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

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：

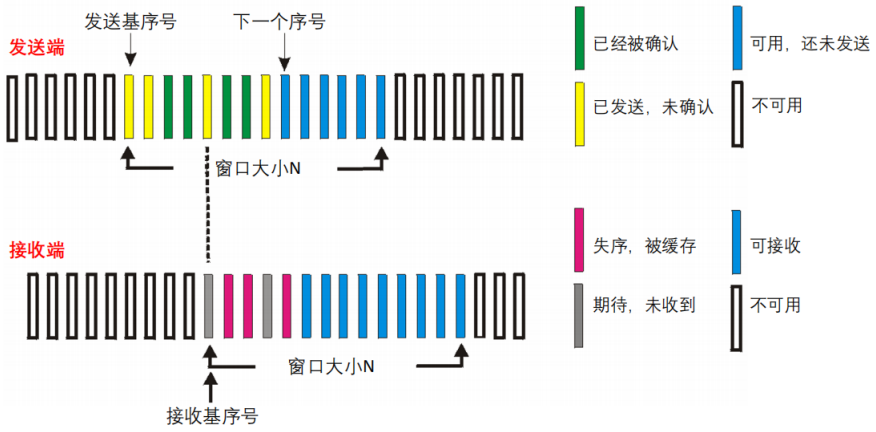

图 2.1: 乱序接收

3 程序设计实现

3.1 发送方

3.1.1 发送缓冲区

发送方首先需要一个缓冲区，用于保存那些已经发送但未确认的的数据包，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。

```
struct packet_timer
{
    Packet* packet;
    timer p_timer;
    bool isacked;
    packet_timer(Packet* packet) : packet(packet), p_timer() {
        p_timer.start_timer();
        isacked = false;
    }
};
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

```
packet_timer *pt=new packet_timer(packet);
   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。

```
void time_out_thread()
{
    while (true)
    {
        if (send_over)
        {
            print_lock.lock();
            cout << "timeout线程结束" << endl;
            print_lock.unlock();
            return;
        }
        buffer_lock.lock();
        for (auto packet_timer : packet_timer_vector)
        {
            if (packet_timer->isacked == false && packet_timer->p_timer.time_out())
            {
                    print_lock.lock();
                    cout << "重传";
                    print_lock.unlock();
                    sendto(client, (char*)packet_timer->packet, sizeof(HeadMsg) +
                        packet_timer->packet->header.len, 0, (sockaddr*)&router_addr,
                        sizeof(SOCKADDR_IN));
                    print_lock.lock();
                    cout << "超时重传数据包，首部为: seq:" <<
                        packet_timer->packet->header.seq << ", ack:" <<
                        packet_timer->packet->header.ack << ", flag:" <<
                        packet_timer->packet->header.flag << ", checksum:" <<
                        packet_timer->packet->header.checkSum << ", len:" <<
                        packet_timer->packet->header.len << endl;
                    print_lock.unlock();

            }
        }
        buffer_lock.unlock();
        if (send_over)
        {
            print_lock.lock();
            cout << "timeout线程结束" << endl;
```

```
            print_lock.unlock();
            return;
        }
        Sleep(200);
    }
}
```

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。

```
void receive_thread() {
    ioctlsocket(client, FIONBIO, &unlockmode);
    char* recv_buffer = new char[sizeof(HeadMsg)];
    HeadMsg* header;
    while (true) {
        if (send_over) {
            ioctlsocket(client, FIONBIO, &lockmode);
            delete[]recv_buffer;
            print_lock.lock();
            cout << "recv线程结束"<<endl;
            print_lock.unlock();
            return;
        }
        while (recvfrom(client, recv_buffer, sizeof(HeadMsg), 0,
            (sockaddr*)&router_addr, &rlen) <= -1) {
            if (send_over) {
                ioctlsocket(client, FIONBIO, &lockmode);
                delete[]recv_buffer;
                print_lock.lock();
                cout << "recv线程结束" << endl;
                print_lock.unlock();
                return;
            }
        }
        header = (HeadMsg*)recv_buffer;
        int chksum = cal_ck_sum((u_short*)recv_buffer, sizeof(HeadMsg));
        if (chksum != 0) {
```

```
            continue;
        }
        else if (header->flag == ACK) {
            int ack_num = header->seq;
            for (auto packet_timer : packet_timer_vector)
            {
                buffer_lock.lock();
                //收到的ack_num是对应包的seq,那么就去找到缓冲区中的包
                    把他的isacked=true;
                if (packet_timer->packet->header.seq == ack_num)
                {
                    // 停止计时
                    packet_timer->p_timer.stop_timer();
                    packet_timer->isacked = true;
                }
                buffer_lock.unlock();
            }
            // 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
            if (base == ack_num)
            {
                buffer_lock.lock();
                // 对缓冲区从第一个开始遍历 并且从缓冲区中删除
                    , 连续删除的个数即为base更新的
                int acked = 0;
                while (packet_timer_vector.size())
                {
                    if (packet_timer_vector.front()->isacked)
                    {
                        acked++;
                        delete packet_timer_vector.front(); // 删除第一个元素
                        packet_timer_vector.erase(packet_timer_vector.begin());
                    }
                    else
                    {
                        break;
                    }
                }
                // 更新base
                base = base + acked;
                buffer_lock.unlock();
            }
            print_lock.lock();
```

```
            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;
            print_lock.unlock();
        }
    }
}
```

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

```
struct Header_Recv
{
    Packet* packet;
    bool isRecv;

    Header_Recv(Packet* packet) : packet(packet), isRecv(true) {}
};
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

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

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

```
void update_recv_base()
{
    while (true) {
        buffer_lock.lock();
        // 使用迭代器遍历容器
        for (auto it = Head_recv_vector.begin(); it != Head_recv_vector.end(); /* no
            increment here */) {
            Header_Recv* hd_rec = *it;
            // 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
            if (hd_rec->packet->header.seq == recv_base) {
                recv_base++;
                memcpy(message + messagepointer,
                    hd_rec->packet->Msg,hd_rec->packet->header.len);
                messagepointer += hd_rec->packet->header.len;
                it = Head_recv_vector.erase(it); // 删除元素，并将迭代器指向下一个元素
                delete hd_rec; // 删除动态分配的资源（如果有的话）
            }
            else
            {
                // 如果条件不满足，只需递增迭代器
                ++it;
            }
        }
        buffer_lock.unlock();
        if (file_finish)
```

```
        {
            print_lock.lock();
            cout << "update_线程结束" << endl;
            print_lock.unlock();
            return;
        }
        //Sleep(200);
    }
}
```

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。

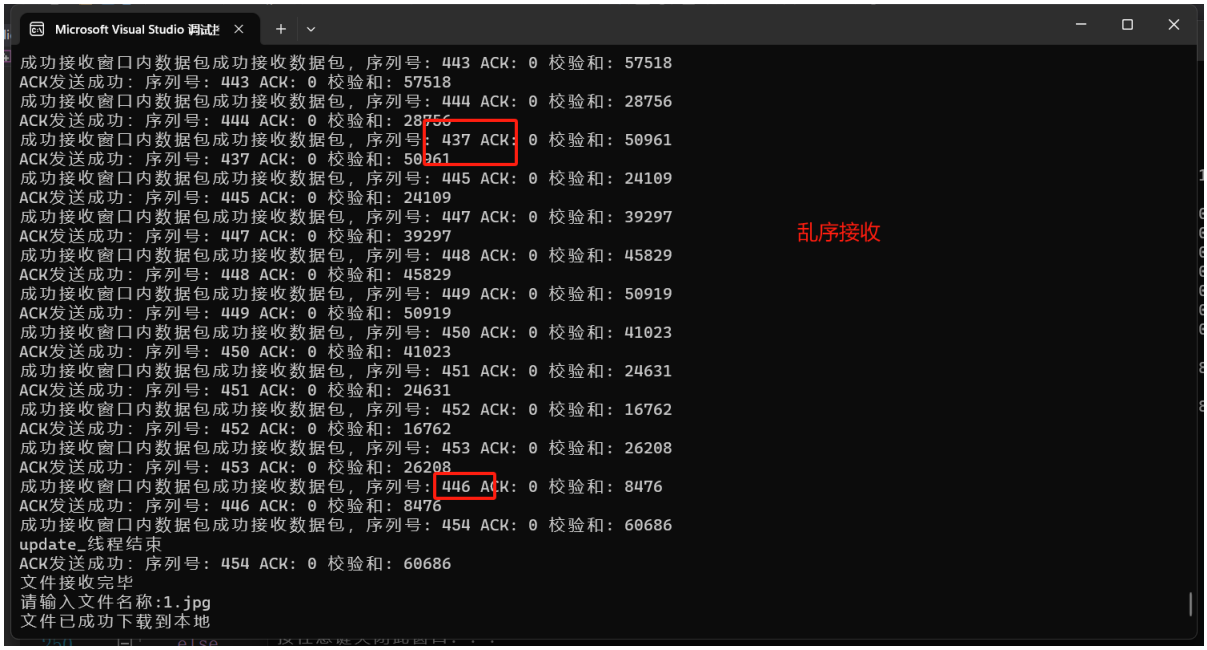

图 4.2: 乱序接收

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。

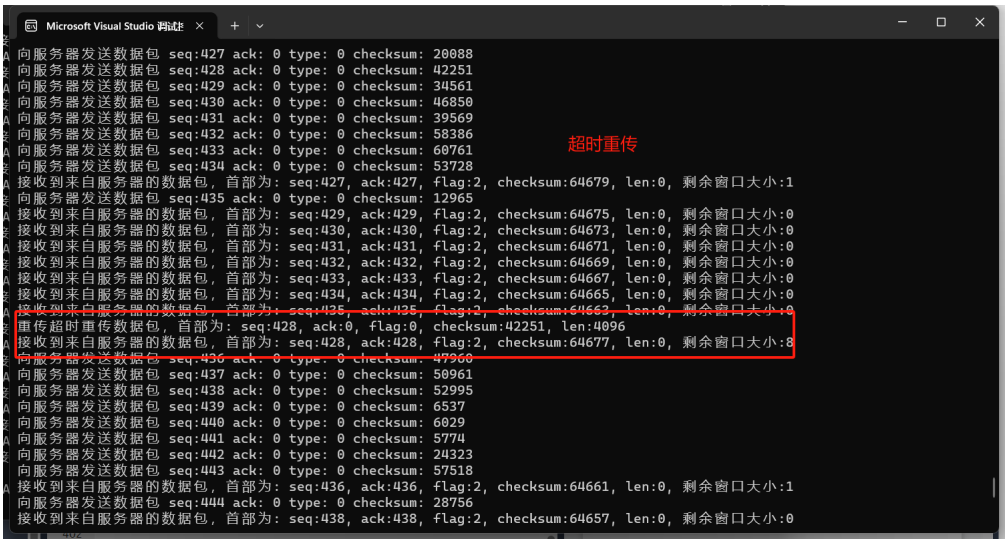

图 4.3: 超时重传

4.3 文件传输实验结果

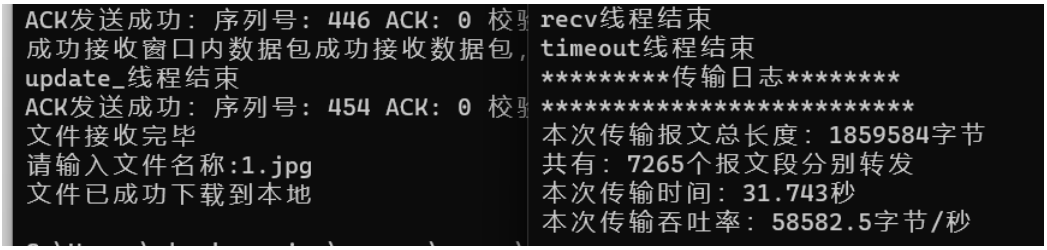

图 4.4: 1.jpg 传输结果

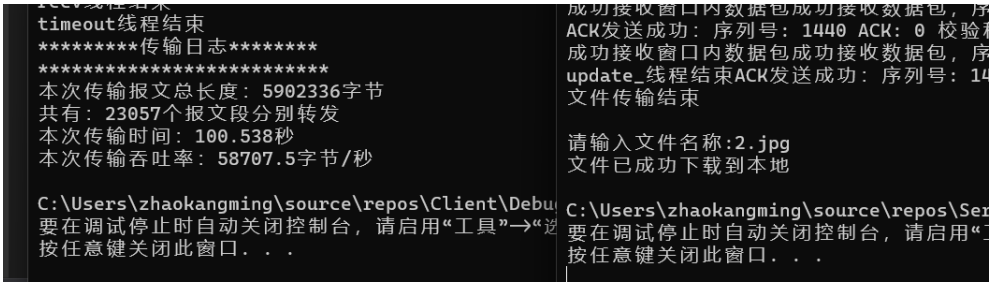

图 4.5: 2.jpg 传输结果

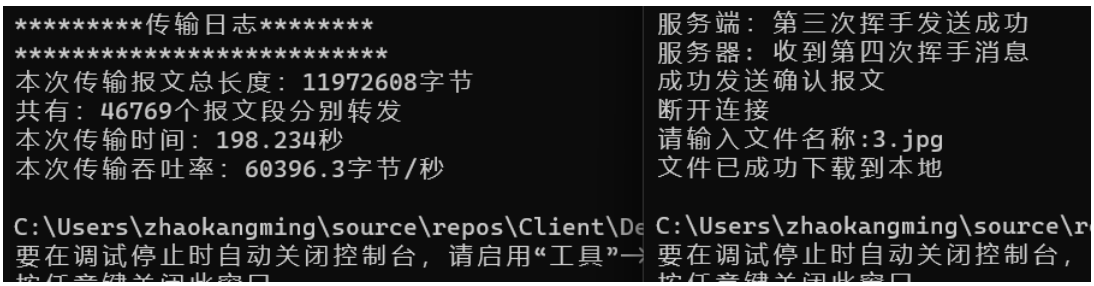

图 4.6: 3.jpg 传输结果

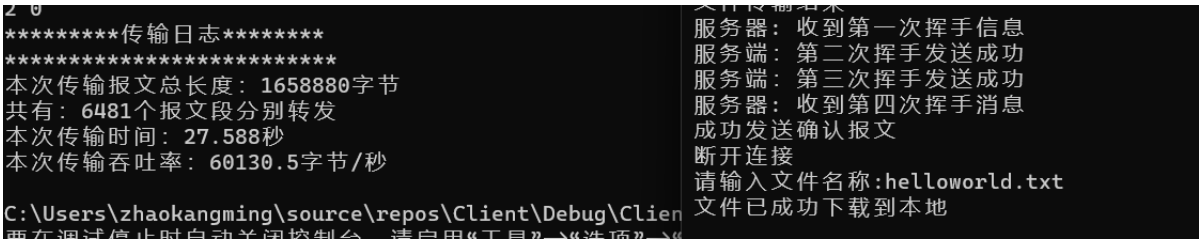

图 4.7: helloworld.txt 传输结果

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。

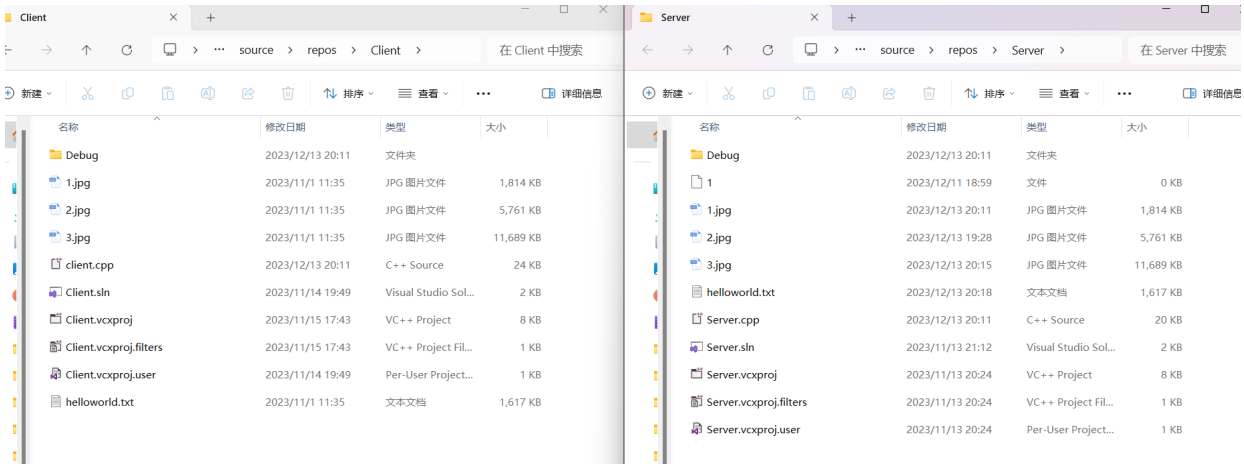

图 4.8: 文件大小传输结果

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。

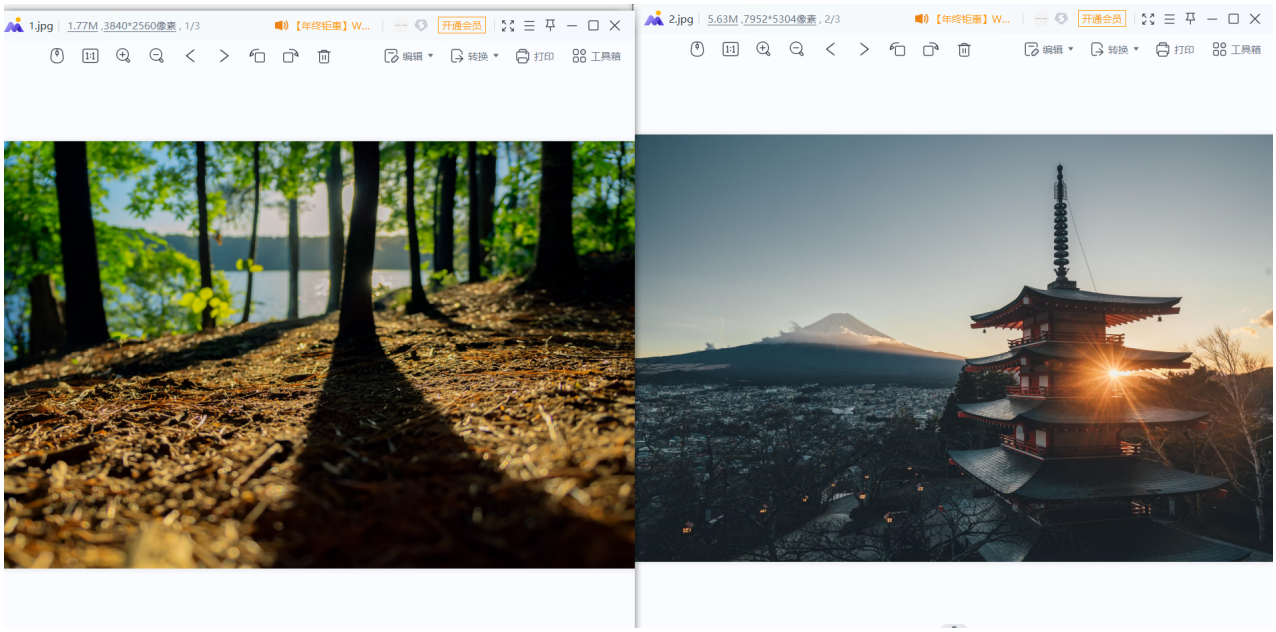

图 4.9: 文件实际传输结果

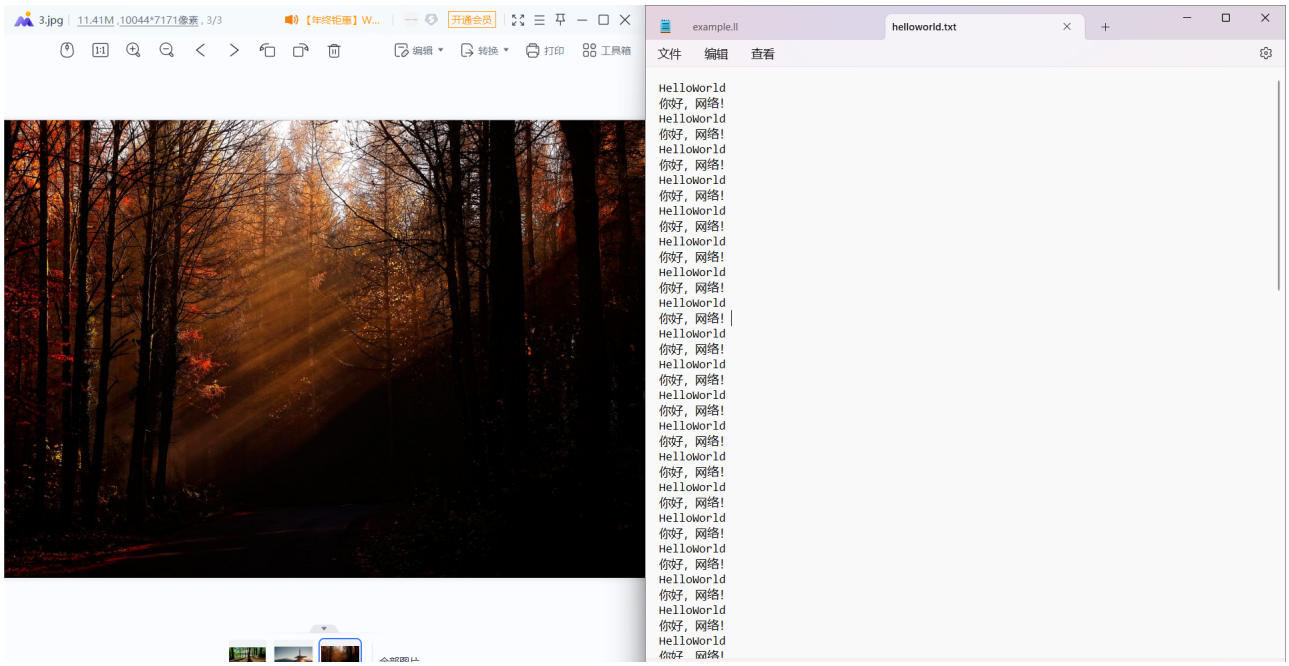

图 4.10: 文件实际传输结果

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。