计算机网络 3-3

个人信息

- 学号：2013750
- 姓名：管昀玫
- 专业：计算机科学与技术

实验要求

在实验3-2的基础上，选择实现一种拥塞控制算法，也可以是改进的算法，完成给定测试文件的传输。

协议设计

本次实验为实现单向传输，并且顺序较为固定。

- 对于客户端：建立链接（主动） -> 选择文件发送 -> 关闭连接（主动）
- 对于服务器：建立连接（被动） -> 接收文件 -> 关闭连接（被动）

本次基于的协议是**拥塞控制算法和快速重传机制**，并根据个人用法稍微做出了一点改变：为了减少数据的传输量与方便编程，本次协议设计时在建立/关闭连接和发送文件内容时使用了两套协议（即两套不同的结构）

1. 建立与关闭连接

格式设计

在建立和关闭连接时，最主要的标志位是SYN和FIN，这两个标志位就能告诉对方本次的数据包的作用。

- SYN同步标志：该标志仅在三次握手建立TCP连接时有效。在三次握手期间，SYN被置为1。
- FIN断开标志：带有该标志置位的数据包用来结束一个TCP回话，在挥手期间被置为1。

TCP协议除了SYN和FIN标志位以外还有另外4个标志位；但本次实验并不会用到这么多功能，因此进行简化处理，去掉另外4个标志位，而增加了另外一个标志位FLAG。

- FLAG标志位：为了避免因为翻转导致歧义，除去SYN、FIN和校验和之外，数据包中还增加了一个FLAG位用来标志本次数据包的作用。此标志位标明是握手还是挥手；且标明是第几次握手/挥手。

简单起见，建立连接是保持三次握手，但关闭连接时只进行两次挥手。

FLAG标志位设计如下：

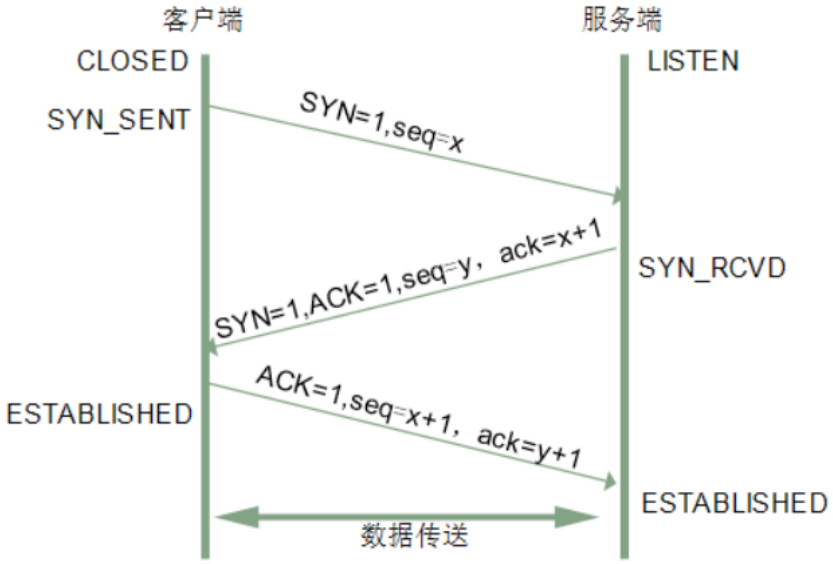
```
// 三次握手的时候对应的flag
const unsigned char SHAKE1 = 0x01;
const unsigned char SHAKE2 = 0x02;
const unsigned char SHAKE3 = 0x03;
// 四次挥手的时候的flag
const unsigned char WAVE1 = 0x04;
const unsigned char WAVE2 = 0x05;
```

数据包格式：`char package[4]`，具体为：

```
package[0]----校验和
package[1]----SYN
package[2]----FIN
package[3]----FLAG
```

建立连接 - 三次握手

1. 由client发起建立连接请求，发送时需要将**SYN置1**，**FIN置0**；第一次握手时**FLAG**需要设置为`SHAKE1`。
2. server接收到第一次握手内容后，若不正确则不做任何相应，让client自行超时重传；若数据包正确，则发送**SYN为1**，**FIN为0**，**FLAG为**`SHAKE2`的数据包。
3. client收到来自server的第二次握手内容后，检查数据包，如果不正确，不做回复，等待server的超时重传；如果正确，向server发送**SYN为1**，**FIN为0**，**FLAG为**`SHAKE3`的数据包。
4. server收到来自client的第三次握手内容，检查数据包，如果不正确，回到最开始的状态重新等待连接；如果正确，回显连接建立。



如图，模仿的TCP三次握手，但没有设计ACK/seq/ack位

关闭连接 - 两次挥手

1. 由client发起关闭连接请求，发送时需要将**SYN置0**，**FIN置1**；第一次挥手时FLAG需要设置为**WAVE1**。
2. server接收到第一次挥手内容后，若不正确则不做任何相应，让client自行超时重传；若数据包正确，则发送**SYN为0**，**FIN为1**，FLAG为**WAVE2**的数据包，并从服务器这边关闭连接。
3. client收到来自server的第二次挥手内容后，检查数据包，如果不正确，重新发送挥手数据包（重发的原因是猜测可能服务器那边没有收到挥手数据包，而在这里因为只有两次挥手，在这里可以简单当作收到第二次挥手的时候无论是否正确直接关闭连接），等待server的超时重传；如果正确，关闭连接。

2. 数据传输

格式设计

由于本次是单项传输：

- 对于client端来说，要发送校验和、seq位、数据长度、判断是否为最后一个包的标志位以及数据内容，因此比较复杂；
- 对于server端来说，只需要回复Ack即可（client作为发送端，server作为接收端），另外需要一个校验和，因此数据包格式比较简单

由于发送端和接收端复杂程度不同，为了简化，设计成两种不同的数据格式。

发送端：每次传送的数据包大小均为1024个字节，具体内容为：

```
data[1024];
data[0]----校验和
data[1]----序号seq最高8位
data[2]----序号seq中间8位
data[3]----序号seq最低8位
data[4]----数据长度高8位
data[5]----数据长度低8位
data[6]----isLast位，用于当前数据包是否为最后一个数据包，是为1，反之为0
data[7~1023]----要传输的数据内容，长度与data[3]、data[4]中算出来的数相同
```

接收端：传输内容仅为2个字节，具体内容为：

```
answer[4];
answer[0]----校验和
answer[1]----ACK最高8位
answer[2]----ACK中间8位
answer[3]----ACK最低8位
```

协议内容：拥塞控制和快速重传

> **拥塞控制**就是为了防止过多的数据注入到网络中，这样可以使网络中的路由器或者链路不至于过载。拥塞控制要做的都有一个前提：就是网络能够承受现有的网络负荷。对比**流量控制**：拥塞控制是一个全局的过程，涉及到所有的主机、路由器、以及降低网络相关的所有因素。流量控制往往指点对点通信量的控制，是端对端的问题。

以下数据包在封装时默认都计算过校验和，并默认在收到数据包时都会检查校验和，校验和出错和下面其他位的出错处理方式相同。

本次实现的协议内容基于AIMD算法并加以改动。

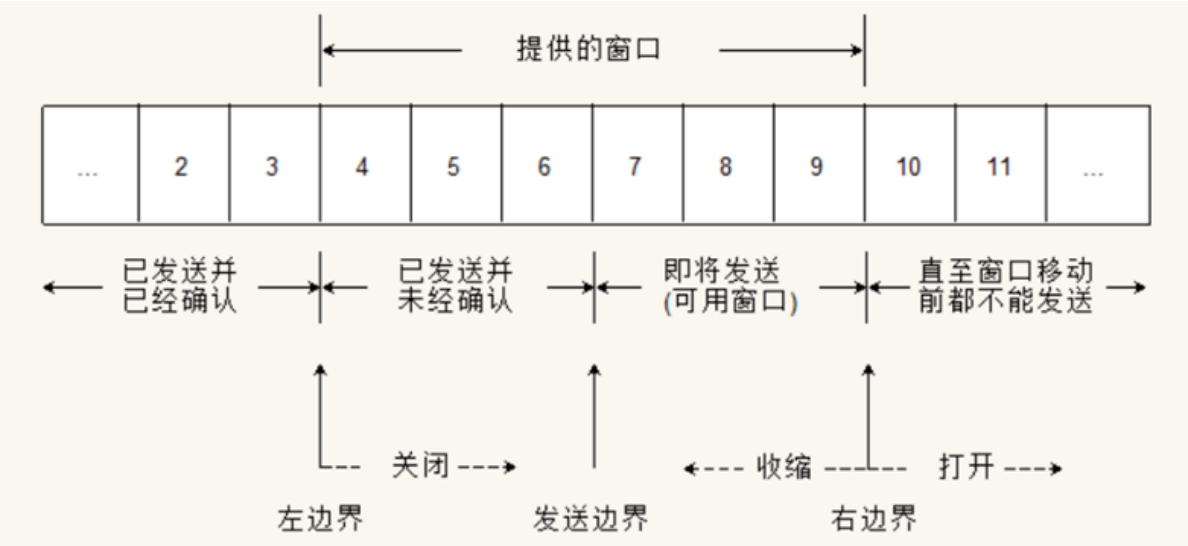
```
拥塞控制相关算法：
    1. 当cwnd < ssthresh,使用慢启动算法，
    2. 当cwnd > ssthresh,使用拥塞避免算法，停用慢启动算法。
    3. 当cwnd = ssthresh，这两个算法都可以。
慢启动算法：1-2-4-8-……
拥塞避免算法：1-2-3-4-……
```

发送端

对于发送端来说，需要维护的数据结构有：

- 阈值`sshtresh`，初始设置为16
- 拥塞窗口`cwnd`，初始设置为1
- 重复收到ack的数量`sameAckTimes`，初始设置为0
- 收到确认的最大ack值为`lastAck`，初始设置为 -1
- 下一个将要发送的数据包编号`curr`

相关执行动作为：



1. 发送当前拥塞窗口`cwnd`内所有的数据包，并在最后一个数据包发送结束后开始计时，进入等待阶段
2. 当收到来自接收端的回复，如果`Ack == lastAck + 1`，则`lastAck ++`，并设置`sameAckTimes = 1`；如果`Ack == lastAck`，则`sameAckTimes += 1`；若收到的`Ack > lastAck`，则说明之前服务器端回给客户端的数据包有丢失，但由于客户端处的处理机制是返回最后一个递交上层的数据包，此时只要置`lastAck = Ack`即可。
3. 在接收的同时发送（两个线程实现收和发）。如果中间出现3次重复ACK，执行**快速重传**：`ssthresh = max(cwnd / 2, 2)`，`cwnd = ssthresh + 3`；如果出现超时，`ssthresh = max(cwnd / 2, 2)`，`cwnd = 1`，并回到**慢启动**阶段（同时需要将记录发送的数据包设置为`lastAck + 1`，重新发送刚刚丢失或者需要重传的数据包）

接收端

接收端需要维护的数据结构有：

- 记录当前需要接收的数据编号`currentNum`

相关执行动作为：

1. 在一轮接收数据包后，逐个计算收到的`seq`，如果`seq == currentNum`，则将当前数据包传递给上层，并返回当前`ack`，将`currentNum + 1`；
2. 其他情况下均回复的`ack`为`currentNum - 1`（即当前已交付上层的最后一个`ack`），表示收到的数据包有问题。
3. 只要收到数据包，就重新开始计时。

计时器

双方各自都会维护一个计时器，当时间为-1的时候表示停止计时，其他均为正常计时。

且发送端的计时器是从当前轮中发送的最后一个包开始计时；如果中间有出现重传， 则以当前重传的最后一个包为基准开始计时；接收端则从回复ack开始计时，每次回复都会刷新。

程序设计

本次程序流程和上述协议的流程相同，这里主要介绍一些重点代码。

拥塞控制与快速重传

1. 慢启动

如果立即将大量的数据注入到网络可能会出现网络的拥塞。慢启动算法就是在刚开始发送数据报的时候先探测一下网络的状况，如果网络状况良好，发送方每发送一次报文都能正确的接受确认报文段，那么就从小到大的增加拥塞窗口的大小，即增加发送窗口的大小。

发送方先设置拥塞窗口`cwnd = 1`，发送第一个报文段M1，接收方接收到M1后，发送方接收到接收方的确认后，把`cwnd`增加到2，接着发送方发送M2、M3，发送方接收到接收方发送的确认后`cwnd`增加到4，慢启动算法每经过一个传输轮次（认为发送方都成功接收接收方的确认），拥塞窗口cwnd就加倍。

伪代码表示如下：

```
if(刚刚发送的所有数据包都收到了相应的回复){
    if(cwnd<ssthresh)
        // 使用慢启动算法
        cwnd*=2
    else
        // 使用拥塞避免算法
        cwnd+=1
}
else{
    if(出现丢包、超时的情况){
        // 重回慢启动
        ssthresh = max(cwnd / 2, 2);
        cwnd = 1;
    }
    else(出现3次重复ack)
        // 使用快速重传
}
```

实际代码为：

```
//最初的慢启动
if (lastAck == (curr - 1)) {
    // 表示此时之前的都已经收到了，拥塞控制的窗口发生变化
    if (lastAck != -1) {
        // 判断此时是用慢启动还是用拥塞避免
        if (cwnd < ssthresh)
            // 此时是慢启动，直接翻倍
            cwnd *= 2;
        else
            // 此时使用拥塞避免
            cwnd += 1;
    }
}
int tempCal = 0;
for (tempCal; (tempCal < cwnd) && (curr < commu_data->totalNum) && (curr <
stop); tempCal++) {
    // 发送数据包
    sendto(sendSocket, (char*)commu_data->sendPackage[curr], 1024, 0,
(SOCKADDR*)&serverAddr, socketAddrLen);
    if (tempCal == (cwnd - 1) || curr == (commu_data->totalNum - 1))
        // 计时器开始计时
        timer = clock();
```

```
    // 输出相关信息
    cout << "当前发送数据包为：\nseq: " << curr << "\nlen高: " << int(commu_data-
>sendPackage[curr][4]) << "\tlen低: " << int(commu_data->sendPackage[curr][5])
        << "\tchecksum: " << int(commu_data->sendPackage[curr][0]) << "\n" <<
"ssthresh：" << ssthresh << "\tcwnd：" << cwnd << "\n";
    curr += 1;
}
    // 创建子线程，用于发送cwnd窗口内的包
    HANDLE send_thread = CreateThread(NULL, NULL, sendMessage, commu_data, NULL,
NULL);


//超时部分
if (((timer != -1) && (clock() - timer > MAX_WAIT_TIME))|| maxAckTimes >= 3) {
    // 超时，需要重新进入慢启动阶段
    cout << "==================超过最长等待时间，进入到慢启动阶段:\n";
    // 拥塞控制的窗口发生变化
    ssthresh = max(cwnd / 2, 16);
    cwnd = 2;
    // 重传刚刚的包
    curr = lastAck + 1;
    int tempCal = 0;
    for (tempCal; (tempCal < cwnd) && (curr < totalNum); tempCal++) {
        // 发送数据包
        sendto(sendSocket, (char*)sendPackage[curr], 1024, 0,
(SOCKADDR*)&serverAddr, socketAddrLen);
        if (tempCal == (cwnd - 1) || curr == (totalNum - 1))
            // 计时器开始计时
            timer = clock();
        // 输出相关信息
        cout << "当前发送数据包为：\nseq: " << curr << "\nlen高: " <<
int(sendPackage[curr][4]) << "\tlen低: " << int(sendPackage[curr][5])
            << "\tchecksum: " << int(sendPackage[curr][0]) << endl <<
"ssthresh：" << ssthresh << "\tcwnd：" << cwnd << endl;
        curr += 1;
    }
}
```

2. 拥塞避免

1. **乘法减小**：无论在慢启动阶段还是在拥塞控制阶段，只要网络出现超时，就是将`ssthresh`置为`cwnd`的一半，`cwnd`置为1，然后开始执行**慢启动**算法（`cwnd < ssthresh`）。
2. **加法增大**：当网络频发出现超时情况时，`ssthresh`就下降的快，为了减少注入到网络中的分组数，而加法增大是指执行拥塞避免算法后，是拥塞窗口缓慢的增大（线性增长），以防止网络过早出现拥塞。

这两个结合起来就是AIMD算法，是使用最广泛的算法。拥塞避免算法不能够完全的避免网络拥塞，通过控制拥塞窗口的大小只能使网络不易出现拥塞。

实际代码如下：

```
// 检查checksum和是不是应该收到的包
if ((newGetChecksum(recvBuf, 4) == 0)) {
    cout << "需要发送curr: " << curr << "\t已收到lastAck: " << lastAck << "\t刚收到
answerNum" << answerNum << "\tmaxAckTimes: " << maxAckTimes << endl;
    if (answerNum == (lastAck + 1)) {
        // 此时表示收到的包是正确的
```

```
            // 设置记录位
            lastAck += 1;
            sameAckTimes = 1;
            maxAckTimes = 1;

            // 判断计时器是否停止计时
            if (lastAck == (totalNum - 1))
                timer = -1;
        }
        else if (answerNum == lastAck) {
            // 记录的次数+1
            sameAckTimes += 1;
            maxAckTimes++;
            //curr = lastAck + 1;
            // 判断是否到达3次，如果到达了3次就进行快速重传
            if (sameAckTimes == 3) {
                cout << "重复ACK次数达到3次，进行重传，并进入到快恢复阶段\n";

                // 进入到快恢复
                ssthresh = max(cwnd / 2, 2);
                cwnd = ssthresh + 3;
                curr = lastAck + 1;
                sameAckTimes = 0;
                cout << "cwnd: " << cwnd << endl;
            }
            continue;
        }
        else if (answerNum > lastAck)
        {
            lastAck = answerNum;
        }
        else {
            // answerNum < lastAck的情况，一般不会出现
            return;
        }
    }
    else {
        // 要么是收到的包错了，重传一下之前的部分
        cout << "==================收到的数据包有误，将刚刚的全部重传:\n";
        int tempCal = 0;
        for (tempCal; (tempCal < cwnd) && (curr < totalNum); tempCal++) {
            // 发送数据包
            sendto(sendSocket, (char*)sendPackage[curr], 1024, 0,
(SOCKADDR*)&serverAddr, socketAddrLen);
            if (tempCal == (cwnd - 1) || curr == (totalNum - 1))
                // 计时器开始计时
                timer = clock();
            // 输出相关信息
            cout << "当前发送数据包为：\nseq: " << curr << "\nlen高: " <<
int(sendPackage[curr][4]) << "\tlen低: " << int(sendPackage[curr][5])
                << "\tchecksum: " << int(sendPackage[curr][0]) << endl <<
"ssthresh：" << ssthresh << "\tcwnd：" << cwnd << endl;
            curr += 1;
        }
        //重新开始计时
        timer = clock();
    }
```

3. 快速重传

要求首先接收方收到一个**失序的报文段**后就立刻发出重复确认，而不要等待自己发送数据时才进行捎带确认。

接收方成功的接受了发送方发送来的M1、M2并且分别给发送了ACK，现在接收方没有收到M3，而接收到了M4，显然接收方不能确认M4，因为M4是失序的报文段。如果根据可靠性传输原理接收方什么都不做，但是按照快速重传算法，在收到M4、M5等报文段的时候，不断**重复的向发送方发送M2的ACK**，如果接收方一连收到三个重复的ACK，那么发送方不必等待重传计时器到期，由发送方尽早重传未被确认的报文段。

快恢复：

1. 当发送发连续接收到三个相同的确认时，就执行乘法减小算法，把慢启动开始门限（`ssthresh`）减半，但是接下来并不执行慢开始算法。
2. 此时不执行慢启动算法，而是把`cwnd`设置为`ssthresh`的一半，然后执行**拥塞避免**算法，使拥塞窗口缓慢增大。

```
if (sameAckTimes == 3) {
    cout << "重复ACK次数达到3次，进行重传，并进入到快恢复阶段\n";

    // 重传server需要的包（也就是丢了的包）
    sendto(sendSocket, (char*)sendPackage[lastAck + 1], 1024, 0,
(SOCKADDR*)&serverAddr, socketAddrLen);
    cout << "当前发送数据包为：\nseq: " << curr << "\nlen高: " <<
int(sendPackage[curr][4]) << "\tlen低: " << int(sendPackage[curr][5])
         << "\tchecksum: " << int(sendPackage[curr][0]) << endl << "ssthresh：" <<
ssthresh << "\tcwnd：" << cwnd << endl;

    // 进入到快恢复
    ssthresh = max(cwnd / 2, 2);
    cwnd = ssthresh + 3;
    curr = lastAck + 2;//lastAck就是前面几行所发的包
}
```

其他相关函数

其他功能函数如下：

4. 计算校验和

将除去第一位的剩下每一位都进行求和，如果求和结果超过8位，则将进位轮到末尾相加。最后将求和的结果取反得到校验和。相关代码如下：

```
/// <summary>
/// 计算校验和，从第1位开始算，一直算到最后一位
/// </summary>
/// <param name="pac">计算的包</param>
/// <param name="len">包的长度</param>
/// <returns>返回校验和</returns>
unsigned char newGetChecksum(char* package, int len) {
    if (len == 0) {
        return ~(0);
    }
    unsigned int sum = 0;
    int i = 0;
```

```
    while (len--) {
        sum += (unsigned char)package[i++];
        if (sum & 0xFF00) {
            sum &= 0x00FF;
            sum++;
        }
    }
    return ~(sum & 0x00FF);
}
```

5. 获取文件列表

```
oid getFileList() {
    string path = "./test/*";

    // 建立句柄
    long handle;
    struct _finddata_t fileinfo;

    // 第一个文件
    handle = _findfirst(path.c_str(), &fileinfo);
    // 判断句柄状态
    if (handle == -1) {
        cout << "文件名称出现错误\n";
        return;
    }
    else {
        cout << "测试文件夹下目录列表为：\n";
        int tempNumOfFile = 0;
        do
        {
            //找到的文件的文件名
            if (fileinfo.name[0] == '.')
                continue;
            cout << ++tempNumOfFile << ": " << fileinfo.name << endl;

        } while (!_findnext(handle, &fileinfo));
        _findclose(handle);
        return;
    }
}
```

6. 获取IP

```
void getIP(char* p) {
    cout << "Please use 'ipconfig' to get your ip and input\n";
    char ip[16];
    cin.getline(ip, sizeof(ip));
    int index = 0;
    while (ip[index] != '\0') {
        p[index] = ip[index];
        index++;
    }
}
```

Client(发送端)相关函数

7. 三次握手 - 发送端

```
// 三次握手，建立连接
void connet2Server() {
    // 创建一个要发送的数据包，并初始化
    // 数据包：check，SYN，FIN，SHAKE/WAVE
    char shakeChar[4];
    shakeChar[1] = 0x01;
    shakeChar[2] = 0x00;
    shakeChar[3] = SHAKE1;
    shakeChar[0] = newGetChecksum(shakeChar+1, 3);

    // 发送shake1
    sendto(sendSocket, shakeChar, 4, 0, (SOCKADDR*)&serverAddr, socketAddrLen);
    cout<< "开始建立连接，请求建立连接的数据包为（第一次握手）：\n";
    cout << "SYN: " << int(shakeChar[1]) << "\tFIN: " << int(shakeChar[2]) <<
"\tSHAKE: " << int(shakeChar[3]) << "\tcheckSum: " << int(shakeChar[0]) << endl;

    // 开始计时
    int beginTime = clock();

    // 接收缓冲区
    char recvShakeBuf[4];
    memset(recvShakeBuf, 0, 4);

    //超时重传
    while((recvfrom(sendSocket, recvShakeBuf, 4, 0, (SOCKADDR*)&serverAddr,
&socketAddrLen))==SOCKET_ERROR){
        if (clock() - beginTime > MAX_WAIT_TIME) {
            cout << "超过最长等待时间，重传:";
            sendto(sendSocket, shakeChar, 4, 0, (SOCKADDR*)&serverAddr,
socketAddrLen);
            beginTime = clock();
            cout << "SYN: " << int(shakeChar[1]) << "\tFIN: " <<
int(shakeChar[2]) << "\tSHAKE: " << int(shakeChar[3]) << "\tcheckSum: " <<
int(shakeChar[0]) << endl;
        }
    }
    // 收到数据包
    cout << "收到数据包为：\n";
    cout << "SYN: " << int(recvShakeBuf[1]) << "\tFIN: " << int(recvShakeBuf[2])
<< "\tSHAKE: " << int(recvShakeBuf[3]) << "\tcheckSum: " << int(recvShakeBuf[0])
<< endl;

    // 收到shake2正确，则发送shake3
    if (newGetChecksum(recvShakeBuf, 4)==0 && recvShakeBuf[1] == 0x01 &&
recvShakeBuf[2] == 0x00 && recvShakeBuf[3] == SHAKE2) {
        memset(shakeChar, 0, 4);
        shakeChar[1] = 0x01;
        shakeChar[2] = 0x00;
        shakeChar[3] = SHAKE3;
        shakeChar[0] = newGetChecksum(shakeChar+1, 3);
        sendto(sendSocket, shakeChar, 4, 0, (SOCKADDR*)&serverAddr,
socketAddrLen);
        cout << "发送第三次握手数据包：\n";
```

```
        cout << "SYN: " << int(shakeChar[1]) << "\tFIN: " << int(shakeChar[2])
<< "\tSHAKE: " << int(shakeChar[3]) << "\tcheckSum: " << int(shakeChar[0]) <<
endl;
        cout << "\n连接建立成功\n\n";
    }
}
```

8. 两次挥手 - 发送端

```
static void sayGoodBye2Server() {
    // 两次挥手，断开连接

    // 数据包：check，SYN，FIN，SHAKE/WAVE
    char shakeChar[4];
    shakeChar[1] = 0x00;
    shakeChar[2] = 0x01;
    shakeChar[3] = WAVE1;
    shakeChar[0] = newGetChecksum(shakeChar+1, 4-1);
    // 发送WAVE1
    sendto(sendSocket, shakeChar, 4, 0, (SOCKADDR*)&serverAddr, socketAddrLen);
    cout << "断开连接，发送数据包为（第一次挥手）：\n";
    cout << "SYN: " << int(shakeChar[1]) << "\tFIN: " << int(shakeChar[2]) <<
"\tSHAKE: " << int(shakeChar[3]) << "\tcheckSum: " << int(shakeChar[0]) << endl;

    // 开始计时
    int beginTime = clock();

    // 停等，等服务器的挥手
    char recvWaveBuf[4];
    while (true) {
        while ((recvfrom(sendSocket, recvWaveBuf, 4, 0, (SOCKADDR*)&serverAddr,
&socketAddrLen))==SOCKET_ERROR) {
            if (clock() - beginTime > MAX_WAIT_TIME) {
                cout << "超过最长等待时间，重传:";
                sendto(sendSocket, shakeChar, 4, 0, (SOCKADDR*)&serverAddr,
socketAddrLen);
                beginTime = clock();
                cout << "SYN: " << int(shakeChar[1]) << "\tFIN: " <<
int(shakeChar[2]) << "\tSHAKE: " << int(shakeChar[3]) << "\tcheckSum: " <<
int(shakeChar[0]) << endl;
            }
        }
        if (newGetChecksum(recvWaveBuf, 4)==0 && recvWaveBuf[1] == 0x00 &&
recvWaveBuf[2] == 0x01 && recvWaveBuf[3] == WAVE2) {
            // 收到握手包正确
            cout << "收到来自服务器的第二次挥手：\n";
            cout << "SYN: " << int(recvWaveBuf[1]) << "\tFIN: " <<
int(recvWaveBuf[2]) << "\tSHAKE: " << int(recvWaveBuf[3]) << "\tcheckSum: " <<
int(recvWaveBuf[0]) << endl;
            memset(recvWaveBuf, 0, 4);
            memset(shakeChar, 0, 4);
            break;
        }
        else {
            cout << "服务器挥手失败，重传挥手数据包\n";
            sendto(sendSocket, shakeChar, 4, 0, (SOCKADDR*)&serverAddr,
socketAddrLen);
```

```
            cout << "断开连接，发送数据包为（第一次挥手）：\n";
            cout << "SYN: " << int(shakeChar[1]) << "\tFIN: " <<
int(shakeChar[2]) << "\tSHAKE: " << int(shakeChar[3]) << "\tcheckSum: " <<
int(shakeChar[0]) << endl;
        }
    }
}
```

Server(接收端)相关函数

9. 三次握手 - 接收端

```
static bool shakeHand() {
    // 握手
    //Package p;
    //Package sendP;
    char recvShakeBuf[4];
    while (1) {
        memset(recvShakeBuf, 0, 4);
        // 停等机制接收数据包
        while (recvfrom(serverSocket, recvShakeBuf, 4, 0,
(sockaddr*)&clientAddr, &sockaddrLen) == SOCKET_ERROR) {

        }

        // 检查接收到的数据包校验和是否正确
        // 先对收到的数据进行转换
        //p = *((Package*)recvBuf);
        // 输出信息
        cout << "收到建立连接的请求，数据包为：\n";
        // packageOutput(p);
        cout << "SYN: " << int(recvShakeBuf[1]) << "\tFIN: " <<
int(recvShakeBuf[2]) << "\tSHAKE: " << int(recvShakeBuf[3]) << "\tcheckSum: " <<
int(recvShakeBuf[0]) << endl;
        // 第一次
        if (newGetChecksum(recvShakeBuf,4)==0 && recvShakeBuf[1] == 0x01 &&
recvShakeBuf[2] == 0x00 && recvShakeBuf[3] == SHAKE1) {
            // 收到握手包正确
            cout << "握手包正确\n";
            // 创建第二次握手包
            char secondShake[4];
            secondShake[1] = 0x01;
            secondShake[2] = 0x00;
            secondShake[3] = SHAKE2;
            secondShake[0] = newGetChecksum(secondShake+1, 4-1);
            // 发送第二次握手包
            sendto(serverSocket, secondShake, 4, 0, (SOCKADDR*)&clientAddr,
sockaddrLen);
            // 输出内容
            cout << "SYN: " << int(secondShake[1]) << "\tFIN: " <<
int(secondShake[2]) << "\tSHAKE: " << int(secondShake[3]) << "\tcheckSum: " <<
int(secondShake[0]) << endl;

            int beginTime = clock();

            // 清空刚刚接收的包，等待第三次握手
            memset(recvShakeBuf, 0, 4);
```

```
            while (recvfrom(serverSocket, recvShakeBuf, 4, 0,
(sockaddr*)&clientAddr, &sockaddrLen) == SOCKET_ERROR) {
                if (clock() - beginTime > MAX_WAIT_TIME) {
                    cout << "超过最长等待时间，重传:";
                    sendto(serverSocket, secondShake, 4, 0,
(SOCKADDR*)&clientAddr, sockaddrLen);
                    beginTime = clock();
                    cout << "SYN: " << int(secondShake[1]) << "\tFIN: " <<
int(secondShake[2]) << "\tSHAKE: " << int(secondShake[3]) << "\tcheckSum: " <<
int(secondShake[0]) << endl;
                }
            }


            cout << "收到数据包为：\n";
            cout << "SYN: " << int(recvShakeBuf[1]) << "\tFIN: " <<
int(recvShakeBuf[2]) << "\tSHAKE: " << int(recvShakeBuf[3]) << "\tcheckSum: " <<
int(recvShakeBuf[0]) << endl;

            if (newGetChecksum(recvShakeBuf, 4)==0 && recvShakeBuf[1] == 0x01 &&
recvShakeBuf[2] == 0x00 && recvShakeBuf[3] == SHAKE3) {
                // 收到握手包正确
                cout << "\n连接建立成功\n\n";
                return true;
            }
        }
        else {
            cout << "收到的数据包有误，重新等待连接\n";
            continue;
        }
    }
}
```

10. 两次挥手 - 接收端

```
static void waveHand() {
    // 挥手，挥两次
    char recvWaveBuf[4];
    while (true) {
        while (recvfrom(serverSocket, recvWaveBuf, 4, 0, (sockaddr*)&clientAddr,
&sockaddrLen) == SOCKET_ERROR) {

        }

        cout << "收到断开连接的请求，数据包为：\n";
        cout << "SYN: " << int(recvWaveBuf[1]) << "\tFIN: " <<
int(recvWaveBuf[2]) << "\tSHAKE: " << int(recvWaveBuf[3]) << "\tcheckSum: " <<
int(recvWaveBuf[0]) << endl;

        // 第一次
        if (newGetChecksum(recvWaveBuf, 4)==0 && recvWaveBuf[1] == 0x00 &&
recvWaveBuf[2] == 0x01 && recvWaveBuf[3] == WAVE1) {
            // 收到挥手包正确
            cout << "挥手包正确\n";
            // 创建第二次挥手包
            char secondShake[4];
```

```
            secondShake[1] = 0x00;
            secondShake[2] = 0x01;
            secondShake[3] = WAVE2;
            secondShake[0] = newGetChecksum(secondShake+1, 4-1);
            // 发送第二次挥手包
            sendto(serverSocket, secondShake, 4, 0, (SOCKADDR*)&clientAddr,
sockaddrLen);
            // 输出内容
            cout << "发送给客户端的第二次挥手：\n";
            cout << "SYN: " << int(secondShake[1]) << "\tFIN: " <<
int(secondShake[2]) << "\tSHAKE: " << int(secondShake[3]) << "\tcheckSum: " <<
int(secondShake[0]) << endl;

            memset(secondShake, 0, 4);
            memset(recvWaveBuf, 0, 4);
            break;
        }
    }
}
```

吞吐率计算

使用C++中的clock函数，从开始传输文件时计时，-件传输完成结束。

吞吐率计算公式为：传输的文件大小/总时长；其中文件大小按字节计算，总时长按秒计算

使用方法

Client - 发送端

1. 运行程序
2. 输入要发送的IP地址
3. 输入本地的IP地址
4. 选择文件发送（需要输入文件的全名）
5. 文件发送完成自动结束程序

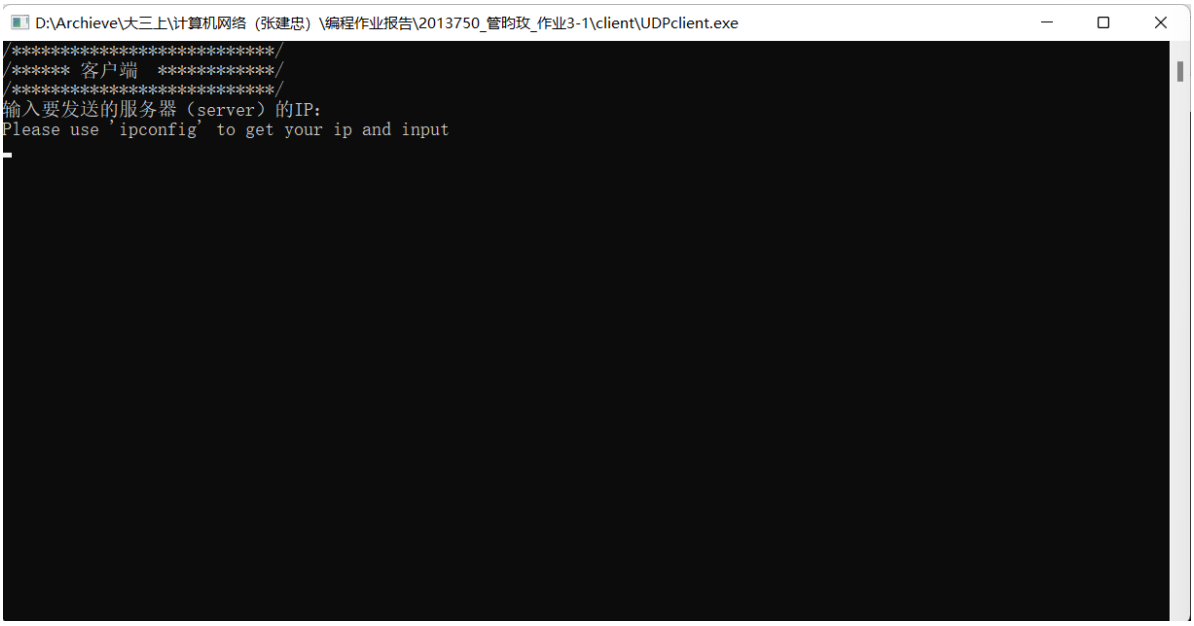
Server - 接收端

1. 运行程序
2. 输入要接收的发送端的IP
3. 等待连接
4. 连接成功后等待发送端发送文件
5. 文件接收完成后自动结束程序

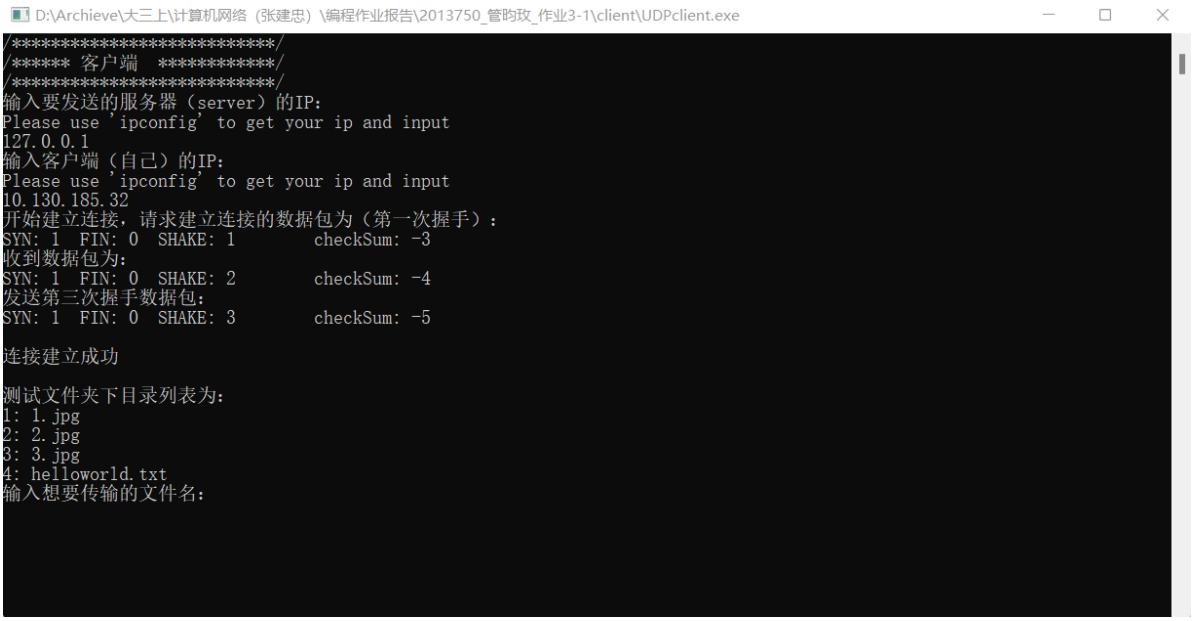
实验成果展示

Client发送端

等待连接页面



输入服务器IP和客户端IP，三次握手成功，自动弹出文件列表并等待输入



输入要发送的文件名，进行发送

发送过程中

```
剩余窗口大小：8
当前发送数据包为：
seq: 650
len高: 3        len低: -7       checksum: 109
剩余窗口大小：7
当前发送数据包为：
seq: 651
len高: 3        len低: -7       checksum: 102
剩余窗口大小：6
当前发送数据包为：
seq: 652
len高: 3        len低: -7       checksum: -67
剩余窗口大小：5
当前发送数据包为：
seq: 653
len高: 3        len低: -7       checksum: 97
剩余窗口大小：4
当前发送数据包为：
seq: 654
len高: 3        len低: -7       checksum: 76
剩余窗口大小：3
当前发送数据包为：
seq: 655
len高: 3        len低: -7       checksum: 62
剩余窗口大小：2
当前发送数据包为：
seq: 656
len高: 3        len低: -7       checksum: 127
剩余窗口大小：1
```

重传

```
剩余需要重传的包个数为：6

当前发送数据包为：
seq: 1047
len高: 3        len低: -7       checksum: 12

剩余需要重传的包个数为：5

当前发送数据包为：
seq: 1048
len高: 3        len低: -7       checksum: 47

剩余需要重传的包个数为：4

当前发送数据包为：
seq: 1049
len高: 3        len低: -7       checksum: -32

剩余需要重传的包个数为：3

当前发送数据包为：
seq: 1050
len高: 3        len低: -7       checksum: -105

剩余需要重传的包个数为：2

当前发送数据包为：
seq: 1051
len高: 3        len低: -7       checksum: -106
```

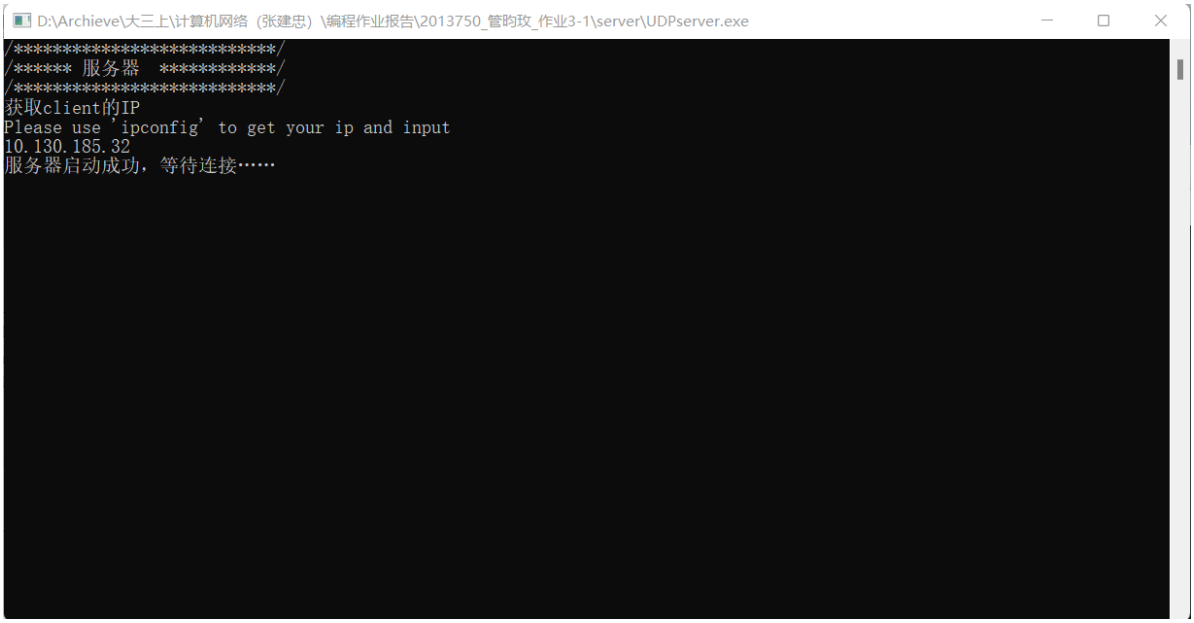
完成发送后，告知用户文件传输完成，并输出传输时间和吞吐率

最后进行挥手，结束连接

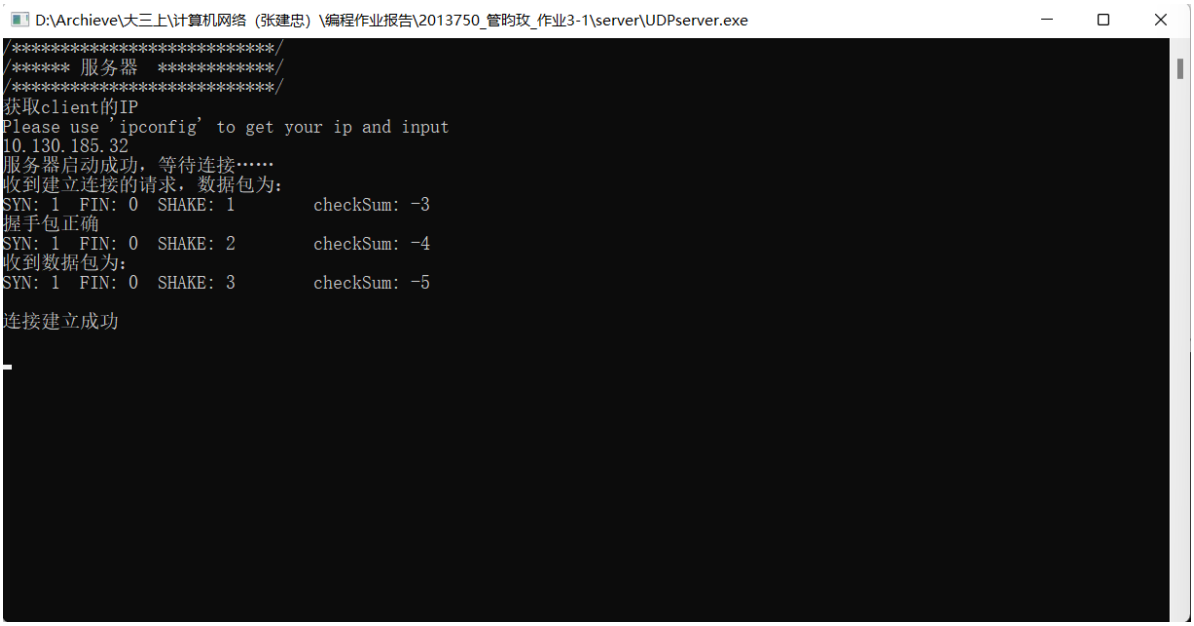
```
本次传输所用时间为：206.18 s.
断开连接，发送数据包为（第一次挥手）：
SYN: 0  FIN: 1  SHAKE: 4        checkSum: -6
收到来自服务器的第二次挥手：
SYN: 0  FIN: 1  SHAKE: 5        checkSum: -7
本次传输吞吐率为：8030.89 Kb/s
请按任意键继续. . .
```

Server接收端

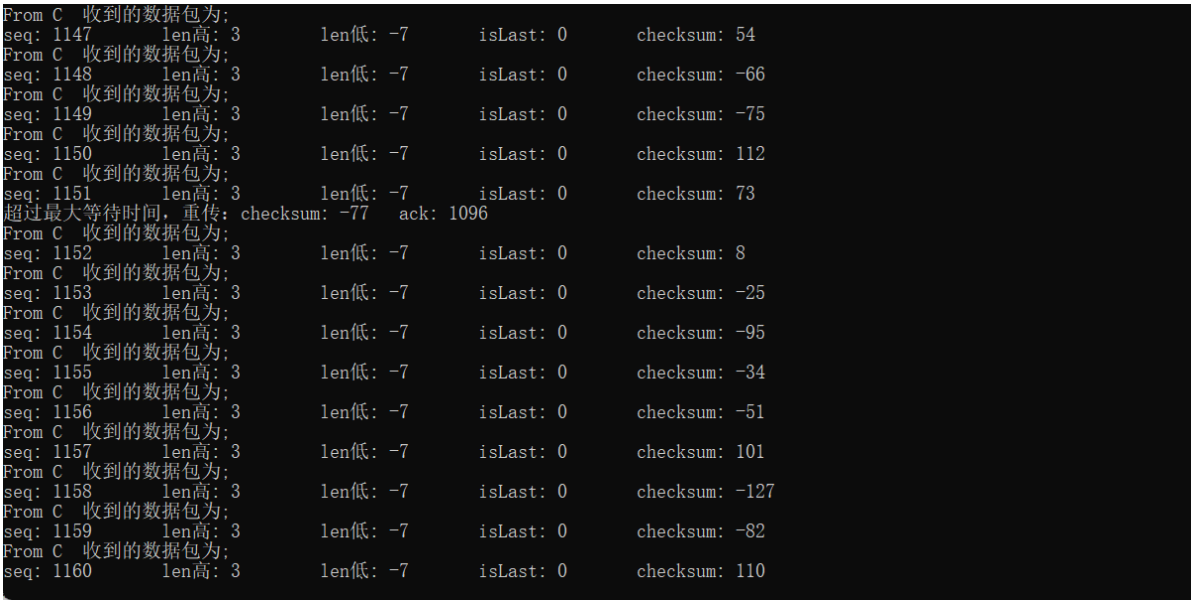
输入接收端IP并等待连接(本该输入本机IP，但此处为了方便直接固定为回环地址127.0.0.1)



三次握手成功页面



接收文件页面



正确接受并交付上层

```
checksum: -71   ack: 1624
From C  收到的数据包为:
seq: 1625       len高: 3        len低: -7       isLast: 0       checksum: -98
delivered:      1625
TO C    正确返回数据包
checksum: -71   ack: 1625
From C  收到的数据包为:
seq: 1626       len高: 3        len低: -7       isLast: 0       checksum: 85
delivered:      1626
TO C    正确返回数据包
checksum: -71   ack: 1626
From C  收到的数据包为:
seq: 1627       len高: 3        len低: -7       isLast: 0       checksum: 84
delivered:      1627
TO C    正确返回数据包
checksum: -71   ack: 1627
From C  收到的数据包为:
seq: 1628       len高: 3        len低: -7       isLast: 0       checksum: -36
delivered:      1628
TO C    正确返回数据包
checksum: -71   ack: 1628
From C  收到的数据包为:
seq: 1629       len高: 0        len低: -124     isLast: 1       checksum: 91
delivered:      1629
TO C    正确返回数据包
checksum: -100  ack: 1629
文件保存成功
数据传输完成，等待客户端退出……
收到断开连接的请求，数据包为:
SYN: 0  FIN: 1  SHAKE: 4        checkSum: -6
```

文件传输完成，并自动进行挥手，结束连接

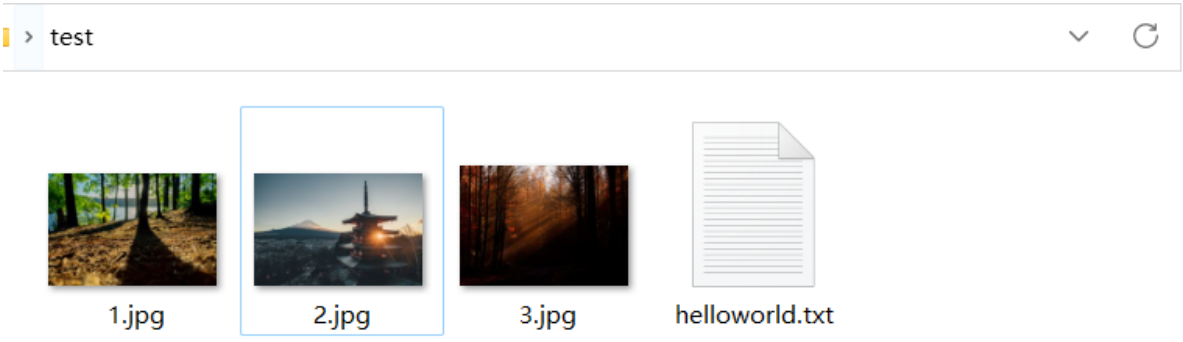
```
TO C    正确返回数据包
checksum: -1    ack: 0
From C  收到的数据包为:
seq: 1  len高: 3        len低: -5       isLast: 0       checksum: 116
temp长度：1019

TO C    正确返回数据包
checksum: -2    ack: 1
From C  收到的数据包为:
seq: 0  len高: 3        len低: -5       isLast: 0       checksum: -102
temp长度：1019

TO C    正确返回数据包
checksum: -1    ack: 0
From C  收到的数据包为:
seq: 1  len高: 3        len低: -72      isLast: 1       checksum: 113
temp长度：952

TO C    正确返回数据包
checksum: -2    ack: 1
文件保存成功
数据传输完成，等待客户端退出……
收到断开连接的请求，数据包为:
SYN: 0  FIN: 1  SHAKE: 4        checkSum: -6
挥手包正确
发送给客户端的第二次挥手:
SYN: 0  FIN: 1  SHAKE: 5        checkSum: -7
挥手结束，BYE
服务正常结束
请按任意键继续. . .
```

接收文件夹



路由器配置

注:

client IP: 127.0.0.1 Port: 8889

Router IP: 127.0.0.1 Port: 12770

Server IP:127.0.0.1 Port: 12660