

实验3-3

学号：2112066 姓名：于成俊

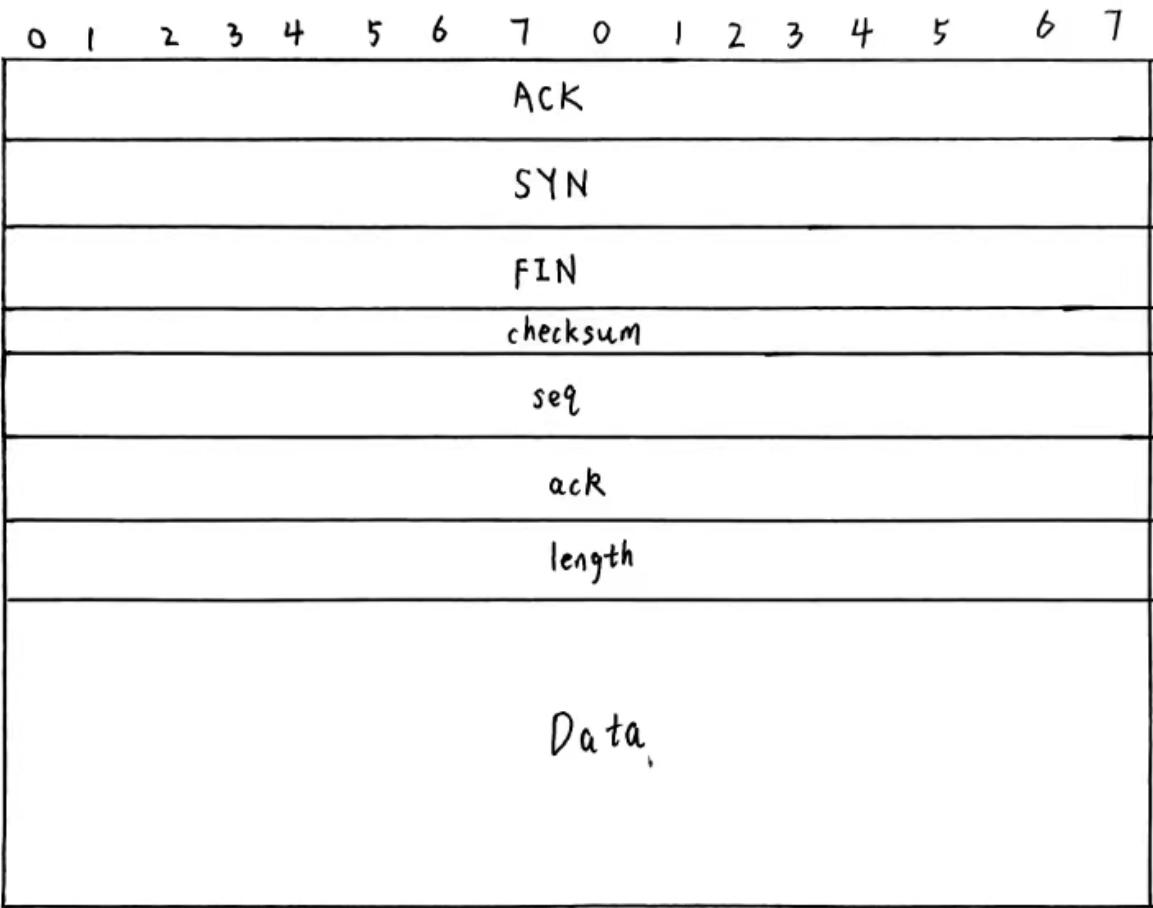
一、实验题目：

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。

二、协议设计

1.报文格式：

所设计的报文格式如下图所示：

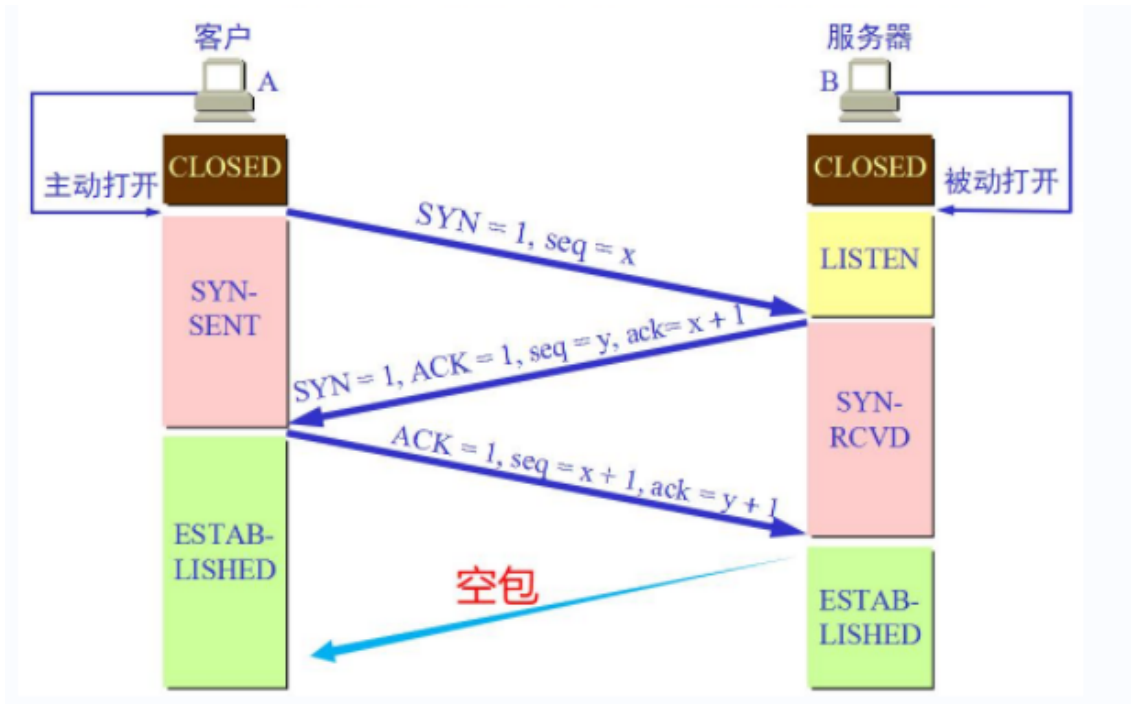


- 报文总长度为82128位，即10266字节。
- 前32位为ACK，用于确认序号有效。
- 33—64位为SYN，用于发起一个连接。
- 65—96位为FIN，用于释放一个连接；以及在数据传输过程中，用于告知数据已传输完毕。
- 97—112位为checksum，即校验和，用于差错检验。
- 113—144位为seq，为传输数据包的序列号。在建立连接和断开连接时，为随机取值；在数据传输时，只为0和1。

- 145—176位为ack，为确认序列号，只在建立连接和断开连接时使用，在数据传输时不涉及，取值为seq+1。
- 177—208位为length，为数据长度。
- 209—82128位为数据。

2.四次握手建立连接

- 参考于TCP的三次握手，所设计的流程图如下：



- 首先客户端向服务器端发送一个报文，其 SYN 标志位置 1，标志请求建立连接；并从1~100中随机选取一个整数x赋值给seq。
- 服务器收到请求后，向客户端回复一个报文，SYN 和 ACK 标志位置 1，标志允许建立连接；并从1~100中随机选取一个整数y赋值给seq。
- 客户端收到服务器反馈后，向服务器发送一个报文，ACK 置 1，标志将要开始传输；且将seq设为x+1，将ack设为y+1；
- 最后服务器再发送个空包（任意一个包也行）给客户端，这样，客户端收到一个空包就知道服务器端已经收到了第三次的报文。这就是第四次握手。

3.可靠数据传输

发送端和接收端均参考SR协议，具体如下：

- 允许客户端（发送端）连续发出N个未得到确认的分组。其中，**N为发送窗口的大小**。
- 与停等协议相比，增加了序列号的范围。seq的取值为1到MaxSeq-1。其中，MaxSeq表示序号空间的大小。注意：**序号空间应大于等于窗口大小的2倍**，即 $MaxSeq \geq 2N$ 。
- 服务器（接收端）采取**选择确认**的方式，只确认刚接收这个分组的序列号。
- 服务器（接收端）面对乱序来的分组，需要将其缓存，最后按序上交上层。

具体的交互过程如下：

客户端：

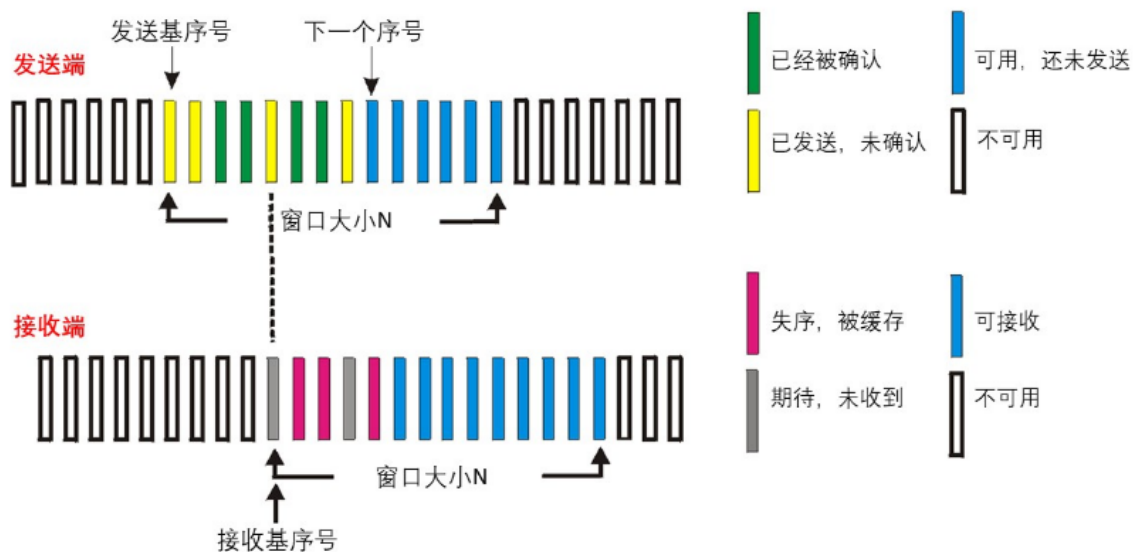
- 开始时，客户端连续发出多个分组直至发送窗口满为止。在此过程中，客户端需要计算每个分组的**校验和**，并将校验和连同数据一起发送。并且，客户端每发送一个分组就开一个超时定时器。

- 如果接收到发送窗口中第一个分组的确认号，则将第一个分组从发送窗口移除，再发送一个新的分组；
- 如果没有接收到发送窗口中第一个分组的确认号而是接收到第一个分组后面的分组的确认号，则将被确认的分组的超时定时器关闭。但发送窗口无法向前移动，需要等待发送窗口中第一个分组的确认。
- 如果没有接收到第一个分组的确认号，也没有接收到第一个分组后面的分组的确认号。则当哪个分组的超时定时器报警后，就将哪个分组重新发送，不需要将其后的所有分组发送。循环以上过程。
- 当数据发送完成后，向服务器发送FIN=-1的数据报，告知数据已发送完。

服务器：

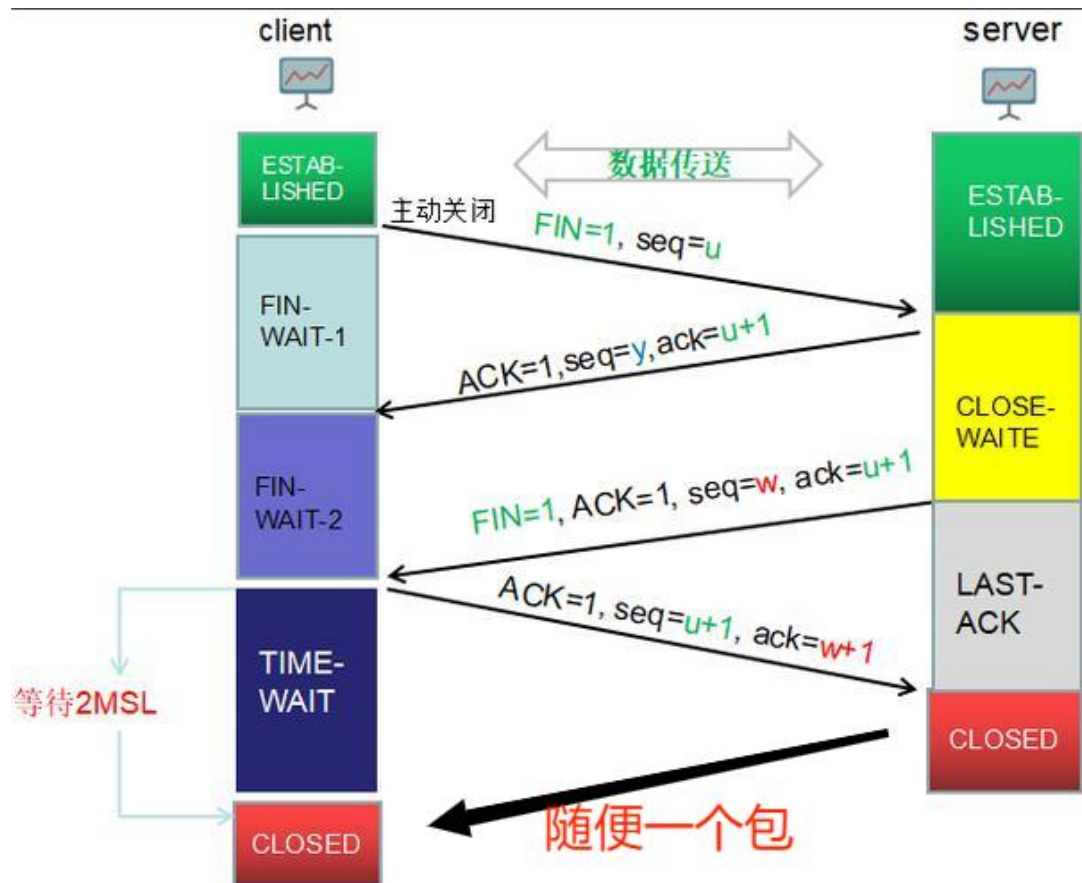
- 接收端等待来自客户端发送的分组，接收到后计算校验和并检查序列号。
- 若校验和不为0，说明分组出现错误，则忽略该分组。
- 若校验和为0且分组的序列号落入接收窗口内，则给客户端发送ACK=seq的分组(该seq为这个分组的seq)。表示已正确接收这个分组。如果这个分组是接收窗口中的第一个，则将这个分组交付给上层，并将接收窗口向前移动；若这个分组不是接收窗口中的第一个，则将其缓存，等待这个分组之前的分组全部到来后，再一起交付给上层，然后将接收窗口向后移动。
- 若校验和为0而分组的序列号没有落入接收窗口内，说明是收到重复分组的情况，则直接丢弃并给客户端发送ACK=seq的分组(该seq为这个分组的seq)。告知客户端我已经接收过了。注意，因为序号空间大于等于窗口大小的2倍，且接收窗口是比发送窗口先向后移动的，所以这个分组不可能是接收窗口后面的分组，即以后应该接受的分组。
- 若收到FIN=-1且校验和为0的分组，则表示数据已发送完毕，不再等待接收该文件数据。并发送个FIN=-1的报文，告知对方我已知道。

发送窗口和接收窗口图示：



4.五次挥手断开连接

- 参考TCP的四次挥手，所设计的流程图如下：



- 客户端向服务器端发送一个报文，将 FIN 标志位置 1，并从 1~100 中随机选取一个整数 u 赋值给 seq ，标识请求断开连接。
- 服务器端收到断开请求后，回应一个报文，将 ACK 标志位置 1，并从 1~100 中随机选取一个整数 y 赋值给 seq ，且令 $ack=u+1$ ，标识接到断开请求。
- 服务器端向客户端发送一个报文，将 ACK 和 FIN 标志位置 1，并从 1~100 中随机选取一个整数 w 赋值给 seq ，且令 $ack=u+1$ ，标识请求断开连接。
- 客户端收到断开请求后，回应一个报文，将 ACK 标志位置 1，令 $seq=u+1$ ， $ack=w+1$ 。
- 最后，服务器随便发送一个包，我已关闭。客户端接收到后，关闭连接。

三、设计实现

1. 报文格式:

```
const int BUFFER_SIZE = 10240;
//数据报
struct Datagram {
    //标志位
    int ACK, SYN, FIN;
    //校验和
    unsigned short int checksum; //unsigned short int是16位
    int seq, ack; //序列号和确认号
    int length; //数据长度
    char data[BUFFER_SIZE]; //数据
};
```

2. 差错检验机制实现（计算校验和）：

- 发送方生成校验和：
 - 将接受的数据报分成若干个16位的位串，每个位串看成一个二进制数。
 - 将校验和域段清零，该字段也参与校验和运算。
 - 对这些16位的二进制数进行反码求和。
 - 将累加的结果再取反，得到校验和，放入校验和域段

```
//发送时计算校验和
void send_calculate_checksum(Datagram& datagram) {
    unsigned short int* buff = (unsigned short int*) & datagram;
    int num = sizeof(Datagram) / sizeof(unsigned short int); //有多少16位
    datagram.checksum = 0; //将校验和域段清0
    unsigned long int checksum = 0; //unsigned long int占四个字节
    while (num--)
    {
        checksum += *buff;
        buff++; //指向下一个16位
        //若超出16位，即有进位
        if (checksum & 0xffff0000)
        {
            //将超出16位的部分置0
            checksum &= 0xffff;
            //进位+1
            checksum++;
        }
    }
    //取反
    datagram.checksum = ~(checksum & 0xffff);
}
```

- 接收方生成校验和：

除了不用将校验和域段清零外，其他步骤一样。若结果为0，则没错误。

3. 超时重传机制实现（与3-1不同）：

在实验3-1中，因为是停等协议，所以我使用了 `setsockopt()` 函数来实现超时重传。而本次实验，是SR协议，发送端可以连续发送未确认的分组，`setsockopt()` 函数当 `recvfrom` 函数在规定时间内收到信息就不会超时，但这次实验需要接受到在发送窗口里的分组序号才不会超时，`setsockopt()` 函数无法满足这个需求。所以我采用了将 `recvfrom()` 函数设置为非阻塞状态来实现。即使用 `ioctlsocket()` 函数

`ioctlsocket()` 函数原型如下：

```
int ioctlsocket( SOCKET s, long cmd, u_long FAR *argp );
```

各参数定义：

- s：一个标识套接口的描述字。
- cmd：对套接口s的操作命令。
- argp：指向cmd命令所带参数的指针。

使用方法如下：

```
// 设置Socket为非阻塞模式
bool setNonBlocking(SOCKET socket) {
    u_long mode = 1; // 1表示非阻塞，0表示阻塞
    if (ioctlsocket(socket, FIONBIO, &mode) == SOCKET_ERROR) {
        cerr << "设置非阻塞模式失败！ " << endl;
        return false;
    }
    return true;
}
```

在某些情况下，还需设置为阻塞状态。

```
// 设置Socket为阻塞模式
bool setBlocking(SOCKET socket) {
    u_long mode = 0; // 0表示阻塞，1表示非阻塞
    if (ioctlsocket(socket, FIONBIO, &mode) == SOCKET_ERROR) {
        cerr << "设置阻塞模式失败！ " << endl;
        return false;
    }
    return true;
}
```

本次实验的**超时重传功能**都是基于下面的方式实现的。（超时时间设置为500毫秒）

我专门创建了一个**线程**，用于检测并处理超时事件。定义了两个全局vector数组timer和SendDataArray

```
//定时器
struct Timer {
    clock_t start;
    bool on;
};
vector<Timer>timer;
//发送窗口
vector<Datagram>SendDataArray;
```

Timer的成员变量on**默认是true**，即定时器是开启状态的。由于每个分组都有属于自己的超时定时器，所以timer和SendDataArray是同步变化的。为了防止多线程竞争访问数据产生冲突，我用到了**互斥锁**。

超时处理过程：超时处理线程不断检测timer中的每个开启的定时器，如果检测到超时，由于timer和SendDataArray是**一一对应**的，所以可以根据timer中超时的定时器的索引，来直接将发生超时的分组重发。定时器的关闭是接收线程的任务，当接收线程收到一个分组的确认后，会将该分组对应的定时器关闭，即令on=false。当接收线程知道任务结束后，会告知超时处理线程关闭。具体代码如下：

超时处理线程关键代码：(接收线程的代码会在后面列出)

```
//检测并处理超时事件
void Handle_Timeout(SOCKET& clientSocket, sockaddr_in& routerAddr, int&
routerAddrLen) {
    {
        lock_guard<mutex> lock(coutmutex);
        cout << "处理超时线程开启！ " << endl;
    }
    closed = false;
    while (true) {
```

```

        //由接收线程将closed置为true
        if (closed) {
            break;
        }
        {
            lock_guard<mutex> lock(mtx);
            for (int i = 0; i < SendDataArray.size(); i++) {
                //超时重发
                if (timer[i].on && clock() - timer[i].start > TIMEOUT) {
                    lock_guard<mutex> lock(coutmutex);
                    cout << "分组" << SendDataArray[i].seq << "丢了!!! 重新发送分
组" << SendDataArray[i].seq << "及其之后的分组" << endl;
                    //将超时的分组重新发送
                    if (sendto(clientSocket, (char*)&SendDataArray[i],
sizeof(SendDataArray[i]), 0, (SOCKADDR*)&routerAddr, sizeof(routerAddr)) ==
SOCKET_ERROR) {
                        cerr << "发送错误!" << endl;
                    }
                    timer[i].start = clock();
                }
            }
        }
        //让其他线程运行
        sleep(1);
    }
    {
        lock_guard<mutex> lock(coutmutex);
        cout << "处理超时线程关闭!" << endl;
    }
    finished = false;
    return;
}

```

4.发送窗口的实现:

- 设置序号空间大小和发送窗口大小:

```

// 发送窗口大小
const int windowSize = 16;
// 传输数据时的序号空间大小
const int MaxSeq = 100;

```

- 通过vector来实现发送窗口（与实验3-2使用queue不同）

```

//发送窗口
vector<Datagram>SendDataArray;

```

- 使用vector来表示窗口有以下几个原因:
 - vector可以模仿像队列这样的先进先出的数据结构，与发送窗口的滑动方式完美契合，使用非常便利。如下:

```
//当发送新的分组时，只需用push_back函数
SendDataArray.push_back(SendData);
//当丢弃已确认的分组时，只需用erase函数
SendDataArray.erase(SendDataArray.begin());
```

- 当想获得位于发送窗口下沿和上沿的分组时，只需如下操作：

```
//位于下沿的分组序号
SendDataArray.front().seq
//位于上沿的分组序号
SendDataArray.back().seq
```

- 但是vector相对于queue来说，维护**先进先出**这样数据结构需要付出更多代价。但是它有一个优点，就是可以通过索引来操作相应的元素，这就可以更好的实现选择确认。
- 发送数据时（关键代码）：

```
//发送数据
while (SendDataArray.size() < windowSize) {
    if (file.eof()) {
        break;
    }
    //清空要发送的数据
    memset(&SendData, 0, sizeof(SendData));
    //设置分组序号
    SendData.seq = sequenceNumber++;
    sequenceNumber = sequenceNumber % MaxSeq;
    //每次从文件中读取最多BUFFER_SIZE 字节的数据
    file.read(SendData.data, BUFFER_SIZE);
    int bytesToSend = static_cast<int>(file.gcount());
    totalBytes += bytesToSend;
    //设置数据长度
    SendData.length = bytesToSend;
    //计算校验和
    send_calculate_Checksum(SendData);
    int size;
    int base;
    int back;
    //加入到窗口
    {
        lock_guard<mutex> lock(mtx);
        SendDataArray.push_back(SendData);
        Timer t = { clock(), true };
        timer.push_back(t);
        size = SendDataArray.size();
        base = SendDataArray.front().seq;
        back = SendDataArray.back().seq;
    }
    if (sendto(clientSocket, (char*)&SendData, sizeof(SendData), 0,
        (SOCKADDR*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR) {
        cerr << "发送错误!" << endl;
    }
    {
        lock_guard<mutex> lock(coutmutex);
        cout << "发送窗口大小为: " << size << endl;
        cout << "发送窗口的下沿为:" << base << endl;
        cout << "发送窗口的上沿为:" << back << endl;
    }
}
```



```
}
```

```
}
```

- 接收确认号，将已被确认的分组的定时器关闭：（需要判断确认号是否在发送窗口中，有两种情况）

①情况一：

确认号为3				
1	2	3	4	5

②情况二：

确认号为10				
9	10	11	0	1
确认号为1				
10	11	0	1	2

代码实现如下：（即接收线程函数关键代码）

```
//接收数据包线程
void receiveACK(SOCKET& clientSocket, sockaddr_in& routerAddr, int&
routerAddrLen) {
    finished = false;
    {
        lock_guard<mutex> lock(coutmutex);
        cout << "接收线程开启!" << endl;
    }
    //等待ACK反馈
    while (true) {
        //清空接收数据
        memset(&ReceiveData, 0, sizeof(ReceiveData));
        // 接收数据
        if (recvfrom(clientSocket, (char*)&ReceiveData, sizeof(Datagram), 0,
        (struct sockaddr*)&routerAddr, &routerAddrLen) <= 0) {
            if (finished) {
                /*-----告诉对方文件传输完了的相关代码*-----*/
            }
            continue;
        }
        //计算校验和
```

```

receive_calculate_Checksum(ReceiveData);
//数据包正确就开始检查是否在发送窗口
if (ReceiveData.checksum == 0 ) {
    if (ReceiveData.FIN != -1) {
        int base;
        int back;
        {
            lock_guard<mutex> lock(mtx);
            if (SendDataArray.size() == 0) {
                continue;
            }
            base = SendDataArray.front().seq;
            back = SendDataArray.back().seq;
            if (base <= back) {
                //情况一
                if (base <= ReceiveData.ACK && ReceiveData.ACK <= back)
{
                    timer[ReceiveData.ACK - base].on = false;
                }
            }
            //情况二
            else {
                if (base <= ReceiveData.ACK) {
                    timer[ReceiveData.ACK - base].on = false;
                }
                if (ReceiveData.ACK <= back) {
                    timer[MaxSeq + ReceiveData.ACK - base].on = false;
                }
            }
        }
    }
}

{
    lock_guard<mutex> lock(coutmutex);
    cout << "接收线程关闭!" << endl;
}
closed = true;
return;
}

```

5.接收窗口的实现:

- 设置序号空间大小和发送窗口大小:

```

// 接收数据时的序号空间大小
const int MaxSeq = 100;
//接收窗口大小
const int windowSize = 16;

```

- 与发送窗口一样，使用vector来实现接收窗口

```
// 接收窗口
vector<Datagram>ReceiveDataArray;
// 记录接收窗口中的序号
vector<int>ReceiveSeqArray;
// 记录接收窗口的分组是否收到
vector<int>isreceived;
```

isreceived数组中，0表示未接收到，1表示已接收到，当全部数据已接收完毕会将isreceived[0]置为-1。

为了方便管理接收窗口，定义了ReceiveSeqArray，专门存储数据包序号。

- 我专门创建了一个交付分组的线程，它不断遍历isreceived[0]，是否为1，即检查接收窗口中第一个分组是否接收到了，如果为1，就会将该分组上交给上层，并将接收窗口向后移。这样，就可以保证是**按序交给上层**。注意：ReceiveDataArray、ReceiveSeqArray和isreceived需要**同步变化**。当检测到isreceived[0]==-1时，就会退出。

交付分组线程代码如下：

```
//交付分组函数
void delivery_group(char filepath[],int& sequenceNumber) {
    // 打开文件用于写入接收的数据
    ofstream outputFile(filepath, ios::binary);
    if (!outputFile) {
        cerr << "Failed to open the output file." << endl;
        return ;
    }
    int flag;
    while (true) {
        {
            lock_guard<mutex> lock(mtx);
            flag = isreceived[0];
            if (flag == 1) {
                //交付分组
                outputFile.write(ReceiveDataArray.front().data,
ReceiveDataArray.front().length);
                //接收窗口向前移动
                ReceiveDataArray.erase(ReceiveDataArray.begin());
                Datagram temp;
                ReceiveDataArray.push_back(temp);
                isreceived.erase(isreceived.begin());
                isreceived.push_back(0);
                ReceiveSeqArray.erase(ReceiveSeqArray.begin());
                ++sequenceNumber;
                sequenceNumber %= MaxSeq;
                ReceiveSeqArray.push_back(sequenceNumber);
                {
                    lock_guard<mutex> lock(coutmutex);
                    cout << "发送窗口大小为: " << ReceiveSeqArray.size() << endl;
                    cout << "发送窗口的下沿为:" << ReceiveSeqArray.front()<< endl;
                    cout << "发送窗口的上沿为:" << ReceiveSeqArray.back()<< endl;
                }
            }
        }
        if (flag == -1) {
            outputFile.close();
            {
                lock_guard<mutex> lock(coutmutex);
```

```

        cout << "交付分组线程关闭!" << endl;
    }
    break;
}
}
}

```

6.乱序接收的实现

接收端接受一个没有错误的分组后，检查是否在接收窗口内（循环遍历），**如果在**，则将对应的 `isreceived[i]` 置为1，并给客户端发送这个分组的ACK。**如果不在**，接收窗口内，依然发送给客户端这个分组的ACK，告知其我已经接收到了，否则客户端会一直发送。注意，**因为序号空间大于等于窗口大小的2倍，且接收窗口是比发送窗口先向后移动的**，从而这个分组不可能是接收窗口后面的分组，所以这个分组一定是之前已经接受过的。

这个部分的代码如下：

```

if (ReceiveData.checksum != 0) {
    continue;
}
//检查接受的数据包是否在接收窗口内（乱序接收）
int tempseq;
for (int i = 0; i < windowSize; i++) {
    lock_guard<mutex> lock(mtx);
    tempseq = ReceiveSeqArray[i];
    if (tempseq == ReceiveData.seq) {
        ReceiveDataArray[i] = ReceiveData;
        isreceived[i] = 1;
        break;
    }
}
//将要发送的数据清空
memset(&SendData, 0, sizeof(Datagram));
SendData.ACK = ReceiveData.seq;
send_calculate_Checksum(SendData);
{
    lock_guard<mutex> lock(coutmutex);
    //打印发送的数据报
    cout << "发送数据报: ";
    printDatagram(SendData);
}
if (sendto(serverSocket, (char*)&SendData, sizeof(Datagram), 0, (struct
sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
    cerr << "发送错误!" << endl;
}
}

```

7.四次握手建立连接：

- 握手过程中的差错检验是靠校验和实现的，若有错误，则等待对方重发。
- 握手过程还需检验ack、ACK、SYN等标志位是否等于预期值，若不等于，则等待对方重发。
- 超时重传机制。
 - 规定发出握手后，时间不能超过1800秒。若超过，则从头再来。

```
const int maxtime = 1800; //1800秒
```

- 规定请求建立连接后，不能超过3600秒。否则，连接失败。

```
const int MAXTIME = 3600; //3600秒
```

- 相关代码如下：

```
/*-----发送端：主动握手-----*/
clock_t startTime = clock();
//主动发出握手
while (!active_shakehands(clientSocket, routerAddr,
sizeof(routerAddr))) {
    //超时
    if ((clock() - startTime) / CLOCKS_PER_SEC > MAXTIME) {
        cout << "连接不成功！" << endl;
        // 关闭套接字和清理winsock
        closesocket(clientSocket);
        WSACleanup();
        return -1;
    }
}
cout << "连接成功！" << endl;
/*-----接收端：被动握手-----*/
clock_t startTime = clock();
//被动接收握手
while (!passive_shakehands(serverSocket, routerAddr, routerAddrLen))
{
    //超时
    if ((clock() - startTime) / CLOCKS_PER_SEC > MAXTIME) {
        cout << "连接不成功！" << endl;
        // 关闭套接字和清理winsock
        closesocket(serverSocket);
        WSACleanup();
        return -1;
    }
}
cout << "连接成功！" << endl;
```

- 四次握手实现过程如下：（只有关键代码）

- 客户端（发送端）：

```
//主动握手
bool active_shakehands(SOCKET& clientsocket, sockaddr_in& routerAddr,
int routerAddrLen) {
    clock_t total_time = clock();
    while (true) {
        //超时连接不成功！
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
            return false;
        }
        //将要发送的数据清空
        memset(&SendData, 0, sizeof(Datagram));
```

```

// 将SYN设置为1, 表示要建立连接
SendData.SYN = 1;
//随机设定seq(1到100)
SendData.seq = (rand() % 100) + 1;
//计算校验和
send_calculate_Checksum(SendData);
if (sendto(clientsocket, (char*)&SendData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
    cerr << "发送错误!" << endl;
}
//将要接受的数据清空
memset(&ReceiveData, 0, sizeof(Datagram));
//定时器启动
clock_t start = clock();
bool istimeout = true;
while (clock() - start <= TIMEOUT) {
    // 接收数据
    if (recvfrom(clientsocket, (char*)&ReceiveData,
sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) <=
0) {
        continue;
    }
    istimeout = false;
    break;
}
if (!istimeout) {
    break;
}
}
cout << "第一次握手成功" << endl;
//计算校验和
receive_calculate_Checksum(ReceiveData);
//改为阻塞状态
if (!setBlocking(clientsocket)) {
    return false;
}
while (!(ReceiveData.ACK == 1 && ReceiveData.SYN == 1 &&
ReceiveData.ack == SendData.seq + 1 && ReceiveData.checksum == 0)) {
    //超时挥手失败!
    if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
        return false;
    }
    //将要接受的数据清空
    memset(&ReceiveData, 0, sizeof(Datagram));
    recvfrom(clientsocket, (char*)&ReceiveData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, &routerAddrLen);
    //计算校验和
    receive_calculate_Checksum(ReceiveData);
}
cout << "第二次握手成功!" << endl;
//改为非阻塞状态
if (!setNonBlocking(clientsocket)) {
    return false;
}
while (true) {
    //超时连接不成功!
    if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
        return false;
    }
}

```

```

    }
    //将要发送的数据清空
    memset(&SendData, 0, sizeof(Datagram));
    SendData.ACK = 1;
    SendData.seq = ReceiveData.ack;
    SendData.ack = ReceiveData.seq + 1;
    //计算校验和
    send_calculate_Checksum(SendData);
    if (sendto(clientsocket, (char*)&SendData, sizeof(Datagram), 0,
    (struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
        cerr << "发送错误! " << endl;
        return false;
    }
    //将要接受的数据清空
    memset(&ReceiveData, 0, sizeof(Datagram));
    //定时器启动
    clock_t start = clock();
    bool istimeout = true;
    while (clock() - start <= TIMEOUT) {
        // 接收数据
        if (recvfrom(clientsocket, (char*)&ReceiveData,
        sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) <=
        0) {
            continue;
        }
        istimeout = false;
        break;
    }
    if (!istimeout) {
        break;
    }
}
cout << "第三次握手成功" << endl;
cout << "第四次握手成功" << endl;
return true;
}

```

- 服务端（接收端）：

```

//被动握手
bool passive_shakehands(SOCKET& serversocket, sockaddr_in& routerAddr,
int routerAddrLen) {
    clock_t total_time = clock();
    //设置为阻塞状态
    if (!setBlocking(serversocket)) {
        return false;
    }
    while (true) {
        //超时连接不成功!
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
            return false;
        }
        //将要接受的数据清空
        memset(&ReceiveData, 0, sizeof(Datagram));
        // 接收数据
    }
}

```

```

        if (recvfrom(serversocket, (char*)&ReceiveData,
sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) ==
SOCKET_ERROR) {
            continue;
        }
        //计算校验和
        receive_calculate_Checksum(ReceiveData);
        if (!(ReceiveData.checksum == 0 && ReceiveData.SYN == 1)) {
            continue;
        }
        break;
    }
    cout << "第一次握手成功" << endl;
    //设置为非阻塞状态
    if (!setNonBlocking(serversocket)) {
        return false;
    }
    while (true) {
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
            return false;
        }
        //将要发送的数据清空
        memset(&SendData, 0, sizeof(Datagram));
        SendData.SYN = 1;
        SendData.ACK = 1;
        SendData.seq = (rand() % 100) + 1;
        SendData.ack = ReceiveData.seq + 1;
        //计算校验和
        send_calculate_Checksum(SendData);
        if (sendto(serversocket, (char*)&SendData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
            cerr << "发送错误!" << endl;
        }
        //将要接受的数据清空
        memset(&ReceiveData, 0, sizeof(Datagram));
        //定时器启动
        clock_t start = clock();
        bool istimeout = true;
        while (clock() - start <= TIMEOUT) {
            // 接收数据
            if (recvfrom(serversocket, (char*)&ReceiveData,
sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) <=
0) {
                continue;
            }
            istimeout = false;
            break;
        }
        if (!istimeout) {
            cout << "第二次握手成功" << endl;
            break;
        }
    }
    //计算校验和
    receive_calculate_Checksum(ReceiveData);
    //设置为阻塞状态
    if (!setBlocking(serversocket)) {
        return false;
    }

```



```

    }
    while (!(ReceiveData.ACK == 1 && ReceiveData.seq == SendData.ack &&
ReceiveData.ack == SendData.seq + 1 && ReceiveData.checksum == 0)) {
        //超时挥手失败!
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
            return false;
        }
        //将要接受的数据清空
        memset(&ReceiveData, 0, sizeof(Datagram));
        if (recvfrom(serversocket, (char*)&ReceiveData,
sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) ==
SOCKET_ERROR) {
            continue;
        }
        //计算校验和
        receive_calculate_Checksum(ReceiveData);
    }
    cout << "第三次握手成功" << endl;
    //将要发送的数据清空
    memset(&SendData, 0, sizeof(Datagram));
    //发送一个空包
    if (sendto(serversocket, (char*)&SendData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
        cerr << "发送错误!" << endl;
    }
    cout << "第四次握手成功" << endl;
    return true;
}

```

8.可靠数据传输 (SR)

- 客户端（发送端）（包括相关日志输出、传输时间、吞吐量、发送窗口大小及其变化、丢包提示）完全发送一个文件后，再调用该函数，发送另一个文件。

```

//传输文件
int transform_file(char filepath[], SOCKET& clientSocket, sockaddr_in&
routerAddr, int& routerAddrLen) {
    cout << "现在开始发送 " << filepath << endl;
    // 以二进制模式打开文件（图片或文本文件）
    ifstream file(filepath, ios::binary);
    if (!file) {
        cerr << "Failed to open the file." << endl;
        closesocket(clientSocket);
        WSACleanup();
        return 0;
    }
    //文件总字节
    int totalBytes = 0;
    // 序列号
    int sequenceNumber = 0; //从0开始
    // 初始化发送窗口
    SendDataArray.clear();
    timer.clear();
}

```

```

//开始传输时间
clock_t startTime = clock();
//启动接收线程
thread receiveACKThread(receiveACK, ref(clientSocket), ref(routerAddr),
ref(routerAddrLen));
receiveACKThread.detach();
//启动检测并处理超时事件的线程
thread Handle_TimeoutThread(Handle_Timeout, ref(clientSocket),
ref(routerAddr), ref(routerAddrLen));
Handle_TimeoutThread.detach();
//执行到文件末尾 (end of file, EOF)
while (true) {
    //发送数据
    while (SendDataArray.size() < windowSize) {
        if (file.eof()) {
            break;
        }
        //清空要发送的数据
        memset(&SendData, 0, sizeof(SendData));
        //设置分组序号
        SendData.seq = sequenceNumber++;
        sequenceNumber = sequenceNumber % MaxSeq;
        //每次从文件中读取最多BUFFER_SIZE 字节的数据
        file.read(SendData.data, BUFFER_SIZE);
        int bytesToSend = static_cast<int>(file.gcount());
        totalBytes += bytesToSend;
        //设置数据长度
        SendData.length = bytesToSend;
        //计算校验和
        send_calculate_Checksum(SendData);
        int size;
        int base;
        int back;
        //加入到窗口
        {
            lock_guard<mutex> lock(mtx);
            SendDataArray.push_back(SendData);
            Timer t = { clock(), true };
            timer.push_back(t);
            size = SendDataArray.size();
            base = SendDataArray.front().seq;
            back = SendDataArray.back().seq;
        }
        //打印发送的数据报
        {
            lock_guard<mutex> lock(coutmutex);
            cout << "发送数据报: ";
            printDatagram(SendData);
        }
        if (sendto(clientSocket, (char*)&SendData, sizeof(SendData), 0,
(SOCKADDR*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR) {
            cerr << "发送错误!" << endl;
        }
        {
            lock_guard<mutex> lock(coutmutex);
            cout << "发送窗口大小为: " << size<< endl;
            cout << "发送窗口的下沿为:" << base<< endl;
            cout << "发送窗口的上沿为:" << back<< endl;
        }
    }
}

```

```

    }

    }
    while (true) {
        lock_guard<mutex> lock(mtx);
        if (timer.size() > 0 && !timer[0].on) {
            SendDataArray.erase(SendDataArray.begin());
            timer.erase(timer.begin());
        }
        else {
            break;
        }
    }
    if (file.eof() && SendDataArray.size() == 0) {
        break;
    }
    //让其他线程运行
    sleep(5);
}
file.close();
clock_t endTime = clock();
double transferTime = static_cast<double>(endTime - startTime) /
CLOCKS_PER_SEC;
double throughput = (totalBytes / 1024) / transferTime; // 计算吞吐量 (单
位: KB/s)
cout << filepath << "传输完毕" << endl;
cout << "传输文件大小为: " << totalBytes / 1024 << "KB" << endl;
cout << "传输时间为: " << transferTime << "s" << endl;
cout << "吞吐量为: " << throughput << "KB/s" << endl;
finished = true;
while (finished) {}
return 1;
}

```

- 服务器（接收端）：（包括相关日志输出和接收窗口大小及其变化）

接收完一个文件，再调用该函数，接收另一个文件。

```

//接收文件
int receive_file(char filepath[], SOCKET& serverSocket, sockaddr_in&
routerAddr, int& routerAddrLen) {
    cout << "开始接收" << filepath << endl;
    int sequenceNumber = -1; //规定发送的第一个序号为0
    //初始化接收窗口
    ReceiveSeqArray.clear();
    ReceiveDataArray.clear();
    isreceived.clear();
    while (ReceiveSeqArray.size() < windowSize) {
        ReceiveSeqArray.push_back(++sequenceNumber);
        Datagram temp;
        ReceiveDataArray.push_back(temp);
        isreceived.push_back(0);
        if (sequenceNumber >= MaxSeq) {
            sequenceNumber %= MaxSeq;
        }
    }
}

```

```

//启动交付线程
thread DeliveryThread(delivery_group, filepath, ref(sequenceNumber));
DeliveryThread.detach();
// 接收并重组数据包
while (true) {
    memset(&ReceiveData, 0, sizeof(Datagram));
    int receivedBytes = recvfrom(serverSocket, (char*)&ReceiveData,
    sizeof(ReceiveData), 0, (SOCKADDR*)&routerAddr, &routerAddrLen);
    if (receivedBytes == SOCKET_ERROR) {
        continue;
    }
    receive_calculate_Checksum(ReceiveData);
    {
        lock_guard<mutex> lock(coutmutex);
        //打印接受的数据报
        cout << "接收数据报: ";
        printDatagram(ReceiveData);
    }
    if (ReceiveData.FIN == -1 && ReceiveData.checksum == 0) {
        lock_guard<mutex> lock(mtx);
        isreceived[0] = -1;
        break;
    }
    if (ReceiveData.checksum != 0) {
        continue;
    }
    //检查接受的数据包是否在接收窗口内（乱序接收）
    int tempseq;
    for (int i = 0; i < windowSize; i++) {
        lock_guard<mutex> lock(mtx);
        tempseq = ReceiveSeqArray[i];
        if (tempseq == ReceiveData.seq) {
            ReceiveDataArray[i] = ReceiveData;
            isreceived[i] = 1;
            break;
        }
    }
    //将要发送的数据清空
    memset(&SendData, 0, sizeof(Datagram));
    SendData.ACK = ReceiveData.seq;
    send_calculate_Checksum(SendData);
    {
        lock_guard<mutex> lock(coutmutex);
        //打印发送的数据报
        cout << "发送数据报: ";
        printDatagram(SendData);
    }
    if (sendto(serverSocket, (char*)&SendData, sizeof(Datagram), 0,
    (struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
        cerr << "发送错误!" << endl;
    }
}
//将要发送的数据清空
memset(&SendData, 0, sizeof(Datagram));
SendData.FIN = -1;
//告诉客户端我知道文件传输完了
if (sendto(serverSocket, (char*)&SendData, sizeof(Datagram), 0, (struct
sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {

```

```

        cerr << "发送错误!" << endl;
    }
    cout << filepath << " 文件接收完毕!" << endl;
    return 1;
}

```

9.五次挥手断开连接:

- 客户端（发送端）：（只粘贴关键代码）

```

//主动挥手
bool active_wakehands(SOCKET& clientsocket, sockaddr_in& routerAddr, int
routerAddrLen) {
    clock_t total_time = clock();
    while (true) {
        //超时挥手失败!
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
            return false;
        }
        //将要发送的数据清空
        memset(&SendData, 0, sizeof(Datagram));
        // 将FIN设置为1, 表示要d断开连接
        SendData.FIN = 1;
        //随机设定seq(1到100)
        SendData.seq = (rand() % 100) + 1;
        //计算校验和
        send_calculate_checksum(SendData);
        if (sendto(clientsocket, (char*)&SendData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
            cerr << "发送错误!" << endl;
        }
        //将要接受的数据清空
        memset(&ReceiveData, 0, sizeof(Datagram));
        //定时器启动
        clock_t start = clock();
        bool istimeout = true;
        while (clock() - start <= TIMEOUT) {
            // 接收数据
            if (recvfrom(clientsocket, (char*)&ReceiveData,
sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) <= 0) {
                continue;
            }
            istimeout = false;
            break;
        }
        if (!istimeout) {
            break;
        }
    }
    cout << "第一次挥手成功" << endl;
    //计算校验和
    receive_calculate_checksum(ReceiveData);
    //改为阻塞状态

```

```

        if (!setBlocking(clientsocket)) {
            return false;
        }
        while (!(ReceiveData.ACK == 1 && ReceiveData.ack == SendData.seq + 1 &&
ReceiveData.checksum == 0)) {
            //超时挥手失败!
            if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
                return false;
            }
            //将要接受的数据清空
            memset(&ReceiveData, 0, sizeof(Datagram));
            recvfrom(clientsocket, (char*)&ReceiveData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, &routerAddrLen);
            //计算校验和
            receive_calculate_Checksum(ReceiveData);
        }
        cout << "第二次挥手成功! " << endl;
        //将要接受的数据清空
        memset(&ReceiveData, 0, sizeof(Datagram));
        while (!(ReceiveData.ACK == 1 && ReceiveData.FIN == 1 && ReceiveData.ack
== SendData.seq + 1 && ReceiveData.checksum == 0)) {
            //超时挥手失败!
            if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
                return false;
            }
            //将要接受的数据清空
            memset(&ReceiveData, 0, sizeof(Datagram));
            if (recvfrom(clientsocket, (char*)&ReceiveData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, &routerAddrLen) == SOCKET_ERROR) {
                continue;
            }
            //计算校验和
            receive_calculate_Checksum(ReceiveData);
        }
        cout << "第三次挥手成功" << endl;
        //改为非阻塞状态
        if (!setNonBlocking(clientsocket)) {
            return false;
        }
        while (true) {
            //超时挥手失败!
            if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
                return false;
            }
            //将要发送的数据清空
            memset(&SendData, 0, sizeof(Datagram));
            SendData.ACK = 1;
            SendData.seq = ReceiveData.ack;
            SendData.ack = ReceiveData.seq + 1;
            //计算校验和
            send_calculate_Checksum(SendData);
            if (sendto(clientsocket, (char*)&SendData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
                cerr << "发送错误! " << endl;
            }
            //定时器启动
            clock_t start = clock();
            bool istimeout = true;

```

```

        while (clock() - start <= TIMEOUT) {
            // 接收数据
            if (recvfrom(clientsocket, (char*)&ReceiveData,
                sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) <= 0) {
                continue;
            }
            istimeout = false;
            break;
        }
        if (!istimeout) {
            break;
        }
    }
    cout << "第四次挥手成功" << endl;
    cout << "第五次挥手成功" << endl;
    return true;
}

```

- 服务器（接收端）：（只粘贴关键代码）

```

//被动挥手
bool passive_wakehands(SOCKET& serversocket, sockaddr_in& routerAddr, int
routerAddrLen) {
    clock_t total_time = clock();
    //设置为阻塞状态
    if (!setBlocking(serversocket)) {
        return false;
    }
    while (true) {
        //超时挥手失败！
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
            return false;
        }
        //将要接受的数据清空
        memset(&ReceiveData, 0, sizeof(Datagram));
        // 接收数据
        if (recvfrom(serversocket, (char*)&ReceiveData, sizeof(Datagram), 0,
            (struct sockaddr*)&routerAddr, &routerAddrLen) == SOCKET_ERROR) {
            continue;
        }
        //计算校验和
        receive_calculate_Checksum(ReceiveData);
        //打印接受的数据报
        cout << "接收数据报: ";
        printDatagram(ReceiveData);
        if (!(ReceiveData.checksum == 0 && ReceiveData.FIN == 1)) {
            continue;
        }
        break;
    }
    cout << "第一次挥手成功" << endl;
    //设置为非阻塞状态
    if (!setNonBlocking(serversocket)) {
        return false;
    }
    //将要发送的数据清空
    memset(&SendData, 0, sizeof(Datagram));
}

```

```

SendData.ACK = 1;
SendData.seq = (rand() % 100) + 1;
SendData.ack = ReceiveData.seq + 1;
//计算校验和
send_calculate_Checksum(SendData);
if (sendto(serversocket, (char*)&SendData, sizeof(Datagram), 0, (struct
sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
    cerr << "发送错误!" << endl;
}
cout << "第二次挥手成功" << endl;
while (true) {
    //超时挥手失败!
    if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
        return false;
    }
    //将要发送的数据清空
    memset(&SendData, 0, sizeof(Datagram));
    SendData.ACK = 1;
    SendData.FIN = 1;
    SendData.seq = (rand() % 100) + 1;
    SendData.ack = ReceiveData.seq + 1;
    //计算校验和
    send_calculate_Checksum(SendData);
    if (sendto(serversocket, (char*)&SendData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
        cerr << "发送错误!" << endl;
    }
    //将要接受的数据清空
    memset(&ReceiveData, 0, sizeof(Datagram));
    //定时器启动
    clock_t start = clock();
    bool istimeout = true;
    while (clock() - start <= TIMEOUT) {
        // 接收数据
        if (recvfrom(serversocket, (char*)&ReceiveData,
sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) <= 0) {
            continue;
        }
        istimeout = false;
        break;
    }
    if (!istimeout) {
        break;
    }
}
cout << "第三次挥手成功" << endl;
//计算校验和
receive_calculate_Checksum(ReceiveData);
//设置为阻塞状态
if (!setBlocking(serversocket)) {
    return false;
}
while (!(ReceiveData.ACK == 1 && ReceiveData.seq == SendData.ack &&
ReceiveData.ack == SendData.seq + 1 && ReceiveData.checksum == 0)) {
    //超时挥手失败!
    if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
        return false;
    }
}

```



```

        //将要接受的数据清空
        memset(&ReceiveData, 0, sizeof(Datagram));
        recvfrom(serversocket, (char*)&ReceiveData, sizeof(Datagram), 0,
        (struct sockaddr*)&routerAddr, &routerAddrLen);
        //计算校验和
        receive_calculate_checksum(ReceiveData);
    }
    cout << "第四次挥手成功" << endl;
    //随便发送个消息，告诉他挥手结束
    if (sendto(serversocket, (char*)&SendData, sizeof(Datagram), 0, (struct
sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
        cerr << "发送错误!" << endl;
    }
    cout << "第五次挥手成功" << endl;
    return true;
}

```

四、实验结果：

1.四次握手建立连接:

①发送端：

```

发送数据报： Seq: 95  ACK: 0  FIN: 0
校验和： 65439  数据长度： 0
第一次握手成功
接收数据报： Seq: 37  ACK: 1  FIN: 0
校验和： 0  数据长度： 0
第二次握手成功！
发送数据报： Seq: 96  ACK: 1  FIN: 0
校验和： 65400  数据长度： 0
第三次握手成功
第四次握手成功
连接成功！
请输入你想传送文件的数量：
|

```

②接收端：

```
UDP server is listening on port 8080...
接收数据报: Seq: 95  ACK: 0  FIN: 0
  校验和: 0  数据长度: 0
第一次握手成功
发送数据包: Seq: 37  ACK: 1  FIN: 0
  校验和: 65400  数据长度: 0
第二次握手成功
接收数据报: Seq: 96  ACK: 1  FIN: 0
  校验和: 0  数据长度: 0
第三次握手成功
第四次握手成功
连接成功!
```

2.文件传输:

图片1:

发送端开始发送:

```
请输入你想传送文件的数量:
4
发送数据报: Seq: 0  ACK: 0  FIN: 0
  校验和: 65531  数据长度: 0
发送数据报: Seq: 0  ACK: 0  FIN: -1
  校验和: 0  数据长度: 0
请输入你想传送的第1个文件:
1
发送数据报: Seq: 0  ACK: 0  FIN: 0
  校验和: 65534  数据长度: 0
发送数据报: Seq: 0  ACK: 0  FIN: -1
  校验和: 0  数据长度: 0
现在开始发送 1.jpg
处理超时线程开启!
接收线程开启!
发送数据报: Seq: 0  ACK: 0  FIN: 0
  校验和: 47914  数据长度: 10240
```

接收端开始接收:

```
接受的文件为文件1
开始接收 ./1.jpg
交付分组线程开启!
接收数据报: Seq: 0 ACK: 0 FIN: 0
校验和: 0 数据长度: 10240
发送数据报: Seq: 0 ACK: 0 FIN: 0
校验和: 65535 数据长度: 0
```

数据传输过程中，发送窗口边界及其大小变化：（窗口大小为16，序号空间大小为100）

①刚开始发送时，窗口的变化

```
发送窗口大小为: 4
发送窗口的下沿为: 0
发送窗口的上沿为: 3
发送数据报: Seq: 4 ACK: 0 FIN: 0
校验和: 16189 数据长度: 10240
发送窗口大小为: 5
发送窗口的下沿为: 0
发送窗口的上沿为: 4
发送数据报: Seq: 5 ACK: 0 FIN: 0
校验和: 28900 数据长度: 10240
发送窗口大小为: 6
发送窗口的下沿为: 0
发送窗口的上沿为: 5
发送数据报: Seq: 6 ACK: 0 FIN: 0
校验和: 8030 数据长度: 10240
发送窗口大小为: 7
发送窗口的下沿为: 0
发送窗口的上沿为: 6
```

②发送窗口满了后的变化

```
发送窗口大小为：1
发送窗口的下沿为：16
发送窗口的上沿为：16
发送数据报： Seq: 17    ACK: 0    FIN: 0
    校验和： 34613    数据长度： 10240
发送窗口大小为：2
发送窗口的下沿为：16
发送窗口的上沿为：17
发送数据报： Seq: 18    ACK: 0    FIN: 0
    校验和： 8621    数据长度： 10240
发送窗口大小为：3
发送窗口的下沿为：16
发送窗口的上沿为：18
发送数据报： Seq: 19    ACK: 0    FIN: 0
    校验和： 50195    数据长度： 10240
发送窗口大小为：4
发送窗口的下沿为：16
发送窗口的上沿为：19
发送数据报： Seq: 20    ACK: 0    FIN: 0
    校验和： 44485    数据长度： 10240
发送窗口大小为：5
发送窗口的下沿为：16
发送窗口的上沿为：20
```

数据传输过程中，接收窗口边界及其大小变化：（窗口大小为16，序号空间大小为100）

```
接收窗口大小为：16
接收窗口的下沿为：78
接收窗口的上沿为：93
接收数据报： Seq: 78 ACK: 0 FIN: 0
校验和： 0 数据长度： 10240
发送数据报： Seq: 0 ACK: 78 FIN: 0
校验和： 65457 数据长度： 0
接收窗口大小为：16
接收窗口的下沿为：79
接收窗口的上沿为：94
接收数据报： Seq: 79 ACK: 0 FIN: 0
校验和： 0 数据长度： 10240
发送数据报： Seq: 0 ACK: 79 FIN: 0
校验和： 65456 数据长度： 0
接收窗口大小为：16
接收窗口的下沿为：80
接收窗口的上沿为：95
```

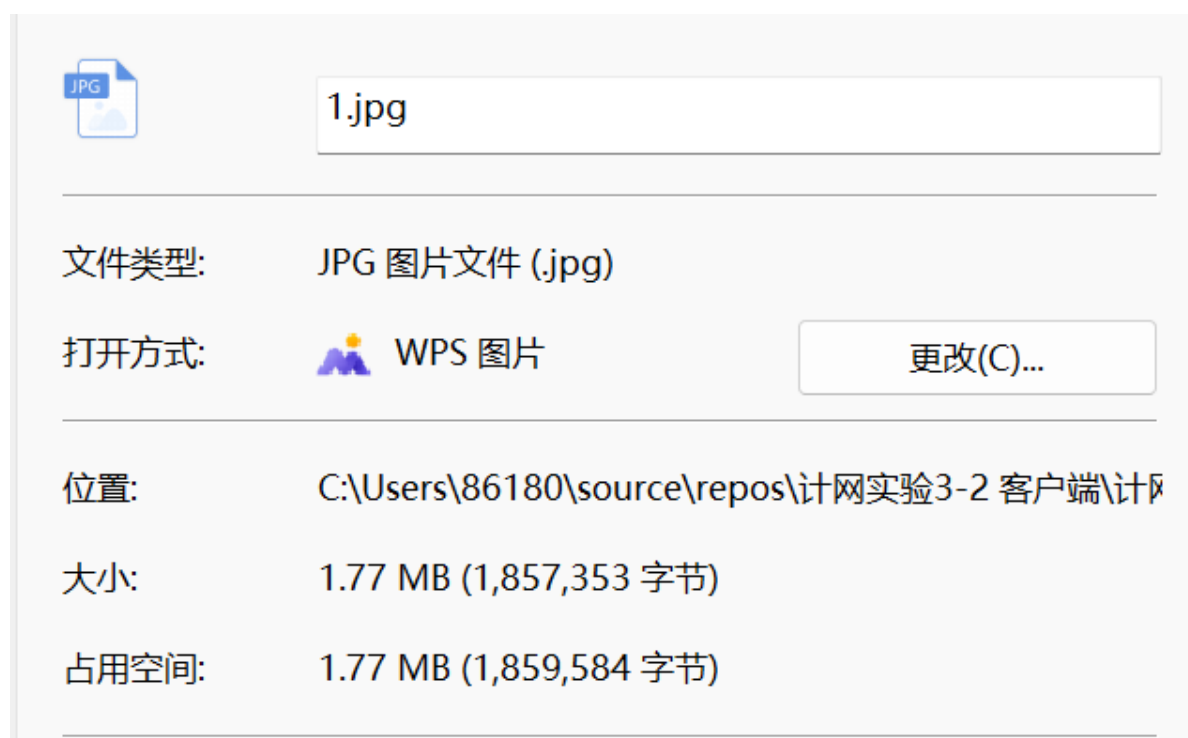
发送过程中，分组丢失情况：

```
接收数据报： Seq: 0 ACK: 27 FIN: 0
校验和： 0 数据长度： 0
接收数据报： Seq: 0 ACK: 28 FIN: 0
校验和： 0 数据长度： 0
分组13丢了！！！重新发送分组13及其之后的分组
发送数据报： Seq: 13 ACK: 0 FIN: 0
校验和： 47797 数据长度： 10240
接收数据报： Seq: 0 ACK: 13 FIN: 0
校验和： 0 数据长度： 0
```

发送端发送完毕，并显示传输时间和吞吐量：

```
1.jpg传输完毕
传输文件大小为： 1813KB
传输时间为： 8.785s
吞吐量为： 206.375KB/s
发送数据报： Seq: 0  ACK: 0  FIN: -1
校验和： 0  数据长度： 0
发送数据报： Seq: 0  ACK: 0  FIN: -1
校验和： 0  数据长度： 0
接收线程关闭！
处理超时线程关闭！
```

图片1的字节数如下图， $1857353/1024=1813.82$ ，因为是整型变量，所以结果为1813，结果正确。



接收端接收完毕：

```
./1.jpg 文件接收完毕！
交付分组线程关闭！
```

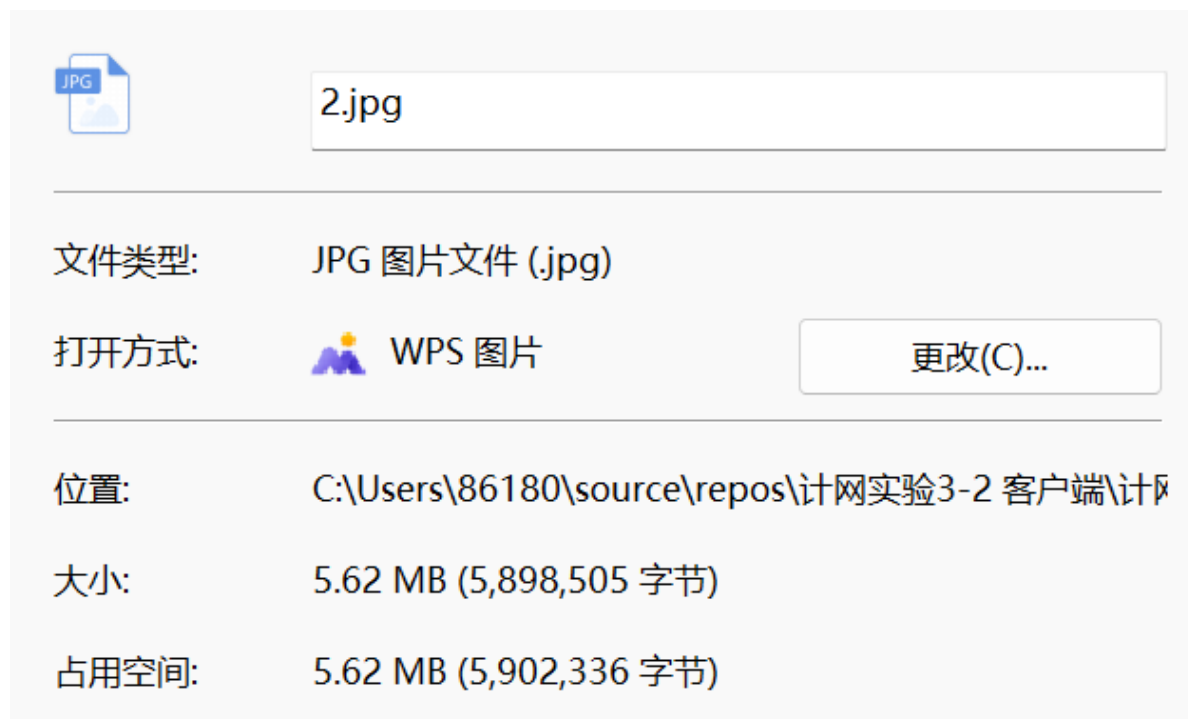
注：由于文件传输过程基本差不多，接下来只显示文件的大小、传输时间和吞吐量。

图片2：

发送端发送完毕，并显示传输时间和吞吐量：

2.jpg传输完毕
传输文件大小为： 5760KB
传输时间为： 41.604s
吞吐量为： 138.448KB/s

图片2的字节数如下图， $5898505/1024=5760.26$ ，因为是整型变量，所以结果为5760，结果正确。



接收端接收完毕：

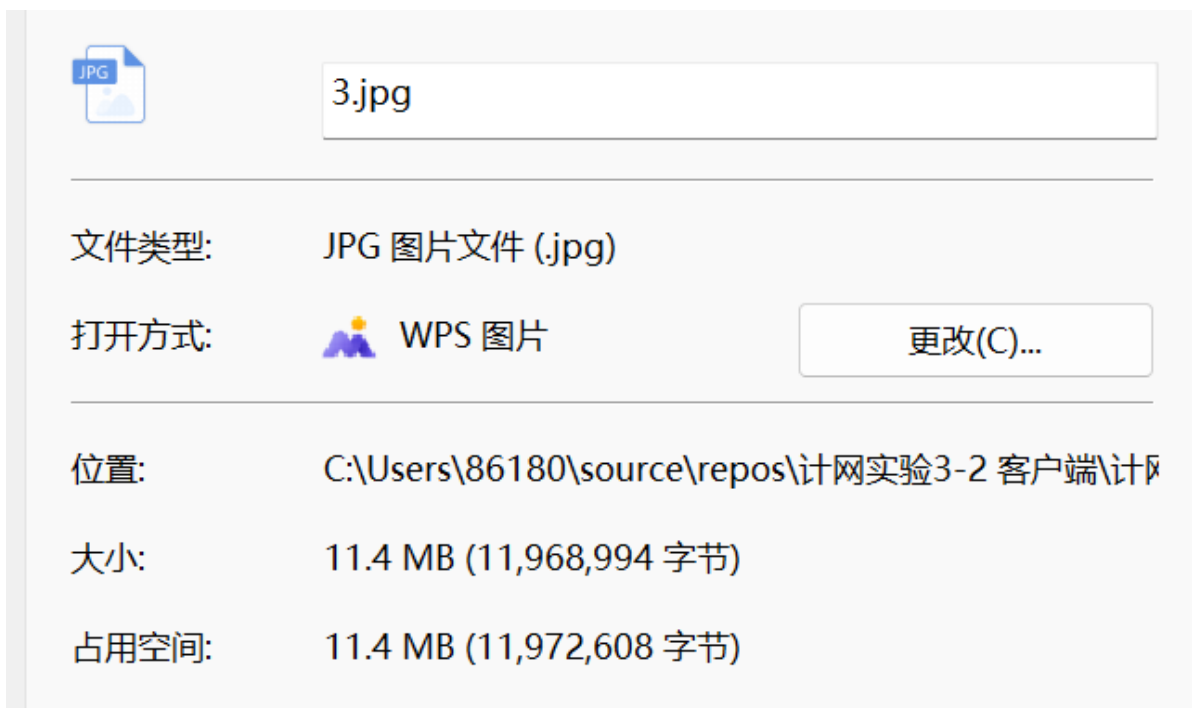
./2.jpg 文件接收完毕！
现在接收第3个文件
交付分组线程关闭！

图片3：

发送端发送完毕，并显示传输时间和吞吐量：

3.jpg传输完毕
传输文件大小为： 11688KB
传输时间为： 84.195s
吞吐量为： 138.821KB/s
发送数据报： Seq: 0 ACK: 0 FIN: -1
校验和： 0 数据长度： 0
接收线程关闭！
处理超时线程关闭！

图片3的字节数如下图， $11968994/1024=11688.47$ ，因为是整型变量，所以结果为11688，结果正确。




文档:

发送端发送完毕，并显示传输时间和吞吐量:

```
helloworld.txt传输完毕
传输文件大小为: 1617KB
传输时间为: 11.544s
吞吐量为: 140.073KB/s
发送数据报: Seq: 0 ACK: 0 FIN: -1
校验和: 0 数据长度: 0
接收线程关闭!
处理超时线程关闭!
```

文档的字节数如下图， $1655808/1024=1617$ ，正好整除，所以结果为1617，结果正确。




helloworld.txt

文件类型:

文本文档 (.txt)

打开方式:

 记事本

更改(C)...

位置:

C:\Users\86180\source\repos\计网实验3-2 客户端\计网

大小:

1.57 MB (1,655,808 字节)

占用空间:

1.58 MB (1,658,880 字节)

接收端接收完毕:

./helloworld.txt 文件接收完毕!

3.路由器日志:

Router

路由器IP: 127 . 0 . 0 . 1

服务器IP: 127 . 0 . 0 . 1

端口: 8081

服务器端口: 8080

丢包率: 5 %

延时: 5 ms

确定

修改

日志

```
count:17.
Delay 5 ms.
count:18.
Delay 5 ms.
count:19.
Delay 5 ms.
Miss a packet.
Delay 5 ms.
count:1.
Delay 5 ms.
count:2
```

4.和客户端的文件对比：

名称	修改日期	类型	大小
x64	2023/11/7 18:10	文件夹	
1.jpg	2023/11/1 11:35	JPG 图片文件	1,814 KB
2.jpg	2023/11/1 11:35	JPG 图片文件	5,761 KB
3.jpg	2023/11/1 11:35	JPG 图片文件	11,689 KB
helloworld.txt	2023/11/1 11:35	文本文档	1,617 KB
计网实验3-1 客户端.cpp	2023/11/12 15:55	C++ Source	16 KB
计网实验3-1 客户端.vcxproj	2023/11/8 18:52	VC++ Project	7 KB
计网实验3-1 客户端.vcxproj.filters	2023/11/8 18:52	VC++ Project Filter...	1 KB
计网实验3-1 客户端.vcxproj.user	2023/11/7 17:18	Per-User Project O...	1 KB
x64	2023/11/22 20:54	文件夹	
计网实验3-2 服务器.cpp	2023/11/23 18:34	C++ Source	18 KB
计网实验3-2 服务器.vcxproj	2023/11/18 17:13	VC++ Project	7 KB
计网实验3-2 服务器.vcxproj.filters	2023/11/18 17:13	VC++ Project Filter...	1 KB
计网实验3-2 服务器.vcxproj.user	2023/11/18 16:56	Per-User Project O...	1 KB
1.jpg	2023/11/24 18:02	JPG 图片文件	1,814 KB
2.jpg	2023/11/24 18:13	JPG 图片文件	5,761 KB
3.jpg	2023/11/24 18:28	JPG 图片文件	11,689 KB
helloworld.txt	2023/11/24 18:30	文本文档	1,617 KB

文件大小一致，且可以正常打开

5.五次挥手断开连接：

发送端：

```
发送数据报: Seq: 34 ACK: 0 FIN: 1
校验和: 65500 数据长度: 0
第一次挥手成功
接收数据报: Seq: 36 ACK: 1 FIN: 0
校验和: 0 数据长度: 0
第二次挥手成功!
接收数据报: Seq: 26 ACK: 1 FIN: 1
校验和: 0 数据长度: 0
第三次挥手成功
发送数据报: Seq: 35 ACK: 1 FIN: 0
校验和: 65472 数据长度: 0
第四次挥手成功
第五次挥手成功
已断开连接
请按任意键继续. . .
```

接收端:

```
接收数据报: Seq: 34 ACK: 0 FIN: 1
校验和: 0 数据长度: 0
第一次挥手成功
发送数据包: Seq: 36 ACK: 1 FIN: 0
校验和: 65463 数据长度: 0
第二次挥手成功
发送数据包: Seq: 26 ACK: 1 FIN: 1
校验和: 65472 数据长度: 0
第三次挥手成功
接收数据报: Seq: 35 ACK: 1 FIN: 0
校验和: 0 数据长度: 0
第四次挥手成功
第五次挥手成功
已断开连接
请按任意键继续. . .
```

五、实验总结

1.本次实验为了测试方便，**新增**通过输入的方式来选择想传的文件数量和文件序号的功能。客户端会把想传文件的数量和文件序号发给服务器。（由于不是关键功能，未在实验报告附上代码，详见.cpp文件）

2.本次实验让我感受到了多线程编程困难，在访问公有数据时，一定要上互斥锁。还有要减少线程的数量，一开始我为每一个分组都开了一个专门的线程来检测超时事件，这就会产生一些意向不到的事情。且即使开很多线程，由于锁的机制，在一些地方，依然不能并行进行，需要等待其他线程的锁打开后，才可以运行，效率并没有提高多少。

3.SR要求序号空间大小大于等于窗口大小的2倍的原因：现实当中序号并不是无限递增下去的，而是**循环使用的**，当SR接收窗口太大时，**接收方会出现无法分辨刚到的分组是一次重传还是新的分组。**