南開大學
Nankai University

南　　开　　大　　学

网 络 空 间 安 全 学 院

计算机网络实验报告

Lab3-2 滑动窗口实现可靠传输

2113946　刘国民

年级：2021 级

专业：信息安全

2023 年 12 月 1 日

目录

一、 实验内容

本次实验在实验 3-1 的基础上，引入基于滑动窗口的流量控制机制，采用累积确认的方式，完成给定测试文件的传输。同时在传输完成后，会计算传输时间和吞吐率等相关性能指标，并改变路由程序的丢包率和延时来进行分析。

二、 协议设计

协议设计分为报文格式、差错检验、建立连接、数据传输和关闭连接五个部分。该部分重点阐述本次实验不同于上次实验的地方（即数据传输部分），差错检验、报文格式、三次握手以及四次挥手的协议设计与上次实验基本一致，本次实验只作简要叙述、

(一) 报文格式

本次实验中采用以下结构的数据报：

```
struct Datagram {
    bool ack, syn, fin;
    uint16_t checksum;//16位校验和
    long long seqnum, acknum;
    int DataLen;// DataLen<=MaxBufferSize
```

```
    char data[MaxBufferSize] = { 0 };
}SendData, ReceiveData;
```

与上次实验一样，依旧用结构体将原始发送数据与 ack,syn,fin 等标志位和校验和等协议开销封装在一起。SendData,ReceiveData 用来填充每次发送和接收的数据包

(二) 差错检验

差错检验使用以下算法实现：

1. 将数据报按 16bit 为一组进行分组，不足一组的用 0 补齐
2. 将 checksum 字段置 0
3. 按位累加所有组的值，每次相加如果最高位有溢出则补到 16 位数的最低位上
4. 将上述步骤的结果取反后填入 checksum 字段

需要注意的是，实验中用 RECEIVE_CHECK 和 SEND_CHECK 两个宏来标识是否取反，发送的时候需要取反，接受时则不用取反，接收时验证算出的校验和与数据包的校验和字段相加是否为 0xffff 即可。

(三) 建立连接

依旧采用以下方式建立连接：

1. 首先发送方发送一个空数据报给接收方（全为 0），仅将 syn 标志位置为 1，表示发送方请求建立连接。同时阻塞等待接收方的 syn+ack 包。
2. 接收方持续阻塞等待发送方的 syn 包，如果成功接收到 syn 标志位为 1 的数据包，则将一个空包的 syn、ack 和 acknum 位置为 1 后发回，告诉发送方成功收到 seqnum 为 0 的 syn 包。
3. 发送方成功收到 syn+ack 包后，发回一个 ack 包，同时设置 seqnum 和 acknum 为 1，表示成功收到 seqnum 为 0 的 syn+ack 包，同时序列号自增 1。至此连接建立完成，

建立连接的具体流程基本与 TCP 三次握手一致，在这里双方的初始序列号 ISN 均取 0。

(四) 传输数据

本次实验设计的协议类似于 GBN 协议，但不同于 GBN 协议的是，协议中使用的是非重复序列号，即从 0 号包开始，不使用循环序列号。接收端持续跟踪期望的序列号即可、下面重点阐述是如何传输数据的。

1. 滑动窗口机制

本次实验与实验 3-1 最大的不同在于，允许发送方在没有接收到确认 (ACK) 的情况下发送多个数据包，而不需要发送一个数据包等一个 ACK 回复然后再开始发送下一个。详细说明如下：

- 窗口大小：窗口大小即传输前预先设定的一个数值，表示发送方在等待 ACK 之前可以发送的最大数据包数量。在未到达窗口右边界之前，发送端将不断发送数据包给接收端。

- 累积确认：接收端发回的 ACK 代表在此之前的数据包全部收到。同时接收端维护一个期望序列号的变量，如果收到数据包的序号大于期望值，则丢弃数据包并发回一个期望值-1 的 ACK；如果收到数据包序号等于期望值，则发回一个期望值序号的 ACK，同时期望值 +1；如果收到数据包的序号小于期望值则直接丢弃即可，但也需要发回一个期望值-1 的 ACK。
- 窗口右移：发送方用一个变量 Base 来记录窗口的左边界，只有当收到 acknum 大于或等于 Base 时，窗口才向右移动。可以知道，acknum 的含义是接收端已经收到在此之前的数据包，所以不管 acknum 大于还是等于 Base，窗口右移时移动到 acknum+1 即可，即 Base=acknum+1。同时，由于窗口左边界的移动，右边界也会相应向右移动，进而发送新的数据包给接收端。
- 超时重传：发送端在发出 Base 包或者右移到新的窗口值时，会启动定时器来等待接收端的 ACK，如果超过设置的时间后，需要将整个窗口重传一次，并重新为 Base 包开启定时器。
- 流量控制：通过限制在确认之前可以发送的数据包数量来实现流量控制。这也是设置窗口大小的意义所在。

基于上述协议，整体的传输流程即是：发送端不断发送数据包，到达发送上限，等待 ACK 回复，收到正确回复就右移窗口继续发送，超时即重传整个窗口；接收端持续接收数据包，不是需要的数据包序号则丢弃。窗口图示如下：

图 1: 滑动窗口

2. 情形分析

基于设计的协议，我们分析丢失、错误等情形下该协议是怎么运转的。

1. 发送数据包丢失/出错：发送端的数据包丢失后，在此之前的数据包都能收到正确的 ACK 回复，窗口左边界滑动到该包时，发送端开启定时器，接收端也在一直等待该包或者出错时丢弃，发送端无法收到 ACK 回复，超时后将该包之后的数据包全部重传。
2. ACK/丢失/出错/提前超时：假设发送端发送一个 Base=x 的数据包，为它开启定时器，接收端正确收到数据包但回复丢失、出错或超时。在此情况下发送端重发整个窗口，由于上次已经成功接收，此时接收端期望的序列号已经是 x+1，但收到的数据包序号小于期望值时，也会发回 acknum=x 的回复，而发送端收到 acknum=Base 的回复，直接移动窗口即可，传输继续正常进行。

（五） 关闭连接

关闭连接也使用与 TCP 协议类似的流程，如下图所示：

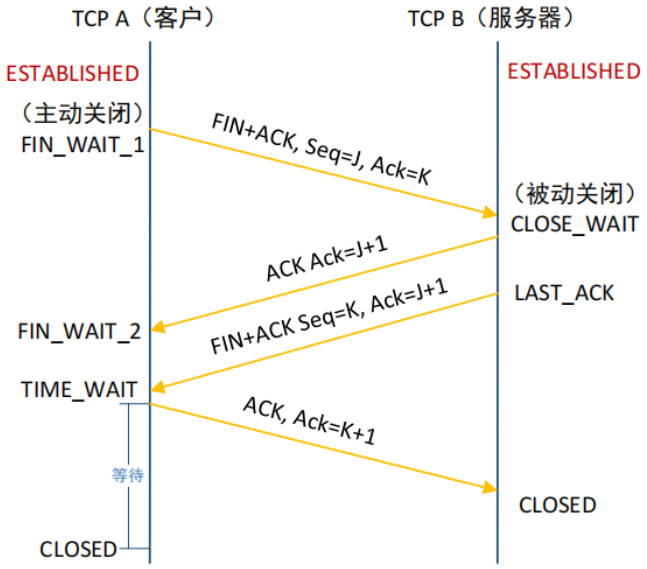

图 2: 关闭连接

如上图所示，但序列号 J 和 K 在此处跟三次握手类似，均取为 0。

三、 代码实现

代码实现部分主要分析传输数据部分的代码，差错检验、Socket 初始化、建立和关闭连接的代码与上次实验相同，不再赘述。建立连接时，我将文件名放入了第三次握手 ACK 报文的空闲缓冲区，一起发送给接收端。

(一) 发送端

根据协议设计，发送端的总体实现思路如下：

- 发送端用主线程 main 和两个创建的子线程 Resend 和 Timer 完成协议给出的功能和操作。
- 主线程 main 负责发送数据包，即一旦窗口左边界右移，主线程负责发送新的数据包给接收端。
- 子线程 Resend 负责持续检测是否“超时”，“超时”则重传, 这里的“超时”指的是在指定时间内没有收到正确的 acknum（即 acknum>=Base)
- 子线程 Timer 负责为 Base 包开启定时器，一旦超时则通知线程 Resend，这里的“通知”是通过全局变量的设置来实现，稍后会在具体的代码实现中看到。

首先我们给出下列全局变量：

```
// 多线程共享数据,写数据的时候用互斥锁
int Flag_Resend = 0;// 表示recvfrom的数据是否超时
int Flag_Timer = 0;// 表示是否需要启用定时器服务
int SeqNumber = 0;// 记录一共有多少个数据包,seqnum:0--SeqNumber-1
long long Base = 0;// 初始序列号从0开始
long long NextSeq = 0;
```

从注释中可以看到，Flag_Resend 用于表示收到的数据是否超时，只要子线程 Timer 发现超时了就将该标志位置为 1，这时候子线程 Resend 就可以检测到，从而开始重发窗口中的数据包。Flag_Timer 用于表示是否需要开启定时器，即主线程或者 Resend 线程发送 Base 包，或者窗口右移更新 Base 值的时候，就将该标志位置为 1，Timer 线程就会开始启用定时器，持续接收可能的 ACK 回复。而 Base,NextSeq 和 SeqNumber 为全局变量，为三个线程共同使用。

在本次实验中，我们用一个 vecotr 容器来保存所有要传输的数据包，在传输之前我们先把原始文件以 4096 字节为一组放入 vector 中，具体代码如下:

```
// 移动文件指针到文件末尾
fseek(p, 0, SEEK_END);
long long FileLen = ftell(p);
fseek(p, 0, SEEK_SET);
long long temp = FileLen;
// 数据以4096字节为单位放入vector序列
while (temp>0) {
    SendData = { 0 };
    fread(SendData.data, MaxBufferSize, 1, p);
    SendData.DataLen = temp < MaxBufferSize ? temp : MaxBufferSize;
    SendData.seqnum = SeqNumber++;
    v.push_back(SendData);
    temp -= SendData.DataLen;
}
```

fread 将数据读取到发送数据包缓冲区中，将相关信息封装好后放入容器，之后涉及重传的时候，直接从容器中拿取数据包重发即可，对于窗口的滑动，直接增加 Base,NextSeq 等变量后通过 vector 随机访问即可，SeqNumber 用来记录一共有多少个数据包。接下来我们逐个分析三个线程的具体代码。

```
while (true) {
    if (NextSeq == SeqNumber) {// 发送完毕
        break;
    }
    if (NextSeq < Base + WindowSize) {
        SendData = v[NextSeq];
        Send();
        if (NextSeq == Base) {// 第一个包
            lock_guard<mutex> lock(mtx);
            Flag_Timer = 1;
        }
        {
            lock_guard<mutex> lock(mtx);
            NextSeq++;
        }
        Sleep(100);;//每次发送睡眠100毫秒，避免短时间内发送大量数据包给接收方
    }
}
```

上述代码是 main 函数中发送数据的核心代码，整体逻辑是持续检测 NextSeq 是否小于 Base+WindowSize，即发送的数据包是否到达右边界，如果没到达就持续发送并增加 NextSeq

的值。如果发送的是 Base 包则需要设置 Timer 标志位，通知子线程启动定时器。其中 v 即是上述提到的 vector 容器，需要注意在这个过程中，对多线程的共享数据（这里为 Flag_Timer）进行写操作时，需要使用互斥锁以阻塞子线程进行操作，避免多线程同时进行写操作从而引发错误。每次发送后，使用 sleep 函数睡眠 100 毫秒，避免短时间发送大量数据包给接收方。如果 NextSeq 如果等于 SeqNumber，则代表窗口右边界已经到达容器末尾，此时主线程直接 break 退出循环即可。后续可能涉及到的重传由 Resend 完成。

```
void Resend() {// 只要检测到Flag_Resend变为1就重发Base--(NextSeq-1)
    while (true) {
        if (Flag_Resend == 1) {
            BackCounter++;
            for (int i = Base; i <= NextSeq-1; i++) {
                SendData = v[i];
                Send();
                if (i == Base) {//Base包
                    lock_guard<mutex> lock(mtx);
                    Flag_Timer = 1;//启动定时，通知Receive线程
                }
            }
            {
                lock_guard<mutex> lock(mtx);
                Flag_Resend = 0;
            }
        }
        if (Base==SeqNumber) {//传输完毕
            cout << "[提示]: ReSend线程正确结束" << endl;
            return;// 结束进程
        }
    }
}
```

上述代码是子线程 Resend 绑定的函数，核心逻辑就是 while 循环持续检测 Flag 标志位，如果为 1 就开始重传窗口，重传数据也是通过容器 v 获得，BackCounter 用于记录整个传输过程中的重传次数。与主线程类似，发送 Base 包需要启动定时器，写操作使用互斥锁。当 Base=SeqNumber，即左边界也到达文件末尾时，此时退出循环。

```
void Timer() {// 持续监测Flag_Timer，只要Flag_Timer=1就对Base包启动定时器
    while (true) {
        if (Base==SeqNumber) {//传输完毕
            cout << "[提示]: Timer线程正确结束" << endl;
            return;// 结束进程
        }
        clock_t start, end;
        if (Flag_Timer == 1) {
            int len = sizeof(Receiver_addr);
            start = clock();
            while (true) {
                int recv = recvfrom(SenderSocket, (char*)&ReceiveData,
                    DatagramLen, 0, (struct sockaddr*)&Receiver_addr, &len);
```

```
                int check = Checksum(ReceiveData, RECEIVE_CHECK);
                // recvfrom非阻塞，如果接收到的数据无误且ack大于Base,则移动窗口左边界，用队列来实现
                // 同时主线程监测到NextSeq<Base+WindowSize，传输新增的右边界窗口，从而实现滑动窗口的功能
                if (recv != -1 && ReceiveData.acknum >= Base && (check ^ ReceiveData.checksum) == 0xffff) {
                    cout << "收到acknum为" << ReceiveData.acknum << "的回复" << endl;
                    lock_guard<mutex> lock(mtx);
                    Base = ReceiveData.acknum + 1;
                    break;//跳出内层循环后，因为Flag_Timer仍然为1，会为新的Base设置定时器
                }
                end = clock();
                if (end - start >= timeout) {// 超时
                    lock_guard<mutex> lock(mtx);
                    Flag_Resend = 1;// 写操作使用互斥锁,通知Resend重传整个窗口
                    Flag_Timer = 0;// 重传的时候再设置为1
                    break;
                }
            }
        }
    }
}
```

与实验 3-1 类似，本次实验在建立连接后将 recvfrom 设置为非阻塞，在传输结束开始断开连接的时候恢复为阻塞模式。上述代码即用 while 循环持续检测 Flag_Timer，如果为 1 启动定时，首先先用 start 获取当前时间，之后再用一个 while 循环持续 recvfrom，如果成功收到正确的数据，且 acknum>=Base, 则改变 Base, 这里需要注意不是 Base+1，而是 acknum+1，因为可能收到的 acknum 大于 Base，同时跳出内层循环。在这里没有将 Flag_Timer 置为 0，因为 Base 增加，需要为新的 Base 包启动定时器。如果一直没有收到符合要求的数据包，end 不断获取时间，当持续时间超过设置的时间时，将 Flag_Resend 设为 1，Resend 线程就会立即进行相应操作。同时将 Flag_Timer 设为 0，跳出内层循环。等到 Resend 发出第一个包时，再将 Flag1，继续开启定时器。

(二) 接收端

相较发送端，接收端的逻辑会简单很多，不需要开多个线程，只需要在主函数中持续监听，获取需要的数据包，发送 ACK 并写回文件中。具体代码如下:

```
while (true) {
    ReceiveData = { 0 };
    bool recv=Receive();
    if (recv && ReceiveData.fin == true && ReceiveData.ack == true) {// 第一次挥手
        break;
```

```
    }
    if (recv) {//数据无误
        if (ReceiveData.seqnum == ExpectedNum) {
            fwrite(ReceiveData.data, ReceiveData.DataLen, 1, p);//写回文件
            cout << "成功接收第" << ExpectedNum << "个数据包" << endl;
            SendData = { 0 };
            SendData.acknum = ExpectedNum;
            ExpectedNum++;
            Send();
        }
        else if(ReceiveData.seqnum > ExpectedNum){// 收到的不是期望的数据包，但也需要回ACK
            SendData = { 0 };
            SendData.acknum = ExpectedNum-1;
            Send();
        }
        else {
            cout << "收到重复数据包，seqnum为： " <<ReceiveData.seqnum << endl;
            SendData = { 0 };
            SendData.acknum = ExpectedNum - 1;
            Send();
        }
    }
}
```

核心逻辑就是 recv 正确的数据包，分 seqnum 大于、等于或者小于期望序列号 ExpectedNum 三种情况进行处理。如果大于则丢弃数据包，但需要回复 ACK，这里 acknum 为 ExpectedNum-1；如果等于则是期望的数据包，需要写文件同时 ExpectedNUm 加 1，同时发送 ACk 回复；如果小于则直接丢弃然后回复 ACK。等到检测到 fin 和 ack 标志时则退出循环，说明发送端开始进行第一次挥手。

四、 实验结果

该部分在路由器的丢包率为 3%，延时为 5ms，窗口大小为 15 的条件下进行测试传输，将传输 4 个给定文件，传输 helloworld 文件结果如下：

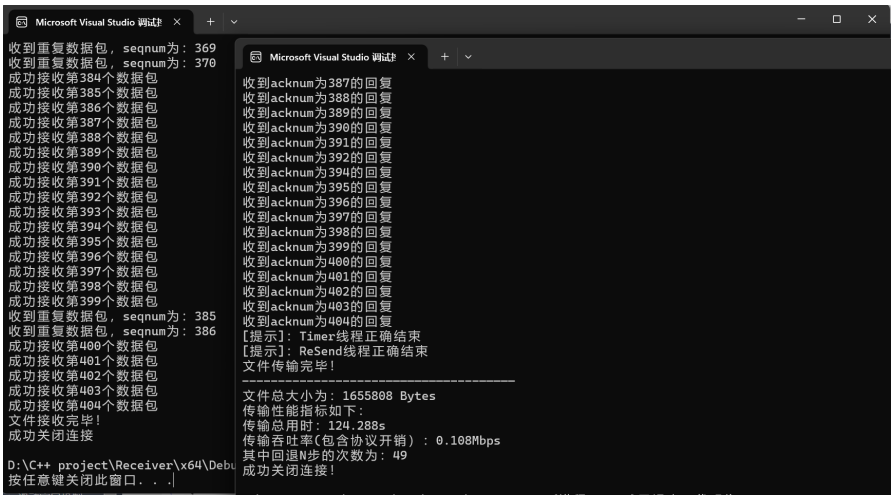

图 3: 传输 helloworld

可以看到，程序成功完成了本次传输并且正常关闭连接，在发送端目录下能看到传输的 helloworld.txt 文件，大小与源文件大小相同。传输 1.jpg 文件结果如下：

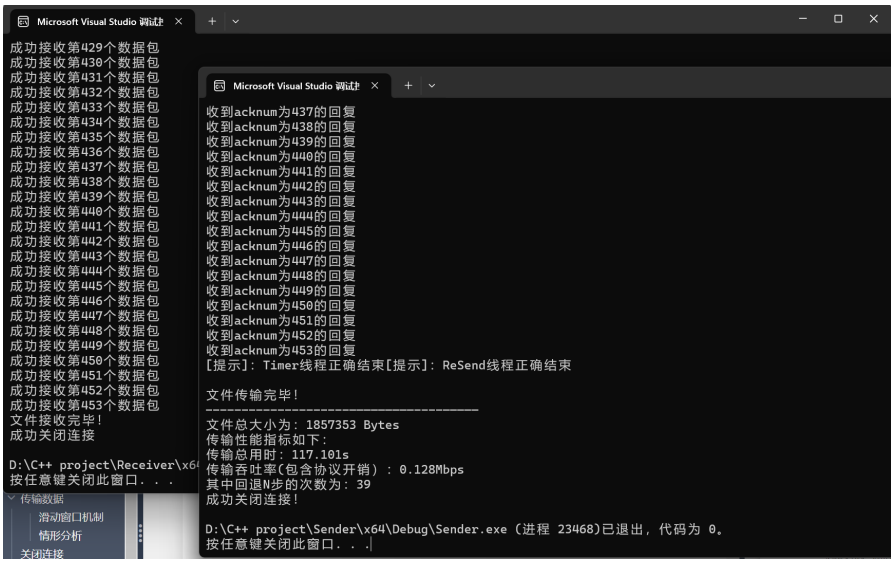

图 4: 传输图片 1

传输 2.jpg 文件结果如下：

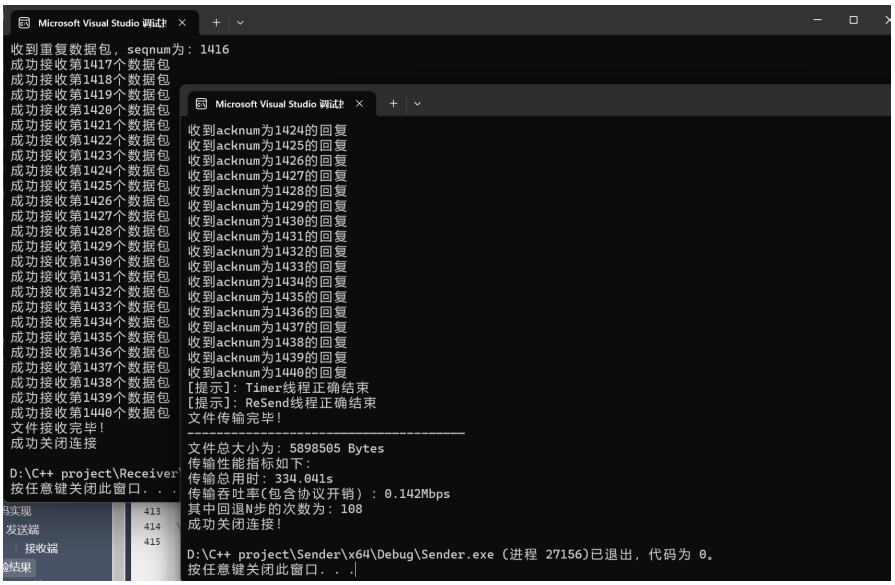

图 5: 传输图片 2

传输 2.jpg 文件结果如下：

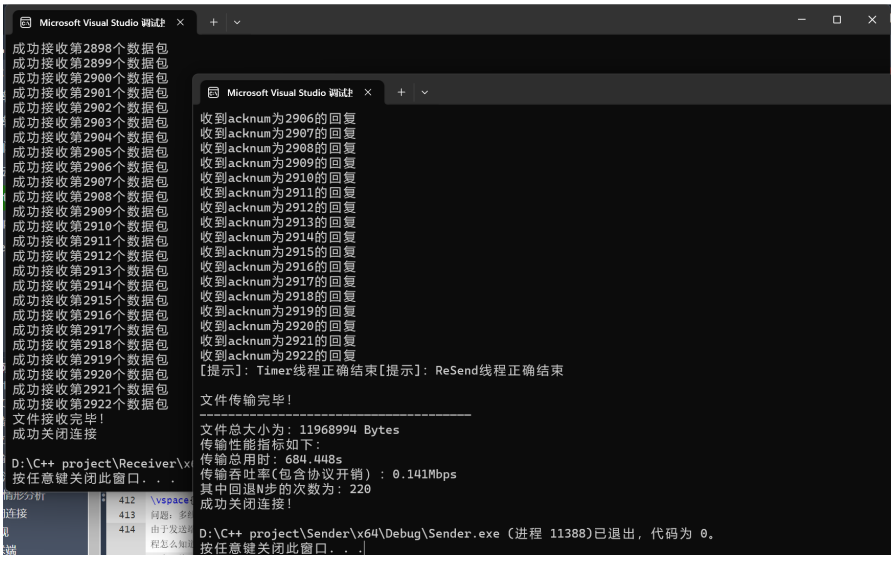

图 6: 传输图片 3

综上，图片都能够正确传输，且文件大小都与源文件大小相同。同时发现，对于 helloworld.txt 和 1.jpg 等较小文件，传输吞吐率波动较大，而对于 2.jpg 和 3.jpg 等大文件，可以从图中看到，传输速率基本都为 0.14Mbps 左右，较为稳定。

五、 遇到的问题 & 解决方法

问题：文件指针不断移动读取原始数据，重传时如何处理?

对于这个问题，反复移动文件指针可能出现一系列问题。所以我最开始尝试使用一个队列来保存 Base-NextSeq 之间的数据包，每次窗口右移的时候队头出队，主线程从原始文件中读取数据，将数据包发送并且放入队尾，同时 NextSeq 增加。重传的时候，只需遍历整个队列，并发送

给接收端即可，从而实现滑动窗口的效果。但由于采用多线程编程，用这种方法的时候程序不时出现运行错误，即窗口的左边界 Base 已经向前移动，而主线程中的 NextSeq 即右边界还没来得及向前移动，即队列可能为空后还没来得及入队就要求出队，造成运行错误。

后来我选择用一个 vector 容器来缓存下所有原始数据，窗口的移动依据 Base 和 NextSeq 的增加即可，即代码中提到的实现逻辑，不需要出队入队的复杂操作。但缺点是需要缓存下整个文件以及附加的协议开销，空间开销较大。

问题：多线程如何互相通信和交互?

由于发送端的逻辑较为复杂，我选择用包含主函数的三个线程来完成发送端的功能，但问题在于线程之间如何相互通信。Resend 线程怎么知道什么时候开始重传，Timer 线程怎么知道什么时候开始启动计时器。对于该问题，我设置了两个全局变量 Flag 标志位，两个子线程只需要在 while 循环中持续监听即可。一旦标志位改变则进行相应操作。但与此同时，另一个问题就是，这部分全局变量属于多线程共享，会进行读写操作，同时读写可能存在风险，所以在程序中我使用了互斥锁来解决这个问题。

六、 思考 & 总结

本次实验用滑动窗口机制来传输数据，在保障可靠传输的前提下实现了流量控制。通过本次实验，我对 GBN 协议有了更深入的理解。但传输效率上还有很多可以优化的地方，比如接收端和发送端使用同一大小的数据包格式，但单就一端传输一端接收的情形来说，这是不必要的，每次接收端发回 ack 的时候，4096 个字节的数据缓冲区根本没有使用，我会尝试在下一次实验中定义两个不同格式的数据包报文，从而减小在实际传输中的数据开销。