



南開大學
Nankai University

计算机学院和网络空间安全学院
计算机网络课程实验报告

实验 3-2：基于滑动窗口的流量控制机制

姓名：魏伯繁

学号：2011395

专业：信息安全

2022 年 12 月 3 日

目录

1 实验要求	2
1.1 实验要求	2
1.2 具体板块	2
2 实验理论及具体实现	2
2.1 建联断连	2
2.2 GBN 流量控制机制	2
2.3 SR 流量控制机制	6
3 实验效果	7
3.1 实验一	8
3.2 实验二	9
3.3 实验三	10
4 实验总结	10

1 实验要求

1.1 实验要求

在实验 3-1 的基础上，将停等机制改成基于滑动窗口的流量控制机制，采用固定窗口大小，支持累积确认，完成给定测试文件的传输。

1.2 具体板块

需要使用多个序列号
发送缓冲区、接受缓冲区；
滑动窗口：Go Back N；
有必要日志输出（须显示传输过程中发送端、接收端的窗口具体情况）。

2 实验理论及具体实现

2.1 建联断连

在本次实验中，我依然沿用了在实验 3-1 中使用的改进后的三次握手和四次挥手来作为具体的建联以及断联的具体实现。

具体内容在上一次实验报告中已经详细说明，在此不做过多赘述

2.2 GBN 流量控制机制

在实验 3-1 中我们实现的传输是一种基于停等机制的信息传输算法，停等机制的特点是实现简单，只需要两个序列号作为包的编号就可以完成停等机制的实现。但其缺点也十分明显，我们必须等待对前一个数据报的 ACK 返回才能发送下一个数据报。当然不可否认这样的信息传输方式在通信可能会存在丢包的情况下其可靠性可得到较大保证，但是在网络畅通时会极大程度上降低传输效率。

于是，对于停等机制的缺点，我们可以做出针对性的改进：不必等待上一个数据报的 ACK 返回就可以发出下一个数据报，这样就可以极大程度提高信息的收发效率。

但同样为了保证相应的传输可靠性，我们需要对这种传输方式进行约束，具体的约束包括：我们不能无限的发送数据报，只能在一个固定的数字内发送数据报，我们管这个数字叫做窗口，顾名思义，窗口的大小决定了能够发送的数据报的数量是什么，当然有了窗口的定义，我们就可以不用无限的增加序列号，而是可以对同一序列号进行反复利用。

我们将这样的数据传输方式成为 GBN，对于收到的 ACK，发送端只接受窗口最左端的 ACK，对于接收端，同样只接受窗口最左端的序列号的数据报，当不存在丢包时，GBN 将很好的提高传输效率。

如果发生丢包的情况，我们同样适用超时重传。为每一个窗口分组开始计时器，当我们每一次调整窗口时，重置计时器的值。当计时器超时的时候重传所有发出未确认的分组。

下面将展示具体的 GBN 要求：

- 允许发送端发出 N 个未得到确认的分组
- 需要增加序列号范围
- 分组首部中增加 k 位的序列号，序列号空间为 $[0, 2^k-1]$
- 采用累积确认，只确认连续正确接收分组的最大序列号

- 可能接收到重复的 ACK
- 发送端设置定时器，定时器超时，重传所有未确认的分组

下图具体展示了 GBN 的实现示意图：



图 2.1: GBN 示意图

为了为具体的编程提供一个清晰的思路，我们用状态机的方式体现 GBN 的实现过程，下图展示了发送端的状态机。

当收到了一个上层的调用消息时，需要首先检查现在是否还能够发送，也就是窗口的大小是否还足够，如果窗口的大小足够，那么就打包、发送，由于不涉及窗口大小的调整，所以不需要对时钟进行重置。如果窗口的大小不够，那么就需要将信息放在缓冲区中，并且检查现在的时钟是否超时，如果超时就重传所有发出未确认的数据报，反之则进入下一次循环。

同样，如果发送端收到了一个 ACK，则需要比较该 ACK 是否是窗口最左端的数据报所对应的 ACK，如果 ACK 对应则调整窗口大小，可以发送更多的数据报，反之则认为收到了错误的的数据报，不予以确认，等待下一个正确的 ACK，并且比较是否超时，如果超时进行超时重传。

■ GBN发送端扩展FSM

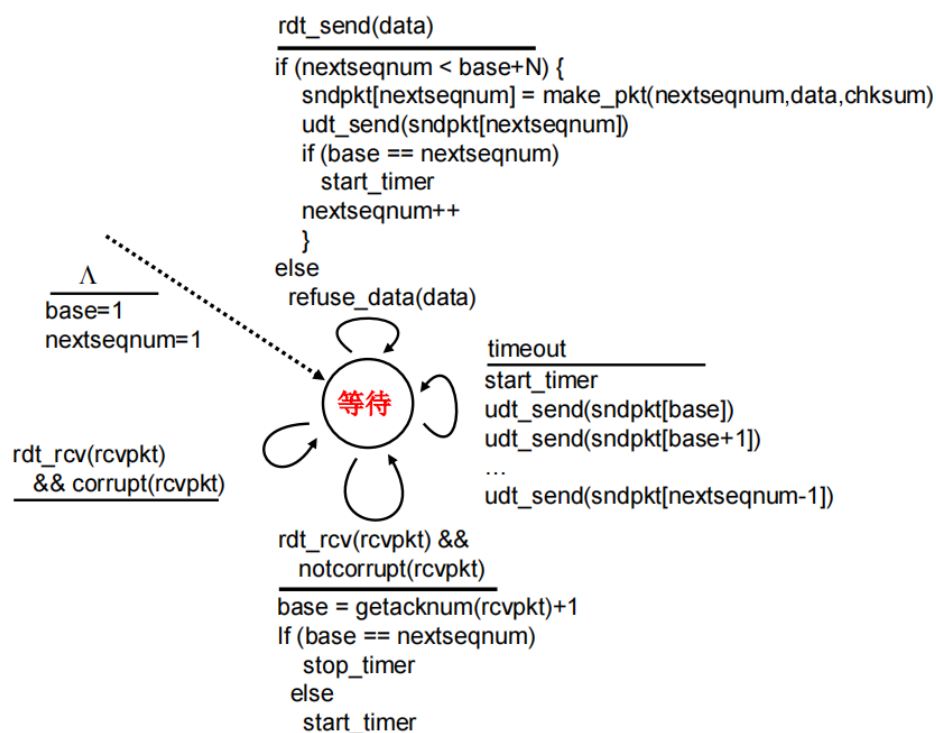


图 2.2: GBN 发送端状态机

下面给出正常完成单词传输的传输端代码（代码片段）

```

1  while (true) { //一直在循环
2      ioctlsocket(client, FIONBIO, &unlockmode); //设置为非阻塞
3      while (recvfrom(client, recvbuffer, sizeof(header), 0, (sockaddr*)&router_addr, &rlen) > 0)
4          Header temp;
5          memcpy(&temp, recvbuffer, sizeof(header));
6          if (temp.ack == sWindow && vericksum((u_short*)recvbuffer, sizeof(header)) == 0) { //收到的是
7              cout << "[ACK] 正确接受 ack 号为" << temp.ack << " 的 ack" << endl;
8              if (sWindow == SEQSIZE) { sWindow = 0; } //调整窗口, 开始位置的窗口从开始重新调整
9              else { sWindow++; }
10             if (eWindow + 1 > SEQSIZE) { //只有当 eWindow=8 时才会发生
11                 eWindow = 0;
12             }
13             else { //否则, 每次 eWindow 只需要 ++
14                 eWindow++;
15             }
16             cout << "[WINDOW] 目前滑动窗口的值为" << sWindow << "--" << eWindow << endl;
17             //因为收到的是我需要的, 所以这个时候一定可以转发一个包, 这个包是新发的, 所以要调整一系列
18             int ml; //本次数据传输长度
19             if (messagepointer > messagelength) { //不需要再发了, 都发完了
20                 if (endsend() == 1) { return 1; }
21                 return -1;
22             }
23             if (messagelength - messagepointer >= MAX_DATA_LENGTH) { //可以按照最大限度发送
24                 ml = MAX_DATA_LENGTH;
25             }
26             else {
27                 ml = messagelength - messagepointer + 1; //需要计算发送的长度
28             }
29             header.seq = nowpointer;
30             header.length = ml;
31             //header.checksum = calcksum((u_short*)&header, sizeof(header));
32             memset(sendbuffer, 0, sizeof(header) + MAX_DATA_LENGTH);
33             memcpy(sendbuffer, &header, sizeof(header));
34             memcpy(sendbuffer + sizeof(header), message + messagepointer, ml);
35             messagepointer += ml;
36             header.checksum = calcksum((u_short*)sendbuffer, sizeof(header) + MAX_DATA_LENGTH);
37             memcpy(sendbuffer, &header, sizeof(header));
38             if (nowpointer == SEQSIZE) { nowpointer = 0; }
39             else { nowpointer++; } //更新 nowpointer
40             //cout << vericksum((u_short*)sendbuffer, sizeof(header) + MAX_DATA_LENGTH) << endl;

```

```

41         sendto(client, sendbuffer, sizeof(header) + MAX_DATA_LENGTH, 0, (sockaddr*)&router_addr, 0);
42         cout << "[SEND] 已发送序列号为" << header.seq << " 的数据报" << endl;
43         startmap[header.seq] = messagepointer - ml; //如果要重发, 记录其相对于 message 的偏移
44         lengthmap[header.seq] = ml; //如果需要重发, 记录其长度
45         start = clock(); //发送成功, 做该序列号的计时
46     }
47     else { //收到的不是我需要的, 说明可能丢包发生了, 那我就重发在我想收到的最小的序列号
48         //这次转发不需要调整 messagepointer 和 windows 因为是重新发送
49         Header h;
50         h.seq = sWindow;
51         h.length = lengthmap[sWindow];
52         memset(sendbuffer, 0, sizeof(h) + MAX_DATA_LENGTH);
53         memcpy(sendbuffer, &h, sizeof(h));
54         memcpy(sendbuffer+sizeof(h), message + startmap[h.seq], h.length);
55         h.checksum = calcksum((u_short*)&sendbuffer, sizeof(h) + MAX_DATA_LENGTH);
56         memcpy(sendbuffer, &h, sizeof(h));
57         sendto(client, sendbuffer, sizeof(h) + MAX_DATA_LENGTH, 0, (sockaddr*)&router_addr, 0);
58         cout << "[ERROR] 接收 ack 失败, 正在重传数据报" << h.seq << endl;
59         start = clock();
60         break; //出去看看能不能发新的包
61     }
62 }
63

```

同理我们可以得到接收端的状态机, 接收端不断的等待发送端发送来的数据报, 并且比较其序列号与接收端窗口的序列号左值是否相等, 如果相等则确认接受并且转发相应的 ACK, 反之则丢弃该数据报, 不做其他处理, 等待发送端发送正确的数据报。

我们给出接收端的部分处理代码:

```

1
2 int receivemessage() {
3     Header header;
4     char* recvbuffer = new char[sizeof(header) + MAX_DATA_LENGTH];
5     char* sendbuffer = new char[sizeof(header)];
6     int nowpointer = 0; //下一个要接收的
7     while (true) { //反正就是一直接收
8         ioctlsocket(server, FIONBIO, &unblockmode); //设置为非阻塞
9         while (recvfrom(server, recvbuffer, sizeof(header) + MAX_DATA_LENGTH, 0, (sockaddr*)&router_addr, 0) > 0) {
10             memcpy(&header, recvbuffer, sizeof(header));
11             if (header.flag == OVER) { endreceive(); return 1; }
12             //cout << header.seq << " " << nowpointer << " " << vericksum((u_short*)recvbuffer,
13             if (header.seq == nowpointer && vericksum((u_short*)recvbuffer, sizeof(header) + MAX_DATA_LENGTH) == header.checksum) {

```

```

14     cout << "[ACK] 成功接收序列号为" << header.seq << " 的数据报, 正在发送 ack" << " 下
15     memcpy(message + messagepointer, rcvbuffer + sizeof(header), header.length); //拷贝
16     messagepointer += header.length; //重置位置指针
17     header.ack = nowpointer; //表示这个包我收到了
18     if (nowpointer == SEQSIZE) { nowpointer = 0; }
19     else { nowpointer++; } //下一个我想要收到的包的号
20     header.checksum = calcksum((u_short*)&header, sizeof(header));
21     memcpy(sendbuffer, &header, sizeof(header));
22     sendto(server, sendbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen); //发送
23     continue;
24 }
25
26 /*
27 if (nowpointer == 0) { header.seq = SEQSIZE; }
28 else { header.seq = nowpointer-1; } //发送上一个包
29 header.checksum = calcksum((u_short*)&header, sizeof(header));
30 memcpy(sendbuffer, &header, sizeof(header));
31 sendto(server, sendbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen); //发送
32 cout << " 不是期待的 ack, 已重发" << header.seq << endl;
33 */
34 cout << "[ERROR] 不是期待的 ack" << endl;
35 }
36

```

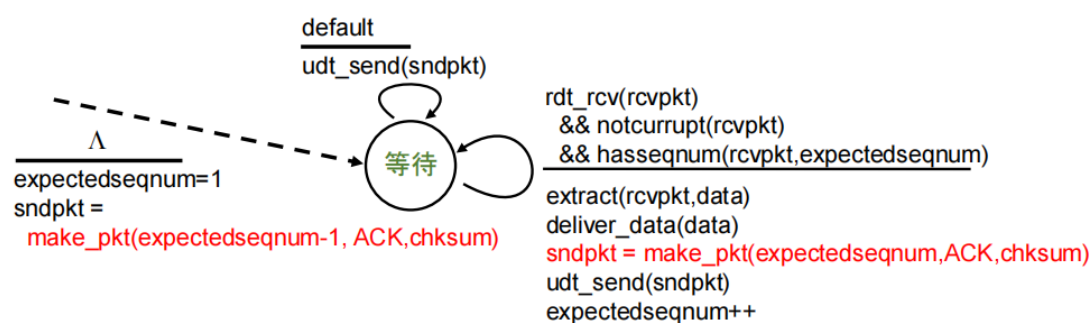


图 2.3: GBN 接受端状态机

2.3 SR 流量控制机制

当然, 我们还可以对 GBN 进行一些改进, 一个显而易见的改进方式就是对于接收端以及发送端, 都不必再对 ACK 以及数据报的序列号做强制的要求, 只要其在窗口内, 都可以相应的接受

当然, 对于超时冲床来说, 也不必重传所有的数据报, 而是只需要将在窗口内的所有未被确认的数据报进行重传即可, 具体的示意图如下图所示:

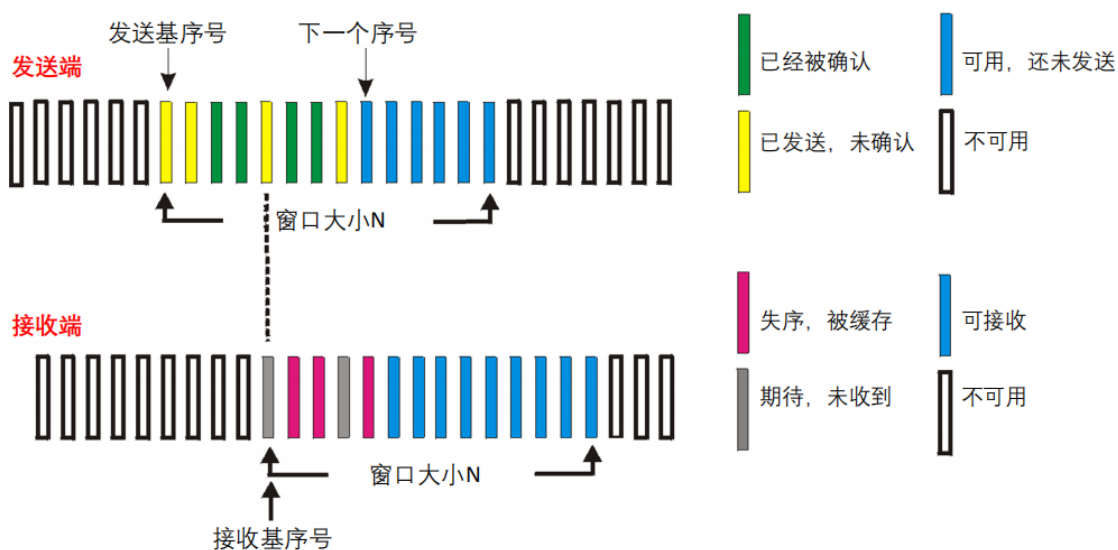


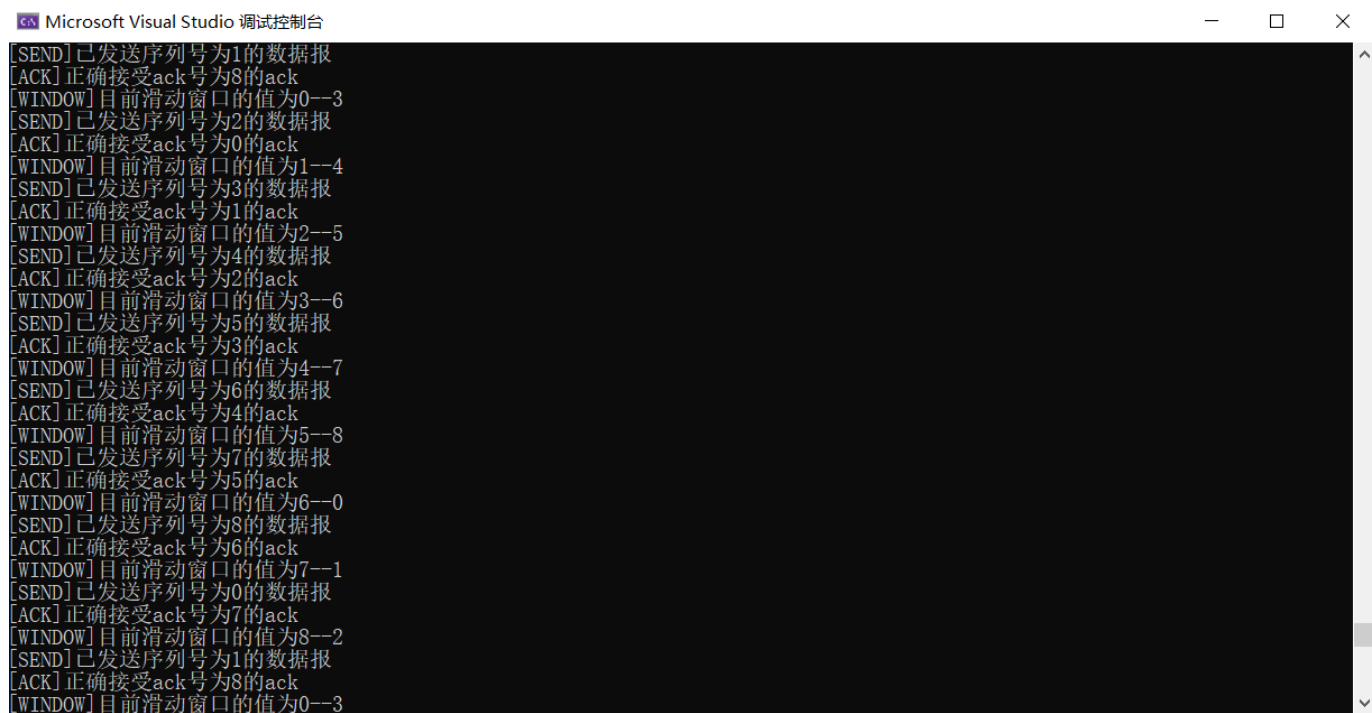
图 2.4: SR 示意图

我们将这种流量控制机制叫做选择重传，其对于发送端以及接收端具体的改进方式如下：

- 发送端接收上层数据：如果发送窗口中有可用的序号，则发送分组
- 发送端超时 (n)：重传分组 n，重启定时器
- 发送端接收 ACK(n)：n 在 $[\text{send_base}, \text{send_base} + N - 1]$ 区间，将分组 n 标记为已接收，如果是窗口中最小的未确认的分组，则窗口向前滑动，基序号为下一个未确认分组的序号
- 接收端 n 在 $[\text{rcv_base}, \text{rcv_base} + N - 1]$ 区间，发送 ACK(n)，缓存失序分组，按序到达的分组交付给上层，窗口向前滑动
- 接收端 n 在 $[\text{rcv_base} - N, \text{rcv_base} - 1]$ 区间，发送 ACK(n)

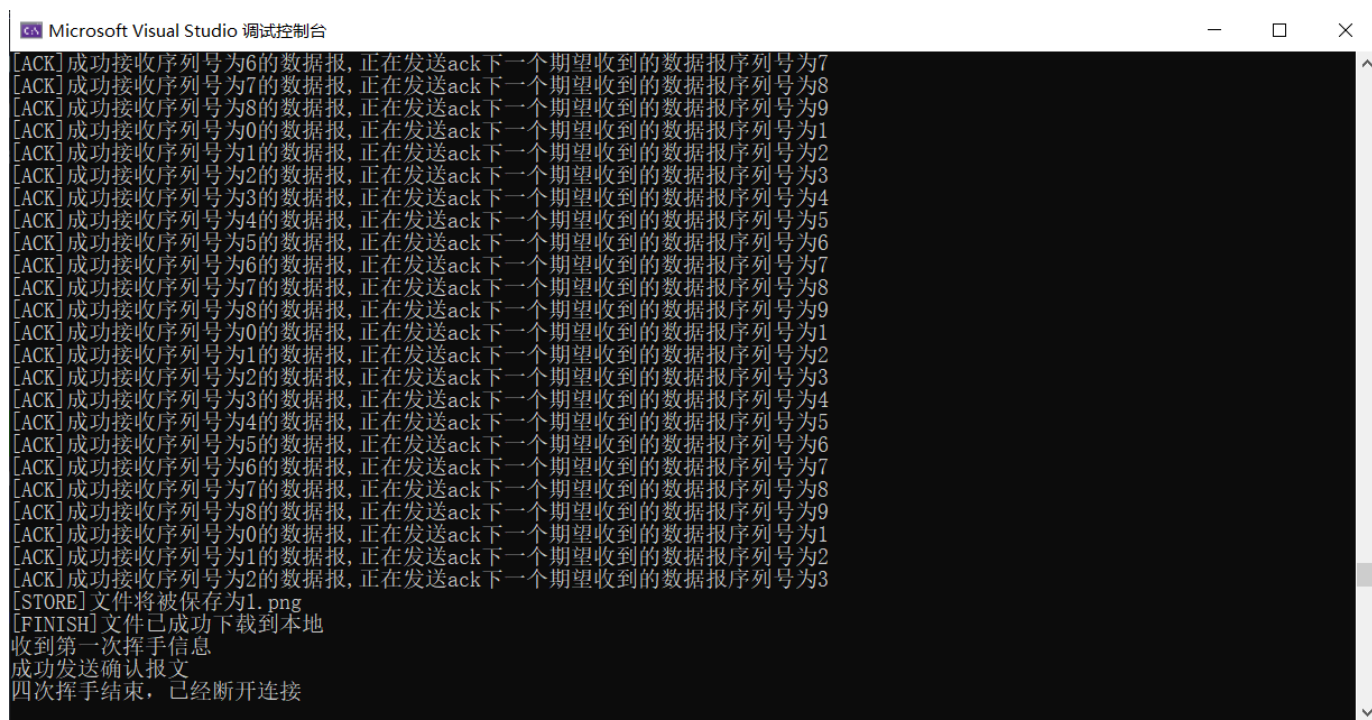
3 实验效果

具体的日志打印效果如下图所示：



```
Microsoft Visual Studio 调试控制台
[SEND] 已发送序列号为1的数据报
[ACK] 正确接受ack号为8的ack
[WINDOW] 目前滑动窗口的值为0--3
[SEND] 已发送序列号为2的数据报
[ACK] 正确接受ack号为0的ack
[WINDOW] 目前滑动窗口的值为1--4
[SEND] 已发送序列号为3的数据报
[ACK] 正确接受ack号为1的ack
[WINDOW] 目前滑动窗口的值为2--5
[SEND] 已发送序列号为4的数据报
[ACK] 正确接受ack号为2的ack
[WINDOW] 目前滑动窗口的值为3--6
[SEND] 已发送序列号为5的数据报
[ACK] 正确接受ack号为3的ack
[WINDOW] 目前滑动窗口的值为4--7
[SEND] 已发送序列号为6的数据报
[ACK] 正确接受ack号为4的ack
[WINDOW] 目前滑动窗口的值为5--8
[SEND] 已发送序列号为7的数据报
[ACK] 正确接受ack号为5的ack
[WINDOW] 目前滑动窗口的值为6--0
[SEND] 已发送序列号为8的数据报
[ACK] 正确接受ack号为6的ack
[WINDOW] 目前滑动窗口的值为7--1
[SEND] 已发送序列号为0的数据报
[ACK] 正确接受ack号为7的ack
[WINDOW] 目前滑动窗口的值为8--2
[SEND] 已发送序列号为1的数据报
[ACK] 正确接受ack号为8的ack
[WINDOW] 目前滑动窗口的值为0--3
```

图 3.5: 发送端日志



```
Microsoft Visual Studio 调试控制台
[ACK] 成功接收序列号为6的数据报, 正在发送ack下一个期望收到的数据报序列号为7
[ACK] 成功接收序列号为7的数据报, 正在发送ack下一个期望收到的数据报序列号为8
[ACK] 成功接收序列号为8的数据报, 正在发送ack下一个期望收到的数据报序列号为9
[ACK] 成功接收序列号为0的数据报, 正在发送ack下一个期望收到的数据报序列号为1
[ACK] 成功接收序列号为1的数据报, 正在发送ack下一个期望收到的数据报序列号为2
[ACK] 成功接收序列号为2的数据报, 正在发送ack下一个期望收到的数据报序列号为3
[ACK] 成功接收序列号为3的数据报, 正在发送ack下一个期望收到的数据报序列号为4
[ACK] 成功接收序列号为4的数据报, 正在发送ack下一个期望收到的数据报序列号为5
[ACK] 成功接收序列号为5的数据报, 正在发送ack下一个期望收到的数据报序列号为6
[ACK] 成功接收序列号为6的数据报, 正在发送ack下一个期望收到的数据报序列号为7
[ACK] 成功接收序列号为7的数据报, 正在发送ack下一个期望收到的数据报序列号为8
[ACK] 成功接收序列号为8的数据报, 正在发送ack下一个期望收到的数据报序列号为9
[ACK] 成功接收序列号为0的数据报, 正在发送ack下一个期望收到的数据报序列号为1
[ACK] 成功接收序列号为1的数据报, 正在发送ack下一个期望收到的数据报序列号为2
[ACK] 成功接收序列号为2的数据报, 正在发送ack下一个期望收到的数据报序列号为3
[ACK] 成功接收序列号为3的数据报, 正在发送ack下一个期望收到的数据报序列号为4
[ACK] 成功接收序列号为4的数据报, 正在发送ack下一个期望收到的数据报序列号为5
[ACK] 成功接收序列号为5的数据报, 正在发送ack下一个期望收到的数据报序列号为6
[ACK] 成功接收序列号为6的数据报, 正在发送ack下一个期望收到的数据报序列号为7
[ACK] 成功接收序列号为7的数据报, 正在发送ack下一个期望收到的数据报序列号为8
[ACK] 成功接收序列号为8的数据报, 正在发送ack下一个期望收到的数据报序列号为9
[ACK] 成功接收序列号为0的数据报, 正在发送ack下一个期望收到的数据报序列号为1
[ACK] 成功接收序列号为1的数据报, 正在发送ack下一个期望收到的数据报序列号为2
[ACK] 成功接收序列号为2的数据报, 正在发送ack下一个期望收到的数据报序列号为3
[STORE] 文件将被保存为1.png
[FINISH] 文件已成功下载到本地
收到第一次挥手信息
成功发送确认报文
四次挥手结束, 已经断开连接
```

图 3.6: 接收端日志

3.1 实验一

我们设置路由器, 并将路由器的丢包率设置为 0%, 时延为 0ms, 传输内容为测试文件的 1.jpg, 随后启动服务端和客户端进行传输, 传输后日志如下:

可以看到, 在实现 GBN 后, 在不丢包的情况下, 较之前停等机制的 18 秒有了较明显的效率提升。

```
-----传输日志-----
本次报文总长度为 1857353字节, 共分为 7256个报文段分别转发
本次传输的总时长为15秒
本次传输吞吐率为123824字节每秒
```

图 3.7: 传输日志 1

传输效果如下图所示, 表示程序成功完成了文件传输工作



图 3.8: 传输效果图

3.2 实验二

接下来, 我们设置路由器, 并将路由器的丢包率设置为 20%, 时延为 0ms, 传输内容为测试文件的 1.jpg, 随后启动服务端和客户端进行传输, 传输后日志如下:

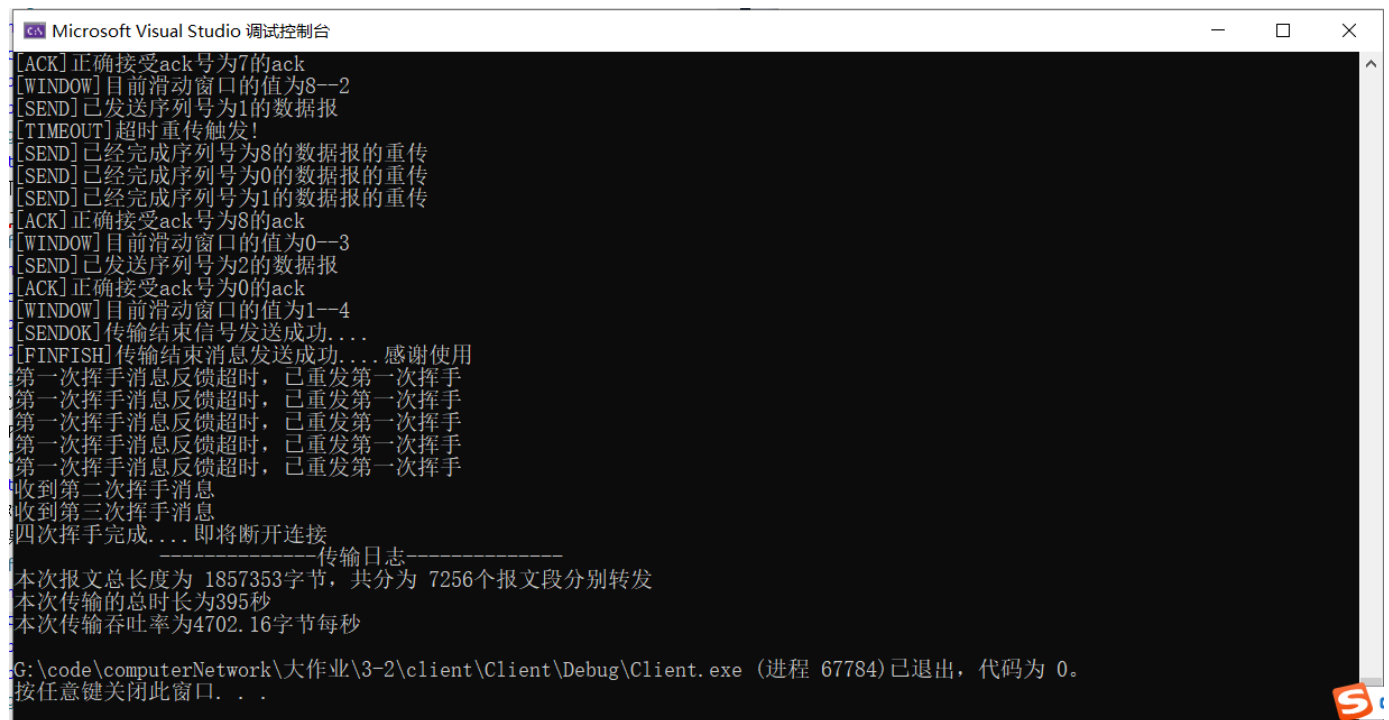
```
-----传输日志-----
本次报文总长度为 1857353字节, 共分为 7256个报文段分别转发
本次传输的总时长为854秒
本次传输吞吐率为2174.89字节每秒
```

图 3.9: 传输日志 2

可以看到, 当丢包率设置为 20% 时, 传输速率大大降低, 因为 GBN 需要等待超时, 并且一次重传分组内所有的数据报, 当然, 绝大部分的时间仍然是等待超时造成的, 本次实验设置的超时等待时间为 0.2 秒。

3.3 实验三

针对实验二，我们将超时等待时间设置为 0.07 秒，其余条件不变，再次进行实验。实验的日志输出结果如下：



```
Microsoft Visual Studio 调试控制台
[ACK] 正确接受ack号为7的ack
[WINDOW] 目前滑动窗口的值为8—2
[SEND] 已发送序列号为1的数据报
[TIMEOUT] 超时重传触发!
[SEND] 已经完成序列号为8的数据报的重传
[SEND] 已经完成序列号为0的数据报的重传
[SEND] 已经完成序列号为1的数据报的重传
[ACK] 正确接受ack号为8的ack
[WINDOW] 目前滑动窗口的值为0—3
[SEND] 已发送序列号为2的数据报
[ACK] 正确接受ack号为0的ack
[WINDOW] 目前滑动窗口的值为1—4
[SENDOK] 传输结束信号发送成功...
[FINISH] 传输结束消息发送成功... 感谢使用
第一次挥手消息反馈超时, 已重发第一次挥手
第一次挥手消息反馈超时, 已重发第一次挥手
第一次挥手消息反馈超时, 已重发第一次挥手
第一次挥手消息反馈超时, 已重发第一次挥手
第一次挥手消息反馈超时, 已重发第一次挥手
收到第二次挥手消息
收到第三次挥手消息
四次挥手完成... 即将断开连接
-----传输日志-----
本次报文总长度为 1857353 字节, 共分为 7256 个报文段分别转发
本次传输的总时长为 395 秒
本次传输吞吐率为 4702.16 字节每秒
G:\code\computerNetwork\大作业\3-2\client\Client\Debug\Client.exe (进程 67784) 已退出, 代码为 0。
按任意键关闭此窗口...
```

图 3.10: 传输日志 3

相比于同样重传时间的停等机制的 457 秒也有了接近 25% 的效率提升

4 实验总结

通过本次实验，我在上一次实验的基础上完成了流量控制的实现，进一步加深了对可靠传输的理解，将理论课上的知识转换为了现实。

同样，对于程序的测试不仅反映了我编写的程序的正确性，也通过不同的参数调整让我观察到了合适的流控机制对于传输速率的巨大提升，为下次拥塞控制的代码实现奠定了理论基础