学号：2111252 姓名：李佳豪

代码链接:https://github.com/FondH/cn/tree/StableUdp-Part3

UDP可靠传输-Part03

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**选择**确认，完成给定测试文件的传输。

SR实验要点

1. 同GBN一样的滑动窗口机制，只是选择确认机制下，当接受线程收到一个窗口范围内ack时只将对应数据包进行确认，而不继续将之前所有数据包累计确认。
2. 超时重传，仅仅对窗口内已经发送，但未被确认的数据包发送。
3. **接收端的接受**，也需要自己的窗口，本次实验默认于发生端相同；接受当前窗口内的包，并发出ack。
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

报文设计

和GBN报文完全一致

```
 0             7 0             7 0                             15
+-----------------------------------------------------------------+
|            Flag               |          Checksum               |
+-----------------------------------------------------------------+
|                         Sequence Number                         |
+-----------------------------------------------------------------+
|                      Acknowledgment Number                      |
+-----------------------------------------------------------------+
|           Payload Length        |       Window Length           |
+-----------------------------------------------------------------+
|                              Data                               |
+-----------------------------------------------------------------+
```

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

1. Sender类成员，同GBN完全一致

```
Sender {
//SR
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**，窗口本质是由sender->base+Window决定，维护一个队列将记录窗口内**已经发送**的数据包
   - data存储了这些数据包，由发送线程进行push，waitAck对应着这些数据包是否被确认,数据包刚被push进入时，对应的waitAck置0；

- SRpop(const Udp& value)，参数是接收到数据包，根据其中ack匹配当前队列中数据包，将其waitAck的值置为，若使得窗口内第一个数据包置为，则表示窗口可以移动，于是进一步扫描移动的步长，并pop已经确认的数据包。

```
class pQueue{
public:
    vector<Udp*> data;
    vector<bool> waitAck;
    ... ...
    // return n 窗口移动n位  0 ack确认，窗口未移动  -1未Ack成功
    int SRpop(const Udp& value) {
        int indx = 0;
        while (indx < this->data.size()) {
            if (indx >= data.size() || indx >= waitAck.size())
                break;
            //找到value对应下标
            if (value.header.ack == data[indx]->header.seq + 1)
                break;
            indx++;
        }
        if (indx >= data.size() || indx >= waitAck.size())
            return -1;

        this->waitAck[indx] = 1;
       // 若indx是0，表示窗口最左侧数据包确认，可以移动窗口
        if (!indx) {
            while (indx < data.size() && indx < waitAck.size() &&                    this-
>waitAck[indx])
                indx++;

            for (int i = 0;i < indx;i++)
                this->pop();
            return indx;
        }

         return 0;
    }
    ... ...
};
```

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；重传时只需将窗口缓存区，也就是上一部分中维护的队列中，waitAck仍然是0的进行发送；
- 此外，关于ST、END的作用：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**。

  代码除重传部分代码外，和GBN完全一致，具体见Sender.h文件220行

```
DWORD WINAPI SRSendHandle(LPVOID param) {
    //重传
    if (FIN || s->Re) {
```

```
    //Recive线程将s->Re置位，对处于队列中的进行发送。
        vector<Udp*>tmp = s->watiBuffer.data;
        vector<bool>waitAck = s->watiBuffer.waitAck;
        int i = 0;
        for (Udp* p : tmp) {
             p->header.set_r(1);
            if (!waitAck[i++])
                s->_send(p, p->header.data_size);
             Sleep(2);
        }
        s->Re = 0;
        continue;
    }
    ... ...
}
```

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，将对应窗口缓存区的waitAcck置位，这些操作均在SRpop(*dst_package)完成,根据它的返回值，若-1则表示该数据包不在当前窗口范围内，当抛弃；若返回0，则表示该数据包是当前窗口内的，但未引起窗口移动；返回大于n(n>0)表示数据包属于当前窗口内，且窗口需要移动n；
- 另外，它负责超时，之后通知发送线程进行重发

```
DWORD WINAPI SRReciHandle(LPVOID param) {
...
    while (true) {

        while (recvfrom(s->s, ReciBuffer, PacketSize, 0, (struct sockaddr*)s->dst_addr,
&dst_addr_len) <= 0){
            if (clock() - s->timer > 5 * MAX_TIME) {
                s->timer = clock();
                s->Re = 1;
            }
        }//解析包
        Udp* dst_package = (Udp*)ReciBuffer;
        if (
            (dst_package->cmp_cheksum())
            && (dst_package->header.get_Ack())
            )
        {
            int rst = s->watiBuffer.SRpop(*dst_package);
            cout << dst_package->header.ack << " buffer: " << s->watiBuffer.data.size()
                << " rst: " << rst;
            switch (rst) {
            case -1:
                //ACK在窗口范围外
                cout << endl;
                print_udp(*dst_package, 3);
                break;
            case 0:
                //ACK可以确认，但第一个包没确认 因此窗口不移动
                cout<< endl;
                print_udp(*dst_package, 2);
                break;
            default:
                s->base += rst;
```

```
                cout << " window --> " << rst<<" base " <<s->base<< endl;
                s->timer = clock();
                print_udp(*dst_package, 2);
                break;
            }
...
}
```

接受端

接收端和GBN、停等不同的是，他也有窗口的概念，需要将收到的数据包seq值进行检测，确认是否在当前窗口范围内，是则将数据加入缓冲区（缓冲区机制见下面1），而接收端的窗口定义，也是`[base - base+N]`，这里base随着接收到的数据不断增加，N则是规定的窗口大小（默认同发送端一致）

1. 接受窗口缓存区：本质是队列
    - `push(Udp* u)`：函数保证了push进入的数据包按照其Header的**seq**从小到大
    - `rpop(char ** iter)`：将队列里从第一个元素开始连续的数据包进行pop，同时利用`*iter`，将这些连续包的数据段写入`*iter`指向的空间。

```
class rQueue {
public:
    vector<Udp*> data;
    //保证从小到大  return 0：  成功；   -1：  重复
    int push(Udp* u) {
        ... ...
        auto it = lower_bound(data.begin(), data.end(), value,
            [](const Udp* a, const Udp* b) {
            return a->header.seq < b->header.seq;
        });
        data.insert(it, value); // 在找到的位置插入新元素
        ...
    }
    int rpop(char** iter) {
        int indx = 0;
        while (indx< data.size()) {
            //连续的区间写入
            memcpy(*iter, data[indx]->payload, data[indx]->header.data_size);
            *iter += data[indx]->header.data_size;

            if (data.size() == 1)
                break;
            // 检查下一个元素是否存在且序列连续
            if (indx + 1 < data.size() && data[indx]->header.seq + 1 != data[indx + 1]-
>header.seq)
                break;
            if (indx + 1 == data.size())
                break;
            indx++;
        }
        for(int i=0;i<=indx;i++) //进行前面的pop
            this->pop();
        return ++indx;
    }
```

2. 接受线程：

- 对接收的dst_seq，大于窗口右端点的直接丢弃，因为这超出了接收端模拟的数据处理速度；
- 若小于右端点，说明存在Ack丢失或者延迟，那么对这个dst_seq发送对应的ack包，以保证发送端窗口正确移动；
- 若在窗口区间内部，调用qbuffer.push()操作将数据包加入缓冲区，此时做一次判断，若这个数据包是窗口左端点，调用qbuffer.rpop(&iter)进行窗口的移动 -- 将窗口内连续的包进行pop、写入文件buffer；

  核心代码如下：

```
DWORD WINAPI SRReciHandler(LPVOID param) {
    ... ...
    int base = 0;
    int N = reci->window;
    rQueue qbuffer = rQueue(N);
    char* iter = reci->FileBuffer;

    while (reci->reci_runner_keep && !fin) {
        if (fin || recvfrom(reci->s, buffer, PacketSize, 0, (sockaddr*)reci->send_addr,
&send_addr_size) <= 0)
            continue;

        Udp* dst_package = (Udp*)buffer;
        print_udp(*dst_package, 1);
        int dst_seq = dst_package->header.seq;
        if (!dst_package->cmp_cheksum() || (dst_seq >= base + N))
            continue;

        // 通过校验和 在窗口内
        if (dst_seq < base) {
            reci->package->header.set_r(1);
            reci->ThreadSend();
            reci->package->header.set_r(0);
            continue;
        }

        if (dst_package->header.get_st()) {
            reci->InforBuffer += dst_package->payload;
            base++;
        }
        else {
            qbuffer.push(dst_package);
            cout << dst_package->header.seq << " buffer_size: " << qbuffer.CurrNum;
            st = 0;
            if (dst_seq == base) {//只有接受的seq是窗口左端点才会导致窗口移动
                shift = qbuffer.rpop(&iter);
                base += shift;
                cout << " window --> " << shift << " base " << base << endl;
            }
        }
        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 = dst_package->header.seq + 1;
            reci->ThreadSend();//发送
        }
        ... ...
    }
```

实验结果

实验数据

传输协议	测试文件	吞吐率	传输时延
SR	helloworld.txt	869240 Bps	2.863 s
SR	1.jpg	508420 Bps	4.094 s
SR	2.jpg	517973 Bps	12.125 s
SR	3.jpg	465884 Bps	27s

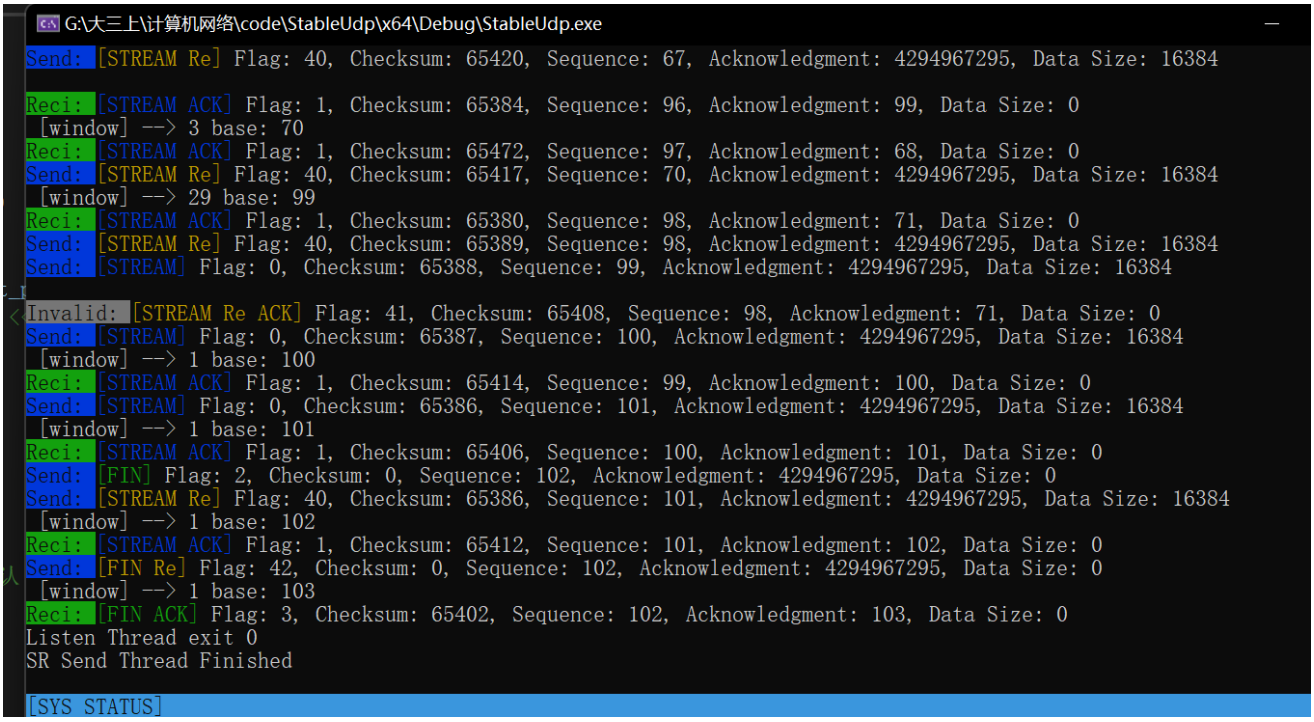
参数

> 丢包率:0.05 延迟 50-300 ms 窗口内连续发送延迟 2ms 超时时间 200 ms

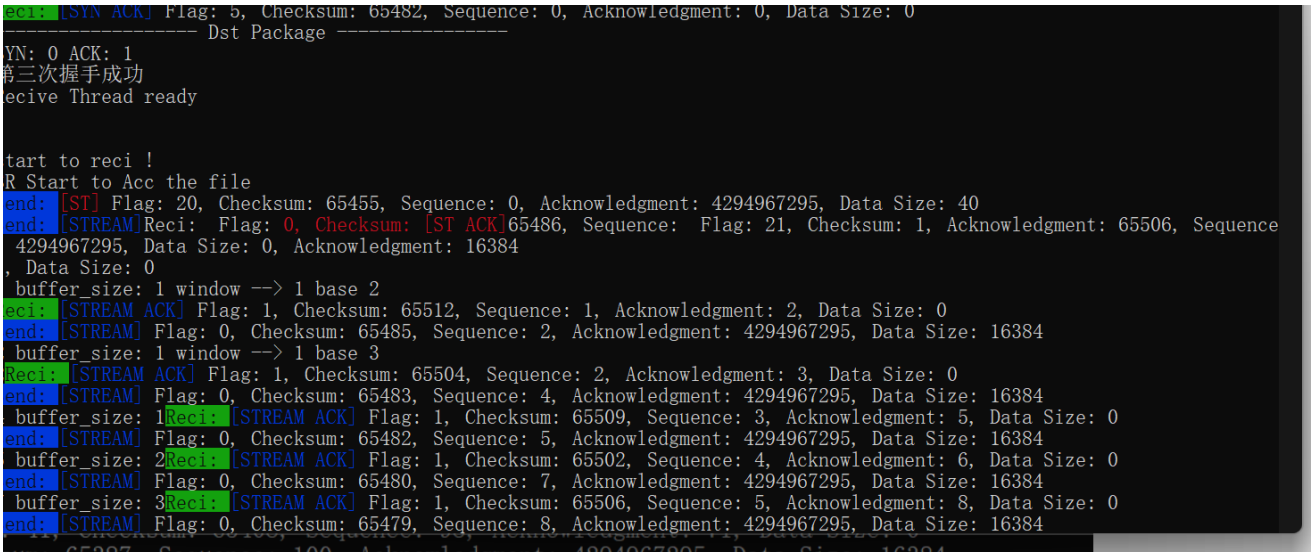
输出日志

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]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]` 则是选择确认中，窗口之外的包
- 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



接收端，输出格式类似上面，输出报文信息以及窗口移动信息，最后输出吞吐量等指标



```
Send: [STREAM Re] Flag: 40, Checksum: 65417, Sequence: 70, Acknowledgment: 4294967295, Data Size: 16384
70 buffer_size: 29 window --> 29 base 99
Reci: [STREAM ACK] Flag: 1, Checksum: 65380, Sequence: 98, Acknowledgment: 71, Data Size: 0
Send: [STREAM Re] Flag: 40, Checksum: 65389, Sequence: 98, Acknowledgment: 4294967295, Data Size: 16384
Reci: [STREAM ACK] Flag: 1, Checksum: 65408, Sequence: 98, Acknowledgment: 71, Data Size: 0
Send: [STREAM] Flag: 0, Checksum: 65388, Sequence: 99, Acknowledgment: 4294967295, Data Size: 16384
99 buffer_size: 1 window --> 1 base 100
Reci: [STREAM ACK] Flag: 1, Checksum: 65414, Sequence: 99, Acknowledgment: 100, Data Size: 0
Send: [STREAM] Flag: 0, Checksum: 65387, Sequence: 100, Acknowledgment: 4294967295, Data Size: 16384
100 buffer_size: 1 window --> 1 base 101
Reci: [STREAM ACK] Flag: 1, Checksum: 65406, Sequence: 100, Acknowledgment: 101, Data Size: 0
Send: [STREAM] Flag: 0, Checksum: 65386, Sequence: 101, Acknowledgment: 4294967295, Data Size: 16384
101 buffer_size: 1 window --> 1 base 102
Reci: [STREAM ACK] Flag: 1, Checksum: 65412, Sequence: 101, Acknowledgment: 102, Data Size: 0
Send: [FIN] Flag: 2, Checksum: 0, Sequence: 102, Acknowledgment: 4294967295, Data Size: 0
102 buffer_size: 1 window --> 1 base 103
Reci: [FIN ACK] Flag: 3, Checksum: 65402, Sequence: 102, Acknowledgment: 103, Data Size: 0
Thread exit...
Total Length: 1696 bytes
Duration: 7 secs
Speed Rate: 242.286 Bps
Write Fin
S
```

思考、分析

本次实验发送端的超时重传，我默认使用一个"Re"标记位，使得重发和窗口内其余包的发送无法并行，这个问题在GBN中并不严重，因为当重发x包时，实际上窗口内还未发送且x后面的包没必要发送；在SR中，由于接收端拥有了对包选择、缓存的机制，因此重发也需要一个新线程，使得效率更高，我也带源码中做出对应的扩充；

我们新增一个SR机制下的重传线程，这些代码原本在SR的发送线程里，此时新开一个线程进行处理；

```
DWORD WINAPI SRReSendHandler(LPVOID param){
    Sender* s = (Sender*)param;
    if (s->Re) {
        //Todo lock 否则导致死循环或者刚发送的数据包还进行重传；
        vector<Udp*>tmp = s->watiBuffer.data;
        for (Udp* p : tmp) {
            p->header.set_r(1);
            s->_send(p, p->header.data_size);
            Sleep(TWICE_GAP);
        }
        //s->watiBuffer.SRpop(0);
        s->Re = 0;

    }
    return 1;
}
```

不过值得一提的是，最后的结果发现传输时间并没有**什么提升**，原因大概如下:我们发送端处理往往较快的，若我窗口内所有数据总是能在超时时间内全部发送完毕，实则上述增加一个新线程进行重发收益并不好；此外，由于重传和发送这俩线程冲突（共同访问queue），需要引入锁，否则导致大量没必要重传的包重传，而引进锁自然导致效率降低。因此实际测试的时候，我并没有很好解决使用**新线程**重传进而提升传输效率；