南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机网络实验报告

大作业第三部分实验报告

汤清云 2013536

年级：2020 级

专业：计算机科学与技术

指导教师：张建忠　徐敬东

2022 年 12 月 29 日

摘要

本次实验为最终大作业的第三部分实现。在本次实验中，我在之前基础上做了一些改进。具体修改如下：将 3-2 中的实验修改为双线程以实现能够同时收发数据包并进行相应处理的功能；此外在本次实验中我选择实现的拥塞控制算法为 TCP 拥塞控制的 RENO 算法，以此实现发送窗口的动态调整。

关键词：拥塞控制　多线程　窗口滑动　吞吐率　GBN

目录

一、 实验目的

在实验 3-2 的基础上，实现一种拥塞控制算法，也可以是改进的算法，实现给定测试文件的传输。

二、 实验知识

(一) 网络拥塞

在某段时间内，如果对网络中某一资源的需求超出了该资源所能提供的上限，则会导致网络性能的下降，这种情况被称为网络拥塞。例如当主机发送的数据过多过快时会造成网络中的路由器无法及时处理，因而引入了排队时延。

(二) 拥塞控制

当出现网络拥塞时，为了降低网络的阻塞同时尽快传输数据而引入了拥塞控制。当发送完数据包后依据是否接收到 ACK 包来进行带宽的检测。如果接收到 ACK 包说明网络并未拥塞，可以提高发送速率；发生丢失事件则降低发送速率。

(三) RENO 算法

RENO 算法将发送窗口增大的过程分为两个阶段：第一个是慢启动阶段，在这一阶段中窗口大小被初始化为 1，每当收到一个新的 ACK 时窗口大小变为原来的两倍，当窗口尺寸大于等于门限值 ssthresh 时则进入拥塞避免阶段。在拥塞避免阶段，每收到一个新的 ACK，则将窗口大小 +1，出现超时未收到 ACK 的情况则返回慢启动阶段，将 ssthresh 置为当前窗口大小的一半，窗口大小恢复为 1。

而当在慢启动阶段或拥塞避免阶段连续收到三个重复 ACK 时则进入快速恢复阶段，ssthresh 设置为当前窗口的一半，窗口大小 =ssthresh+3

RENO 算法的状态机图示如下：

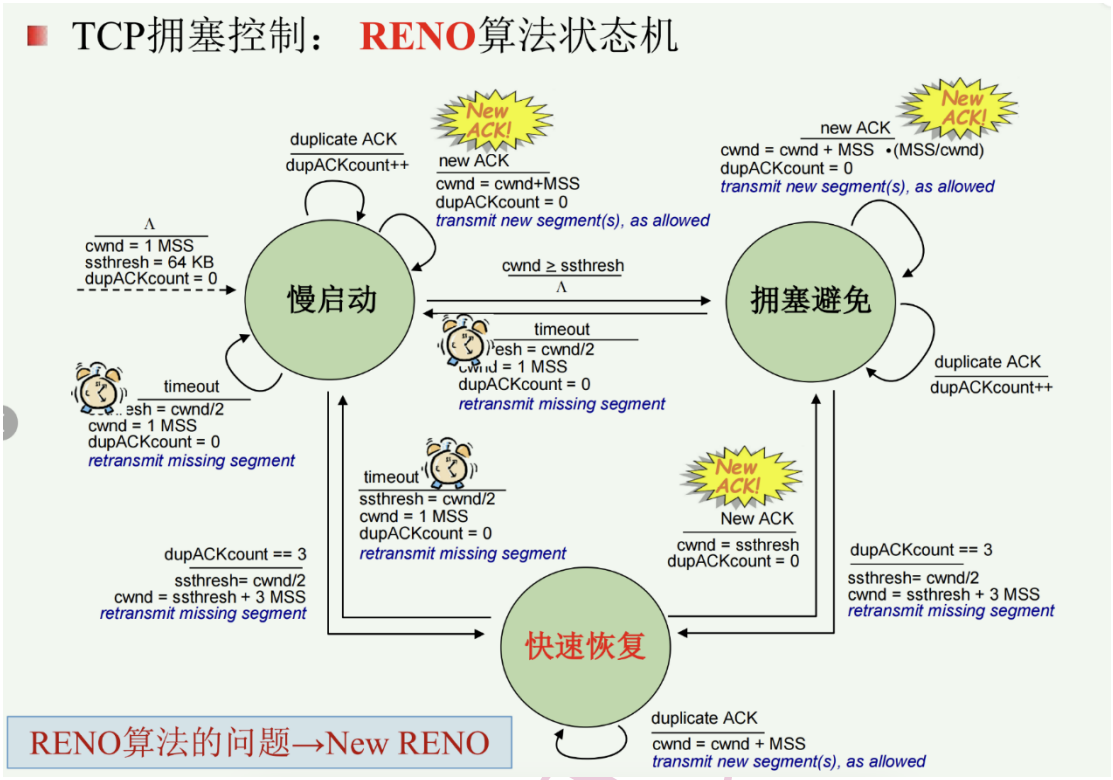

图 1: RENO 算法状态机

三、 实验设计

(一) 窗口格式

在发送端 client 创建一个结构体用来记录发送窗口，其具体变量如下。

```
struct sendwindowstruct//发送窗口
{
        unsigned int size;//窗口大小
        unsigned int ceiling;//上限
        unsigned int floor;//下限，已发送的最大seq
        unsigned int sentbutnotchecked;//记录已发送但未确认的最大seq
        unsigned int ssthresh;//慢开始门限
        unsigned int curACKseq;//重复ACK的序号
        unsigned int dupACKcount;//重复ACK的个数
        unsigned char state;//当前状态
        sendwindowstruct()
        {
                size = 1;//窗口大小初始化为1
                floor = 1;//从1开始编号
                ceiling = 1;//上限
                sentbutnotchecked = 0;//初始时没有包被确认
                ssthresh = 14;
                state = SLOW_START;//初始状态为慢启动状态
        }
```

```
};
```

(二) 数据报格式

数据报由两部分组成，第一部分是 head，即数据报头，第二部分是所携带的数据部分，长度上限为 2048 字节。在数据报头 UDPpackethead 和数据报 UDPpacket 中包含以下内容：

```
    struct UDPpackethead
{
        unsigned int seq;//序列号 16位
        u_short Check;//校验 16位
        u_short len;//数据部分总长度
        unsigned char flags;//标志位
        UDPpackethead()
        {
                len = 0;
                seq = 0;
                Check = 0;
                flags = 0;
        }
};
struct UDPpacket
{
        UDPpackethead head;
        char data[2048];//每段data长度上限
};
```

(三) 双线程代码实现

对于客户端 client，将发送数据包作为主线程，而将收到 ACK 包并确认作为分线程。其具体流程如下：

A. 主线程调用发送文件函数，在此函数中完成对全局变量总发送数据包个数的定义后开辟新线程，用于接收服务器端返回的 ACK 包。

B. 主线程发送初始窗口中（不包含最后一个数据包）的所有包，在发送第一个数据包时开始计时；每发送一个数据包则更新当前窗口的已发送未确认 seq 号 sentbutnotchecked 以及下一个要发送数据包的序列号 sequence，之后只要发送窗口的 sentbutnotchecked != ceiling 则说明收到了新的 ACK 包，继续发送窗口内未发送的数据包（序列号在 sentbutnotchecked ceiling 之间）；一旦超时则重发整个窗口内所有的数据包。当识别到全局变量 final=1 时表示已经接收到最后一个数据包的 ACK，结束运行。

C. 新线程负责接收确认包，对于接收到的确认包，如果其校验码正确且序列号大于等于当前窗口底部，则将发送窗口对应右移，保证当前窗口内序列号对应数据包均为未确认状态；如果其序列号为最后一个确认包（校验码正确且头部标志位为 FINAL_ACK），则代表均已正确完成，等待两个时钟周期后以 return 方式关闭此线程。

主线程发送数据伪代码实现：

```
int sendmessage(int messagelength, char* message, int addrlength, SOCKET&
    client, SOCKADDR_IN& ServerAddr)
```

```
{
        //设置为非阻塞模式，便于在之后进行超时判断
        //计算要传多少个包packetnum，向上取整
        //创建接受数据包并确认的线程
    //主线程使用while循环防止中断
        while (sequence <= packetnum)
        {
                //if（当前文件传输完成）then return
                //窗口移动会导致sentbutnotcheck<ceiling
                else if (sendwindow.sentbutnotchecked < sendwindow.ceiling)
                {
                        //将窗口内未发送的数据包发送出去
                        //更新发送时间
                }
                //如果超时则重传当前窗口内所有包
                else if (超时)
                {
                        //超时重传则进入慢启动阶段，相关变量修正
                        //超时重传窗口内所有数据包
                        for (int i = sendwindow.floor;i <= sendwindow.ceiling
                            ;i++)
                        {
                                //发送包
                        }
                        //更新发送时间
                }
                else//其他情况下不需要做任何处理
                {
                        continue;
                }
        }
}
```

分线程接受数据包伪代码实现:

```
DWORD WINAPI recvMsgThread(LPVOID IpParameter)
{
        //相关参数设置完毕后使用while循环接受信息
        while (1)
        {
                //收到包后先判断是否是最后一个数据包
                if (res == 0 && recvpacket.head.flags == FINAL_ACK)
                {
                        finished = 1;//全局变量，=1即全部传输完成
                        //等待2MSL，防止和挥手包混淆，之后return方式关闭线程
                }
        //校验码正确且收到的ACK包序号是否>=当前窗口底部（此处为累计确认思想实
            现）
                else if (res == 0 && recvpacket.head.seq >= sendwindow.floor)
```

```
14                  {
15                          //慢启动阶段收到新的ACK, cwnd翻倍
16                          if (sendwindow.state == SLOW_START)
17                          {
18                                  //对应窗口前移, sendwindow内部变量更新
19                                  sendwindow.floor = recvpacket.head.seq + 1;
20                                  sendwindow.size *= 2;
21                                  sendwindow.ceiling = sendwindow.floor +
                                        sendwindow.size - 1;
22                                  if (sendwindow.ceiling >= packetnum)
23                                  {
24                                          sendwindow.ceiling = packetnum;
25                                  }
26                                  sendwindow.curACKseq = recvpacket.head.seq;
27                                  sendwindow.dupACKcount = 0;
28                                  //>ssthresh则进入拥塞避免阶段
29                                  if (sendwindow.size >= sendwindow.ssthresh)
30                                  {
31                                          sendwindow.state = CONGESTION_AVOID;
32                                  }
33                          }
34                          //拥塞避免阶段收到新的ACK, 线性增长
35                          if (sendwindow.state == CONGESTION_AVOID)
36                          {
37                                  //对应窗口前移
38                                  sendwindow.floor = recvpacket.head.seq + 1;
39                                  sendwindow.size += 1;
40                                  sendwindow.ceiling = sendwindow.floor +
                                        sendwindow.size - 1;
41                                  if (sendwindow.ceiling >= packetnum)
42                                  {
43                                          sendwindow.ceiling = packetnum;
44                                  }
45                                  sendwindow.curACKseq = recvpacket.head.seq;
46                                  sendwindow.dupACKcount = 0;
47                          }
48                          //快速恢复阶段收到新的ACK, 进入拥塞避免
49                          if (sendwindow.state == QUICK_RECOVERY)
50                          {
51                                  //对应窗口前移
52                                  sendwindow.state = CONGESTION_AVOID;
53                                  sendwindow.floor = recvpacket.head.seq + 1;
54                                  sendwindow.size = sendwindow.ssthresh;
55                                  sendwindow.ceiling = sendwindow.floor +
                                        sendwindow.size - 1;
56                                  if (sendwindow.ceiling >= packetnum)
57                                  {
58                                          sendwindow.ceiling = packetnum;
```

```
59                                  }
60                                  sendwindow.curACKseq = recvpacket.head.seq;
61                                  sendwindow.dupACKcount = 0;
62                          }
63                  }
64                  else
65                  {
66                          if (重复的ACK)
67                          {
68                                  sendwindow.dupACKcount += 1;
69                          }
70                          //一旦收到三个重复ACK就进入快速恢复阶段
71                          if (sendwindow.dupACKcount == 3)
72                          {
73                                  //进入快速恢复阶段, 窗口变量更新
74                          }
75                          continue;
76                  }
77          }
78  }
```

对于接收方 server 而言，修改为双线程的意义不大，因此维持 3-1 中的编写即可，逻辑如下：

A. 本地维护一个 Ssequence 用于记录希望得到的数据包序列号，对于接收到的数据包进行判断。

B. 如果收到的数据包校验和正确 && 序列号为期望得到的数据包序列号，那么接收此数据包并写入文件缓冲区。维护的 Ssequence+1，之后发送回去的 ACK 确认包序列号为当前的 Ssequence，表示接下来希望接收的数据包编号为 Ssequence。

C. 如果收到的数据包校验和错误 || 序列号不是期望得到的数据包序列号，那么拒绝接收此数据包，维护的 Ssequence 不做变动，之后发送回去的 ACK 确认包序列号为当前的 Ssequence，表示接下来希望接收的数据包编号为仍然为 Ssequence。

D. 直到接收到的数据包标志位为 FINAL 时表示为最后一个数据包，此时发回 ACK 包头部置位为 FINAL_ACK，等待 2MSL 查看是否收到来自客户端发送的数据包，如果没收到说明 FINAL_ACK 包被正确接收，结束等待。如果收到数据包说明 FINAL_ACK 包丢失，此时等待客户端发来数据包后重传 FINAL_ACK 包即可。

(四) 建立连接

此处协议设计模拟了 TCP 的三次握手协议。

第一次握手：客户端向服务器发送头部标志位为 SYN 的数据包，表示想与服务器建立连接：

A. 此数据包校验和错误/中途丢失，服务器接收后直接丢弃，继续处于等待接收状态，不发送数据包——客户端在经过一段时间后收不到返回数据包，因此重发标志位为 SYN 的数据包。

B. 此数据包被服务器正确接收，进入第二次握手。

第二次握手：服务器向客户端发送头部标志位为 SYN_ACK 的数据包

A. 此数据包校验和错误/中途丢失，客户端接收后直接丢弃，继续处于等待接收状态，不发送数据包——服务器在经过一段时间后收不到返回数据包，因此重发标志位为 SYN_ACK 的数据包。

B. 此数据包被客户端正确接收，进入第三次握手。

第三次握手：客户端向服务器发送头部标志位为 ACK 的数据包

A. 此数据包校验和错误/中途丢失，服务器接收后直接丢弃，继续处于等待接收状态，不发送数据包——客户端在经过一段时间后收不到返回数据包，因此重发标志位为 ACK 的数据包。

B. 此数据包被服务器正确接收，握手完成。

客户端建立连接函数伪代码：

```
int connect(SOCKET& client, SOCKADDR_IN& ClientAddr, int &len)
{
	//设置为非阻塞模式，避免卡在recvfrom
	//第一次握手，客户端发送SYN
	//记录第一次握手发送时间
	clock_t pack1starttime = clock();
	//第二次握手，等待服务器发来的SYN_ACK
	UDPpacket pack2;
	//缓冲区
	char* buffer2 = new char[sizeof(pack2)];
	//一直没有收到包，说明之前的包出问题了，要重新发送第一次握手
	while (1)
	{
		//如果超时则重传
		if (clock()-pack1starttime>= 2 * CLOCKS_PER_SEC)
		{
			//重传
		}
		flag = recvfrom(client, buffer2, sizeof(pack2), 0, (sockaddr
		    *)&ClientAddr, &len);
		//收到了包判断正确性
		if (pack2.head.flags == SYN_ACK && res == 0)
		{
			//跳出循环
			break;
		}
		//否则一直处于等待状态
	}
	//第三次握手 客户端发送ACK
	//发送第三个握手包，之后2CLOCKS_PER_SEC内若收到数据包则说明需要重传
	while ((clock() - checktime) <= 2* CLOCKS_PER_SEC)
	{
		//收到数据包则重传第三个握手包
	}
	iMode = 0; //0: 阻塞
	ioctlsocket(client, FIONBIO, (u_long FAR*) & iMode);//恢复成阻塞模式
	return 1;
}
```

服务器端建立连接函数：

```
int connect(SOCKET& server, SOCKADDR_IN& ServerAddr, int& len)
{
```

```
        //进入while循环接收第一次握手请求信息
        while (1)
        {
                //没收到包就一直循环
                //接收成功判断正确性
                if (pack1.head.flags == SYN && res == 0)
                {
                        //进入第二次握手
                        break;
                }
                //如果发来的包不正确的话，继续这个循环，等待客户端重新发送
        }
        //第二次握手，服务器发送SYN_ACK
    clock_t starttime = clock();//第二次握手发送时间

        //第三次握手 服务器接收
        UDPpacket pack3;
        //缓冲区
        char* buffer3 = new char[sizeof(pack3)];
        //如果一直没有收到包的话，说明上一个包需要重传
        while (1)
        {
                //超时重传
                //没收到包就一直循环
                //接收成功则跳出
                if (pack3.head.flags == ACK && res == 0)
                {
                        break;//跳出循环
                }
                //否则的话就一直等到能接收正确的包
        }
        //等两个时钟周期
        starttime = clock();
        while ((clock() - starttime) <= 2 * CLOCKS_PER_SEC)
        {
        }
}
```

（五） 断开连接

此处协议设计模拟了 TCP 的四次挥手协议。

第一次挥手：客户端向服务器发送头部标志位为 FIN 的数据包，如果超时未收到来自服务器发送的 FIN_ACK 包，则重发第一次挥手包 FIN。如果收到了来自服务器发送的正确的 FIN_ACK 包，则进入第三次握手。

第二次挥手：服务器收到客户端发来的第一次挥手包后，确认无误后发送第二次挥手包，其标志位为 FIN_ACK。等待两个 CLOCKS_PER_SEC，如果在此期间没有收到客户端发来的包的话则说明客户端正确接收，第二次挥手结束。如果收到了来自客户端的包则需要重传第二次挥

手。

第三、四次挥手机制与第一、二次相同，只是 FIN 发送端更改为服务器，FIN_ACK 为客户端。自此正式断开连接。

客户端断开连接函数伪代码实现:

```
int disconnect(SOCKET& client, SOCKADDR_IN& ClientAddr, int len)
{
        //第一次挥手, 客户端发送FIN
        //记录第一次挥手发送时间
        clock_t starttime = clock();
        //接收第二次挥手请求信息
        while (1)
        {
                if (超时)
                {
                        //重传第一次挥手包
                }
                if (未接收到)
                {
                        continue;
                }
                //接收成功进行判断
                if (pack2.head.flags == FIN_ACK && res == 0)
                {
                        //跳出循环
                        break;
                }
                //传来的包不正确则一直等待
        }
        //等待两个时钟周期, 跟server同步
        while (clock() - starttime <= CLOCKS_PER_SEC)
        {

        }
        //接收第三次挥手请求信息
        while (1)
        {
                flag = recvfrom(client, buffer3, sizeof(pack3), 0, (sockaddr
                    *)&ClientAddr, &len);
                if (flag <= 0)//未接收到
                {
                        continue;
                }
                //是第三次挥手
                if (pack3.head.flags == FIN && res == 0)
                {
                        //跳出循环
                        break;
```

```
                }
                //否则一直处于等待状态
        }
        //发送第四次挥手，客户端发送FIN_ACK
        starttime = clock();//第四次挥手发送时间
    //两个时钟周期内收到消息说明需要重传第四次挥手包
        while (clock() - starttime <= 2 * CLOCKS_PER_SEC)
        {
                //没收到
                if (recvfrom(client, buffer3, sizeof(pack1), 0, (sockaddr*)&
                    ClientAddr, &len) <= 0)
                {
                        continue;
                }
                else//收到了，说明第四次挥手没发出去，需要重新发送
                {
                        //重发，更新时间
                }
        }
        return 1;
}
```

服务器端断开连接函数伪代码:

```
    int disconnect(SOCKET& server, SOCKADDR_IN& ServerAddr, int& len)
{
        //接收第一次挥手请求信息
        while (1)
        {
                //没有收到消息则一直循环
                if ( flag <= 0)
                {
                        continue;
                }
                //接收成功
                //是第一次握手
                if (pack1.head.flags == FIN && res == 0)
                {
                        //跳出循环
                        break;
                }
                //否则就一直等传送来正确的包
        }
        //第二次挥手，服务器发送FIN_ACK
        clock_t starttime = clock();//记录第二次挥手发送时间
    //接下来的2CLOCKS_PER_SEC如果收到消息说明需要重发第二次挥手包
        while (clock() - starttime <= 2 * CLOCKS_PER_SEC)
        {
                //没收到则continue
```

```
                //收到了，说明第二次挥手没发出去，需要重新发送
                {
                        //发送挥手包
                        //更新发送时间
                        starttime = clock();
                }
        }

        //第三次挥手，服务器要发送FIN
        starttime = clock();//第三次挥手发送时间
        //接收第四次挥手请求信息
        UDPpacket pack4;
        //缓冲区
        char* buffer4 = new char[sizeof(pack4)];
        while (1)
        {
                if (超时)
                {
                        starttime = clock();
                        //重新发送第三次挥手
                }
                if (flag <= 0)//未收到消息
                {
                        continue;
                }
                //接收成功
                //是第四次挥手
                if (pack4.head.flags == FIN_ACK && res == 0)
                {
                        //跳出循环
                        break;
                }
                //否则的话一直循环等待
        }
        return 1;
}
```

(六) 差错检测

客户端在发送数据包时，先设置该数据包头部中的序列号 seq、数据长度 len、标志位 flags，将校验和 check 置 0，之后将数据部分用文件信息填充。之后调用 packetcheck 函数，其作用为将整个数据包按 16 位为一组进行划分，不足 16 位的添加 0 补齐，之后进行直接相加的运算（UDP 发送端真正校验和计算中是按照反码计算，但是由于数据本身就以反码形式计算，因此代码中直接相加），如果相加出现进位，则舍弃进位，所得结果 +1，最终返回 16 位计算结果的反码。

服务器在接收数据包时可以直接调用 packetcheck 函数计算校验结果是否为 0，如果为 0 则说明数据包无误（在 UDP 接收端中是将所有的数据以及包头部按照 16 位反码相加，其无误的结果应该为 1111 1111 1111 1111，但是由于 packetcheck 最终返回值取反，因此调用此函数后无

误结果应为 0)。

具体 packetcheck 函数实现如下:

```
u_short packetcheck(u_short* packet, int packelength)
{
	register u_long sum = 0;
	int count = (packelength + 1) / 2;//两个字节的计算
	u_short* buf = (u_short*)malloc(packelength + 1);
	memset(buf, 0, packelength + 1);
	memcpy(buf, packet, packelength);
	//cout << "count=" << packelength << endl;
	while (count--)
	{
		sum += *buf++;
		if (sum & 0xFFFF0000)
		{
			sum &= 0xFFFF;
			sum++;
		}
	}
	return ~(sum & 0xFFFF);
}
```

(七) 性能测试

建立连接的三次握手包发送接受结果:

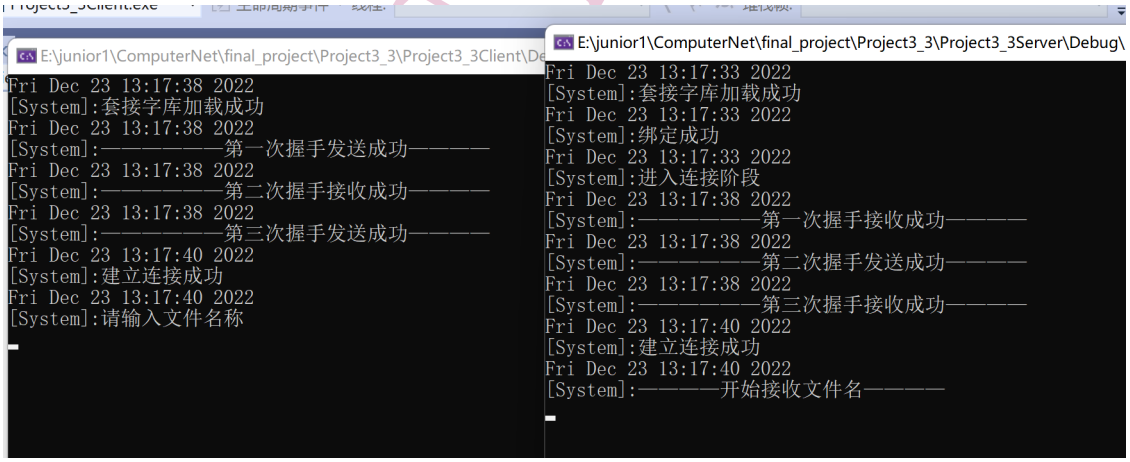

图 2: 三次握手结果

挥手结果图如下:

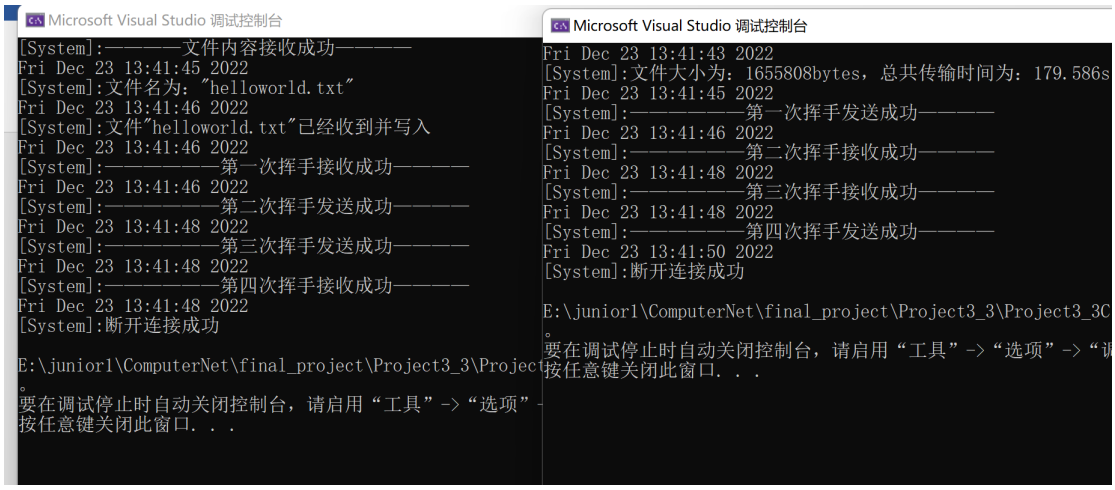

图 3: 四次挥手结果

窗口变更图如下:

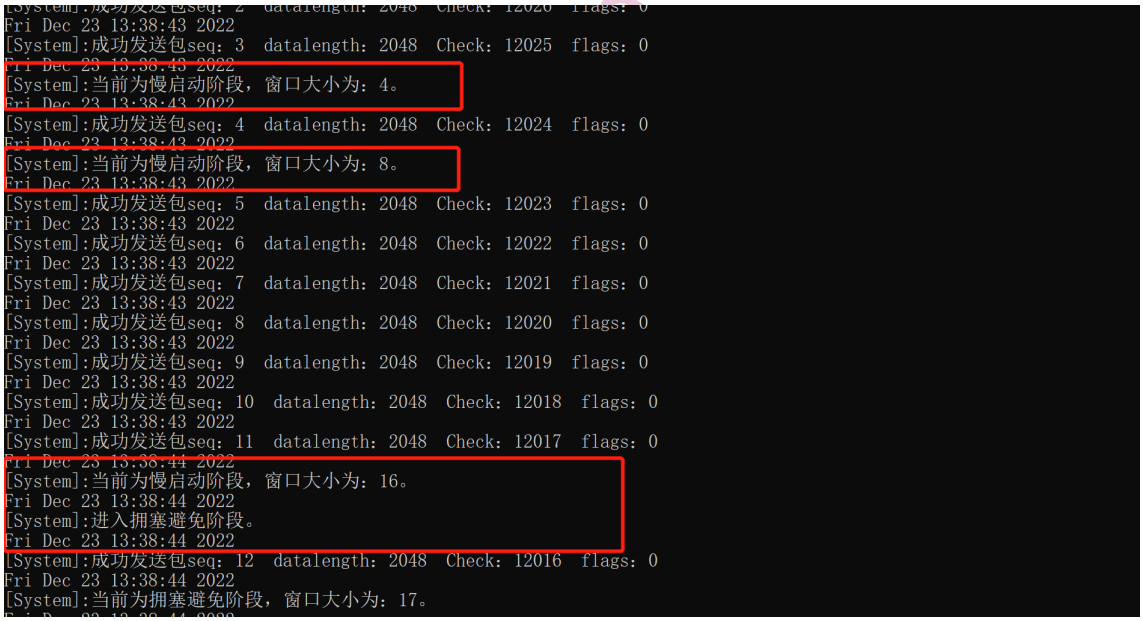

图 4: 窗口变更结果

在使用用路由器情况下设置丢包率 2%，延时为 1ms

helloworld.txt 文件传输结果如下，共传输时间为：179.586s，平均吞吐率为：73761.1bps

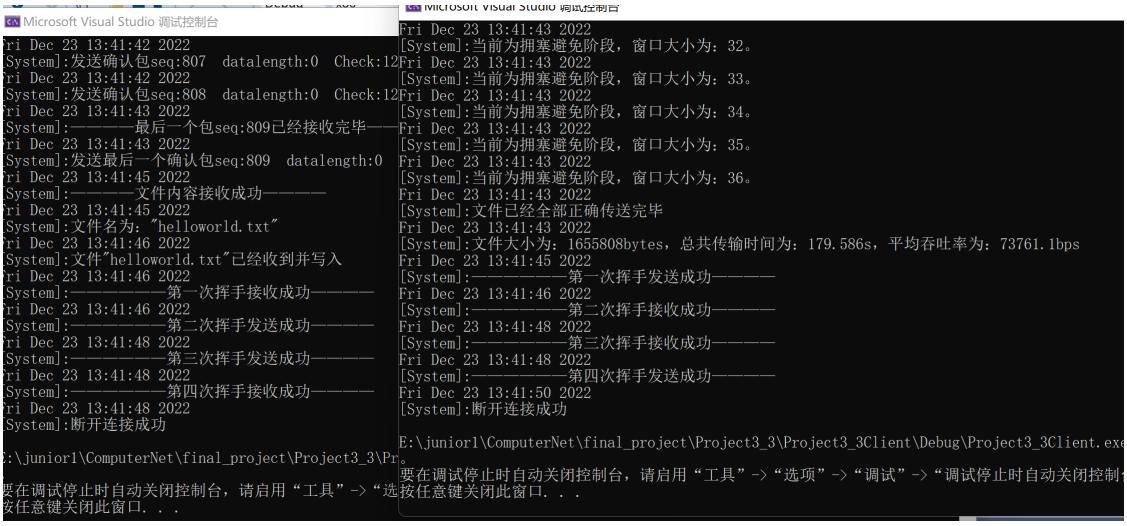

图 5: txt 文件传输结果

第一张图片传输结果如下，共传输 177.932s，平均吞吐率 83508.4bps

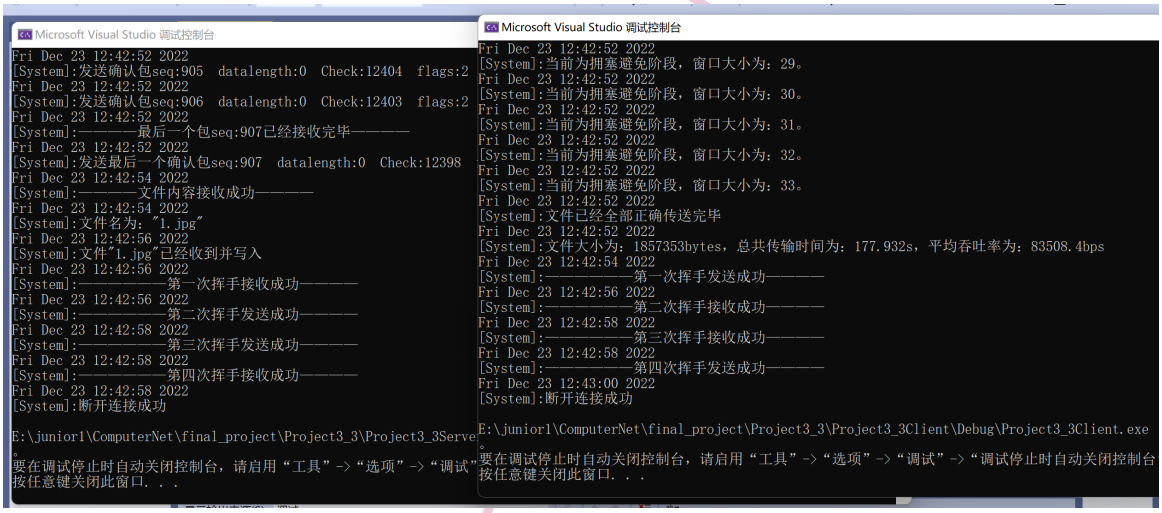

图 6: 图片一传输结果

第二张图片传输结果如下，共传输 534.273s，平均吞吐率 88322bps

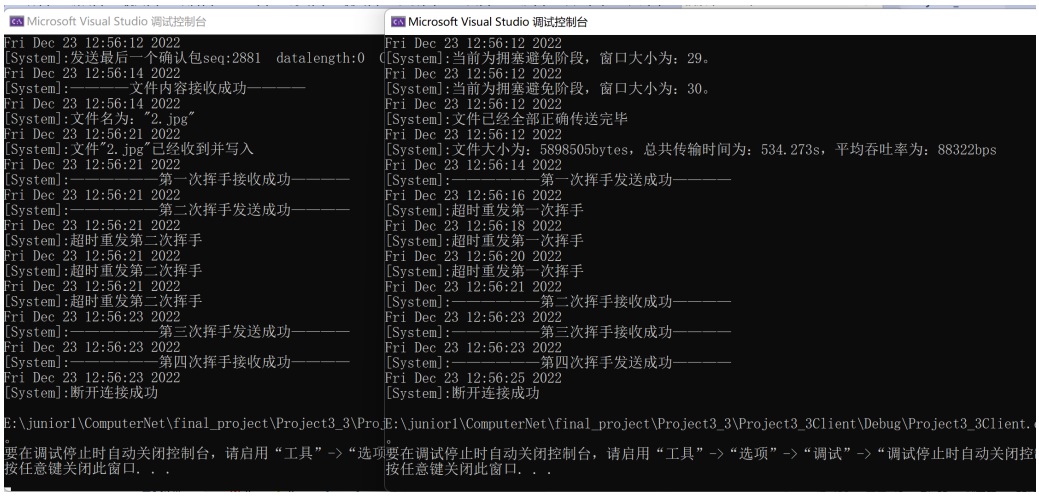

图 7: 图片二传输结果

第三张图片传输结果如下，传输时间 1120.87s，平均吞吐率为：85426.4bps

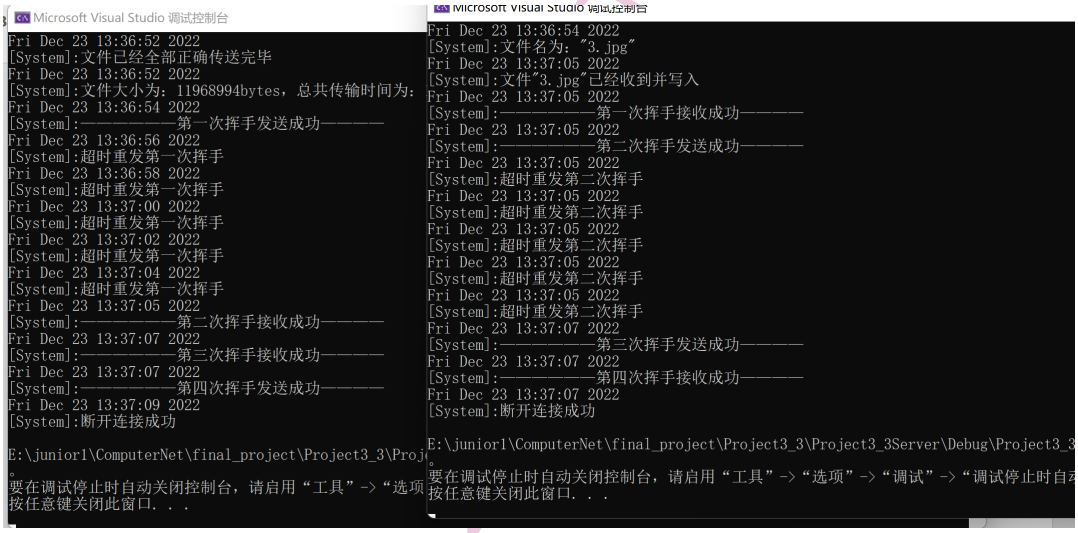

图 8: 图片三传输结果

最终传输结果如下，可以看到图片均正确传输，而传输结果图片打开后与源文件相同，可知传送成功。

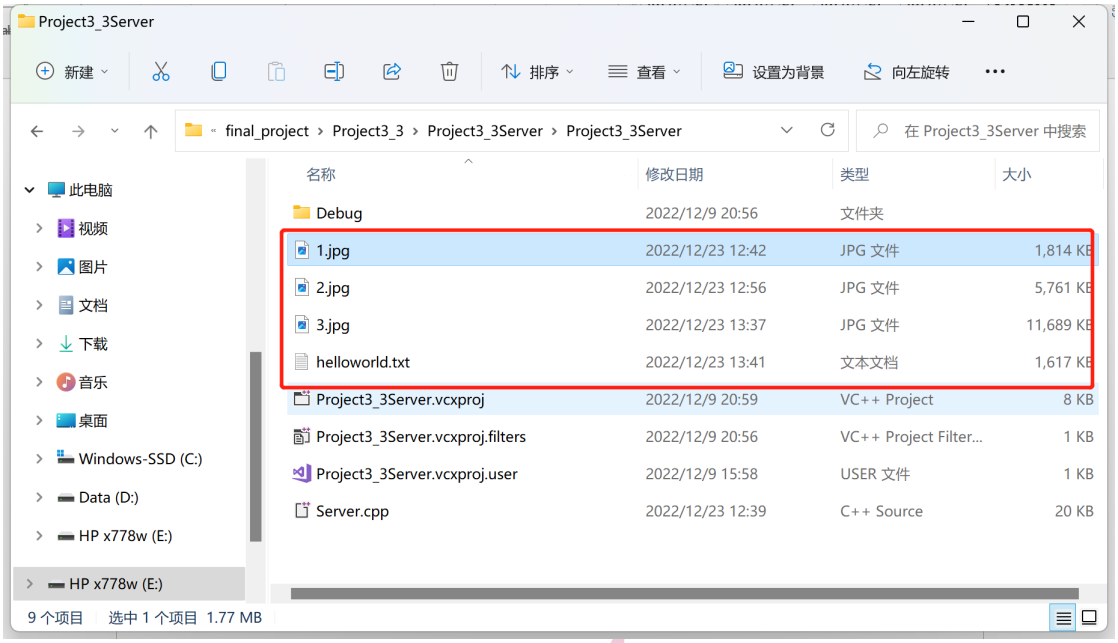

图 9: 本地接收结果 1

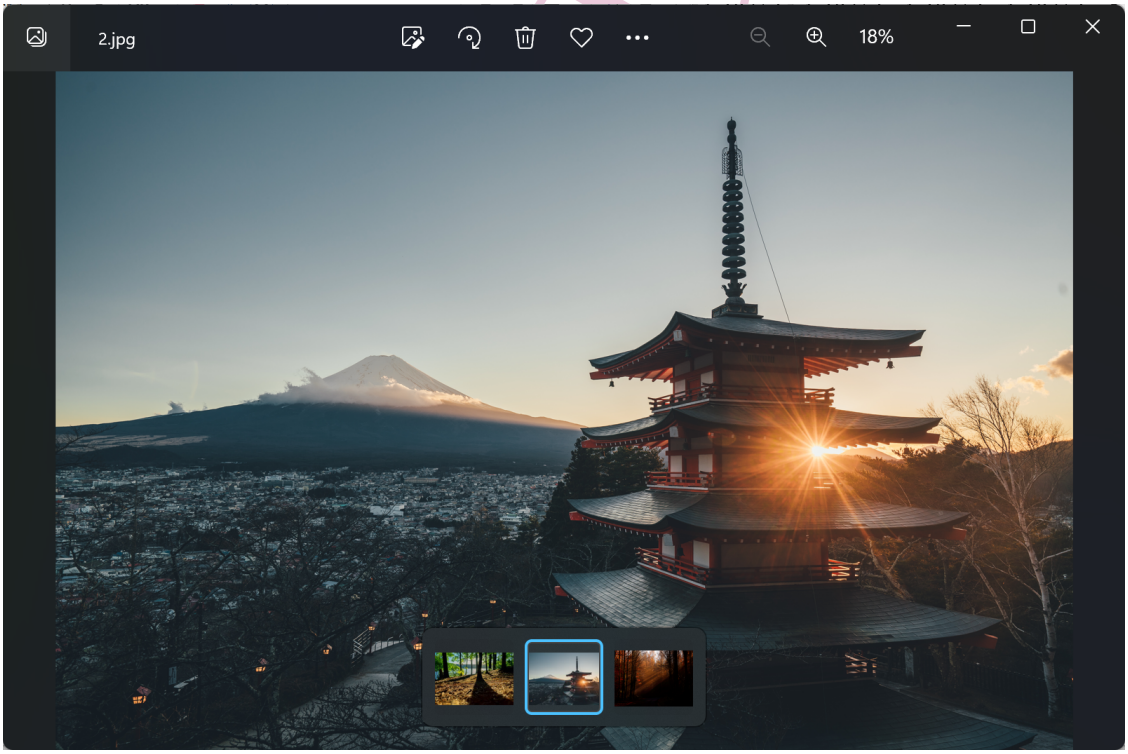

图 10: 本地接收结果 2