编程作业三：基于UDP服务设计可靠传输协议并编程实现

实验3-2：

在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**累计确认**，完成给定测试文件的传输

一.作业要求

在3-1的实验中基于停等协议实现了数据的传输，在3-2中，将停等协议更改为滑动窗口的控制，通过Client端和Server端窗口的不断变化来控制当前的发送进程。需要实现的功能有：

1. 多个序列号：通过序列号的值来判断窗口的动作以及数据包的顺序。
2. 发送缓冲区，接收缓冲区：在Client端与Server端添加额外的缓冲区作为滑动窗口。对于Client而言，未收到ack确认的报文将被置于缓冲区中，当经过一定时间持续未收到确认时，进行超时重传。
3. 滑动窗口：采用Go back N的模式，累计确认，
4. 有必要的日志输出。（需显示传输过程中发送端，接收端的窗口具体情况）

注：关于运行过程中缓冲窗口的滑动过程在12月2日晚讲解过程中没来的及进行介绍，在报告补充了相应的文字说明

二.滑动窗口

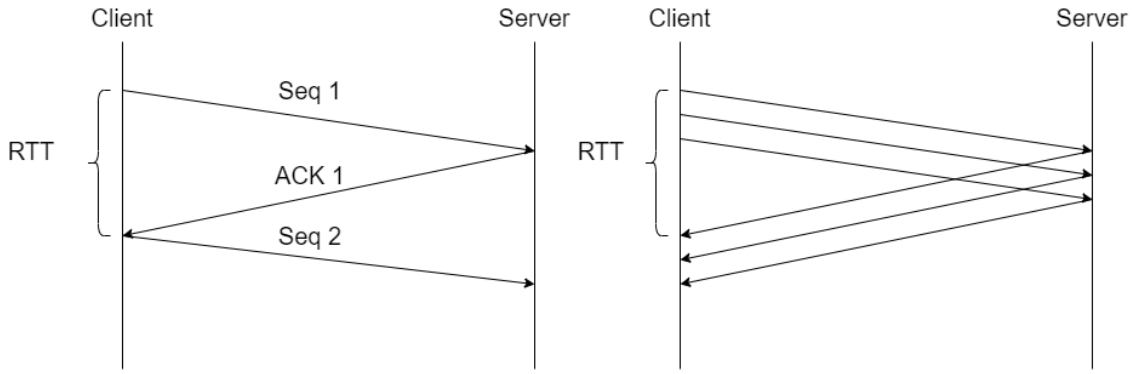
与实验3-1相比，通信协议有一定调整，增加了窗口大小的部分

序号的设置：

数据包之间的顺序关系通过seq序号来进行判定。在三次握手过程中，由Client发起的第三次握手报文，seq值为0，将通告Server端将ack值置为0，代表已确认三次握手。随后Client开始发送seq为1的数据包。Server收到该包后，将ack值更新为1，并回复ACK报文。Client端通过收到回复的ACK报文得知1号数据包已被确认，从而移动滑动窗口，

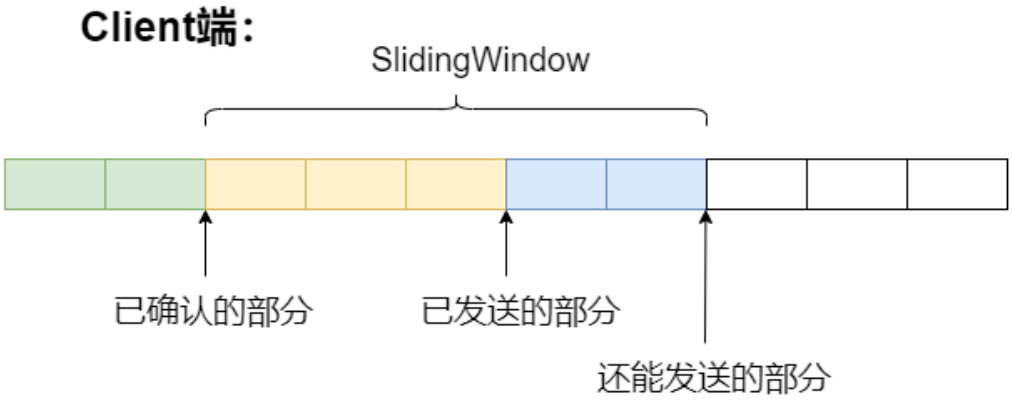
滑动窗口

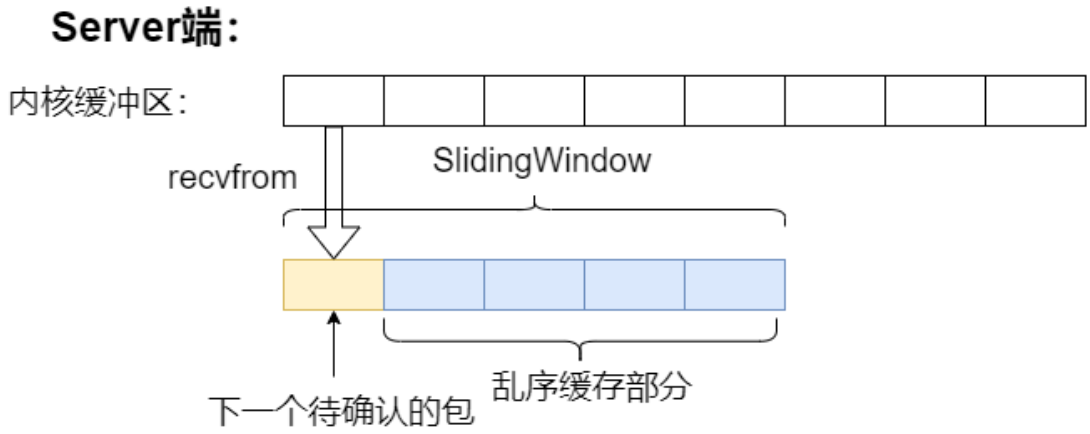
在3-1实验中采用停等机制，Client端发出一个报文后，需等待Server端回复ACK报文后才能进行下一次发送，这样的传输机制效率较低。



如图为停等机制与采取了滑动窗口的情况下报文的发送情况。左侧为停等机制，在1个RTT内只能发送一个报文。而右侧为采用流水线的情况下。，Client无需等待Server的ACK确认便可以发送下一个报文，在1个RTT内可以发送多个报文。

典型的流水线协议包括GBN算法和SR算法两种。二者都采用滑动窗口的机制来保证传输的可靠性。在具体的实现逻辑上有所区别。在本次实验中，选择以GBN为核心的可靠传输协议，并做了一定的调整。

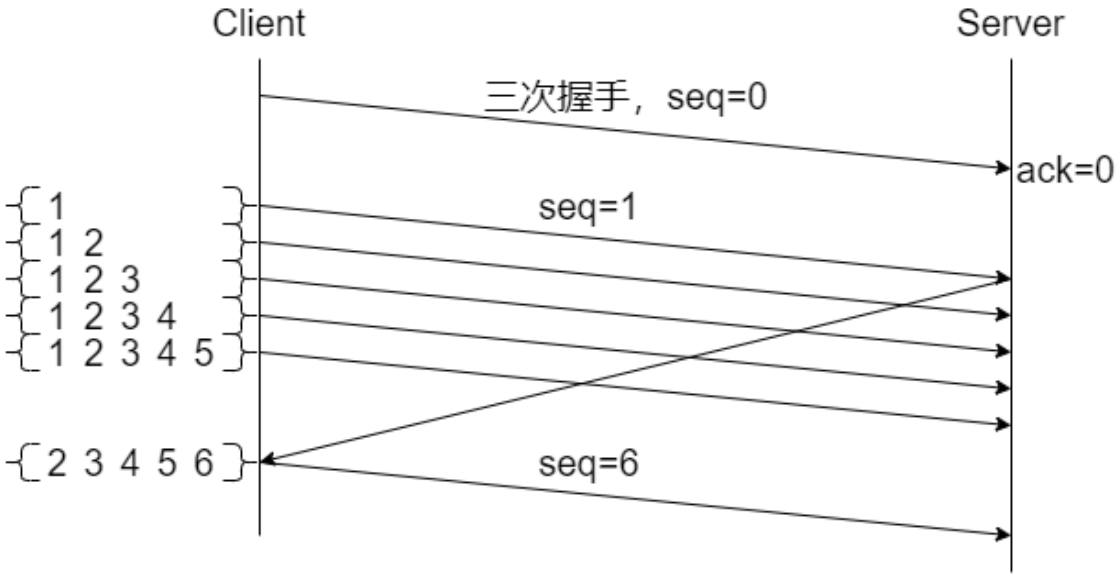




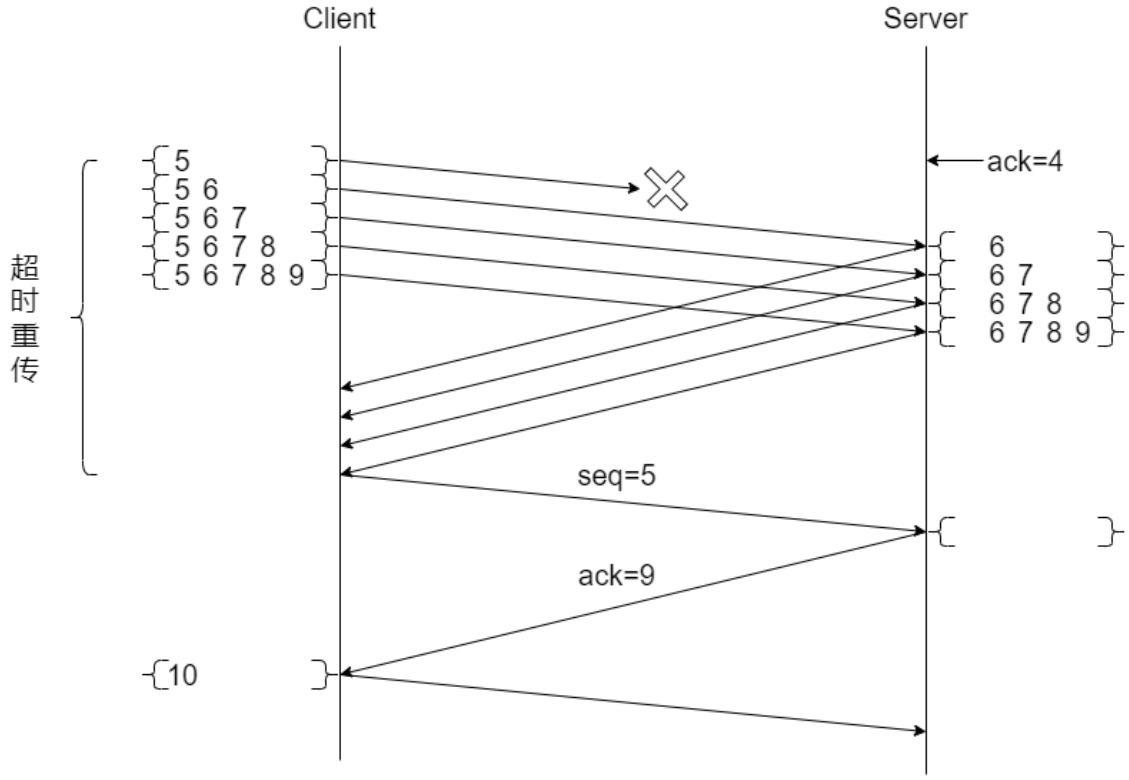
滑动窗口的设计如图所示，对于Client端，最多可以发送n个未确认的数据包，n为窗口长度，每次收到ACK确认报文时，若确认的数据报为当前窗口最左的一个数据包，则窗口向右移动，Client可继续发送下一个数据包。

对于Server端，Client端发送来的数据首先会在内核缓冲区进行缓存，这部分对程序不可见。需通过recvfrom操作从内核缓冲区中把数据加载至应用进程中。在应用中设置了接收端的窗口缓冲区。当接收到的报文的序号就是下一次需要接收的包时，程序不做处理，直接通过信息处理函数进行相关的文件操作。当收到的包序号并不为下一次需要接收的包时，代表之前出现了丢包登异常情况。Server端可以缓存乱序到达的包，存储在应用进程中。待正确的包到达后，会一并进行处理，采用**累计确认**的模式，当缓存窗口中有多个待处理的包时，会回复最大的连续包的序号。

Client端窗口的滑动：



Server端窗口的滑动：



累计确认

Server端在应用层设置了一层缓冲窗口，可以缓存一定数量乱序到达的数据包。每次回复ack时，只会回复当前已经确认的连续的序号最大的数据包。当缓冲窗口中有多个需要处理的数据包时，Server端会一并进行处理，并回复一个ACK报文，并不需要逐一进行回复，提高了传输效率。

超时重传：

Client端在每次发送一个数据包时，会将其添加至滑动窗口中，同时一并记录下来的除了完整的数据包以外，还有发送的时间。在程序中，启动了额外的一个线程用于进行超时的检测。当发现滑动窗口中有数据包在一定时间内始终没有被确认时，会主动的进行重传操作。

三.多线程的设置

在本次实验中，程序中共包含三个线程分别执行不同的任务。主线程的工作为不断读取本地的文件至发送缓冲区，进行数据的发送。在完成三次握手后，主线程会启动接收线程和重传线程。接收线程负责不断的执行recvfrom，接收来自Server的ACK，并调整窗口的滑动。重传线程的工作如上文所述，负责超时重传的工作。

主线程需要根据窗口大小，向窗口中追加数据，接收线程需要根据收到的报文，从窗口中移除已经确认的包，重传线程需要不断读取窗口中的数据。为避免三个线程直接的同时访问造成数据的错误，采用互斥锁的方式，每个线程在执行自己的任务前，首先需要等待其他进行释放该互斥锁，通过加锁操作后，才能开始执行各自的任务，通过这种方式保证了程序的正确性与稳定性。

```
//线程之间的互斥锁
std::mutex mtx;

mtx.lock()  //加锁操作
/*
    原子操作区
*/
mtx.unlock()  //解锁操作
```

接收线程recvthread

```
void recvthread()
{
    while (1)
    {
        mtx.lock();
        if (recvfrom(sockSrv, recvBuffer, sizeof(recvBuffer), 0,
(SOCKADDR*)&Server, &len) > 0)
        {
            //收到了一个消息，
            if (checkCheckSum() && checkACK)
            {
                recvlog();
                getack();
                if (recvack < slidingWindow[0].seq)
                {
                    //不在窗口范围内
                    //考虑进行重传
                }
                else
                {
                    recvWindowLen += recvack - slidingWindow[0].seq + 1;
                    while (slidingWindow[0].seq <= recvack)
                    {
                        slidingWindow.erase(slidingWindow.begin());
                        if (slidingWindow.size() == 0)
                            break;
                    }
                }
            }
        }
        mtx.unlock();
```

```
    }
}
```

重传线程retransmission

```
void retransmission()
{
    while (1)
    {
        mtx.lock();
        if (slidingWindow.size() > 0)
        {
            if (clock() - slidingWindow[0].clock > 1000)
            {
                //窗口中的第一个超时了
                //重发第一个包
                for (int i = 0; i < packetLen; i++)
                {
                    resendBuffer[i] = slidingWindow[0].buffer[i];
                }
                slidingWindow[0].clock = clock();
                sendto(sockSrv, resendBuffer, sizeof(resendBuffer), 0,
(SOCKADDR*)&Server, len);
                resendlog();
            }
        }
        mtx.unlock();
    }
}
```

四.代码介绍

在3-1的基础上，对程序的框架进行了调整，使得整个程序更便于维护，同时也降低了程序的冗余度

相关全局变量（Client端）

```
/**********关于报文信息-start**********/
char sendBuffer[packetLen]; //单个数据包的发送缓冲区
char recvBuffer[packetLen]; //单个数据包的接收缓冲区
char resendBuffer[packetLen]; //重传数据包的专门的发送缓冲区
char timeRecord[20] = { 0 }; //时间记录
int length = 0; //数据包的长度
int seq; //发出的序号
int ack; //发出的ack确认
int recvseq; //收到的seq序号
int recvack; //收到的ack确认
int window; //发送端的窗口大小
int recvWindowLen; //接收端通告的接收端窗口大小
/**********关于报文信息-end**********/
/**********关于文件系统-start**********/
vector<char> dataContent;
vector<string> files;
string path("E:\\mycode\\testfiles");
/**********关于文件系统-end**********/
```

```
//滑动窗口中数据结构DataInWIndow，
struct DataInWindow {
    bool ack = false; //是否被确认
    char buffer[packetLen]; //完整的数据报文
    int seq; //序号值
    int clock; //上次发出的时间
};
//通过vector维护的滑动窗口
vector<DataInWindow> slidingWindow;
//Client端窗口大小上限：10
int windowLength = 10;
//线程互斥锁。
std::mutex mtx;
```

相关工具函数

```
void initial()  //socket对象的设置
void calCheckSum()  //计算校验和
bool checkCheckSum()    //检查校验和
void getTime()  //获取当前系统时间
void setPort()  //设置端口号
void setIP()    //设置IP
void setseq(int newSeq) //设置发送缓冲区的seq
void getseq()   //获取接收缓冲区的seq
void setack(int newack) //设置发送缓冲区的ack
void getack()   //获取接收缓冲区的ack
void setLength(int len) //设置报文长度
void getLength()    //获取报文长度
void setWindow(int newWindow)   //设置窗口大小
int getWindow() //获取窗口大小
void clearFlag()    //清除标志位
void setAck()   //设置标志位ACK
void setSYN()   //设置标志位SYN
void setFIN()   //设置标志位FIN
void setST()    //设置标志位ST
void setOV()    //设置标志位OV
bool checkACK() //检查标志位ACK
bool checkSYN() //检查标志位SYN
bool checkFIN() //检查标志位FIN
bool checkST()  //检查标志位ST
bool checkOV()  //检查标志位OV
void sendlog()  //发送消息日志
void resendlog()    //重发消息日志
void recvlog()      //接收消息日志
```

关于多线程的设置部分在上文已经介绍

一个报文的发送

```
 //填满了一组报文，准备发送
whethersend = false;
while (!whethersend)
{
    //开始执行前需要加锁，执行结束后需要解锁
    mtx.lock();
    //判断窗口大小是否可以继续进行发送，
    if ((slidingWindow.size() < windowLength)&&recvWindowLen>0)
    {
        whethersend = true;
        //调整窗口大小
        recvWindowLen--;
        //准备发送缓冲区
        setseq(seq + 1);
        setack(0);
        setLength(messageLen);
        clearFlag();
        calCheckSum();
        //将当前的数据包添加到滑动窗口之中
        DataInWindow message;
        for (int x = 0; x < packetLen; x++)
            message.buffer[x] = sendBuffer[x];
        message.seq = seq;
        message.clock = clock();
        slidingWindow.push_back(message);
        //进行发送
        sendto(sockSrv, sendBuffer, sizeof(sendBuffer), 0, (SOCKADDR*)&Server,
len);
        sendlog();
    }
    mtx.unlock();
}
```

接收端的信息处理：

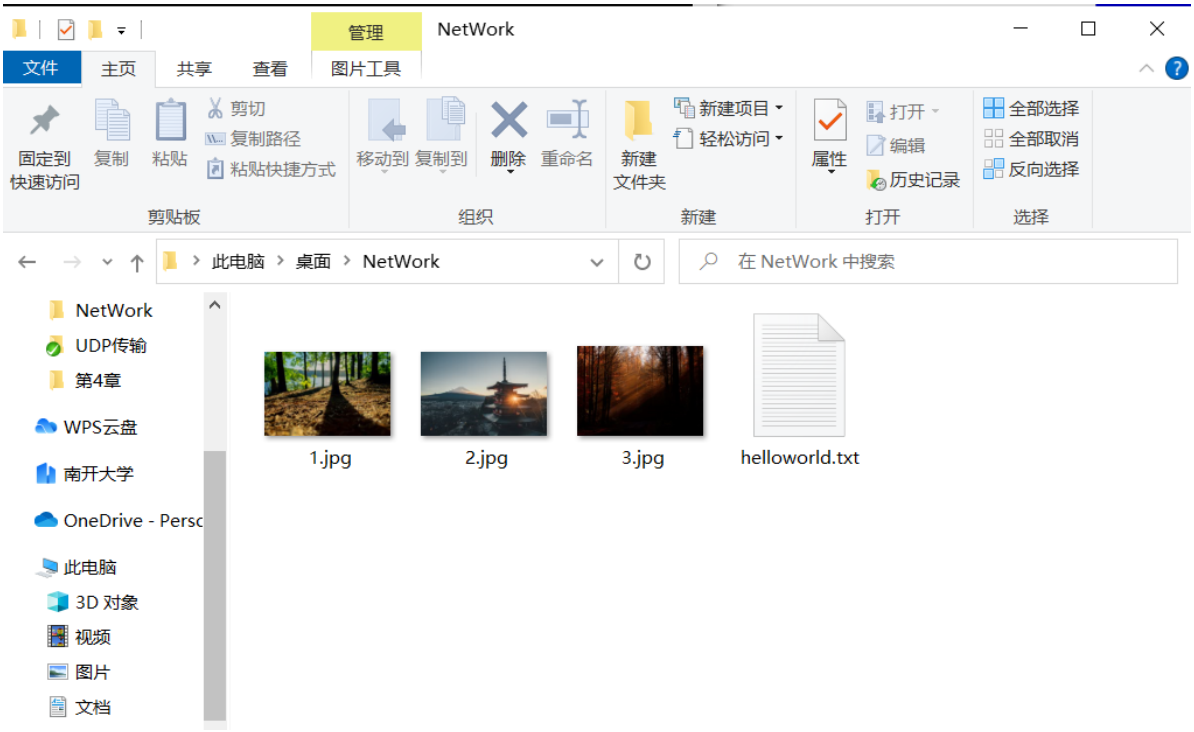
```
while (1)
    {
    //接收一条消息（实际是从内核缓冲区读取）
        recvfrom(sockSrv, recvBuffer, sizeof(recvBuffer), 0, (SOCKADDR*)&Client,
&len);
        recvlog();
    //首先检查校验和
        if (checkCheckSum())
        {
            //判断是不是挥手
            if (checkFIN())
            {
                break;
            }
            getseq();
            if (checkSeq())
            {
```

```
                //该分支表明收到的包正好就是下一次需要接收的包，直接进行处理
                messageProcessing();
                setack(recvseq);
                //同时看一下缓冲区里有没有需要处理的信息。
                while ((slidingWindow.size() > 0) && (slidingWindow[0].seq ==
ack + 1))
                {
                    for (int j = 0; j < packetLen; j++)
                    {
                        recvBuffer[j] = slidingWindow[0].buffer[j];
                    }
                    slidingWindow.erase(slidingWindow.begin());
                    getseq();
                    setack(recvseq);
                    messageProcessing();
                }
                setseq(0);
                setLength(0);
                //setWindow,通告Client端现在Server端的窗口大小，即从现在开始Client还能连
续发送几个未确认的包
                setWindow(windowLength - slidingWindow.size());
                clearFlag();
                setAck();
                calCheckSum();
                //累计确认，统一回复一个ACK
                sendto(sockSrv, sendBuffer, sizeof(sendBuffer), 0,
(SOCKADDR*)&Client, len);
                sendlog();
            }
            else
            {
                //校验码正确，但序号不正确，即之前出现了丢包，这里需要将这个包存入窗口缓存
中。
                //累计确认下，回复现在的ack
                if (recvseq <= ack + windowLength)
                {
                    //在窗口范围的进行缓存，不在窗口内的直接舍弃。
                    DataInWindow message;
                    message.seq = recvseq;
                    for (int i = 0; i < packetLen; i++)
                        message.buffer[i] = recvBuffer[i];
                    slidingWindow.push_back(message);
                    setWindow(windowLength - slidingWindow.size());
                    calCheckSum();
                    sendto(sockSrv, sendBuffer, sizeof(sendBuffer), 0,
(SOCKADDR*)&Client, len);
                    sendlog();
                }
            }
        }
        else
        {
            //校验码错误，不予理会
            clearFlag();
        }
```

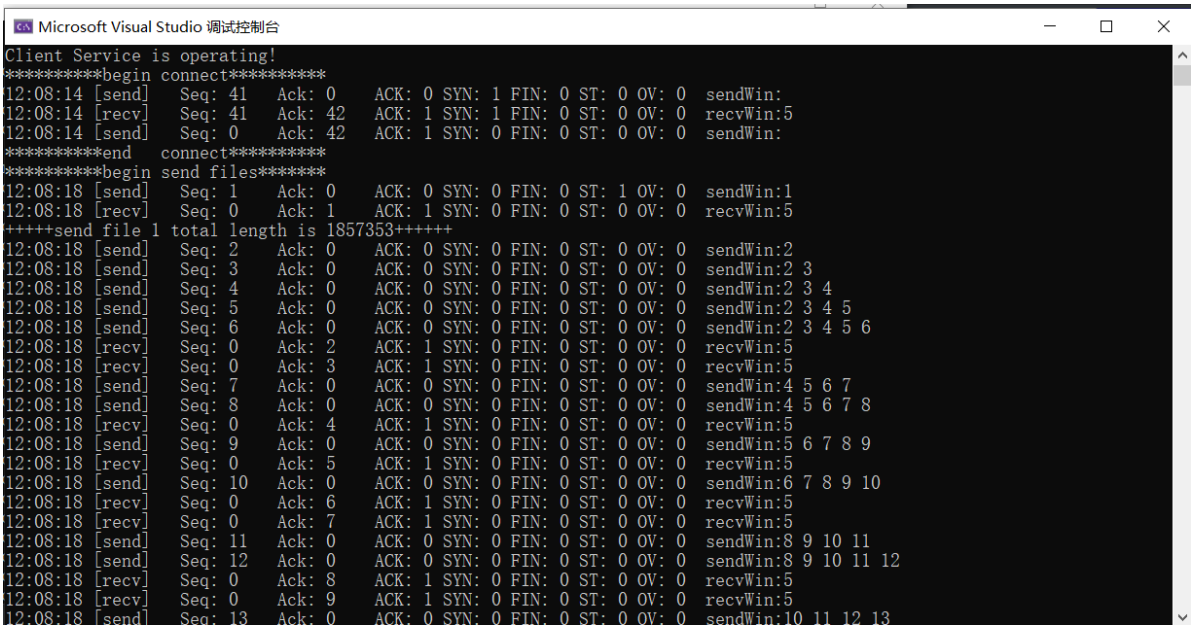
```
    }
```

五.程序运行分析（在演示时没来得及介绍完）

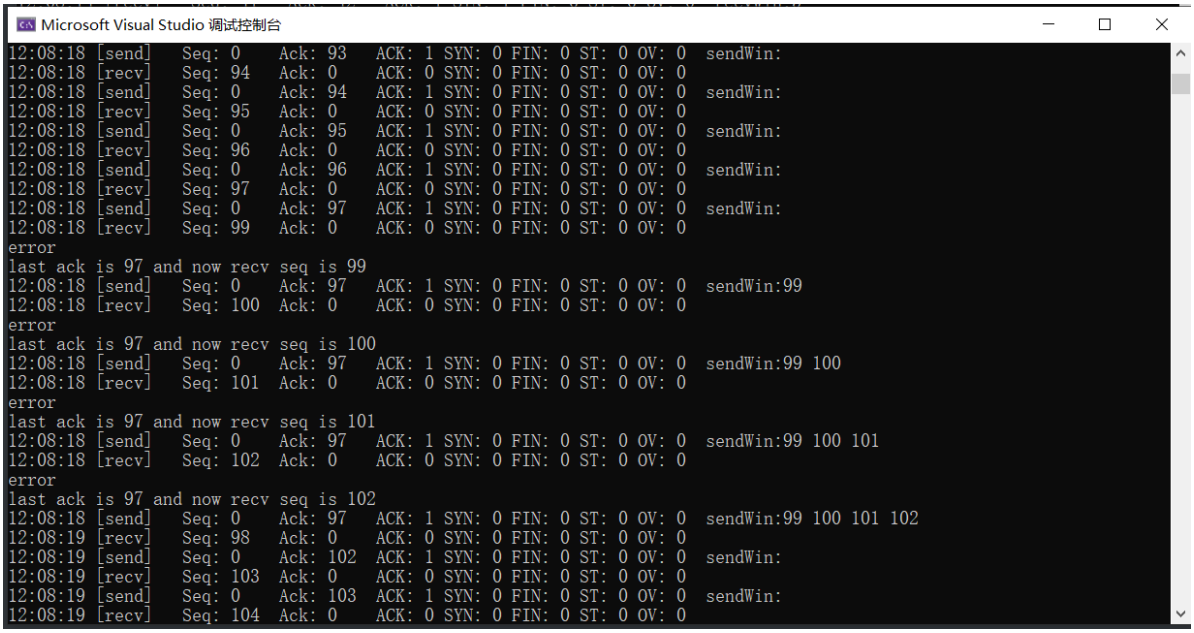
通过路由进行转发操作，同时设置丢包率为1%，即每100个数据包中丢失一个数据包。



在设置了丢包的情况下，仍可正常完成文件的发送，说明可靠传输协议工作正常



如图为Client端运行日志，可发现通过三次握手环节，Client收到了Server的通告，接收端缓冲区窗口大小为5，在文件发送过程中，可发送未确认数据包的上限就是5，在发送6号包后，此时窗口已满[2,3,4,5,6],就必须等待Server 端的确认，在确认了2号包和3号包后，窗口向前移动，变为[4,5,6,7,8]



如图为Server端的运行日志。在接收到97号包后，下一次直接接收到了99号包，说明98号包发生的丢包，此时Server会对乱序到达的99，100，101，102进行缓存，在Client一直没有收到98号包的ack，进行超时重传后，Server端收到了98号包，并会连同当前缓冲区内的99，100，101，102一起处理，并直接回复ack102

六.实验总结

通过本次实验加深了对滑动窗口的理解，应用了多线程以及互斥锁等特性，同时也优化了原本的程序框架。