



南開大學  
Nankai University

计算机学院  
计算机网络实验报告

实验 3-3 滑动窗口与选择确认

姓名：赵康明

学号：2110937

专业：计算机科学与技术

# 目录

<b>1 实验要求</b>	<b>2</b>
<b>2 协议设计</b>	<b>2</b>
2.1 选择重传 (SR)	2
<b>3 程序设计实现</b>	<b>3</b>
3.1 发送方	3
3.1.1 发送缓冲区	3
3.1.2 发送数据包	3
3.1.3 超时重传线程	4
3.1.4 接收 ACK 数据包线程	5
3.2 接收方	7
3.2.1 接收缓冲区	7
3.2.2 接收数据包	7
<b>4 实验结果</b>	<b>9</b>
4.1 乱序接收	9
4.2 超时重传检验	9
4.3 文件传输实验结果	10
<b>5 实验总结与思考</b>	<b>12</b>

## 1 实验要求

在实验 3-1 的基础上，将停等机制改成基于滑动窗口的流量控制机制，发送窗口和接收窗口采用相同大小，支持选择确认，完成给定测试文件的传输。

1. 实现单向数据传输（一端发数据，一端返回确认）。
2. 对于每个任务要求给出详细的协议设计。
3. 完成给定测试文件的传输，显示传输时间和平均吞吐率。
4. 性能测试指标：吞吐率、延时，给出图形结果并进行分析。
5. 完成详细的实验报告（每个任务完成一份，主要包含自己的协议设计、实现方法、遇到的问题、实验结果，不要抄写太多的背景知识）。
6. 编写的程序应该结构清晰，具有较好的可读性。
7. 提交程序源码、可执行文件和实验报告。

## 2 协议设计

本次实验要求我们在基于滑动窗口的基础上进行改进。在实验 3-2 中，我们实现了基于滑动窗口的流量控制机制，发送窗口大小可设定，接收方的窗口固定为 1，采取累积确认，即接收方只能按照接收顺序进行接收。

而在本次实验中我们将设置发送方窗口和接收方窗口大小一致，把累积确认修改为选择确认，实现新的传输机制。

### 2.1 选择重传 (SR)

选择重传的机制如下：

1. 窗口机制：发送方和接收方都维护一个窗口。发送方窗口内的帧可以连续发送，无需等待每个帧的确认。接收方窗口内的帧是它期待接收的帧。
2. 帧编号：每个发送的帧都有一个唯一的序列号。
3. 接收确认：接收方对接收到的每个帧单独发送确认（ACK）。
4. 非连续接收：接收方可以接收并确认非连续的帧。例如，如果接收方已经接收了序列号为 1, 2, 4 的帧，它会单独确认这些帧，而不是等待序列号为 3 的帧。
5. 重传：如果发送方没有在预定时间内接收到某个帧的确认，它会重传该帧。
6. 缓存：接收方需要有足够的缓存空间来存储乱序到达的帧。

实验原理图：

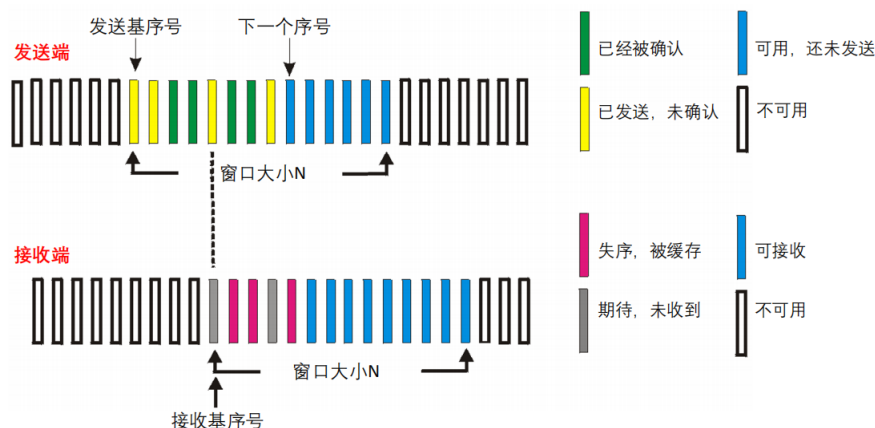


图 2.1: 乱序接收

### 3 程序设计实现

#### 3.1 发送方

##### 3.1.1 发送缓冲区

发送方首先需要有一个缓冲区，用于保存那些已经发送但未确认的数据包，同时还需要对每个数据包有一个单独的计时器，因此我在原有的 **Packet 数据包类** 和 **timer 计时器类** 的基础上新增了一个类型，即 **Packet\_timer** 类。每个数据包发送后都将启动该数据包的计时器。

```

1 struct packet_timer
2 {
3     Packet* packet;
4     timer p_timer;
5     bool isacked;
6     packet_timer(Packet* packet) : packet(packet), p_timer() {
7         p_timer.start_timer();
8         isacked = false;
9     }
10 };
11 vector<packet_timer*> packet_timer_vector;

```

##### 3.1.2 发送数据包

发送数据包和 3-2 实验中的类似，即在窗口有剩余位置时进行发送数据包，并将发送的数据包保存到缓冲区当中。同时启动数据包的计时器

```

1 packet_timer *pt=new packet_timer(packet);
2 packet_timer_vector.push_back(pt);

```

### 3.1.3 超时重传线程

为了计算缓冲区中的各个数据包是否反馈超时，我设计添加了一个计时线程，不断计算缓冲区中的各个包是否超时，如果超时，则启动超时重发。

```
1 void time_out_thread()
2 {
3     while (true)
4     {
5         if (send_over)
6         {
7             print_lock.lock();
8             cout << "timeout线程结束" << endl;
9             print_lock.unlock();
10            return;
11        }
12        buffer_lock.lock();
13        for (auto packet_timer : packet_timer_vector)
14        {
15            if (packet_timer->isacked == false && packet_timer->p_timer.time_out())
16            {
17                print_lock.lock();
18                cout << "重传";
19                print_lock.unlock();
20                sendto(client, (char*)packet_timer->packet, sizeof(HeadMsg) +
21                    packet_timer->packet->header.len, 0, (sockaddr*)&router_addr,
22                    sizeof(SOCKADDR_IN));
23                print_lock.lock();
24                cout << "超时重传数据包, 首部为: seq:" <<
25                    packet_timer->packet->header.seq << ", ack:" <<
26                    packet_timer->packet->header.ack << ", flag:" <<
27                    packet_timer->packet->header.flag << ", checksum:" <<
28                    packet_timer->packet->header.checkSum << ", len:" <<
29                    packet_timer->packet->header.len << endl;
30                print_lock.unlock();
31            }
32        }
33        buffer_lock.unlock();
34        if (send_over)
35        {
36            print_lock.lock();
37            cout << "timeout线程结束" << endl;
38        }
39    }
40 }
```

```
32     print_lock.unlock();
33     return;
34 }
35 Sleep(200);
36 }
37 }
```

### 3.1.4 接收 ACK 数据包线程

- 遍历缓冲区当中的数据包，查看收回到的 ACK 是否和数据包的序列号相等，相等的话将停止单个数据包的计时，并将标志位 isacked 置为 1。
- 如果当前收到的 ACK 序列号恰好等于 base, 那么将去遍历缓冲区中连续 acked 为 1 的数据包的个数，将 base 更新为 base+ 连续确认的个数。

```
1 void receive_thread() {
2     ioctlsocket(client, FIONBIO, &unlockmode);
3     char* recv_buffer = new char[sizeof(HeadMsg)];
4     HeadMsg* header;
5     while (true) {
6         if (send_over) {
7             ioctlsocket(client, FIONBIO, &lockmode);
8             delete[]recv_buffer;
9             print_lock.lock();
10            cout << "recv线程结束"<<endl;
11            print_lock.unlock();
12            return;
13        }
14        while (recvfrom(client, recv_buffer, sizeof(HeadMsg), 0,
15            (sockaddr*)&router_addr, &rlen) <= -1) {
16            if (send_over) {
17                ioctlsocket(client, FIONBIO, &lockmode);
18                delete[]recv_buffer;
19                print_lock.lock();
20                cout << "recv线程结束" << endl;
21                print_lock.unlock();
22                return;
23            }
24            header = (HeadMsg*)recv_buffer;
25            int chksum = cal_ck_sum((u_short*)recv_buffer, sizeof(HeadMsg));
26            if (chksum != 0) {
```

```
27         continue;
28     }
29     else if (header->flag == ACK) {
30         int ack_num = header->seq;
31         for (auto packet_timer : packet_timer_vector)
32         {
33             buffer_lock.lock();
34             //收到的ack_num是对应包的seq,那么就去找到缓冲区中的包
35             //把他的isacked=true;
36             if (packet_timer->packet->header.seq == ack_num)
37             {
38                 // 停止计时
39                 packet_timer->p_timer.stop_timer();
40                 packet_timer->isacked = true;
41             }
42             buffer_lock.unlock();
43         }
44         // 如果收到的ack是base 更新base 怎么更新 应该是连续更新已经接受了的
45         if (base == ack_num)
46         {
47             buffer_lock.lock();
48             // 对缓冲区从第一个开始遍历 并且从缓冲区中删除
49             // , 连续删除的个数即为base更新的
50             int acked = 0;
51             while (packet_timer_vector.size())
52             {
53                 if (packet_timer_vector.front()->isacked)
54                 {
55                     acked++;
56                     delete packet_timer_vector.front(); // 删除第一个元素
57                     packet_timer_vector.erase(packet_timer_vector.begin());
58                 }
59                 else
60                 {
61                     break;
62                 }
63             }
64             // 更新base
65             base = base + acked;
66             buffer_lock.unlock();
67         }
68     }
69     print_lock.lock();
```

```
67     cout << "接收到来自服务器的数据包，首部为: seq:" << header->seq << ",  
        ack:" << header->ack << ", flag:" << header->flag << ", checksum:" <<  
        header->checkSum << ", len:" << header->len << ", 剩余窗口大小:" <<  
        WND - (nextseqnum - base) << endl;  
68     print_lock.unlock();  
69 }  
70 }  
71 }
```

## 3.2 接收方

接收方和之前有所不同的地方在于，接收方也拥有了一个缓冲区，用于存储那些乱序到达的数据包，因此本次实验为其多添加了一个缓冲区，用于缓存落在接收窗口范围内的数据包，并多设计了一个线程，用于将缓冲区中的乱序数据包按序拷贝到本地，同时更新接收窗口的基址 `recv_base`。

### 3.2.1 接收缓冲区

设计了一个结构体，用于保存收到的数据包

```
1 struct Header_Recv  
2 {  
3     Packet* packet;  
4     bool isRecv;  
5  
6     Header_Recv(Packet* packet) : packet(packet), isRecv(true) {}  
7 };  
8 vector<Header_Recv*> Head_recv_vector;
```

### 3.2.2 接收数据包

与 3-2 实验不同的是，我们在接收到数据包之后，对其进行校验，并返回对应数据包的 ACK 报文。区别在于，之前我们都是按序接收，因此收到报文后拷贝到本地即可。但现在我们是乱序接收，因此我们需要先保存到缓冲区中。为了解决保存到本地的问题，我设计了另一个线程，遍历缓冲区，更新接收缓冲区的基址 `Recv_base`，并且按序将数据包的内容拷贝到本地。

#### 保存数据包到缓冲区

```
1 ioctlsocket(server, FIONBIO, &unblockmode);  
2     while (recvfrom(server, recvbuffer, sizeof(header) + MAX_DATA_LENGTH, 0,  
        (sockaddr*)&router_addr, &rlen) <= 0) {}  
3     memcpy(&header, recvbuffer, sizeof(header));  
4     if (header.seq >= recv_base && header.seq <= recv_base + WND &&  
        cal_ck_sum((u_short*)recvbuffer, sizeof(header) + header.len) == 0)  
5     {
```



```
6         print_lock.lock();
7         cout << "成功接收窗口内数据包";
8         cout << "成功接收数据包, 序列号: " << header.seq << " ACK: " <<
            header.ack << " 校验和: " << header.checkSum << endl;
9         print_lock.unlock();
10        buffer_lock.lock();
11        Packet* packet=new Packet(header);
12        //memcpy(&packet->header, recvbuffer, sizeof(header));
13        memcpy(&packet->Msg, recvbuffer + sizeof(header), header.len);
14        // 保存缓冲区
15        Header_Recv* hr = new Header_Recv(packet);
16        Head_recv_vector.push_back(hr);
17        buffer_lock.unlock();
```

下列函数用于更新基址和按序拷贝数据包到本地。

#### 更新基址

```
1 void update_recv_base()
2 {
3     while (true) {
4         buffer_lock.lock();
5         // 使用迭代器遍历容器
6         for (auto it = Head_recv_vector.begin(); it != Head_recv_vector.end(); /* no
            increment here */) {
7             Header_Recv* hd_rec = *it;
8             // 如果缓存中有当前基址的数据包, 那么更新recv_base并删除该元素
9             if (hd_rec->packet->header.seq == recv_base) {
10                 recv_base++;
11                 memcpy(message + messagepointer,
12                        hd_rec->packet->Msg,hd_rec->packet->header.len);
13                 messagepointer += hd_rec->packet->header.len;
14                 it = Head_recv_vector.erase(it); // 删除元素, 并将迭代器指向下一个元素
15                 delete hd_rec; // 删除动态分配的资源 (如果有的话)
16             }
17             else
18             {
19                 // 如果条件不满足, 只需递增迭代器
20                 ++it;
21             }
22         }
23         buffer_lock.unlock();
24         if (file_finish)
```

```
24 {
25     print_lock.lock();
26     cout << "update_线程结束" << endl;
27     print_lock.unlock();
28     return;
29 }
30 //Sleep(200);
31 }
32 }
```

## 4 实验结果

此处展示实验结果均为传输 1.jpg 图片，发送窗口和接收窗口均为 8，丢包率为 10%，时延为 1ms 的情况下测试得出。

### 4.1 乱序接收

图中展示的是接收端的结果，可以发现接收端实现了乱序接收。

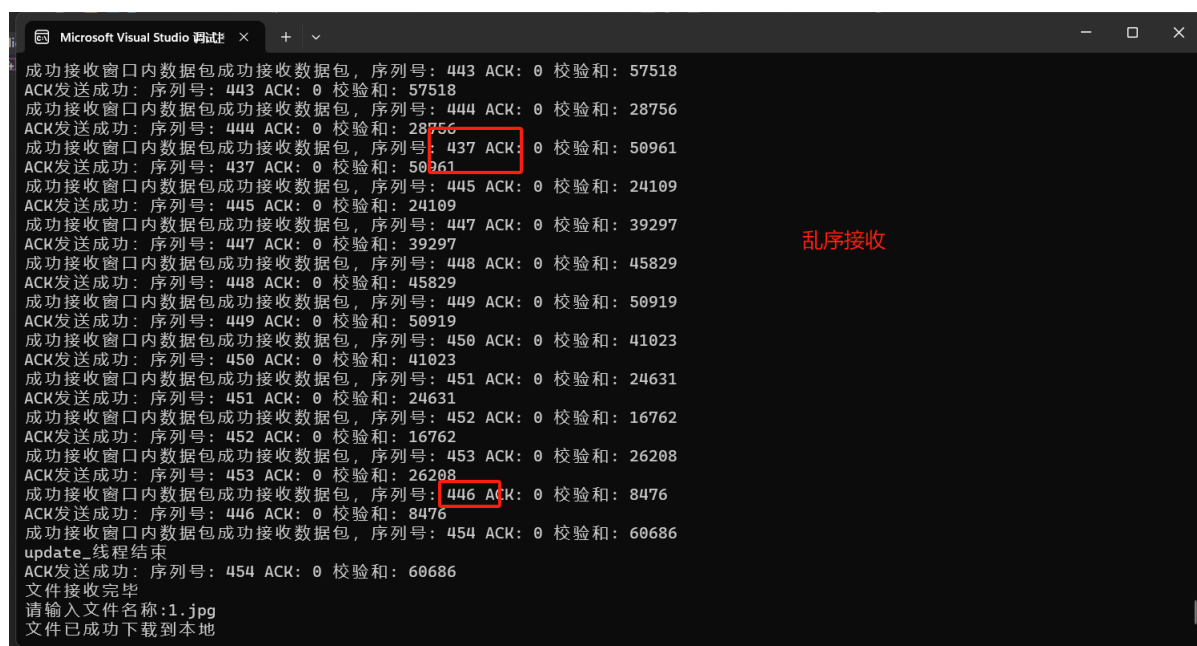


图 4.2: 乱序接收

### 4.2 超时重传检验

图中发现，发回关于序列号为 428 报文的 ACK 报文在返回途中被丢失了，于是在超时之后将其重新发送，并重新开启计时，此时接收端的接受窗口的基址一直无法移动，因为 428 恰好卡在了窗口边界限上，因此在重发 428 的报文之后，迅速返回关于 428 的 ACK 报文，此时滑动窗口迅速调整，重置为 8，之后便可以持续发送数据包。此处证明超时重传检验成功。

```

Microsoft Visual Studio 调试
向服务器发送数据包 seq:427 ack: 0 type: 0 checksum: 20088
向服务器发送数据包 seq:428 ack: 0 type: 0 checksum: 42251
向服务器发送数据包 seq:429 ack: 0 type: 0 checksum: 34561
向服务器发送数据包 seq:430 ack: 0 type: 0 checksum: 46850
向服务器发送数据包 seq:431 ack: 0 type: 0 checksum: 39569
向服务器发送数据包 seq:432 ack: 0 type: 0 checksum: 58386
向服务器发送数据包 seq:433 ack: 0 type: 0 checksum: 60761
向服务器发送数据包 seq:434 ack: 0 type: 0 checksum: 53728
接收来自服务器的数据包, 首部为: seq:427, ack:427, flag:2, checksum:64679, len:0, 剩余窗口大小:1
向服务器发送数据包 seq:435 ack: 0 type: 0 checksum: 12965
接收来自服务器的数据包, 首部为: seq:429, ack:429, flag:2, checksum:64675, len:0, 剩余窗口大小:0
接收来自服务器的数据包, 首部为: seq:430, ack:430, flag:2, checksum:64673, len:0, 剩余窗口大小:0
接收来自服务器的数据包, 首部为: seq:431, ack:431, flag:2, checksum:64671, len:0, 剩余窗口大小:0
接收来自服务器的数据包, 首部为: seq:432, ack:432, flag:2, checksum:64669, len:0, 剩余窗口大小:0
接收来自服务器的数据包, 首部为: seq:433, ack:433, flag:2, checksum:64667, len:0, 剩余窗口大小:0
接收来自服务器的数据包, 首部为: seq:434, ack:434, flag:2, checksum:64665, len:0, 剩余窗口大小:0
接收来自服务器的数据包, 首部为: seq:435, ack:435, flag:2, checksum:64663, len:0, 剩余窗口大小:0
重传超时重传数据包, 首部为: seq:428, ack:0, flag:0, checksum:42251, len:4096
向服务器发送数据包 seq:428 ack: 0 type: 0 checksum: 42251
向服务器发送数据包 seq:436 ack: 0 type: 0 checksum: 47360
向服务器发送数据包 seq:437 ack: 0 type: 0 checksum: 50961
向服务器发送数据包 seq:438 ack: 0 type: 0 checksum: 52995
向服务器发送数据包 seq:439 ack: 0 type: 0 checksum: 6537
向服务器发送数据包 seq:440 ack: 0 type: 0 checksum: 6029
向服务器发送数据包 seq:441 ack: 0 type: 0 checksum: 5774
向服务器发送数据包 seq:442 ack: 0 type: 0 checksum: 24323
向服务器发送数据包 seq:443 ack: 0 type: 0 checksum: 57518
接收来自服务器的数据包, 首部为: seq:436, ack:436, flag:2, checksum:64661, len:0, 剩余窗口大小:1
向服务器发送数据包 seq:444 ack: 0 type: 0 checksum: 28756
接收来自服务器的数据包, 首部为: seq:438, ack:438, flag:2, checksum:64657, len:0, 剩余窗口大小:0
  
```

图 4.3: 超时重传

### 4.3 文件传输实验结果

```

ACK发送成功: 序列号: 446 ACK: 0 校验和: 47360
成功接收窗口内数据包成功接收数据包, 序列号: 446
update_线程结束
ACK发送成功: 序列号: 454 ACK: 0 校验和: 57518
文件接收完毕
请输入文件名称:1.jpg
文件已成功下载到本地

recv线程结束
timeout线程结束
*****传输日志*****
*****
本次传输报文总长度: 1859584字节
共有: 7265个报文段分别转发
本次传输时间: 31.743秒
本次传输吞吐率: 58582.5字节/秒
  
```

图 4.4: 1.jpg 传输结果

```

recv线程结束
timeout线程结束
*****传输日志*****
*****
本次传输报文总长度: 5902336字节
共有: 23057个报文段分别转发
本次传输时间: 100.538秒
本次传输吞吐率: 58707.5字节/秒

C:\Users\zhaokangming\source\repos\Client\Debug>
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“在调试停止时自动关闭控制台”, 按任意键关闭此窗口...

成功接收窗口内数据包成功接收数据包, 序列号: 1440
ACK发送成功: 序列号: 1440 ACK: 0 校验和: 57518
成功接收窗口内数据包成功接收数据包, 序列号: 1440
update_线程结束ACK发送成功: 序列号: 1440
文件传输结束

请输入文件名称:2.jpg
文件已成功下载到本地

C:\Users\zhaokangming\source\repos\Server\Debug>
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“在调试停止时自动关闭控制台”, 按任意键关闭此窗口...
  
```

图 4.5: 2.jpg 传输结果

```

*****传输日志*****
*****
本次传输报文总长度: 11972608字节
共有: 46769个报文段分别转发
本次传输时间: 198.234秒
本次传输吞吐率: 60396.3字节/秒

C:\Users\zhaokangming\source\repos\Client\Debug>
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“在调试停止时自动关闭控制台”, 按任意键关闭此窗口...

服务端: 第三次挥手发送成功
服务器: 收到第四次挥手消息
成功发送确认报文
断开连接
请输入文件名称:3.jpg
文件已成功下载到本地

C:\Users\zhaokangming\source\repos\Server\Debug>
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“在调试停止时自动关闭控制台”, 按任意键关闭此窗口...
  
```

图 4.6: 3.jpg 传输结果

```
2 0
*****传输日志*****
*****
本次传输报文总长度：1658880字节
共有：6481个报文段分别转发
本次传输时间：27.588秒
本次传输吞吐率：60130.5字节/秒

C:\Users\zhaokangming\source\repos\Client\Debug\Client
而在调试停止时自动关闭控制台，请使用“工具”->“选项”->“
文件传输结果
服务器：收到第一次挥手信息
服务端：第二次挥手发送成功
服务端：第三次挥手发送成功
服务器：收到第四次挥手消息
成功发送确认报文
断开连接
请输入文件名称:helloworld.txt
文件已成功下载到本地
```

图 4.7: helloworld.txt 传输结果

查看左右图对比我们可以发现，两边文件大小一致，说明没有文件传输损失。

名称	修改日期	类型	大小
Debug	2023/12/13 20:11	文件夹	
1.jpg	2023/11/1 11:35	JPG 图片文件	1,814 KB
2.jpg	2023/11/1 11:35	JPG 图片文件	5,761 KB
3.jpg	2023/11/1 11:35	JPG 图片文件	11,689 KB
client.cpp	2023/12/13 20:11	C++ Source	24 KB
Client.sln	2023/11/14 19:49	Visual Studio Sol...	2 KB
Client.vcxproj	2023/11/15 17:43	VC++ Project	8 KB
Client.vcxproj.filters	2023/11/15 17:43	VC++ Project Fil...	1 KB
Client.vcxproj.user	2023/11/14 19:49	Per-User Project...	1 KB
helloworld.txt	2023/11/1 11:35	文本文档	1,617 KB

名称	修改日期	类型	大小
Debug	2023/12/13 20:11	文件夹	
1	2023/12/11 18:59	文件	0 KB
1.jpg	2023/12/13 20:11	JPG 图片文件	1,814 KB
2.jpg	2023/12/13 19:28	JPG 图片文件	5,761 KB
3.jpg	2023/12/13 20:15	JPG 图片文件	11,689 KB
helloworld.txt	2023/12/13 20:18	文本文档	1,617 KB
Server.cpp	2023/12/13 20:11	C++ Source	20 KB
Server.sln	2023/11/13 21:12	Visual Studio Sol...	2 KB
Server.vcxproj	2023/11/13 20:24	VC++ Project	8 KB
Server.vcxproj.filters	2023/11/13 20:24	VC++ Project Fil...	1 KB
Server.vcxproj.user	2023/11/13 20:24	Per-User Project...	1 KB

图 4.8: 文件大小传输结果

再点开传输的各个文件，发现文件显示正常，说明数据包拷贝无误。

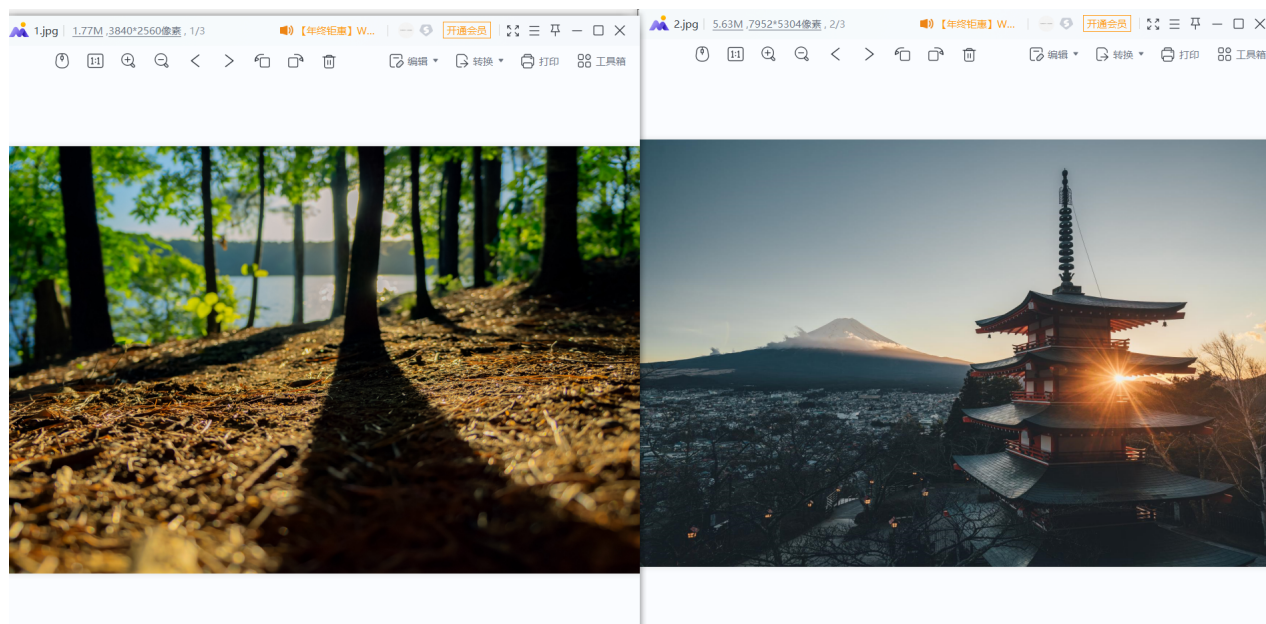


图 4.9: 文件实际传输结果

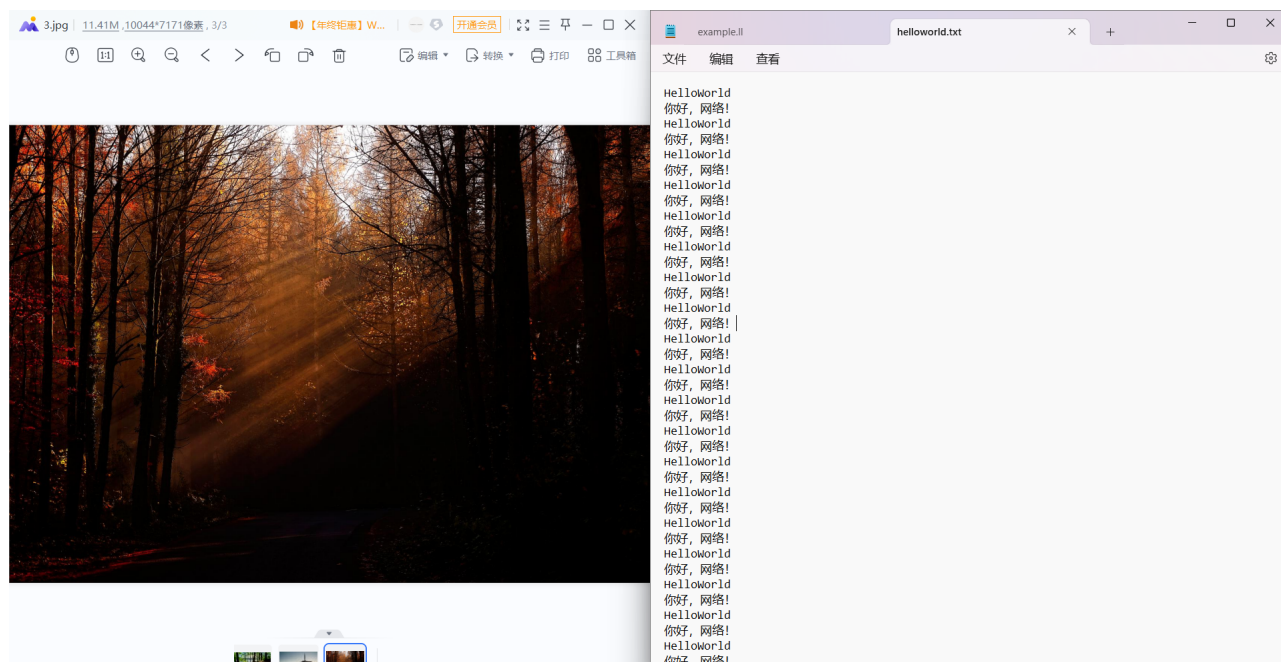


图 4.10: 文件实际传输结果

## 5 实验总结与思考

本次实验成功实现了滑动窗口和选择重传，成功的实现了接收方的乱序接收，相较于滑动窗口和累积确认有明显的效率提升。而本次实验认为的不足之处在于，发送方和接收方的缓冲区均为多个线程共享，本次实验在处理共享变量的过程中采取了信号控制的方式，个人认为在切换线程访问共享资源的过程中会对传输效率有所影响，希望在 3-4 中能有所改善，并比较停等协议、滑动窗口和选择重传的效率。