计算机网络实验三

学号：2011273

姓名：姚翔载

实验原理

建立连接

通过模仿TCP三次握手四次挥手的原理来建立连接。

原理如下：

三次握手：

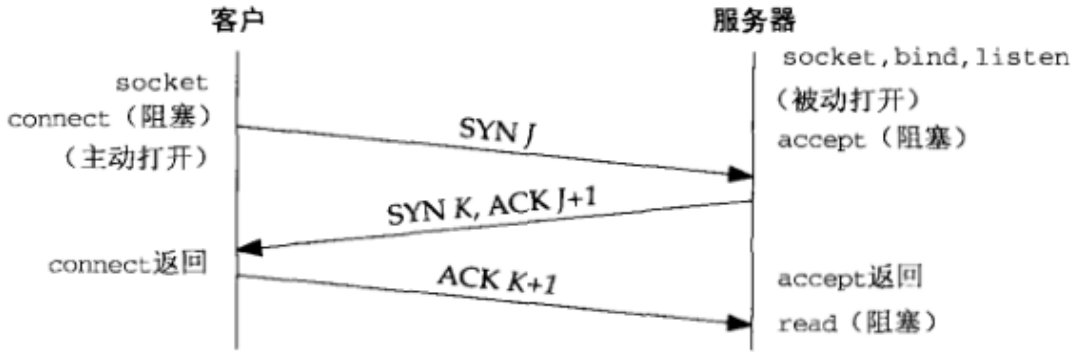
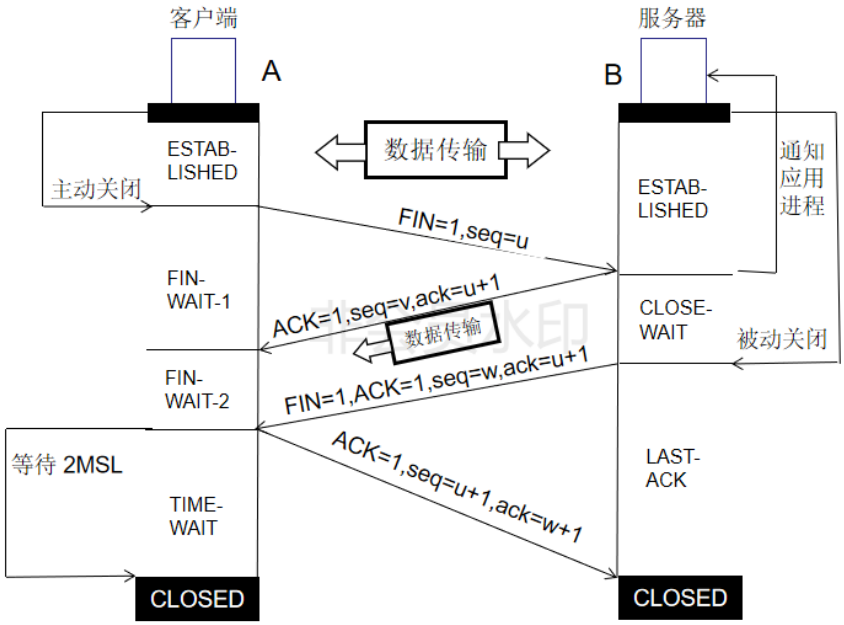


图2-2　TCP的三路握手

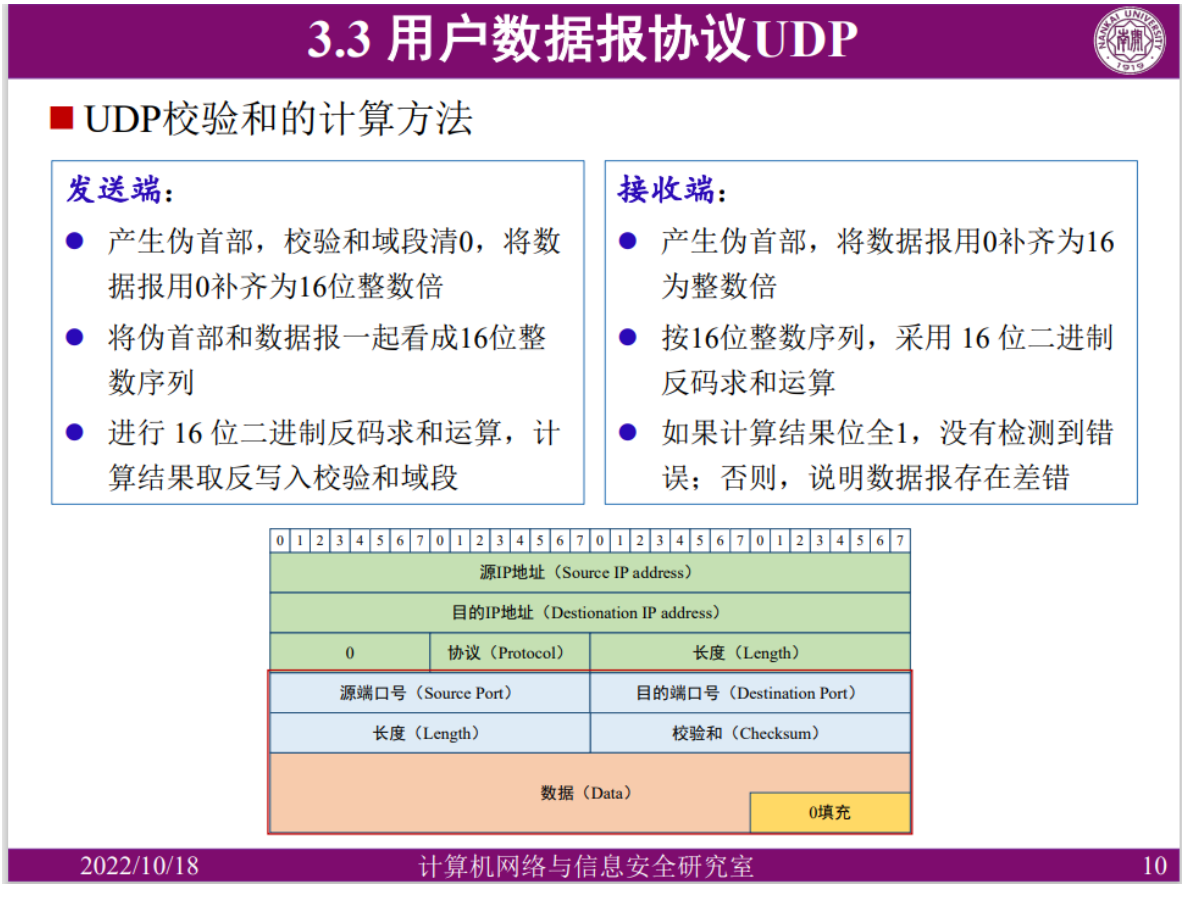
四次挥手：



将三次握手和四次挥手需要的各个标志位压缩在UDP包伪首部传输，具体见代码说明。三次握手四次挥手的详细原理在上次实验中已阐述，不再赘述。

差错检测

校验和的计算，与课上讲的相同。产生伪首部，用0补齐，与数据报（如果有）一起看成一整个序列，进行反码求和，并写入校验和域段。



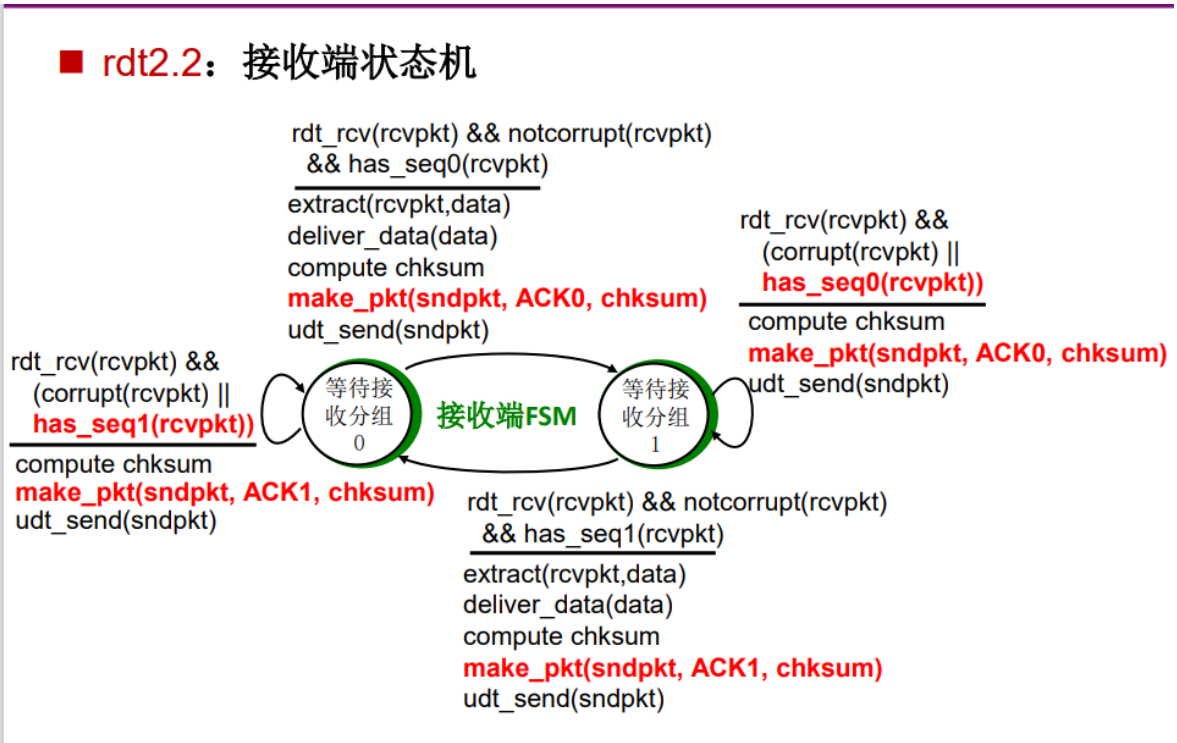
```
u_short cksum(u_short *buf, int count)
{
register u_long sum = 0;
while (count--)
{
sum += *buf++;
if (sum & 0XFFFF0000)
{
sum &= 0XFFFF;
sum++;
}
}
Return ~(sum & 0XFFFF);
}
```

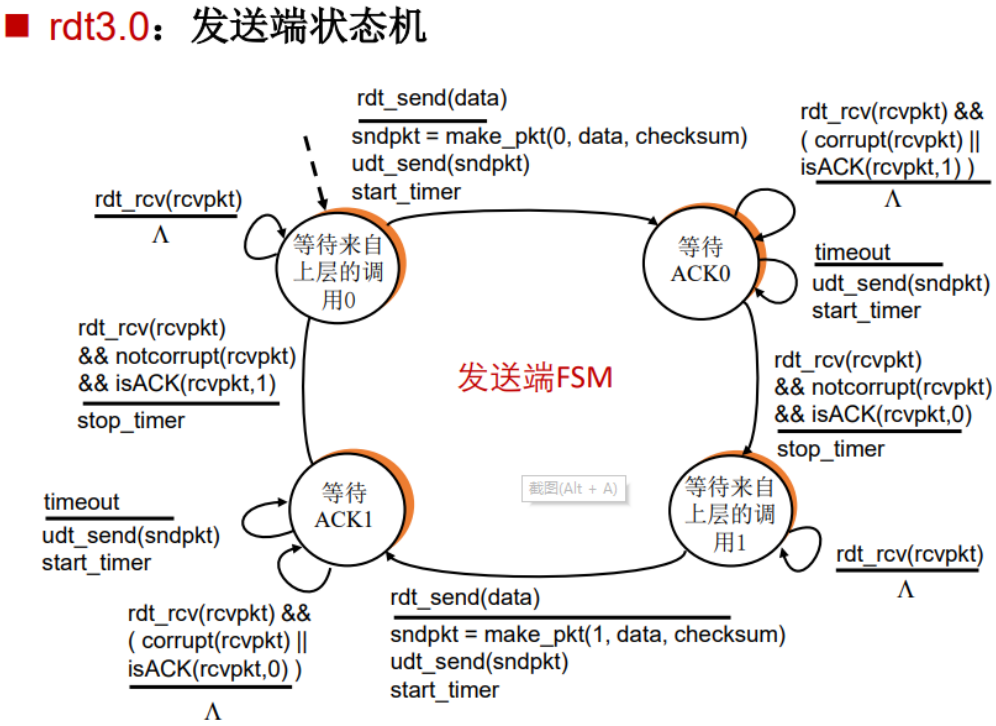
代码结构与伪代码相同，buf为存储的一整个序列指针，count实际上是要计算序列的位数。

这个过程的实现实操过程中，需要注意在计算除了校验和域段的校验和时，需要事先将校验和域段清零（用memset函数，或直接将结构体中的校验和部分清零），再计算校验和写入或比较。

确认重传

模仿rdt3.0协议实现，但保留了序列号（方便观察与debug）而不是一位的ACK号。





（rdt3.0接收端状态机与2.2基本相同）

因为采用了序列号SEQ而不是0或1的ACK，因此实际上实现的协议的FSM应该是一个很大的“圆圈”，从最小的SEQ到最大的SEQ。其余部分与上述相同，例如超时重传当前的pkt、数据包位反转、校验失败等操作都与上图的FSM相同。

停等机制

停等协议大致是指，发送端发送一个包，必须等待接收端的信号回应后才能继续发出下一个包；与此同时接收端必须在接收后发回一个应答信号。如果超过了计时的限制就采用超时重传机制。这一部分实际上已经包含在rdt3.0的设计实现当中与api的既有实现当中。例如一直循环调用recvfrom函数，直到对方sendto送到之后才执行下一步。

单向传输

两个cpp文件，两个可执行文件，分别绑定不同的ip以及端口（发送至对方的ip端口，或路由器的ip和端口），并根据FSM和握手挥手来分别进行实现。

日志输出

使用FILE的fopen方法来读取路径下的txt文件并进行输出即可。注意到fopen不是线程安全的，即在本实验中，如果server和client同时打开文件进行写操作，可能产生冲突（实际上实验过程中确实有这一点）。由于这部分并不是特别重要，除了命令行输出外，写文件的操作改为server和client分别写不同的txt文件。

代码思路

建立连接

初始化

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

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

原理如实验原理一节所述，参考伪代码实现即可。

```
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）。

确认重传

按照rdt3.0实现，但保留了SEQ（在实验中具体为8位，能表示0-255的seq号）而不是ack。

接收端

初始化

首先对需要用到的一些伪首部以及seq、指针等进行初始化。

```
long int tailpointer=0;//由于按包来传输，记录每次拼接后末尾的指针便于下一次拼接
HEADER header;
HEADER temp1;
u_short calc_chksum_rst;
```

char* buffer = new char[MAX_BUFFER_SIZE+sizeof(header)];

一个缓冲区的大小最大是数据的大小+伪首部的大小

接收

基本的api实际上与三次握手四次挥手相同。

首先循环中先阻塞地调用recvfrom，试图从发送端接收消息。它的返回值可以直接作为整个udp包接收的缓冲区大小（即数据的大小+伪首部的大小）。返回值小于0为错误，猜想按丢包一样的方法处理即可，但这里直接返回错误并不再接收（函数直接返回）。

```
int recvLen=recvfrom(servSock,buffer,sizeof(header)+MAX_BUFFER_SIZE,0,
(sockaddr*)&clntAddr,&clntAddrLen);
        if(recvLen<0)
        {
            addtoservlog("udp接收错误",myippointer);
            printf("接收错误\n");
            return -1;
        }
```

接收后校验。需要对校验码进行校验，成功后再进入下一步（不成功则continue外层循环，继续调用recvfrom等待发送端发回这个错误的数据包即可。校验码无误则可以找出伪首部header中的flag以及序列号SEQ。

flag中over一位为1，则代表对方已经发送来所有数据包传输完毕的消息，再经过一些后续处理就可以直接退出了。如果flag全0，则表示只是正常的数据包传输，那么对数据包的序列号进行检查，看是否是当前记录的“应该”收到的那个序列号。如果相等，则返回确认的消息（rdt3.0中是ack，这里直接设置seq，可以达到一样的效果。）如果不相等，这说明传输的序列有误，此时根据有限状态机，需要重发“应该”收到的那个序列号（即rdt3.0中相反的那个ack）。实际上在代码中可以直接通过一个结构体的实例来记录这个“应该”的seq所在的伪首部，重新计算校验和，发回这个伪首部（数据部分没有即可）。

```
if(servSeq!=(int)header.SEQ)
            {//has_seq1(),重传，当前记录的应该的seq与收到的seq不同
                //buffer不变，checksum不变，传回当前的seq
                unsigned char waitSEQ=header.SEQ;
                header=HEADER();
                header.SEQ=waitSEQ;
                header.checksum=0;
                //只传回一个header，不用传回数据
                //也不用数据计算校验和，否则那边无法校验
```

```
                //只需要seq
                calc_chksum_rst=CalcChecksum((u_short*)&header,sizeof(header));
                header.checksum=calc_chksum_rst;
                memcpy(buffer,&header,sizeof(header));
                //发回
                if (sendto(servSock,buffer,sizeof(header),0,
(sockaddr*)&clntAddr, clntAddrLen) == -1)
                {
                    printf("seq值不正确，重新发送当前seq值错误\n");
                    return -1;
                }
                addtoservlog("SEQ值不正确，重新发送当前SEQ值成功",myippointer);
                printf("%d!=%d,seq不对信息发送成功\n",servSeq,waitSEQ);
            }
            else
            {//has_seq0,发回正确的seq，更新服务端的seq
                ...
                    这里发送回正确的seq，也即rdt3.0中的ack值
                servSeq++;
                if(servSeq>255)//SEQ只有八位
                {
                    servSeq=0;
                }
                //更新“应该的”seq。用八位记录seq，那么大于255时需要回0
            }
```

在校验码和seq校验均通过之后，就是fsm中的extract and deliver data的部分。除了记录日志之外，将接受到的包通过指针”接“到整个数据序列之后，最后接受所有包完毕后就能组成发来的文件的为二进制序列的char数组。

```
char* tail=new char[recvLen-sizeof(header)];
                //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);
```

将上述部分封装为一个函数，最后返回-1则代表整个有一些错误，返回一个整数值，为接收到所有数据的长度，则代表接受正确（在当前层面的rdt3.0可以处理的范围内）

通过一个函数调用上述封装好的函数，根据读取的消息、存储二进制序列的char数组指针以及长度等，通过ofstream的方法来写入文件。最后能在同级目录下找到接收好的文件。

```
void RecvFileHelper()
{
    char* filename=new char[10000];
    char* filebuff=new char[INT_MAX];
    int lenn=sizeof(sockAddr);
    //发送端会发回文件名和文件的char数组
    int nameLen=RecvFile(servSock,sockAddr, lenn,filename);
    memset(message,0,sizeof(message));
    sprintf(message,"文件名:%s接收完毕",filename);
    addtoservlog((const char*)message,myippointer);
    printf("%s\n",message);
    int fileLen=RecvFile(servSock,sockAddr, lenn,filebuff);
```

```
    memset(message,0,sizeof(message));
    sprintf(message,"文件:%s接收完毕,长度为%dbytes",filename,fileLen);
    addtoservlog((const char*)message,myippointer);
    printf("%s\n",message);
    string namestr=filename;
    // ofstream fout(namestr.c_str(),ofstream::binary);
    ofstream fout(namestr.c_str(),ios::out|ios::binary);
    for(int i=0;i<fileLen;i++){fout<<filebuff[i];}
    fout.close();
    addtoservlog("文件写入完毕",myippointer);
    printf("文件写入完毕\n");
}
```

发送端

客户端发送，首先根据读取文件为二进制的结果，将文件存入到表示二进制的char数组中，以及记录出它的长度。分成数据包是通过对这个数组的指针操作进行的：设置一个最大的数据序列大小，对数组大小进行整除操作，能够得到包的个数和包的大小（前面的包即为最大的数据序列大小，最后一个为整除的余数大小，如下：

```
int pktnum=dataLen%MAX_BUFFER_SIZE==0?
dataLen/MAX_BUFFER_SIZE:dataLen/MAX_BUFFER_SIZE+1;
//计算包的个数
```

```
char* nowPktPointer=fullData+nowpkt*MAX_BUFFER_SIZE;
        memcpy(buffer,&header,sizeof(header));
        memcpy(buffer+sizeof(header),nowPktPointer,pktlen);
        calc_chksum_rst=CalcChecksum((u_short*)buffer,sizeof(header)+pktlen);
//通过对指针的处理来分配每个包的char型的部分序列。
```

最后计算校验和，调用sendto发出。注意包的长度是伪首部+数据部分的总长度，由于参数命名相同，在写这部分时盲目ctrlcv握手挥手部分的sendto形参表，导致debug用了很久。

发送完后等待接收端的确认消息。同样循环阻塞调用recvfrom接收，接收返回的udp包（实际上只有一个头部），并进行校验。在recv所处while调用的外层还有一个while，是为了handle接收接收端返回的消息后是否因为有误继续recv的处理。具体可见代码以及注释。

memcpy时同样需要注意整个udp包的长度是伪首部+数据部分的长度。同样的方法处理超时重传

```
mode = 1;
            ioctlsocket(clntSock, FIONBIO, &mode);
            while(recvfrom(clntSock, buffer, sizeof(header), 0,
(sockaddr*)&servAddr, &servAddrLen)<0){
                if(clock()-now_clocktime>MAX_TIME){

                    memcpy(buffer,nowbuffer,sizeof(header)+pktlen);
                    if(sendto(clntSock, buffer, sizeof(header)+pktlen, 0,
(sockaddr*)&servAddr, servAddrLen)<0){
                        addtoclntlog("确认消息超时，udp包重传发送错误",myippointer);
                        printf("超时重传错误\n");
                        return -1;
                    }
                    addtoclntlog("确认消息超时，udp包重传发送成功",myippointer);
                    printf("超时重传\n");
                    now_clocktime=clock();
```

```
                }
            }
```

注意需要用iotclsocket来设置非阻塞模式（并在处理完毕后设置回阻塞模式）

同样的方法对校验码和seq进行校验。同样要比较发回的包的seq和当前记录“应该”的seq。实际上在rdt3.0中校验码不通过（比特位反转）和序列号（ack）错误都是和超时重传一块处理即可，即continue到中层循环的开始，调用recvfrom进行接收。

如果没有continue，会跳出中层循环，此时可以设置socket回阻塞模式，并更新当前的pkt序号（便于移动指针）以及更新记录的“应该的”seq号。

如果遍历的当前pkt序号（从0开始）等于计算出的包个数-1时，表示所有包都发送完毕了，这时发回一个flag的over位为1的header的包，经过后续处理退出即可。

将上述封装为一个函数，外层一个函数调用它。外层的函数通过ifstream进行读文件，记录到char型数组，并记录指针和数据长度，传到上述函数的参数中方便分包处理。由于ofstream需要一个文件名来打开一个文件，因此可以在发送端先后发送文件名和文件本身，方便ofstream操作。

停等机制

实际上在上述过程的超时重传等方面就有体现，这里不再赘述。

日志输出

由于FILE的fopen不是线程安全的，即发送端和接收端同时fopen一个文件并写入时可能会混乱或乱码，为避免·在不必要的方面花费太多时间，分别用两个文件进行日志记录。

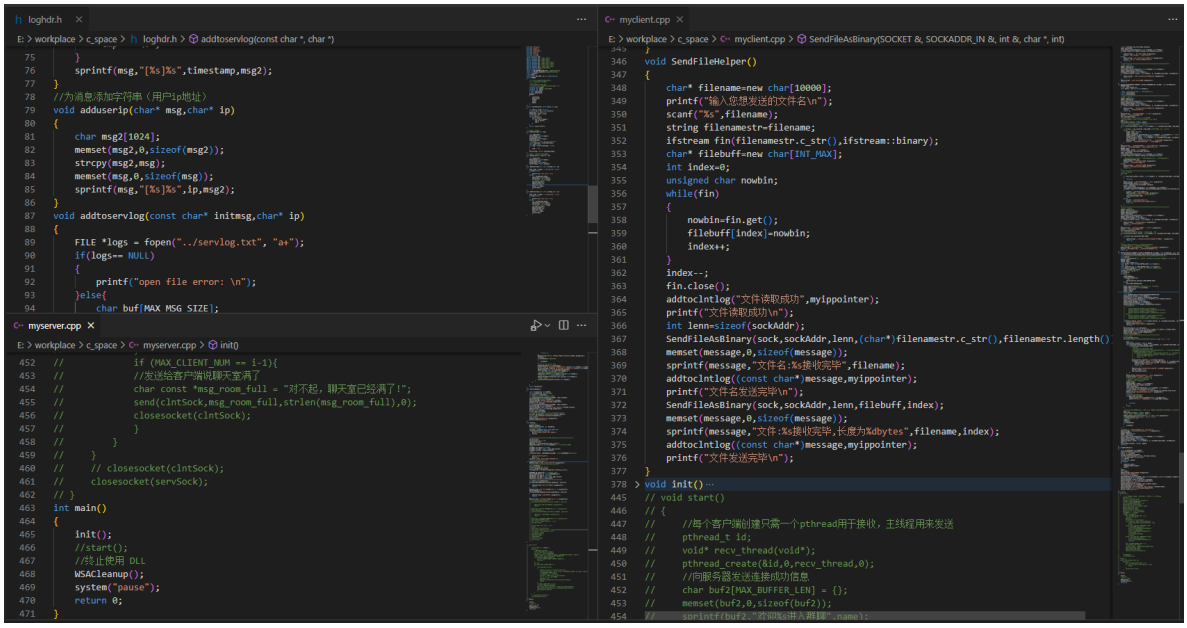
日志记录的函数大概如下：

```
void addtoservlog(const char* initmsg,char* ip)
{
    FILE *logs = fopen("../servlog.txt", "a+");
    if(logs== NULL)
    {
        printf("open file error: \n");
    }else{
        char buf[MAX_MSG_SIZE];
        memset(buf,0,sizeof(buf));
        sprintf(buf, "%s",initmsg);
        //为消息添加时间戳和用户ip
        addtimestamp(buf);
        adduserip(buf,ip);
        fputs(buf,logs);
        fputs("\n",logs);
        fclose(logs);
        }
}
```

打开同级文件夹下的一个txt文件并在尾部写入（a+）。addtimestamp和adduserip是向消息的string头部添加时间戳和ip信息，直接照搬了第一次实验中的函数。

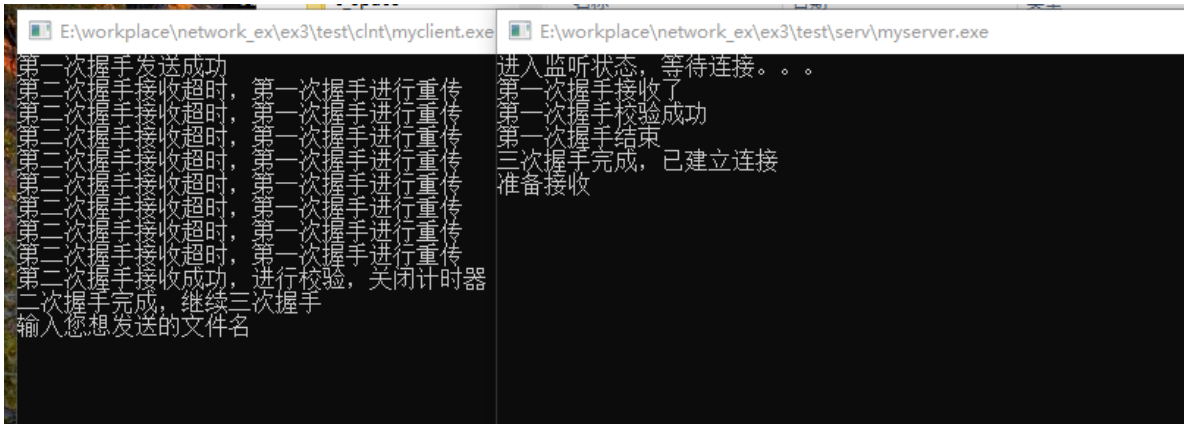
结果展示

工作区

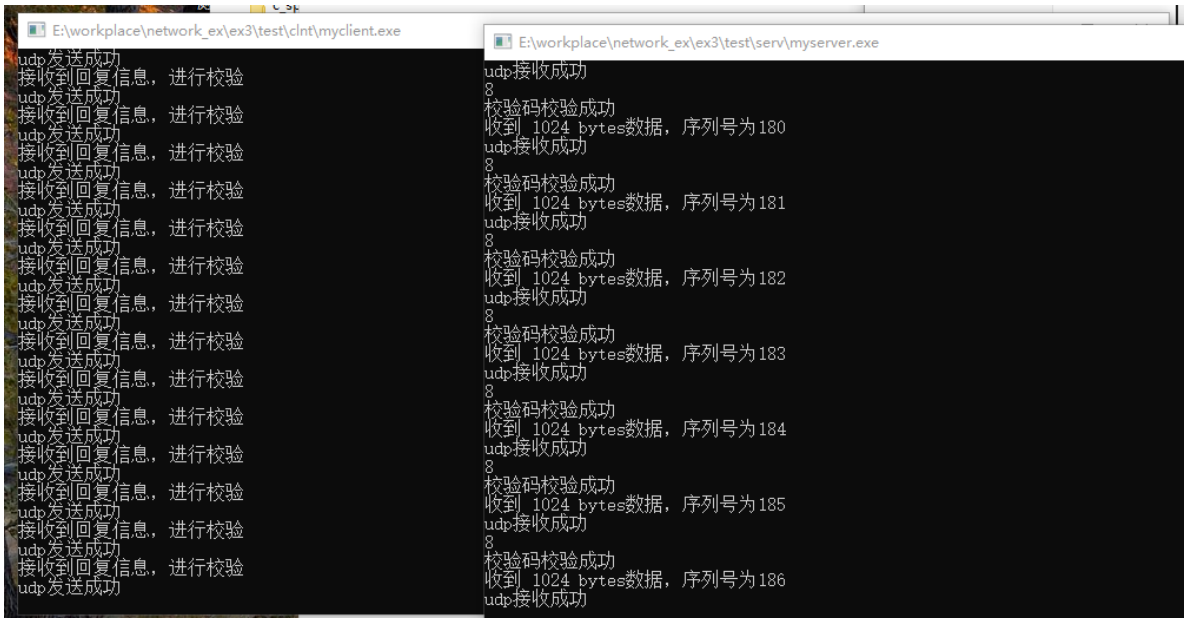


设置路由器（ip和端口与程序中绑定或发向的一致）

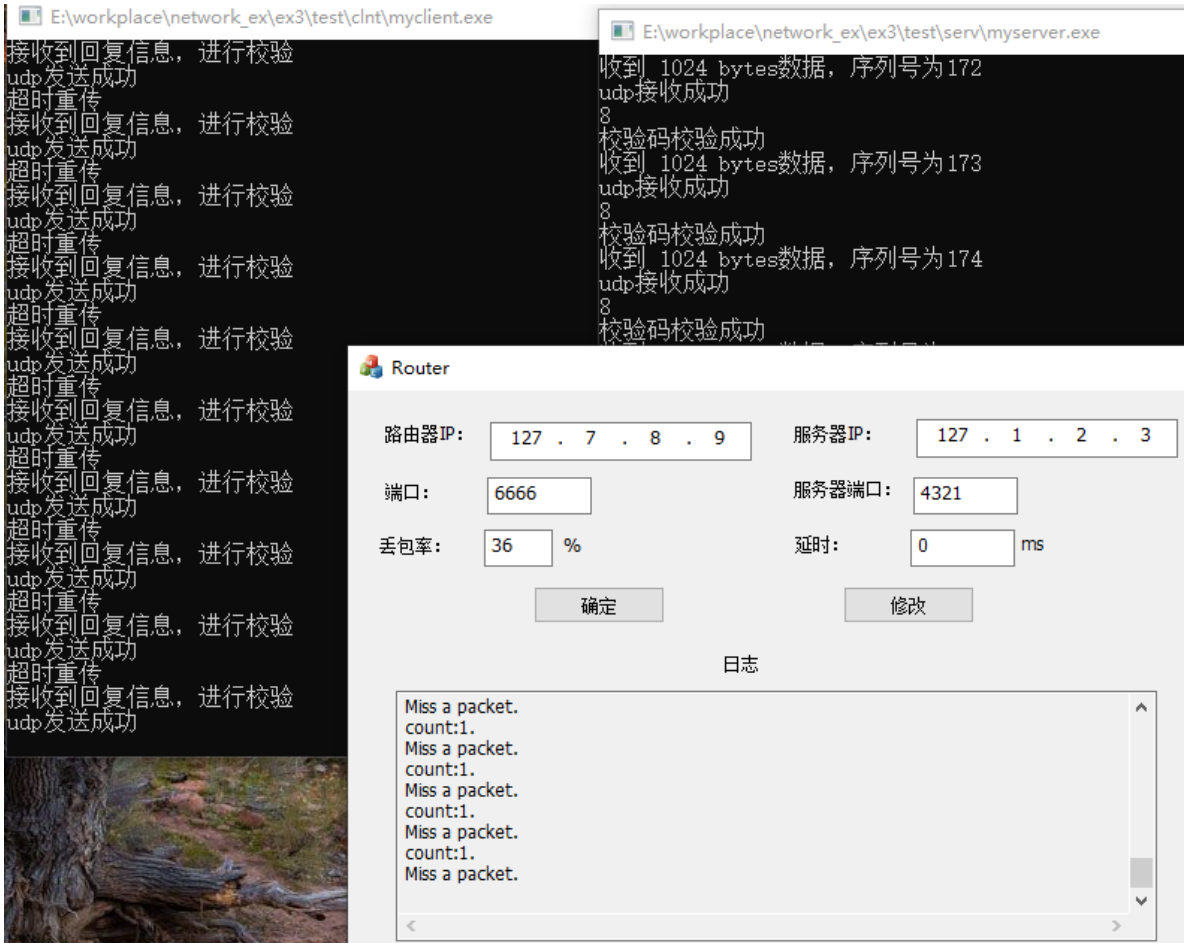
三次握手（这里先打开了client端，因为server没有接入，第一次没有得到ack会重传，指导server接入。）



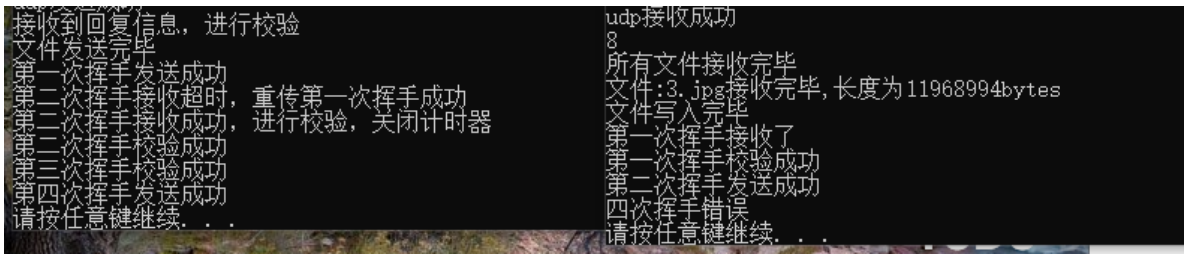
输入一个文件名，数据传输，在终端打印出部分提示



在这个过程中更改丢包率，可能会有超时重传的情况，但都能正确handle，且最后传输结果无误。



四次挥手，也可能有超时重传的情况，但能正确handle。



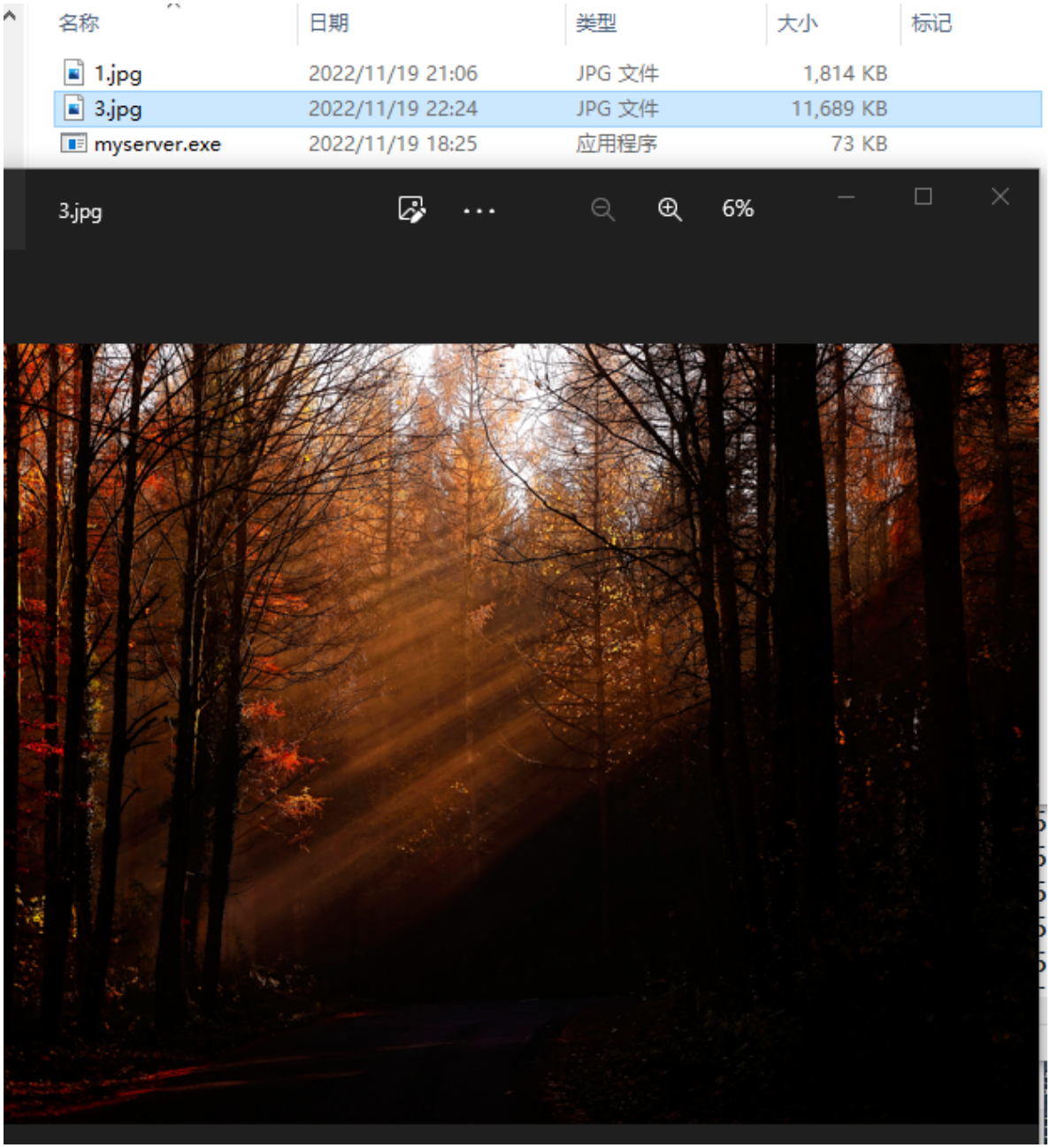
日志输出，包括三次握手和**文件名（而不是文件）**的传输

日志输出，包括四次挥手和文件的传输结果

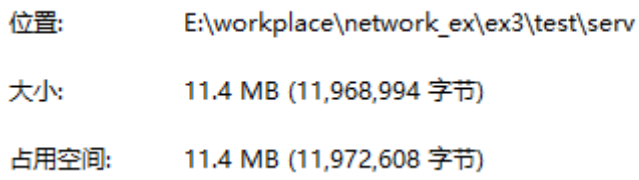
[127.4.5.6][Sat Nov 19 22:24:53 2022]接收到确认信息，进行校验
[127.4.5.6][Sat Nov 19 22:24:53 2022]udp包确认信息校验通过，传输下一个包
[127.4.5.6][Sat Nov 19 22:24:53 2022]成功发送 1024 bytes数据，序列号为167
[127.4.5.6][Sat Nov 19 22:24:53 2022]udp包发送成功
[127.4.5.6][Sat Nov 19 22:24:53 2022]接收到确认信息，进行校验
[127.4.5.6][Sat Nov 19 22:24:53 2022]udp包确认信息校验通过，传输下一个包
[127.4.5.6][Sat Nov 19 22:24:53 2022]成功发送 482 bytes数据，序列号为168
[127.4.5.6][Sat Nov 19 22:24:53 2022]udp包发送成功
[127.4.5.6][Sat Nov 19 22:24:53 2022]接收到确认信息，进行校验
[127.4.5.6][Sat Nov 19 22:24:53 2022]udp包确认信息校验通过，传输下一个包
[127.4.5.6][Sat Nov 19 22:24:53 2022]所有包发送完成，发送结束标志
[127.4.5.6][Sat Nov 19 22:24:53 2022]文件:3.jpg接收完毕,长度为11968994bytes
[127.4.5.6][Sat Nov 19 22:24:53 2022]第一次挥手发送成功，开启计时器
[127.4.5.6][Sat Nov 19 22:24:53 2022]第二次挥手接收超时，重传第一次挥手成功
[127.4.5.6][Sat Nov 19 22:24:54 2022]第二次挥手接收成功，进行校验，关闭计时器
[127.4.5.6][Sat Nov 19 22:24:54 2022]第二次挥手校验成功
[127.4.5.6][Sat Nov 19 22:24:54 2022]第三次挥手接收成功，进行校验
[127.4.5.6][Sat Nov 19 22:24:54 2022]第三次挥手校验成功
[127.4.5.6][Sat Nov 19 22:24:54 2022]第四次挥手发送成功，开启计时器
[127.4.5.6][Sat Nov 19 22:24:55 2022]第四次挥手后经2*MSL未收到数据，成功断开
[127.4.5.6][Sat Nov 19 22:24:55 2022]四次挥手断连成功，返回值为1

[127.1.2.3][Sat Nov 19 22:24:52 2022]成功收到 1024 bytes数据，序列号为197
[127.1.2.3][Sat Nov 19 22:24:52 2022]udp包接收成功
[127.1.2.3][Sat Nov 19 22:24:52 2022]udp包校验码校验成功
[127.1.2.3][Sat Nov 19 22:24:52 2022]SEQ值验证正确
[127.1.2.3][Sat Nov 19 22:24:52 2022]校验码和SEQ均校验成功，确认消息发送成功
[127.1.2.3][Sat Nov 19 22:24:52 2022]成功收到 1024 bytes数据，序列号为198
[127.1.2.3][Sat Nov 19 22:24:52 2022]udp包接收成功
[127.1.2.3][Sat Nov 19 22:24:52 2022]udp包校验码校验成功
[127.1.2.3][Sat Nov 19 22:24:52 2022]SEQ值验证正确
[127.1.2.3][Sat Nov 19 22:24:52 2022]校验码和SEQ均校验成功，确认消息发送成功
[127.1.2.3][Sat Nov 19 22:24:52 2022]成功收到 1024 bytes数据，序列号为199
[127.1.2.3][Sat Nov 19 22:24:52 2022]udp包接收成功
[127.1.2.3][Sat Nov 19 22:24:52 2022]udp包校验码校验成功
[127.1.2.3][Sat Nov 19 22:24:52 2022]SEQ值验证正确
[127.1.2.3][Sat Nov 19 22:24:52 2022]校验码和SEQ均校验成功，确认消息发送成功
[127.1.2.3][Sat Nov 19 22:24:52 2022]成功收到 1024 bytes数据，序列号为200
[127.1.2.3][Sat Nov 19 22:24:52 2022]udp包接收成功
[127.1.2.3][Sat Nov 19 22:24:52 2022]udp包校验码校验成功
[127.1.2.3][Sat Nov 19 22:24:52 2022]SEQ值验证正确
[127.1.2.3][Sat Nov 19 22:24:52 2022]校验码和SEQ均校验成功，确认消息发送成功
[127.1.2.3][Sat Nov 19 22:24:52 2022]成功收到 1024 bytes数据，序列号为201

服务端的同级目录能正确收到图片



大小与发送端的测试文件完全相同



（必须完全相同，例如txt文件最后如果多写入了制表符等等可能整个文件都会显示乱码，实验中也遇到了这种情况并正确debug）

实验心得

网络课的实验并不简单，不仅需要对课上知识有很深刻的了解，并且极其需要注重细节。做完实验后真正地对校验码、rdt、停等协议等重要知识有了更加细致的认识。并且实验过程中也因为细节吃了不少亏，例如：接收数据包只拿出了伪首部来处理、读文件时index多加了1没退回、没有讲校验码置零就计算新的校验码……这些过程都是痛并快乐着的过程，提高了debug能力，也为后续实验的细节方面、debug方法等提供了很好的思路，~~在深夜都debug完之后感觉灵魂得到了升华~~