
南開大學
Nankai University

# 南　　开　　大　　学

## 计　算　机　学　院

**计算机网络实验 3-2**

---

# 基于 UDP 服务设计可靠传输协议并编程实现

---

杨浩甫　2113824

年级：2021 级

专业：计算机科学与技术

指导教师：张建忠、徐敬东

2023 年 12 月 1 日

# 目录

# 一、 实验基本描述

在实验 3-1 中，利用了数据报套接字在用户空间实现了面向连接的可靠数据传输。这包括建立连接、差错检测以及确认重传等功能并使用停等机制进行流量控制。本次实验则在此基础上做了改进，引入了基于 GBN 的滑动窗口流量控制机制，并在接收端采用了累计确认的方式。

主要改进在于数据传输方面，采用了 **GBN 流水线协议**，流量控制则是基于 GBN 滑动窗口机制，使用固定窗口大小。这次实验采用了累计确认的方式，即只确认连续正确接收分组的最大序列号

# 二、 实验具体工作

1. 在 3-1 的基础上，增加滑动窗口功能
2. 采用 GBN 协议
3. 采用累计确认
4. 对不同情况下的传输时间和吞吐率进行探究。

## (一) 协议设计

### 1. 报文格式

这次实验对报文格式进行了一点修改：

**增加了确认重传的标志位 RE**，下面是新的报文的结构：

| 字段 | 位置 (比特位) | 大小 (比特位) |
|---|---|---|
| flag | 0-31 | 32 |
| seq | 32-47 | 16 |
| ack | 48-63 | 16 |
| len | 64-95 | 32 |
| num | 96-127 | 32 |
| checksum | 128-143 | 16 |
| data | 144-1175 | 8192 |

表 1: 报文结构

| 32 - 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| | RE | EXT | ACK | END | START | FIN | SYN |

表 2: flag 标志位设计

### 2. 状态转换图

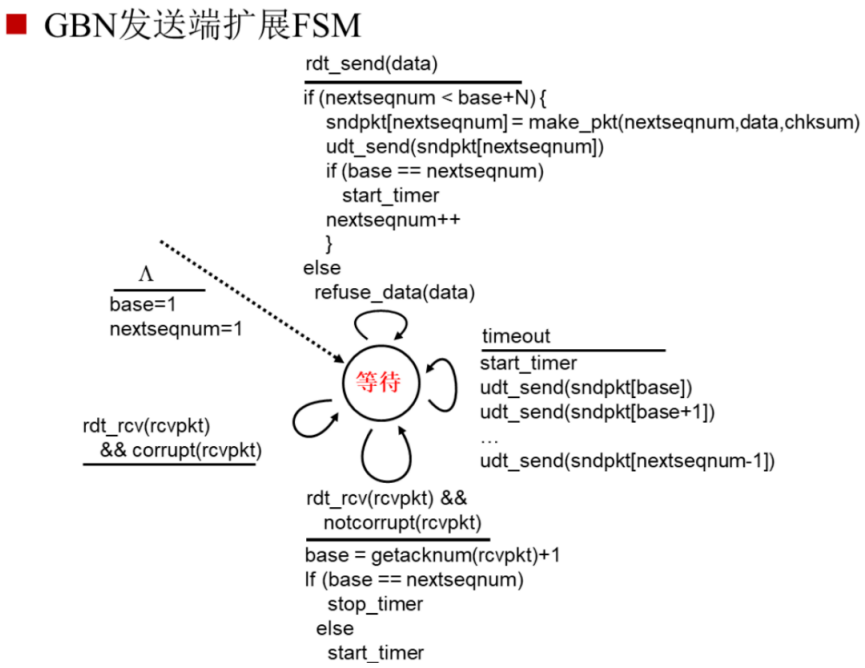


图 1: 交互过程

### 3. 流水线设计

考虑到 rdt3.0 停等协议的性能问题，未能够充分的提高链路利用率。

故本实验 3-2 发送端将采用流水线协议进行性能优化：在确认未返回之前允许发送多个分组。

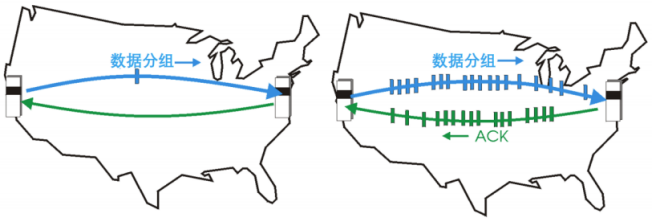


图 2: 流水线协议

### 4. 滑动窗口机制

在实验 3-2 中，采用了基于滑动窗口的流量控制机制。这种方法涉及发送方和接收方各自拥有缓存数组，其中发送方缓存包含已发送且得到确认的包序号、已发送但未确认的包序号以及未发送的包序号。接收方的缓存则包括已接收的包序号、正在接收的包序号和未接收的包序号。这两个数组各自有两个扫描指针，形成一个窗口。窗口的大小为 N，允许一次性发送 N 个报文段，实现了流水线机制，以提升传输性能。窗口随着协议运行的进程在序列号空间内向前滑动，允许新的数据进入发送窗口并发送到接收方。但是由于本实验的要求是接收端窗口为 1，所以在实际设计中并为给接收端分配发送和接收缓冲区。

具体来说，窗口指代允许使用的序列号范围，其大小为 N，随着协议的运行在序列号空间内向前滑动。窗口被划分为左边界、发送边界和右边界，大小固定。左边界左侧是已经发送并得到

确认的数据，左边界到发送边界是已发送但未得到确认的数据，发送边界到右边界是等待发送的数据，右边界右侧是不可发送的数据。

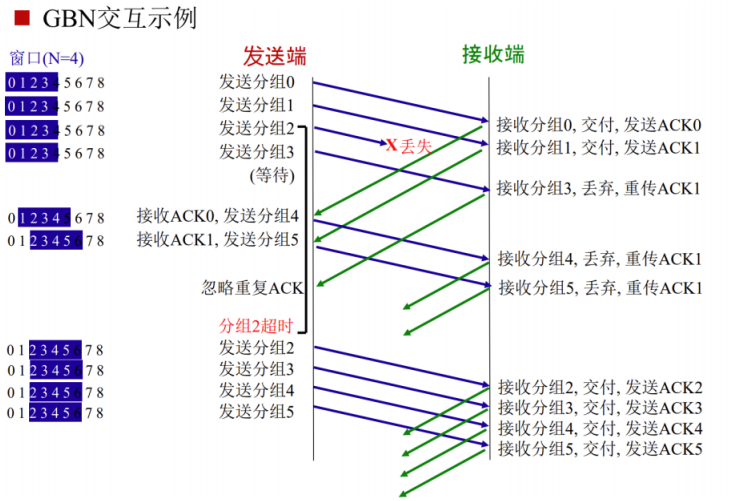


图 3: 流水线协议

## 5. GBN 协议

在 GBN（Go-Back-N）中，发送方可以发送 N 个未得到确认的分组，每次发送时发送滑动窗口剩余大小个数据包。发送方使用累计确认，只确认连续正确接收分组的最大序列号。若在规定时间内未收到正确应答信号，则触发重传机制，重新发送所有未被确认的分组。接收方按序确认最高序号的正确接收分组。GBN 算法可能会产生重复的 ACK，因此需要保存期望接收的分组序号。当接收端面对失序分组时，不会缓存该数据包而是直接丢弃，并重发最高正确序号对应的 ACK 通知发送端。

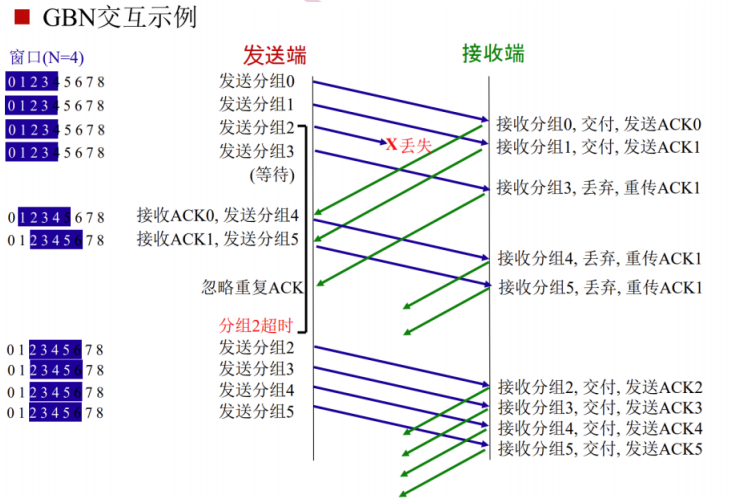


图 4: GBN 协议

## 6. 累积确认

累计确认机制是一种有效的方式，当接收端收到编号为 n 的 ACK 报文时，表示序号 0 至 n 的报文段都已成功接收。这个机制使得窗口能够向前移动，直至基准值（base）与该 ACK 报文

确认的下一个报文序号相等。在此过程中，新的更大序号的报文也会进入窗口，从而在窗口内产生空闲区间。

这种机制的作用是动态调整窗口，确保它不断地适应网络条件和接收端的处理能力。一旦确认了一批连续的报文段，发送方就可以将窗口向前滑动，释放已经确认的部分，同时允许新的报文进入窗口。

这个过程使得通信过程中窗口的大小和位置不断变化，窗口内的空闲区间则代表了可以用于发送新数据的空间。这种灵活性允许发送方根据接收端的状态和网络的情况来调整发送的数据量，从而实现了有效的流量控制和可靠的数据传输。

#### 7. 确认重传

为了实现确认重传，我为报文结构增加了一个标识符：re，当接收端接收到与预期 seq 不符合的数据包时，会设置 re 标志位并返回所期望的 ack，以此来通知发送端重新发送未被确认的数据包。

当发送端接收到数据包时，会检查 re 标志位，若置为一，则将缓冲区内所有数据包依次重传，以实现确认重传。

#### 8. 超时重传

在本次实验的 GBN（Go-Back-N）协议中，超时重传机制是应对丢包情况的关键解决方案。当某个数据包发送后，在规定的时间段内未收到回传的确认信号，系统便会启动超时重传机制。这时，发送端会重新发送窗口缓冲区中所有未被确认的分组。

这种机制确保了在网络丢包或延迟的情况下依然能够保证数据的完整性和可靠性。通过及时的重传机制，即使发生了部分数据包的丢失，也能够及时补发，确保了信息交互的可靠性和数据的正确性。

### (二) 具体实现

#### 1. 窗口设置

首先，我定义了一些与滑动窗口协议相关的参数和数据结构：

- WINDOW_SIZE：表示发送窗口的大小，即允许同时发送但未收到确认的数据包数量。
- message window[WINDOW_SIZE]：这是一个数组，用于存储发送窗口内的数据包。在滑动窗口协议中，数据包按顺序发送，但并非所有数据包都已收到确认。这个数组用于存储已发送但未收到确认的数据包。
- base：是滑动窗口的基序号，代表发送窗口的起始位置。它指示已确认的数据包序号，小于 base 的数据包都已经收到了确认。
- nextSeqNum：表示下一个待发送的序号，即当前窗口中最后一个未发送的数据包序号加一。
- acked[WINDOW_SIZE]：这是一个布尔数组，用于标记发送窗口内的消息是否被确认。每个位置对应窗口中一个数据包的确认状态。
- timeout：定义了超时时间，表示在等待确认时允许的最大时间。
- packetTimes[WINDOW_SIZE]：这个数组记录了每个数据包的发送时间，用于计算超时。

滑动窗口设置

```
const int WINDOW_SIZE = 10; // 窗口大小
message window[WINDOW_SIZE]; // 发送窗口
int base = 0; // 发送窗口的基序号
int nextSeqNum = 0; // 下一个待发送的序号
bool acked[WINDOW_SIZE] = { false }; // 标记发送窗口内的消息是否被确认
std::chrono::milliseconds timeout(800); // 800毫秒超时时间
int packetTimes[WINDOW_SIZE];// 记录每个数据包的发送时间
```

## 2. 发送端

**接收线程：**

这个线程函数内主要完成以下功能：

- while (isRunning) 通过这个循环保持函数运行，只要 isRunning 标志为真，就会持续执行。
- message ackMsg = recvmessage(client, serveraddr); 接收来自服务器的消息。如果接收到 ACK（确认消息），则会根据确认号执行操作。
- 累积确认：它首先检查接收到的确认号是否正确，然后根据情况更新 base（基序列号）。base 是一个序列号，表示接收到的数据的起始位置。如果收到一个 ACK 消息，就会根据 ACK 的确认号更新 base，并将对应的序号标记为已确认。
- 确认重传：if (ackMsg.isRE()) 这段代码来处理，如果接收到的 ACK 消息表示需要重传，则检查窗口内的数据包状态。如果某个数据包已经确认但仍在窗口内，说明需要重传，于是它会发送相应的数据包。
- if (ackMsg.isEND()) 这里处理结束信号。如果接收到的消息表示结束，则调用 exit(EXIT_SUCCESS); 来正常结束进程。

接收文件线程

```
void receiveACK() {
    while (isRunning) {
        message ackMsg = recvmessage(client, serveraddr);
        if (ackMsg.isACK()) {
            std::lock_guard<std::mutex> lock(mtx); // 在代码块中自动上锁，出
                作用域时自动释放
            int ackNum = ackMsg.ack;
            while (base < ackNum) {
                acked[base % WINDOW_SIZE] = false;
                (base)++;
            }

            if (ackMsg.isRE()) {
                for (int i = base; i < nextSeqNum; ++i) {
                    int index = i % WINDOW_SIZE;
                    if (acked[index]) {
                        SetColor(11, 0);
```

```
                        cout << "Client:确认重传" << window[index].seq << "
                            号数据包" << endl;
                        sendmessage(client, serveraddr, window[index]);
                        packetTimes[index] = clock(); // 重新记录发送时间
                        //acked[index] = false;
                    }
                }
                continue;
            }
        }
        if (ackMsg.isEND()) {
            exit(EXIT_SUCCESS);
        }
    }
    return;
}
```

**发送函数：**

我通过一个滑动窗口发送文件。

在发送数据的部分，首先创建一个消息（message），将数据分块读取到消息中，然后发送该消息到服务器端。发送完成后，记录已发送的数据信息和时间，并更新序号，以便发送下一个数据块。发送的数据被存储在一个发送窗口内，窗口大小为 WINDOW_SIZE。

超时重传的检查是为了处理网络环境中可能发生的丢包、延迟等问题。当某个数据包超过了设定的超时时间仍未收到确认时，会触发超时重传机制。代码中使用了 packetTimes 数组来记录发送每个数据包的时间。当基序号对应的数据包超时时，会对发送窗口内的所有已发送但未确认的数据包进行重传。这样可以确保在网络环境不稳定或数据包丢失的情况下，及时地进行重传，提高数据的可靠性和完整性。

传输文件

```
while (filelen || base != nextSeqNum) {
    if (nextSeqNum < base + WINDOW_SIZE && filelen > 0) {
        std::lock_guard<std::mutex> lock(mtx);
        SetColor(15, 0);
        message msg;
        msg.seq = nextSeqNum;
        msg.len = min(filelen, 1024);
        inFile.read(msg.data, msg.len);
        filelen -= msg.len;
        msg.setchecksum();
        sendData(client, serveraddr, msg);
        cout << "Client:发出" << msg.seq << "号数据包" << endl;
        cout << "Client:此时已发送数据右端在窗口的位置:" << nextSeqNum %
            WINDOW_SIZE << endl;
        msg.output();
        cout << endl;

        acked[nextSeqNum % WINDOW_SIZE] = true;
        packetTimes[nextSeqNum % WINDOW_SIZE] = clock(); // 记录发送时间
```

```
            nextSeqNum++;
        }


        // 超时重传检查
        if ((clock() - packetTimes[base % WINDOW_SIZE]) > timeout.count()) {
            // 超时重传
            SetColor(11, 0); // 红色文本, 蓝色背景
            for (int i = base; i < nextSeqNum; ++i) {
                int index = i % WINDOW_SIZE;
                if (acked[index]) {
                    cout << "Client:␣超时重传" << window[index].seq << "号数据包"
                         << endl;
                    sendmessage(client, serveraddr, window[index]);
                    packetTimes[index] = clock(); // 重新记录发送时间
                    //acked[index] = false;
                }
            }
            cout << endl;
        }
}
```

### 3. 接收端

函数 ReceiveFile 主要用于在服务器端接收来自客户端的文件数据。其核心部分是通过套接字 serverSocket 接收数据，并对接收到的数据包进行处理。

首先我初始化了一些变量：

- expectedSeq 表示期望的初始序列号。
- last_ack 记录最后确认的序列号。
- 一个窗口大小 WINDOW_SIZE 的数组 acked，用于标记已确认的数据包。

然后是数据接收和处理：通过 recvmessage 从客户端接收数据包 msg。然后对接收到的消息进行判断：

- 若消息是结束标志（msg.isEND()），则发送一个确认消息给客户端，关闭文件流，并等待接收下一个文件的名称。
- 若消息的序列号在发送窗口内且与期望的序列号匹配，则进行处理：发送对应的确认消息（ACK），将数据写入文件，并更新相关状态。
- 若收到的消息序列号为期望序列号 + 1，则表示希望重复发送，会发送之前确认的 ACK，避免重复接收。
- 滑动窗口：在确认收到消息后，根据已确认的消息情况滑动窗口，更新发送窗口的基序号 base 和已确认消息的标志位。

最后设计了丢包模拟：代码中包含了一个随机数，如果随机值小于设定的丢包率，则模拟丢包，不发送确认消息，继续等待接收数据。

接收文件

```
message msg;
u_short expectedSeq = 0; // 期望的初始序列号
int last_ack=0;
while (1) {
    SetColor(15, 0);
    msg = recvmessage(serverSocket, clientaddr);

    if (!msg.isEXT()) {
        continue;
    }
    if (msg.isEND()) {
        message ackMsg;
        ackMsg.setACK();
        ackMsg.ack = msg.ack + 1;
        ackMsg.setEND();
        sendmessage(server, clientaddr, ackMsg);
        SetColor(13, 0);
        cout << "Server: 接收文件"<< receivedFilePath<<"成功!!" << endl <<
            endl;
        cout << "***********************************************" << endl;
        outFile.close();
        outFile.clear();
        int iMode = 0; //1: 非阻塞, 0: 阻塞
        ioctlsocket(server, FIONBIO, (u_long FAR*) & iMode);//非阻塞设置
        SetColor(15, 0);
        return ReceiveName(server, clientaddr);
    }
    else if (msg.seq >= base && msg.seq < base + WINDOW_SIZE) {
        if (msg.seq == expectedSeq ) {
            // 收到正确的数据包, 发送 ACK
            message ackMsg;
            ackMsg.setACK();
            ackMsg.ack = msg.seq + 1;
            last_ack = ackMsg.ack;
            acked[msg.seq % WINDOW_SIZE] = true;

            double random_value = (double)rand() / RAND_MAX;
            if (random_value < loss_rate) {
                SetColor(11, 0);
                cout << endl;
                cout << "Server: 模拟丢包, 未发送数据包" << endl;
                cout << endl;
                //sendmessage(server, serveraddr, message()); // 返回空消息
                continue;
            }

            sendmessage(server, clientaddr, ackMsg);
```

```
            cout << endl;
            msg.output();
            cout << "Server: 发送确认收到的数据包(对应的ack)" << endl;


            outFile.write(msg.data, msg.len);

            num--;
            expectedSeq++;

            // 滑动窗口
            while (acked[base % WINDOW_SIZE]) {
                acked[base % WINDOW_SIZE] = false;
                base = base++;
            }
        }
    }
    else if(msg.seq == expectedSeq+1){
        SetColor(10, 0);
        message ackMsg;
        ackMsg.setACK();
        ackMsg.ack = last_ack-1;
        ackMsg.setRE();
        sendmessage(server, clientaddr, ackMsg);
        cout << endl;
        //msg.output();
        cout << "Server: 重复发送确认收到的数据包: " << ackMsg.ack << endl;
    }
}
```

# 三、 结果分析

## (一) 传输日志

下面我将在窗口大小为 10，丢包率为 0.03 的条件下以 1.jpg 为例，来介绍传输日志。首先，我们来看建立连接和发送端打印窗口大小：

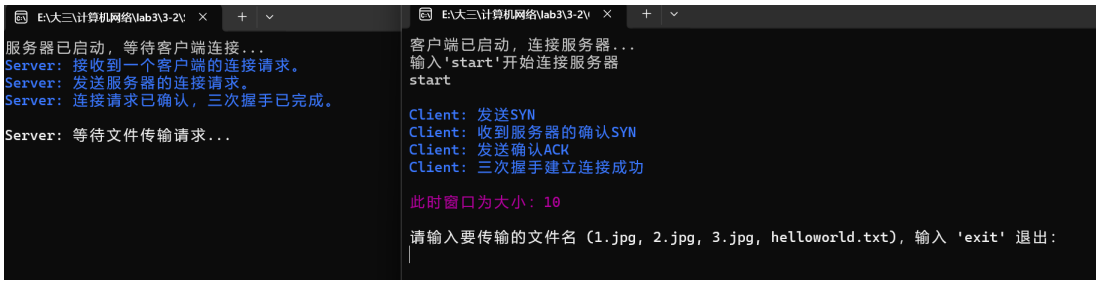


图 5: 传输准备

接下来是传输过程，接收端和发送端都打印传输日志，相较于 3-1 主要的改进是：

- 发送端接收到文件时，根据累积确认后的结果，完成窗口滑动与变化。
- 发送端发送文件时，打印已发送数据包在窗口中的位置，用来反映当前窗口剩余空间。

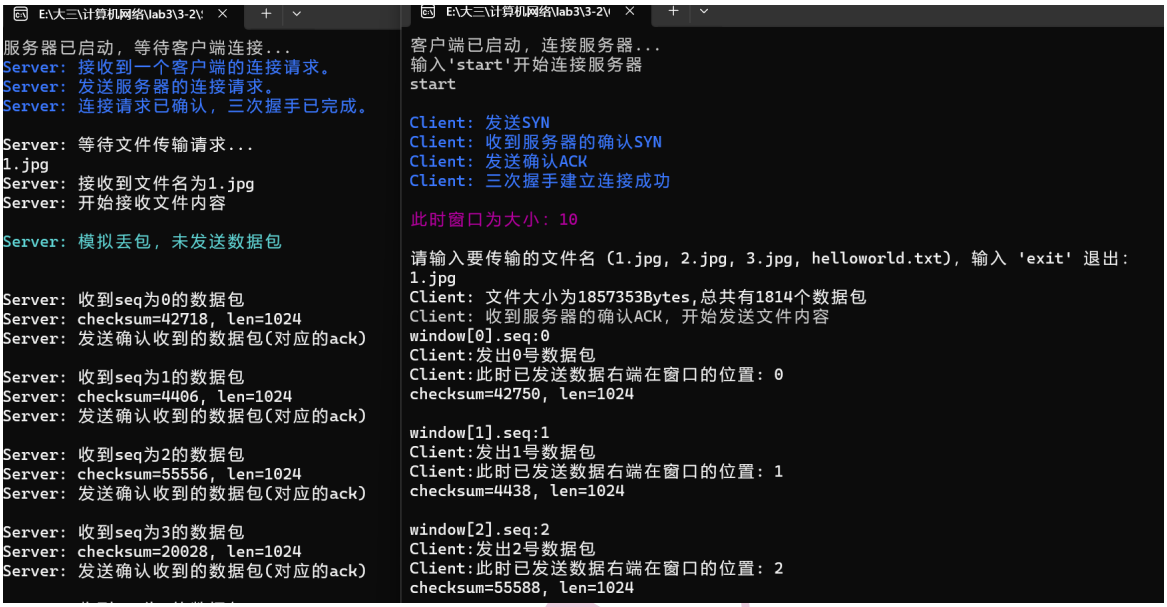


图 6: 传输过程

再来分析一下重传效果：

- 接收端收到 seq 不匹配的数据包：重新发送以收到的数据包最大的 ack（1724）并打印。
- 发送端收到 RE 标识符：立刻将将已发送未确认（1724-1733）的数据包全部重发。

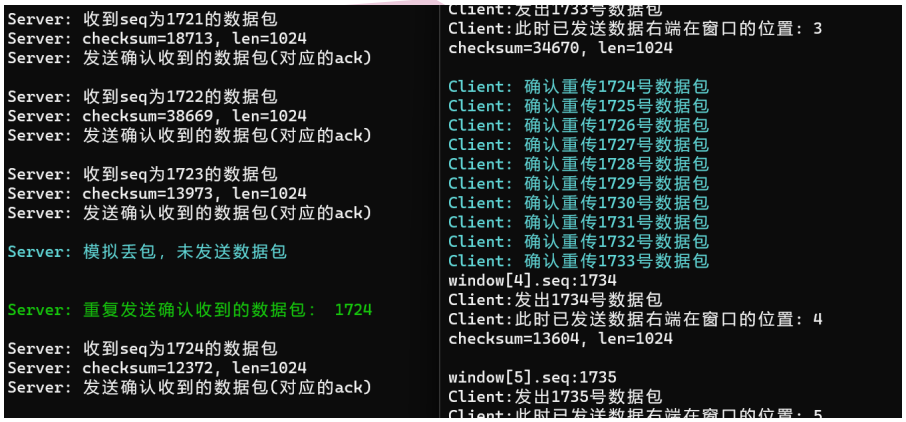


图 7: 重传

最后，文件接收成功，打印日志，并输出传输时间和吞吐率。之后是断开连接的日志。

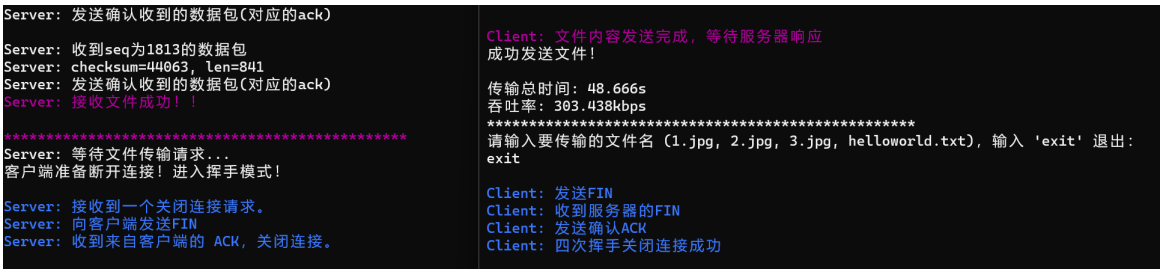


图 8: 传输成功

## (二) 文件传输结果

传输四个文件后，都可以接收到相应的文件：


图 9: 1.jpg


图 10: 2.jpg

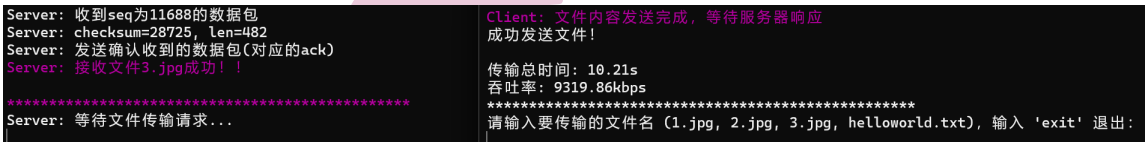


图 11: 3.jpg

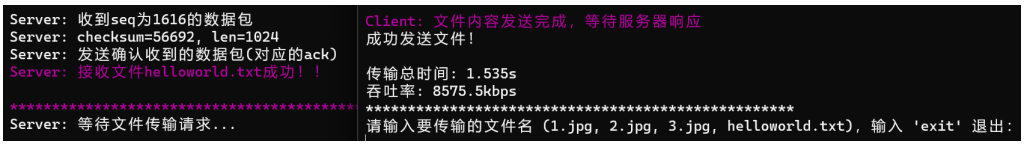


图 12: helloworld.txt

可以看到，四个文件都可以被 Server 端正确接收：

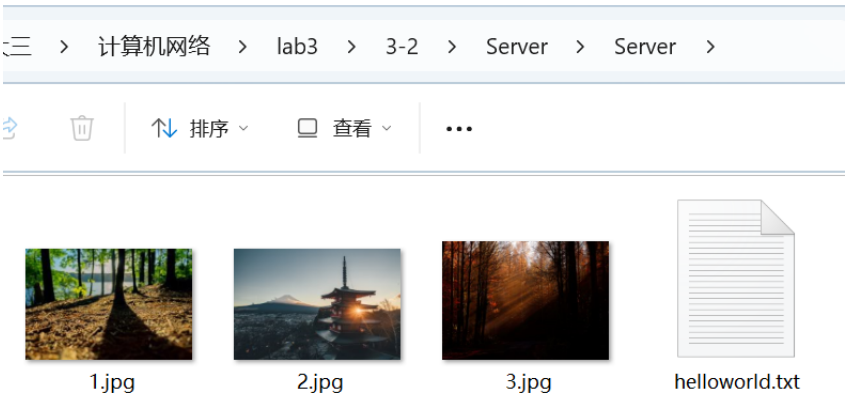


图 13: 传输效果

## (三) 传输性能测试结果

我对四个文件都进行了传输性能测试，接下来展示在丢包率和延时为 0 的情况下的传输结果:

| 文件名 | 窗口大小 | 数据包数（个） | 传输时间 (s) | 吞吐率（kbps） |
|---|---|---|---|---|
| 1.jpg | 4 | 1813 | 1.496 | 9871.05 |
| 2.jpg | 8 | 5760 | 4.866 | 9637.92 |
| 3.jpg | 16 | 11688 | 10.17 | 9356.52 |
| helloworld.txt | 32 | 1616 | 1.444 | 9115.92 |

表 3: 传输文件统计

为了探究滑动窗口大小对传输性能的影响，我在传输 1.jpg，丢包率为 0.01 时，测试了不同窗口大小下的传输性能:

| 窗口大小 | 传输时间 (s) | 吞吐率（kbps） |
|---|---|---|
| 1 | 21.681 | 681.108 |
| 4 | 1.519 | 9721.59 |
| 8 | 1.517 | 9734.41 |
| 16 | 1.565 | 9435.84 |

表 4: 1.jpg 传输数据

分析结果可知:

- 滑动窗口先对于依次传输, 传输时间显著减少。窗口大小从 1 增加到 16, 传输时间从 21.681 秒降低到 1.565 秒。但是当窗口大于 1 时，传输时间并没有很大的改变
- 吞吐率变化：从窗口大小 1 到 16，吞吐率从 681.108 kbps 增加到 9435.84 kbps。这表明滑动窗口使得网络的利用率提高，单位时间内传输的数据量也随之增加。但是更大的窗口对于吞吐率也没有显著提升。

# 四、 总结与反思

本次实验成功利用 GBN 协议实现滑动窗口功能，对累积确认、确认重传等概念有了深入的理解，也尝试了多线程编程，实验完成度较高，但是也存在一定的问题，比如说线程的结束还需改进。