学号：2111252 姓名：李佳豪

代码链接:https://github.com/FondH/cn/tree/StableUdp-Part2

UDP可靠传输-Part02

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，发送窗口和接收窗口采用相同大小，支持累积确认，完成给定测试文件的传输。

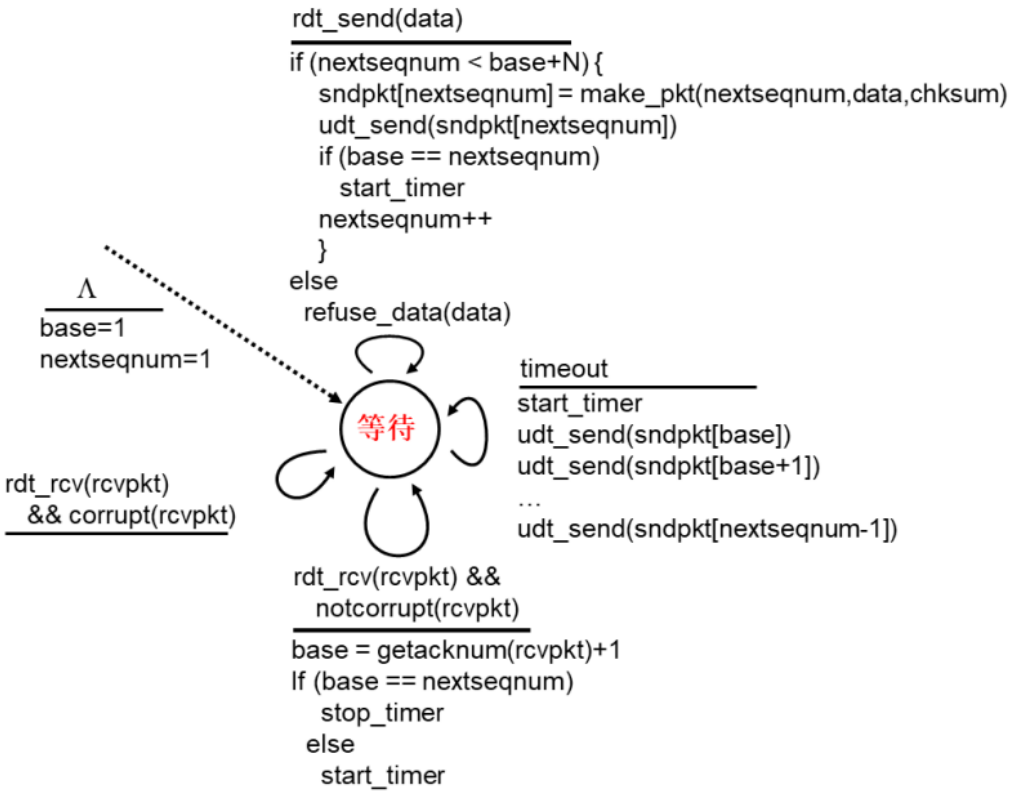
GBN实验原理要点

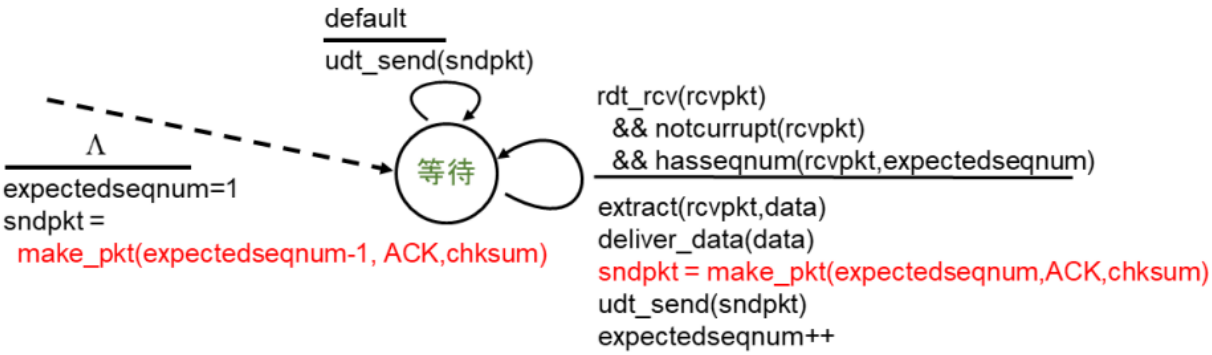
1. **滑动窗口机制**：GBN使用固定大小的发送窗口来控制发送的数据量。发送窗口内的所有数据包可以在没有收到确认的情况下连续发送，而**发送端**要对每一个每个数据包编写**唯一的序号**，用于接收方识别和确认，将“窗口”内的所有数据包发送
2. **确认和超时**：**接收方**收到数据包后，发送确认（ACK）回复，而接受端的发送的每一个ack值一定是自己希望获得下一个数据包的seq值，保证了接受数据包的**顺序可靠性。**如果**发送方**在设定的超时时间内未收到ACK，它假定该数据包已丢失。
3. **重传机制**：在超时后，发送方不仅重传丢失的数据包，而且还重传该数据包之后的所有数据包（即窗口内的剩余数据包）。
4. 握手与挥手：和上一次实验一样为了保证**传输通道的可靠**，增加一个**三次握手状态机**；而**二次挥手**嵌入在文件传输的结尾，在发送端发送到最后一个数据包时，将这个包报头的FLAG的`FIN`置位，意思将发送完毕，之后等待接收端回复`[FIN, ACK]`后结束发送线程。

   值得注意的是，当延迟很小（或者每一个包都有相同的时延），且接收端的丢包率为0，此时GBN退化为Part-01中的停等机制；原因则是这种情况，发送端发送的每一个数据包很快接受到了相应的回复，窗口向右移动1...

状态机参考课程中：

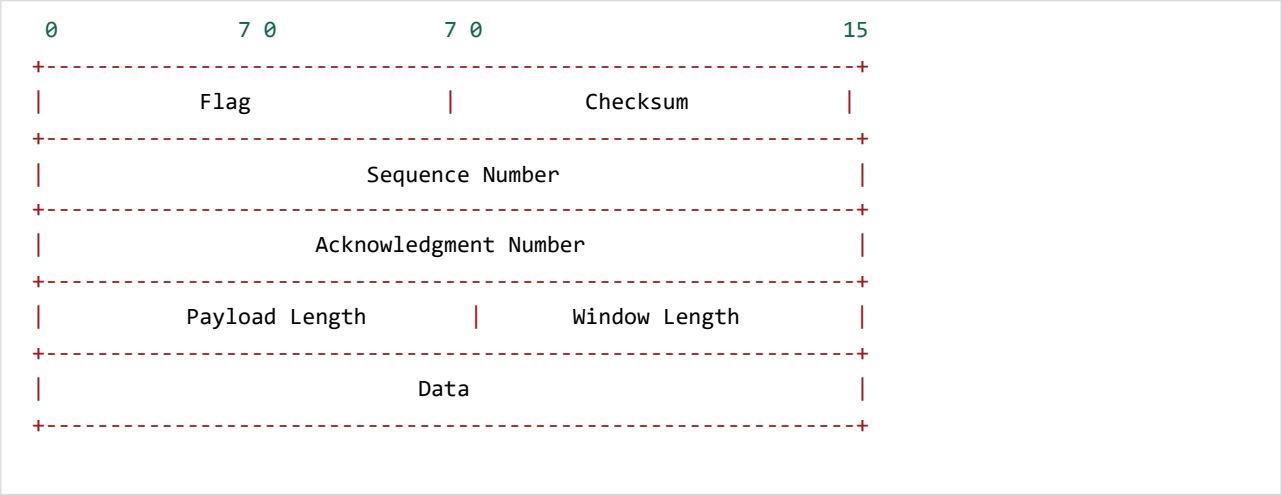
GBN发送端扩展FSM





报文设计

与上一次实验相比，GBN和SR机制中，FLAG标志位增加RE位，意思重传，同时选项内增加16位WINDOW，意位窗口大小。



字段名称	大小 (位)	描述
Flag	16	0... **RE** St Status SYN ACK FIN
Checksum	16	整个报文的校验和
Sequence Number	32	报文的序列号
Acknowledgment Number	32	确认序列号
Payload Length	16	Data的长度
WindowLength	16	窗口大小
Data	——	实际数据

程序设计

发送端

异步的发送和接受

1. Sender类成员

```
Sender {
//GBN
volatile int Window = 8;     //窗口大小
volatile int base = 0;       //窗口的左端点，是全局变量，由发送线程和接受线程共享
volatile int nextseq = 0;    //即将发送的数据序号
volatile bool Re = 0;        //一个标记位，接收线程得知超时时候置位，让发送线程重传
volatile clock_t timer;
pQueue watiBuffer = pQueue(Window); //窗口分为两部分（已发送和未发生，将已经发送的push进队列
```

2. **窗口缓存区** 发送区窗口本质是由sender->base+Window决定，而为了重发已经发送的窗口内的数据包，维护一个队列将这些数据包记录，便于将ack对应的数据包之前的全部数据包全部pop。下面展示核心函数GBNpop，在接受线程调用，若返回-1，则是表示当前接受的ACK没有在缓存区匹配的；若返回n(n>0)，表示匹配到第n个数据包，并且已经将这n个数据包pop出去。

```
class pQueue{
```

```
public:
    vector<Udp*> data;

    ... ...
    //return  -1 ACK小于窗口内最小值 >1 ACK大于循环第一个元素pop
    int GBNpop(const Udp& value){
        int va = value.header.ack;
        if ( this->empty() || va < (this->front()->header.seq + 1) )
            return -1;
      //注: va大于front,表示回复的ack丢失或者延迟，但是接受方已经收到，可以累计确认
        else {
            int rst = va - this->front()->header.seq;
            int n = rst;
            while (n--)
                this->pop();
            return rst;
        }
    }
};
```

3. **发送线程**将窗口内的数据按照序列发送，之后将其放入一个缓存区等待被ACK；同时还需要在超时重传时对超时的包进行重新发送；此外，关于ST、END的作用详细见上一个实验：Sender发送的第一个包包含文件的描述信息，设为ST包，发送的最后一个包是Fin包，既包含文件最后一部分，也意味着**二次挥手的开始**；最后关于ack和seq的设置，ack本次实验无用，而seq从0开始递增，使得让每一个数据包有了**唯一编码。**

```
DWORD WINAPI GBNSendHandle(LPVOID param){
    ... ...
    //status ST  ACK FIN  ack seq
    bool ST = 1;
    bool ACK = 0;
    bool FIN = 0;
    int ack = -1;
    int seq = 0;
    int payloadSize = PayloadSize;
    while (s->send_runner_keep) {
        //动态设置当前Window大小
        int N = s->Window;
        //根据Re变量得知现在是否应该重发
        if (FIN || s->Re) {
            vector<Udp*>tmp = s->watiBuffer.data;
            for (Udp* p : tmp) {
                p->header.set_r(1);
                s->_send(p, p->header.data_size);
                Sleep(32);
            }
            s->Re = 0;
            continue;
        }
        while (s->nextseq < s->base + N) {
            //发送
            Udp *package = new Udp(ST, ACK, FIN, seq, ack);
            if (FIN) {
                s->watiBuffer.push(package);
                s->_send(package, 0);
               // s->send_runner_keep=0;
                break;
```

```
            }

            else if (ST)
                package->packet_data(name_size.c_str(), sizeof(name_size));

            else {
                package->packet_data(iter, payloadSize);
                iter += PayloadSize;
            }
            //将已发送数据入队列
            s->watiBuffer.push(package);
            s->_send(package,payloadSize);

            if (s->base == s->nextseq)
                s->timer = clock();

            seq++;
            s->nextseq++;

            //状态改变
            if (ST && !FIN)
                ST = 0;
            if (!ST && (iter + PayloadSize) > end) {
                payloadSize = end - iter;
                FIN = 1;
            }
            Sleep(32);
        }
    }
    cout << "GDB Send Thread Finished" << endl;
    return 0;
}
```

4. **接受线程**：接受ACK包，将**对应**的存在缓存区内**待确认**的包进行确认，注意确认机制是累积确认，即这个ACK确认的是对应缓存区内包之前的所有数据;另外，它负责超时，之后通知发送线程进行重发

```
DWORD WINAPI GBNReciHandle(LPVOID param) {
    ... ...
    while (true) {
        while (recvfrom(s->s, ReciBuffer, PacketSize, 0, (struct sockaddr*)s->dst_addr,
  &dst_addr_len) <= 0) {
            if (clock() - s->timer > 10*MAX_TIME) { //重发
                s->timer = clock();
                s->Re = 1;
            }
        }
         Udp* dst_package = (Udp*)ReciBuffer;
         if (
             (dst_package->cmp_cheksum())
             && (dst_package->header.get_Ack())
             )
         {
             int rst = s->watiBuffer.GBNpop(*dst_package);
             switch (rst) {
             case -1:
                 //重复接受ACK
```

```
                print_udp(*dst_package, 3);
                break;
            default :
                //存在发送端ACK丢失或者延迟
                s->base+=rst;
                cout << " window --> " << rst<<" base " <<s->base<< endl;
                s->timer = clock();
                print_udp(*dst_package, 2);
                break;
            }
            if (dst_package->header.get_Fin()) {//关闭发送进程
                s->send_runner_keep = false;
                break;
            }
        }

    }
    cout << "Listen Thread exit 0"<<endl;
    return 0;
}
```

接收端

只需要根据接受到的数据包，进行判断，因此开一个线程接受，再结合状态控制进行发送。基本逻辑和**停等机制下的接收端相同**，差异只在只有expect_ack的机制与收到非法数据包是、重新发送expect_ack包。

- 首先有一组状态决定当前回应报文的Header； 其次声明指向文件的iter = reci->FileBuffer，用于将接受的报文写入。

```
bool fin = 0;
bool st = 1;
//bool RE = 0;
int expect_ack = 1;
int seq = 0;
char* iter = reci->FileBuffer;
```

- 若通过校验和、判断接受的ack是否是自己想要的，若不是，则重新发送自己expect_ack的数据包;否则根据当前状态和expect_ack发送ACK

```
    while (reci->reci_runner_keep && !fin) {
        if (fin || recvfrom(reci->s, buffer, PacketSize, 0, (sockaddr*)reci->send_addr,
&send_addr_size) <= 0)
            continue;
        //接受到数据包
        Udp* dst_package = (Udp*)buffer;
        print_udp(*dst_package, 1);
        //校验和检验 seq检验是否连续 否则置为r（重发位）
        if (!dst_package->cmp_cheksum() || !(dst_package->header.seq + 1 == expect_ack))
        {
            //cout << "---------- Lost or CheckSum error ---------\n" << endl;
            reci->package->header.set_r(1);
            reci->ThreadSend();
            reci->package->header.set_r(0);
```

```
            continue;
        }
        // 通过校验和

        if (dst_package->header.get_st())
            reci->InforBuffer += dst_package->payload;

        else {
            st = 0;
            memcpy(iter, dst_package->payload, dst_package->header.data_size);
            iter += dst_package->header.data_size;
        }

        if (dst_package->header.get_Fin())
      //fin报文收到后，则完全收到了，回复一个fin报文，让sender结束。即挥手第二次
            fin = 1;

        reci->package->header.set_Ack(1);
        reci->package->header.set_St(st);
        reci->package->header.set_Fin(fin);
        reci->package->header.seq = seq++;
        reci->package->header.ack = expect_ack++;
        reci->ThreadSend();

    }

    Sleep(100);
    cout << "Thread exit... \nTotal Length: " << reci->bytes << " bytes\n" << "Duration:
" << (double)((clock() - send_st) / CLOCKS_PER_SEC) << " secs" << endl;
    cout << "Speed Rate: " << (double)reci->bytes / ((clock() - send_st) /
CLOCKS_PER_SEC) << " Bps" << endl;
    reci->to_file();
    return 1;
}
```

实验结果

实验数据

传输协议	测试文件	吞吐率	传输时延
GBN	helloworld.txt	152078 Bps	11.374 s
GBN	1.jpg	155805 Bps	12.724 s
GBN	2.jpg	148011 Bps	40.044 s
GBN	3.jpg	148005 Bps	81.073 s

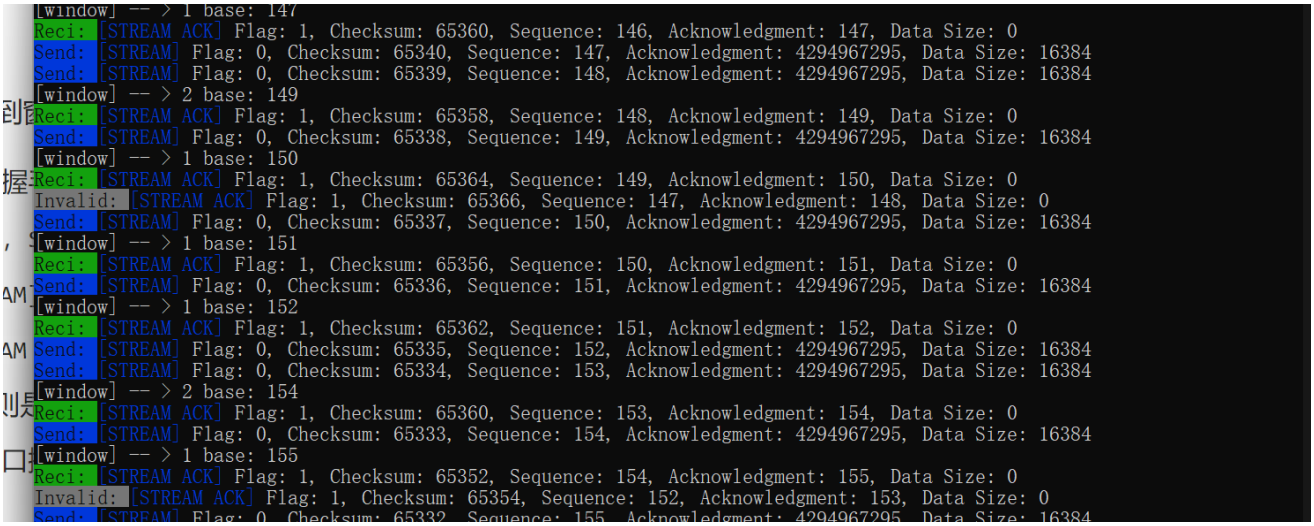
参数

> 丢包率:0.05　延迟 50-300ms　窗口内连续发送延迟 50ms

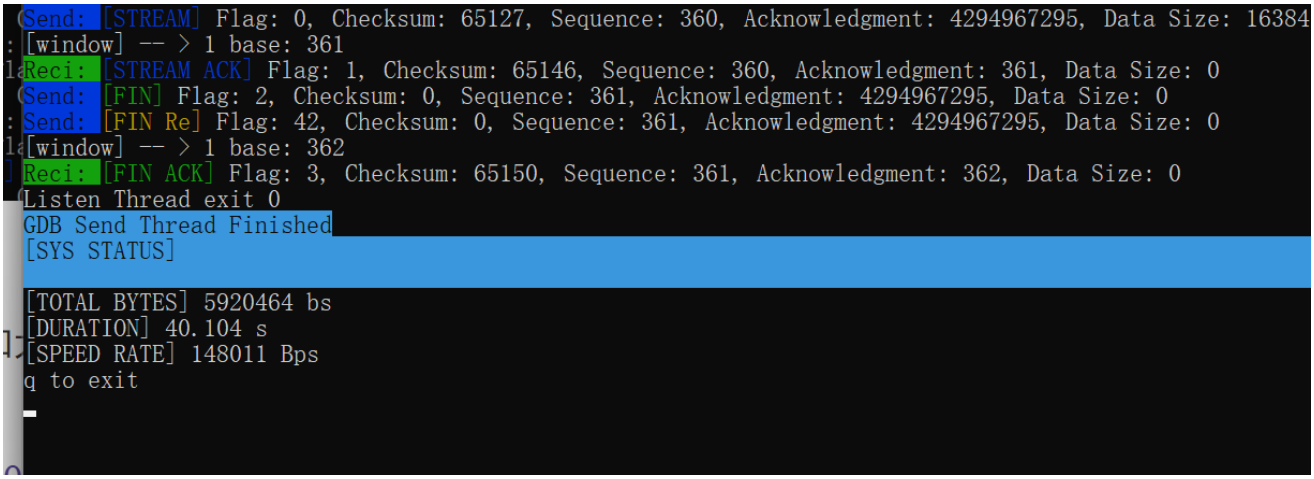
实验截图

发送端，涉及到窗口的改变，输出日志内容有如下信息：

- 首先是三次握手状态信息，输出[SYN ACK]相关报文
- 文件传输中，Send:[ST]报文是第一个数据包，里面包含了传输文件的描述信息，名字、大小等
- Send:[STREAM]报文是正常传输的报文，[STREAM RE]则是重传的数据包
- Reci:[STREAM ACK]是**正常**回复、且使得发送端窗口**变化**的**有效**ack报文
- [INVALID] 则是累积确认中，n+1的ack已经接受，但是此时 n 的ack才到达，这个 n 就是INVALID。
- 此外还有窗口挪动的信息，[window] -- > n base: x , n是窗口移动的位数，x是此时窗口的左区间



最后计算、输出吞吐率相关指标



接收端由于窗口大小为1，不存在窗口改变。输出的日志信息中如下，它输出的报文是上述发送端的子集。

```
wait...
------------------- Dst Package ----------------
SYN: 1 ACK: 0
第一次挥手成功
Reci: [SYN ACK] Flag: 5, Checksum: 65482, Sequence: 0, Acknowledgment: 0, Data Size: 0
------------------- Dst Package ----------------
SYN: 0 ACK: 1
第三次握手成功
Recive Thread ready

GBN RECIVING
GBN Start to Acc the file
Send: [ST] Flag: 20, Checksum: 65455, Sequence: 0, Acknowledgment: 4294967295, Data Size: 40
Reci: [ST ACK] Flag: 21, Checksum: 65506, Sequence: 0, Acknowledgment: 1, Data Size: 0
Send: [STREAM] Flag: 0, Checksum: 65486, Sequence: 1, Acknowledgment: 4294967295, Data Size: 16384
Reci: [STREAM ACK] Flag: 1, Checksum: 65512, Sequence: 1, Acknowledgment: 2, Data Size: 0
Send: [STREAM] Flag: 0, Checksum: 65485, Sequence: 2, Acknowledgment: 4294967295, Data Size: 16384
Reci: [STREAM ACK] Flag: 1, Checksum: 65504, Sequence: 2, Acknowledgment: 3, Data Size: 0
Send: [STREAM] Flag: 0, Checksum: 65484, Sequence: 3, Acknowledgment: 4294967295, Data Size: 16384
Reci: [STREAM ACK] Flag: 1, Checksum: 65510, Sequence: 3, Acknowledgment: 4, Data Size: 0
Send: [STREAM] Flag: 0, Checksum: 65483, Sequence: 4, Acknowledgment: 4294967295, Data Size: 16384
Reci: [STREAM ACK] Flag: 1, Checksum: 65502, Sequence: 4, Acknowledgment: 5, Data Size: 0
Send: [STREAM] Flag: 0, Checksum: 65482, Sequence: 5, Acknowledgment: 4294967295, Data Size: 16384
Reci: [STREAM ACK] Flag: 1, Checksum: 65508, Sequence: 5, Acknowledgment: 6, Data Size: 0
Send: [STREAM] Flag: 0, Checksum: 65481, Sequence: 6, Acknowledgment: 4294967295, Data Size: 16384
Reci: [STREAM ACK] Flag: 1, Checksum: 65500, Sequence: 6, Acknowledgment: 7, Data Size: 0
Send: [STREAM] Flag: 0, Checksum: 65480, Sequence: 7, Acknowledgment: 4294967295, Data Size: 16384
```

```
Send: [STREAM] Flag: 0, Checksum: 65128, Sequence: 359, Acknowledgment: 4294967295, Data Size:
Reci: [STREAM ACK] Flag: 1, Checksum: 65154, Sequence: 359, Acknowledgment: 360, Data Size: 0
Send: [STREAM] Flag: 0, Checksum: 65127, Sequence: 360, Acknowledgment: 4294967295, Data Size:
Reci: [STREAM ACK] Flag: 1, Checksum: 65146, Sequence: 360, Acknowledgment: 361, Data Size: 0
Send: [FIN] Flag: 2, Checksum: 0, Sequence: 361, Acknowledgment: 4294967295, Data Size: 0
Reci: [FIN ACK] Flag: 3, Checksum: 65150, Sequence: 361, Acknowledgment: 362, Data Size: 0
Thread exit...
Total Length: 5808 bytes
Duration: 40 secs
Speed Rate: 145.2 Bps
Write Fin
```

失序、效率思考

失序问题

在上一次停等实验中，一旦中间链路导致的延迟过大，配合发送端的重传机制导致了包失序问题，而这次实验，接收端expect_ack机制，强制接受连续的数据包，保证了接受数据包不存在失序，但也同时导致时间的浪费，如下讨论：

效率问题

首先有如下结论，当链路层传输的时延**接近**接收端发送端两次发送的间隔，且**无丢包**现象，GBN机制退化成停等(只是不再是0/1状态的转移)，因为发送端每发一个包，立即接受ACK，窗口右移动1，再次发送下一个包....最后使得发送缓存区队列最多一个元素。

当仅加入丢包，此时停等机制效率进一步降低；因为一旦x丢包，窗口队列里x,x+1,x+2...等包在超时后重新发送，而后续的包在接受端也无确认只是丢弃，导致在相同丢包率下，GBN容易发送大量无效包占用宽带，效率反**不如停等。**

当仅加入波动性时延（即每一个数据包对应的时延是不一定的），会导致Sender接受ack=x,但是x-1、甚至x-n都没收到，但由于累积确认的机制，发送端此时跳过了对x-1至于x-n的接受，这种角度**GBN似乎提升了效率**

因此有一个粗浅的结论，时延波动性大、丢包率较低的网络参数下、窗口长度小的情况下，GBN是优化的，否则会导致占用大量宽带，反而导致效率更低。而第四次实验将会有更充分的测试数据进行验证。