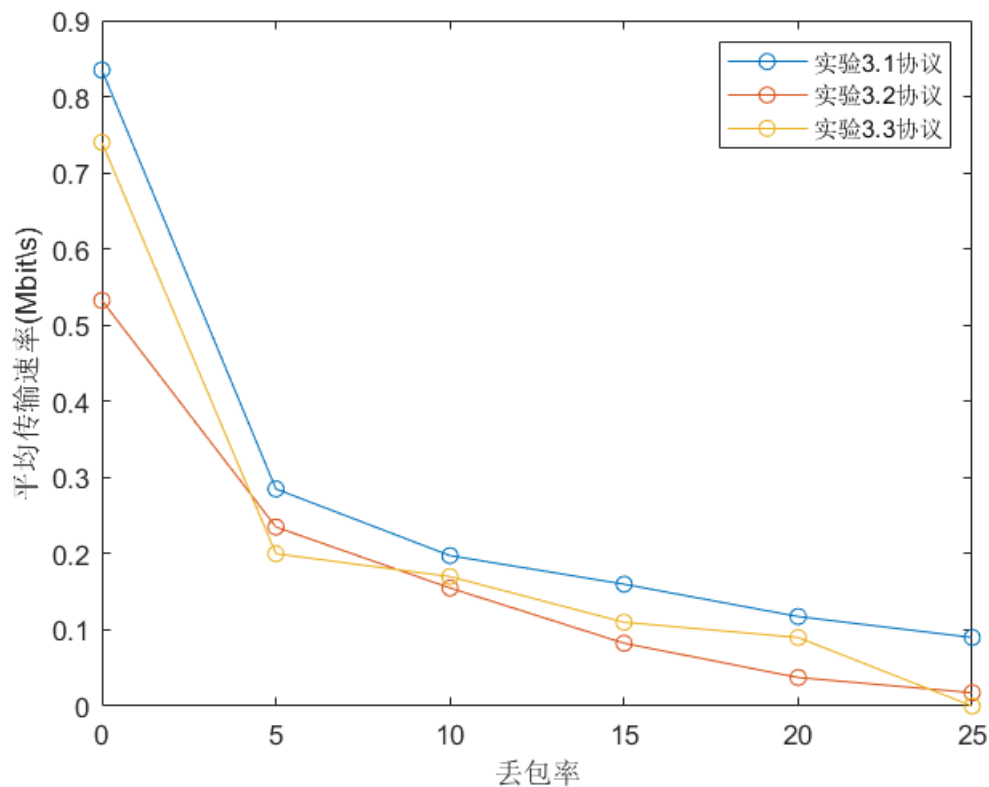


图 4.12: 传输速率随丢包率的变化

可以看到，当初始丢包率较小时，3.3 协议要快于 3.2 协议但略差于 3.1 的协议，这是因为 3.2, 3.3 的协议都有拥塞控制的功能，且 3.3 我在实现的时候对其做了一定优化，当丢包率逐渐增大时，三个协议的传输速率相差不大，我推测这是因为我在测试时未去掉 `printf` 的语句，而这一操作又是很费时的，导致三个协议的效率没有明显差别。同时注意到当丢包率到达 25% 时，也即每四个包丢弃一个包，3.3 协议传输速率降为 0，这是因为修改了 3.3 中的计时器机制，三次握手及发送的第一个数据包没有计时，导致前四个包若被丢弃则程序会直接停止。

接下来比较传输速率随路由器时延的变化，在测量过程中丢包率设为 0%

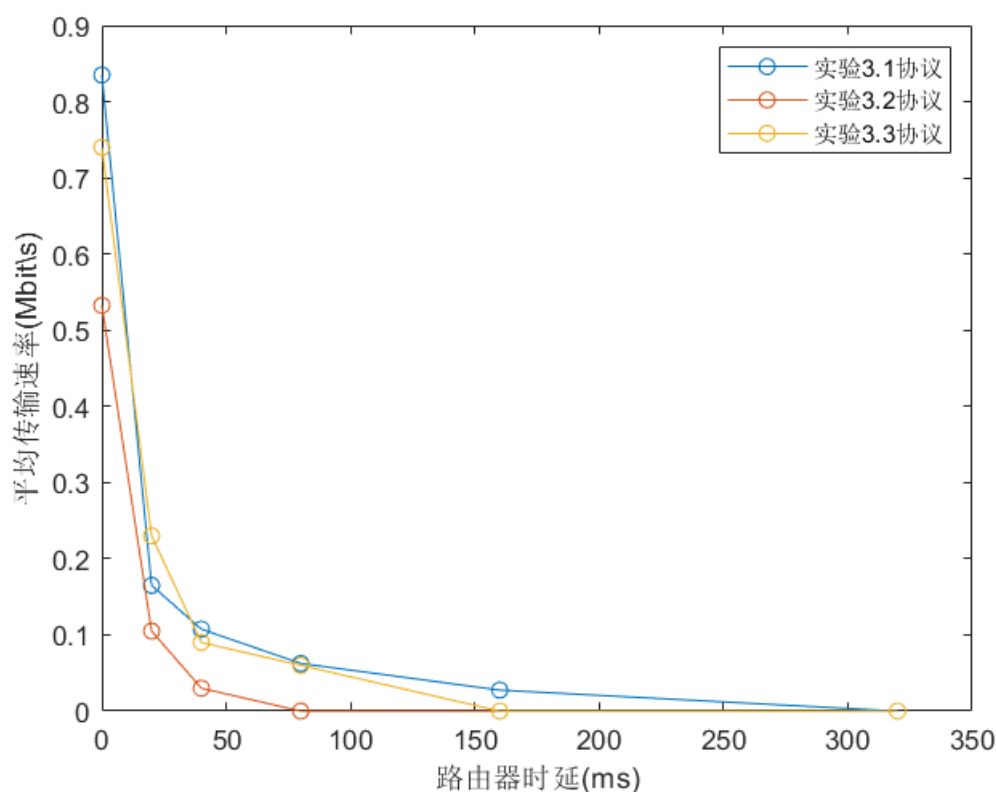


图 4.13: 传输速率随路由器时延的变化

可以看到路由器时延对三个协议的影响都非常大, 实际测试中我发现当时延大于 $150ms$ 时 3.2 与 3.3 协议传输速率已经让人无法接受, 所以我直接将上图中对应位置的传输速率设为 0。

5 实验中遇到的问题

多线程编程不易调试, 如果不仔细考虑线程间同步与竞争的情况, 很可能会出现意料之外的 *bug*, 同时出现的 *bug* 也不好复现, 很可能是因为增加了一些无关语句而改变了线程的同步情况而出现了 *bug*, 类似这样的问题很难找到其根本原因。本次实验过程中就遇到了这样的问题。

在最初的代码中, 我是按照如下思路来实现设立计时器这一功能的。

```
if(timerNum==0)
{
    timerNum++;
    timeoutData=i;

    CreateThread(NULL,NULL,&handleTimer,NULL,0,NULL);
}
```

图 5.14: 发送端负责发送的线程

```
if(i!=ackDataNum)
{
    timeoutData=ackDataNum;
    CreateThread(NULL,NULL,&handleTimer,NULL,0,NULL);
    timerNum++;
}
```

图 5.15: 发送端负责接收 ACK 的线程

在负责发送的线程中每发送一个数据包就判断当前是否有计时器，若无计时器，则为刚刚发送的数据包设立一个计时器，在负责接受的线程中每接收到一个新的 ACK 时判断当前是否还有已发送但未被 ACK 的数据包，若有则为此数据包设立一个计时器，若没有则不做任何事，图 5.1 中的发送线程在发送下一个数据包时检测当前没有计时器，然后为此数据包设立计时器。所以上述两段代码运行的理想情况应该是程序运行的任何时刻至多只有一个计时器，但我在实际测试的过程中却出现了许多由于发送线程与接收线程不同步导致的问题。

实际执行过程中可能会出现如下情况：两个线程同时通过了 $timerNum == 0$ 与 $i! = ackDataNum$ 则会创建两个计时器线程，如果仅是创建两个计时器，这当然是没有问题的，但是在每个计时器创建之前为 `timeoutData` 这个变量赋值的操作表示了当计时器超时重传时要发送哪个包，两个线程同时进入就会导致 `timeoutData` 的值不确定，最终导致错误。同时，由于这个 bug 是线程间不同步导致的，当我尝试 `printf` 一些中间信息时这个 bug 又会神奇的消失掉，导致我排查了很久才找到问题根源。

解决问题的方法也很简单，一种是加入信号量等同步机制，还有一种是修改设置计时器的原理，仅在一个线程中设置计时器。实际实现中，我发现由于程序中会频繁的创建和销毁计时器，采用信号量同步机制会大大拖垮程序的性能，所以我采用了第二种方式。修改后的代码如下所示：

```
//if(i!=ackDataNum)
//{
timeoutData=ackDataNum;

CreateThread(NULL,NULL,&handleTimer,NULL,0,NULL);
//timerNum++;
//}
```

图 5.16: 发送端负责接收 ACK 的线程

实际实现过程中我直接去掉了发送端设置计时器的代码，将设置计时器的功能全部放到了接收线程里，这样会有一些不足就是接收线程可能会为一些仍未发送的包设置计时器，但这样做不会产生不良的后果，只是接收方会接收到一个重复的包。

经过上述修改，代码顺利运行。