计算机网络实验3-2报告

实验原理

多个序列号

实际上在实验一中就保留了多个序列号。序列号在伪首部的数据结构header以及client中的一个全局变量nextseqnum中体现：

header的成员SEQ是一个unsigned char型变量，用于表示八位二进制数，即0-255的序列号

nextseqnum是client中的一个int型全局变量，表示0-INT_MAX的序列号。在client.cpp中不直接采用一个0-255的序列号，并在累加更新到256时归零，而是采用一个int型数来记录唯一序列号的原因是方便与基序号比对，以及方便判断是否数据包已经全部发送完而分支到发送文件函数SendFileAsBinary的结束处理阶段（发送一个OVER标志并返回1）。

易知，对于同一个数据包，它们的关系如下：

`nextseqnum%256=header.SEQ`

这个转换关系再makepkt时可以用到，例如一个文件发送到了第257个包，那么nextseqnum=257，它的伪首部的SEQ就是257%256=1。

为方便对比说明，这里同样给出基序号的处理方法：

client中维护一个全局变量base，由于不用放在伪首部中转发，它的数据类型直接定义为int型（方便起见，省去复杂指针转换），但它只能表示0-255的int数，因为它对应了GBN发送端有限状态机中的：

`base=getacknum(rcvpkt)+1`

因为定义的header数据结构中对序列号的ack，也即序列号SEQ，只能表示8位二进制数即0-255，base的表示范围只有0-255（应为1-256，但实际实现中做了进位处理）。这时要与nextseqnum进行比较，算出滑动窗口的长度时就会出现如下问题：

假设一个文件分为500个包发送，最大的窗口大小为16，那么假设某时发送端的

base=255，nextseqnum=260，那么窗口大小=5，可以接受；

这时发送端接到了一个新的包，那么base要变成它的伪首部的ack+1，那么由于它的伪首部SEQ只能表示8位，假设它的ack正好为255，即正好是发送端已验证的所有包的下一个包，那么它的base就应为256。但由于进位机制，它只能表示0-255，这时base会运算=0。

这时比较滑动窗口，会发现nextseqnum-base=260-0=260>32。

为了解决这个问题，发送端中维护了一个全局int变量：basejwnum。它表示当前base进位过的次数。那么，如果我们记实际的基序号（即如果文件分500个包，ack了255个包，那实际的基序号就是256而不是base=0）为actbase，那么有

`actbase=base+256*basejwnum`

这样在上面的例子比较滑动窗口大小时，在base变化进位时，basejwnum由0变为1，那么滑动窗口大小是

nextseqnum-(base+256*basejwnum)=260-0+256=4

当然符合要求。

在具体实现中，这个数的处理逻辑如下：

当发送端recv一个ack信息中，提取出它的SEQ，如果它由255变为0，那么表示接收端验证的信息个数满了256个，此时发送端收到这个ack消息，对其base进行更新，由于base的含义就是已经验证收到的序列号（由0开始则+1），发送端的basejwnum就要+1

```
recvfrom(...);
if((int(header.SEQ)+1)%256<base){basejwnum++;}
```

这种方式需要满足窗口大小<256即header.SEQ能表示的最大大小，因为如果最大窗口大小大于256，base和ack_num相差超过256，例如发送端的base=1，收到的ack（SEQ）为0，但实际上它对actbase之后的256,512,768...等等个包发回的ack，其8位的SEQ都是0，但此时发送端的basejwnum只能判断为+1，实际的base就会比记录的base多出数个256，这样在后面就会出问题。（实际上在gbn协议中窗口大小不应超过256/2-1即127，这一点使上述条件恒得到满足）

实际上也有解决办法，对于这样一个课程作业的程序，在伪首部中加几位来记录八位序列号进位的次数就可。但还是保留了如上的设计。

在程序中根据需要来灵活运用这几个变量，例如状态机中的

`if(base=nextseqnum){...}`

实际的代码是

`if(base+256*basejwnum==nextseqnum){...}`

例如makepkt时

`header.SEQ=(unsigned char)(nextseqnum%256);`

发送缓冲区、接收缓冲区

sendto、recvfrom函数中的缓冲区指针

由于完整的需要发送的文件被读取成二进制数据，存放在char型数组中，发送时也是根据指针来发送，例如发送第k个包，它的长度为pktlen，并假设这之前的每个包长度都为PKT_SIZE，那么发送函数为

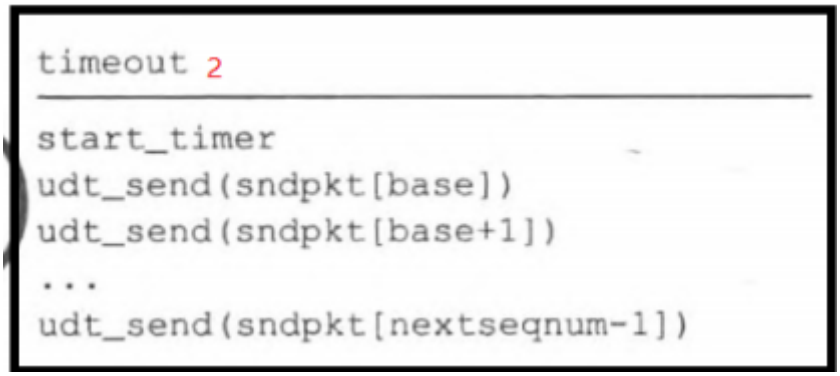
```
char* nowPktPointer=fullData+k*PKT_SIZE;
//这里校验和已算好
memcpy(nowbuffer,&header,sizeof(header));
memcpy(nowbuffer+sizeof(header),nowPktPointer,pktlen);

sendto(clntSock, buffer, sizeof(header)+pktlen, 0, (sockaddr*)&servAddr,
servAddrLen)
```

sendto的第二个参数为这个缓冲区buffer的开头的指针，它往后sizeof(header)个字节（按字节编址）为伪首部，再往后pktlen个字节为要发生的数据，要发送的数据的指针由一个指针复制而来，这个指针就是完整数据的开头指针+当前发送包的序号k乘以前面所有包的同一大小。

这一部分实际上在3-1实验中就有同样的实现。

状态机中的缓冲区“数组“

这一发送缓冲区的概念实际上对应着有限状态机中的sndpkt数组。



makepkt时会将pkt放在这个数组，也即缓冲区内，若有超时重传的情况则更新计时器并发送数组中index为base到nextseqnum的项。但若完全按照这样来存储，对数组或链表的实现与操作繁琐，并且如果及时移除base之前的数组项，多花费的内存空间大小最大会达到

最大窗口大小*每个包（包括数据和伪首部）的大小

若一开始就new一个最大包数的数组来按序号插入数据包，那么会多花费

文件大小+最大包数*伪首部大小

的内存空间。

鉴于以上弊端，同样采用了指针的方式来实现发送缓冲区。

这样的实现很简单，要超时重传base到nextseqnum-1时，若每个包大小相同，它们有如下关系：

base->base+pktlen->base+2*pktlen->...->nextseqnum-1-pktlen->nextseqnum-1

假设从第一个base的指针开始，与上面同样的方法可以得到它在完整数据的char数组的指针

```
char* nowPktPointer=fullData+base*PKT_SIZE;
//这里校验和已算好
memcpy(nowbuffer,&header,sizeof(header));
memcpy(nowbuffer+sizeof(header),nowPktPointer,pktlen);

sendto(clntSock, buffer, sizeof(header)+pktlen, 0, (sockaddr*)&servAddr,
servAddrLen)
```

对于每个之后的指针，迭代更新nowPktPointer即可

```
char* nowPktPointer1=fullData+base*PKT_SIZE;
char* nowPktPointer2=fullData+(base+1)*PKT_SIZE;
...
char* nowPktPointern=fullData+(nextseqnum-1)*PKT_SIZE;
```

用一个for循环即可解决问题

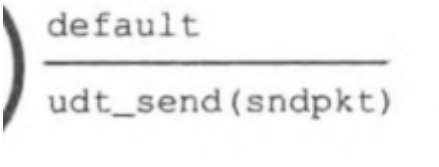
```
int resend_index=base+256*basejwnum;
//通过状态机可知，从base到nextseqnum-1
//udt_send from base to nextseqnum-1
for(resend_index;resend_index<nextseqnum;resend_index++)
{
    char* nowPktPointer1=fullData+resend_index*PKT_SIZE;
    make packet然后发送
}
```

当然对于最后一个包，其长度可能不满最大数据包的长度，需要额外处理。

如果额外实现中，每个包的大小不一，通过额外的一个数组来记录0到最大包数的分别的包大小即可，这样的开销也比直接记下缓冲区的char二维数组小。

接收缓冲区

接收缓冲区除了recvfrom和1sendto中的参数缓冲区指针以外，还可能会重传一个pkt



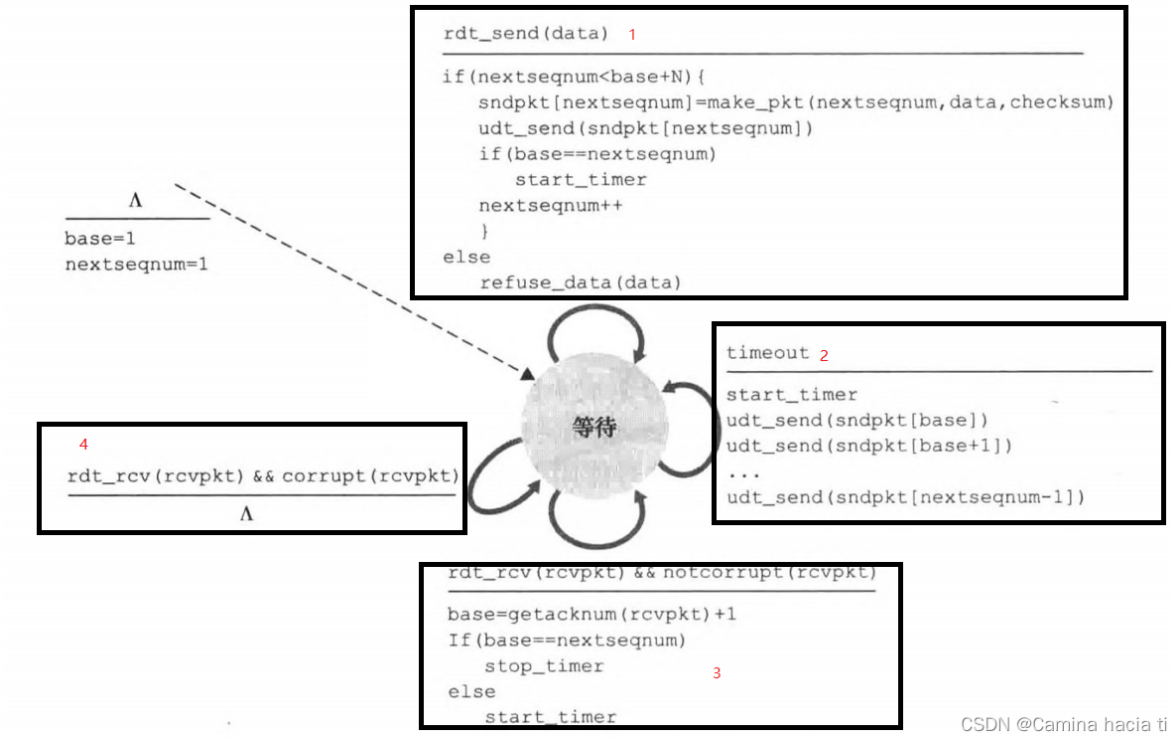
由于接收方只发送伪首部，建立一个HEADER数据结构来缓存它即可。

GBN

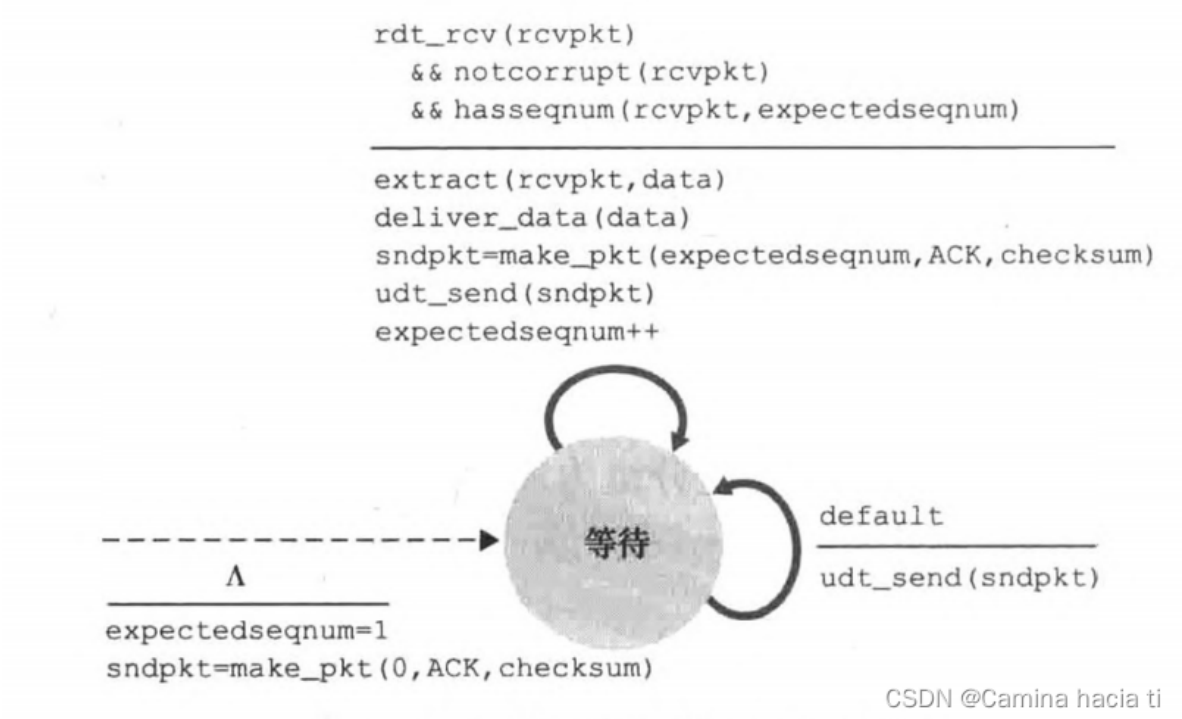
GBN协议的主要特点是发送方不需要等待上一个帧的ack，可以连续发送帧。发送端在发完一个数据帧后，连续发送若干个数据帧，即使此过程中接到了应答帧也可以继续发送。发送方每发完一个帧设置一个定时器，在超时时间内为收到确认帧，则要收到它之后的到最近发送的所有的数据帧。

接收端只按顺序验证帧，若连续收到好几个正确帧，也只对最后一个发送ack。这样发送端接收到一个ack就可以判断它之前的所有帧均被收到。

发送端FSM



接收端FSM



其他

其他实验原理，包括三次握手四次挥手等，其实现均与上次实验相同，这里不再重复展示。

代码实现

三次握手和四次挥手、差错检测、伪首部等与上次实验相同，这里贴出代码并表明与上次实验相同方便阅读。

初始化（与上次实验相同）

使用UDP进行实现。

C语言的UDP编程需要先初始化DLL、

```
WSADATA wsaData;
WSAStartup(MAKEWORD(2, 2), &wsaData);
```

具体的创建过程实际上与第一次实验基本相同，不过是替换成了UDP。

```
SOCKET sock;
SOCKADDR_IN sockAddr;
sock = socket(AF_INET, SOCK_DGRAM, 0);//使用UDP
...
//**************************本地socket************************
    //绑定套接字
    char clntIP[20];
    string mylocalip="127.4.5.6";
    myippointer=clntIP;
    myippointer = strcpy(myippointer,mylocalip.c_str());
```

```
    SOCKADDR_IN addrLocal;// 发送端IP和端口
    memset(&addrLocal, 0, sizeof(SOCKADDR_IN));
    addrLocal.sin_family = AF_INET;
    addrLocal.sin_addr.s_addr = inet_addr(clntIP);
    addrLocal.sin_port = htons(9876);
    if(bind(sock, (SOCKADDR*)&addrLocal, sizeof(addrLocal))==-1)//绑定
    {
        perror("BIND FAILED!");
        exit(-1);
    }
```

例如如上的代码初始化了一个socket和sockaddr，并将他绑定到本地，也即编译后的可执行文件运行后的ip和端口就是127.4.5.6:9876。需要转发到路由器，同时对路由器的sockaddr进行同样操作的初始化，不要绑定，并在之后的sendto和recvfrom中调用这个路由器的sockaddr即可。

程序最后使用WSACleanup函数终止DLL的使用即可。

三次握手&四次挥手（与上次实验相同）

根据三次握手的原理来实现。

例如，客户端接收第一次握手：

```
while(1)
    {//第一次recv不重传，不用计时
        if(recvfrom(servSock,buffer,sizeof(header),0,
(sockaddr*)&clntAddr,&clntAddrLen)<0)
        {
            addtoservlog("第一次握手接收错误",myippointer);
            printf("第一次握手接收错误\n");
            return -1;
        }
        addtoservlog("第一次握手接收成功，进行校验",myippointer);
        printf("第一次握手接收了\n");
        memcpy(&header,buffer,sizeof(header));
        u_short checksum_from_recv=header.checksum;
        header.checksum=0;
        calc_chksum_rst=CalcChecksum((u_short*)&header,sizeof(header));
        if(header.flag==HSK1OF3&&calc_chksum_rst==checksum_from_recv)
        {
            addtoservlog("第一次握手校验成功",myippointer);
            printf("第一次握手校验成功\n");
            break;
        }else{
            addtoservlog("第一次握手校验失败",myippointer);
            printf("校验码错误\n");
        }
    }
    printf("第一次握手结束\n");
    addtoservlog("第一次握手完毕",myippointer);
```

循环调用recvfrom来接收发送端发来的第一次握手消息。使用memcpy将伪首部对应位按位复制到当前的一个header实例中，取出它的校验和，将收到的buffer的校验和位全部格式化为0，之后再计算校验和，并与取出的校验和进行比较。如果校验和比较成功，且经过flag的比较确实是第一次握手该有的flag（一个宏定义，见**伪首部**一节），即syn=1，ack=0，则校验成功，进入下一步，否则继续循环监听。addtoservlog是记录到日志文件的函数。

剩余部分的实现基本用了这样的计算校验和并比较的逻辑，并用了相同的一些api（sendto、recvfrom、memcpy等）。

对于需要**超时重传**的部分，开启调用clock()，计算两个clock()返回值的差，与一个阈值比较作为计时器，在循环调用recvfrom监听的过程中，循环体时刻计算是否超时，超时则重传需要重传的数据包并更新计时器即可。例如服务端接收第三次握手：

```
while(recvfrom(servSock,buffer,sizeof(header),0,
(sockaddr*)&clntAddr,&clntAddrLen)<=0){
        if(clock()-now_clocktime>MAX_TIME)
        {//超时重传
            header.flag=HSK2OF3;
            header.checksum=0;
            calc_chksum_rst=CalcChecksum((u_short*)&header,sizeof(header));
            memcpy(&header,buffer,sizeof(header));
            if (sendto(servSock,buffer,sizeof(header),0,(sockaddr*)&clntAddr,
clntAddrLen) == -1)
            {
                addtoservlog("第二次握手超时重传失败",myippointer);
                return -1;
            }
            now_clocktime=clock();
            printf("第三次握手超时，第二次握手重传发送成功\n");
            addtoservlog("第三次握手超时，重传发送成功",myippointer);
        }
    }
```

制作伪首部，计算校验和并写入，sendto目标socket，更新计时器。

客户端和服务端的其余部分实现的api和逻辑基本与上述相同，按照三次握手四次挥手来实现即可。具体的代码和一些特殊说明具体可见打包的代码和注释，这里不再进行赘述。

差错检测（与上次实验相同）

```
u_short CalcChecksum(u_short* bufin,int size)
{
    int count = (size + 1) / 2;
    //u_short* buf = new u_short[size+1];
    u_short* buf = (u_short*)malloc(size + 1);
    memset(buf, 0, size + 1);
    memcpy(buf, bufin, size);
    u_long sum = 0;
    while (count--) {
        sum += *buf++;
        if (sum & 0xffff0000) {
            sum &= 0xffff;
            sum++;
        }
    }
    return ~(sum & 0xffff);
}
```

输入一个缓冲区char型指针和长度，返回一个16位（u_short)的反码和。

伪首部

```
struct HEADER
{
    u_short checksum=0;
    u_short datasize=0;
    //为把四个用到的flag“压缩”成四位
    //用一个unsigned char记录
    //使用低4位
    //分别是OVER FIN ACK SYN
    //其中over是指一个文件发送完毕后发送的标志位
    unsigned char flag=0;
    unsigned char SEQ=0;
    HEADER()
    {
        checksum=0;
        datasize=0;
        flag=0;
        SEQ=0;
    }
};
```

使用两个16位的u_short，分别存储校验和和数据的长度。再用三个8位的unsigned char存储flag、ack和SEQ，flag的说明如上。并声明了构造函数。

```
#define HSK1OF3 0x1 //SYN=1 ACK=0
#define HSK2OF3 0x2 //SYN=0,ACK=1
#define HSK3OF3 0x3 //SYN=1 ACK=1
#define WAV1OF4 0x4 //SYN=0 ACK=0 FIN=1
#define WAV2OF4 0x2 //SYN=0 ACK=1 FIN=0
#define WAV3OF4 0x6 //SYN=0 ACK=1 FIN=1
#define WAV4OF4 0x2 //SYN=0 ACK=1 FIN=0
#define OVERFLAG 0x8 //1000,OVER=1
#define INITFLAG 0x0 //发送数据包，全部为0
```

并宏定义了一些参数，都是flag来用。例如第三次握手时SYN=1，ACK=1，那么“翻译”到unsigned char的对应位上就是16进制的3（00000011）。

发送端

发送端发送函数的调用需要对各种变量进行声明以及必要的初始化，包括上次的header、校验码结果calc_checksum_rst以及这次的base、basejwnum、nextseqnum等。

发送

使用非阻塞，首先是发送函数的代码。

发送包时首先需要验证是否已经全部发送完，否则跳过发送部分，直接跳转接收部分看是否还有未接收完的ack。

```
if(nextseqnum<pktnum){
            //如果“下一个”序列号等于包数
            //而包数从0开始，实际上就是
            //下一个序列号超过最大包数
            //这时就不能再发，而是等待剩余的接收端ack

            if(判断窗口大小合格)
            ...makepkt，sendpkt
            else 记录日志，不发送
}
```

接下来判断窗口大小是否合规，即nextseqnum-实际的base是否小于设置的最大滑动窗口大小WINDOW_SIZE。是则makepkt并发送，否则记录日志并且拒绝data

makepkt的实现与上次的基本相同。

```
if(nextseqnum==pktnum-1){
                //最后一个
                pktlen=dataLen-(pktnum-1)*MAX_BUFFER_SIZE;
            }else{
                pktlen=MAX_BUFFER_SIZE;
            }
            header=HEADER();
            header.datasize=pktlen;//此时pktlen为不加头部的大小
            header.SEQ=(unsigned char)(nextseqnum%256);
            header.checksum=0;
            header.flag=0;
            header.ack=0;
            //初始化为0
            char* nowPktPointer=fullData+nextseqnum*MAX_BUFFER_SIZE;
            memcpy(buffer,&header,sizeof(header));
            memcpy(buffer+sizeof(header),nowPktPointer,pktlen);

 calc_chksum_rst=CalcChecksum((u_short*)buffer,sizeof(header)+pktlen);
            header.checksum=calc_chksum_rst;
            memcpy(buffer,&header,sizeof(header));
            memcpy(nowbuffer,&header,sizeof(header));
            memcpy(nowbuffer+sizeof(header),nowPktPointer,pktlen);
            printf("\n发送校验和=%d\n",calc_chksum_rst);
            //udt_send(sndpkt)
            if(sendto(clntSock, buffer, sizeof(header)+pktlen, 0,
(sockaddr*)&servAddr, servAddrLen)<0){
            printf("udp发送错误\n");
            addtoclntlog("[ERR ]UDP包发送错误",myippointer);
            continue;
            }
            memset(message,0,sizeof(message));
            sprintf(message,"[INFO]成功发送 %d bytes数据，
nextseqnum=%d",pktlen,nextseqnum);
            printf("%s\n",message);
            addtoclntlog((const char*)message,myippointer);
            sprintf(message,"[INFO]实际基序号base=%d，滑动窗口大小
=%d",base+256*basejwnum,nextseqnum-(base+256*basejwnum));
            printf("%s\n",message);
            addtoclntlog((const char*)message,myippointer);
            sprintf(message,"[INFO]发送UDP包的校验和checksum=%u，伪首部的8位
SEQ=%d",calc_chksum_rst,int(header.SEQ));
```

```
            printf("%s\n",message);
            addtoclntlog((const char*)message,myippointer);
            addtoclntlog("[MESG]UDP包发送成功",myippointer);
            printf("udp包发送成功\n");
```

根据gbn的fsm，发送成功时如果实际的base=nextseqnum需要开启计时器。同时需要更新nextseqnum，并记录日志

```
 //start timer
if(base+256*basejwnum==nextseqnum){
    addtoclntlog("[MESG]base==nextseqnum,开启计时器",myippointer);
    now_clocktime=clock();
    }
    //if(nextseqnum==pktnum-1){break;}
    //直接break的话，实际上由于recv在send之后，send完但是没有recv完
    //需要等到所有都recv完，即base=nextseqnum
     nextseqnum++;
     addtoclntlog((const char*)message,myippointer);
     sprintf(message,"[INFO]更新nextseqnum=%d",nextseqnum);
     printf("%s\n",message);
     addtoclntlog((const char*)message,myippointer);
```

接下来判断窗口如果不合规，则拒绝data。

```
else{
拒绝data，不进行发送
记录日志
}
```

接收

使用非阻塞模式。

```
u_long mode = 1;
ioctlsocket(clntSock, FIONBIO, &mode);
```

首先是状态机中redt_rcv && notcorrupt的部分

这部分的校验工作与上次实验基本相同，对接收到的放入buffer，取出伪首部的seq和checksum，并checksum置0后计算checksum并比较。

如果校验码不通过，根据状态机，回到等待状态。

```
if(calc_chksum_rst!=checksum_from_recv1){
                //step 4 in GBN FSM
                // i.e. rdt_rcv && corrupt
                // go back to waiting status, i.e. do nothing indeed and
continue
                printf("%u!=%u,回传校验码不通过
\n",calc_chksum_rst,checksum_from_recv1);
                addtoclntlog("[ERR ]确认信息校验码校验不通过",myippointer);
                continue;
            }
```

如果没有continue，进入下面分支，则是校验码通过。根据状态机，更新base，并根据实验原理中的解释，这里更新basejwnum。

```
if((int(temp1.SEQ)+1)%256<base){basejwnum++;}
            //seq+1为0-255，如果到了256要变回0

            base=(int(temp1.SEQ)+1)%256;
            memset(message,0,sizeof(message));
            sprintf(message,"[INFO]base变量更新=%d，实际的基序号
base=%d",base,base+256*basejwnum);
            printf("%s\n",message);
```

实际的base=nextseqnum时需要stop_timer，实际上并没有显式的stop_timer函数，而是在此分支下先前的starttimer中记录的clock()返回值在后边用不上。更多的一些解释见代码注释

```
 if(base==nextseqnum%256)
            {
                //TODO:stop_timer
                //stop_timer即是不再超时重传，
                //实际上，也就是可以直接跳过下面代码的分支
                //包括这里的else now_clocktime=clock();
                //和下面的超时重传即FSM中的timeout
                //故回到原始状态，模式设为阻塞
                //continue等待即可

                //实际上需要判断，此时全部收完，那么就跳出，发送结束消息
                //否则就是没有收完文件的所有包，但当前发送端已经发送的包
                //已经ack完了
                //继续发送，并等待ack
                if(base+basejwnum*256==pktnum){
                    addtoclntlog("[MESG]所有信息已发送，超时不重传",myippointer);
                    break;
                }
                else{
                    memset(message,0,sizeof(message));
                    sprintf(message,"[INFO]实际的基序号base=%d，nextseqnum=%d，包总
数=%d",base+256*basejwnum,nextseqnum,pktnum);
                    printf("%s\n",message);
                    addtoclntlog((const char*)message,myippointer);
                    printf("所有包验证完毕，继续发送\n");
                    addtoclntlog("[MESG]已发送的所有包验证完毕，继续发送与验
证",myippointer);
                    continue;
                }


            }else{
                addtoclntlog("[MESG]开启计时器",myippointer);
                now_clocktime=clock();
            }
```

非阻塞模式，if(recv)一直没有接受的话会进入超时重传。根据状态机，这一部分需要重传base到nextseqnum-1的所有包。根据**实验原理**中**状态机中的缓冲区“数组”**一节的阐述，使用index，以指针操作进行makepkt并发送。

```

```

```
else{
            //TODO 应把超时重传放到这里
            //step 2 in GBN FSM
            //timeout
            if(clock()-now_clocktime>MAX_TIME){
                now_clocktime=clock();
                //base经过%256处理,但nextseqnum没有
                int resend_index=base+256*basejwnum;
                //通过状态机可知，从base到nextseqnum-1
                //udt_send from base to nextseqnum-1
                for(resend_index;resend_index<nextseqnum;resend_index++)
                {
                    if(resend_index==pktnum-1){
                        //最后一个
                        pktlen=dataLen-(pktnum-1)*MAX_BUFFER_SIZE;
                    }else{
                        pktlen=MAX_BUFFER_SIZE;
                    }
                    header=HEADER();
                    header.datasize=pktlen;//此时pktlen为不加头部的大小
                    header.SEQ=(unsigned char)(resend_index);
                    header.checksum=0;
                    header.flag=0;
                    header.ack=0;
                    //初始化为0
                    char* nowPktPointer=fullData+resend_index*MAX_BUFFER_SIZE;
                    memcpy(buffer,&header,sizeof(header));
                    memcpy(buffer+sizeof(header),nowPktPointer,pktlen);

 calc_chksum_rst=CalcChecksum((u_short*)buffer,sizeof(header)+pktlen);
                    header.checksum=calc_chksum_rst;
                    memcpy(buffer,&header,sizeof(header));
                    memcpy(nowbuffer,&header,sizeof(header));
                    memcpy(nowbuffer+sizeof(header),nowPktPointer,pktlen);
                    //udt_send(sndpkt)
                    if(sendto(clntSock, buffer, sizeof(header)+pktlen, 0,
(sockaddr*)&servAddr, servAddrLen)<0){
                    printf("udp发送错误\n");
                    addtoclntlog("[ERR ][超时重传]udp包发送错误",myippointer);
                    continue;
                    }
                    memset(message,0,sizeof(message));
                    sprintf(message,"[INFO][超时重传]成功发送 %d bytes数据，
nextseqnum=%d",pktlen,nextseqnum);
                    printf("%s\n",message);
                    addtoclntlog((const char*)message,myippointer);
                    sprintf(message,"[INFO][超时重传]当前实际基序号base=%d，滑动窗口大
小=%d",resend_index,nextseqnum-(base+256*basejwnum));
                    printf("%s\n",message);
                    addtoclntlog((const char*)message,myippointer);
                    sprintf(message,"[INFO][超时重传]发送UDP包的校验和checksum=%u，
伪首部的8位SEQ=%d",calc_chksum_rst,int(header.SEQ));
                    printf("%s\n",message);
                    addtoclntlog((const char*)message,myippointer);
                    addtoclntlog("[MESG][超时重传]UDP包发送成功",myippointer);

                }
            }
```

```
        }
```

最后进行一些收尾的工作即可，例如发送一个结束的flag。

接收端

同样进行初始化操作。首先先初始化好一个ack0的sendpkt，应对一开始就接收不对的局面。

```
HEADER sndpkt;//只发送伪首部
    // make sndpkt
    servSeq=0;//初始化seq为0
    sndpkt.SEQ=servSeq;
    u_short calc_chksum_rst;
    calc_chksum_rst=CalcChecksum((u_short*)&sndpkt,sizeof(sndpkt));
    sndpkt.checksum=calc_chksum_rst;
```

循环调用接收函数，接收则进行校验，这一部分与上次实验基本相同，且上面发送端中也有相似的代码，不再贴出。这里需要判断接收的是否为一个over的标志，是则表示发送端所有的均已发送，直接退出循环，进行收尾。

如果不是over的flag，且校验码校验正确，接下来就判断seqnum是否是期望的那个。是则根据状态机，解包、处理、发送ack，并更新expectedseqnum。

```
//解包、发送ack：
if(servSeq==(int)header.SEQ)
            {
                memset(message,0,sizeof(message));
                sprintf(message,"[INFO]应收到的SEQ=%d，收到UDP包的SEQ=%d",servSeq,
(int)header.SEQ);
                addtoservlog((const char*)message,myippointer);
                addtoservlog("[INFO]SEQ值验证正确",myippointer);
                header=HEADER();
                header.SEQ=(unsigned char)servSeq;
                // header.flag=123;
                header.checksum=0;
                //sndpkt=makepkt(expextedseqnum)
                //udt send
                //seq++
                calc_chksum_rst=CalcChecksum((u_short*)&header,sizeof(header));
                header.checksum=calc_chksum_rst;
                memcpy(buffer,&header,sizeof(header));
                memcpy(&sndpkt,&header,sizeof(header));
                if (sendto(servSock,buffer,sizeof(header),0,
(sockaddr*)&clntAddr, clntAddrLen) == -1)
                {
                    printf("ack发送错误\n");
                    continue;
                    //return -1;
                }
                printf("校验码和SEQ均校验成功，确认消息发送成功\n");
                printf("发送的校验码=%d\n",calc_chksum_rst);
                addtoservlog("[MESG]校验码和SEQ均校验成功，确认消息发送成
功",myippointer);
                memset(message,0,sizeof(message));
                sprintf(message,"[INFO]发送的校验码chechsum=%d",calc_chksum_rst);
                addtoservlog((const char*)message,myippointer);
```

更新expected seq

```
servSeq++;
                if(servSeq>255)//SEQ只有八位
                {
                    servSeq=0;
                }
```

对数据进行处理

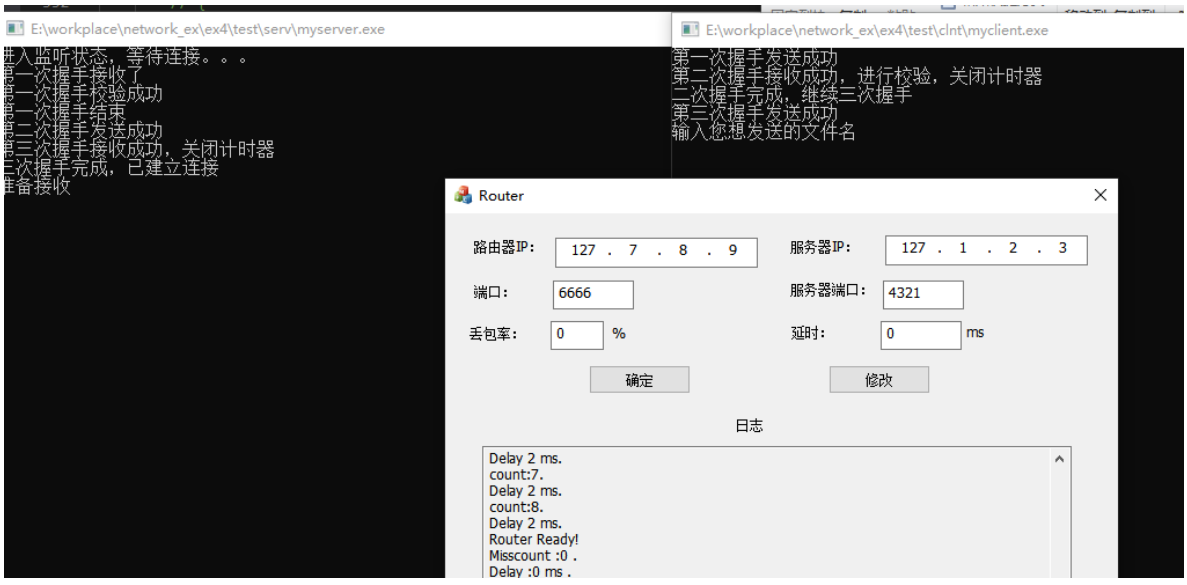
```
 //把除了头部的缓冲区（及数据）解码记录
                memcpy(tail,buffer+sizeof(header),recvLen-sizeof(header));
                //将这部分作为尾部，接到已接收的包后面
                memcpy(fullData+tailpointer,tail,recvLen-sizeof(header));
                //更新尾部指针
                tailpointer=tailpointer+recvLen-sizeof(header);
```

如果seq不对应，直接丢弃当前包，即什么都不做。

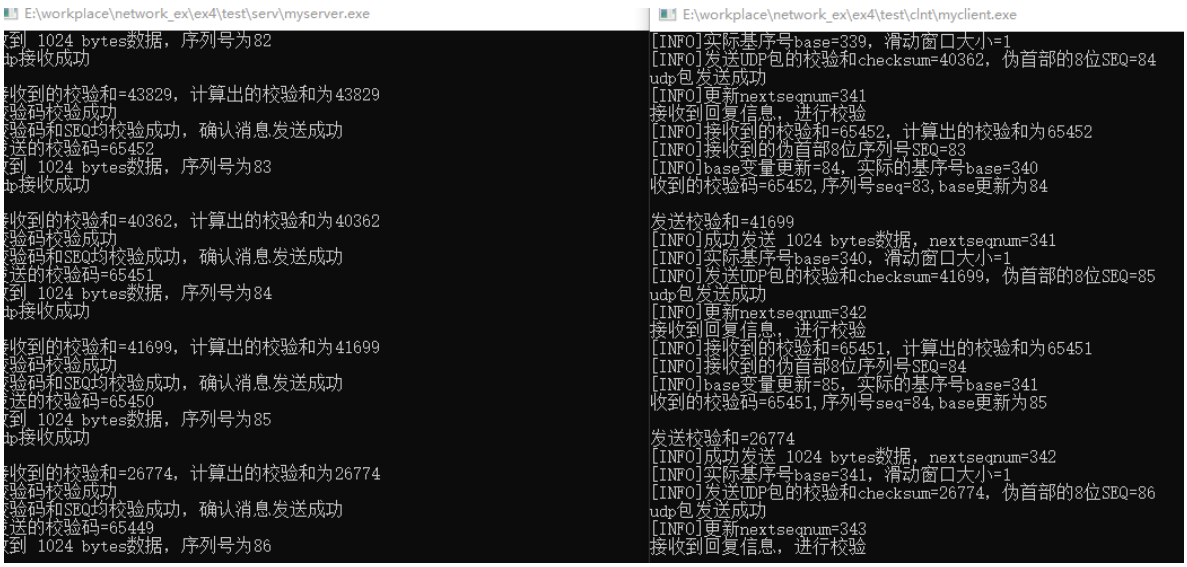
```
else{
                addtoservlog("[ERR ]seq值不对应，丢弃当前包",myippointer);
            }
```

实验截图

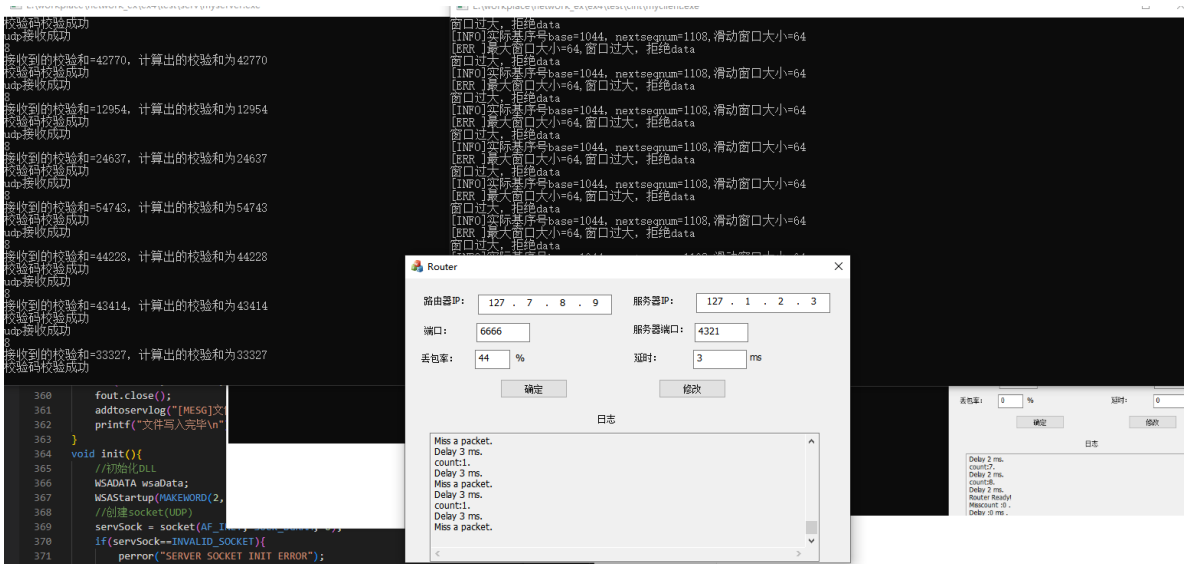
建立连接成功



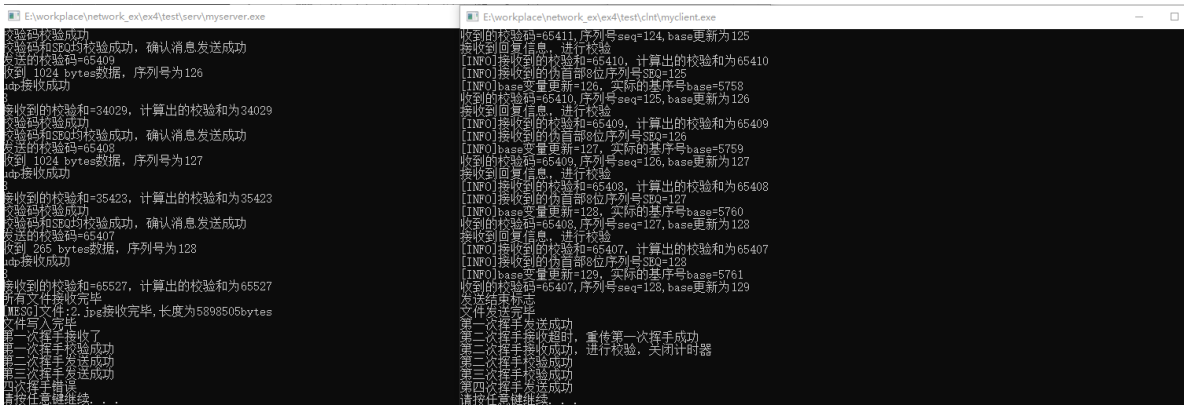
丢包率延迟都为0时接收

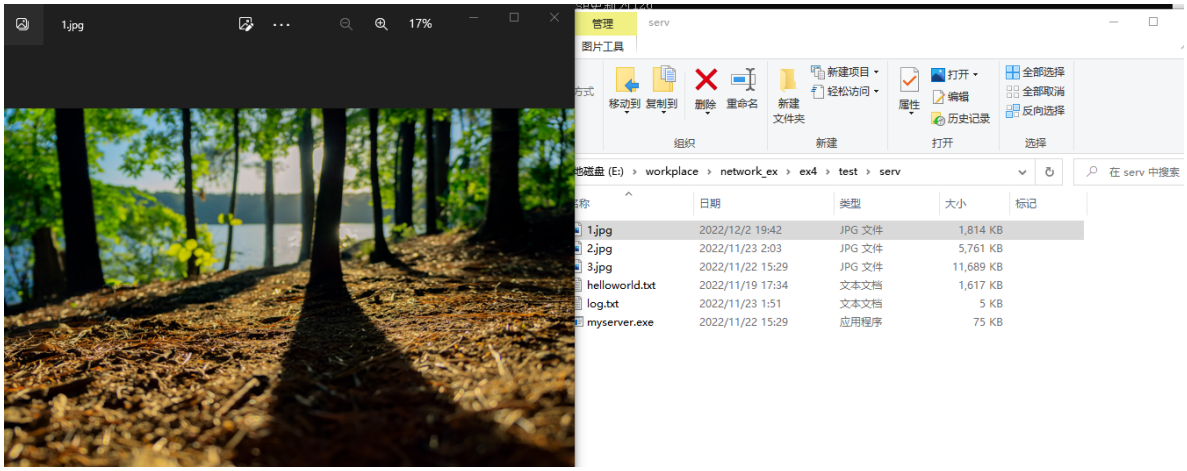


随意设置丢包率和延时，会产生阻塞，发生窗口过大以及重传等现象

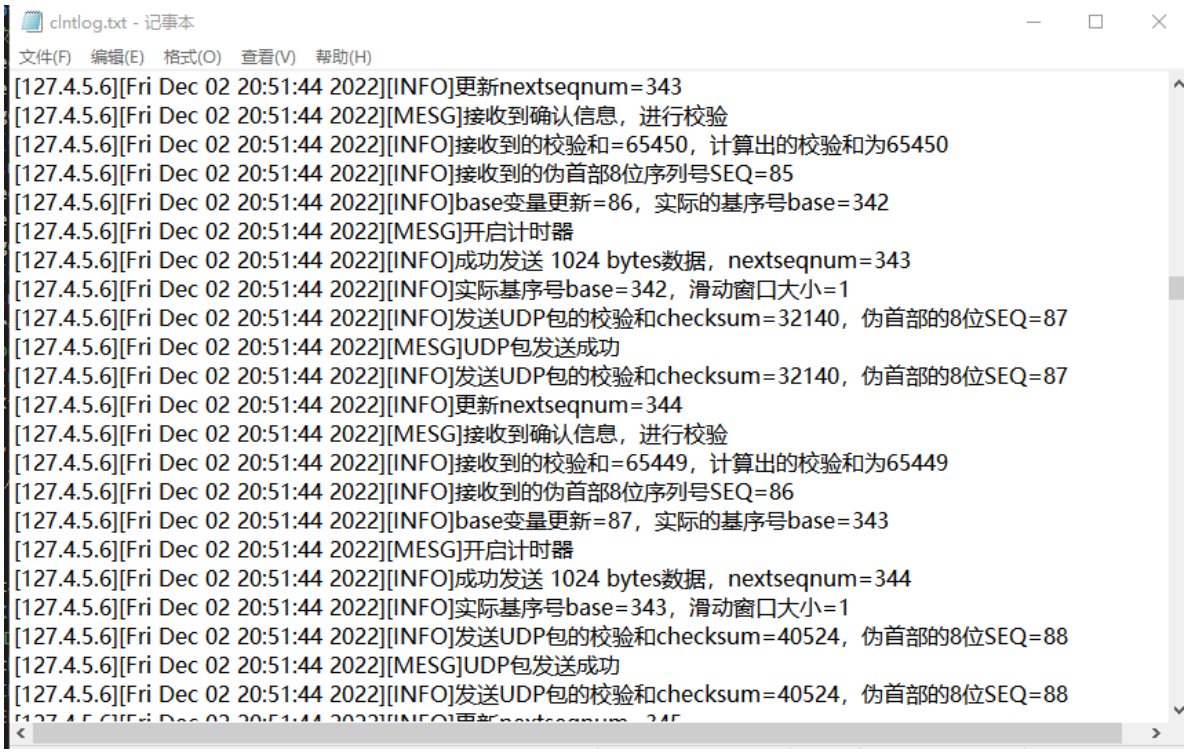


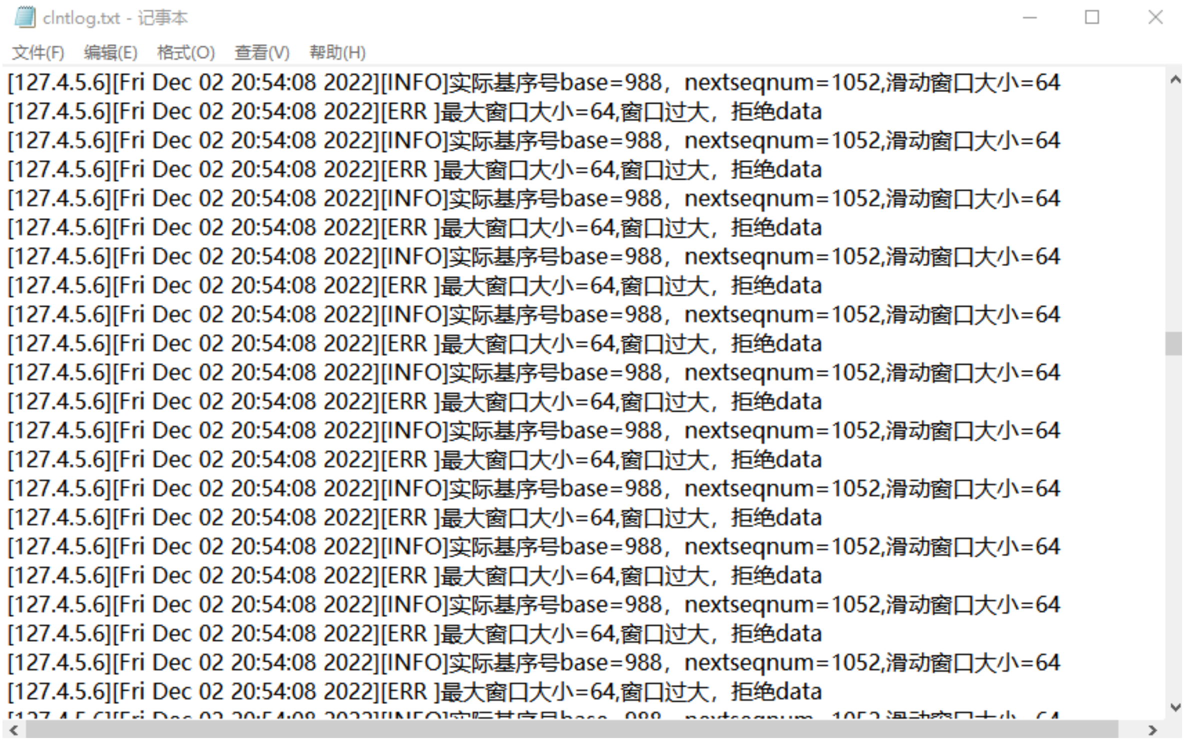
最后能正确传输并断开连接。





发送端日志记录时间戳、ip、信息类型、窗口大小等信息，并记录下超齿重传及窗口过大等情况





clntlog.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
[127.4.5.6][Fri Dec 02 20:54:08 2022][INFO]实际基序号base=988，nextseqnum=1052,滑动窗口大小=64
[127.4.5.6][Fri Dec 02 20:54:08 2022][ERR ]最大窗口大小=64,窗口过大，拒绝data
[127.4.5.6][Fri Dec 02 20:54:08 2022][INFO]实际基序号base=988，nextseqnum=1052,滑动窗口大小=64
[127.4.5.6][Fri Dec 02 20:54:08 2022][ERR ]最大窗口大小=64,窗口过大，拒绝data
[127.4.5.6][Fri Dec 02 20:54:08 2022][INFO]实际基序号base=988，nextseqnum=1052,滑动窗口大小=64
[127.4.5.6][Fri Dec 02 20:54:08 2022][ERR ]最大窗口大小=64,窗口过大，拒绝data
[127.4.5.6][Fri Dec 02 20:54:08 2022][INFO]实际基序号base=988，nextseqnum=1052,滑动窗口大小=64
[127.4.5.6][Fri Dec 02 20:54:08 2022][ERR ]最大窗口大小=64,窗口过大，拒绝data
[127.4.5.6][Fri Dec 02 20:54:08 2022][INFO]实际基序号base=988，nextseqnum=1052,滑动窗口大小=64
[127.4.5.6][Fri Dec 02 20:54:08 2022][ERR ]最大窗口大小=64,窗口过大，拒绝data
[127.4.5.6][Fri Dec 02 20:54:08 2022][INFO]实际基序号base=988，nextseqnum=1052,滑动窗口大小=64
[127.4.5.6][Fri Dec 02 20:54:08 2022][ERR ]最大窗口大小=64,窗口过大，拒绝data
[127.4.5.6][Fri Dec 02 20:54:08 2022][INFO]实际基序号base=988，nextseqnum=1052,滑动窗口大小=64
[127.4.5.6][Fri Dec 02 20:54:08 2022][ERR ]最大窗口大小=64,窗口过大，拒绝data
[127.4.5.6][Fri Dec 02 20:54:08 2022][INFO]实际基序号base=988，nextseqnum=1052,滑动窗口大小=64
[127.4.5.6][Fri Dec 02 20:54:08 2022][ERR ]最大窗口大小=64,窗口过大，拒绝data
[127.4.5.6][Fri Dec 02 20:54:08 2022][INFO]实际基序号base=988，nextseqnum=1052,滑动窗口大小=64
[127.4.5.6][Fri Dec 02 20:54:08 2022][ERR ]最大窗口大小=64,窗口过大，拒绝data
[127.4.5.6][Fri Dec 02 20:54:08 2022][INFO]实际基序号base=988，nextseqnum=1052,滑动窗口大小=64
[127.4.5.6][Fri Dec 02 20:54:08 2022][ERR ]最大窗口大小=64,窗口过大，拒绝data
[127.4.5.6][Fri Dec 02 20:54:08 2022][INFO]实际基序号base=988，nextseqnum=1052,滑动窗口大小=64
[127.4.5.6][Fri Dec 02 20:54:08 2022][ERR ]最大窗口大小=64,窗口过大，拒绝data
```

clntlog.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
127.4.5.6][Fri Dec 02 20:54:36 2022][INFO]实际基序号base=1036，nextseqnum=1100,滑动窗口大小=64
127.4.5.6][Fri Dec 02 20:54:36 2022][ERR ]最大窗口大小=64,窗口过大，拒绝data
127.4.5.6][Fri Dec 02 20:54:36 2022][INFO]实际基序号base=1036，nextseqnum=1100,滑动窗口大小=64
127.4.5.6][Fri Dec 02 20:54:36 2022][ERR ]最大窗口大小=64,窗口过大，拒绝data
127.4.5.6][Fri Dec 02 20:54:36 2022][INFO]实际基序号base=1036，nextseqnum=1100,滑动窗口大小=64
127.4.5.6][Fri Dec 02 20:54:36 2022][ERR ]最大窗口大小=64,窗口过大，拒绝data
127.4.5.6][Fri Dec 02 20:54:36 2022][INFO][超时重传]成功发送 1024 bytes数据，nextseqnum=1100
127.4.5.6][Fri Dec 02 20:54:36 2022][INFO][超时重传]当前实际基序号base=1036，滑动窗口大小=64
127.4.5.6][Fri Dec 02 20:54:36 2022][INFO][超时重传]发送UDP包的校验和checksum=29695，伪首部的8位SEQ=1
127.4.5.6][Fri Dec 02 20:54:36 2022][MESG][超时重传]UDP包发送成功
127.4.5.6][Fri Dec 02 20:54:36 2022][INFO][超时重传]成功发送 1024 bytes数据，nextseqnum=1100
127.4.5.6][Fri Dec 02 20:54:36 2022][INFO][超时重传]当前实际基序号base=1037，滑动窗口大小=64
127.4.5.6][Fri Dec 02 20:54:36 2022][INFO][超时重传]发送UDP包的校验和checksum=27282，伪首部的8位SEQ=1
127.4.5.6][Fri Dec 02 20:54:36 2022][MESG][超时重传]UDP包发送成功
127.4.5.6][Fri Dec 02 20:54:36 2022][INFO][超时重传]成功发送 1024 bytes数据，nextseqnum=1100
127.4.5.6][Fri Dec 02 20:54:36 2022][INFO][超时重传]当前实际基序号base=1038，滑动窗口大小=64
127.4.5.6][Fri Dec 02 20:54:36 2022][INFO][超时重传]发送UDP包的校验和checksum=1197，伪首部的8位SEQ=14
127.4.5.6][Fri Dec 02 20:54:36 2022][MESG][超时重传]UDP包发送成功
127.4.5.6][Fri Dec 02 20:54:36 2022][INFO][超时重传]成功发送 1024 bytes数据，nextseqnum=1100
127.4.5.6][Fri Dec 02 20:54:36 2022][INFO][超时重传]当前实际基序号base=1039，滑动窗口大小=64
127.4.5.6][Fri Dec 02 20:54:36 2022][INFO][超时重传]发送UDP包的校验和checksum=19126，伪首部的8位SEQ=1
127.4.5.6][Fri Dec 02 20:54:36 2022][MESG][超时重传]UDP包发送成功
```

接收端日志记录了时间戳、ip、校验码、seq等信息

servlog.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
[127.1.2.3][Fri Dec 02 20:54:59 2022][INFO]接收到的校验和=17468，计算出的校验和=17468
[127.1.2.3][Fri Dec 02 20:54:59 2022][MESG]udp包校验码校验成功
[127.1.2.3][Fri Dec 02 20:54:59 2022][INFO]应收到的SEQ=45，收到UDP包的SEQ=45
[127.1.2.3][Fri Dec 02 20:54:59 2022][INFO]SEQ值验证正确
[127.1.2.3][Fri Dec 02 20:54:59 2022][MESG]校验码和SEQ均校验成功，确认消息发送成功
[127.1.2.3][Fri Dec 02 20:54:59 2022][INFO]发送的校验码chechsum=65490
[127.1.2.3][Fri Dec 02 20:54:59 2022][INFO]更新服务端应收到的SEQ=46
[127.1.2.3][Fri Dec 02 20:54:59 2022][INFO]成功收到 1024 bytes数据，序列号为45
[127.1.2.3][Fri Dec 02 20:54:59 2022][MESG]udp包接收成功
[127.1.2.3][Fri Dec 02 20:54:59 2022][INFO]收到UDP包长度=1032
[127.1.2.3][Fri Dec 02 20:54:59 2022][INFO]接收到的校验和=59143，计算出的校验和=59143
[127.1.2.3][Fri Dec 02 20:54:59 2022][MESG]udp包校验码校验成功
[127.1.2.3][Fri Dec 02 20:54:59 2022][INFO]应收到的SEQ=46，收到UDP包的SEQ=46
[127.1.2.3][Fri Dec 02 20:54:59 2022][INFO]SEQ值验证正确
[127.1.2.3][Fri Dec 02 20:54:59 2022][MESG]校验码和SEQ均校验成功，确认消息发送成功
[127.1.2.3][Fri Dec 02 20:54:59 2022][INFO]发送的校验码chechsum=65489
[127.1.2.3][Fri Dec 02 20:54:59 2022][INFO]更新服务端应收到的SEQ=47
[127.1.2.3][Fri Dec 02 20:54:59 2022][INFO]成功收到 1024 bytes数据，序列号为46
[127.1.2.3][Fri Dec 02 20:54:59 2022][MESG]udp包接收成功
[127.1.2.3][Fri Dec 02 20:54:59 2022][INFO]收到UDP包长度=1032
[127.1.2.3][Fri Dec 02 20:54:59 2022][INFO]接收到的校验和=52988，计算出的校验和=52988
[127.1.2.3][Fri Dec 02 20:54:59 2022][MESG]udp包校验码校验成功
```