Lab3-1——基于UDP服务设计可靠传输协议并编程实现

学号：2212452
姓名：孟启轩
专业：计算机科学与技术

一、实验要求

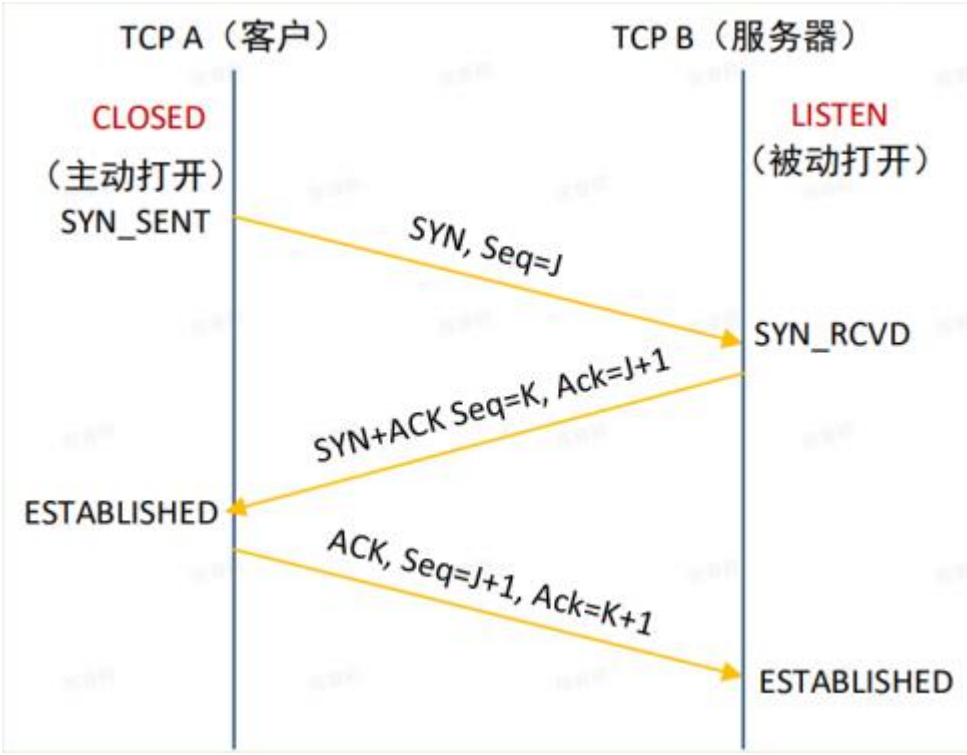
利用数据报套接字在用户空间实现面向连接的可靠数据传输，功能包括：建立连接、差错检测、确认重传等。流量控制采用停等机制，完成给定测试文件的传输。

二、实验设计

(一)原理探究

> 三次握手

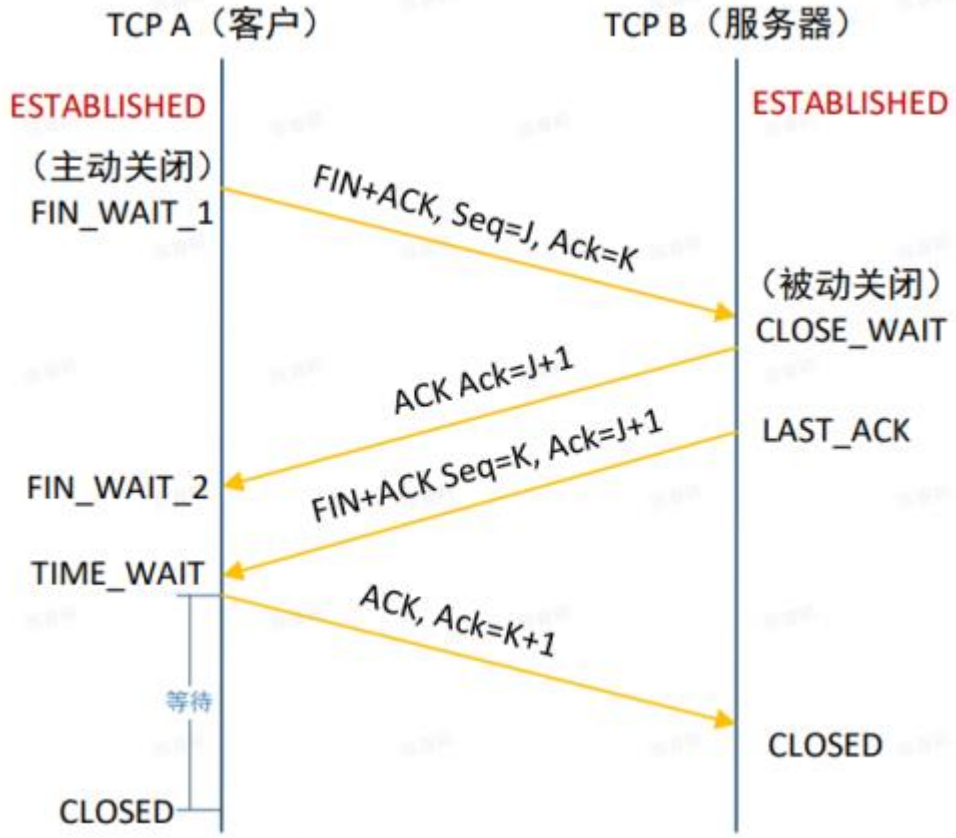
1. 第一次握手：客户端发送SYN包(seq=x)到服务器，并进入SYN_SEND状态，等待服务器确认
2. 第二次握手：服务器收到SYN包，必须确认客户的SYN(ack=x+1)，同时自己也发送一个SYN包(seq=y)，即SYN+ACK包，此时服务器进入SYN_RECV状态
3. 第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=y+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP连接都将被一直保持下去。



> 四次挥手

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。

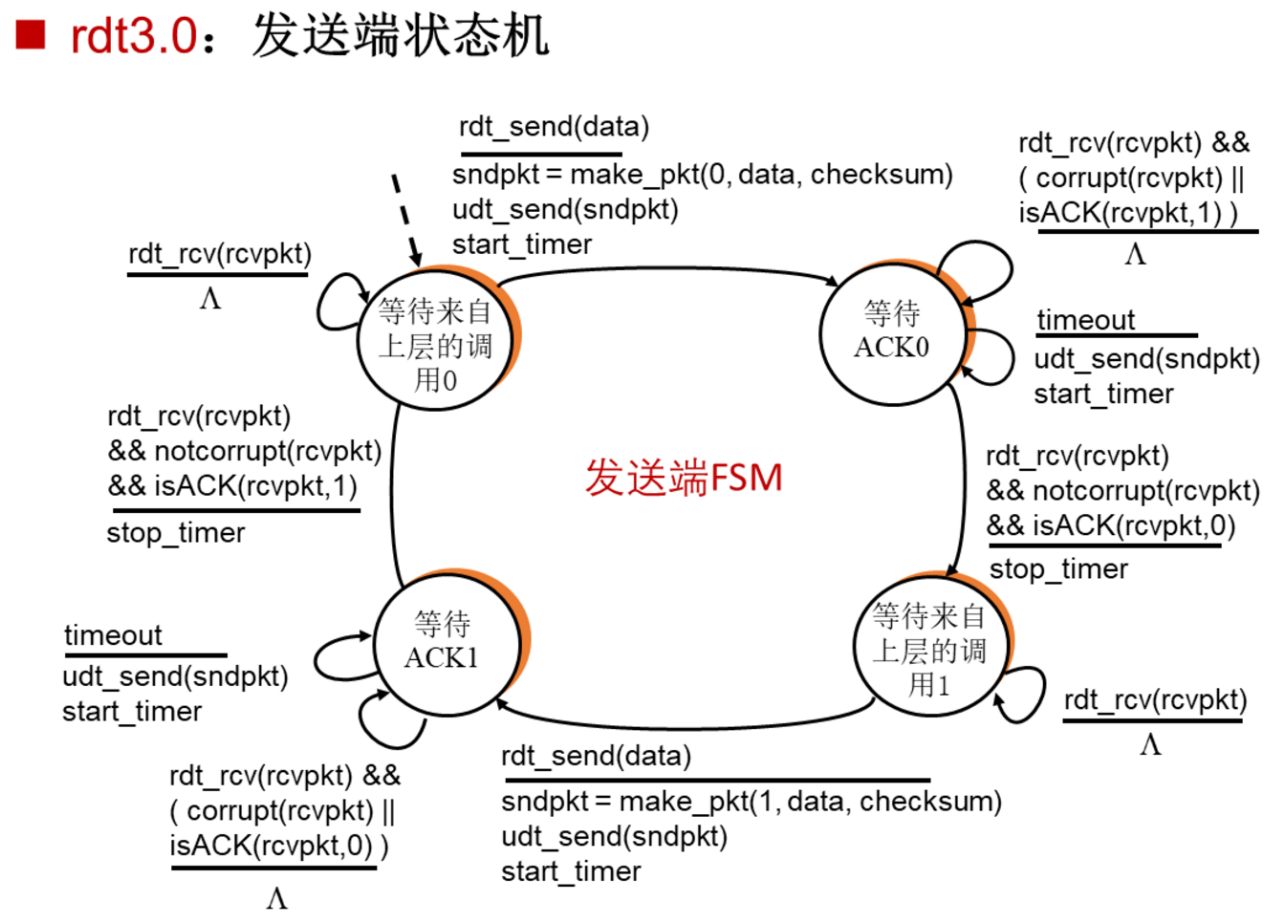
3. 第三次挥手：Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST ACK状态。
4. 第四次挥手：Client收到FIN后，Client进入TIME_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手。



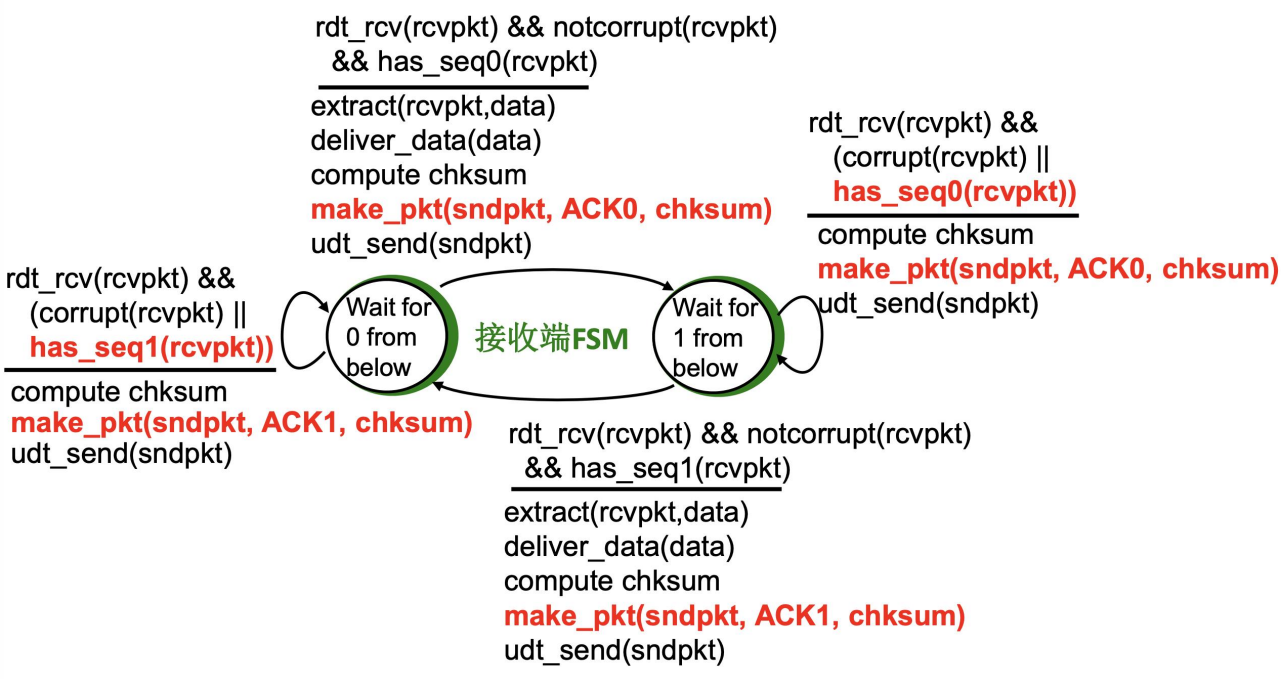
数据传输

发送端和接收端的接收机都采用rdt3.0的设计原则。

- 发送端的有限状态机：



- 接收端的有限状态机：



序列号与确认应答号

数据报的传输顺序通过序列号来保障，以确保数据传输的可靠性。每个响应包中也包含一个序列号，用于表明接收方已准备好处理对应的包。在传输数据报时，这个数据报会被放入重发队列，同时启动一个计时器。如果收到该数据包的确认信息，发送方会将此数据包从队列中删除。如果在计时器超时后仍未收到确认信息，则需

要重新发送该数据包。此外，通过数据分段中的序列号，可以确保所有传输的数据按正确的顺序进行重组，从而保证数据传输的完整性。

> 差错检测

差错检测（error detection），是指在发送的码序列（码字）中加入适当的冗余度以使得接收端能够发现传输中是否发生差错的技术。除了用于通信外，差错检测技术也广泛用于信息存储中。

> 超时重传

超时重传指的是在发送数据报文段后开始计时，设置一个等待确认应答到来的那个时间间隔。如果超过这个时间间隔，仍未收到对方发来的确认应答，发送端将进行数据重传。

> UDP协议

UDP是User Datagram Protocol的简称,中文名是用户数据报协议,是OSI参考模型中的传输层协议,它是一种无连接的传输层协议,提供面向事务的简单不可靠信息传送服务。

(二)协议设计

本程序中的相关设计如下：

> 三次握手建立连接

1. 客户端首先向服务器发送一个带有Hello标志（值为0x4）数据包。
2. 服务器接收到这个Hello数据包后，会进行校验并确认其有效性。
3. 随后，服务器向客户端发送一个带有Hello标志（值为0x5）的确认数据包。
4. 经过以上步骤，握手成功，从而建立起通信连接。

> 四次挥手断开连接

1. 客户端向服务器发送一个带有Goodbye标志（值为0x3）数据包。
2. 服务器收到这个Goodbye数据包后，会进行校验并确认其有效性。
3. 随后，服务器向客户端发送一个带有Goodbye标志（值为0x2）的确认数据包。
4. 完成以上步骤后，挥手成功，通信正式断开。

> 数据传输

- 客户端将文件数据划分为多个数据包，并通过UDP协议将这些数据包发送给服务器。每个数据包包含数据头部（包括数据大小、校验和、标志等信息）以及实际的文件数据。服务器在收到数据包后，会进行校验和确认，并根据数据包的标志进行相应的处理。
- 如果接收到的是正常的数据包，服务器会向客户端发送确认数据包。如果接收到的是最后一个数据包，服务器将发送一个带有结束标志（0x7）的确认数据包给客户端。

> 超时重传

- 在初始化函数中，设置了超时时间（TIMEOUT）。
- 在packet_send函数和初始化函数中，使用计时器（基于QueryPerformanceCounter函数）来检测是否超时，并在超时后进行相应的重传操作。

> 差错检验

- 在Header类中，使用sum字段来存储校验和。

- 在cksum函数和setSum函数中，实现了报文的校验和计算和设置。
- 而在packet_send函数中，在构建发送的报文时计算校验和，并在接收端对收到的报文进行校验和的验证。
- 此外，在check_sign函数中，验证了报文的标志位和校验和，以确保数据的完整性和可靠性。

三、设计实现

(一)校验和计算

- 发送方生成检验和

  1. 对数据报进行校验和运算的数据分成若干个16位的位串，每个位串看成一个二进制数
  2. 将首部中的检验和字段置为全0，该字段也参与检验和运算
  3. 对这些16位的二进制数进行1的补码和运算，累加的结果再取反码即生成了检验码。将检验码放入检验和字段中
  4. 其中1的补码和运算，即带循环进位的加法，最高位有进位应循环进到最低位

- 接收方校验检验和

  1. 接收方将接收的数据报按发送方的同样的方法进行1的补码和运算，累加的结果取反码
  2. 如果第一步的结果为0，表示传输正确
  3. 否则，说明传输出现差错

```
// 计算校验和的函数，用于差错检测
u_short cksum(u_short *buff, int size)
{
    int count = (size + 1) / 2;           // 计算16位数据块数量
    u_short *buf = new u_short[size + 1]; // 分配缓冲区
    memset(buf, 0, size + 1);             // 初始化缓冲区
    memcpy(buf, buff, size);              // 拷贝数据到缓冲区
    u_short sum = 0;                      // 初始化校验和

    // 计算校验和：将每个16位数据加到总和中，检测溢出
    while (count--)
    {
        sum += *buf++;
        if (sum & 0xFFFF0000)
        { // 每次累加后，如果sum溢出（即高于16位），则将其截断到16位，并增加1
            sum &= 0xFFFF;
            sum++;
        }
    }
    return ~(sum & 0xFFFF); // 返回校验和的反码
}

// 设置校验和
void setSum(Header &header)
{
    u_short s = cksum((u_short *)&header, sizeof(header));
    header.sum = s;
}
```

```
// 检查标志位和校验和是否正确
bool check_sign(Header header, u_char sign)
{
    if (header.flag == sign && cksum((u_short *)&header, sizeof(header)) == 0)
        return true;
    else
        return false;
}
```

(二)数据包和标志位

Header类负责管理数据包的必要元信息，比如大小、校验和、标志、确认号和序列号，以确保数据传输的可靠性和顺序。而Packet结构体则利用这些头部信息以及必要的数据存储空间，代表一个完整的网络数据包。

- 数据包的头部信息

```
class Header
{
public:
    u_short datasize = 0; // 数据包大小
    u_short sum = 0;      // 校验和
    u_char flag = 0;      // 标志位 (SYN, ACK, FIN 等)
    u_char ack = 0;       // 确认号 (ACK = seq + 1)
    u_short seq = 0;      // 序列号

    Header() { flag = 0; }

    // 设置头部字段
    void setHeader(u_short d, u_char f, u_short se)
    {
        this->datasize = d;
        this->seq = se;
        this->flag = f;
    }

    // 打印头部信息
    void show_header()
    {
        cout << " sum: " << (int)this->sum
             << " seq: " << (int)this->seq
             << " flag: " << (int)this->flag << endl;
    }
};
```

- 数据包结构体

```
struct Packet
{
```

```
    Header header; // 包含头部信息
    char *Buffer;  // 用于存储数据部分

    // 初始化数据缓冲区
    Packet()
    {
        Buffer = new char[MAXSIZE + sizeof(header)]();//分配存储空间
    }
};
```

(三)握手与挥手

- 客户端

```
bool interact(SOCKET &sockServ, SOCKADDR_IN &ClientAddr, int &ClientAddrLen,
string type, int flag)
{
    // type: hello 或 goodbye
    int sendsign = 0, recvsign = 0;
    if (type == "hello")
    {
        sendsign = 0x4; // 00000100
        recvsign = 0x5; // 00000101 ack+syn
    }
    else
    {
        sendsign = 0x2; // 00000010
        recvsign = 0x3; // 00000011 ack+fin
    }
    Header recvh;
    Header sendh;
    char *Buffer = new char[sizeof(recvh)];
    int res = 0;

    SOCKADDR_IN client_addr;
    client_addr.sin_family = AF_INET;
    client_addr.sin_port = htons(4002);              // 替换为客户端期望端口
    client_addr.sin_addr.s_addr = htonl(INADDR_ANY); // 允许任何IP地址

    if (flag && bind(sockServ, (SOCKADDR *)&client_addr, sizeof(client_addr)) ==
SOCKET_ERROR)
    {
        cout << "绑定失败!" << endl;
        return false;
    }
    sendh.setHeader(0, sendsign, 0); // 设置flag
    setSum(sendh);                   // 校验和初始化为0，计算后填充
    memcpy(Buffer, &sendh, sizeof(sendh));
    // 发送第一次握手信息
    res = sendto(sockServ, Buffer, sizeof(recvh), 0, (sockaddr *)&ClientAddr,
ClientAddrLen);
```

```
    if (res == -1)
    {
        cout << "第一次" << type << "失败 " << endl;
        return false;
    }
    else
        cout << "第一次" << type << " 成功 " << endl;
    long long head;
    QueryPerformanceCounter((LARGE_INTEGER *)&head);
    u_long mode = 1; // 设置非阻塞模式
    ioctlsocket(server, FIONBIO, &mode);
    while (recvfrom(sockServ, Buffer, sizeof(recvh), 0, (sockaddr *)&ClientAddr,
&ClientAddrLen) <= 0)
    {
        if (time(head) > TIMEOUT)
        {
            memcpy(Buffer, &sendh, sizeof(sendh));
            sendto(sockServ, Buffer, sizeof(recvh), 0, (sockaddr *)&ClientAddr,
ClientAddrLen);
            QueryPerformanceCounter((LARGE_INTEGER *)&head);
            cout << "第一次" << type << "超时，重新发送......" << endl;
            continue;
        }
    }
    memcpy(&recvh, Buffer, sizeof(recvh)); // 解析接受的头部信息
    if (check_sign(recvh, recvsign))
    {

        cout << "第二次" << type << " 成功 " << endl;
        cout << "---------------------------- 成功 ------------------------------
--" << endl;
        return true;
    }
    else
    {
        cout << "----------------------------成功--------------------------------
" << endl;
        return false;
    }
}
```

- 服务端

```
bool interact(SOCKET &sockServ, SOCKADDR_IN &ClientAddr, int &ClientAddrLen,
string type)
{
    // type 是hello或goodbye
    int sendsign = 0, recvsign = 0;
    if (type == "hello")
    {
        sendsign = 0x5; // ack+syn
```

```
            recvsign = 0x4; //
        }
        else
        {
            sendsign = 0x3;
            recvsign = 0x2;
        }

        Header recvh;
        Header sendh;
        char *Buffer = new char[sizeof(recvh)];
        int res = 0;
        // 接收第一次握手信息
        while (true)
        {
            res = recvfrom(sockServ, Buffer, sizeof(recvh), 0, (sockaddr
*)&ClientAddr, &ClientAddrLen);
            if (res == -1)
            {
                cout << "第一次 " << type << " 失败" << endl;
                return false;
            }
            memcpy(&recvh, Buffer, sizeof(recvh));
            if (check_sign(recvh, recvsign))
            {
                SEQ = recvh.seq;
                cout << "第一次 " << type << " 成功" << endl;
                sendh.setHeader(0, sendsign, 0);
                setSum(sendh);
                memcpy(Buffer, &sendh, sizeof(sendh));
                res = sendto(sockServ, Buffer, sizeof(sendh), 0, (sockaddr
*)&ClientAddr, ClientAddrLen);
                if (res == -1)
                {
                    return false;
                }
                else
                {
                    cout << "连接/断开成功!" << endl;
                    break;
                }
            }
            else
            {
                cout << "连接/断开失败!" << endl;
                return false;
            }
        }
    }
```

建立连接：

- 服务端建立连接：

    - 在init函数中，服务端初始化了server_addr结构体，包括IP地址、端口等信息，并创建了UDP套接字server。
    - 通过bind函数将套接字绑定到指定的IP和端口，然后进入监听状态，以等待客户端的连接请求。

- 客户端建立连接：

    - 在init函数中，客户端同样初始化了server_addr结构体，然后创建了UDP套接字server。
    - 客户端使用bind函数绑定其端口，以便能够接收来自服务端的响应。

- 握手过程：

    - 服务端和客户端通过interact函数进行握手。interact函数根据传入的参数type来区分是建立连接还是断开连接。根据不同的type，设置相应的发送和接收标志位：

    - 客户端首先发送第一次握手信息（带有特定的标志位），服务端接收并作出回应，完成第一次握手。

    - 客户端接收到服务端的回应后，完成第二次握手，连接建立成功。

    - 倘若客户端没有接收到ACK回应，则重新进行第二次握手。

断开连接：

- 客户端断开连接：

    - 在main函数中，客户端在文件传输结束后，通过interact函数进行断开连接的挥手：发送带有断开连接标志位的报文，服务端收到后进行回应，从而完成断开连接。

- 服务端断开连接：

    - 在main函数中，服务端在文件传输结束后，通过interact函数进行断开连接的挥手：发送带有断开连接标志位的报文，客户端收到后进行回应，完成断开连接。

(四)客户端发送数据

发送数据主要涉及到以下几个步骤：

1. 将所发数据包进行分包，每个包最大大小为4096字节。
2. 设置数据头信息：在数据包中设置数据头的序列号、标志位等信息。对于发送的最后一个数据包，需将标志位设置为结束标志。
3. 计算校验和：利用`cksum`函数计算数据头的校验和，以确保数据的完整性。
4. 发送数据包：使用`sendto`函数将构建好的数据包发送到目标地址。
5. 待确认消息：启用非阻塞模式，等待接收确认消息。如果在规定时间内未收到确认，则进行数据包的重传。
6. 处理接收到的确认消息：解析接收到的确认消息，检查其是否为期望的 ACK 序列号，以及是否为结束标志。如果满足条件，则表示数据发送成功。

- 分包传输

```
void send(SOCKET &clientsocket, SOCKADDR_IN &serveraddr, int &addrlen, char
*message, int len)
{
```

```
    // 将数据拆分成4096字节的块进行分包传输
    int num = 0;
    if (len % MAXSIZE == 0)
    { // 整除
        num = len / MAXSIZE;
    }
    else
    {
        num = len / MAXSIZE + 1;
    }
    cout << "总包数:" << num << endl;
    SEQ = 0; // 序列号从零开始
    for (int i = 0; i < num; i++)
    {
        cout << "send No.[" << i + 1 << "]packet" << endl;
        if (i != num - 1)
        {
            int templen = MAXSIZE;
            packet_send(clientsocket, serveraddr, addrlen, message + i * MAXSIZE,
templen, false);
        }
        else
        {
            int templen = len - (num - 1) * MAXSIZE;
            packet_send(clientsocket, serveraddr, addrlen, message + i * MAXSIZE,
templen, true); // 最后一个包
        }
        SEQ++;
    }
}
```

- 数据包发送

```
void packet_send(SOCKET &clientsocket, SOCKADDR_IN &serveraddr, int &addrlen, char
*message, int len, bool END)
{
    Header recvh;
    Packet sendp;
    char *buf = new char[sizeof(recvh)];
    sendp.header.seq = SEQ;
    if (!END)
        sendp.header.flag = 0x0;
    else
        sendp.header.flag = 0x7; // 00000111 结束标志
    sendp.header.datasize = len;
    memcpy(sendp.Buffer, &sendp.header, sizeof(sendp.header)); // 确保头部准确填充
    memcpy(sendp.Buffer + sizeof(sendp.header), message, sizeof(sendp.header) +
len);
    u_short tempsum = cksum((u_short *)&sendp.Buffer, sizeof(sendp.Buffer));
    sendp.header.sum = tempsum;
    memcpy(sendp.Buffer, &sendp.header, sizeof(sendp.header));
    sendto(clientsocket, sendp.Buffer, sizeof(sendp.header) + len, 0, (sockaddr
```

```
*)&serveraddr, addrlen); // 发送数据包
    cout << "发送数据包 " << len << " Byte ";
    sendp.header.show_header();
    long long head;
    QueryPerformanceCounter((LARGE_INTEGER *)&head);
    while (true)
    {
        u_long mode = 1;
        ioctlsocket(clientsocket, FIONBIO, &mode); // 非阻塞模式
        while (recvfrom(clientsocket, buf, sizeof(recvh), 0, (sockaddr
*)&serveraddr, &addrlen) <= 0)
        {
            if (time(head) > TIMEOUT)
            {
                sendp.header.setHeader(len, 0x0, SEQ);
                memcpy(sendp.Buffer, &sendp.header, sizeof(sendp.header));
                memcpy(sendp.Buffer + sizeof(sendp.header), message,
sizeof(sendp.header) + len);
                u_short tempsum = cksum((u_short *)&sendp, sizeof(sendp.Buffer));
                sendp.header.sum = tempsum;
                memcpy(sendp.Buffer, &sendp.header, sizeof(sendp.header));
                sendto(clientsocket, sendp.Buffer, sizeof(sendp.header) + len, 0,
(sockaddr *)&serveraddr, addrlen);
                cout << "超时重传:";
                sendp.header.show_header();
                QueryPerformanceCounter((LARGE_INTEGER *)&head);
                continue;
            }
        }
        memcpy(&recvh, buf, sizeof(recvh));
        if (recvh.flag)
        {
            ACK = recvh.ack;
            if (ckack() && ckend(recvh))
            {
                cout << " 服务器已接收 " << endl;
                break;
            }
            else if (ckack && cksend(recvh))
            {
                cout << "发送成功!" << endl;
                break;
            }
            else
            {
                continue;
            }
        }
    }
    u_long mode = 0;
    ioctlsocket(clientsocket, FIONBIO, &mode); // 恢复为组塞模式
}
```

> 阻塞模式和非阻塞模式

- 阻塞和非阻塞是对操作请求者在等待返回结果时所处状态的描述。在阻塞模式下，当前线程会被挂起，直到操作请求的结果返回；而在非阻塞模式下，如果系统无法立即提供结果，线程不会被挂起，而是直接返回一个错误信息。因此，在非阻塞的情况下，调用者需要定期轮询以检查处理状态。
- 本质上，阻塞和非阻塞是对socket的不同处理方式，不会影响socket的连接及通信双方。通信的任一方都可以选择使用阻塞或非阻塞模式，例如，客户端采用阻塞模式，而服务器的`accept`可以设置为非阻塞模式，这都是可以的。
- 非阻塞是对阻塞方式的一种改进方案，通常能在大部分情况下提供更好的性能。虽然设置超时时间可以部分解决阻塞带来的问题，但它仍然会占用一定的时间，并且超时时间的设定往往较为复杂：若设置过短，可能导致操作过早中止；若设置过长则没有实际意义。因此，系统实现了非阻塞方式并结合异步编程，从而大幅提升了效率。

> 超时重传/丢包/校验和错误

倘若客户端发送数据包后，没有收到服务端返回的ACK确认消息，或者超出时间限制，则客户端需要重新发送数据包。

(五)接收数据

接收数据主要涉及到以下几个步骤：

- 丢包模拟：通过随机丢弃部分数据包以测试系统的稳健性。
- 数据完整性验证：通过校验和确保数据包在传输过程中未被损坏。
- 序列号管理：确保数据包按照正确的顺序被接收和处理。
- 确认机制：对于每个接收的数据包进行确认，并在必要时请求重传。
- 最终确认：在所有数据接收完成后，发送结束确认包，以通知客户端数据传输的结束。

```
int receive(SOCKET &servsocket, SOCKADDR_IN &clientaddr, int &len, char *message)
{
    Packet recvp;
    Header sendh;
    int offset = 0; // 总文件偏移量
    char *buf = new char[sizeof(sendh)];
    SEQ = 0;
    while (true)
    {
        int recvlen = recvfrom(servsocket, recvp.Buffer, sizeof(recvp.header) +
MAXSIZE, 0, (sockaddr *)&clientaddr, &len);
        int drop = rand() % 100;
        // 模拟丢包，有 5/100 的概率丢包
        if (drop < 5)
            continue;
        memcpy(&recvp.header, recvp.Buffer, sizeof(recvp.header));
        if (recvp.header.flag == 0x7 && cksum((u_short *)&recvp,
sizeof(recvp.Buffer)))
        {
            // 仅有头部
            SEQ = recvp.header.seq;
            memcpy(message + offset, recvp.Buffer + sizeof(recvp.header), recvlen
- sizeof(recvp.header));
```

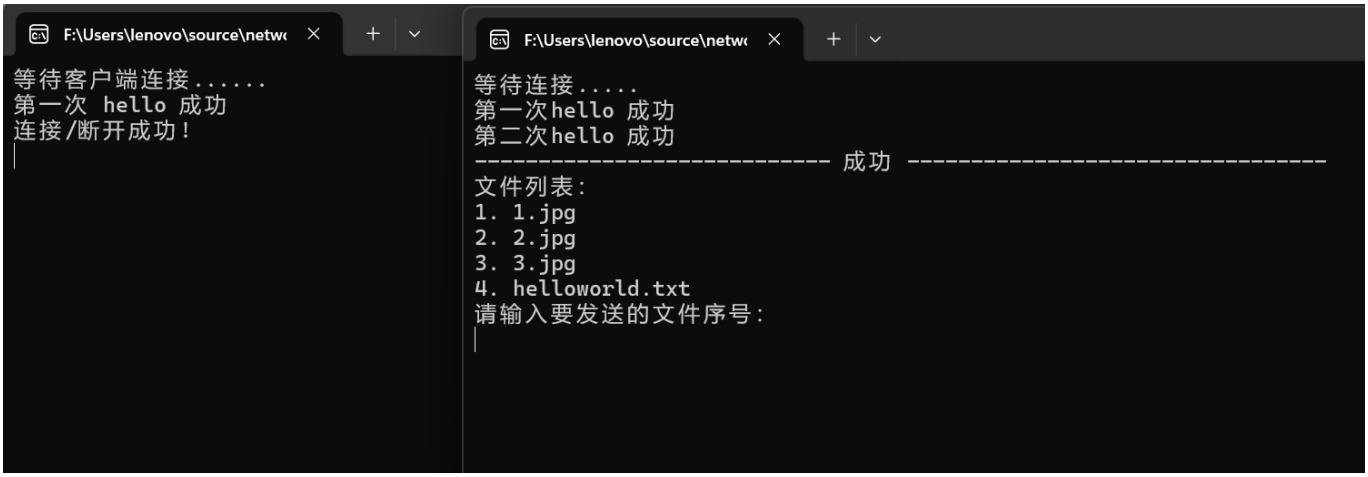
```
            offset += recvp.header.datasize;
            cout << "the file has been received." << endl;
            break;
        }
        if (recvp.header.flag == 0x0 && cksum((u_short *)&recvp,
sizeof(recvp.Buffer)))
        {
            // 判断包
            if (SEQ != recvp.header.seq)
            {
                ACK = (SEQ % 255) + 1;
                sendh.ack = (u_char)ACK;
                sendh.setHeader(0, 0x1, SEQ);
                setSum(sendh);
                memcpy(buf, &sendh, sizeof(sendh));
                sendto(servsocket, buf, sizeof(sendh), 0, (sockaddr *)&clientaddr,
len);
                cout << "不是这个数据包，请重新发送!";
                sendh.show_header();
                continue; // 丢弃
            }
            SEQ = recvp.header.seq;
            memcpy(message + offset, recvp.Buffer + sizeof(recvp.header), recvlen
- sizeof(recvp.header));
            offset += recvp.header.datasize;
            // 成功收到后，返回ACK
            ACK = (SEQ % 255) + 1;
            sendh.ack = (u_char)ACK;
            sendh.setHeader(0, 0x1, SEQ);
            setSum(sendh);
            memcpy(buf, &sendh, sizeof(sendh));
            sendto(servsocket, buf, sizeof(sendh), 0, (sockaddr *)&clientaddr,
len);
            cout << "检查数据包! send ack:" << (int)sendh.ack;
            sendh.show_header();
            SEQ++;
        }
    }

    ACK = (SEQ % 255) + 1;
    sendh.ack = (u_char)ACK;
    sendh.setHeader(0, 0x7, SEQ);
    setSum(sendh);
    memcpy(buf, &sendh, sizeof(sendh));
    if (sendto(servsocket, buf, sizeof(sendh), 0, (sockaddr *)&clientaddr, len) ==
-1)
    {
        return -1;
    }
    return offset;
}
```

四、实验结果

(一)建立连接

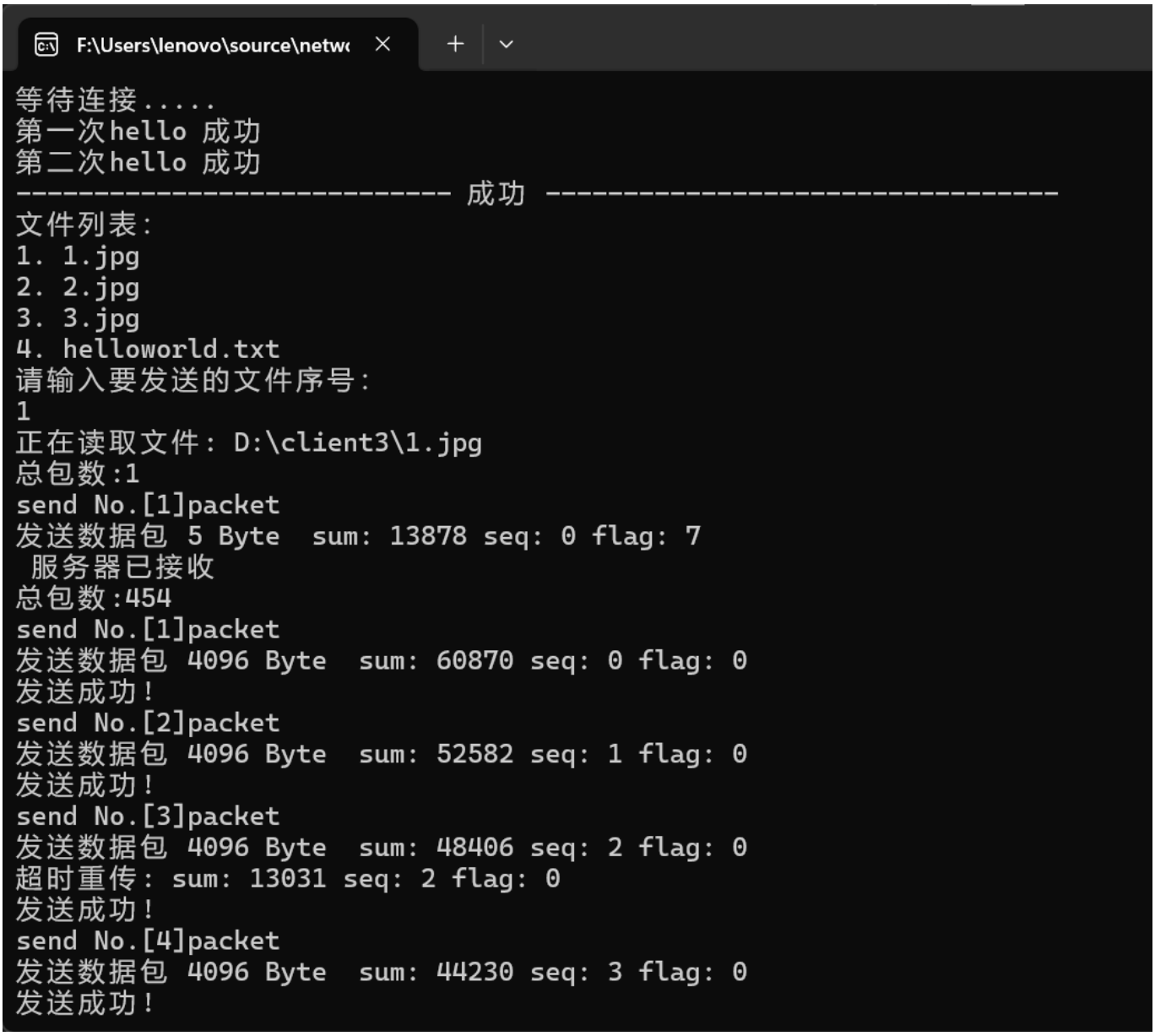
首先打开客户端和服务器端，握手建立连接如下图所示：



(二)数据传输

本次共需要传送四个文件，每个文件的传输及其传输后的总时间以及吞吐率如以下图片所示：

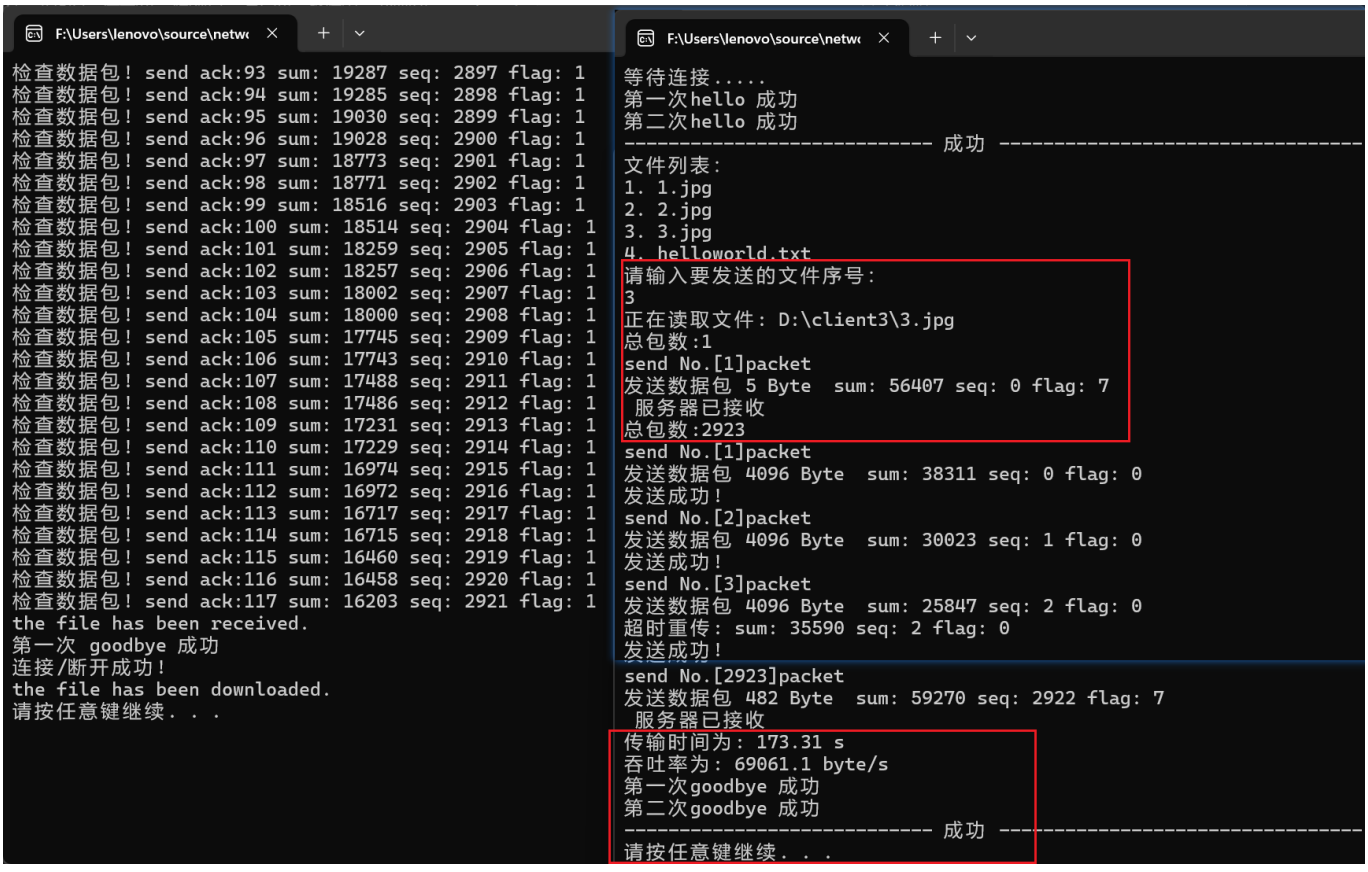
> 1.jpg


F:\Users\lenovo\source\netw
等待连接.....
第一次hello 成功
第二次hello 成功
---------- 成功 ----------
文件列表:
1. 1.jpg
2. 2.jpg
3. 3.jpg
4. helloworld.txt
请输入要发送的文件序号:
1
正在读取文件: D:\client3\1.jpg
总包数:1
send No.[1]packet
发送数据包 5 Byte sum: 13878 seq: 0 flag: 7
服务器已接收
总包数:454
send No.[1]packet
发送数据包 4096 Byte sum: 60870 seq: 0 flag: 0
发送成功!
send No.[2]packet
发送数据包 4096 Byte sum: 52582 seq: 1 flag: 0
发送成功!
send No.[3]packet
发送数据包 4096 Byte sum: 48406 seq: 2 flag: 0
超时重传: sum: 13031 seq: 2 flag: 0
发送成功!
send No.[4]packet
发送数据包 4096 Byte sum: 44230 seq: 3 flag: 0
发送成功!


F:\Users\lenovo\source\netw
检查数据包! send ack:174 sum: 10280 seq: 428 flag: 1
检查数据包! send ack:175 sum: 10025 seq: 429 flag: 1
检查数据包! send ack:176 sum: 10023 seq: 430 flag: 1
检查数据包! send ack:177 sum: 9768 seq: 431 flag: 1
检查数据包! send ack:178 sum: 9766 seq: 432 flag: 1
检查数据包! send ack:179 sum: 9511 seq: 433 flag: 1
检查数据包! send ack:180 sum: 9509 seq: 434 flag: 1
检查数据包! send ack:181 sum: 9254 seq: 435 flag: 1
检查数据包! send ack:182 sum: 9252 seq: 436 flag: 1
检查数据包! send ack:183 sum: 8997 seq: 437 flag: 1
检查数据包! send ack:184 sum: 8995 seq: 438 flag: 1
检查数据包! send ack:185 sum: 8740 seq: 439 flag: 1
检查数据包! send ack:186 sum: 8738 seq: 440 flag: 1
检查数据包! send ack:187 sum: 8483 seq: 441 flag: 1
检查数据包! send ack:188 sum: 8481 seq: 442 flag: 1
检查数据包! send ack:189 sum: 8226 seq: 443 flag: 1
检查数据包! send ack:190 sum: 8224 seq: 444 flag: 1
检查数据包! send ack:191 sum: 7969 seq: 445 flag: 1
检查数据包! send ack:192 sum: 7967 seq: 446 flag: 1
检查数据包! send ack:193 sum: 7712 seq: 447 flag: 1
检查数据包! send ack:194 sum: 7710 seq: 448 flag: 1
检查数据包! send ack:195 sum: 7455 seq: 449 flag: 1
检查数据包! send ack:196 sum: 7453 seq: 450 flag: 1
检查数据包! send ack:197 sum: 7198 seq: 451 flag: 1
检查数据包! send ack:198 sum: 7196 seq: 452 flag: 1
the file has been received.
第一次 goodbye 成功
连接/断开成功!
the file has been downloaded.
请按任意键继续. . .
F:\Users\lenovo\source\netw
send No.[447]packet
发送数据包 4096 Byte sum: 18048 seq: 446 flag: 0
发送成功!
send No.[448]packet
发送数据包 4096 Byte sum: 48640 seq: 447 flag: 0
发送成功!
send No.[449]packet
发送数据包 4096 Byte sum: 13680 seq: 448 flag: 0
发送成功!
send No.[450]packet
发送数据包 4096 Byte sum: 4944 seq: 449 flag: 0
发送成功!
send No.[451]packet
发送数据包 4096 Byte sum: 44272 seq: 450 flag: 0
发送成功!
send No.[452]packet
发送数据包 4096 Byte sum: 9312 seq: 451 flag: 0
发送成功!
send No.[453]packet
发送数据包 4096 Byte sum: 18063 seq: 452 flag: 0
发送成功!
send No.[454]packet
发送数据包 1865 Byte sum: 13695 seq: 453 flag: 7
服务器已接收
传输时间为: 30.386 s
吞吐率为: 61125.4 byte/s
第一次goodbye 成功
第二次goodbye 成功
---------- 成功 ----------
请按任意键继续. . .

2.jpg

3.jpg



helloworld.txt

(三)断开连接

- 挥手断开连接及文件的存储如下图所示：

(四)结果展示

接收到的文件：

名称	修改日期	类型	大小
helloworld.txt	2024/11/29 20:06	文本文档	1,617 KB
3.jpg	2024/11/29 20:01	JPG 图片文件	11,689 KB
2.jpg	2024/11/29 19:57	JPG 图片文件	5,761 KB
1.jpg	2024/11/29 19:54	JPG 图片文件	1,814 KB
server.exe	2024/11/29 18:54	应用程序	141 KB
server.pdb	2024/11/29 18:54	Program Debug Da...	2,100 KB
client.exe	2024/11/29 18:52	应用程序	166 KB
client.pdb	2024/11/29 18:52	Program Debug Da...	2,332 KB

可以看到四个文件都与原文件信息完全相同！

六、遇到的问题与总结

(1)问题

本次实验主要遇到以下几个问题：

- 我将测试文件都放在了D盘的某一文件夹下，客户端读取文件列表时，由于没有正确设置路径，回导致程序会从程序所在文件夹（即debug文件夹）进行读取与发送，这就导致一直重新握手建立连接，程序无法正常运行。最后通过重新设置文件输出路径解决。
- 超时时间设置为`2s`，时间过长。

(2)总结

本次实验通过基于UDP协议的可靠传输协议设计与实现，提升了我对网络传输可靠性、效率及协议设计的理解。通过理论研究和实践编码，成功实现了多个文件的可靠传输，所有接收到的文件均与原文件一致，验证了传输过程的有效性。实验过程中，我深入学习了数据包结构、校验和机制、超时重传及差错检测方法，提高了网络编程与调试能力，同时也培养了处理异常的思维。希望能进一步探索更高级的网络协议与技术，并结合所学知识，提升网络数据传输的效率与可靠性。