

实验3-1基于UDP服务设计可靠传输协议

姓名: 孟笑朵

学号: 2010349

- 实验要求
- 参考资料
- 实验过程
 1. UDP编程过程
 2. 协议设计与实验原理
 3. 建立连接
 4. 断开连接
 5. 差错检测
 6. 停等机制
 7. 确认重传机制
 8. 读写数据
- 程序运行界面

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。

本实验根据TCP设计可靠传输协议, 建立连接为三次握手模式🤝, 断开连接为两次挥手模式👋, 包含差错检测, 设计确认重传机制, 采用的停等机制为**rdt3.0**模式。

参考资料

1. [\(33条消息\) C++开发UDP通信: 使用socket创建UDP服务器端和客户端 智能之心的博客-CSDN博客](#)
2. [4.1 TCP 三次握手与四次挥手面试题 | 小林coding \(xiaolincoding.com\)](#)
3. [4.17 如何基于 UDP 协议实现可靠传输? | 小林coding \(xiaolincoding.com\)](#)

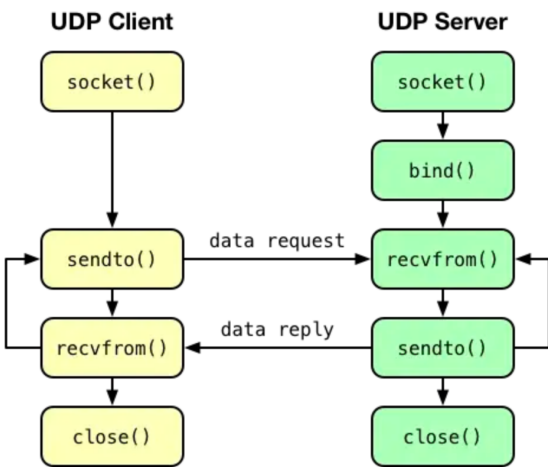
实验过程

在实验开始部分, 需要先学习UDP的编程方式, 如下所示:

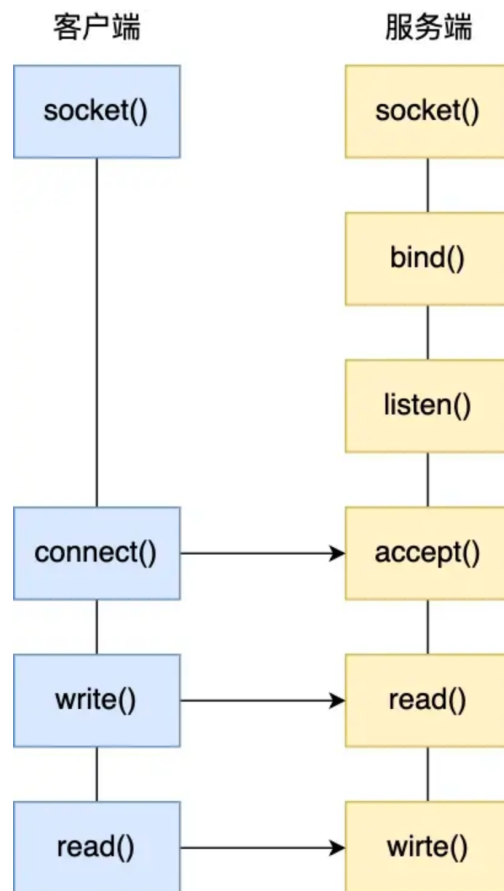
UDP编程过程

UDP编程过程和TCP编程过程很类似, 但是也有区别, 如下图比较所示

UDP编程过程



TCP编程过程



具体的UDP编程代码框架如下图所示:

客户端:

C++

```
//配置socket环境
WSADATA wsaData;
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
    cout << "WSAStartup error:" << GetLastError() << endl;
    return false;
}
//建立客户端socket对象
SOCKET m_Socket;
m_Socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
if (m_Socket == INVALID_SOCKET)
{
    closesocket(m_Socket);
    m_Socket = INVALID_SOCKET;
    return false;
}
//配置远程连接地址
SOCKADDR_IN m_RemoteAddress; //远程地址
int m_RemoteAddressLen;
```

```

    const char* ip = "127.0.0.1";
    int port = 1000;
    m_RemoteAddress.sin_family = AF_INET;
    m_RemoteAddress.sin_port = htons(port);
    m_RemoteAddressLen = sizeof(m_RemoteAddress);
    inet_pton(AF_INET, ip, &m_RemoteAddress.sin_addr);
//建立连接接收和发送消息
    char recvBuf[1024] = { 0 };
    char sendBuf[1024] = "Nice to meet you!";
    while (1) {
        int sendLen = sendto(m_Socket, sendBuf, strlen(sendBuf), 0,
(sockaddr*)&m_RemoteAddress, m_RemoteAddressLen);
        if (sendLen > 0) {
            std::printf("发送到远程端连接, 其ip: %s, port: %d\n",
inet_ntoa(m_RemoteAddress.sin_addr), ntohs(m_RemoteAddress.sin_port));
            cout << "发送到远程端的信息: " << sendBuf << endl;
        }

        int recvLen = recvfrom(m_Socket, recvBuf, 1024, 0, NULL, NULL);
        if (recvLen > 0) {
            std::printf("接收到一个连接, 其ip: %s, port: %d\n",
inet_ntoa(m_RemoteAddress.sin_addr), ntohs(m_RemoteAddress.sin_port));
            cout << "接收到一个信息: " << recvBuf << endl;
        }
    }
//关闭套字节
    closesocket(m_Socket);
    WSACleanup();

```

服务端:

```

// socket环境
WSADATA wsaData;
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
    cout << "WSAStartup error:" << GetLastError() << endl;
    return false;
}
// 建立服务端socket
SOCKET m_Socket;
m_Socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

```

C++

```

    if (m_Socket == INVALID_SOCKET)
    {
        closesocket(m_Socket);
        m_Socket = INVALID_SOCKET;
        return false;
    }
// 绑定地址
    SOCKADDR_IN m_BindAddress;    //绑定地址
    SOCKADDR_IN m_RemoteAddress; //远程地址
    int m_RemoteAddressLen;
    const char* ip = "127.0.0.1";
    int port = 1000;
    m_BindAddress.sin_family = AF_INET;
    m_BindAddress.sin_addr.S_un.S_addr = inet_addr(ip);
    m_BindAddress.sin_port = htons(port);
    auto ret = bind(m_Socket, (sockaddr*)&m_BindAddress,
sizeof(SOCKADDR));
    if (ret == SOCKET_ERROR)
    {
        closesocket(m_Socket);
        m_Socket = INVALID_SOCKET;
        return false;
    }
// 接收和发送
    m_RemoteAddressLen = sizeof(m_RemoteAddress);
    while (1) {
        int recvLen = recvfrom(m_Socket, recvBuf, 1024, 0,
(sockaddr*)&m_RemoteAddress, &m_RemoteAddressLen);
        if (recvLen > 0) {
            std::printf("接收到一个连接, 其ip: %s, port: %d\n",
inet_ntoa(m_RemoteAddress.sin_addr), ntohs(m_RemoteAddress.sin_port));
            cout << "接收到一个信息: " << recvBuf << endl;

        }
        int sendLen = sendto(m_Socket, sendBuf, strlen(sendBuf), 0,
(sockaddr*)&m_RemoteAddress, m_RemoteAddressLen);
        if (sendLen > 0) {
            cout << "发送到远程端的信息: " << sendBuf << endl;
        }
    }

```

```
        Sleep(2000);
    }
//关闭连接
    closesocket(m_Socket);
    WSACleanup();
}
```

协议设计与实验原理

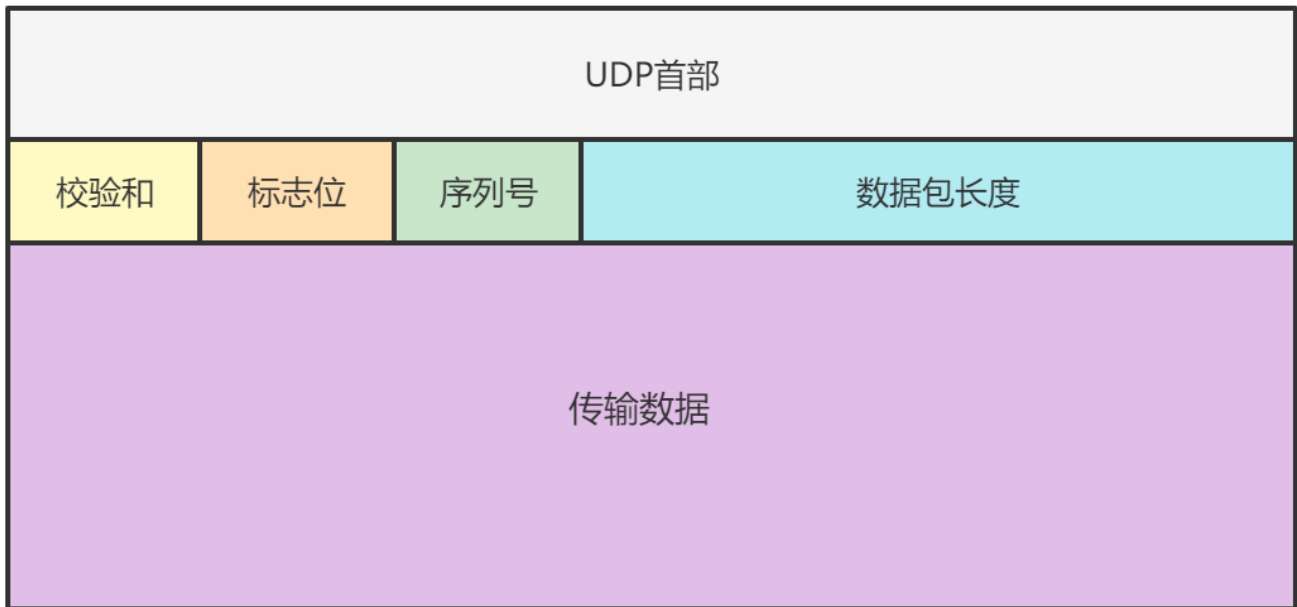
我们知道UDP传输本身有自己的数据报格式, 如下图所示

■ UDP数据报格式

- 长度：包含头部、以字节计数
- 校验和：为可选项，用于差错检测

0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
源端口号（Source Port）																目的端口号（Destination Port）															
长度（Length）																校验和（Checksum）															
数据（Data）																															

另外, UDP在发送和接收数据报的时候会产生**伪首部**, 用于计算校验和, 伪首部包含了源IP地址和目的IP地址, 以及协议类型等字段, 在这里为了专门检验传输数据部分, 我们专门计算了针对当前传输数据的校验和, 在UDP本身数据报报头的基础上, 我们在数据部分增加专门用于可靠传输的协议头部分, 如下所示为当前数据报报头:



说明

1. 校验和: 一个字节, 专门用于计算标志位数据部分的校验和;
2. 标志位: 一个字节, 包含三次连接的SYN, ACK, 两次挥手的FIN, 数据传输的ACK, 数据传输是否为最后一个数据报的END标志位;
3. 序列号: 一个字节, 数据传输的SEQ
4. 数据包长度: 传输数据为可选长度, 标记传输数据的长度, 4个字节(也就是说Data部分不能超过4个字节表示的数据范围)

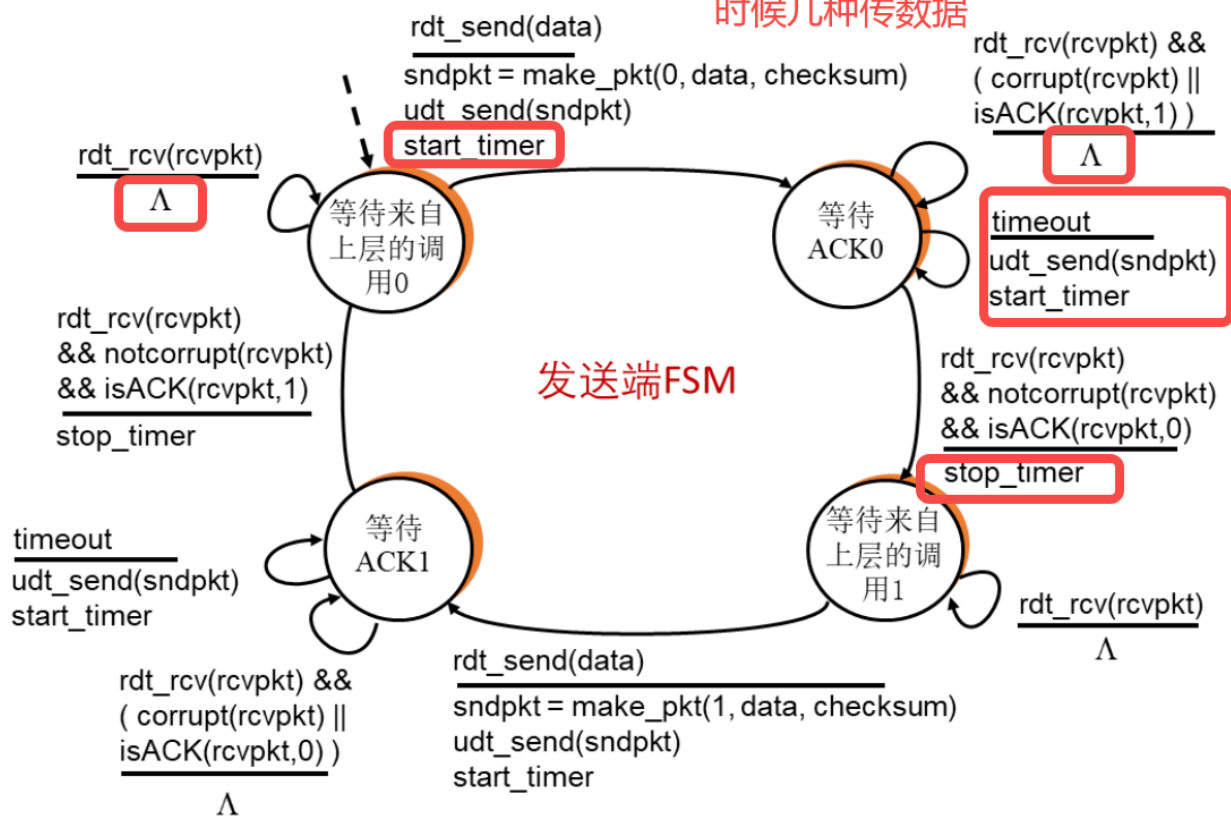
```
//设计报文格式
struct msgHead {
    unsigned char checknum;
    unsigned char flag; //标志位
    unsigned char seq; //表示数据包的序列号, 一个字节为周期
    int length; //表示数据包的长度 (包含头)
};
```

C++

在设计的过程中, 本程序实现了RDT3.0的功能, 按照RDT3.0中**超时重传**, **差错检验**的机制设计了本程序的对应的机制, 其中RDT3.0的发送端状态机如下图所示:

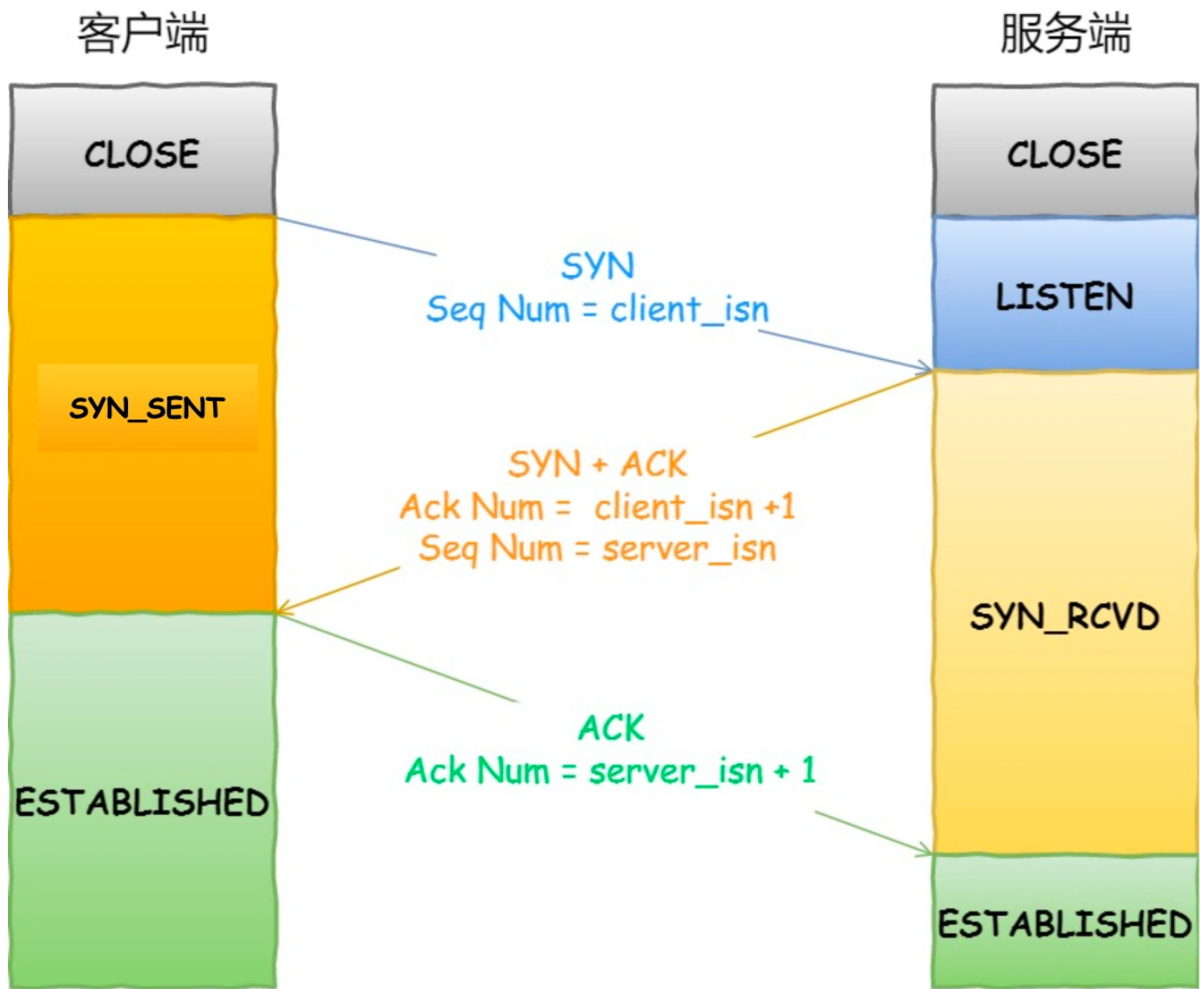
■ rdt3.0: 发送端状态机

Λ代表发送端不做事情, 在timeout的时候几种传数据



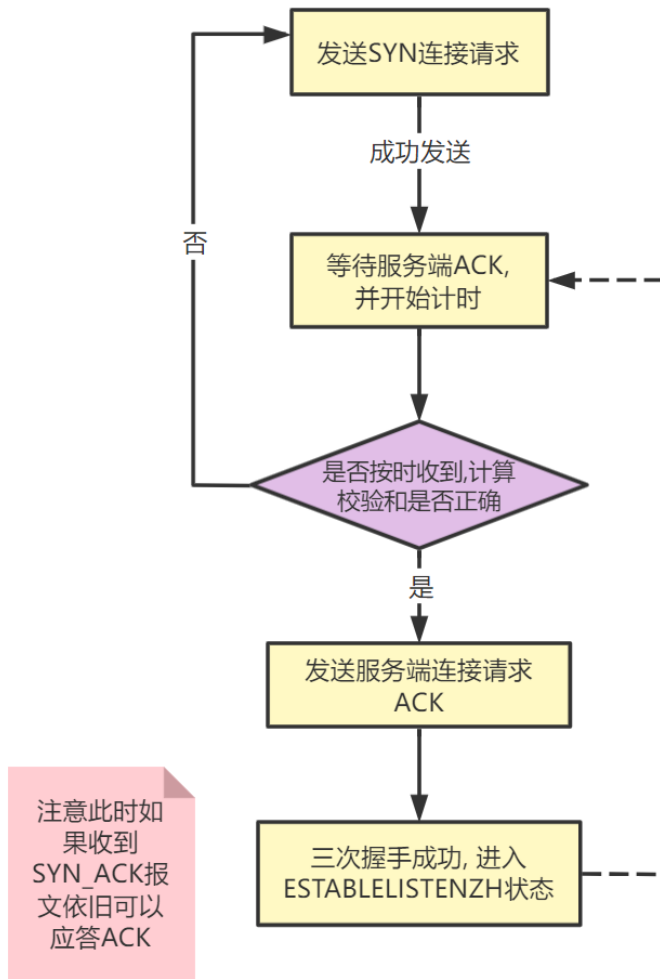
建立连接

在这里用一张简单的图来回顾三次握手的过程

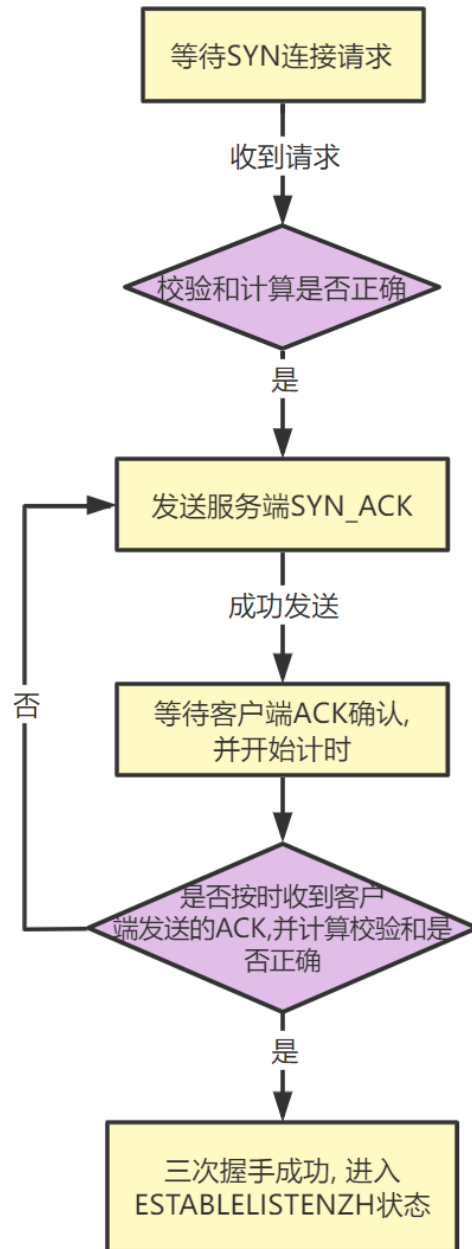


原理不再赘述, 我们来研究其中的过程, 我们需要考虑到其中ACK或者SYN丢失的情况, 这就需要设计对应的**超时重传**机制, 主体的实现逻辑如下图所示:

客户端三次握手处理流程图



服务端三次握手处理流程图



注: 这里没有设计对应的重传次数, 在滑动窗口的设计当中补上了这个不足之处

根据以上的处理流程图, 分别设计客户端和服务端的三次连接的代码, 如下所示:

客户端

```
//三次握手建立连接
while (1)
{
    char sendBuf[2];
    sendBuf[1] = FIRST_SHAKE;
    sendBuf[0] = check_sum(sendBuf + 1, 1);
    int ret = sendto(m_ClientSocket, sendBuf, 2, 0,
```

C++

```

(sockaddr*)&m_ServerAddress, sizeof(m_ServerAddress));
    if (ret != SOCKET_ERROR)
    {
        printf("%s\n", "成功发送第一次握手SYN请求!");
    }
    else
        return 0;
    //开始计时
    int begin = clock();
    char recv[2];
    int len = sizeof(m_ServerAddress);
    int fail = 0;
    //超时重发
    while (recvfrom(m_ClientSocket, recv, 2, 0,
(sockaddr*)&m_ServerAddress, &len) == SOCKET_ERROR)
    {
        if (clock() - begin > TIMEOUT)
        {
            fail = 1;
            break;
        }
    }
    //接收成功第一次握手ACK, 向服务端第二次握手SYN发送ACK
    if (fail == 0 && check_sum(recv, 2) == 0 && recv[1] ==
SECOND_SHAKE)
    {
        printf("%s\n", "成功接收服务端SYN_ACK!");
        sendBuf[1] = THIRD_SHAKE;
        sendBuf[0] = check_sum(sendBuf + 1, 1);
        sendto(m_ClientSocket, sendBuf, 2, 0,
(sockaddr*)&m_ServerAddress, sizeof(m_ServerAddress));
        printf("%s\n", "发送第三次握手ACK!");
        break;
    }
}
printf("三次握手成功连接服务端\n");

```

服务端

```

//服务端与客户端建立连接
while (1)
{
    //三次握手实现
    char recv[2];
    int len_tmp = sizeof(m_ClientAddress);
    while (recvfrom(m_ServerSocket, recv, 2, 0,
(sockaddr*)&m_ClientAddress, &len_tmp) == SOCKET_ERROR);
    //一次握手
    if (check_sum(recv, 2) != 0 || recv[1] != FIRST_SHAKE)
        continue;
    printf("%s\n", "成功接收客户端SYN请求!");
    while (1)
    {
        recv[1] = SECOND_SHAKE;
        recv[0] = check_sum(recv + 1, 1);
        //二次握手
        sendto(m_ServerSocket, recv, 2, 0,
(sockaddr*)&m_ClientAddress, sizeof(m_ClientAddress));
        printf("%s\n", "成功发送服务端SYN_ACK!");
        int begin = clock();
        int fail = 0;
        while (recvfrom(m_ServerSocket, recv, 2, 0,
(sockaddr*)&m_ClientAddress, &len_tmp) == SOCKET_ERROR)
        {
            if (clock() - begin > TIMEOUT)
            {
                fail = 1;
                break;
            }
        }
        if (fail == 0 && check_sum(recv, 2) == 0 && recv[1] ==
FIRST_SHAKE)
            continue;
        //三次握手
        if (fail == 0 && check_sum(recv, 2) == 0 && recv[1] ==
THIRD_SHAKE) {
            printf("%s\n", "成功接收客户端ACK!");
            break;
        }
    }
}

```

```

    }
    break;
}
printf("建立三次握手成功!\n");

```

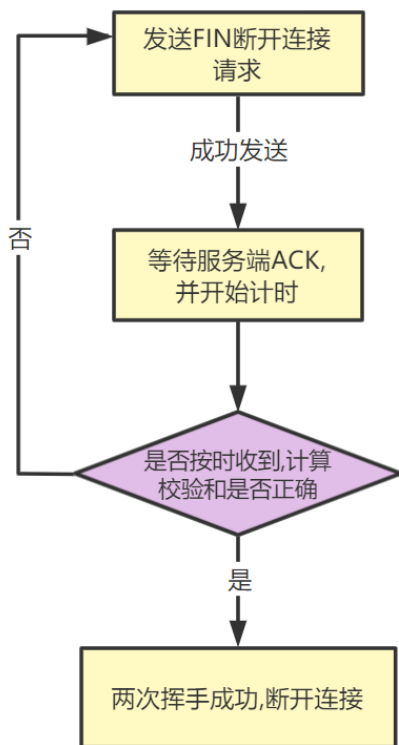
断开连接

我们知道四次挥手是考虑到当前服务器有可能没有将任务都处理完毕, 在这里因为我们简化了四次挥手的机制, 将四次挥手变为简单的两次挥手的过程, 大致的流程为:

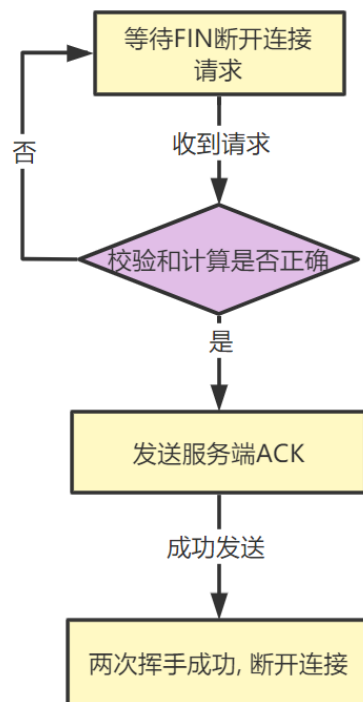
1. 客户端发送完毕, 向服务端发送关闭连接请求FIN;
2. 服务端收到请求, 返回关闭连接确认ACK;
3. 客户端收到服务端关闭连接的确认ACK, 关闭连接;

当然其中也涉及对应的重传机制, 如下图所示:

客户端断开连接流程



服务端断开连接流程



客户端代码如下:

```

while (1)
{
    char sendBuf[2];
    sendBuf[1] = FIRST_WAVE;

```

C++

```

        sendBuf[0] = check_sum(sendBuf + 1, 1);
        sendto(m_ClientSocket, sendBuf, 2, 0,
(sockaddr*)&m_ServerAddress, sizeof(m_ServerAddress));
        printf("%s\n", "客户端发送挥手消息");
        int begin = clock();
        char recv[2];
        int len = sizeof(m_ServerAddress);
        int fail = 0;
        //超时重传
        while (recvfrom(m_ClientSocket, recv, 2, 0,
(sockaddr*)&m_ServerAddress, &len) == SOCKET_ERROR)
        {
            if (clock() - begin > TIMEOUT)
            {
                fail = 1;
                break;
            }
        };
        if (fail == 0 && check_sum(recv, 2) == 0 && recv[1] ==
SECOND_WAVE)
        {
            printf("%s\n", "接收第二次挥手");
            break;
        }
    }
    printf("挥手成功, 断开连接\n");

```

服务端代码如下:

```

while (1)
{
    char recv[2];
    int len_tmp = sizeof(m_ClientAddress);
    while (recvfrom(m_ServerSocket, recv, 2, 0,
(sockaddr*)&m_ClientAddress, &len_tmp) == SOCKET_ERROR);
    if (check_sum(recv, 2) != 0 || recv[1] != (char)FIRST_WAVE)
        continue;
    printf("%s\n", "成功接收第一次挥手消息");
    recv[1] = SECOND_WAVE;
    recv[0] = check_sum(recv + 1, 1);

```

C++

```

        sendto(m_ServerSocket, recv, 2, 0, (sockaddr*)&m_ClientAddress,
sizeof(m_ClientAddress));
        printf("%s\n", "成功发送第二次挥手");
        break;
    }
    printf("挥手成功, 断开连接\n");

```

差错检测

我们在课上已经学习过了UDP的差错检测的计算机制, 如下图所示

■ 计算 UDP 校验和示例

伪首部	153.19.8.104			
	171.3.14.11			
UDP 首部	0	17	15	
	1087		13	
	15		0	
数据	01010100	01000101	01010011	01010100
	01001001	01001110	01000111	0填充

```

10011001 00010011 → 153.19
00001000 01101000 → 8.104
10101011 00000011 → 171.3
00001110 00001011 → 14.11
00000000 00010001 → 0 和 17
00000000 00001111 → 15
00000100 00111111 → 1087
00000000 00001101 → 13
00000000 00001111 → 15
00000000 00000000 → 0 (校验和)
01010100 01000101 → 数据
01010011 01010100 → 数据
01001001 01001110 → 数据
01000111 00000000 → 数据和 0 (填充)

```

按二进制反码运算求和
将得出的结果求反码

```

10010110 11101101 → 求和得出的结果
01101001 00010010 → 校验和

```

这里采用的计算方式是每16位求和, 计算求和的结果是否为0

```

//计算校验和
//UDP校验和的计算方法是:
//1. 按每16位求和得出一个32位的数;
//2. 如果这个32位的数, 高16位不为0, 则高16位加低16位再得到一个32位的数;
//3. 重复第2步直到高16位为0, 将低16位取反, 得到校验和。

```

```

//检验校验和

```

C++


```

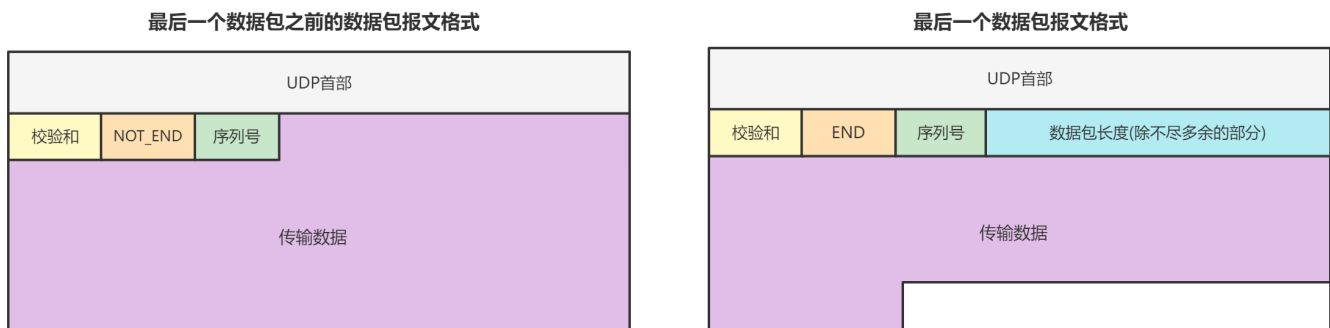
unsigned char check_sum(char* package, int len)
{
    if (len == 0) {
        return ~(0);
    }
    unsigned long sum = 0;
    int i = 0;
    while (len--) {
        sum += (unsigned char)package[i++];
        if (sum & 0xFF00) {
            sum &= 0x00FF;
            sum++;
        }
    }
    return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

```

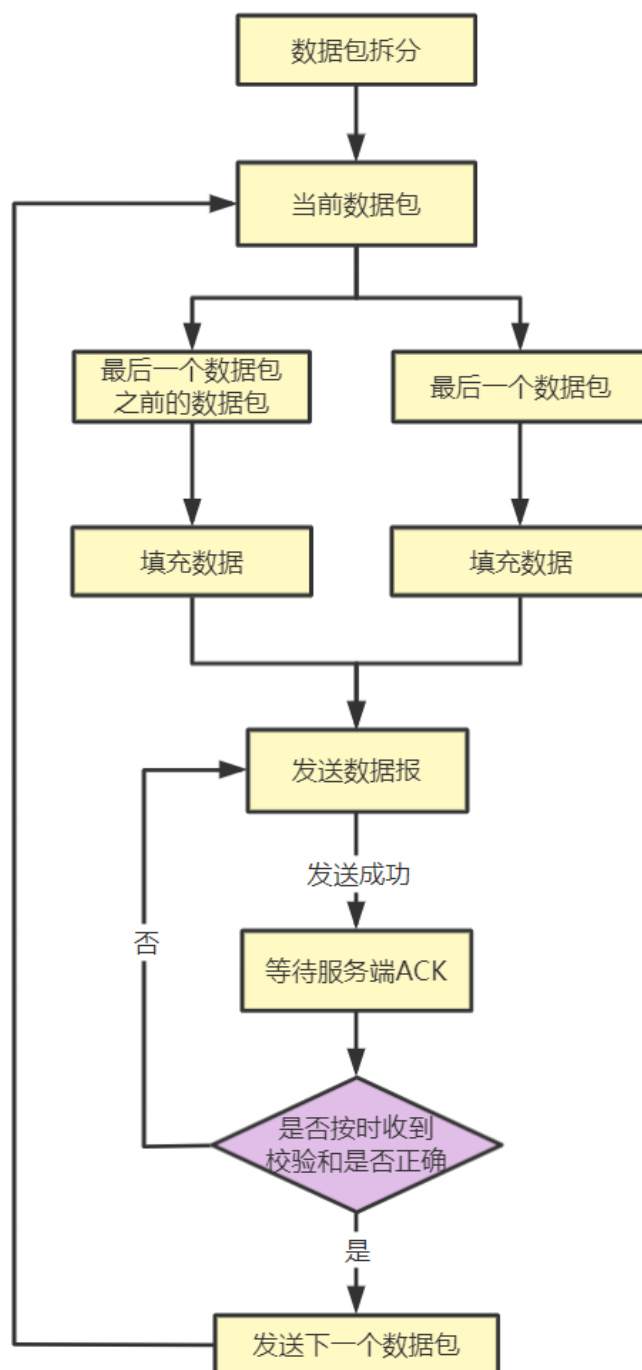
在使用校验和的过程中, 我们使用其对除了原本的UDP报头的内容进行校验.

停等机制

采用的是RDT3.0的停等机制实现的, 这里额外增加了区分是否为最后一个数据包的flag位(为了**防止RDT2出现的乱序传输的现象**), 如下所示为最后一个数据包与之前数据包的报文格式对比



如下图所示是按照停等机制设计的发送端发送数据报的流程示意图



代码如下所示:

```
//发送数据函数
void send_msg(char* message, int length) {
    //1. 计算发送数据包的数目
    int package_num = length / MAX_UDP_LEN + (length % MAX_UDP_LEN != 0);
    static int order = 0;
    //2. 一个一个发送
    int flag_last = 0;
    for (int i = 0; i < package_num; i++) {
```

C++

```

        if (i == package_num - 1) {
            flag_last = 1;
        }
        send_single_package(message + i * MAX_UDP_LEN, i == package_num -
1 ? length - (package_num - 1) * MAX_UDP_LEN : MAX_UDP_LEN, order++,
            flag_last);
        cout << "发送包的序列号: " << order << endl;
    }
}

```

// 发送单个数据包

// 1. 确认是否为最后一个数据包

// 2. 注意按顺序进行发送

// 3. 超时重传

```

bool send_single_package(char* message, int length, int order, int isLast
= 0) {
    char* sendBuf;
    int lenofsend;

    if (isLast) {
        //协议设计格式
        /*
        struct msgHead {
            unsigned char checknum;
            unsigned char flag;//标志位
            unsigned char seq; //表示数据包的序列号, 一个字节为周期
            int length; //表示数据包的长度 (包含头)
        };
        */
        sendBuf = new char[length + 4];
        sendBuf[1] = END;
        sendBuf[2] = order;
        sendBuf[3] = length;

        for (int i = 4; i < length + 4; i++)
            sendBuf[i] = message[i - 4];
        sendBuf[0] = check_sum(sendBuf + 1, length + 3);
        lenofsend = length + 4;
    }
    else {

```

```

        sendBuf = new char[length + 3];
        sendBuf[1] = NOT_END;
        sendBuf[2] = order;
        for (int i = 3; i < length + 3; i++)
            sendBuf[i] = message[i - 3];
        sendBuf[0] = check_sum(sendBuf + 1, length + 2);
        lenofsend = length + 3;
    }
    while (1) {
        sendto(m_ClientSocket, sendBuf, lenofsend, 0,
(sockaddr*)&m_ServerAddress, sizeof(m_ServerAddress));
        int begintime = clock();
        char recv[3];
        int lentmp = sizeof(m_ServerAddress);
        int fail_send = 0;
        //超时重传
        while (recvfrom(m_ClientSocket, recv, 3, 0,
(sockaddr*)&m_ServerAddress, &lentmp) == SOCKET_ERROR)
            if (clock() - begintime > TIMEOUT) {
                fail_send = 1;
                break;
            }
        if (fail_send == 0 && check_sum(recv, 3) == 0 && recv[1] == ACK
&& recv[2] == (char)order)
        {
            //cout << "发送端收到序列号: " << recv[2] << endl;
            return true;
        }
    }
}

```

服务端作为接收端的处理过程

```

void recv_message(char* message, int& len_recv) {
    char recv[MAX_UDP_LEN + 4];
    int lentmp = sizeof(m_ClientAddress);
    //接收端的顺序
    static unsigned char receive_order = 0;
    len_recv = 0;
    while (1) {

```

C++

```

        while (1) {
            memset(recv, 0, sizeof(recv));
            while (recvfrom(m_ServerSocket, recv, MAX_UDP_LEN + 4, 0,
(sockaddr*)&m_ClientAddress, &lentmp) == SOCKET_ERROR);
            char send[3];
            int flag = 0;
            //发送ACK确认的序列号
            if (check_sum(recv, MAX_UDP_LEN + 4) == 0 && (unsigned
char)recv[2] == receive_order) {
                receive_order++;
                flag = 1;
            }
            send[1] = ACK;
            send[2] = receive_order - 1;
            send[0] = check_sum(send + 1, 2);
            cout << "接收端确认序列号: " << receive_order - 1 << endl;
            sendto(m_ServerSocket, send, 3, 0,
(sockaddr*)&m_ClientAddress, sizeof(m_ClientAddress));
            if (flag)
                break;
        }
        // 接收端将消息放入缓存区中
        if (END == recv[1]) {
            for (int i = 4; i < recv[3] + 4; i++)
                message[len_recv++] = recv[i];
            break;
        }
        else {
            for (int i = 3; i < MAX_UDP_LEN + 3; i++)
                message[len_recv++] = recv[i];
        }
    }
}

```

确认重传机制

在上面的代码实现过程中我们已经多次涉及过确认重传机制(确认重传包括差错重传和超时重传), 超时重传的实现机制是发送数据报时建立一个时钟start开始时刻, 如果到时间还没有收到对应的

数据报则重新传输。

读写数据

这里采用的是分别传输文件名和文件内容的传输方式

```
//先发送文件名
send_msg((char*)(filename.c_str()), filename.length());
//然后发送具体的文件内容
send_msg(buffer, len);
```

C++

文件数据的传输过程是先将文件读入到一个大的缓冲区域中, 然后再进行传输

```
string filename;
while (1) {
    printf("输入要发送文件名: ");
    cin >> filename;
    ifstream fin(filename.c_str(), ifstream::binary);
    if (!fin) {
        printf("文件找不到\n");
        continue;
    }
    unsigned char t = fin.get();
    while (fin) {
        buffer[len++] = t;
        t = fin.get();
    }
    fin.close();
    break;
}
```

C++

文件的写入是先接受文件名称, 创建对应名称的文件, 然后接收文件内容, 将其中内容存放在缓存区当中, 写入对应的文件内容:

```
//先接收文件名称
recv_message(buffer, len);
buffer[len] = 0;
cout << buffer << endl;
```

C++

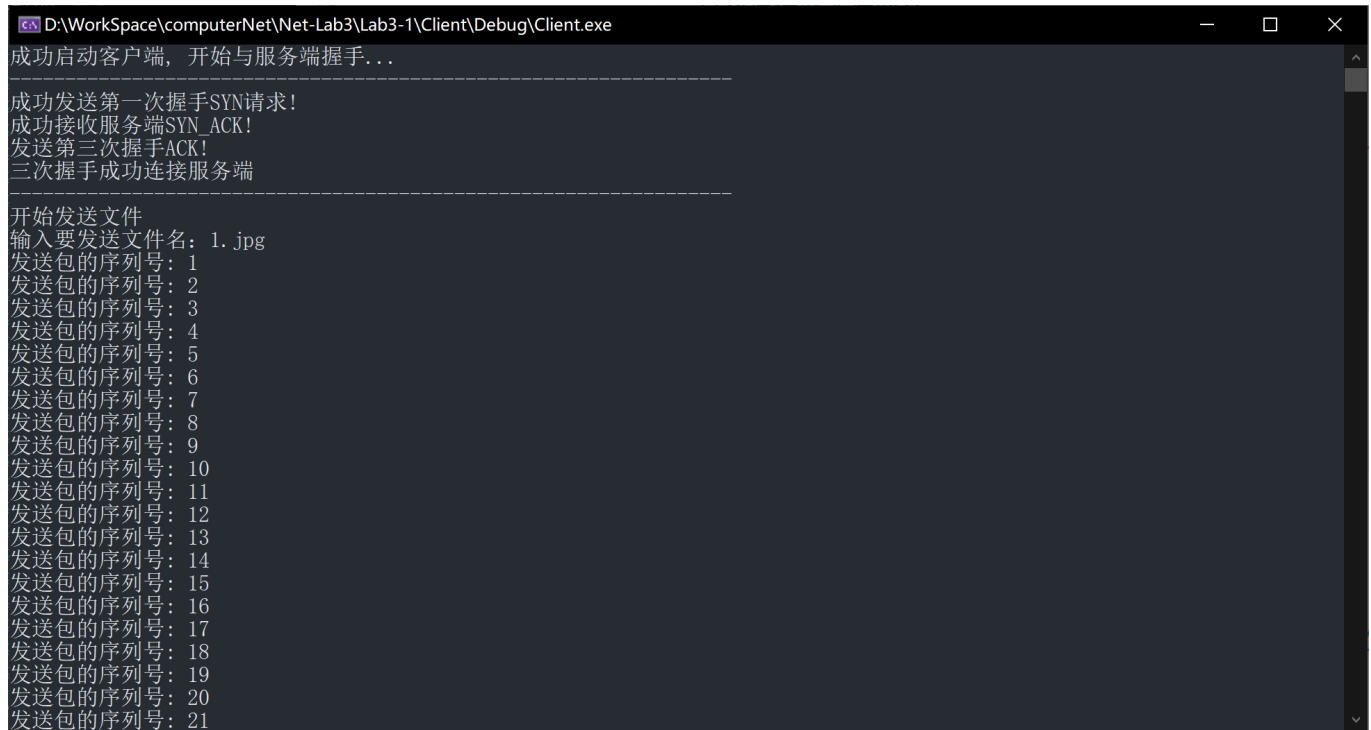
```
//然后接收文件内容
string file_name(buffer);
recv_message(buffer, len);
printf("信息接收成功\n");
ofstream fout(file_name.c_str(), ofstream::binary);
for (int i = 0; i < len; i++)
    fout << buffer[i];
fout.close();
```

程序运行界面

程序运行方式:

1. 先运行服务端,再运行客户端;
2. 将对应的素材放置于Client.exe所在文件夹中,在客户端运行输入对应文件,名称即可看到程序传输运行状态,结束后在对应的Server.exe所在文件夹中,可以看到对应传输成功的文件.

客户端界面



```
D:\Workspace\computerNet\Net-Lab3\Lab3-1\Client\Debug\Client.exe
成功启动客户端, 开始与服务端握手...
-----
成功发送第一次握手SYN请求!
成功接收服务端SYN_ACK!
发送第三次握手ACK!
三次握手成功连接服务端
-----
开始发送文件
输入要发送文件名: 1. jpg
发送包的序列号: 1
发送包的序列号: 2
发送包的序列号: 3
发送包的序列号: 4
发送包的序列号: 5
发送包的序列号: 6
发送包的序列号: 7
发送包的序列号: 8
发送包的序列号: 9
发送包的序列号: 10
发送包的序列号: 11
发送包的序列号: 12
发送包的序列号: 13
发送包的序列号: 14
发送包的序列号: 15
发送包的序列号: 16
发送包的序列号: 17
发送包的序列号: 18
发送包的序列号: 19
发送包的序列号: 20
发送包的序列号: 21
```

服务端界面

```
选择 D:\Workspace\computerNet\Net-Lab3\Lab3-1\Server\Debug\Server.exe
服务端已设置绑定占用的连接, 其ip: 127.0.0.1, port: 9000
服务端等待握手

-----
成功接收客户端SYN请求!
成功发送服务端SYN_ACK!
成功接收客户端ACK!
建立三次握手成功!

-----
开始接收文件
接收端确认序列号: 0
1.jpg
接收端确认序列号: 1
接收端确认序列号: 2
接收端确认序列号: 3
接收端确认序列号: 4
接收端确认序列号: 5
接收端确认序列号: 6
接收端确认序列号: 7
接收端确认序列号: 8
接收端确认序列号: 9
接收端确认序列号: 10
接收端确认序列号: 11
接收端确认序列号: 12
接收端确认序列号: 13
接收端确认序列号: 14
接收端确认序列号: 15
接收端确认序列号: 16
接收端确认序列号: 17
接收端确认序列号: 18
接收端确认序列号: 19
```

发送文件成功截图:

```
D:\Workspace\computerNet\Net-Lab3\Lab3-1\Client\Debug\Client.exe
发送包的序列号: 359
发送包的序列号: 360
发送包的序列号: 361
发送包的序列号: 362
发送包的序列号: 363
发送包的序列号: 364
发送包的序列号: 365
发送包的序列号: 366
发送包的序列号: 367
发送包的序列号: 368
发送包的序列号: 369
发送包的序列号: 370
发送包的序列号: 371
发送包的序列号: 372
发送包的序列号: 373
文件发送成功
文件传输时间 0 s
吞吐量: inf kbps

-----
开始断开连接
客户端发送挥手消息
接收第二次挥手
挥手成功, 断开连接
请按任意键继续. . .
```