计算机网络lab3-2

*姓名：**罗瑞***

*学号：**2210529***

*专业：**密码科学与技术***

1.【实验内容】

✅ 基于滑动窗口的流量控制机制，采用固定窗口大小，支持累积确认，完成给定测试文件的传输。采用的具体协议为:Go back N，其是一种可靠的数据传输协议，基于滑动窗口机制和累积确认方法，允许发送方在等待确认前连续发送多个数据帧。发送方维护一个固定大小的窗口，表示可以连续发送的未确认帧数，而接收方只需确认已按序收到的最后一个数据帧即可，最后一个数据帧对应到我的编程实现就是累积确认指针(accumulate_point)。

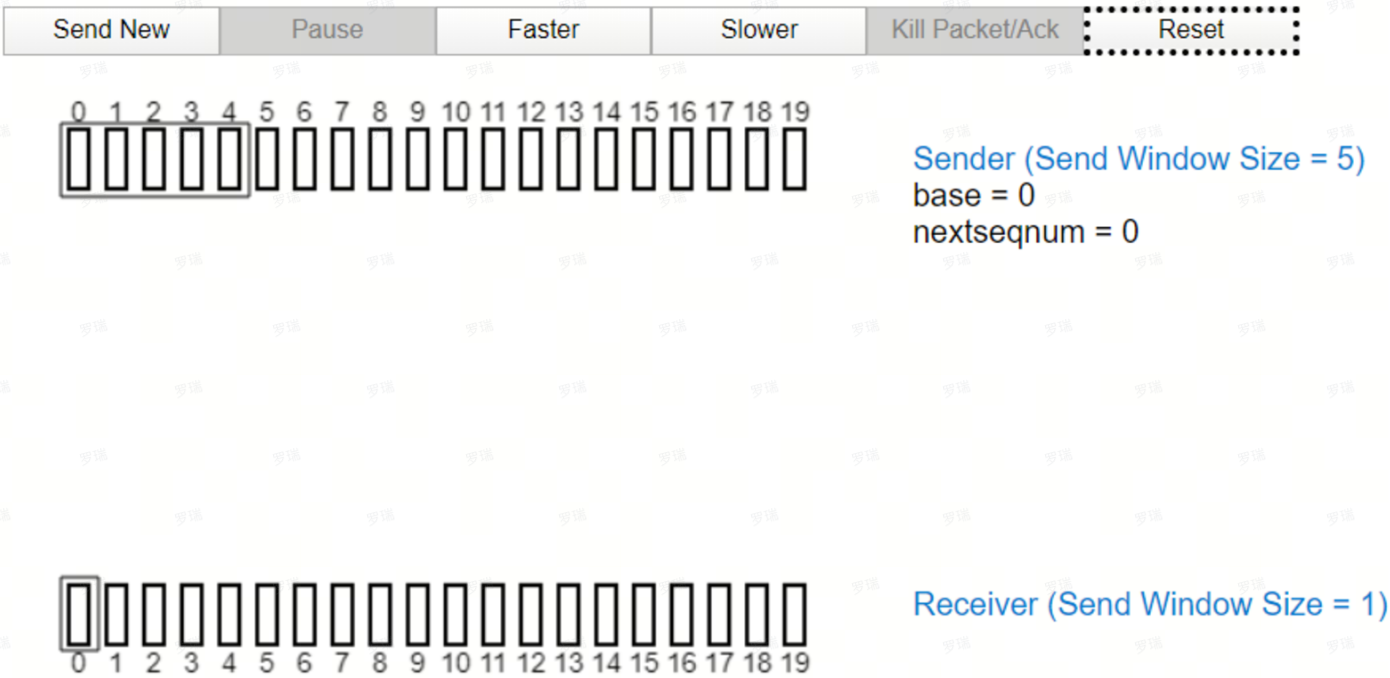

图1：Go back N 协议图解

2.【协议设计与C++编程实现】

2.0本协议基本说明

本次实验的数据报文类 `UDP_DATAGRAM` 格式完全沿用实验3-1，类似于TCP建连和断连的三次握手、挥手机制完全沿用实验3-1，校验和差错检验以及文件的读写等基本过程的实现也完全沿用实验3-1，本实验报告不再做详细说明。本协议使用的4个状态位也完全沿用3-1，但用途略有不同，主要是发送端文件的报文发送中都加入了SYN位标记，以及最后的发送完成确认。

> ✅ **本实验为了完成支持滑动窗口和累积确认的协议，引入了多线程的处理方法，并且使用阻塞式的socket套接字功能函数，维护了确认数组acked[]和累积确认指针accumulate_point。**

2.1完成本协议传输所需的重要变量和定义说明

threads	vector<std::thread>类型变量，是支持窗口内多数据报同时发送互不影响的线程池
accumulate-point	累积确认指针，在接收端更新好后由ACK回复同步给发送端，发送端再通过获取指针的更新来更新ack[]数组，从而影响窗口内数据报的发送选择
acked[]	确认数组，为了便于程序实现维护的布尔型数据结构
windowtop	滑动窗口底端指针
	滑动窗口顶端指针

windowbase	
buffer	缓冲区，用于暂时接收文件数据
is_thread	布尔型数组，用来记录对应序列号的数据报有没有建立过发送线程，建立过的则不用再建立

2.2完成传输使用的线程的解释

本协议基于滑动窗口和累积应答，因此窗口的实时调整是很有必要的，而ACK的接收、窗口内需要发送的数据报的发送和窗口的滑动是不能互相干扰的，因此需要用到多线程的思路。

✅ 这里我对于**ACK的接收**和**窗口数据报的发送**分别采用两个不同的线程，**窗口的实时滑动调整**则“阻塞”在主进程中，这样三者便实现了互不干扰，共同作用。

```
// 创建接收线程，负责接收服务器确认消息
thread recv(receive);
recv.detach(); // 将线程分离

// 创建发送线程，负责发送文件分片
cout << "开始传输文件对应的UDP数据报……" << endl;
thread send(sending);
send.detach(); // 将线程分离


//窗口滑动确认处理“阻塞器”,直到所有报文都被正确接收确认
windowsending();
```

2.2滑动窗口设计与实现

- #1 是已发送并收到 ACK确认的数据：1~31 字节
- #2 是已发送但未收到 ACK确认的数据：32~45 字节
- #3 是未发送但总大小在接收方处理范围内（接收方还有空间）：46~51字节
- #4 是未发送但总大小超过接收方处理范围（接收方没有空间）：52字节以后

图2：发送端滑动窗口示例（图中累积确认指针指向seq=45）

2.2.1滑动窗口内部发送数据报的选择

`sending()` 是一个多线程的核心函数（注意这个函数本身也是一个线程），主要实现基于**滑动窗口**机制的 UDP 分片数据传输。

✅ 在主循环内，要完成滑动窗口内数据包发送，其**循环检测滑动窗口内的数据分片（`udpdatagram`）状态，并为需要发送的未申请过处理线程的数据包（基于累积确认指针回复的acked[]数组index对应为false）启动线程进行发送。**由于我申请了线程池：**std::vector<std::thread> threads，**因此窗口内申请的所有线程都会在数据报完成累积确认范围内结束掉，从而不造成内存资源的浪费，维护了程序的健壮和高效性。在**等待所有数据报发送线程完成后，再发送传输完成信号报文udp_end并等待接收端的传输完成信号确认报文udp_end_recv，**这组报文是为了避免这样一种错误逻辑：**接收端确认了所有报文但返回的对应累积确认回复并没有立即被发送到接收到，发送端就会继续等待接收端，但是此时接收端由于所有的报文都已正确确认，已经关闭了报文的接收和回复，则发送端就会进入死循环状态。**

```
1 //发送端核心处理函数，处理方式：滑动窗口，这个函数为了和滑动窗口设置函数并行，设置为线程
2 int sending() {
3     // 不断遍历检测，发送滑动窗口内的未确认分片
4     while (true) {
5         // 检测到滑动窗口完毕后直接退出
6         if (windowbase == udp_num - 1 && acked[windowbase])
7             break;
8 
9         for (int i = windowbase; i <= windowtop; i++) {
```

```
            if (!acked[i] && !isthread[i]) { // 检查分片状态，只发送窗口内表示未确认的报文（其实这里ack[]数组也没必要一定维护，实际上使用累计确认指针就可以了）
                // 启动一个线程发送分片,建立过的线程没必要再建立
                threads.emplace_back(std::thread(udpdatagramcopyandsend, i));
                isthread[i] = true;
            }
        }
    }

    // 等待所有发送线程完成
    for (auto& t : threads) {
        if (t.joinable()) {
            t.join(); // 阻塞等待线程结束
        }
    }

    //发送文件传输确认报文
    int timeold = clock();
    UDP_DATAGRAM udp_end;
    memset(&udp_end, 0, sizeof(UDP_DATAGRAM));
    udp_end.setstate(SYN);
    udp_end.setstate(FIN);

    if (sendto(sock, (char*)&udp_end, sizeof(udp_end), 0, (SOCKADDR*)&server, l) == SOCKET_ERROR)
        return 0;


    while(true){
        int timenew = clock();
        if (timenew - timeold > timelimit)
        {
            //超时重传
            int timeold = clock();//更新计时
            UDP_DATAGRAM udp_end;
            memset(&udp_end, 0, sizeof(UDP_DATAGRAM));
            udp_end.setstate(SYN);
            udp_end.setstate(FIN);

            if (sendto(sock, (char*)&udp_end, sizeof(udp_end), 0, (SOCKADDR*)&server, l) == SOCKET_ERROR)
                return 0;

        }

    UDP_DATAGRAM udp_end_recv;
    memset(&udp_end_recv, 0, sizeof(UDP_DATAGRAM));
```

```
    recvfrom(sock, (char*)&udp_end_recv, sizeof(UDP_DATAGRAM), 0,
(SOCKADDR*)&server, &l);
    if (udp_end_recv.getstate() & ACK && udp_end_recv.getstate() & FIN)
    {
        cout << "恭喜，所有数据报文均被正确接收！";
        break;
    }
    }

    return 1; // 发送函数完成
}
```

对于"停等" 协议而言:，其信道利用率低，发送完一帧后等待，这个时候信道完全是空闲的。我们的滑动窗口为了提高信道利用率，发送端发送完一个数据报后，无需等待接收端对应的的 ACK 确认帧，立刻发送窗口内后面的数据报，这样信道的利用率就提高了。这里的关键还是不同数据报的发送不能互相干扰，所以对于一个窗口内每一个需要发送的数据报，我都单独开启一个线程，并支持超时重传，无论是传输出现差错还是传输过程中出现丢包，最终反映在发送端的应对措施都是触发超时重传。

2.2.2滑动窗口内部发送单个数据报的处理细节（也采用独立线程）

`udpdatagramcopyandsend` 是一个用于发送单个 UDP 数据报的核心函数，该函数实现了基于超时重传机制的 UDP 数据报可靠发送，是滑动窗口协议的重要组成部分。

✅ 通过线程化处理和超时机制，有效提升了传输效率和可靠性，同时保证单个数据报的独立性和正确性。关键处理机制包括：

1. 数据分片的初始化与校验。
2. 数据报的发送及超时重传机制。
3. 独立线程处理，避免数据报间的干扰。

```
//发送一个数据报的核心函数，处理方式：丢包&差错不会引发确认，最终会触发超时重传，
// 注意每一个窗口内的数据报不能互相影响，每一个数据报的处理都应该是一个线程
int udpdatagramcopyandsend(int seq) {
    UDP_DATAGRAM udp_now;
    memset(&udp_now, 0, sizeof(UDP_DATAGRAM));
    if (seq != udp_num - 1)
    {
        memcpy(udp_now.getmessage(), buffer[seq], UDP_DATAGRAM_SIZE);
        udp_now.setlength(UDP_DATAGRAM_SIZE);
```

```
10          udp_now.setseq(seq);
11      }
12      else
13      {
14          memcpy(udp_now.getmessage(), buffer[seq], rest);
15          udp_now.setlength(rest);
16          udp_now.setseq(seq);
17      }
18      udp_now.setstate(SYN);
19      udp_now.calculate_checksum(); // 计算并设置校验和
20  //================================================================================
    ============
21      //发送数据报
22      if (sendto(sock, (char*)&udp_now, sizeof(udp_now), 0, (SOCKADDR*)&server,
    l) == SOCKET_ERROR) {
23          cout << "序列为" << seq << "的数据报文发送错误，错误码：" <<
    WSAGetLastError() << endl;
24          return 0;
25      }
26      //cout << "序列为" << seq << "的数据报文发送成功！"  << endl;
27
28  // 注意差错的报文也是未确认，启动超时检测重传机制
29      int start = clock(); // 记录发送起始时间
30
31      while (true)
32      {
33          int end = clock(); // 获取当前时间
34
35          // 等待报文确认，超时重传
36          if (end - start > timelimit_udp)
37          {
38              print_red_message(seq);
39              //cout << "序列为" <<  << "的报文超时未确认，触发重传机制" << endl;
40              //发送数据报
41              if (sendto(sock, (char*)&udp_now, sizeof(udp_now), 0,
    (SOCKADDR*)&server, l) == SOCKET_ERROR) {
42                  //cout << "序列为" << seq << "的数据报文发送错误，错误码：" <<
    WSAGetLastError() << endl;
43                  return 0;
44              }
45              else
46                  //cout << "序列为" << seq << "的数据报文发送成功！" << endl;
47                  start = clock();  // 重置起始时间
48          }
49
50          //线程退出的条件
```

```
        if (acked[seq])
            return 1; // 返回 1 表示消息发送成功并被确认

        //Sleep(50);
    }

}
```

2.2.3滑动窗口的更新

滑动窗口协议的核心是通过动态调整窗口范围，在有限大小的窗口内高效管理数据报文的发送和确认。`windowsending` 是一个阻塞函数，用于处理滑动窗口的动态调整。**滑动窗口更新函数直接以循环形式阻塞在主进程中，直到所有数据报文都被成功发送并确认。**本函数实现了滑动窗口的核心操作——**窗口右移**，从而支持持续发送和确认新数据。

✅ **判断滑动窗口底部是否可移动**

- 使用 `acked[windowbase]` 检查滑动窗口底部的分片是否已被确认：
  - 如果已确认，说明当前窗口的底部数据传输成功，可以右移。
  - 如果未确认，窗口底部保持不动，等待确认。

滑动窗口的右移

- 当底部（`windowbase`）数据确认后：
  - **底部右移**：将 `windowbase` 向右移动一位。
  - **顶部调整**：如果窗口顶部未到最后一个分片（`udp_num - 1`），将 `windowtop` 也向右移动一位。
- 每次滑动后，调用 `print_yellow_message(windowbase, windowtop)` 打印窗口范围，便于调试与观察。

退出条件

- 如果窗口底部到达最后一个分片（`windowbase == udp_num - 1`）且已确认，函数退出，表示所有数据报文的发送与确认完成。

```
//阻塞函数处理滑动窗口，一直循环到所有数据报文发送且接收确认完毕
void windowsending() {
    while (true) {
        if (acked[windowbase])
        {
            if (windowbase == udp_num - 1)
                break;//发送序列全部确认完毕
```

```
            windowbase++;      // 滑动窗口底部右移一位
            if (windowtop < udp_num - 1)
                windowtop++;
            print_yellow_message(windowbase, windowtop);
        }

    }
    return;
}
```

2.3累积确认设计与实现

本部分重点说明累计确认指针是怎么完成在接收端的更新以及是如何从接收端到发送端完成同步、以及如何在发送端的窗口数据报发送中发挥作用的。

2.3.1累积确认指针在接收端的更新与同步到发送端的过程（接收端回复累积确认ACK）

在接收端接收函数循环中，每一次循环都执行一段逻辑：**连续检查按序接收的报文：从累积确认点的下一位开始，逐步向后移动确认指针，直到遇到未确认的报文序列。**

✅ **标记接收状态**

- 将当前接收到的数据报标记为已确认，通过 `recv.getseq()` 获取当前接收的数据报的序列号，并在 `acked` 数组中将其对应位置设置为 `true`，表示已接收。

尝试更新累积确认点

- 从当前累积确认点 `accumulate_point` 的下一位开始遍历，检查是否存在按序接收的数据报，如果发现连续的已确认数据，临时累积确认点 `accumulate_point_now` 向前推进。

判断是否需要发送确认

- 如果累积确认点发生更新（`accumulate_point_now != accumulate_point`），发送 对应ACK 给发送端。

更新累积确认点

- 最后 `accumulate_point = accumulate_point_now` 更新累积确认指针：

```
acked[recv.getseq()] = true;
int accumulate_point_now = accumulate_point;
for (int k = accumulate_point + 1; k <= recv.getseq(); k++)
{
    if (acked[k])
```

```
        accumulate_point_now++;
    else
        break;
}
if (accumulate_point_now != accumulate_point)
{
    //回复确认
    UDP_DATAGRAM sen;
    memset(&sen, 0, sizeof(UDP_DATAGRAM));
    sen.setack(accumulate_point_now);
    sen.setstate(ACK);
    sendto(sock, (char*)&sen, sizeof(sen), 0, (SOCKADDR*)&client, l);
    cout << endl;
    cout << "累积确认到" << accumulate_point_now << "的数据报正确接收！" << endl;
    cout << "================================================================"
<< endl;
    accumulate_point = accumulate_point_now;
}
```

那ack[]数组在接收端如何完成更新呢，需要接收循环函数的完整处理逻辑：

✅ 接收端循环函数 `receiveandack` 的功能概括

该函数实现了基于 UDP 的文件接收和累积确认机制，具备以下主要功能：

1. **持续接收数据报**
   使用 `recvfrom` 不断接收来自客户端的 UDP 数据报，进行处理。
2. **随机丢包模拟**
   - 通过生成随机数模拟 1% 的随机丢包场景，跳过部分数据报的处理以测试协议的鲁棒性。
3. **校验和检查与数据处理**
   - 如果接收到的数据报符合以下条件：
     - 状态为 `SYN` 且无 `FIN` 标志。
     - 校验和正确。
       则：
   - **更新累积确认指针**：更新 `acked` 数组，累积确认已接收的连续数据报。
   - **发送累积确认**：发送包含最新累积确认点的 `ACK` 报文给客户端。
4. **文件信息处理**
   - 对于序列号为 `0` 的数据报，提取初始文件信息（如文件名、总数据报数量 `udp_num` 和最后一个数据报的大小 `rest`），并设置服务器文件路径。

5. **数据存储**
   - 对于非初始数据报，将其有效负载复制到 `buffer` 中，对应序列号存储。
6. **错误反馈**
   - 如果数据报校验和错误，则发送 `REQ` 报文通知客户端重传。
7. **文件传输结束确认**
   - 当接收到文件发送完成的报文（带 `FIN` 标志）时，发送 `ACK` + `FIN` 报文确认传输完成，并退出循环。
8. **返回传输结果**
   - 返回 `1` 表示成功接收并处理所有数据报。

```
int receiveandack() {
    UDP_DATAGRAM recv;
    while(true){
    memset(&recv, 0, sizeof(UDP_DATAGRAM));
    recvfrom(sock, (char*)&recv, sizeof(recv), 0, (SOCKADDR*)&client, &l);

//=============================================================================

    //真：随机丢包
     //设置随机种子为当前时间
    countcut++;
    // 生成范围为 1 到 10000 的随机数
    int a = countcut % 100;//随机丢包率，1%
    if (!a)
        continue;//跳过后面操作

//=============================================================================
=
    if (recv.getstate()&SYN && (!(recv.getstate() &FIN)) &&
recv.check_checksum())
    {
        acked[recv.getseq()] = true;
        int accumulate_point_now = accumulate_point;
        for (int k = accumulate_point + 1; k <= recv.getseq(); k++)
        {
            if (acked[k])
                accumulate_point_now++;
            else
                break;
        }
        if (accumulate_point_now != accumulate_point)
```

```
27          {
28              //回复确认
29              UDP_DATAGRAM sen;
30              memset(&sen, 0, sizeof(UDP_DATAGRAM));
31              sen.setack(accumulate_point_now);
32              sen.setstate(ACK);
33              sendto(sock, (char*)&sen, sizeof(sen), 0, (SOCKADDR*)&client, l);
34              cout << endl;
35              cout << "累积确认到" << accumulate_point_now << "的数据报正确接收！"
    << endl;
36              cout <<
    "================================================================" << endl;
37              accumulate_point = accumulate_point_now;
38          }
39
40          if (recv.getseq() == 0)
41          {
42
43              // 接收初始的文件传输请求报文（包含文件名信息和UDP个数信息）
44              // 解析接收到的报文，获取 udp_num 和文件名
45              // 用于存放文件名
46                  char extractedFileName[100];
47
48                  char fileInfo[1024];
49                  memset(fileInfo, 0, sizeof(fileInfo));
50                  strcpy(fileInfo, recv.getmessage());
51
52
53                  sscanf(fileInfo, "%d %s %d", &udp_num, extractedFileName,
    &rest);
54                  strcpy(FileName, SERVER_FILE_PATH);
55                  strcat(FileName, extractedFileName);
56
57          }
58          else {
59              memcpy(buffer[recv.getseq()], recv.getmessage(), recv.getlength());
60          }
61      }
62
63
64      else if(recv.getstate()&SYN && (!(recv.getstate() & FIN))){
65          UDP_DATAGRAM sen;
66          memset(&sen, 0, sizeof(UDP_DATAGRAM));
67          sen.setack(recv.getseq());
68          sen.setstate(REQ);
69          sendto(sock, (char*)&sen, sizeof(sen), 0, (SOCKADDR*)&client, l);
70      }
```

```
71      else {
72          //接收到文件发送确认报文
73          UDP_DATAGRAM sen;
74          memset(&sen, 0, sizeof(UDP_DATAGRAM));
75          sen.setstate(ACK);
76          sen.setstate(FIN);
77          sendto(sock, (char*)&sen, sizeof(sen), 0, (SOCKADDR*)&client, l);
78
79          break;
80      }
81      }
82      return 1;
83  }
```

2.3.2累积确认指针（accumulate_point）在发送端的作用

在接收ACK回复线程中，**收到的ACK回复的对应序列值就是接收端回复的累积确认指针的同步**，然后发送端根据新收到的累计确认指针来完成原累积确认指针的更新和ack[]确认数组的更新，ack[]的更新从而影响当前滑动窗口内UDP数据报的发送选择。

✅ 1. **判断是否为 ACK 报文**

接收到一个数据报后，通过检查其状态字段 (recv.getstate()) 判断是否为 ACK 报文：

- 如果是 ACK 报文，则执行累积确认逻辑。
- 累积确认表示：序列号小于或等于当前 ACK 序列号的所有数据报均已成功接收。

2. **更新确认数组 (`acked[]`)**

确认数组 `acked[]` 是一个布尔类型的数组，用于记录每个分片是否已被确认：

- **目标**：将 ACK 序列号之前的所有数据标记为已确认。
- **更新逻辑**：
  - 遍历序列号从 `accumulate_point + 1` 到 `recv.getack()` 的范围。
  - 对该范围内的每个分片设置 `acked[j] = 1` ，表示它们已确认。

3. **更新累计确认指针 (`accumulate_point`)**

累计确认指针 `accumulate_point` 是一个标志变量，用于记录目前已确认的最大序列号：

- 每次接收到新的 ACK 报文后，将 `accumulate_point` 更新为 ACK 的序列号 (`recv.getack()`)，表示所有分片到该序列号为止都已确认。

```
    int receive() {
        while (true)
        {
            UDP_DATAGRAM recv;
            memset(&recv, 0, sizeof(UDP_DATAGRAM));
            recvfrom(sock, (char*)&recv, sizeof(UDP_DATAGRAM), 0, (SOCKADDR*)&server, &l);
            if (recv.getstate() & REQ)
            {
                cout << "发送序列为" << recv.getack() << "的报文传输出现数据错误！" << endl;
                continue;
            }

            else if (recv.getstate() & ACK)
            {
                cout << "累积确认，序列号" << recv.getack() << "之前的所有数据报已确认！" << endl;
                for(int j= accumulate_point+1;j<= recv.getack();j++)
                    acked[j] = 1;           // 更新状态，表示累积确认
                accumulate_point = recv.getack();//更新累积确认指针
                cout << "================================================================" << endl;



            }
            //线程退出条件
            //条件：若滑动窗口到最后，且检查窗口内的是否都已确认
            bool exitjudge = true;
            for (int i = windowbase; i <= udp_num - 1; i++)
                if (!acked[i])
                    exitjudge = false;

            if (exitjudge)
                return 1; // 如果确认号是最后一个分片，退出接收线程
        }
    }

```

3.【程序测试结果】

目标：传输文件：3.png

丢包率设置：1%

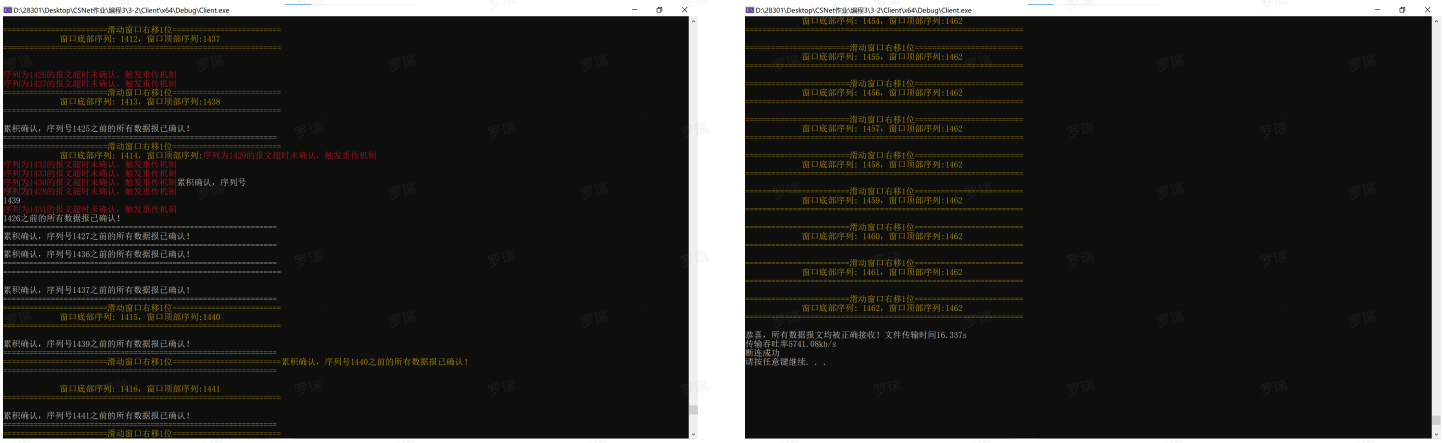

图3：发送端日志记录

图4：接收端日志记录

文件传输时间16.337s，传输吞吐率5741.08kb/s，传输的文件已自动命名处理，可以直接打开，试验成功。

4.【不同方法性能对比与反思】

设置在停等协议的重传超限时间为滑动窗口重传超限的1/7左右（使其都达到程序响应的临近极限）

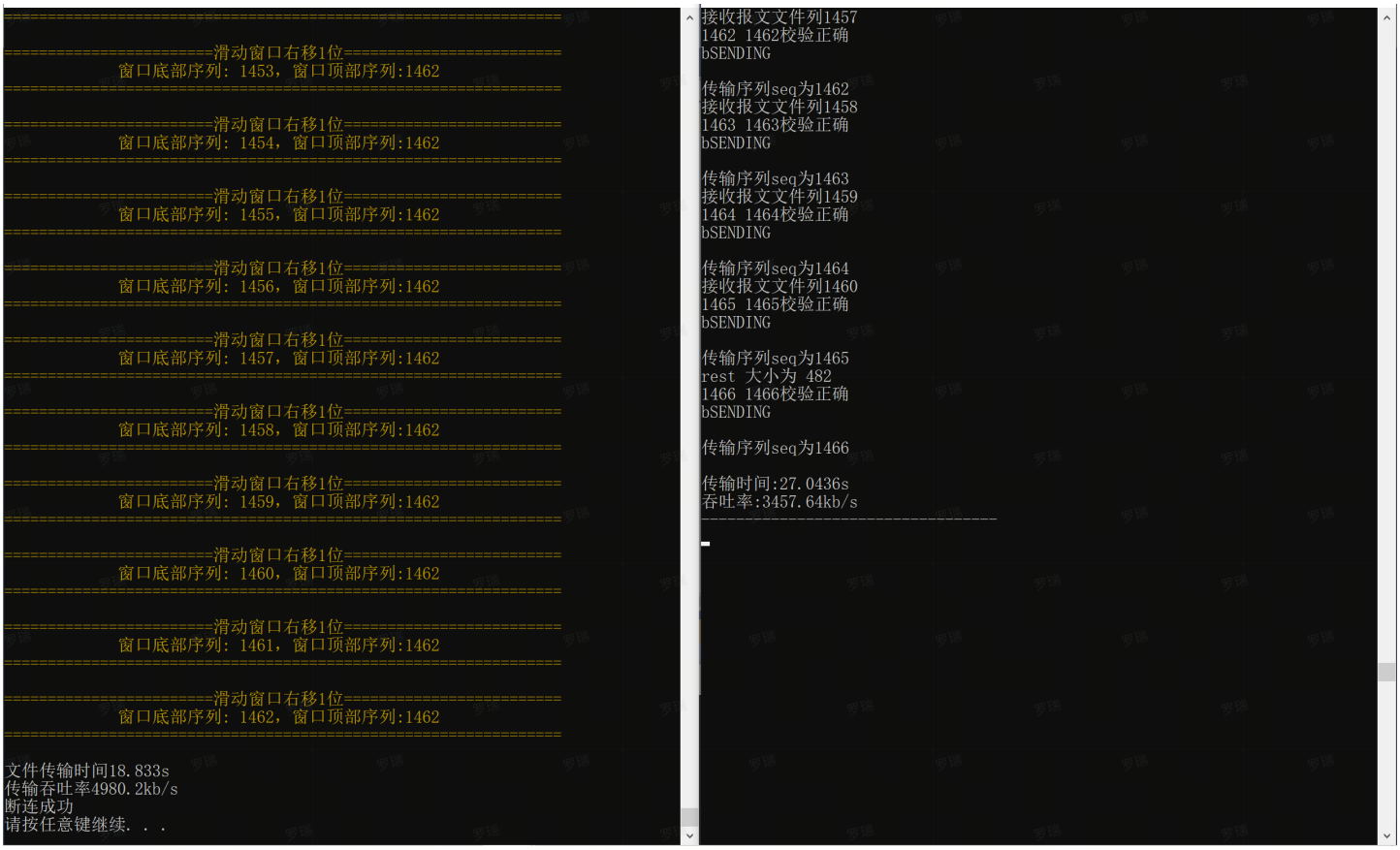

图5：丢包率为25%的情况下对比

此时滑动窗口比停等协议快30%

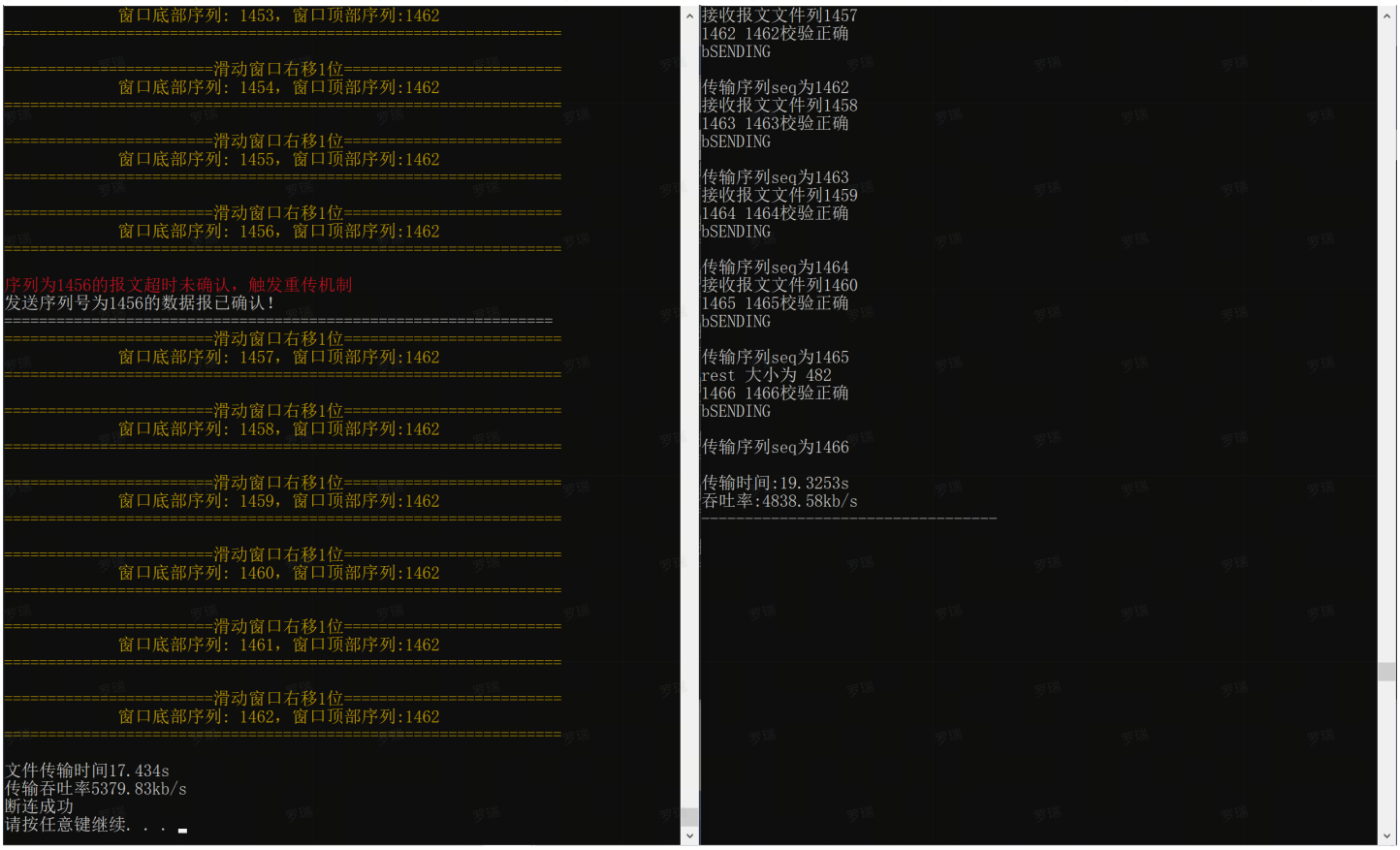

图6：丢包率为10%的情况下对比

此时滑动窗口比停等协议快11%

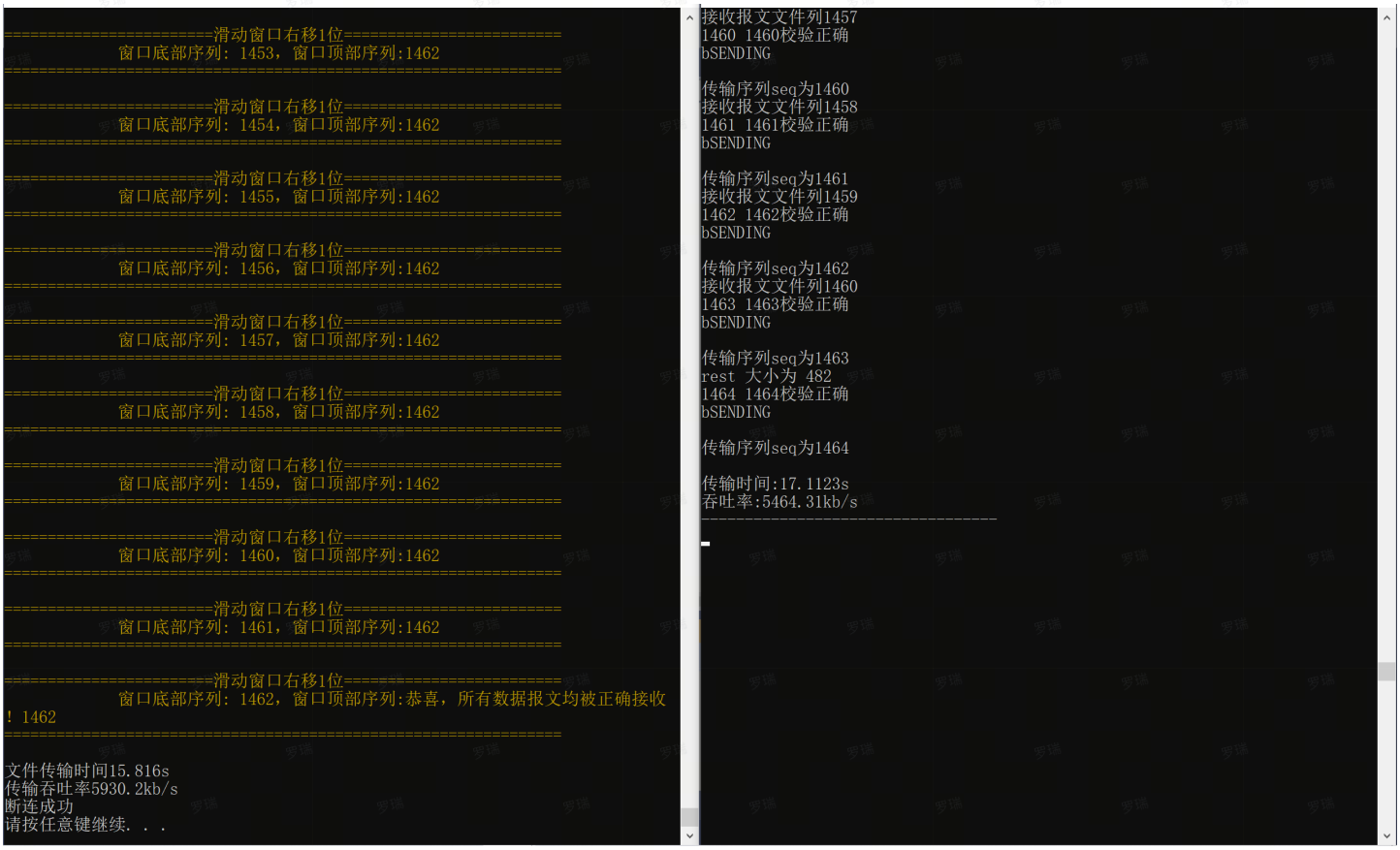

图7：丢包率为1%的情况下对比

滑动窗口比停等协议快8%

✅ 可见，当网络环境越差时，滑动窗口相对停等协议的性能越好，原因可能是我在滑动窗口内的数据报发送真正实现了互不干扰的多线程处理，这样前一个未确认的数据报不会影响窗口内后面几帧数据报的发送线程的处理，这是优于停等协议的核心因素。