南開大學
Nankai University

计算机学院

计算机网络实验报告

基于 UDP 服务设计可靠传输协议并编程实现-3.1

姓名：李欣

学号：2011165

专业：计算机科学与技术

2022 年 11 月 19 日

目录

Abstract

本次实验中，我们利用 UDP 数据报套接字在用户空间实现面向连接的可靠数据传输，实现了建立连接的**三次握手**和**四次挥手**，基于 **rdt3.0** 实现了**差错检测**和**确认重传**等功能，在流量控制方面采用了**停等机制**，完成了给定测试文件的**完整传输**。

在打印日志中，具体展现了三次握手，数据传输，四次挥手，吞吐率，传输时间等重要信息. 最后，成功完成了文件的完整传输。

1 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。

- 数据报套接字：UDP；
- 建立连接：实现类似 TCP 的握手、挥手功能；
- 差错检测：计算校验和；
- 确认重传：rdt2.0、rdt2.1、rdt2.2、rdt3.0 等，亦可自行设计协议；
- 单向传输：发送端、接收端；
- 有必要日志输出。

2 实验环境

VS2019
语言:C++
需要使用 router 路由器

3 设计与实现

3.1 数据包套接字: UDP

3.1.1 协议设计

- 长度: 包含头部、以字节计数
- 校验和: 为可选项, 用于差错检测

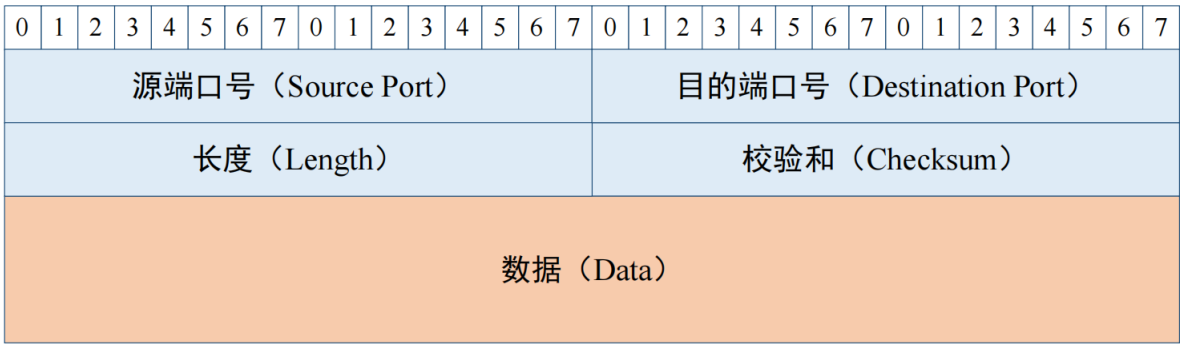

图 3.1: Caption

UDP 数据报格式如图所示, 有源端口号，目的端口号，长度，校验和和数据段。

头部:

在我们的协议中，我们的头部不仅包含源端口号，目的端口号，长度，校验和。仿照 TCP，我们**添加了发送序号 (seq)**，**确认序号 (ack)** 和**标志位 (flag)**。用于记录发送数据的信息。

伪首部:

其次，对于发送端与接收端自己产生的伪首部，我们认为伪首部包括了**源 IP** 和**目的 IP**，特别注意在**计算校验和**时需要将伪首部加入进行计算。

数据

收到正确的 ack 后，更新数据域。

3.1.2 程序设计

我们将头部封装成一个 struct 结构体 **header**，其中，每个成员的类型都是 u_short,u_short 是两字节的，即 16bit，方便校验和的计算。

在发送数据包时，将要发送的数据存入发送缓冲区 sendbuff 中，在不同的情况下，sendbuff 的大小不同。

在三次握手、四次挥手与接收端发送回来的 ack 数据包的情况下，发送缓冲区的大小均为头部大小。

在发送端发送文件数据时,开辟发送缓冲区的大小为头部大小 header_size+ 最大带宽 maxsize_data(1024 字节), 除了最后一个数据需要需要特殊处理外，每次发送的数据都为设置的网络最大带宽为 maxsize_data 大小。

3.1.3 代码实现

```
1  struct header {
2      // u_short 2 字节 (2B) 对应 16 位 (16 bit) 无符号比特
3      u_short ack;            //0 或 1
4      u_short seq;            //0 或 1
5      u_short flag;            //bigend 0 位 SYN, 1 位 ACK, 2 位 FIN
6      u_short source_port;    //源端口
7      u_short dest_port;       //目的端口
8      u_short length;           //消息长度
```

```
    u_short checksum;        //校验和

    header()
    {
        source_port = SourcePort;
        dest_port = DestPort;
        ack = seq = flag = length = checksum = 0;
    }
};
```

```
    //准备数据包
    //将发送缓冲区全部填充为 0
    memset(sendbuff, 0, header_size + maxsize_data);
    //处理数据头
    e_header.flag = 0;
    e_header.ack = 0;
    e_header.seq = current_seq;
    //如果时最后一个数据包
    if (countpacknum == packnum)
    {
        e_header.length = thislen = len_of_wholeMSG % maxsize_data;//最后一个数据包大小
        e_header.flag = OVER;//结束标志
    }
    else
        e_header.length = thislen = maxsize_data;//最大数据包
    //将数据头放入发送缓冲区
    memcpy(sendbuff, &e_header, header_size);
    //将数据填入发送缓冲区
    int len = countpacknum * maxsize_data;
    memcpy(sendbuff + header_size, message + len, thislen);
    e_header.checksum = 0;
    memcpy(sendbuff, &e_header, header_size);
    //计算校验和
    e_header.checksum = checksum((u_short*)sendbuff, header_size + maxsize_data);

    //重新填充头部
    memcpy(sendbuff, &e_header, header_size);
```

3.2 三次握手

3.2.1 协议设计

本次实验中，实现了三次握手建立连接，其协议如下图所示

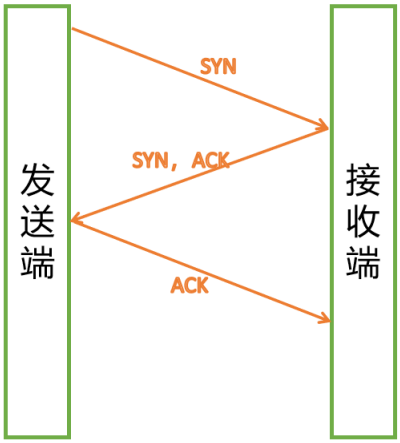

图 3.2: 三次握手

(1) 首先，由发送端将头部的 flag 位设置位 SYN，即，请求接收端与之建立连接。、

若这期间，发送端超过最大等待时长没有收到接收端的 (SYN,ACK) 回复，则再次发送 SYN 连接请求。若连续发送 SYN 请求超过 2 秒没有收到 (SYN,ACK) 回复，则说明服务器并未打开或者出现了故障，此时不再发送 SYN 连接请求，而是主动关闭程序。

此时，接收端处于阻塞通信状态，若收到 SYN 请求，则回复 (SYN,ACK) 数据包，即第二次握手请求，若没有收到信息或校验和出现错误，则继续进入等待状态。

(2) 若发送端成功收到 (SYN,ACK) 回复，则说明第一次和第二次发送握手请求都成功了。

(3) 此时发送端发送第三次握手请求 ACK，此处有两种方法让发送端确认接收端成功接收到了第三次握手 ACK。

第一种方法是等待一定时长没有再次收到 (SYN,ACK) 数据包，即，接收端在等待时长内收到了 ACK，故不会认为 (SYN,ACK) 丢失，所以不用重发 (SYN,ACK) 数据包，因此，发送端不会收到任何数据包。

第二种方法是基于我们的实验假定，接收端到发送端发送的包是不会丢失的，所以，只要在接收端收到第三次握手 (ACK) 时，再向发送端发一个确认收到的包就可以了。

这两种方法看似都可行，但在实际运用中，并不能保证接收端到发送端的传输一定是可靠的，故选取了第一种方式确定第三次握手 ACK 发送成功

3.2.2 程序设计

在编程实现中，我们设置了 SYN,SYN_ACK 和 ACK 三个状态，用来确定 flag 的值，当接收端或发送端收到这个包时，只要将**期待的 flag** 和收到的 flag 进行比较，若相等，则进行下一步握手请求，若不相等，针对发送端，重新发送数据，针对接收端，继续等待。当然，我们也不能忘记**检验校验和**。

3.2.3 代码实现

发送端:

```
//建立三次握手连接
const u_short SYN = 0x1;
const u_short ACK = 0x2;
const u_short SYN_ACK = 0x3;
const int max_waitingtime = 0.01*CLOCKS_PER_SEC;
u_long non_block = 1;
u_long block = 0;

```

由于代码过于庞大，这里仅展示关键代码部分，且第一次握手请求和第三次握手的代码实现又非常相似，这里仅展示第一次握手和第二次握手请求，主要是超时重传和差错检测，省略了第三次握手请求以及部分的数据头准备。

```
// 建立连接
void establish_connection()
{
    //
        //这里是数据头准备
    //

    bool over_time_flag = false;
    //第一次和第二次握手
    while (true)
    {
        over_time_flag = false;//未超时
        //第一次握手 发送 SYN
        result = sendto(s, sendbuff, header_size, 0, (sockaddr*)&router_addr, routerlen);
        if (result == -1)
        {
            cout << "[握手请求]:   [1] 发送数据包 失败" << endl;
            exit(-1);//退出程序
        }
        else
            cout << "[握手请求]:   [1] 发送数据包 成功" << endl;

        ioctlsocket(s, FIONBIO, &non_block); //设置非阻塞
        clock_t start; //开始计时器
        clock_t end;   //结束计时器
        start = clock();
        //连接正常关闭返回 0, 错误返回-1
        while (recvfrom(s, recbuff, header_size, 0, (sockaddr*)&router_addr, &routerlen) <= 0)
```

```
29          {
30              // 判断超时与否
31              end = clock();
32              if ((end - start) > max_waitingtime)
33              {
34                  over_time_flag = true; //超时设置 flag 为 1
35                  break;
36              }
37          }
38
39          //超时重传
40          if (over_time_flag)
41          {
42              cout << "[握手请求]:   [1] 发送数据包 失败 具体原因: 反馈超时, 重新发送" << endl;
43              continue;
44          }
45
46          //未超时接收, 判断消息是否正确
47          memcpy(&e_header, recbuff, header_size);
48          cout << "[ACK]:         [2] " << e_header.ack << endl;
49          cout << "[计算校验和]: [2] " << checksum((u_short*)&e_header, header_size) << endl;
50          if ((e_header.flag == SYN_ACK) && checksum((u_short*)&e_header, header_size) == 0)
51          {
52              cout << "[握手请求]:   [2] 接收数据包 成功" << endl;
53              break;//接收到正确的信息并跳出循环
54          }
55          else
56          {
57              cout << "[握手请求]:   [2] 接收数据包 失败 具体原因: 收到错误的数据包, 重新发送" << e
58              continue;//重传
59          }
60      }
61
62      //
63          //这里是第三次握手请求
64      //
65
66      cout << "[握手请求]:   [3] 发送数据包 成功" << endl;
67      cout << "\n[建立连接]: Congratulations! 三次握手成功! " << endl;
68      cout << "[建立连接]: 客户端与服务端成功建立连接! \n" << endl;
69  }
70
```

接收端:

```
// 建立连接
void establish_connection()
{
    cout << "[系统]: 服务端开启，等待连接客户端\n" << endl;
    header e_header;
    int header_size = sizeof(e_header);
    char* sendbuff = new char[header_size];
    char* recbuff = new char[header_size];
    int result = 0;
    bool over_time_flag = false;

    while (true)
    {
        //阻塞接收消息
        result = recvfrom(s, recbuff, header_size, 0, (sockaddr*)&router_addr, &routerlen);
        if (result == -1)
        {
            cout << "[握手请求]:   [1] 第一次握手接收失败" << endl;
            exit(-1);
        }
        memcpy(&e_header, recbuff, header_size);
        cout << "[计算校验和]: [1] " << checksum((u_short*)&e_header, header_size) << endl;
        if (checksum((u_short*)&e_header, header_size) == 0 && e_header.flag == SYN)
        {
            cout << "[握手请求]:   [1] 成功接收第一次握手请求" << endl;
            break;
        }
        else
        {
            cout << "[握手请求]:   [1] 失败，请求数据包错误" << endl;
        }
    }

    //
        //省略了部分代码 数据头准备，第二、三次握手请求
    //

    //cout << "[握手请求]: [3] 成功 第三次握手成功" << endl;
    cout << "\n[建立连接]: Congratulations!" << endl;
    cout << "[建立连接]: 服务端与客户端成功建立连接! \n" << endl;
```

```
41 }
42
```

3.3 四次挥手

3.3.1 协议设计

四次挥手协议如下图所示：

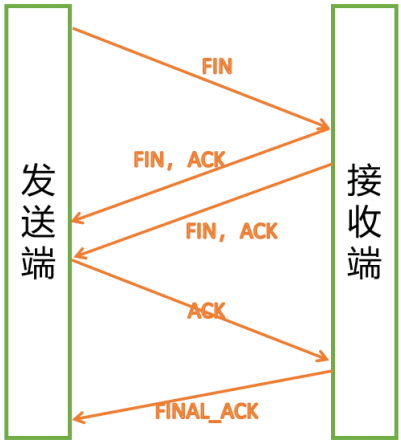

图 3.3: 四次挥手

在四次挥手中，本实验假定的是由发送端最开始请求关闭连接。

(1) 首先，由发送端发送 FIN 数据包请求关闭连接。若超过最长等待时间没有收到接收端送来的 (FIN,ACK), 则说明 FIN 数据包丢失了，那么再发送，再次等待，知道收到 (FIN,ACK) 数据包。

(2) 若收到接收端发来的 (FIN,ACK), 则继续等待下一个 (FIN,ACK) (这里为什么需要两个 (FIN,ACK), 我们在前面的实验中其实已经涉及过了，有时候虽然单向的连接关闭，但反向的数据传输可能还没有结束等原因，在此不做赘述。)

(3) 接收到第二个 (FIN,ACK)

(4) 继续向接收端发送一个确认 ACK，由于发送端到接收端可能存在丢包的情况，所以这里和握手请求的最后一个 ACK 类似，要么等待一定时间，要么由可靠的接收端向服务器发送最后一个 FINAL_CHECK，这里采取了后者.

(5) 发送端接收到了 FINAL_CHEAK，此时就可以安心的关闭了。

3.3.2 程序设计

在编程实现中，我们设置了 FIN,FIN_ACK,ACK 和 FINAL_CHEAK 三个状态，用来确定 flag 的值，当接收端或发送端收到这个包时，只要将**期待的 flag** 和收到的 flag 进行比较，若相等，则进行下一步握手请求，若不相等，针对发送端，重新发送数据，针对接收端，继续等待。当然，我们也不能忘记**检验校验和**。

3.3.3 代码实现

三次握手和四次挥手的程序设计高度相似，由于在三次握手中给出了详细的代码，这里仅贴出关键代码，具体代码也可查看接收端和发送端的 cpp 代码。

```
const u_short FIN = 0x4;//OVER=0,FIN=1,ACK=0,SYN=0
const u_short FIN_ACK = 0x6;//OVER=0,FIN=1,ACK=1,SYN=0
const u_short FINAL_CHECK = 0x20;//FC=1,OVER=0,FIN=0,ACK=0,SYN=0
```

3.4 计算校验和

3.4.1 协议设计

检验和 (checksum)，在数据处理和数据通信领域中，用于校验目的地一组数据项的和。它通常是以十六进制为数制表示的形式。在我们的实验中，如果校验和的数值超过十六进制的 FFFF，就要求其补码作为校验和。通常用来在通信中，尤其是远距离通信中保证数据的完整性和准确性。

我们这里设置的对于发送的数据包 (头部部分 + 数据部分) 进行校验和计算，在计算校验和前，要将头部的校验和域段清零，然后计算校验和后重新填充该域段。

在计算校验和前，通过在最后补 0 将计算对象填充为 16bit 对齐的数据，校验和每 16bit 进行加和运算，若有进位，则取结果的低 16 位与进位进行加和。

需要注意的是，我们要自己创建伪首部加入到校验和的计算中去，所以在实际实现中，需要注意这一点。

所有的数据累加完毕后，取结果低 16 位并对其求反码作为结果。

3.4.2 程序设计

将上述结果实现即可，需要注意的是补零的步骤，我认为，其实很好的设置传入数据的大小，即发送缓冲区的大小 (因为在最初就要将发送缓冲区全填 0)，就不用在校验和计算段做更多的操作。

3.4.3 代码实现

发送端:

```
u_short checksum(u_short* mes, int size)
{
    // 分配所需的内存空间，返回一个指向它的指针，size+1 为内存块的大小，以字节为单位
    size_t size_need;
    if (size % 2 == 0)//2B 16it 的倍数
        size_need = size;
    else
        size_need = size + 1;
    int count = size_need / 2; //size 向上取整
    u_short* buf = (u_short*)malloc(size_need);//开辟的空间
    memset(buf, 0, size_need); //开的 buff 将数据报用 0 补齐为 16 的整数倍
    memcpy(buf, mes, size);       //发送的 buf 不要发送补的 0
    u_long sum = 0;
    //按二进制反码运算求和
    while (count--)
```

```
    {
        if (count == 0)
            sum += *buf;
        else
            sum += *buf++;
        if (sum & 0xffff0000)//实际就是看有没有进位
        {
            sum &= 0xffff;//取低 16 位
            sum++;//进位加到最后
        }
    }
    //
        //省略了部分代码，对伪首部也要进行运算求和!
    //
    //得到的结果求反码得到校验和
    return ~(sum & 0xffff);
}
```

接收端可能在计算校验核时出现校验和错误或接收到的 seq 号与期待的 seq 号不相同的情况，这个时候只要继续等待就可以。(发送端会超时重传)

3.5 确认重传: rdt3.0

3.5.1 协议设计

由于我们采用的是 rdt3.0 停等机制，发送端在收到前一个消息的 ack 后才会发送下一个数据包，如:

发送端: 发送端发送的 seq0, 若收到 ack0，则发送 seq1，注意要修改头部的 seq 和数据区的内容，更新校验和。

接收端: 接收发送端发来的数据，首先判断是不是期待的 seq，并计算校验和是否为 0，若两者任何一个出现了错误，那就什么也不做继续等待下一个数据包。(发送端会超时重传)。

3.5.2 程序设计

将上述思想实现。

3.5.3 代码实现

仅展示关键代码发送端确认重传

```
    while (true)
    {
        result = sendto(s, sendbuff, header_size + thislen, 0, (sockaddr*)&router_addr, router
        if (result == -1)
```

```
        {
            cout << "[传输数据包]: [" << countpacknum % 2 << "] 号 发送失败" << endl;
            exit(-1);
        }
        cout << "[传输数据包]: [" << countpacknum % 2 << "] 号 发送成功" << endl;

        //超时未收到回复，进行重传
        bool flag = false;
        start = clock();
        while (recvfrom(s, recbuff, header_size, 0, (sockaddr*)&router_addr, &routerlen) <= 0)
        {
            end = clock();
            if (end - start > max_waitingtime)
            {
                flag = true;
                break;
            }
        }
        if (flag)
        {
            cout << "[传输数据包]: [" << countpacknum % 2 << "] 号 反馈超时，重新发送" << endl;
            continue;
        }

        //收到回复，判断回复的对不对
        //s 省略的代码
    }
```

3.6 传输过程

3.6.1 发送端: 文件加载与发送

加载文件的关键代码:

```
ifstream fin(file_name.c_str(), ios::in | ios::binary);
if (!fin.is_open())
{
    cout << "[文件打开]: 失败! " << endl;
    exit(-1);
}
......
while (fin)
{
```

```
        message[len_of_wholeMSG++] = temp;
        temp = fin.get();
    }
    fin.close();
```

发送数据与接收 ack 的关键代码:

```
    while (true)
    {
            //准备数据包
            //发送数据包 Seq0/1
            //等待 ack
            //收到回复，判断回复的对不对
            memcpy(&e_header, recbuff, header_size);
            if (e_header.ack == current_seq && checksum((u_short*)&e_header, header_size) == 0)
            {
                cout << "[接收 ACK]:    " << e_header.ack << endl;
                if (e_header.flag == OVER_ACK)
                    cout << "[接收 ACK]:    收到结束 OVER_ACK" << endl;
                break;
            }
            else
            {
                cout << "[传输数据包]: [" << countpacknum % 2 << "] 号 收到错误数据包" << endl;
                continue;
            }
        }
        //更新下一个要发送的 seq 号
        current_seq = (current_seq + 1) % 2;//下一个要传输的 seq
    }
```

3.6.2 接收端: 文件下载与保存

```
nst int maxsize_total = 100000000;//文件总的字节数
nst int maxsize_data = 1024;          //贷款
nst unsigned char OVER = 0x8;//OVER=1,FIN=0,ACK=0,SYN=0
nst unsigned char OVER_ACK = 0xA;//OVER=1,FIN=0,ACK=1,SYN=0
ar* message = new char[maxsize_total];//存储所有收到的数据
```

接收数据:

```
    void receive_MSG()
```

```
{
    //一些准备工作
    ioctlsocket(s, FIONBIO, &block);//阻塞接收
    while (true)
    {
        //等待接收数据包
        while (recvfrom(s, recbuff, header_size + maxsize_data, 0, (sockaddr*)&router_addr, &r
            cout << "[接收数据]:   连接有误，继续等待" << endl;
        }
        //接收到了数据包，将数据包头部存入 e_header
        memcpy(&e_header, recbuff, header_size);
        //判断数据包是否正确，
        if (e_header.seq == hope_seq && checksum((u_short*)recbuff, header_size + e_header.leng
        {
            //读入数据
            //...........
            //注意是否为最后一个数据
            if (e_header.flag == OVER_ACK)
            {
                cout << "[接收数据]:   成功接收文件!" << endl;
                break;
            }
        }
        //不正确，重新等待
        else{
        cout << "[接收数据]:   接收到了错误的数据包" << endl;
        continue;
        }
    }
    //保存文件
    savefile(a,offset);
}
```

保存文件:

```
void savefile(string name,int len)
{
    string filename = name;
    ofstream fout(filename.c_str(), ofstream::binary);
    for (int i = 0; i <len; i++)
    {
        fout << message[i];
```

```
        }
        fout.close();
        cout << "[文件保存]:    文件已成功保存到本地\n" << endl;
    }
```

3.7 日志输出

将传输时间，文件大小，数据头大小总和核平均吞吐率进行输出

3.7.1 代码实现

```

            cout << "[传输数据]:    文件传输完毕! \n" << endl;
            cout << "[传输时间]:        " << time_record << " s" << endl;
            cout << "[文件大小]:        " << len_of_wholeMSG * 8 << " bit" << endl;
            cout << "[数据头大小总和]: " << header_size * packnum * 8 << " bit" << endl;
            double throu = (double)(len_of_wholeMSG + packnum * header_size) * 8 / time_record;
            cout << "[平均吞吐率]:      " << throu << "bit/s" << endl;
```

4 结果展示

4.1 三次握手

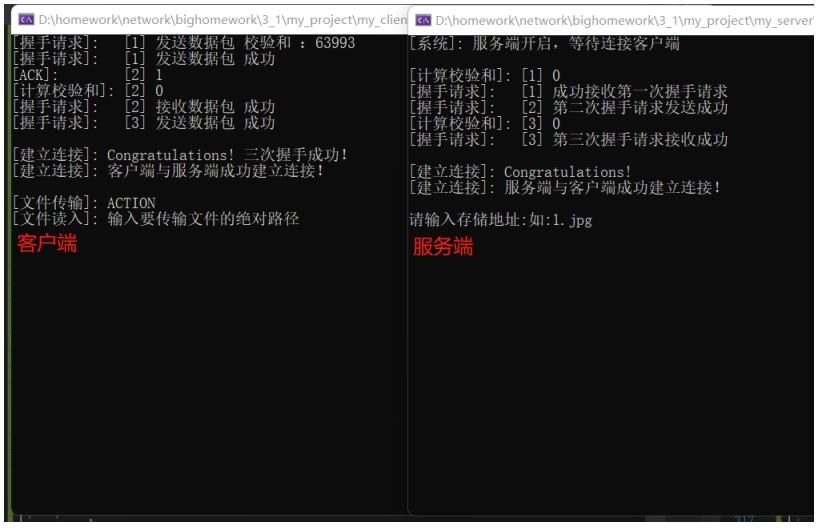

图 4.4: 三次握手

如图所示为三次握手的情况

4.2 超时重传

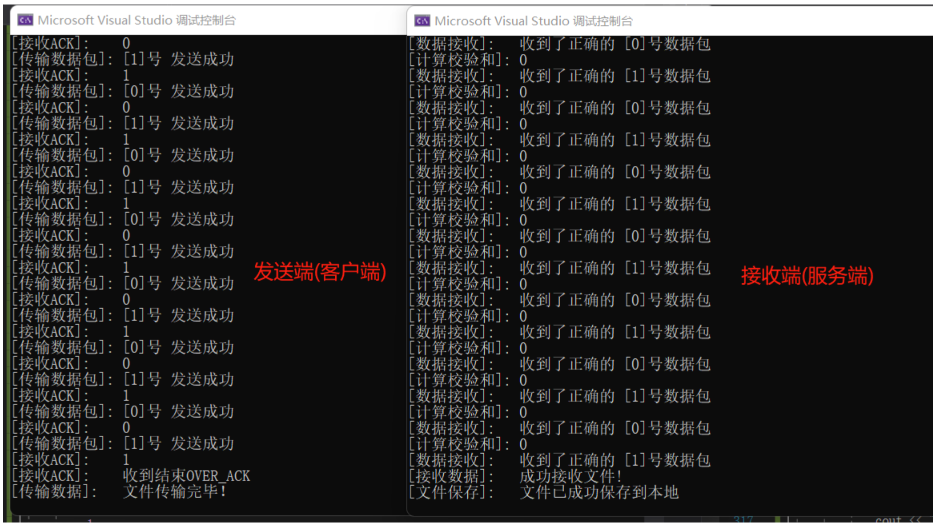

图 4.5: 未超时

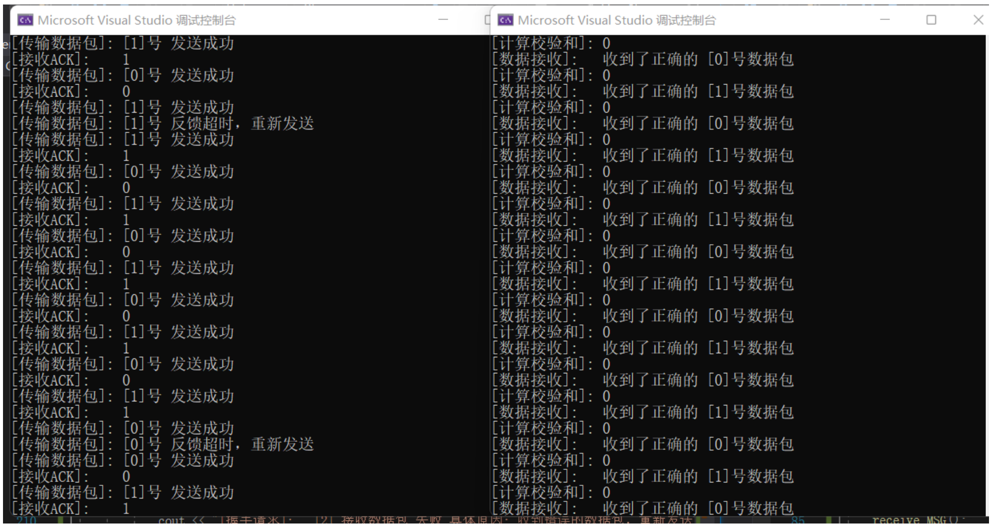

图 4.6: 超时重传: 丢包率 10%

遇到超时重传的情况，会进行相应的日志的打印，在 10% 的丢包率的情况下，能够正确进行超时重传。

4.3 数据传输：ACK,Seq

数据传输部分的 ACK 和 seq 一一对应 (0/0,1/1)，可以查看超时重传部分的图片查看结果。

4.4 传输时间，总吞吐量，平均吞吐率

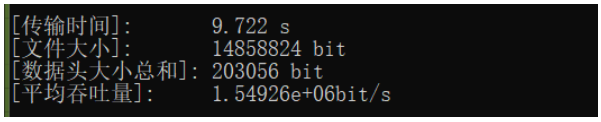

图 4.7: 传输时间，总吞吐量，平均吞吐率

如图所示，本次实验传输 1.jpg 图片的传输时间，总吞吐量核平均吞吐率均被打印。

4.5 四次挥手

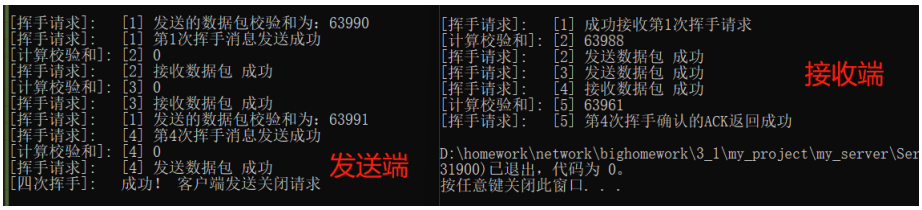

图 4.8: 四次挥手

如图所示为四次挥手，将四次挥手的过程展示。

4.6 图片传输

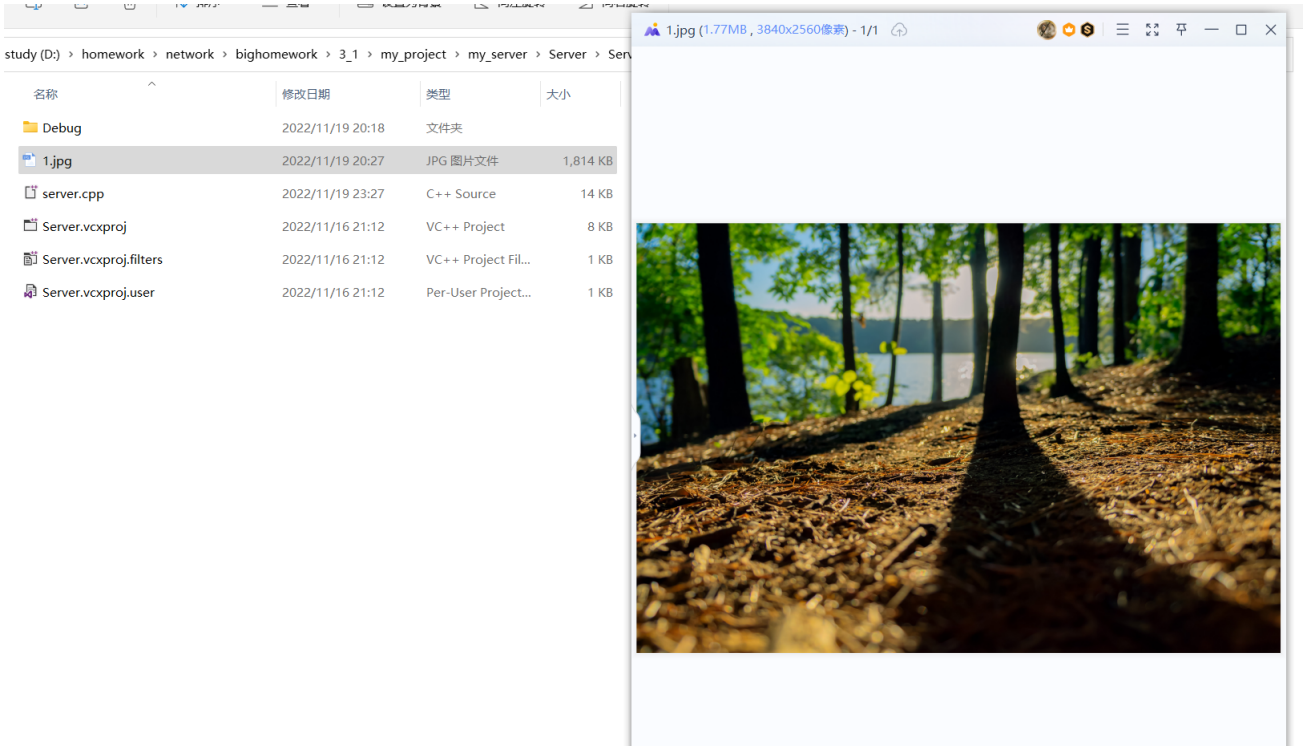

图 4.9: 图片传输

如图所示为最终传输图片的结果，可以看到，图片被完整的传输了。