lab3-2实验报告

姓名：吴静
学号：2113285
专业：信息安全

一、实验目的

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体制，发送窗口大于1（处于4-32之间），接收窗口等于1；支持累积确认，并且完成给定文件的传输。

1. 流水线协议：允许发送方在收到`ACK`之前连续发送多个分组
2. 滑动窗口：可以设置窗口大小，可以一直发送数据包直到窗口大小的数量的数据包；
3. 累积确认：如果收到了一个`ack`确认序号包，则默认收到了这个`ack`的数据包，以及以前所有的数据包；发送方为发送出去的数据包保留副本直至收到该数据包的确认

二、实验原理

1. 流水线协议

停等机制的弊端：在停等机制下，发送方在发送完一个数据包后直接进入等待模式，需要等待接收方返回的`ack`到达才能进行下一个包的传送，这大大降低了信道利用率。

流水线协议的缺点：需要更大的序列号范围，同时发送方和/或接收方需要更大的存储空间以缓存分组

2. 滑动窗口

滑动窗口：为了提高信道利用率，设置一个窗口大小，发送方可以持续发送窗口大小数量的数据包，即不用进入等待`ack`的阻塞状态，在发送完数据包后若没有收到`ack`的确认数据包，也不用卡死等待，而是继续发送数据包；

- 发送窗口大小大于1
- 接收窗口大小等于1

于是衍生出了两种基于滑动窗口的两个协议：

- 回退N帧协议：这是本次实验主要用上的协议，具体介绍在后文。
- 选择重传协议：这是下次实验主要用上的协议，具体在之后会解释。

滑动窗口的特点：随着协议的运行，窗口在序列号空间内向前滑动。

3. 累积确认

在发送的过程中，发送方可以一次性发送许多的数据包，接收方在收到数据包后返回对应的`ack`确认：

- 按序到达的数据包，返回对应`ack`
- 乱序到达的数据包，返回上一个确认的`ack`

注意：如果采用“不逐一确认”，则接收方不需要对每一个收到的数据包发送确认，而是每隔一段时间发送一个确认帧。

所以对于发送方来说，他可能一次性收到很多的`ack`，这里面的`ack`有按序到达的`ack`，有因为丢包乱序到达的`ack`，**发送方只确认连续正确接受分组的最大序列号。**如果这里面的`ack`将窗口第一个数据包确认，则窗口左边界右移；如果没有，且窗口大小已满，则需要等到超时，重传窗口内的所有未确认数据包。

4. 超时重传

上文说到，如果没有收到合适的`ack`确认，可能导致窗口内全是已发送未确认的数据包，这时候就要等待设置的定时器超时进行超时重传；另一方面，当在给定时间内如果没有收到正确的`ack`，也有可能引发超时。（引发上述两种情况中的哪一种超时就看网速以及超时时间的设置了）

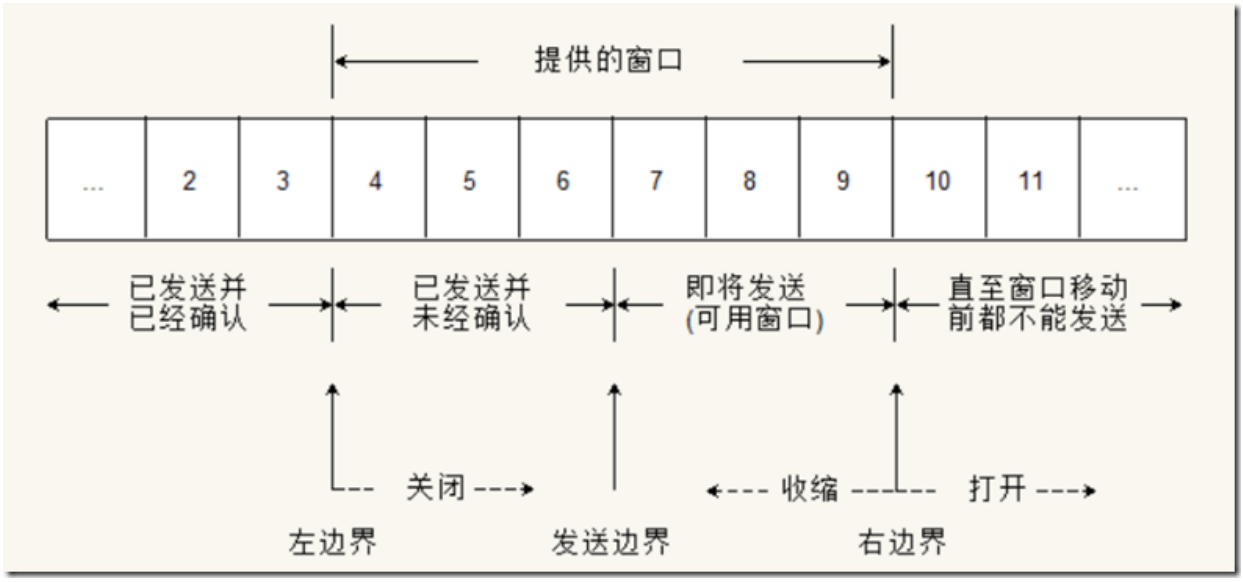
当引发了超时重传，这说明可能网络拥塞或者发生了丢包，此时会重传窗口内的所有已发送未确认的数据包，然后重新色湖之定时器。

5. 回退N帧协议`GBN`

回退N帧协议`GBN`，即Go-Back-N。

（1）窗口

窗口表示允许使用的序列号范围，窗口尺寸为N即表示最多有N个等待确认的消息：



如图所示，这里面总共有四种数据包：

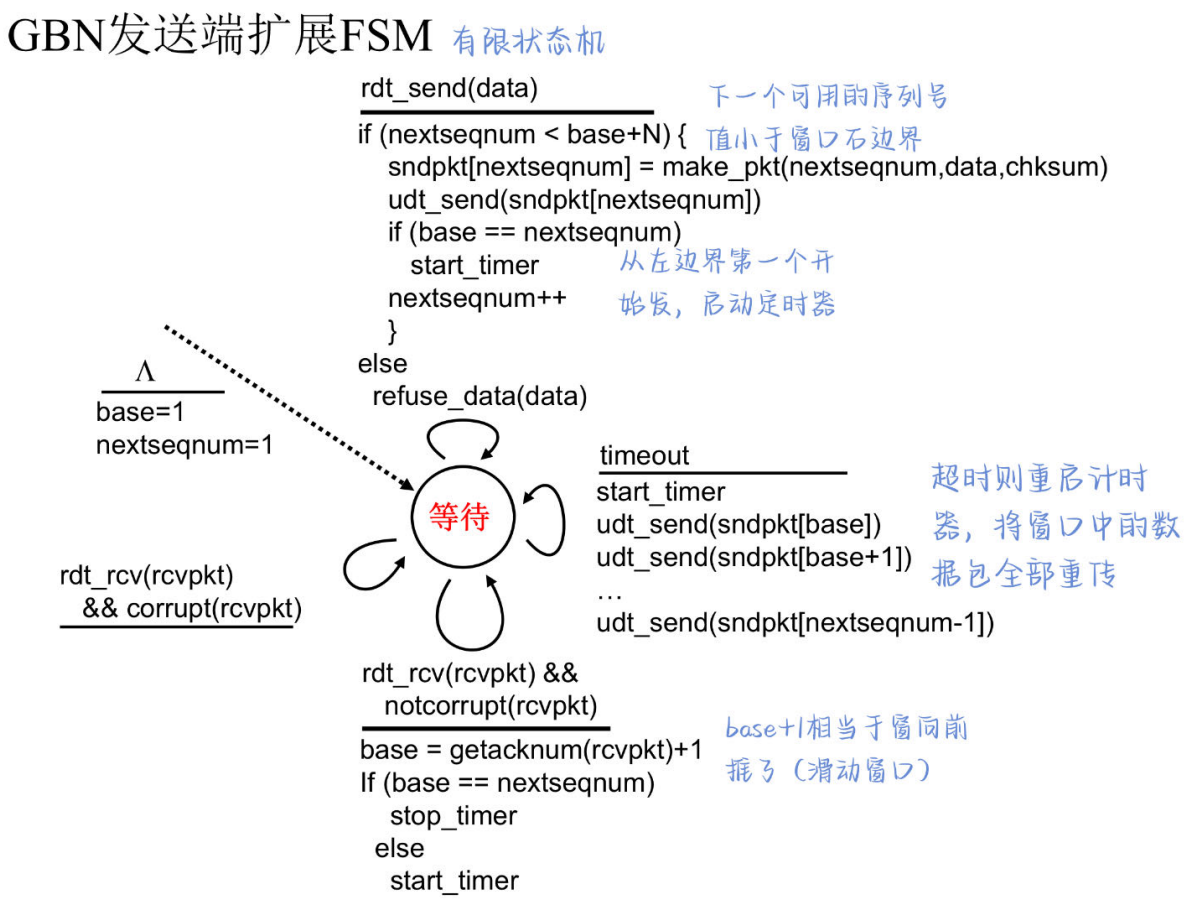
1. 已发送并已经确认的数据包：这一部分的数据包不存在于窗口中，不被缓存
2. 已发送并未经确认：这一部分的数据包存在于窗口中，需要等待`ack`包的确认
3. 即将发送的数据包：这一部分的数据包属于可发送数据包，在窗口大小内可供发送方发送的部分
4. 直至窗口移动前不能发送的数据包：在窗口外，无法进行发送

其中，左边界在接收到正确有序的`ack`确认后进行右移；右边界在发送数据包后进行右移。

左边界和右边界的移动是互相独立的，但是他们共同影响了窗口大小。

(2) GBN发送端

GBN发送端状态机如图：

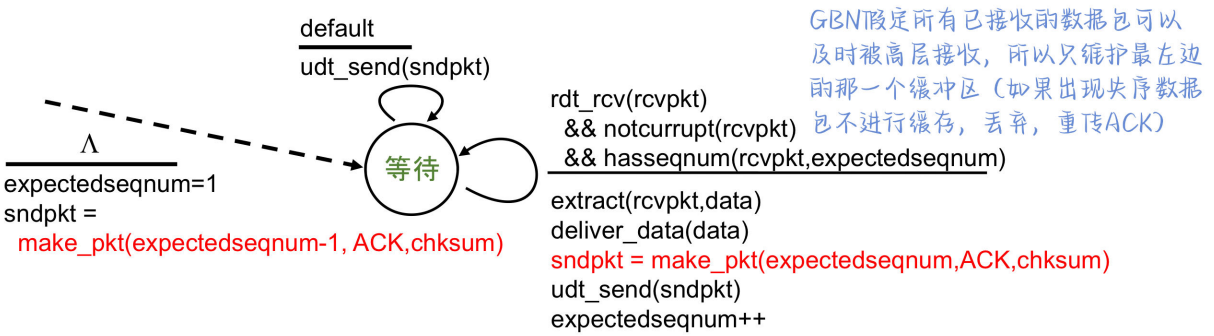


从图中可以得知几条信息：

1. 发送数据包的时间：`nextseqnum<base+N`，其中`N`表示窗口大小，`base`表示左边界，即基序号，`nextseqnum`表示下一个要发送的数据包，当窗口大小没有满的时候可以继续发送数据
2. 定时器设置的时间：当窗口开始发送第一个数据包的时候，即`base==nextseqnum`时，有两种情况：
    - 开始发送时
        - 第一次发送该窗口数据包
        - 超时后重新发送数据包
    - 收到`ack`时，由于窗口移动，所以需要重新计时，即重新设置定时器
3. 超时后，要重启计时器，将窗口中的数据包全部重传
4. 过程中仍然要进行校验和的检验和标志位的确认，这一部分我们在上一节中已经进行阐述，这里不再赘述
5. `base`的变化：在收到确认包后，要根据收到的确认包的`ack`的值进行进一步的确认和变化

(3) GBN接收端

GBN接收端状态机如图：



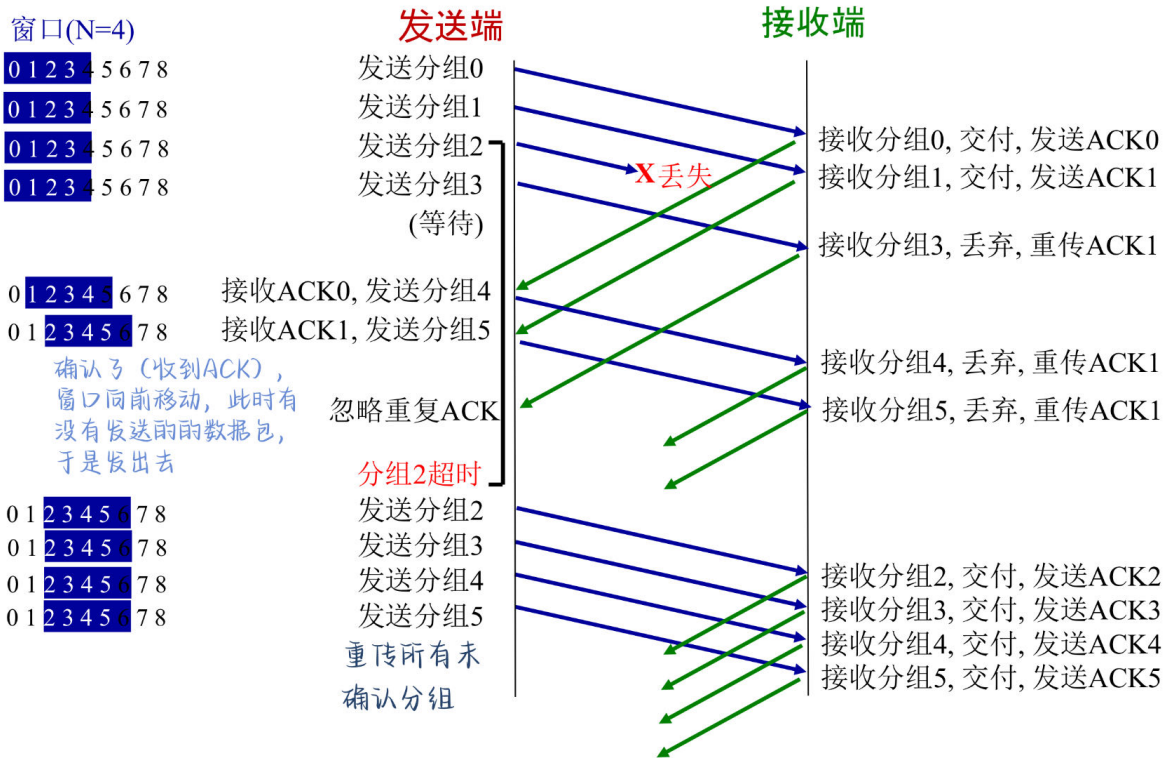
从图中可以得知几条信息：

1. 当收到有序到达的数据包时：发送正确ack，对数据包进行缓存
2. 当收到乱序到达的数据包时：直接丢弃，发送上一个确认的ack，同时不进行缓存；继续进行下一个包的确认和判断。所以有可能产生重复的ack

(4) 交互

数据交互部分，在窗口内如果有一个数据包没有正确送达而被网络丢掉，则会导致该数据包之后发送的数据包都是无效发送，因为得重新进行发送。所以他虽然相对于停等机制的信道利用率大大提高，但是降低了传输效率（停等协议一个包丢了只用重新发一个包，但是GBN的一个包丢了得重新发一个窗口的已发送未确认的包）。

交互数据图如下所示：



可以从图中验证我们以上通过状态机分析的结论。

三、协议设计

本次实验是在上一次实验——停等协议的基础上做的，所以这里只对改动的部分进行分析：

1. 传输单个数据包

由于不再是传输单个数据包后，就接受该数据包的返回的`ack`，于是这里去掉了发送数据包后等待`ack`返回以及进行超时检测的部分，修改后代码如下：

```
int send_package(SOCKET& client, SOCKADDR_IN& server_addr, int& serveraddr_len, char* data_content, int datasize, int& seq) {
    //传输单个数据包：每个数据包=头部+数据
    //printf("%d\n", (u_char)seq);
    Packet* sendpkt = new Packet();
    //要发送的内容，包括头和数据部分

    //初始化数据头：
    sendpkt->set_datasize(datasize);//初始化数据长度
    sendpkt->clear_sum();//对序列号进行清零
    sendpkt->set_ack((unsigned char)seq);//初始化序列号seq，注意此时seq是u_char类型
    //sendpkt->print_pkt();
    //此时数据部分不为0，所以要等数据部分初始化后再开始计算校验和

    //初始化数据：
    sendpkt->set_datacontent(data_content);

    //初始化数据头的校验和：
    sendpkt->set_sum(cksum((u_short*)sendpkt, sendpkt->get_size()));

    //检验发送数据包
    printTime();
    printf("检查数据包内容：\n");
    sendpkt->printpacketmessage();

    //发送数据包
    if (sendto(client, (char*)sendpkt, sendpkt->get_size(), 0, (SOCKADDR*)&server_addr, serveraddr_len) == -1) {
        printf("[Failed send]\nPacket\n\n");
        return -1;
    }
    printTime();
    printf("已成功发送数据包：");
    sendpkt->printpacketmessage();

    return 1;
}
```

该代码主要进行了两件事情：

- 完成了状态机中的`make_pkt()`，进行了数据包的包装
- 完成了状态集中的`udt_send()`，进行了数据包的发送

2. 发送文件

这一部分对应发送端状态机进行了改动，代码如下：

```
int send(SOCKET& client, SOCKADDR_IN& server_addr, int& serveraddr_len, char* data_content, int datasize) {
    //先算要发多少包：num=len/MAXSIZE+是否有余数（有余数就要多一个包）
    int package_num = datasize / MAXSIZE + (datasize % MAXSIZE == 0 ? 0 : 1);

    //确认号（序列号）从0开始
    //int seqnum = 0;

    printf("------即将开始发送当前文件------\n\n");

    //做好前期准备工作：
    int base = -1;
    //base指向被确认的最后一个数据包
    //base+1就是发送窗口的第一个数据包
    //(nextseqnum-(base+1)就是当前发送了但是没有被确认的数据包的数量
    int nextseqnum = 0;
    //nextseqnum指向即将发送的数据包
    clock_t start;
    //设置一个定时器

    while (base != package_num - 1) {
        //在base不等于最后一个包的时候，持续进行发送以及判断

        if ((nextseqnum - 1) - (base + 1) < Windows && nextseqnum != package_num) {
            //增加两个条件：一是发送位置不能等于包的数量，二是发送处和确认处的差值大小必须小于窗口大小（发送缓冲区）
            //base+1指向已发送但是没有被确认的第一个数据包

            //循环发送所有分开后的数据包
            printf("即将发送当前文件中的第%d号数据包：\n", nextseqnum);

            int len = 0;//每次要发送数据包的长度，前面都为MAXSIZE，最后一次发送剩下的
            if (nextseqnum == package_num - 1)
                len = datasize - (package_num - 1) * MAXSIZE;
            else
                len = MAXSIZE;

            //发送每一个数据包
            int seqnum = nextseqnum % 256;
            if (send_package(client, server_addr, serveraddr_len, data_content + nextseqnum * MAXSIZE, len, seqnum) == -1) {
                //如果发送失败的话
                printf("[send package Failed]\n\n");
                //重新发送该数据包
                nextseqnum--;
                continue;
```

```
            }
            printf("当前文件中的第%d号数据包发送成功! \n\n", nextseqnum);
            printf("packages:%d\n", package_num);
            printf("windows:%d\n", (nextseqnum - (base + 1)));
            //设置定时器：发送窗口第一个数据包的时候进行初始化
            if (nextseqnum == base + 1)
                start = clock();

            //如果发送成功，发送窗口右端也要向前移动
            nextseqnum++;
        }

        /*此时仍然默认为阻塞模式，需要设置为非阻塞模式*/
        u_long mode = 1;
        ioctlsocket(client, FIONBIO, &mode);

        Header header;
        char *recv_buffer=new char[sizeof(header)];
        //开始接收ACK

        if (recvfrom(client, recv_buffer, sizeof(header), 0, (SOCKADDR*)&server_addr, &serveraddr_len) != -1) {//这里一定要写上条件！因为if的判断只有0和非0!
            //如果收到了返回的数据包，首先要进行差错检测和ACK的确认
            //累积确认

            memcpy(&header, recv_buffer, sizeof(header));
            if (header.get_tag() == ACK && cksum((u_short*)&header, sizeof(header)) == 0) {
                //返回了确认包，首先要检查校验和之类的

                printf("发送的数据包已经被确认:\n");
                header.print_header();
                printf("base:%d,nextseqnum:%d\n\n", base, nextseqnum);

                //接下来，由于seq是从0-255而index不是，所以要进行相应转化
                if (int(header.get_ack()) == (base + 1) % 256)
                {
                    //base向前移动，移动的距离是header.get_ack()和(base + 1) % 256的差值加上1
                    base += (int(header.get_ack()) - (base + 1) % 256 + 1);

                    //重置定时器
                    start = clock();
                }

                //else {
                //  //返回的包不是正常我们需要的包，重传
                //  nextseqnum = base + 1;
                //}

            }
```

```
92
93          }
94          else {
95              if (clock() - start > MAX_TIME) {
96                  //超时
97                  printf("[timeout]\nresend Windows again……\n\n");
98                  //该窗口所有没有被确认的数据包均要进行重传，所以令nextseqnum等于base+1
99                  nextseqnum = base + 1;
100             }
101         }
102         delete []recv_buffer;
103         mode = 0;
104         ioctlsocket(client, FIONBIO, &mode);
105     }
106
107     clock_t now;
108     //for循环结束，发送数据包结束，接着开始发送结束标志：over
109     //初始化要发送的结束包：
110     Header header;
111     header.set_tag(OVER);
112     header.set_datasize((u_short)0);
113     header.set_ack((u_char)0);
114     header.clear_sum();
115     header.set_sum(cksum((u_short*)&header, sizeof(header)));
116
117     //初始化要发送的数据：
118     char* send_buffer = new char[sizeof(header)];
119     memcpy(send_buffer, &header, sizeof(header));
120     if (sendto(client, send_buffer, sizeof(header), 0,
(SOCKADDR*)&server_addr, serveraddr_len) == -1) {
121         printf("[Failed send]\nOVER\n\n");
122         return -1;
123     }
124     printTime();
125     printf("[send]\nOVER\n\n");
126
127     //存储当前时间
128     now = clock();
129
130     u_long mode = 1;
131     ioctlsocket(client, FIONBIO, &mode);
132
133     char* recv_buffer = new char[sizeof(header)];
134
135     while (true)
136     {
137         while (recvfrom(client, recv_buffer, sizeof(header), 0,
(SOCKADDR*)&server_addr, &serveraddr_len) <= 0) {
138             if (clock() - now > MAX_TIME) {
139                 printf("[timeout]\nresend OVER again……\n\n");
140                 if (sendto(client, send_buffer, sizeof(header), 0,
(SOCKADDR*)&server_addr, serveraddr_len) == -1) {
```

```
141                     printf("[Failed send]\nOVER\n\n");
142                 }
143                 now = clock();
144             }
145         }
146 
147         mode = 0;
148         ioctlsocket(client, FIONBIO, &mode);
149 
150         memcpy(&header, recv_buffer, sizeof(header));
151         if (header.get_tag() == OVER && cksum((u_short*)&header,
    sizeof(header)) == 0) {
152             printTime();
153             printf("[recv]\nOVER\n\n");
154             printf("------对方已接受到文件------\n\n");
155             break;
156         }
157         else
158             continue;
159     }
160     return 1;
161 }
```

首先对大概的思路进行说明：

- 首先注意代码中的`base`和状态机中的`base`表示的意思不对等：
  - 代码中的`base`表示已确认的数据包的最后一个，所以初始化为0；
  - 状态机的`base`表示基序号，实际应该初始化为0；
  - 代码中的`base`和状态机中的`base`的关系应该是`（状态机）base=（代码）base+1`
- 首先进行一个`while`循环，如果`base`不等于要发送文件的最后一个数据包，则此时没有确认到要发送文件的最后一个数据，说明文件还没有发送完成，要不断进行`while`循环
- `while`循环中首先要进行一个判断，`(nextseqnum - 1) - (base + 1) < Windows && nextseqnum != package_num`；前者表示窗口大小还没有满，还可以继续发送，后者表示还存在需要发送的数据包
  - `(nextseqnum - 1)`表示最后一个已发送的数据包
  - `(base+1)`表示想要确认的第一个数据包

  如果两个条件都满足，则可以继续发送数据包。
- 发送完数据包，如果这是窗口中的第一个数据包，则要启动定时器，即`nextseqnum == base + 1`时启动定时器；然后将要发送的数据包需要加1，即`nextseqnum+1`
- 然后检查是否有`ack`返回：**注意这里必须设置为非阻塞模式，否则将变为停等协议！**当有`ack`返回时，首先仍然按惯例检查标志位和校验和
- 如果都正确了，则判断返回的确认号是不是我们所需要的
  - 如果是我们所需要的序列号，则将`base`右移，同时由于此时是一个新的窗口，所以还要重新设置定时器
  - 如果不是我们需要的序列号，这里有两种解决措施：

    - 不管他，继续进行下面的步骤
    - 考虑到接收到错误的`ack`肯定是因为中途发生了丢包，所以要及时止损，停止发送当前即将发送的数据包，而是直接开始重新发送该窗口中未确认的数据包

  这里我选择了第一种算法，这可能会导致后面的包全部进行无效发送而降低传输效率，但是由于更能显示出超时重传的过程，所以这里没有选择第二种效率更高的方式。

- 如果没有`ack`返回，则继续进行操作
- 首先仍然是一个超时检测，如果当前时间已经超时，那么将`nextseqnum`赋值为`base+1`进行超时重传；如果当前时间没有超时，则进行下一轮循环

3.丢包和延时代码

为了方便调试，本次实验自己写了丢包和延时的代码，主要加在了服务器端：

丢包

利用随机数，当产生的随机数小于设置的丢包率的时候进行丢包，使丢包率趋于我们所设置的丢包率，代码如下：

```
        //设置丢包
        if ((rand() % 100) + 1 < random)
        {
            printf("该包不做存储，进行人工丢弃\n\n");
            continue;
        }
```

延时

每一个包到达后先睡眠一定时间再进行操作，代码如下：

```
        printf("延时%dmseconds\n\n", delay);
        Sleep(delay);
```

delay即为设置的延时毫秒数。

4. 单线程模拟多线程效果

在了解了实验原理后，意识到这大概是需要使用多线程实现：

（一个思路：）

- 一个线程利用`nextseqnum`进行数据包的发送
- 一个线程持续不断地接收对方传过来的`ack`并进行判断，同时影响`base`的值
- 两个线程共同影响窗口大小的值并进行协同运行

但是本次实验我仍然是使用单线程进行操作的，所以我在这里进行解释说明：

窗口

多线程：持续发送数据包直到窗口数量大小

单线程：在将套接字设置为非阻塞模式后，`recvfrom`的返回很快，如果没有收到消息立刻返回-1，所以在没有收到`ack`的情况下，发送端持续不断地发送数据包，达到“发送窗口大小的数据包”的效果

滑动窗口

多线程：一个线程一边发送数据包，窗口右边界右移；一个线程持续接收`ack`，窗口边界右移

单线程：没有`ack`返回的情况下，持续发送数据包，窗口有边界右移；有`ack`返回的情况下，会对`ack`返回值进行判断，如果是我们需要的`ack`就将窗口右移；如果不是需要的`ack`，则会不做操作继续发送数据包。

由于判断返回`ack`的值的过程相比于发送数据包和网络延时的时间非常短，几乎可以忽略不计，所以达到了“在发送数据包的同时进行了`ack`的接收”的模拟效果。

累积确认

由于网络延时和丢包等一系列外界原因，返回的`ack`包可能会堆积在发送端，达到“累积确认”的效果

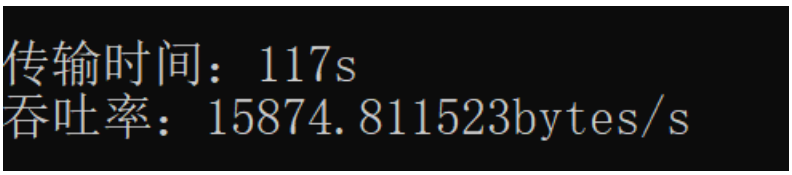
四、实验结果

1.四个文件的传输验证

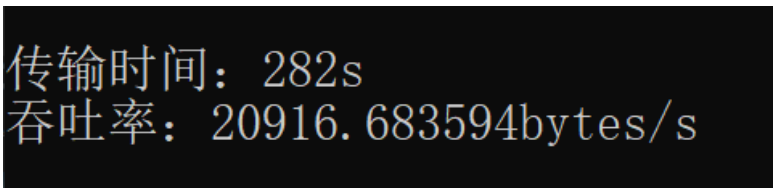
条件：丢包率为5%，延时10ms，发送窗口大小为20

四个文件传输结果如下：

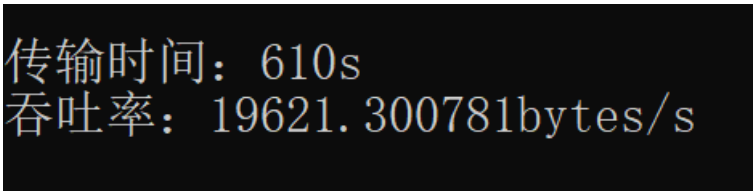
- `1.jpg`



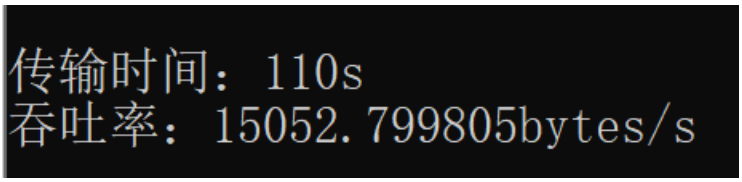
- `2.jpg`



- `3.jpg`



- `helloworld.txt`



传输结果：

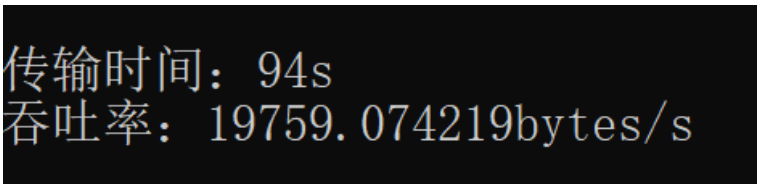
名称	修改日期	类型	大小
x64	2023/11/27 16:56	文件夹	
1.jpg	2023/11/29 18:56	JPG 文件	1,814 KB
2.jpg	2023/11/29 19:02	JPG 文件	5,761 KB
3.jpg	2023/11/29 18:53	JPG 文件	11,689 KB
helloworld.txt	2023/11/29 19:04	文本文档	1,617 KB
Server.cpp	2023/11/29 14:04	C++ Source	16 KB
Server.vcxproj	2023/11/26 19:23	VC++ Project	7 KB
Server.vcxproj.filters	2023/11/27 10:12	VC++ Project Filter...	1 KB
Server.vcxproj.user	2023/11/26 19:23	Per-User Project O...	1 KB

可以发现，图片和文本文件均传输成功，实验正确。

2.不同窗口大小的传输差别

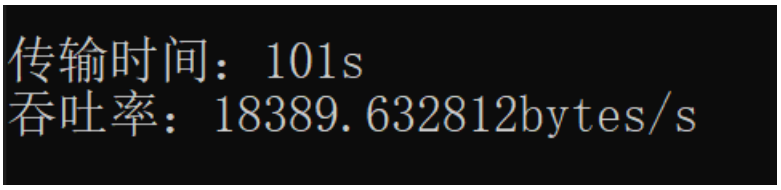
以`1.jpg`为例，将分别对窗口大小为4，8，16，20，32为例进行传输验证来观察差别，结果如下：

- 窗口大小为4：

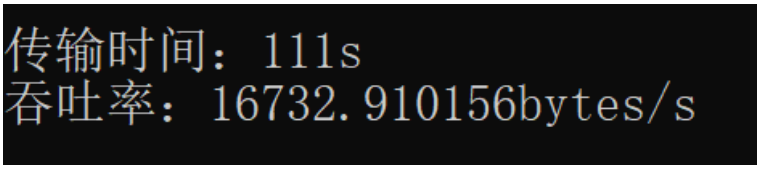


窗口大小为4时命令行经常有明显停顿，这是因为窗口大小为4时很容易满，在丢包时只能等待超时进行重传。

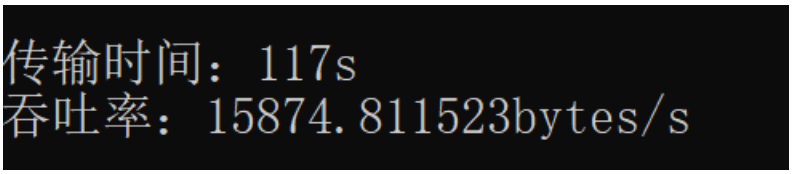
- 窗口大小为8：



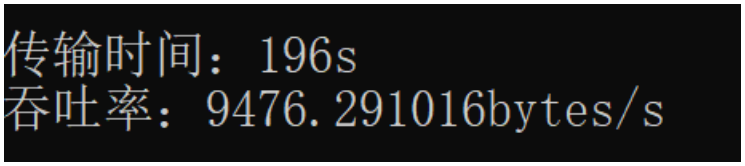
- 窗口大小为16：



- 窗口大小为20：



- 窗口大小为32：



总结如下：

窗口大小	传输时间（s）	吞吐率（bytes/s）
4	94	19759.074219
8	101	18389.632812
16	111	16732.910156
20	117	15874.811523
32	196	9476.291016

可以看见，在其他条件一致的情况下（丢包=5%，延迟=10ms），随着窗口大小的增加，传输时间逐渐增加，吞吐率逐渐降低，推测有几个原因：

- 在丢一个包后，该窗口后的所有包再传过去就已经是无用功了，属于是浪费时间的白传，在传过去之后再等待超时重传自然时间会更长
- 窗口大了，该窗口内部丢包的可能性更大，触发超时重传的概率也更大
- 本次实验属于`GBN`，而不是`SR`（选择重传），所以在丢包后要重传后面所有未确认的包而不是只重传丢失的包，所以时间也会长一点

3.丢包延时和乱序数据包示例

```
该包不做存储，进行人工丢弃

延时10seconds

seq_predict:22
[2023-11-29 19:18:57]
该数据包非按序数据包，不进行存储：
Packet size=1030 bytes, tag=0, seq=19, sum=33471, datasize=1024
[2023-11-29 19:18:57]
已发送确认：
datasize:0, ack:21, tag:2
```

4.累积确认部分代码示例

```
发送的数据包已经被确认:
datasize:0，ack:255
base:1791,nextseqnum:1814

发送的数据包已经被确认:
datasize:0，ack:0
base:1791,nextseqnum:1814

发送的数据包已经被确认:
datasize:0，ack:1
base:1792,nextseqnum:1814

发送的数据包已经被确认:
datasize:0，ack:2
base:1793,nextseqnum:1814

发送的数据包已经被确认:
datasize:0，ack:3
base:1794,nextseqnum:1814

发送的数据包已经被确认:
datasize:0，ack:4
base:1795,nextseqnum:1814
```

五、总结

通过这次实验，我了解了流水线协议，懂得了滑动窗口，学会编写`GBN`相关代码，了解并运用了超时重传，累积确认，对于可靠传输协议的知识了解得更加透彻。