



南开大学  
Nankai University

南 开 大 学

计 算 机 学 院

计算机网络实验报告

---

## UDP 可靠传输协议——part1 停等实现

---

姓名：姚知言

年级：2022 级

专业：计算机科学与技术

指导教师：张建忠 徐敬东

2024 年 11 月 30 日

## 摘要

本次实验借助 UDP 协议, 设计了报文类, 参考 rdt3.0 的协议思想, 设计了三次握手, 四次挥手和可靠数据传输协议。利用 router 对协议进行性能测试, 并进行分析。

**关键词:** UDP, 可靠数据传输, rdt3.0, router, 超时和校验和处理

## 目录

一、 实验要求	1
二、 实验环境	1
三、 实验设计	2
(一) 报文设计	2
(二) 三次握手	2
(三) 数据传输	2
1. 客户端核心逻辑	3
2. 服务器端核心逻辑	3
3. 超时重传和校验	3
(四) 四次挥手	3
四、 编程实现	4
(一) IP 及端口定义	4
(二) 报文类的实现 (message.h)	4
(三) 报文类内函数的定义 (message.cpp)	5
1. 构造函数	5
2. calchecksum 函数	5
3. storechecksum 和 verifychecksum 函数	6
4. printdetails 函数	6
5. prepare 函数	6
(四) 服务器端 (server.cpp) 和客户端 (client.cpp) 的核心流程	7
(五) 初始化函数 server_init/client_init	8
(六) 三次握手实现	9
1. 客户端 client_shake 函数	9
2. 服务器端 server_shake 函数	11
(七) 数据传输实现	12
1. 客户端 client_sendfile 函数	12
2. 服务器端 server_recvfile 函数	16
(八) 四次挥手实现	19
1. 客户端 client_wave 函数	19
2. 服务器端 server_wave 函数	21

<b>五、 性能测试</b>	<b>22</b>
(一) 测试环境配置 . . . . .	22
(二) 正确性测试 . . . . .	23
(三) 性能分析 . . . . .	25
<b>六、 总结</b>	<b>25</b>

## 一、实验要求

1. 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、接收确认、超时重传等。流量控制采用停等机制，完成给定测试文件的传输。
2. 数据报套接字：UDP
3. 协议设计：数据包格式，发送端和接收端交互，详细完整
4. 建立连接、断开连接：类似 TCP 的握手、挥手功能
5. 差错检验：校验和
6. 接收确认、超时重传：rdt2.0、rdt2.1、rdt2.2、rdt3.0 等，亦可自行设计协议
7. 单向传输：发送端、接收端
8. 日志输出：收到/发送数据包的序号、ACK、校验和等，传输时间与吞吐量
9. 测试文件：必须使用助教发的测试文件（1.jpg、2.jpg、3.jpg、helloworld.txt）

## 二、实验环境

本次实验在 x86-64 架构物理机中进行。通过 VS Code 完成实验代码的编写，通过 G++ 编译器完成源代码的编译。本实验的程序对环境没有严苛的要求，理论上来说现代 x86 架构都可以成功运行。

文件结构如图1所示。

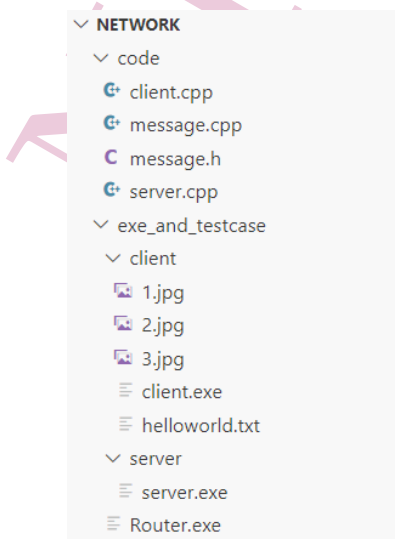


图 1: 文件结构

此处为测试准备好了样例，当测试通过时，收到的文件应出现在 server 目录下。

## 三、 实验设计

本实验的主体协议设计参考了上一实验的 TCP 抓包结果。总体来说，无论是客户端还是服务器端，SEQ 的更新逻辑都是上一报文的 SEQ+ 上一报文的 LEN，ACK 的更新逻辑是上一次接收到的正确报文的 SEQ+LEN。

对于 SYN 或 FIN 置位的报文来说，情况略有不同。为体现出三次握手，四次挥手的过程，保证实验结果的可靠性（现实中也是这么实现的），这两类报文的 LEN 虽然是 0，但对应的 SEQ/ACK 需要自增 1。（也可以理解为，这两类报文的 LEN 可以视为 1）。

### （一） 报文设计

报文设计需要包含以下元素：（由于本次实验为回环传输，没有设计 IP 地址元素）

- 源端口 & 目的端口
- SEQ & ACK
- 标志位 FLAGS
- 报文体长度
- 校验和
- 报文体内容

### （二） 三次握手

三次握手的实现如下：

注：在握手过程中，客户端生成随机数 R1，服务器端生成随机数 R2。

1. 第一次握手：

客户端-> 服务器端：SEQ=R1, ACK=0, FLAGS=SYN。

2. 第二次握手：

服务器端-> 客户端：SEQ=R2, ACK=R1+1, FLAGS=SYN | ACK。

3. 第三次握手：

客户端-> 服务器端：SEQ=R1+1, ACK=R2+1, FLAGS=ACK。

随后，客户端在发送第三次握手后建立连接，服务器端在接收第三次握手后建立连接。

在三次握手实现中，同样增加了超时重传机制，保证可靠性。

### （三） 数据传输

在这一部分中，报文的 SEQ/ACK 设计遵循本节开头的原则。（由于单向传输，客户端的 ACK 和服务端端的 SEQ 并没有真正发生改变过，但客户端 SEQ 和服务端 ACK 的对应关系已经足以进行报文的区分。）对于标记位置位，文件发送不进行 ACK 置位，文件接收进行 ACK 置位。

### 1. 客户端核心逻辑

对于客户端，需要将要发送的文件分包发送。

首先要发送一个包括文件名和文件长度的包，以便服务器端开辟合适的接收缓存。

随后每次尽可能将报文体填满，即除最后一个包外，每个包的大小都应该是报文体的最大大小，最后一个包发送剩余的所有内容。

此外，客户端还需要在发送前设计文件读取。即将文件全部内容读取到 buffer 中，以便后续发送。

### 2. 服务器端核心逻辑

对于服务器端，需要接收客户端发送的内容，并返回 ACK。

对于文件名和文件长度的包，进行合理的拆分，分别保存文件名和文件长度，并动态分配 buffer 大小。

在后续包的接收时，把客户端发送的内容保存到 buffer 中。

在客户端发送完毕后，将 buffer 中的内容写入文件。

### 3. 超时重传和校验

在停等机制下，无论是超时重传，还是校验错误，都可以通过重传上一个包来解决。

因为该情况的原因，要么是发过去的包丢了或者错了，要么是对方收到后，返回的包丢了或者错了。

在第一种情况下，对上一个包的重传可以很显然的解决问题，在第二种情况下，通过重传上一个包可以提示对方传输失败，使得对方进行上一个包的重传，从而解决问题。

## (四) 四次挥手

四次挥手的实现如下：

注：下文中的 L1 表示在数据传输阶段，客户端发送的最后一个报文的 SEQ+LEN；L2 表示在数据传输阶段，服务器端发送的最后一个报文的 SEQ+LEN。

#### 1. 第一次挥手：

客户端-> 服务器端：SEQ=L1, ACK=L2, FLAGS=FIN | ACK。

#### 2. 第二次挥手：

服务器端-> 客户端：SEQ=L2, ACK=L1+1, FLAGS=ACK。

#### 3. 第三次挥手：

服务器端-> 客户端：SEQ=L2, ACK=L1+1, FLAGS=FIN | ACK。

#### 4. 第四次握手：

客户端-> 服务器端：SEQ=L1+1, ACK=L2+1, FLAGS=ACK。

随后，客户端在第四次挥手发送后等待 2RTT 断开连接，服务器端在收到第四次挥手后断开连接。

在四次挥手实现中，同样增加了超时重传机制，保证可靠性。

## 四、 编程实现

### (一) IP 及端口定义

1. 服务器端 port: 2222
2. Router 端 port: 3333
3. 客户端 port: 4444

虽然服务器端设置了使用全部 IP 地址,但在后续测试中都使用 127.0.0.1, Router 和客户端则是确定了 IP 地址为 127.0.0.1。

### (二) 报文类的实现 (message.h)

在 message.h 中,我进行了报文类的实现。

在文件中,我定义了最长报文内容长度为 14000,由于 Router 限制报文长度不能超过 15000。还定义了最大等待时间和最多发送次数,以便后续停等机制实现。

在报文类设计中,我设计的类对象包括:

- 源端口 (srcport): 2 字节无符号整数
- 目的端口 (dstport): 2 字节无符号整数
- seq: 4 字节无符号整数
- ack: 4 字节无符号整数
- flags: 2 字节无符号整数,其中第 0 位表示 ACK,第 1 位表示 SYN,第 2 位表示 FIN,其余位保留,留待后续扩展
- 报文长度 (len): 4 字节无符号整数
- 校验和 (checksum): 2 字节无符号整数
- 报文内容 (data): maxlen (14000) 长度的 1 字节无符号字符数组

此外,我还设计了部分类内函数,用于计算和检查校验和,打印日志信息,将在下一部分中展开介绍。

以下是具体实现代码:

message.h

```
1 #include <iostream>
2 #include <windows.h>
3 using namespace std;
4
5 const unsigned int maxlen=14000;//不超过15000字节
6 const int maxwait=2000;//最大等待时间, 2s
7 const int maxsend=10;//最多发送次数, 10
8 #pragma pack(1)
9 class message{
10     public:
```

```

11     u_short  srcport; //源端口
12     u_short  dstport; //目的端口
13     u_long   seq;
14     u_long   ack;
15     u_short  flags; // [0]:ACK, [1]:SYN, [2]:FIN
16     u_long   len; //报文长度
17     u_short  checksum; //校验和
18     BYTE data[maxlen];
19     message();
20     u_short  calchecksum();
21     void storechecksum();
22     void prepare();
23     bool verifychecksum();
24     void printdetails();
25 };
26 #pragma pack()

```

### (三) 报文类内函数的定义 (message.cpp)

主要的实现包括：

#### 1. 构造函数

本质上就是清空报文中全部内容。

message 构造函数

```

1 message::message() {
2     srcport=0;
3     dstport=0;
4     seq=0;
5     ack=0;
6     flags=0;
7     len=0;
8     checksum=0;
9     memset(&data, 0, sizeof(data)); //clear all
10 }

```

#### 2. calchecksum 函数

用于计算校验和，被后续存储校验和/验证校验和函数调用。

calchecksum 函数

```

1 u_short message::calchecksum() {
2     u_long nowlen=len; //存储长度
3     u_long nowsum=0;
4     BYTE* ptr = data;
5     while(nowlen--){
6         nowsum+=*(ptr++);

```



```

7         if (nowsum & 0xffff0000) {
8             nowsum &= 0xffff;
9             nowsum++;
10        }
11    }
12    return ~(nowsum & 0xffff);
13 }

```

### 3. storechecksum 和 verifychecksum 函数

分别在构建报文时用于将计算的校验和存储在校验和对象中，以及在收到报文时用于验证校验和。调用了 calchecksum 函数。

#### storechecksum 和 verifychecksum 函数

```

1 void message::storechecksum() {
2     checksum=calchecksum();
3 }
4 bool message::verifychecksum() {
5     return (checksum==calchecksum());
6 }

```

### 4. printdetails 函数

用于在日志中输出除报文体内容之外的全部报文信息。

#### printdetails 函数

```

1 void message::printdetails() {
2     cout<<" 报文详细信息如下:\n";
3     cout<<" 源端口:"<<srcport<<" 目的端口:"<<dstport<<"\n";
4     cout<<"SEQ:"<<seq<<" ACK:"<<ack<<"\n";
5     cout<<"FLAGS: [0]ACK:"<<((flags&0x1)?1:0)<<" [1]SYN:"<<((flags&0x2)?1:0)
    <<" [2]FIN:"<<((flags&0x4)?1:0)<<"\n";
6     cout<<" 报文长度:"<<len<<" 校验和:"<<checksum<<"\n";
7 }

```

### 5. prepare 函数

封装了 storechecksum 函数和 printdetails 函数，使得在之后实现中，在构造报文时只需要调用 prepare 函数即可。

#### printdetails 函数

```

1 void message::prepare() {
2     storechecksum();
3     printdetails();
4 }

```

#### (四) 服务器端 (server.cpp) 和客户端 (client.cpp) 的核心流程

在构建中,我都把二者的模块拆分为: init 初始化, shake 三次握手, 文件发送或接收, wave 四次挥手四个部分。

在执行任何一个部分发生错误时,直接终止程序,在四个部分均正确执行后,通过调用 closesocket 和 WSACleanup 函数以实现关闭 socket 释放内存和清除 WSA 环境,并最终结束程序。

以下是两文件中 main 函数的具体实现:

server.cpp-main

```
1 int main() {
2     //创建wsa存储socket数据
3     WSADATA wsa;
4     //创建socket
5     SOCKET ser_socket;
6     struct sockaddr_in addr,r_addr;
7     if(!server_init(wsa,ser_socket,addr,r_addr) || !server_shake(ser_socket,
8         r_addr) || !server_recvfile(ser_socket,r_addr) || !server_wave(
9         ser_socket,r_addr)){
10         cout<<"请输入任意字符,以退出程序.\n";
11         char end;
12         cin>>end;
13         return 0;
14     }
15     closesocket(ser_socket);
16     WSACleanup();
17     cout<<"程序执行完成,请输入任意字符以退出程序.\n";
18     char end;
19     cin>>end;
20     return 0;
21 }
```

client.cpp-main

```
1 int main() {
2     //创建wsa存储socket数据
3     WSADATA wsa;
4     //创建socket
5     SOCKET u_socket;
6     struct sockaddr_in u_addr,r_addr;
7     if(!client_init(wsa,u_socket,u_addr,r_addr) || !client_shake(u_socket,
8         r_addr) || !client_sendfile(u_socket,r_addr) || !client_wave(u_socket,
9         r_addr)){
10         cout<<"请输入任意字符,以退出程序.\n";
11         char end;
12         cin>>end;
13         return 0;
14     }
15     closesocket(u_socket);
16 }
```

```

14     WSACleanup();
15     cout<<"程序执行完成，请输入任意字符以退出程序。\\n";
16     char end;
17     cin>>end;
18     return 0;
19 }

```

### (五) 初始化函数 server\_init/client\_init

以 client\_init 为例，二者在日志输出和错误检查上有一些区别，但主要调用的函数并没有区别。对于实现不同的地方，已经在下面展示的代码中以注释的形式给出，对于日志和错误检查的区别不再给出。

init 的主要流程为：

- WSAStartup 函数：创建 WSA，存储 socket 数据。2, 2 表示使用 winsock2.2 版本。
- socket 函数：以 UDP 协议创建 Socket。
- ioctlsocket 函数：设置套接字非阻塞，使得无论后续执行是否成功都立刻返回，否则会一直等待，无法实现停等机制。
- sockaddr\_in 结构体的初始化：对于服务器端初始化了服务器端和 router 端的结构体，对于客户端则需要初始化客户端和 router 端的结构体。
- bind 函数：为创建的套接字绑定 sockaddr\_in 结构体信息，对于服务器端需要把 socket 和服务器端结构体绑定，对于客户端需要把 socket 和客户端结构体绑定。

#### server\_init

```

1  bool server_init(WSADATA &wsa,SOCKET &ser_socket ,sockaddr_in &addr ,
    sockaddr_in &r_addr){
2      cout<< "[UDP可靠传输协议part1--服务器端 by 2211290姚知言] \\n";
3      if (WSAStartup(MAKEWORD(2, 2), &wsa))
4      {
5          cout << "\\033[31m WSA 创建失败!  \\033[0m\\n";
6          return false;
7      }
8      ser_socket = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);//使用UDP协议
9
10     u_long mode=1;
11     if (ioctlsocket(ser_socket , FIONBIO, &mode)){
12         cout<<"\\033[31m 设置套接字非阻塞失败。  \\033[0m \\n";
13         return false;
14     }
15
16     addr.sin_family = AF_INET;
17     addr.sin_port = htons(ser_port);
18     addr.sin_addr.S_un.S_addr = INADDR_ANY;//在client中，这里需要指定为127
        .0.0.1
19 }

```

```

20     r_addr.sin_family = AF_INET;
21     r_addr.sin_port = htons(r_port);
22     r_addr.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");
23
24     if(bind(server_socket, (struct sockaddr *)&r_addr, sizeof(r_addr))!=
        SOCKET_ERROR){
25         cout<<"\033[34m 服务器启动成功! \033[0m \n";
26         return true;
27     }
28     else{
29         cout<<"\033[31m 服务器启动失败。 \033[0m \n";
30         return false;
31     }
32 }

```

## (六) 三次握手实现

注：在本文实现的所有报文中，两端分别使用全局变量 smsg 和 rmsg 表示发送报文和接收报文，在报文构建时，对于没有必要更新的变量可能没有进行更新（如 srcport, dstport, 数据传输部分的 flags, 握手挥手部分的 len 和 data 等），但显然并不影响使用。

### 1. 客户端 client\_shake 函数

对于第一次握手，由客户端发送，需要为端口初始化，生成 seq, flag=SYN，然后发送到 router。

实现代码如下：

#### 客户端第一次握手

```

1  memset(&smsg, 0, sizeof(smsg));
2  srand(time(NULL));
3  smsg.srcport = u_port;
4  smsg.dstport = r_port;
5  smsg.seq = rand() % 500;
6  smsg.flags = 2; //SYN
7  smsg.prepare();
8  int r_addrsize=sizeof(r_addr);
9  if(sendto(u_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&r_addr,
        r_addrsize)>0)
10     cout<<"\033[34m 第一次握手发送成功! \033[0m \n";
11  else{
12     cout<<"\033[31m 第一次握手发送失败。 \033[0m \n";
13     return false;
14  }

```

第二次握手由服务器端发送，设计重传逻辑（即若迟迟未能收到握手信息，则重传第一次握手）。当校验和/FLAG/ACK/SEQ 不匹配时，输出一行日志，随后逻辑同未收到报文。

注：在握手挥手的重传逻辑中，在校验和/FLAG/ACK/SEQ 不匹配时逻辑为同未收到报文，在数据传输逻辑中，对于该情况处理为立即重传并刷新时钟。（这是因为后来对数据传输进行了

逻辑优化，但并未更改握手挥手逻辑)

当超时后，进行重传并刷新时钟，达到重传限制/在过程中发生失败后产生 false。

#### 客户端第二次握手

```

1 auto lastsend = clock();
2 int sendcnt=1;
3 while(true){
4     if(recvfrom(u_socket, (char*)&rmsg, sizeof(rmsg), 0, (sockaddr*)&r_addr,
5         &r_addrsize)>0){
6         if((rmsg.flags == 3) && rmsg.verifychecksum() && rmsg.ack == smsg.seq
7             +1){
8             cout<<"\033[34m 第二次握手接收成功! \033[0m \n";
9             break;
10        }
11        else
12            cout<<"接收到了信息，但未能通过验证。 \n";
13    }
14    if(clock()-lastsend >= maxwait){
15        if(sendcnt++ >= maxsend){
16            cout<<"\033[31m 超时重传已达到最大次数，握手失败。 \033[0m \n";
17            return false;
18        }
19        else{
20            if(sendto(u_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&
21                r_addr, r_addrsize)>0){
22                cout<<"接收超时，第一次握手重传成功。 \n";
23                lastsend=clock();
24            }
25            else{
26                cout<<"\033[31m 接收超时，且第一次握手重传失败。 \033[0m \n";
27                return false;
28            }
29        }
30    }
31 }

```

收到第二次握手后，发送第三次握手，并连接成功。第三次握手的细节包括：接收第二次握手 ACK 并赋值给第三次握手 SEQ（本质上等价  $SEQ+1$ ），接收第二次握手 SEQ 并 +1 赋值给 ACK。

#### 客户端第三次握手

```

1 //第三次握手
2 memset(&smsg, 0, sizeof(smsg));
3 smsg.srcport = u_port;
4 smsg.dstport = r_port;
5 smsg.seq = rmsg.ack;
6 smsg.ack = rmsg.seq+1;
7 smsg.flags = 1;

```

```

8  smsg.prepare();
9
10 if(sendto(u_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&r_addr, sizeof
    (r_addr))>0)
11     cout<<"\033[34m 第三次握手发送成功! \033[0m \n";
12 else{
13     cout<<"\033[31m 第三次握手发送失败。 \033[0m \n";
14     return false;
15 }
16 cout<<"\033[34m 连接成功! \033[0m \n";
17 return true;

```

## 2. 服务器端 server\_shake 函数

服务器端的第一次握手则是一直等待客户端的信息, 接收到能够通过验证的报文后 break 进入后续逻辑:

### 服务器端第一次握手

```

1  while (true){
2      if(recvfrom(ser_socket, (char*)&rmsg, sizeof(rmsg), 0, (sockaddr*)&r_addr
        , &r_addrsize)>0){
3          if((rmsg.flags == 2) && rmsg.verifychecksum() && rmsg.ack == 0){
4              cout<<"\033[34m 第一次握手接收成功! \033[0m \n";
5              break;
6          }
7          else
8              cout<<"接收到了信息, 但未能通过验证。 \n";
9      }
10 }

```

服务器端的第二次握手包括了更新 ack 为接收到的 seq+1, 初始化 seq, 设置 flag 为 SYN ACK, 并且正确初始化源端口和目的端口, 具体代码如下。

### 服务器端第二次握手

```

1  smsg.srcport = ser_port;
2  smsg.dstport = r_port;
3  smsg.seq = rand() % 500;
4  smsg.ack = rmsg.seq+1;
5  smsg.flags = 3; //SYN ACK
6  smsg.prepare();
7
8  if(sendto(ser_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&r_addr,
    r_addrsize)>0)
9      cout<<"\033[34m 第二次握手发送成功! \033[0m \n";
10 else{
11     cout<<"\033[31m 第二次握手发送失败。 \033[0m \n";
12     return false;
13 }

```

服务器端第三次握手逻辑同样是等待 + 超时重传的模式，具体实现与上文提到的客户端第二次握手类似，不过对于接收报文的判定并不相同（包括了 flags，校验和和 ack 的判定）。

接收到第三次握手后，即为连接成功。

#### 服务器端第三次握手

```

1  auto lastsend = clock();
2  int sendcnt=1;
3  while(true){
4      if(recvfrom(ser_socket, (char*)&rmsg, sizeof(rmsg), 0, (sockaddr*)&r_addr
5          , &r_addrsize)>0){
6          if((rmsg.flags == 1) && rmsg.verifychecksum() && rmsg.ack == smsg.seq
7              +1){
8              cout<<"\033[34m 第三次握手接收成功! \033[0m \n";
9              break;
10             }
11             else
12                 cout<<"接收到了信息，但未能通过验证。 \n";
13         }
14         if(clock()-lastsend >= maxwait){
15             if(sendcnt++ >= maxsend){
16                 cout<<"\033[31m 超时重传已达到最大次数，握手失败。 \033[0m \n";
17                 return false;
18             }
19             else{
20                 if(sendto(ser_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&
21                     r_addr, r_addrsize)>0){
22                     cout<<"接收超时，第二次握手重传成功。 \n";
23                     lastsend=clock();
24                 }
25                 else{
26                     cout<<"\033[31m 接收超时，且第一次握手重传失败。 \033[0m \n";
27                     return false;
28                 }
29             }
30         }
31     }
32 }
33 cout<<"\033[34m 连接成功! \033[0m \n";
34 return true;

```

## (七) 数据传输实现

### 1. 客户端 client\_sendfile 函数

传输的文件名设置阶段：我预定义了老师要求的四个文件的传输，可以通过输入 1-4 开始文件传输，同样，也保留了输入其他文件进行传输的端口。

#### 文件名设置

```

1  int mode;

```

```

2 string file;
3 cout << "可靠数据传输测试开始:\n";
4 cout << "请输入一个数字, 表示测试文件:\n1:1.jpg\n2:2.jpg\n3:3.jpg\n4:
   helloworld.txt\n5:自定义输入文件\n";
5 cin >> mode;
6 switch(mode){
7     case 1: file="1.jpg"; break;
8     case 2: file="2.jpg"; break;
9     case 3: file="3.jpg"; break;
10    case 4: file="helloworld.txt"; break;
11    default: cout<<"请输入文件名:"; cin>>file; break;
12 }
13 cout<<"开始对文件"<<file<<"的传输测试。 \n";

```

文件名读取阶段: 在文件名设置完成后, 首先创建一个时钟, 用于后续传输时间和吞吐量的计算。随后以二进制打开文件, 并且逐字节读取到 buffer 中, 此处 buffer 的大小是查看了要求的四个文件大小后设置。

#### 文件名读取

```

1 auto timestart = clock();
2 ifstream textfile(file.c_str(), ifstream::binary);
3 if(!textfile){//如果文件不能打开
4     cout<<"\033[31m 文件读取失败。 \033[0m \n";
5     return false;
6 }
7
8 u_long size=0;
9 BYTE* buffer = new BYTE[15000000];
10 BYTE byte = textfile.get();
11 while (textfile) {
12     buffer[size++] = byte;
13     byte = textfile.get();
14 }
15 textfile.close();

```

文件信息报文封装与传输阶段:

构建报文格式为 (以 1.jpg 为例, 已知其字节数为 1,857,353):

文件名						文件字节数						
1	.	j	p	g	\0	3	5	3	7	5	8	1

在读取的时候, 通过 len 可以找到开始位置, 通过与 \0 的比对可以确定结束位置, 从而完成文件名和文件字节数的传输。

代码实现如下:

#### 文件信息报文封装与传输

```

1 //port seq ack无需更新
2 msg.flags=0;

```



```

3  int nowlen=0;
4  for (;nowlen<file.size();nowlen++)
5      smsg.data[nowlen]=file[nowlen];
6  smsg.data[nowlen++]='\0';
7  u_long tempsize=size;
8  while(tempsize){
9      smsg.data[nowlen++]=tempsize%10+48;
10     tempsize/=10;
11 }
12 smsg.len=nowlen;
13 smsg.prepare();
14 int r_addrsz=sizeof(r_addr);
15 if(sendto(u_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&r_addr,
16     r_addrsz)>0)
17     cout<<"\033[34m 文件信息报文发送成功! \033[0m \n";
18 else{
19     cout<<"\033[31m 文件信息报文发送失败。 \033[0m \n";
20     return false;
21 }

```

文件报文传输与 ACK 接收阶段:

在首先通过文件大小和报文最大长度计算出要分为多少个批次 (totbatch), 以及最后一个批次的大小 (lastbatch)。初始批次为 0, 因为开始的时候是接收文件信息确认报文, 并发送第一个文件报文。

对校验和/ACK/FLAG 进行比对, 接收到正确报文后打包下一个文件报文, 将文件要发送的信息端即从 (nowbatch-1)\*maxlen 开始的一段存入发送报文的 data 中, 并进行发送。

若收到了不匹配的报文, 立刻重传并刷新时钟以提升效率。同时设计超时重传逻辑。二者共享重传次数, 达到最大发送次数后 false。

在收到最后一个 ACK 报文后 (具体体现为: nowbatch==totbatch 时收到了通过验证的报文), 说明传输已经结束, 可以退出。

注: 在握手挥手的重传逻辑中, 在校验和/FLAG/ACK/SEQ 不匹配时逻辑为同未收到报文, 在数据传输逻辑中, 对于该情况处理为立即重传并刷新时钟。(这是因为后来对数据传输进行了逻辑优化, 但并未更改握手挥手逻辑)

以下是代码实现:

#### 文件报文传输与 ACK 接收

```

1  auto lastsend = clock();
2  int sendcnt=1;
3
4  u_long nowbatch=0,totbatch=(size-1)/maxlen+1,lastbatch=size%maxlen;
5  if(!lastbatch) lastbatch=maxlen;
6  while(true){
7      if(recvfrom(u_socket, (char*)&rmsg, sizeof(rmsg), 0, (sockaddr*)&r_addr,
8          &r_addrsz)>0){
9          if((rmsg.flags == 1) && rmsg.verifychecksum() && rmsg.ack == smsg.seq
10             +smsg.len){//想要的就是ta
11              if(nowbatch==totbatch)break;

```

```

10         cout<<"\033[34m 收到了新的确认报文，发送新的报文! \033[0m \n";
11         nowbatch++;
12         smsg.seq+=smsg.len;
13         smsg.len=((nowbatch==totbatch)?lastbatch:maxlen);
14         memset(&smsg.data,0,sizeof(smsg.data));
15         for(int i=0;i<smsg.len;i++)
16             smsg.data[i]=buffer[i+(nowbatch-1)*maxlen];
17         smsg.prepare();
18         if(sendto(u_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&
19             r_addr, r_addrsize)>0)
20             cout<<"\033[34m 新的报文发送成功! \033[0m \n";
21         else{
22             cout<<"\033[31m 新的报文发送失败。 \033[0m \n";
23             return false;
24         }
25         sendcnt=1;//重置发送次数
26     }
27     else{//想要的不是ta
28         cout<<"接收到了信息，但未能通过验证，发送重传报文。 \n";
29         if(sendcnt++>= maxsend){
30             cout<<"\033[31m 重传已达到最大次数，验证失败。 \033[0m \n";
31             return false;
32         }
33         if(sendto(u_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&
34             r_addr, r_addrsize)>0)
35             cout<<"\033[34m 重传报文发送成功! \033[0m \n";
36         else{
37             cout<<"\033[31m 重传报文发送失败。 \033[0m \n";
38             return false;
39         }
40     }
41     lastsend=clock();//无论是哪一种，刷新时钟
42 }
43 if(clock()-lastsend >= maxwait){
44     if(sendcnt++>= maxsend){
45         cout<<"\033[31m 重传已达到最大次数，验证失败。 \033[0m \n";
46         return false;
47     }
48     else{
49         if(sendto(u_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&
50             r_addr, r_addrsize)>0){
51             cout<<"接收超时，重传报文发送成功。 \n";
52             lastsend=clock();
53         }
54         else{
55             cout<<"\033[31m 接收超时，重传报文发送失败。 \033[0m \n";
56             return false;
57         }
58     }
59 }

```

```

55     }
56 }
57 }
58 cout<<"\033[34m 文件已成功发送完毕! \033[0m \n";

```

传输时间和吞吐量打印阶段:

在文件发送完毕后, 调用文件读取前存储的 `timestart`, 和当前时钟计算传输时间, 并通过文件大小除以时间计算吞吐量。

#### 传输时间和吞吐量打印

```

1 auto timeconnect=clock()-timestart;
2 cout<<"传输时间:"<<double(timeconnect)/CLOCKS_PER_SEC<<"s\n";
3 cout<<"吞吐量:"<<double(size)/double(timeconnect)*CLOCKS_PER_SEC<<"Byte/s\n";
4 return true;

```

## 2. 服务器端 `server_recvfile` 函数

文件信息报文接收和接收报文发送阶段:

首先接收文件信息报文并验证 (`flag=0`, `ack=seq+1`, 校验和验证), 直到收到了正确的报文, 发送确认接收报文, 解包保存 `size` 和 `filename` 信息。

#### 文件信息报文接收和接收报文发送

```

1 int r_addrsz=sizeof(r_addr);
2 char filename[50]={'\0'};
3 while (true){
4     if(recvfrom(ser_socket, (char*)&rmsg, sizeof(rmsg), 0, (sockaddr*)&r_addr,
5         &r_addrsz)>0){
6         if((rmsg.flags == 0) && rmsg.verifychecksum() && rmsg.ack == rmsg.seq
7             +1){
8             cout<<"\033[34m 文件信息报文接收成功! \033[0m \n";
9             break;
10        }
11        else
12            cout<<"接收到了信息, 但未能通过验证.\n";
13    }
14}
15
16 u_long nowlen=rmsg.len;
17 u_long size=0;
18 while(rmsg.data[--nowlen]!='\0'){
19     size+=10;
20     size+=(rmsg.data[nowlen]-48);
21 }
22 for(int i=0;i<=nowlen;i++){
23     filename[i]=rmsg.data[i];
24 }
25 rmsg.seq +=1;
26 //后续rmsg.seq 因为在传输阶段长度一直为0, 不再更新
27 rmsg.ack = rmsg.seq+rmsg.len;

```

```

25 msg.flags = 1; //ACK
26 msg.prepare();
27
28 if(sendto(ser_socket, (char*)&msg, sizeof(msg), 0, (sockaddr*)&r_addr,
    r_addrsz)>0)
29     cout<<"\033[34m 文件信息接收报文发送成功! \033[0m \n";
30 else{
31     cout<<"\033[31m 文件信息接收报文发送失败。 \033[0m \n";
32     return false;
33 }

```

文件接收并发送确认阶段:

首先根据接收到的 size 动态开辟 buffer, 定义 ptr 让 buffer 一直按顺序写入, 检查 flag, 校验和, SEQ 和 ACK。(这样可以保证只要发送端是按顺序传输的, 无论是否把包装满都可以正常执行。)

当收到的字节量与之前声明的字节量匹配的时候, 就结束了接收。

报文检查, 超时重传等逻辑同客户端。

#### 文件报文接收和接收报文发送

```

1 BYTE* buffer = new BYTE[size];
2 u_long ptr=0;
3 auto lastsend = clock();
4 int sendcnt=1;
5 u_long totseq=msg.ack+size, nowseq=msg.ack;
6 while(true){
7     if(recvfrom(ser_socket, (char*)&rmsg, sizeof(rmsg), 0, (sockaddr*)&r_addr,
    &r_addrsz)>0){
8         if((rmsg.flags == 0) && rmsg.verifychecksum() && rmsg.ack == msg.seq
    && rmsg.seq==nowseq){ //想要的就是ta
9             cout<<"\033[34m 收到了新的文件报文, 发送确认报文! \033[0m \n";
10            msg.ack = rmsg.seq+rmsg.len;
11            msg.prepare();
12            nowseq=msg.ack;
13            if(sendto(ser_socket, (char*)&msg, sizeof(msg), 0, (sockaddr*)&
    r_addr, r_addrsz)>0)
14                cout<<"\033[34m 确认报文发送成功! \033[0m \n";
15            else{
16                cout<<"\033[31m 确认报文发送失败。 \033[0m \n";
17                return false;
18            }
19            //store
20            for(int i=0;i<rmsg.len;i++){
21                buffer[ptr++]=rmsg.data[i];
22            if(rmsg.seq+rmsg.len==totseq)break; //最后一个ack
23            sendcnt=1; //重置发送次数
24        }
25        else{//想要的不是ta
26            cout<<"接收到了信息, 但未能通过验证, 发送重传报文。 \n";

```

```

27         if(sendcnt++ >= maxsend){
28             cout<<"\033[31m 重传已达到最大次数，验证失败。 \033[0m \n";
29             return false;
30         }
31         if(sendto(ser_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&
32             r_addr, r_addrsize)>0)
33             cout<<"\033[34m 重传报文发送成功! \033[0m \n";
34         else{
35             cout<<"\033[31m 重传报文发送失败。 \033[0m \n";
36             return false;
37         }
38     }
39     lastsend=clock(); //无论是哪一种，刷新时钟
40     if(clock()-lastsend >= maxwait){
41         if(sendcnt++ >= maxsend){
42             cout<<"\033[31m 重传已达到最大次数，验证失败。 \033[0m \n";
43             return false;
44         }
45         else{
46             if(sendto(ser_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&
47                 r_addr, r_addrsize)>0){
48                 cout<<"接收超时，重传报文发送成功。 \n";
49                 lastsend=clock();
50             }
51             else{
52                 cout<<"\033[31m 接收超时，重传报文发送失败。 \033[0m \n";
53                 return false;
54             }
55         }
56     }
57     cout<<"\033[34m 文件已成功接收完毕，开始写入! \033[0m \n";

```

文件写入阶段:

在文件完整接收后，建立文件流 file，以先前的 filename 创建文件，并把 buffer 内容写入，然后关闭文件。

#### 文件写入

```

1  ofstream file(filename, ios::binary);
2  if (!file.is_open()) {
3      cout<<"\033[31m 无法创建文件"<<filename<<"。 \033[0m \n";
4      return false;
5  }
6  // 将 buffer 的内容写入文件
7  file.write(reinterpret_cast<char*>(buffer), size);
8  // 关闭文件
9  file.close();

```

```

10 cout<<"\033[34m 文件已成功写入完毕! \033[0m \n";
11 return true;

```

## (八) 四次挥手实现

挥手实现与握手实现有很多相似的地方，但由于挥手逻辑不同于握手，也有一些区别。

### 1. 客户端 client\_wave 函数

对于第一次挥手报文，需要置位 FIN, ACK, 调整 SEQ,ACK(因为之前已经调整过，所以不需要在此调整)，设置长度为 0。

#### 客户端第一次挥手

```

1 memset(&smsg.data,0,sizeof(smsg.data)); //清空data
2 //port seq ack不更新
3 smsg.flags = 5; //FIN,ACK
4 smsg.len = 0;
5 smsg.prepare();
6 if(sendto(u_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&r_addr,
7   r_addrsize)>0)
8   cout<<"\033[34m 第一次挥手发送成功! \033[0m \n";
9 else{
10   cout<<"\033[31m 第一次挥手发送失败。 \033[0m \n";
11   return false;
12 }

```

对于第二次挥手，我们期望收到的标记位是 ACK，同时 ACK 为之前发送的 SEQ+1，SEQ 为之前发送的 ACK。若迟迟没能接收到通过验证的第二次挥手，则需要重传第一次挥手，重传逻辑与握手相同，不再赘述。

#### 客户端第二次挥手

```

1 auto lastsend = clock();
2 int sendcnt=1;
3 while(true){
4   if(recvfrom(u_socket, (char*)&rmsg, sizeof(rmsg), 0, (sockaddr*)&r_addr,
5     &r_addrsize)>0){
6     if((rmsg.flags == 1) && rmsg.verifychecksum() && rmsg.ack == smsg.seq
7       +1){
8       cout<<"\033[34m 第二次挥手接收成功! \033[0m \n";
9       break;
10     }
11   }
12   else
13     cout<<"接收到了信息，但未能通过验证。 \n";
14 }
15 if(clock()-lastsend >= maxwait){
16   if(sendcnt++ >= maxsend){
17     cout<<"\033[31m 超时重传已达到最大次数，挥手失败。 \033[0m \n";
18     return false;
19   }
20 }

```

```

16     }
17     else{
18         if(sendto(u_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&
19             r_addr, r_addrsize)>0){
20             cout<<"接收超时，第一次挥手重传成功。\\n";
21             lastsend=clock();
22         }
23         else{
24             cout<<"\\033[31m 接收超时，且第一次挥手重传失败。\\033[0m \\n";
25             return false;
26         }
27     }
28 }

```

在第二次挥手接收成功后，进入第三次挥手的等待逻辑，直至接收到匹配报文。

希望接收到的报文标记为 FIN ACK, SEQ 和 ACK 与第二次挥手相同，同样需要通过校验和验证。

等待接收到正确的报文后，发送第四次挥手。

#### 客户端第三次挥手

```

1 while (true){
2     if(recvfrom(u_socket, (char*)&rmsg, sizeof(rmsg), 0, (sockaddr*)&r_addr,
3         &r_addrsize)>0){
4         if((rmsg.flags == 5) && rmsg.verifychecksum() && rmsg.ack == smsg.seq
5             +1){
6             cout<<"\\033[34m 第三次挥手接收成功! \\033[0m \\n";
7             break;
8         }
9         else
10            cout<<"接收到了信息，但未能通过验证。\\n";
11     }
12 }

```

第四次挥手的标记为 ACK, SEQ 自增 1 (因为之前发送了 FIN 报文), ACK 为收到的 SEQ+1 (因为接收了 FIN 报文)。发送后，等待 2RTT (这里直接用等待 maxwait 代替)，然后结束程序。

#### 客户端第四次挥手

```

1 smsg.seq +=1;
2 smsg.ack = rmsg.seq+1;
3 smsg.flags = 1; //ACK
4 smsg.prepare();
5
6 if(sendto(u_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&r_addr,
7     r_addrsize)>0)
8     cout<<"\\033[34m 第四次挥手发送成功! \\033[0m \\n";
9 else{

```

```

9      cout<<"\033[31m 第四次挥手发送失败。 \033[0m \n";
10      return false;
11  }
12  Sleep(maxwait);
13  return true;

```

## 2. 服务器端 server\_wave 函数

服务器端一直等待客户端发送的第一次挥手，通过验证后跳出循环。

### 服务器端第一次挥手

```

1  while (true){
2      if(recvfrom(ser_socket, (char*)&rmsg, sizeof(rmsg), 0, (sockaddr*)&r_addr,
3          &r_addrsize)>0){
4          if((rmsg.flags == 5) && rmsg.verifychecksum() && rmsg.ack == smsg.seq
5              +smsg.len){
6              cout<<"\033[34m 第一次挥手接收成功! \033[0m \n";
7              break;
8          }
9          else
10             cout<<"接收到了信息，但未能通过验证。 \n";
11     }
12 }

```

收到第一次挥手后，服务器端先后发送第二次挥手、第三次挥手，设置 SEQ, ACK, FLAG 分别为 ACK 和 ACK FIN。

### 服务器端第二三次挥手

```

1  //第二次挥手
2  memset(&smsg.data,0,sizeof(smsg.data)); //清空data
3  smsg.seq = rmsg.ack;
4  smsg.ack = rmsg.seq+1;
5  smsg.flags = 1; //ACK
6  smsg.prepare();
7
8  if(sendto(ser_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&r_addr,
9      r_addrsize)>0)
10     cout<<"\033[34m 第二次挥手发送成功! \033[0m \n";
11 else{
12     cout<<"\033[31m 第二次挥手发送失败。 \033[0m \n";
13     return false;
14 }
15
16 //第三次挥手
17 smsg.flags = 5; //FIN ACK
18 smsg.prepare();
19 if(sendto(ser_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&r_addr,
20     r_addrsize)>0)

```



```

19     cout<<"\033[34m 第三次挥手发送成功! \033[0m \n";
20 else{
21     cout<<"\033[31m 第三次挥手发送失败。 \033[0m \n";
22     return false;
23 }

```

随后,服务器端等待第四次挥手的接收,直到接收到正确的第四次挥手报文。同样实现了超时重传机制,这里仅重传了第三次挥手作为演示,理论上还应考虑重传第二次挥手,不过由于服务器端并不会丢包,所以不会造成影响。

#### 服务器端第四次挥手

```

1 auto lastsend = clock();
2 int sendcnt=1;
3 while(true){
4     if(recvfrom(ser_socket, (char*)&rmsg, sizeof(rmsg), 0, (sockaddr*)&r_addr, &r_addrsize)>0){
5         if((rmsg.flags == 1) && rmsg.verifychecksum() && rmsg.ack == smsg.seq+1){
6             cout<<"\033[34m 第四次挥手接收成功! \033[0m \n";
7             break;
8         }
9         else
10            cout<<"接收到了信息, 但未能通过验证.\n";
11    }
12    if(clock()-lastsend >= maxwait){
13        if(sendcnt++ >= maxsend){
14            cout<<"\033[31m 超时重传已达到最大次数, 挥手失败. \033[0m \n";
15            return false;
16        }
17        else{
18            if(sendto(ser_socket, (char*)&smsg, sizeof(smsg), 0, (sockaddr*)&r_addr, r_addrsize)>0){
19                cout<<"接收超时, 第三次挥手重传成功.\n";
20                lastsend=clock();
21            }
22            else{
23                cout<<"\033[31m 接收超时, 且第三次挥手重传失败. \033[0m \n";
24                return false;
25            }
26        }
27    }
28 }
29 return true;

```

## 五、性能测试

### (一) 测试环境配置

## 生成可执行文件

1 生成可执行文件的指令如下：

2 //服务器端

3 g++ code/server.cpp code/message.cpp -o exe\_and\_testcase/server/server -  
lws2\_32

4 //客户端

5 g++ code/client.cpp code/message.cpp -o exe\_and\_testcase/client/client -  
lws2\_32

测试用 Router 设置如下：



图 2: Router 设置

## (二) 正确性测试

四个文件均正确通过了测试，以图像 1 为例进行展示。

图3展示了图像 1.jpg 的传输结果，大小完全相同且可以正确打开。



图 3: 接收到的图像 1.jpg

接下来对日志信息进行分析，由于中间报文传输的包数量众多，仅截取开头部分和结尾部分进行分析（已经涵盖了每一个部分的测试）。

图4展示了开头段的日志。

从开头部分可以看出三次握手完全正确，随后对文件信息报文的传输（len=13，与之前分析相同）和文件开头传输也没有问题。

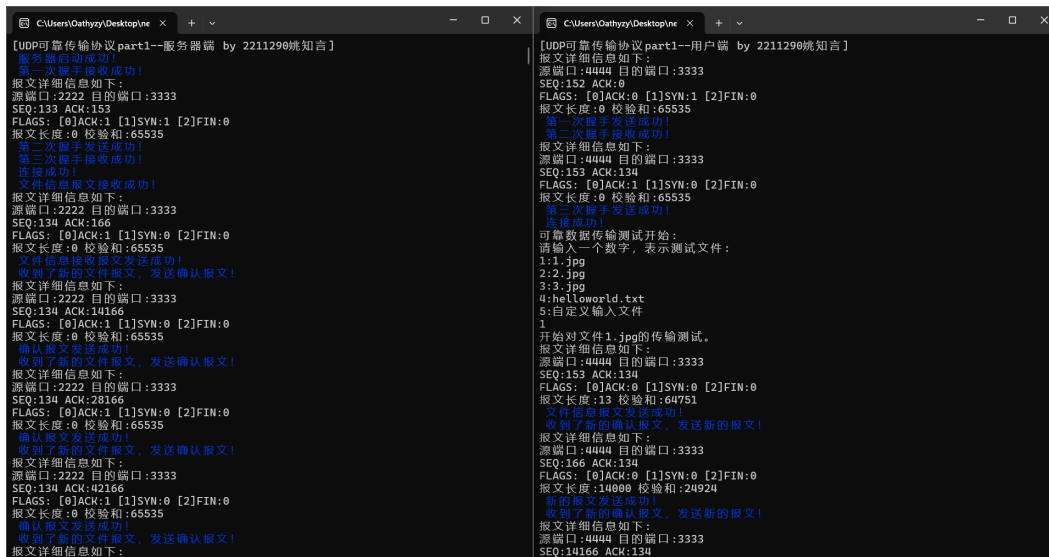


图 4: 图像 1.jpg 最终测试结果 (开头段)

图5展示了结尾段的日志。

从结尾部分可以看出对丢包的正确处理:

客户端发送的报文丢失，会先导致服务器端超时，服务器端超时重传上一个包，会导致客户端验证失败，并重传上一个包，回到正常逻辑。

然后可以看出在文件传输结束后，都可以正常退出，且最后的挥手逻辑也完全正确。

在文件发送结束后，客户端也成功打印了传输时间和吞吐量，用于后续测试。

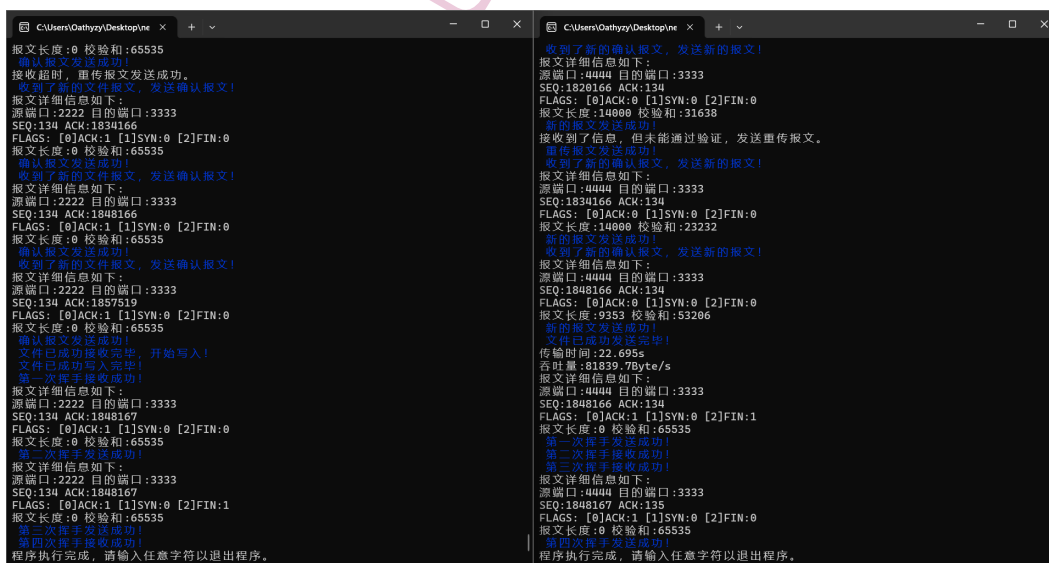


图 5: 图像 1.jpg 最终测试结果 (结尾段)

### (三) 性能分析

在本实验中的测试属性：（在之前 message.h 中定义，在此总结）

超时时间：2000ms 最多发送次数：10

Router 的设置属性：

客户端丢包率：5% 发送时延：50ms

四个文件的性能测试结果如表1所示：

文件名	传输时间 (s)	吞吐量 (Byte/s)
1.jpg	22.695	81839.7
2.jpg	71.411	81458.7
3.jpg	158.817	75363.4
helloworld.txt	23.573	70241.7

表 1: 性能测试结果

## 六、 总结

在本次实验中，我利用 UDP 协议，通过停等机制实现了可靠的数据传输。在后续实验中，我也将进一步探索，对其进行进一步的了解。