南開大學
Nankai University

计算机网络实验三

基于**UDP**服务设计可靠传输协议并编程实现

实验3-1：停等机制

姓名：刘伟

学号：**2013029**

专业：物联网工程

一、实验要求

利用数据报套接字在用户空间实现面向连接的可靠数据传输，功能包括：`建立连接、差错检测、确认重传`等。流量控制采用停等机制，完成给定测试文件的传输。

(1) 实现单向传输。
(2) 给出详细的协议设计。
(3) 给出实现的拥塞控制算法的原理说明。
(4) 完成给定测试文件的传输，显示传输时间和平均吞吐率。
(5) 性能测试指标：吞吐率、时延，给出图形结果并进行分析。
(6) 完成详细的实验报告（每个任务完成一份）。
(7) 编写的程序应结构清晰，具有较好的可读性。
(8) 提交程序源码和实验报告。

二、实验协议设计&功能实现

1. 报文格式：

作为客户端与服务端交互信息最重要的组成部分，报文格式的设计对实现可靠传输有着举足轻重的作用。

```
typedef struct Packet_Header
{
    WORD datasize;      // 数据长度
    BYTE tag;           // 标签  八位，使用后五位，排列是END OVER FIN ACK SYN
    BYTE window;        // 窗口大小
    BYTE seq;           // 序列号
    BYTE ack;           // 确认号
    WORD checksum;      // 校验和
};
```

0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
datasize数据长度															
			END	OVER	FIN	ACK	SYN	window							
seq序列号								ack确认号							
checksum校验和															

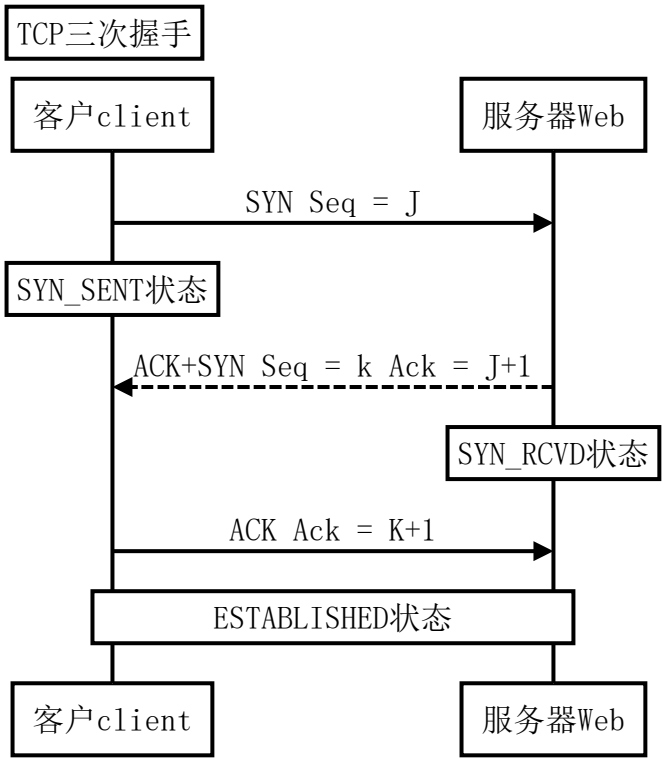
在本次实验的报文格式设计中，报文头总长64位。

- 0-15位，即前16位标记为数据长度
- 16-23位，标记为tag，即各个标签位。实验使用八位中的后五位：END、OVER、FIN、ACK、SYN
- 24-31位，标记为window窗口，该部分在本次实验部分未被使用，是为了后续的实验设计做准备
- 32-39位，标记为seq序列号，序列号在0-255之间循环使用
- 40-47位，标记为ack确认号，确认号与序列号之间有一定的对应关系去确保接发数据的正确性
- 48-63位，标记为检验和，该部分用来检验数据在传递过程中有无出错，检查正确性

2. 建立连接&断开连接方式：仿照TCP协议

> 建立连接：三次握手

TCP协议中三次握手过程如下：



> 1. 第一次握手：客户端发送 `syn` 包。Client将标志位SYN置为1，随机产生一个值seq=J，并将该数据包发送给Server，Client进入SYN_SENT状态，等待Server确认。
> 2. 第二次握手：服务器返回客户端 `SYN+ACK段` 。Server收到Client发来的 `SYN` 数据包后，由标志位SYN=1知道Client请求建立连接。Server将标志位SYN和ACK都置为1，ack=J+1，随机产生一个值seq=K，并将该数据包发送给Client以确认连接请求，Server进入SYN_RCVD状态。
> 3. 第三次握手：客户端收到服务器发送的 `SYN+ACK` 段，给服务器响应 `ACK` 段。Client收到确认后，检查ack是否为J+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=K+1，并将该数据包发送给Server，Server检查正确则连接建立成功，Client和Server进入ESTABLISHED状态。

在本实验中，借助TCP三次握手的核心流程，客户端与服务端的三次握手过程总结如下：

第一次：客户端向服务端发送请求建立连接的SYN数据包，其中tag = SYN，等待服务端回传响应

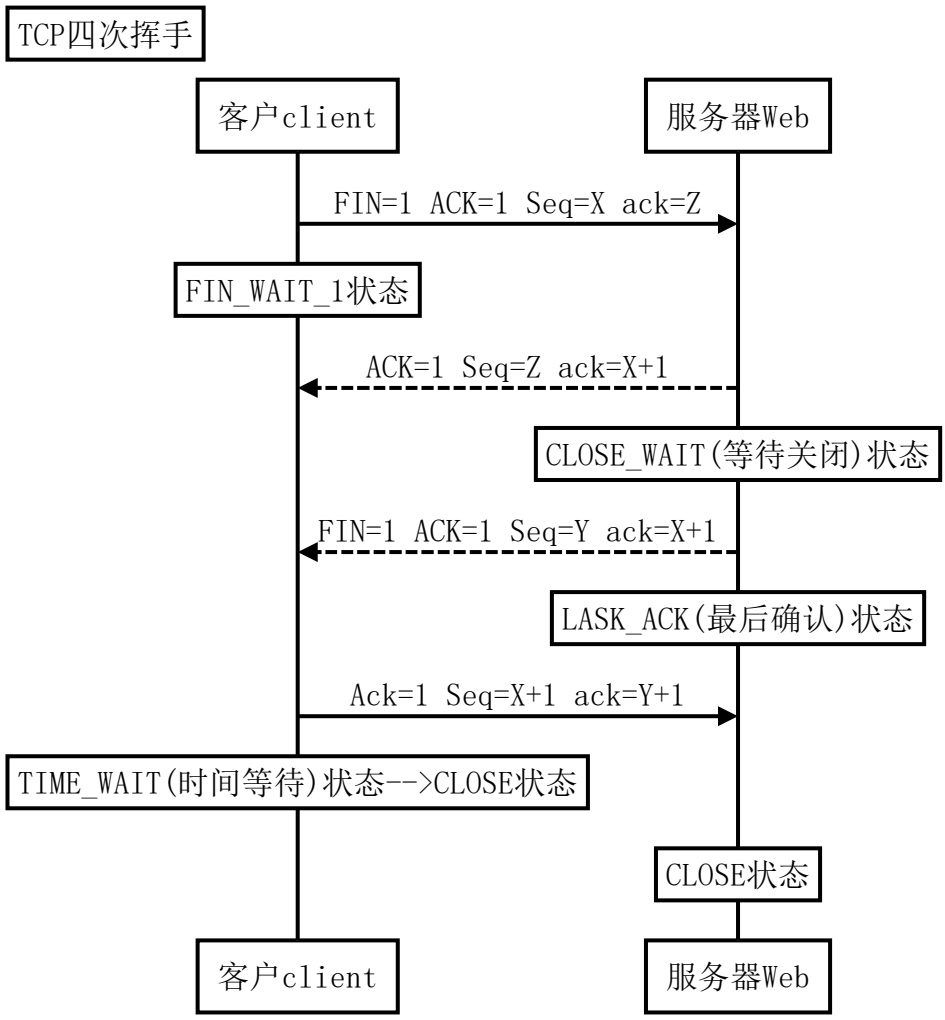
第二次：服务端在收到客户端发来的第一次SYN请求后，对其内的消息进行校验判断合格后，回传ACK响应(tag = ACK)

第三次：客户端收到服务端发来的ACK响应后，校验判断合格后，向服务端发ACK_SYN数据包(tag = ACK_SYN)

> 期间在对消息的发送过程中有设计相应的超时重传机制！

> 断开连接：四次挥手

TCP协议中四次挥手过程如下：



> 1. 第一次挥手：Client发送一个FIN，用来关闭Client到Server的数据传送，Client进入FIN_WAIT_1状态。
> 2. 第二次挥手：Server收到FIN后，发送一个ACK给Client，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），Server进入CLOSE_WAIT状态。
> 3. 第三次挥手：Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST_ACK状态。
> 4. 第四次挥手：Client收到FIN后，Client进入TIME_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手。

在本实验中，借助TCP四次挥手的核心流程，客户端与服务端的四次挥手过程总结如下：

第一次：客户端向服务端发送断开连接的数据包，其中tag = FIN，并等待服务端的后俩次回传响应

第二次：服务端在接收到客户端发来的FIN数据包后，便得知客户端请求断联，服务端在第二次的过程中给客户端发送ACK数据包(tag = ACK)，告知客户端其已接受到它的断联请求

第三次：服务端发送FIN_ACK(tag = FIN_ACK)，告知客户端其也要断开连接，服务端等待客户端确认

第四次：客户端最后时间收到服务端发来的断联请求，做出响应应答，回传ACK数据包(tag = ACK)，告知服务端其已接受到它的断联请求

3. 流量控制：

在实验3-1部分，流量控制采用的是 停等机制 。

停等机制：发送方发送一帧，就得等待应答信号回应后，继续发出下一帧，接收站在接收到一帧后，发送回一个应答信号给接收方，发送方如果没有收到应答信号则必须等待，超出一定时间后启动重传机制。

在停等机制下，发送方每次发送的数据包必须在收到接收方的应答响应之后，才能进行下一次的发包。若长时间未收到应答响应，那么就会启动 超时重传机制 。

```
Begin sending message..... datasize:1024 bytes! flag:0 seq:2 checksum:63997
Successful send and recv response  ack:3
Begin sending message..... datasize:1024 bytes! flag:0 seq:3 checksum:63740
Successful send and recv response  ack:4
Begin sending message..... datasize:1024 bytes! flag:0 seq:4 checksum:63483
Successful send and recv response  ack:5
Begin sending message..... datasize:1024 bytes! flag:0 seq:5 checksum:63226
Successful send and recv response  ack:6
Begin sending message..... datasize:1024 bytes! flag:0 seq:6 checksum:62969
Successful send and recv response  ack:7
Begin sending message..... datasize:1024 bytes! flag:0 seq:7 checksum:62712
Successful send and recv response  ack:8
Begin sending message..... datasize:1024 bytes! flag:0 seq:8 checksum:62455
Successful send and recv response  ack:9
Begin sending message..... datasize:1024 bytes! flag:0 seq:9 checksum:62198
Successful send and recv response  ack:10
Begin sending message..... datasize:1024 bytes! flag:0 seq:10 checksum:61941
Successful send and recv response  ack:11
Begin sending message..... datasize:1024 bytes! flag:0 seq:11 checksum:61684
Successful send and recv response  ack:12
Begin sending message..... datasize:1024 bytes! flag:0 seq:12 checksum:61427
Successful send and recv response  ack:13
Begin sending message..... datasize:1024 bytes! flag:0 seq:13 checksum:61170
Successful send and recv response  ack:14
```

4. 差错检验：

在该实验中，实验设计是基于UDP服务设计展开的可靠传输协议设计。差错检验对于客户端与服务端之间建立可靠通信提供了前提保证。利用一定的校验和生成方式（UDP检验和生成方法），对发送信息帧的头部进行计算并存入进16位的checksum字段内。接收方接收到数据后，需要对数据的校验和进行验证，进行差错检验。

> 1)：累加，对每个16位的字进行累加
>
> 2)：回卷，将可能多出来的第17位消除（与16位的sum进行相加
>
> 3)：取反，对和进行反码运算

5. 确认重传：

实验设计中，接收方对发送方发送来的数据不仅需要利用检验和进行确认，而且还要对其 `序列号` 加以确认。

接收方收到的数据需要确保是正确的，且是其所需的而不是冗余的。实验设计上接收方对发送方发来的数据序列号加以确认，如果不是正确所需的，那么便会回传给发送方一个告知：告知其需要重发正确的数据，接受发也会抛弃掉当前接收的错误数据包。

```
// 对接收数据加以确认，否则发送标记告知客户端需要重发
if (packet.seq != seq) {  // 对其序列号加以确认
    Packet_Header temp;
    // 填充相关数据……
    memcpy(buffer, &temp, sizeof(temp));
    sendto(socketServer, buffer, sizeof(temp), 0, (sockaddr*)&clieAddr, clieAddrlen);
    cout << "Send a resend flag to the client" << endl;
    continue;
}
```

```
// 服务端未接受到数据 重传数据
// 服务端接收到的数据 校验和出错 需要重传
if (packet.ack == Seq_num || (compute_sum((WORD*)&packet, sizeof(packet)) != 0)) {
    cout << "服务端未接受到数据，正在重传！！！" << endl;
    i--;
    continue;
}
```

6. 超时重传：

可靠传输设计需要满足信息通信双方对数据的交互的严苛把控。在实际实验设计上，采用超时重传机制以避免在网络不流畅情况下，保证信息交互的可靠性。

客户端/服务器端每发送一个报文（部分报文发送无超时重传机制。eg：三次握手的第三次，四次挥手的第四次）时，启动一个计时器，利用计时器进行时间记录，当超时（长时间没有接受到回传响应）时，重发该数据报。

```
clock_t start = clock();
while (recvfrom(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
&servAddrlen) <= 0) {
    if ((clock() - start) / 1000 > 1) {  //时间超过1秒 标记为超时重传
        sendto(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
servAddrlen);
        cout << "数据发送超时，正在重传！！！" << endl;
        start = clock();
    }
}
```

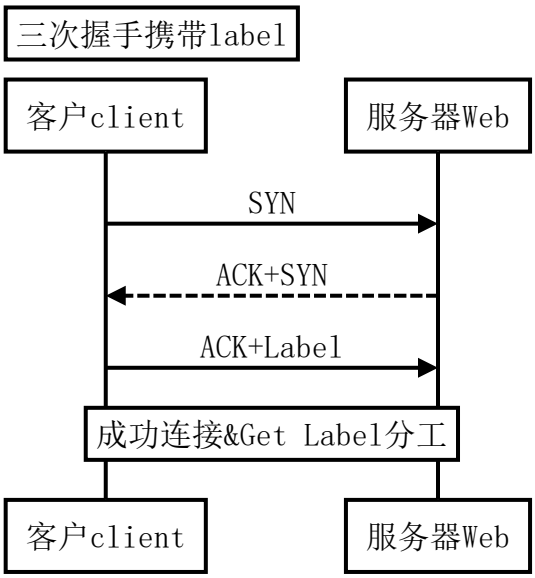
7. 双向传输：

实验创新设计：客户端既可以作为发送方，也可以作为接受方接受服务端发送的数据文件！

实验设计的时候遇到的问题是：该怎么让客户端与服务端确定谁作为发送方！

解决方法：利用三次握手的第三次握手让客户端携带label信息，告知服务端：客户端是否作为发送方！如果客户端想要发送数据，那么自然的将服务端作为接受方。如果客户端不想要发送数据，而是想要接收数据，那么服务端收到并解析出label后，就会作为发送方给客户端发送数据文件。

```
// 输入选项
cout << "请输入选项告知—（发送0 / 接收1）数据：";
int label;
cin >> label;
cout << "开始向服务端建立连接请求....." << endl;
Client_Server_Connect(client, serveraddr, length, label);
// 第三次：客户端发送第三次握手数据
packet.tag = ACK_SYN;
// 携带label标签 告知服务端：客户端扮演的角色
packet.datasize = label;
packet.checksum = 0;
packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
memcpy(buffer, &packet, sizeof(packet));
if (sendto(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr, servAddrlen)
== -1) {
    cout << "sendto函数发送数据出错！" << endl;
    return 0;
}
```



三、核心函数&代码实现

1. 创建套接字——客户端&服务端

```
WORD wVersionRequested = MAKEWORD(2, 2);
WSADATA wsaData;
int err = WSAStartup(wVersionRequested, &wsaData);
if (err != 0) {
    cout << "Startup 出错! " << endl;
    WSACleanup();
    return 0;
}
SOCKET client = socket(AF_INET, SOCK_DGRAM, 0);
if (client == -1) {
    cout << "socket 生成出错! " << endl;
    WSACleanup();
    return 0;
}
cout << "客户端套接字创建成功! " << endl;
```

套接字socket对象创建好后，需要对其进行相关信息的填充：IP、PORT端口好之类的。

2. 数据帧格式结构体

```
typedef struct Packet_Header {
    WORD datasize;      // 数据长度
    BYTE tag;           // 标签  八位，使用后四位，排列是OVER FIN ACK SYN
    BYTE window;        // 窗口大小
    BYTE seq;           // 序列号
    BYTE ack;           // 确认号
    WORD checksum;      // 校验和
    // 初始化
    Packet_Header() {
        datasize = 0;
        tag = 0;
        window = 0;
        seq = 0;
        ack = 0;
        checksum = 0;}
};

const BYTE SYN = 0x1;       //SYN = 1 ACK = 0 FIN = 0
const BYTE ACK = 0x2;       //SYN = 0 ACK = 1 FIN = 0
const BYTE ACK_SYN = 0x3;   //SYN = 1 ACK = 1 FIN = 0
const BYTE FIN = 0x4;       //FIN = 1 ACK = 0 SYN = 0
const BYTE FIN_ACK = 0x6;   //FIN = 1 ACK = 1 SYN = 0
const BYTE OVER = 0x8;      //结束标志
const BYTE END = 0x16;      //全局结束标志
```

3. 计算校验和

```
// 计算校验和
WORD compute_sum(WORD* message,int size) {    // size = 8
    int count = (size + 1) / 2;  // 防止奇数字节数 确保WORD 16位
    WORD* buf = (WORD*)malloc(size + 1);
    memset(buf, 0, size + 1);
    memcpy(buf, message, size);
    u_long sum = 0;
    while (count--) {
        sum += *buf++;
        if (sum & 0xffff0000) {
            sum &= 0xffff;
            sum++;
        }
    }
    return ~(sum & 0xffff);
}
```

4. 建立连接——三次握手

客户端与服务端的Client_Server_Connect()函数内部内容是不一样的，这是由于二者对于数据的接收发送顺序导致的。这里仅展示客户端的作为样例展示。（这并不影响对于整体内容的判别，因为本质上函数内容仅是代码顺序的不同，以及服务端建连函数需要返回第三次握手中的Label内容

```
// 客户端与服务端建立连接（采取三次握手的方式
int Client_Server_Connect(SOCKET& socketClient, SOCKADDR_IN& servAddr, int& servAddrlen,
int label)
{
    Packet_Header packet;
    int size = sizeof(packet);  // size = 8
    char* buffer = new char[sizeof(packet)];  // 发送buffer

    // 第一次：客户端首先向服务端发送建立连接请求
    packet.tag = SYN;
    packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
    memcpy(buffer, &packet, sizeof(packet));

    if (sendto(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
servAddrlen) == -1)
    {
        cout << "sendto函数发送数据出错！" << endl;
        return 0;
    }

    // 设定超时重传机制
    u_long mode = 1;
    ioctlsocket(socketClient, FIONBIO, &mode);  // 非阻塞模式
    clock_t start = clock(); //记录发送第一次握手时间
```

```
    // 第二次：客户端接收服务端回传的握手
    while (recvfrom(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
&servAddrlen) <= 0) {
        if ((clock() - start) / 1000 > 1)   //时间超过1秒 标记为超时重传
        {
            packet.tag = SYN;
            packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
            memcpy(buffer, &packet, size);
            sendto(socketClient, buffer, size, 0, (sockaddr*)&servAddr, servAddrlen);
            start = clock();
            cout << "第一次握手超时，正在进行重传" << endl;
        }
    }
    cout << "成功发送第一次握手信息" << endl;

    // 对接收数据进行校验和检验
    memcpy(&packet, buffer, sizeof(packet));
    if ((packet.tag == ACK) && (compute_sum((WORD*)&packet, sizeof(packet)) == 0)) {
        cout << "成功收到第二次握手信息" << endl;
    }
    else {
        cout << "无法接收服务端回传建立可靠连接" << endl;
        return 0;
    }

    // 第三次：客户端发送第三次握手数据
    packet.tag = ACK_SYN;
    packet.datasize = label;  // 传送label标签将其保存在da'ta'si
    packet.checksum = 0;
    packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
    memcpy(buffer, &packet, sizeof(packet));
    if (sendto(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
servAddrlen) == -1) {
        cout << "sendto函数发送数据出错！" << endl;
        return 0;
    }
    cout << "成功发送第三次握手信息" << endl;
    cout << "客户端与服务端成功进行三次握手建立连接！---------可以开始发送/接收数据" << endl;
    mode = 0;  // 阻塞模式
    ioctlsocket(socketClient, FIONBIO, &mode);
    return 1;
}
```

5. 客户端与服务端交换数据缓冲区大小

考虑到实际情况下，客户端与服务端实际上双方的数据发送缓冲区大小是不一样的。对该情况进行模拟，让客户端与服务端在开始进行文件数据发送之前，先进行一次各自数据区大小的互相告知，这样双方就以二者的min作为缓冲区大小进行传递。

```
int Client_Server_Size(SOCKET& socketClient, SOCKADDR_IN& servAddr, int& servAddrlen,
int Size) {
    Packet_Header packet;
    int size = sizeof(packet);  // size = 8
    char* buffer = new char[sizeof(packet)];  // 发送buffer
    // 告知服务端 客户端的缓冲区为多大
    packet.tag = SYN;
    packet.datasize = Size;
    packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
    memcpy(buffer, &packet, sizeof(packet));
    if (sendto(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
servAddrlen) == -1) {
        cout << "sendto函数发送数据出错！" << endl;
        return 0;
    }
    // 设定超时重传机制
    u_long mode = 1;
    ioctlsocket(socketClient, FIONBIO, &mode);  // 非阻塞模式
    clock_t start = clock(); //记录发送第一次握手时间

    // 接收服务端回传告知的缓冲区大小
    while (recvfrom(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
&servAddrlen) <= 0) {
        if ((clock() - start) / 1000 > 1)   //时间超过1秒 标记为超时重传 {
            sendto(socketClient, buffer, size, 0, (sockaddr*)&servAddr, servAddrlen);
            start = clock();
            cout << "数据发送超时！正在进行重传" << endl;
        }
    } cout << "成功告知对方缓冲区大小" << endl;

    memcpy(&packet, buffer, sizeof(packet));  // 对接收数据进行校验和检验
    if ((packet.tag == ACK) && (compute_sum((WORD*)&packet, sizeof(packet)) == 0)) {
        cout << "成功收到对方缓冲区大小信息" << endl;
    }
    else {
        cout << "无法接收服务端回传建立可靠连接" << endl;
        return 0;
    }
    mode = 0;
    ioctlsocket(socketClient, FIONBIO, &mode);  // 阻塞模式

    if (Size > packet.datasize)     return packet.datasize;
    else    return Size;
}
```

6. 二进制读取&写入文件

读取文件

```
char InFileName[20];
//输入要读取的图像名
cout << "Enter File name:";
cin >> InFileName;
//文件指针
FILE* fp;
//以二进制方式打开图像
if ((fp = fopen(InFileName, "rb")) == NULL){
  cout << "Open file failed!" << endl;
  exit(0);
}
//获取文件数据总长度
fseek(fp, 0, SEEK_END);
int F_length = ftell(fp);
rewind(fp);
//根据文件数据长度分配内存buffer
char* FileBuffer = (char*)malloc(F_length * sizeof(char));
//将图像数据读入buffer
fread(FileBuffer, F_length, 1, fp);
fclose(fp);
```

写入文件

```
char* F_name = new char[20];
char* Message = new char[100000000];
int name_len = Recv_Messsage(client, serveraddr, length, F_name);   //获取文件名bytes长度
int file_len = Recv_Messsage(client, serveraddr, length, Message);  //获取文件bytes长度
string a;
for (int i = 0; i < name_len; i++){
 a = a + F_name[i];
}
cout << "接收文件名: " << a << endl;
cout << "接收文件数据大小: " << file_len << "bytes!" << endl;
FILE* fp;
if ((fp = fopen(a.c_str(), "wb")) == NULL){
 cout << "Open File failed!" << endl;
 exit(0);
}
//从buffer中写数据到fp指向的文件中
fwrite(Message, file_len, 1, fp);
fclose(fp);
```

7. 发送方发送文件数据

```
// 定义最大发送数据长度
int MAX_SIZE = 1024;
void Send_Message(SOCKET& socketClient, SOCKADDR_IN& servAddr, int& servAddrlen, char*
Message, int mes_size)
{
    int packet_num = mes_size / (MAX_SIZE) + (mes_size % MAX_SIZE != 0);
    int Seq_num = 0;  //初始化序列号
    Packet_Header packet;
    u_long mode = 1;
    ioctlsocket(socketClient, FIONBIO, &mode);  // 非阻塞模式

    for (int i = 0; i < packet_num; i++)
    {
        int data_len = (i == packet_num - 1 ? mes_size - (packet_num - 1) * MAX_SIZE :
MAX_SIZE);
        char* buffer = new char[sizeof(packet) + data_len];  // 发送buffer
        packet.tag = 0;
        packet.seq = Seq_num;
        packet.datasize = data_len;
        packet.checksum = 0;
        packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
        memcpy(buffer, &packet, sizeof(packet));
        char* mes = Message + i * MAX_SIZE;
        memcpy(buffer + sizeof(packet), mes, data_len);

        // 发送数据
        sendto(socketClient, buffer, sizeof(packet)+data_len, 0, (sockaddr*)&servAddr,
servAddrlen);
        cout << "Begin sending message..... datasize:" << data_len << " bytes!" << "
flag:"
            << int(packet.tag) << " seq:" << int(packet.seq) << " checksum:" <<
int(packet.checksum) << endl;
        clock_t start = clock();    //记录发送时间

        while (recvfrom(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
&servAddrlen) <= 0) {
            if ((clock() - start) / 1000 > 1) {  //时间超过1秒 标记为超时重传
                sendto(socketClient, buffer, sizeof(packet) + data_len, 0,
(sockaddr*)&servAddr, servAddrlen);
                cout << "数据发送超时，正在重传！！！" << endl;
                start = clock();
            }
        }

        memcpy(&packet, buffer, sizeof(packet));
        if (packet.ack == (Seq_num+1)%(256) && (compute_sum((WORD*)&packet,
sizeof(packet)) == 0)) {
            cout << "Successful send and recv response  ack:"<<int(packet.ack) << endl;
```

```
        }
        else {
            // 服务端未接受到数据 重传数据
            if (packet.ack == Seq_num && (compute_sum((WORD*)&packet, sizeof(packet)) ==
0)) {
                cout << "服务端未接受到数据，正在重传！！！" << endl;
                i--;
                continue;
            }
            else {
                cout << "无法接收服务端回传" << endl;
                return;
            }
        }
        // Seq_num在 0-255之间
        Seq_num++;
        Seq_num = (Seq_num > 255 ? Seq_num - 256 : Seq_num);
    }

    //发送结束标志
    packet.tag = OVER;
    char* buffer = new char[sizeof(packet)];
    packet.checksum = 0;
    packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
    memcpy(buffer, &packet, sizeof(packet));
    sendto(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr, servAddrlen);
    cout << "The end tag has been sent" << endl;

    clock_t start = clock();
    while (recvfrom(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
&servAddrlen) <= 0) {
        if ((clock() - start) / 1000 > 1) {  //时间超过1秒 标记为超时重传
            sendto(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
servAddrlen);
            cout << "数据发送超时，正在重传！！！" << endl;
            start = clock();
        }
    }

    memcpy(&packet, buffer, sizeof(packet));
    if (packet.tag == OVER && (compute_sum((WORD*)&packet, sizeof(packet)) == 0))
        cout << "The end token was successfully received" << endl;
    else
        cout << "无法接收服务端回传" << endl;
    mode = 0;
    ioctlsocket(socketClient, FIONBIO, &mode);  // 阻塞模式
    return;
}
```

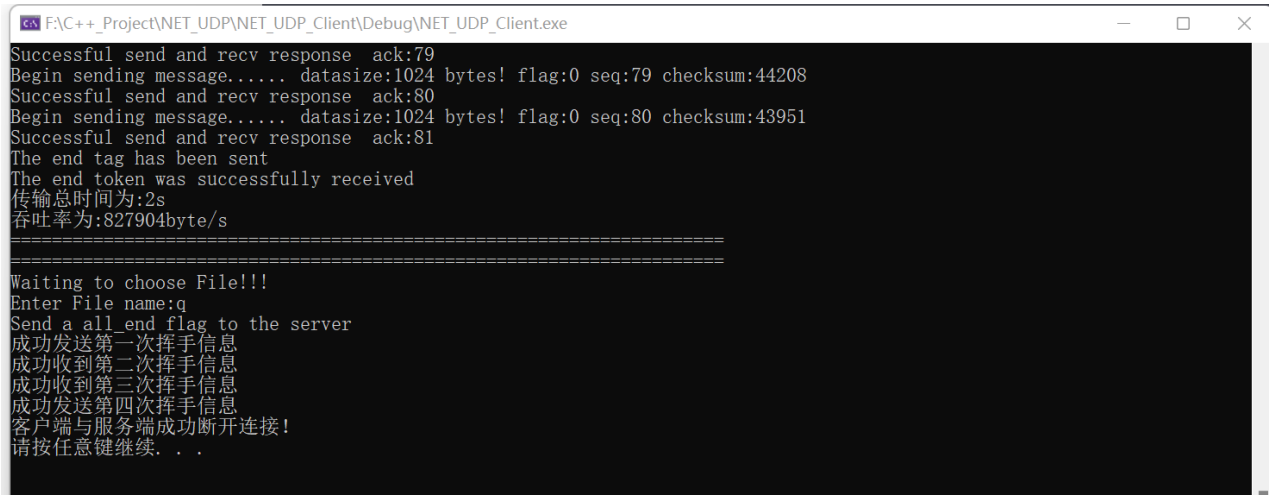
8. 接受方接收文件数据

```
// 定义最大接收数据长度
int MAX_SIZE = 1024;
int Recv_Messsage(SOCKET& socketServer, SOCKADDR_IN& clieAddr, int& clieAddrlen, char*
Mes)
{
    Packet_Header packet;
    char* buffer = new char[sizeof(packet) + MAX_SIZE];  // 接收buffer
    int ack = 1;  // 确认序列号
    int seq = 0;
    long F_Len = 0;      //数据总长
    int S_Len = 0;       //单次数据长度

    // 接收文件
    while (1) {
        while (recvfrom(socketServer, buffer, sizeof(packet) + MAX_SIZE, 0,
                        (sockaddr*)&clieAddr, &clieAddrlen) <= 0);
        memcpy(&packet, buffer, sizeof(packet));

        if (((rand() % (255 - 1)) + 1) == 199) {    // 模拟接收端未及时给发送方响应的情况
            cout << int(packet.seq) << "随机超时重传测试" << endl;
            continue;
        }
        // 结束标记
        if (packet.tag == OVER && (compute_sum((WORD*)&packet, sizeof(packet)) == 0)) {
            cout << "The end tag has been recved" << endl;
            break;
        }

        // 接收数据
        if (packet.tag == 0 && (compute_sum((WORD*)&packet, sizeof(packet)) == 0)) {

            // 对接收数据加以确认，否则发送标记告知客户端需要重发
            if (packet.seq != seq) {
                Packet_Header temp;
                temp.tag = 0;
                temp.ack = seq;
                temp.checksum = 0;
                temp.checksum = compute_sum((WORD*)&temp, sizeof(temp));
                memcpy(buffer, &temp, sizeof(temp));
                sendto(socketServer, buffer, sizeof(temp), 0, (sockaddr*)&clieAddr,
clieAddrlen);
                cout << "Send a resend flag to the client" << endl;
                continue;
            }

            // 成功收到数据
            S_Len = packet.datasize;
```

```
            cout << "Begin recving message...... datasize:" << S_Len << " bytes!" << "
flag:"
                << int(packet.tag) << " seq:" << int(packet.seq) << " checksum:" <<
int(packet.checksum) << endl;
            memcpy(Mes + F_Len, buffer + sizeof(packet), S_Len);
            F_Len += S_Len;

            // 返回 告知客户端已成功收到
            packet.tag = 0;
            packet.ack = ack++;
            packet.seq = seq++;
            packet.datasize = 0;
            packet.checksum = 0;
            packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
            memcpy(buffer, &packet, sizeof(packet));
            sendto(socketServer, buffer, sizeof(packet), 0, (sockaddr*)&clieAddr,
clieAddrlen);
            cout << "Successful recv and send response  ack : " << int(packet.ack) <<
endl;
            seq = (seq > 255 ? seq - 256 : seq);
            ack = (ack > 255 ? ack - 256 : ack);
        }
    }

    // 发送结束标志
    packet.tag = OVER;
    packet.checksum = 0;
    packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
    memcpy(buffer, &packet, sizeof(packet));
    sendto(socketServer, buffer, sizeof(packet), 0, (sockaddr*)&clieAddr, clieAddrlen);
    cout << "The end tag has been sent" << endl;
    return F_Len;
}
```

9. 循环发送：可以持续发送文件（主动按q程序才会退出

```
while(1){
    ......
    // 全局结束
    if (InFileName[0] == 'q' && strlen(InFileName) == 1) {
        Packet_Header packet;
        char* buffer = new char[sizeof(packet)];  // 发送buffer
        packet.tag = END;
        packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
        memcpy(buffer, &packet, sizeof(packet));
        sendto(client, buffer, sizeof(packet), 0, (sockaddr*)&serveraddr, length);
        cout << "Send a all_end flag to the server" << endl;
        break;
    }
    ......
}

// 接收文件
while (1) {
    while (recvfrom(socketServer, buffer, sizeof(packet) + MAX_SIZE, 0,
(sockaddr*)&clieAddr, &clieAddrlen) <= 0);
    memcpy(&packet, buffer, sizeof(packet));

    // END 全局结束
    if (packet.tag == END && (compute_sum((WORD*)&packet, sizeof(packet)) == 0)) {
        cout << "The all_end tag has been recved" << endl;
        return 999;
    }
    ......
}
```



0. 断开连接——四次挥手

客户端

```
int Client_Server_Disconnect(SOCKET& socketClient, SOCKADDR_IN& servAddr, int&
servAddrlen)
{
    Packet_Header packet;
    char* buffer = new char[sizeof(packet)];  // 发送buffer
    int size = sizeof(packet);
    // 客户端第一次发起挥手
    packet.tag = FIN_ACK;
    packet.checksum = 0;
    packet.checksum = compute_sum((WORD*)&packet, size);
    memcpy(buffer, &packet, size);
    if (sendto(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
servAddrlen) == -1) {
        cout << "sendto函数发送数据出错! " << endl;
        return 0;
    }
    u_long mode = 1;
    ioctlsocket(socketClient, FIONBIO, &mode);  // 非阻塞模式
    clock_t start = clock();   // 记录第一次挥手的时间

    // 客户端接收服务端发来的第二次挥手
    while (recvfrom(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
&servAddrlen) <= 0) {
        if ((clock() - start) / 1000 > 1) {  //时间超过1秒 标记为超时重传
            sendto(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
servAddrlen);
            start = clock();
            cout << "第一次握手超时，正在进行重传" << endl;
        }
    }
    cout << "成功发送第一次挥手信息" << endl;

    // 对接收数据进行校验和检验
    memcpy(&packet, buffer, sizeof(packet));
    if ((packet.tag == ACK) && (compute_sum((WORD*)&packet, sizeof(packet)) == 0)) {
        cout << "成功收到第二次挥手信息" << endl;
    }
    else {
        cout << "无法接收服务端回传断开连接" << endl;
        return 0;
    }

    // 客户端接收服务端发来的第三次挥手
    while (1) {
```

```
        if (recvfrom(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
&servAddrlen) == -1) {
            cout << "无法接收服务端回传断开连接" << endl;
            return 0;
        }
        memcpy(&packet, buffer, sizeof(packet));    // 对接收数据进行校验和检验
        if ((packet.tag == FIN_ACK) && (compute_sum((WORD*)&packet, sizeof(packet)) ==
0)) {
            cout << "成功收到第三次挥手信息" << endl;
            break;
        }
    }
    mode = 0;
    ioctlsocket(socketClient, FIONBIO, &mode);  // 阻塞模式
    // 第四次：客户端发送第四次挥手数据
    packet.tag = ACK;
    packet.checksum = 0;
    packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
    memcpy(buffer, &packet, sizeof(packet));
    if (sendto(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
servAddrlen) == -1) {
        cout << "sendto函数发送数据出错！" << endl;
        return 0;
    }
    cout << "成功发送第四次挥手信息" << endl;
    cout << "客户端与服务端成功断开连接！" << endl;
    return 1;
}
```

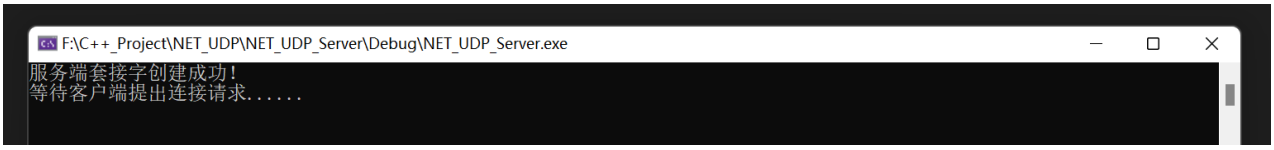
服务端

```
int Client_Server_Disconnect(SOCKET& socketServer, SOCKADDR_IN& clieAddr, int&
clieAddrlen)
{
    Packet_Header packet;
    char* buffer = new char[sizeof(packet)];  // 发送buffer
    int size = sizeof(packet);
    // 接收客户端发来的第一次挥手信息
    while (1) {
        if (recvfrom(socketServer, buffer, sizeof(packet), 0, (sockaddr*)&clieAddr,
&clieAddrlen) == -1) {
            cout << "无法接收客户端发送的连接请求" << endl;
            return 0;
        }
        memcpy(&packet, buffer, sizeof(packet));
        if (packet.tag == FIN_ACK && (compute_sum((WORD*)&packet, sizeof(packet)) == 0))
{
            cout << "成功接收第一次挥手信息" << endl;
            break;
        }
    }
```

```
    // 第二次：服务端向客户端发送挥手信息
    packet.tag = ACK;
    packet.checksum = 0;
    packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
    memcpy(buffer, &packet, sizeof(packet));
    if (sendto(socketServer, buffer, sizeof(packet), 0, (sockaddr*)&clieAddr,
clieAddrlen) == -1){
        return 0;
    }
    cout << "成功发送第二次挥手信息" << endl;

    // 第三次：服务端向客户端发送挥手信息
    packet.tag = FIN_ACK;
    packet.checksum = 0;
    packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
    memcpy(buffer, &packet, sizeof(packet));
    if (sendto(socketServer, buffer, sizeof(packet), 0, (sockaddr*)&clieAddr,
clieAddrlen) == -1){
        return 0;
    }
    u_long mode = 1;
    ioctlsocket(socketServer, FIONBIO, &mode);  // 非阻塞模式
    clock_t start = clock();//记录第三次挥手发送时间

    // 第四次：服务端接收客户端发送的挥手信息
    while (recvfrom(socketServer, buffer, sizeof(packet), 0, (sockaddr*)&clieAddr,
&clieAddrlen) <= 0) {
        if ((clock() - start) / 1000 > 1) {  //时间超过1秒 标记为超时重传
            sendto(socketServer, buffer, sizeof(packet), 0, (sockaddr*)&clieAddr,
clieAddrlen);
            start = clock();
            cout << "第三次挥手超时，正在进行重传" << endl;
        }
    }
    mode = 0;
    ioctlsocket(socketServer, FIONBIO, &mode);  // 阻塞模式
    cout << "成功发送第三次挥手信息" << endl;

    // 对接收数据进行校验和检验
    memcpy(&packet, buffer, sizeof(packet));
    if ((packet.tag == ACK) && (compute_sum((WORD*)&packet, sizeof(packet)) == 0))
        cout << "成功收到第四次挥手信息" << endl;
    else {
        cout << "无法接收客户端回传建立可靠连接" << endl;
        return 0;
    }
    cout << "客户端与服务端成功断开连接！" << endl;
    return 1;
}
```

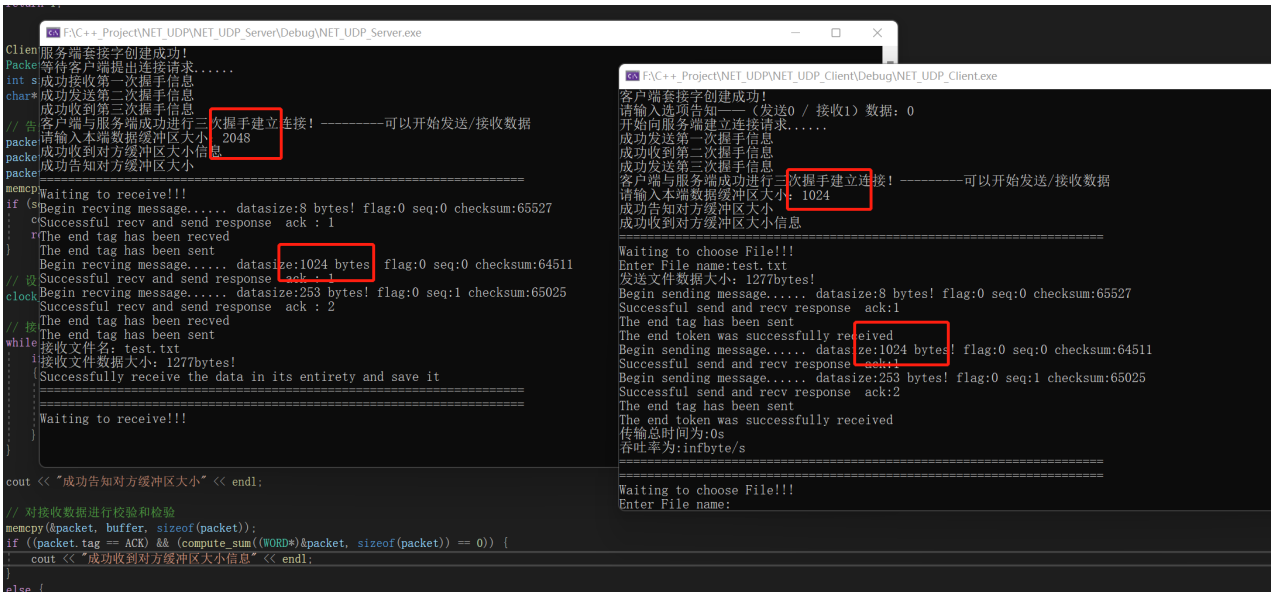
四、实际效果展示

- 先打开服务端，再打开客户端。在客户端应用界面输入本次客户端的功能定位：发送方OR接收方



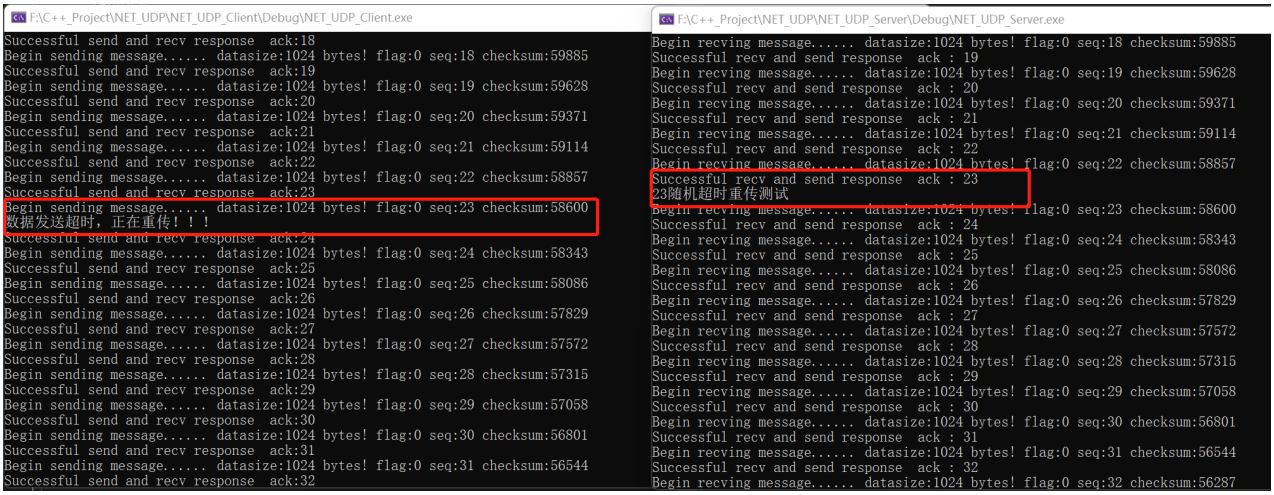
- 客户端选定好功能定位后，客户端与服务端之间便开始三次握手的过程，建立连接（本次展示：客户端作为接收方、服务端作为发送方

- 客户端与服务端双方互相告知各自数据缓冲区大小，二者取最小作为双方最终数据传输缓冲区交互的大小SIZE

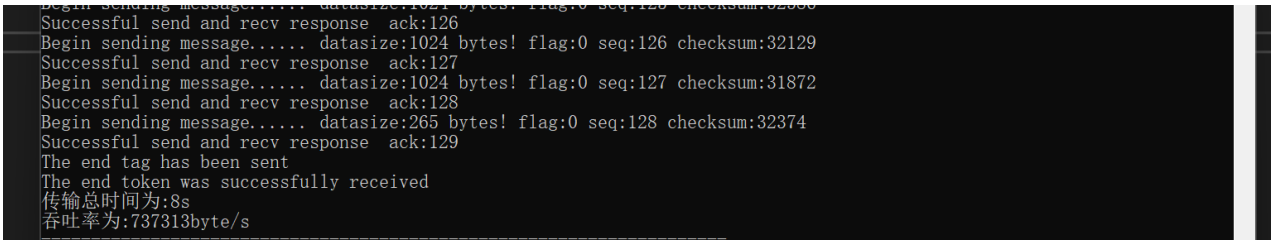


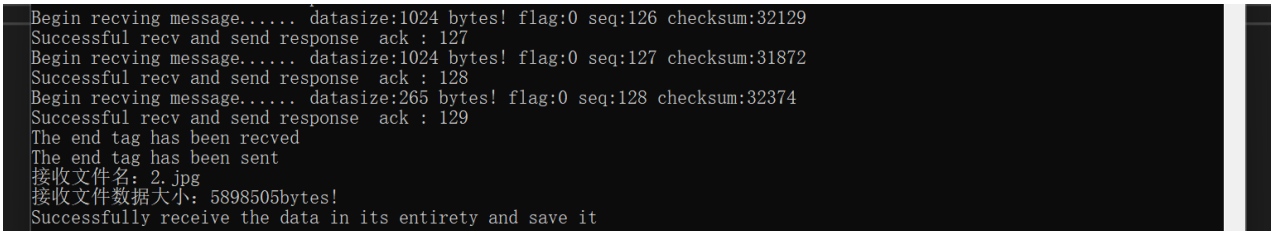
- 在实际实验冲添加对于超时重传的一个测试环节，检验代码的正确性与应用性。确保如若发生超时，发送方&接收方可以正确处理数据，保证数据不会出现错误失序、缺失的情况

```
if (((rand() % (255 - 1)) + 1) == 199) {    // 模拟接收端未及时给发送方响应的情况
    cout << int(packet.seq) << "随机超时重传测试" << endl;
    continue;
}
```

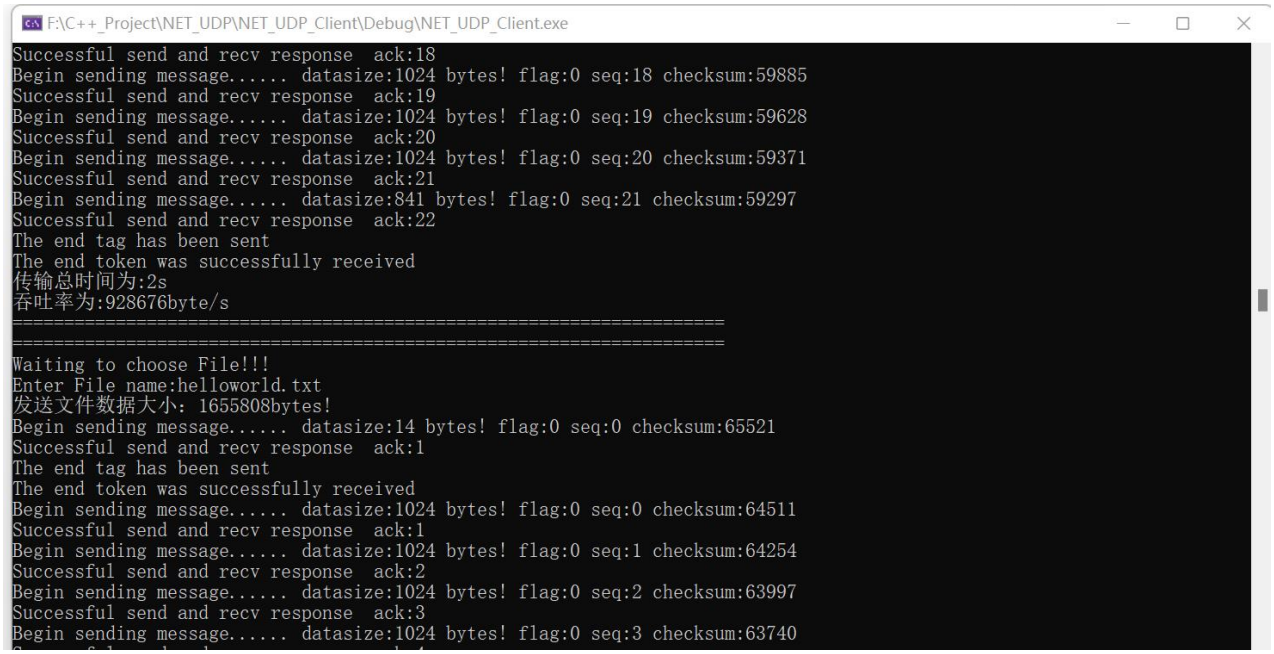


- 发送方需要Enter File Name选择发送的文件数据。选定后，发送方便会使用fstrean二进制方式打开文件，进行传输。接收方在收到所有内容之后，会将其保存写入





- 程序可以一直运行，发送文件数据



- 在用户主动按下q键的情况下，数据文件传输结束，进行断联过程，四次挥手双方断开连接

五、实验总结

基于UDP服务实现可靠传输实验第一部分，成功利用数据报套接字在用户空间实现面向连接的可靠数据传输，功能包括：建立连接、差错检测、确认重传。流量控制采用停等机制，完成给定测试文件的传输。

除却成功实现上述规定实验功能，作者还额外实现如下功能：

1. 双向传输：客户端与服务端双方都可以作为发送方，另一端作为接收方。这样的功能定义通过客户端在三次握手的第三次握手数据包发送时将**label**标签在双方之间进行传递，以此双方进行确定本次应用双方的角色。
2. 缓冲器大小协商确定：客户端与服务端在文件数据**send&recv**之前，先进行一次关于缓冲区大小的协商，客户端与服务端各自在数据帧中告知对方：自身的数据缓冲区大小，这样双方就可以选择出**min**值作为最终的缓冲区大小。
3. 程序不会在完成一次文件数据传输后就退出程序，而是在用户主动按**q**键后，才会**end**程序。
4. 在实验的过程中，主动添加超时重传的测试环节，测试发送方与接收方遇到超时情况下是否能够正确处理。

在后续进一步的实验设计中，希望能够实现更加安全、可靠、快速的文件数据传输。