计算机网络实验报告

实验3-3：选择确认

姓名：谢雯菲 学号：2110803

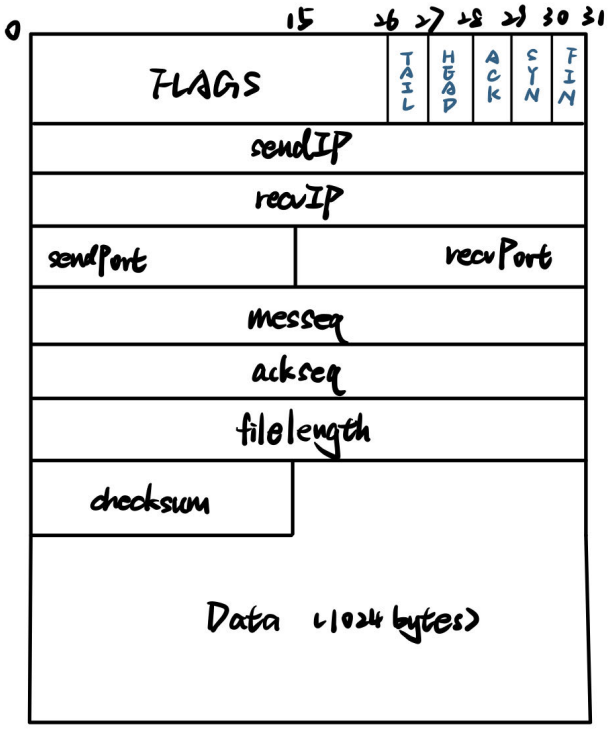
实验要求

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。

协议设计

数据包格式

设计的数据报头部如下所示：



- 32位FLAGS位（后五个比特位分别为：TAIL（最后一个文件标志位）、HEAD（头文件标志位）、ACK（确认位）、SYN（同步位）、FIN（结束位））
- 32位发送端IP地址（没用到）
- 32位接收端IP地址（没用到）
- 16位发送端端口号，16位接收端端口号（没用到）
- 32位发送序列号（实际只用了1位）
- 32位确认序列号（实际只用了1位）
- 32位文件长度
- 16位校验和

数据包格式的代码和3-1相同，在此不过多赘述。

选择重传

选择重传协议中，发送方仅重传那些怀疑接受方出错的分组，从而避免不必要的重传。

选择重传中发送方和接受方的窗口如下所示：

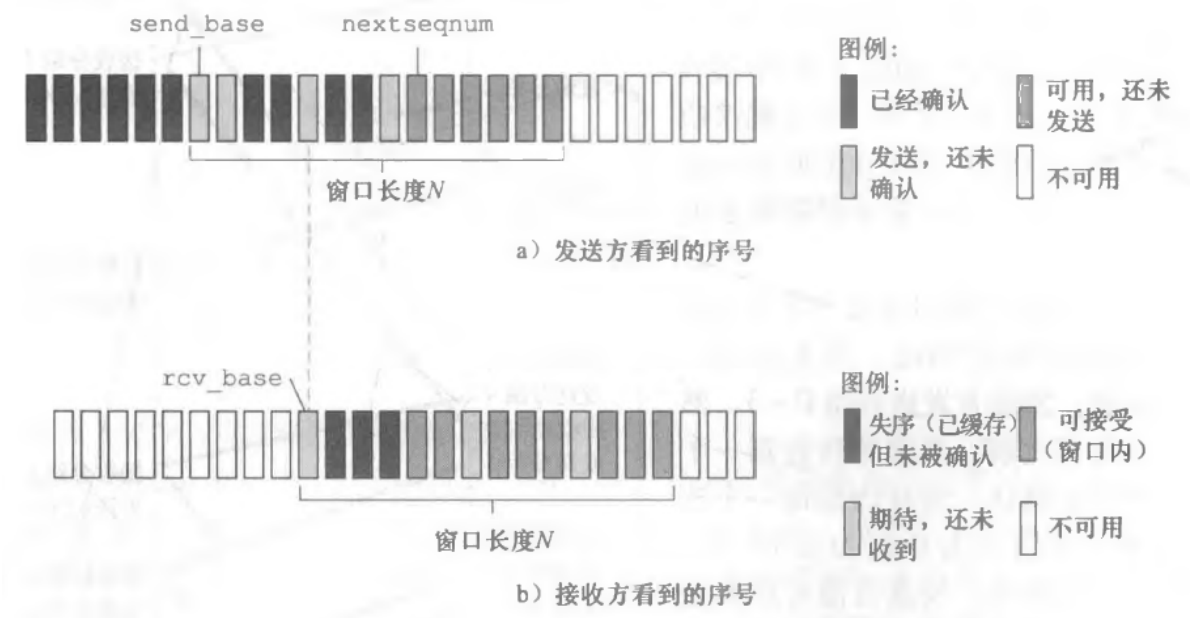


图 3-23　选择重传（SR）发送方与接收方的序号空间

使用窗口长度N（程序中设置为`windowsize`，窗口长度必须小于或等于序号空间大小的一半）来限制流水线中未完成、未被确认的分组数。接受方将确认一个正确接收的分组而不管其是否按序。失序的分组将被缓存直到所有丢失分组（即序号更小的分组）都被收到为止。

发送方交互

流程

根据教材，发送端需要能对以下事件做出反应：

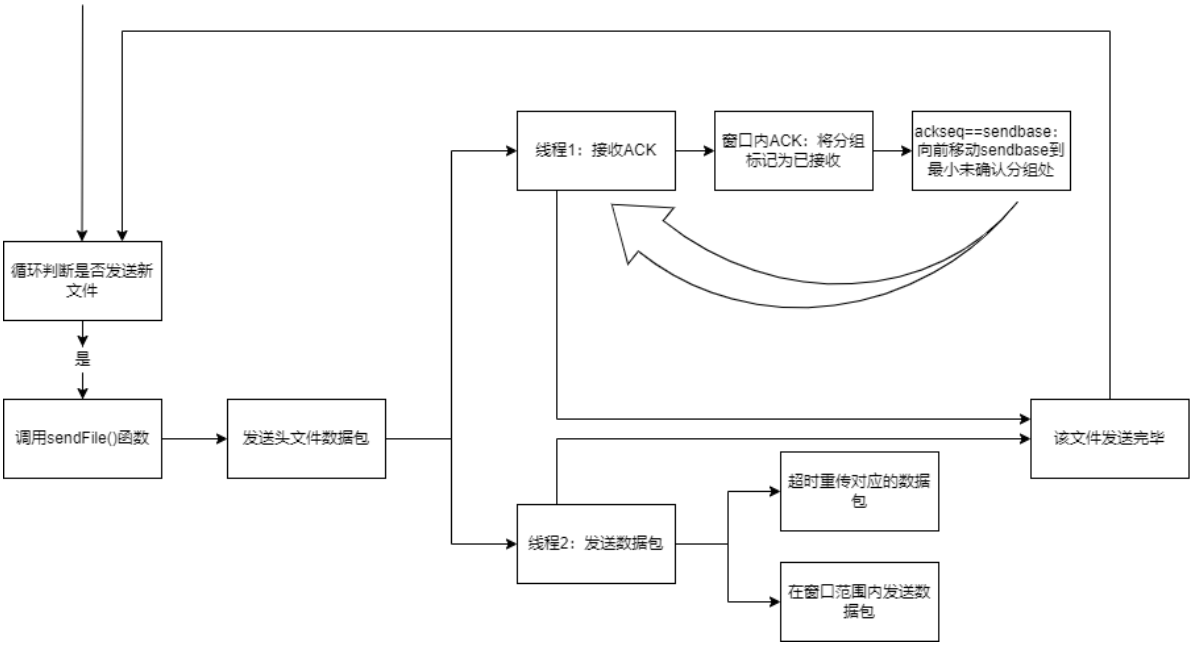
1. 从上层收到数据。当从上层接收到数据后，SR 发送方检查下一个可用于该分组的序号。如果序号位于发送方的窗口内，则将数据打包并发送；否则就像在 GBN 中一样，要么将数据缓存，要么将其返回给上层以便以后传输。

2. 超时。定时器再次被用来防止丢失分组。然而，现在每个分组必须拥有其自己的逻辑定时器，因为超时发生后只能发送一个分组。可以使用单个硬件定时器模拟多个逻辑定时器的操作［Varghese 1997］。

3. 收到 ACK。如果收到 ACK，倘若该分组序号在窗口内，则 SR 发送方将那个被确认的分组标记为已接收。如果该分组的序号等于 send_base，则窗口基序号向前移动到具有最小序号的未确认分组处。如果窗口移动了并且有序号落在窗口内的未发送分组，则发送这些分组。

图 3-24　SR 发送方的事件与动作

设计发送方流程如下所示：



总体上较实验3-2没有改变，将发送方的`sendFile()`函数分为两个线程：

1. 一个线程`recvThread`接收ACK：如果收到的ACK是窗口内ACK，即ACK序号在[sendbase, nextseqnum)范围内，则将对应窗口的分组设为已确认；如果收到的ACK序号等于sendbase，向前移动sendbase到最小未确认分组处，在代码中依靠循环实现该功能。
2. 一个线程发送数据包：在发送新的数据包之前调用`resendMes()`函数重传窗口内超时的数据包；然后，如果窗口未满且数据包尚未发送完毕，则继续向后发送数据包。

接收ACK

在接收ACK的线程中，需要按照上述描述对接收ACK的线程进行判断。并且在确认窗口中某个位置的包已接收后清空窗口缓存，防止影响后续数据传输。

`recvThread`代码实现如下：

```
void recvThread() {
    while (sendbase <= packetNum) {
        if (recvfrom(clisock, (char*)recvMes, BUFFER_SIZE, 0, (SOCKADDR*)&(seraddr), &seraddr_len) > 0) {
            printTime();
            if (recvMes->FLAGS == ACK && !isCorrupt(recvMes)) {
                int recvACKnum = recvMes->ackseq;
                cout << "收到ACK: " << recvACKnum << endl;
                //如果是窗口内的ACK
                if (recvACKnum >= sendbase && recvACKnum < nextseqnum) {
                    std::lock_guard<std::mutex> mylockguard(mylock);
                    //将分组标记为已接收,清空包内缓存
                    ACKMes[recvACKnum % windowsize] = true;
                    sendMes[recvACKnum % windowsize]->clearMes();
                    cout << "分组已接收，清空包内缓存" << endl;
                    //如果等于sendbase
                    if (recvACKnum == sendbase) {
                        //从recvACKnum后的第一个包开始查找最小未确认分组
                        for (int i = sendbase; i < nextseqnum; i++) {
                            int index = i % windowsize;
                            if (!ACKMes[index]) {
                                sendbase = i;
```

```
                                break;
                            }
                            //如果已经到最后一个，说明已经全部确认了
                            if (i == nextseqnum - 1) {
                                sendbase = nextseqnum;
                            }
                        }
                        cout << "窗口边界base变为：" << sendbase << endl;
                    }

                }
            }
            else {
                cout << "接收的包已损坏或不是ACK包" << endl;
            }
        }
    }
}
```

超时判断

程序中发送方窗口的发送缓冲区用一个数组`sendMes`表示，另外用一个数组`StartTimer`表示窗口内每个分组的计时器，还有一个数组`ACKMes`表示每个分组是否已经确认。

在发送每个包时，更新该包的计时器，在调用发送文件函数时取得窗口内尚未确认的分组的时间差判断是否超时。如果超时，只需重传当前分组的数据包即可。重传完成后，需要更新超时计时器。

`resendMes()`函数代码实现如下：

```
int sendbase = 0;
int nextseqnum = 0; //下一个要发送包的序号
message** sendMes;
message* recvMes;
int lostnum = 0;//丢包数
int allsendnum = 0; //所有发送的包数
//定时器数组
clock_t* StartTimer;

//窗口内包是否确认
bool* ACKMes;

//重传函数
void resendMes() {
    for (int i = sendbase; i < nextseqnum; i++) {
        int index = i % windowsize;
        //如果该包还未被确认，判断是否需要重传
        if (!ACKMes[index]) {
            clock_t StopTimer = clock();
            //单个包的超时时间
            double interval = (StopTimer - StartTimer[index]) * 1000 /
CLOCKS_PER_SEC;
            if ((interval > timeout)) { //超时重传
                cout << "超时时间为：" << interval << endl;
                cout << "--重传第" << i << "个数据包--" << endl;
                //加锁防止冲突
```

```
                std::lock_guard<std::mutex> mylockguard(mylock);
                //重传当前包
                sendto(clisock, (char*)sendMes[index], BUFFER_SIZE, 0, (SOCKADDR*)&seraddr, seraddr_len);
                printTime();
                cout << "【重传第" << i << "个数据包】";
                printSendMessage(sendMes[index]);
                lostnum++;
                allsendnum++;
                //重新设置超时计时器
                StartTimer[index] = clock();
            }
        }
    }
}
```

发送文件

根据以上设计，编程实现的`sendFile()`函数如下所示：

```
//发送文件
void sendFile() {
    sendMes = new message * [windowsize];
    sendbase = 0;
    nextseqnum = 0; //下一个要发送包的序号
    //初始化定时器数组
    StartTimer = new clock_t[windowsize];
    //初始化记录是否确认数组
    ACKMes = new bool[windowsize];
    for (int i = 0; i < windowsize; i++) {
        ACKMes[i] = false;
    }
    //用于计算丢包率
    allsendnum = 0; //所有发送的包数
    lostnum = 0;//丢包数
    //先发送一个记录文件名的数据包，并设置HEAD标志位为1，表示开始文件传输
    for (int i = 0; i < windowsize; i++) {
        sendMes[i] = new message();
    }
    recvMes = new message();
    printTime();
    clock_t start = clock();
    cout << "发送文件头数据包……" << endl;
    char* fileNamec = new char[128];
    strcpy(fileNamec, fileName.c_str());
    sendMes[nextseqnum-sendbase]->setHEAD(0, fileSize, fileNamec);
    sendMes[nextseqnum - sendbase]->setchecksum((uint16_t*)sendMes[nextseqnum - sendbase]);
    printSendMessage(sendMes[nextseqnum - sendbase]);
    sendto(clisock, (char*)sendMes[nextseqnum - sendbase], BUFFER_SIZE, 0, (SOCKADDR*)&seraddr, seraddr_len);
    //设置定时器
    StartTimer[0] = clock();
    allsendnum++;
    nextseqnum++;
```

```
    //创建接收消息的线程
    thread rThread(recvThread);
    while (sendbase <= packetNum) { //当base小于发送包数量时，继续发送
        //先调用超时函数查看是否有需要超时重传的包
        resendMes();
        //如果窗口没满且没有发送完毕
        if (nextseqnum < sendbase + windowsize && nextseqnum <= packetNum) {
            printTime();
            {
                std::lock_guard<std::mutex> mylockguard(mylock);
                cout << "正在发送第" << nextseqnum << "个数据包！" << endl;
                cout << "base为：" << sendbase << endl;
                int index = nextseqnum % windowsize;
                //设置发送包的计数器和确认位
                ACKMes[index] = false;
                StartTimer[index] = clock();
                if (nextseqnum == packetNum) { //如果是最后一个文件，设置文件尾
                    sendMes[index]->FLAGS = TAIL;
                }
                sendMes[index]->fillData(nextseqnum, (fileSize - (nextseqnum - 1) * 1024) > 1024 ? 1024 : fileSize - (nextseqnum - 1) * 1024, fileBuffer + (nextseqnum - 1) * 1024);
                sendMes[index]->setchecksum((uint16_t*)sendMes[index]);
                printSendMessage(sendMes[index]);
                res = sendto(clisock, (char*)sendMes[index], BUFFER_SIZE, 0, (SOCKADDR*)&seraddr, seraddr_len);
                if (res > 0) {
                    allsendnum++;
                    cout << "数据包发送成功！" << endl;
                    nextseqnum++;
                }
                else {
                    cout << "数据包发送失败！" << endl;
                }
            }
        }
        else if (nextseqnum >= sendbase + windowsize || nextseqnum > packetNum) {
            cout << "窗口已满或该文件已发送完毕，等待确认。" << endl;
            Sleep(50);
        }
        Sleep(haha);
    }
    //所有包都已经确认了
    rThread.join();
    printTime();
    cout << "当前文件发送完毕。" << endl;
    clock_t end = clock();
    cout << "总传输时间为：" << (end - start)*1000 / CLOCKS_PER_SEC << "毫秒" << endl;
    cout << "吞吐率为：" << (float)fileSize / ((end - start) * 1000 / CLOCKS_PER_SEC) << "字节/毫秒" << endl;
    cout << "错误（丢包+超时）率为：" << (float)lostnum / allsendnum << endl;
    //清理内存
    for (int i = 0; i < windowsize; i++) {
```

```
        delete sendMes[i];
    }
    delete[] sendMes;
    delete recvMes;
    delete[] fileNamec;
    delete[] ACKMes;
    delete[] StartTimer;
}
```

接收方交互

流程

根据教材，接收方需要对以下事件做出反应：

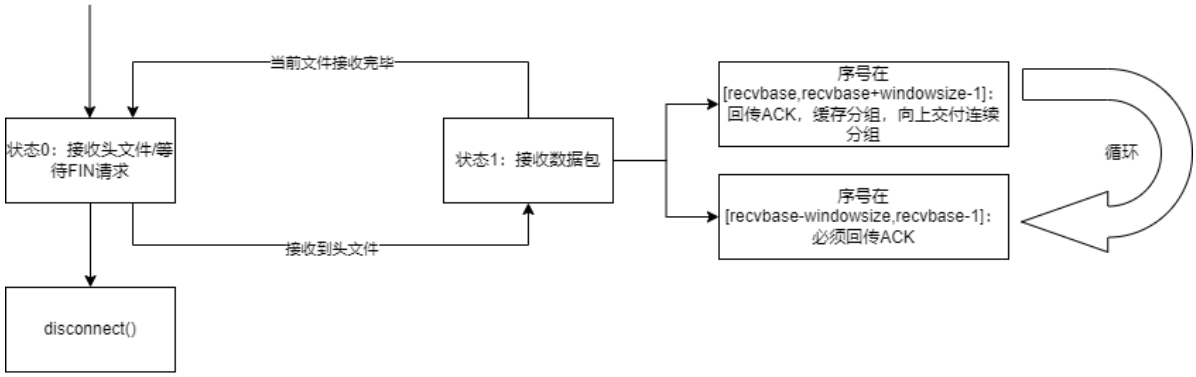
1. 序号在 [rcv_base, rcv_base + N − 1] 内的分组被正确接收。在此情况下，收到的分组落在接收方的窗口内，一个选择 ACK 被回送给发送方。如果该分组以前没收到过，则缓存该分组。如果该分组的序号等于接收窗口的基序号（图 3-23 中的 rcv_base），则该分组以及以前缓存的序号连续的（起始于 rcv_base 的）分组交付给上层。然后，接收窗口按向前移动分组的编号向上交付这些分组。举例子来说，考虑一下图 3-26。当收到一个序号为 rcv_base = 2的分组时，该分组及分组 3、4、5 可被交付给上层。

2. 序号在 [rcv_base − N, rcv_base − 1] 内的分组被正确收到。在此情况下，必须产生一个 ACK，即使该分组是接收方以前已确认过的分组。

3. 其他情况。忽略该分组。

图 3-25　SR 接收方的事件与动作

根据以上描述设计接收方流程如下所示：



在`receiveFile()`函数中，使用`while`循环判断接收端状态，接收端有两个状态：

- 状态0：等待接收文件头数据包或等待接收FIN请求
- 状态1：接收发送端数据包，并作出相应的反应

接收缓冲区

接收方缓冲区用一个数组`recvWindow`表示，还用了一个数组`haveMes`表示对应的窗口位置是否有数据包。

接收方接收数据包时，回传对应序号的ACK包，如果收到的包序号`messeq`在[recvbase, recvbase+windowsize-1]之间，则将当前包中数据缓存至缓冲区，并且用一个循环向上交付从`recvbase`开始的连续分组，并且清空窗口中已经交付的包的缓冲区，避免影响后续缓存；如果序号在[recvbase-windowsize, recvbase-1]之间，则必须回传ACK告知发送方，即使接收方已经交付了对应的包；除了这两种情况以外的包都可以忽略。

`receiveFile()`函数代码实现如下：

```
//接收文件
void receiveFile() {
    bool flag = true;
    int state = 0;
    int res = 0;    //接收返回
    //初始化
    recvbase = 0;
    recvWindow = new message * [windowsize];
    haveMes = new bool[windowsize];
    for (int i = 0; i < windowsize; i++) {
        recvWindow[i] = new message();
        haveMes[i] = false;
    }
    recvMes = new message();
    while (flag) {  //循环接收文件数据包
        switch (state) {
        case 0: //等待数据包头文件状态
            res = recvfrom(sersock, (char*)recvMes, BUFFER_SIZE, 0, (SOCKADDR*)&(cliaddr), &cliaddr_len);
            if (res > 0) {
                if (randomNum(randomGen) <= LOSS_RATE) {//主动丢包
                    printTime();
                    cout << "主动丢包" << endl;
                }
                else {
                    if (randomNum(randomGen) <= DELAY_RATE) {//延时
                        int time = DELAY_BASE * randomNum(randomGen);
                        Sleep(time);
                        printTime();
                        cout << "延时" << time << "毫秒" << endl;
                    }
                    printTime();
                    printRecvMessage(recvMes);
                    if (recvMes->FLAGS == FIN) {
                        flag = false;
                        disconnect();
                    }
                    else if (!isCorrupt(recvMes) && recvMes->messeq == 0) {
                        if (recvMes->FLAGS == HEAD) {
                            cout << "接收数据包头成功！" << endl;
                            //更新窗口
                            recvbase = 1;
                            //初始化接收文件缓冲区
                            recvSize = 0;
                            fileSize = recvMes->filelen;
                            fileBuffer = new char[fileSize];
                            fileName = new char[128];
                            memcpy(fileName, recvMes->data, strlen(recvMes->data) + 1);
                            //计算文件包数量
                            packetnum = fileSize % 1024 ? fileSize / 1024 + 1 : fileSize / 1024;
                            cout << "开始接收来自发送端的文件，文件名为：" << fileName << endl;
```

```
51                              cout << "文件大小为：" << fileSize << "比特，总共需
要接收" << packetnum << "个数据包" << endl;
52                              cout << "等待发送文件数据包……" << endl;
53                              //发送ACK_0数据包
54                              sendACK(0);
55                              state = 1;
56                          }
57                          else {
58                              cout << "收到的数据包不是文件头，等待发送端重传……" <<
endl;
59                          }
60                      }
61                  }
62              }
63              recvMes->clearMes();
64              break;
65          case 1:
66              res = recvfrom(sersock, (char*)recvMes, BUFFER_SIZE, 0,
(SOCKADDR*)&(cliaddr), &cliaddr_len);
67              if (res > 0) {
68                  if (randomNum(randomGen) <= LOSS_RATE) {//主动丢包
69                      printTime();
70                      cout << "主动丢包" << endl;
71                  }
72                  else {
73                      if (randomNum(randomGen) <= DELAY_RATE) {//延时
74                          int time = DELAY_BASE * randomNum(randomGen);
75                          Sleep(time);
76                          printTime();
77                          cout << "延时" << time << "毫秒" << endl;
78                      }
79                      printTime();
80                      cout << "recvbase为：" << recvbase << endl;
81                      printRecvMessage(recvMes);
82                      //选择重传，两种情况
83                      int recvseq = recvMes->messeq;
84                      cout << "正在接收分组为：" << recvseq << endl;
85                      cout << "边界为：" << recvbase + windowsize - 1 << endl;
86                      if (!isCorrupt(recvMes) && recvseq >= recvbase &&
recvseq <= (recvbase + windowsize - 1)) {
87                          //缓存分组
88                          int index = recvseq % windowsize;
89                          recvWindow[index]->storeMes(recvMes);
90                          haveMes[index] = true;
91                          cout << "缓存该分组完毕" << endl;
92                          //向上交付连续分组
93                          for (int i = recvbase; i < recvbase + windowsize;
i++) {
94                              int j = i % windowsize;
95                              if (haveMes[j]) {
96                                  cout << "正在交付第" << i << "个分组" <<
endl;
97                                  memcpy(fileBuffer + recvSize,
recvWindow[j]->data, recvWindow[j]->filelen);
98                                  recvSize += recvWindow[j]->filelen;
```

```
                                if (recvWindow[j]->FLAGS == TAIL) {
                                    //如果是最后一个包，进入状态0，也就是等待头文件或者断开连接
                                    state = 0;
                                    cout << "当前文件接收完毕！" << endl;
                                    cout << "--------------------" << endl;
                                    myclear();
                                    //保存文件
                                    saveFile();
                                    break;
                                }
                                //交付后清空窗口数据
                                recvWindow[j]->clearMes();
                                haveMes[j] = false;
                            }
                            else {
                                //如果之后没有分组了，更新recvbase
                                recvbase = i;
                                break;
                            }
                            //如果窗口内分组全部保存完毕
                            if (i == recvbase + windowsize - 1) {
                                //更新recvbase;
                                recvbase = recvbase + windowsize;
                            }
                        }
                        //回传ACK
                        sendACK(recvseq);
                    }
                    //如果序号在[recvbase-windowsize, recvbase-1]之间
                    else if (recvseq >= (recvbase - windowsize) && recvseq <= (recvbase - 1)) {
                        //回传ACK
                        sendACK(recvseq);
                    }
                    else if (isCorrupt(recvMes)) {
                        cout << "包损坏" << endl;
                        cout << "校验和为：" << recvMes->checksum << endl;
                    }
                }
            }
            else {
                cout << "接受包错误！" << endl;
            }
            recvMes->clearMes();
            break;
        }
    }
    //清空内存
    delete recvMes;
    delete[] haveMes;
    for (int i = 0; i < windowsize; i++) {
        delete recvWindow[i];
    }
    delete[] recvWindow;
```

```
152 }
```

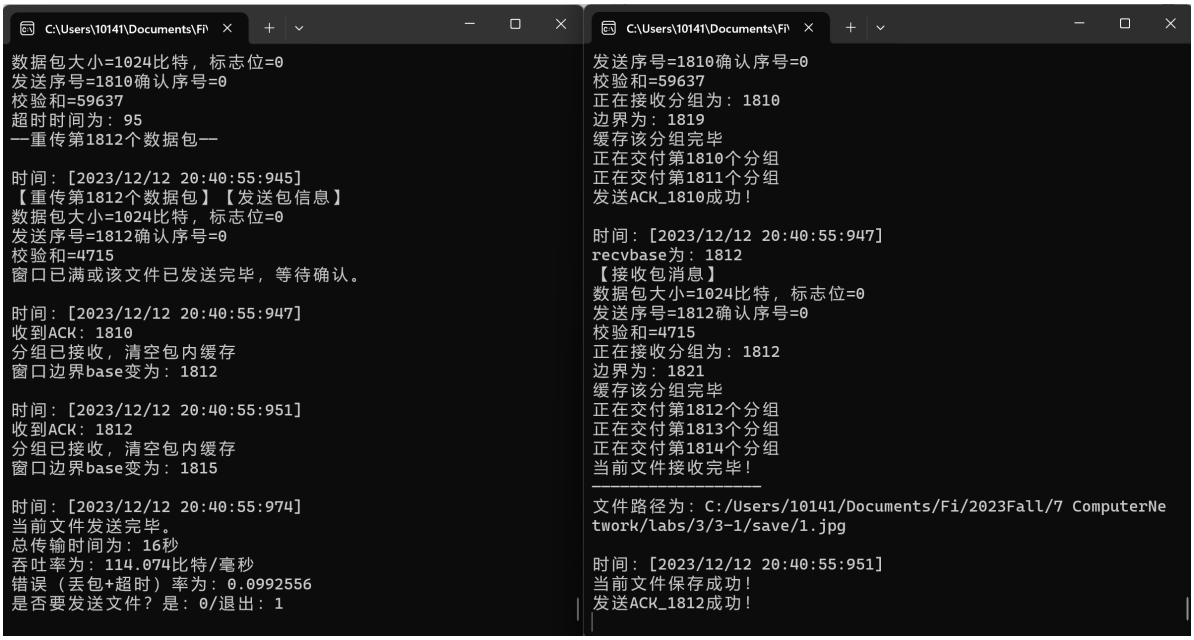
运行结果

丢包率0.1，窗口大小10

使用自己写的基于随机数的丢包，设置丢包率为0.1，发送方超时响应时间为80ms，在窗口大小为10时的各文件传输结果如下所示（左侧为发送端，右侧为接收端）：

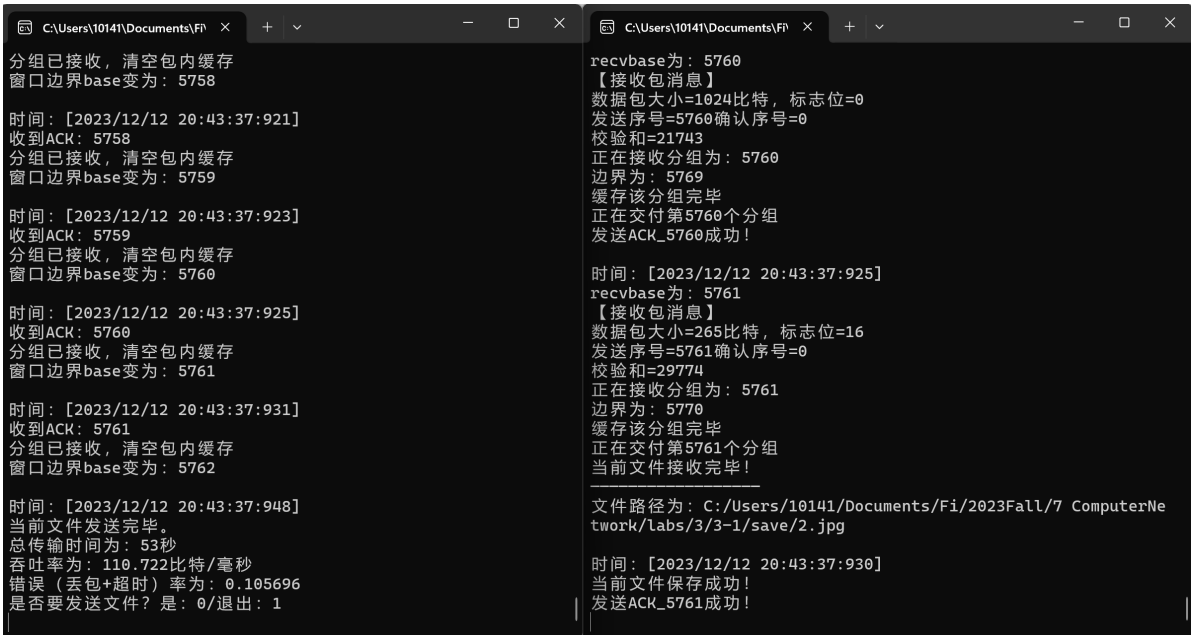
`1.jpg`：

传输时间为16秒，吞吐率为114比特/毫秒



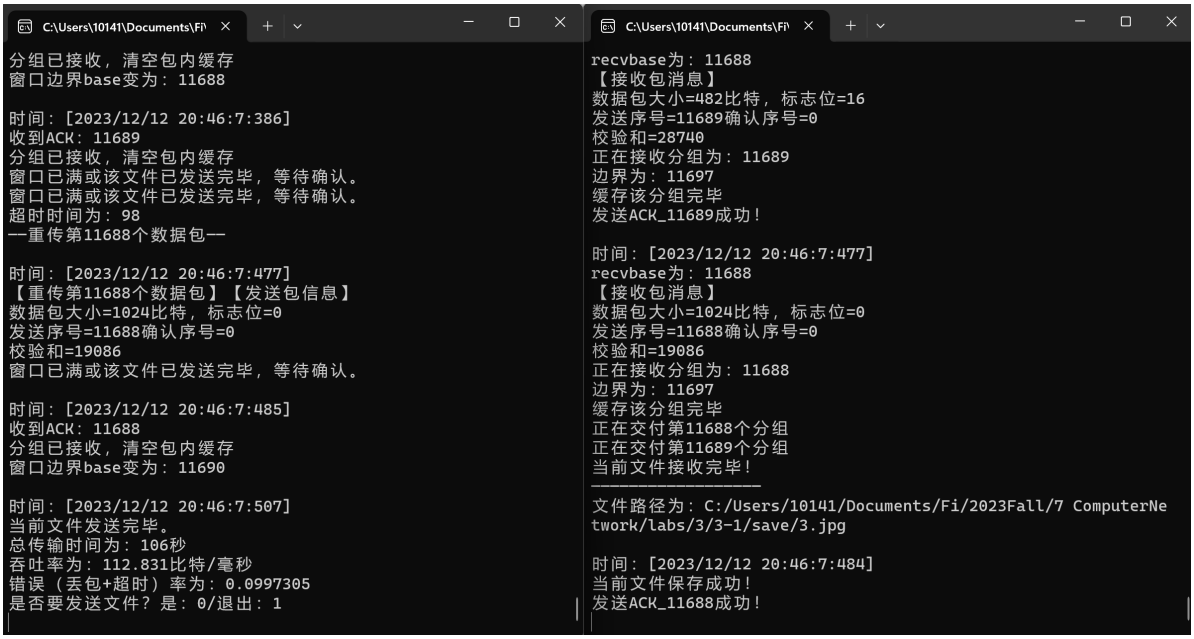
`2.jpg`：

传输时间为53秒，吞吐率为111比特/毫秒



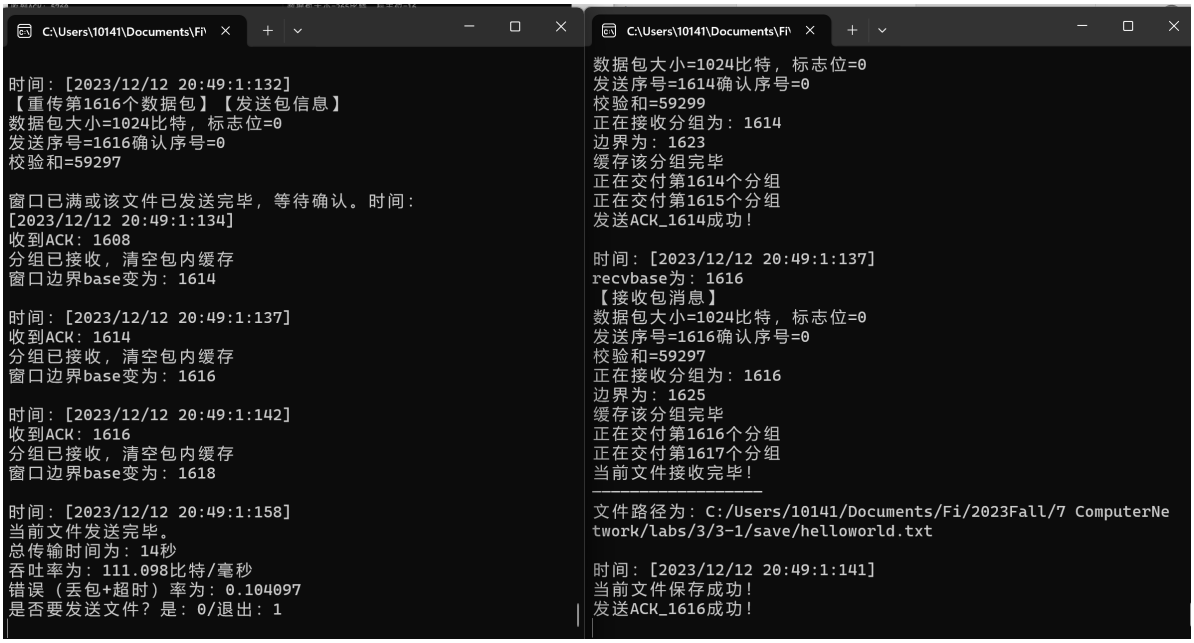
`3.jpg`：

传输时间为106秒，吞吐率为113比特/毫秒



`helloworld.txt`：

传输时间为14秒，吞吐率为111比特/毫秒

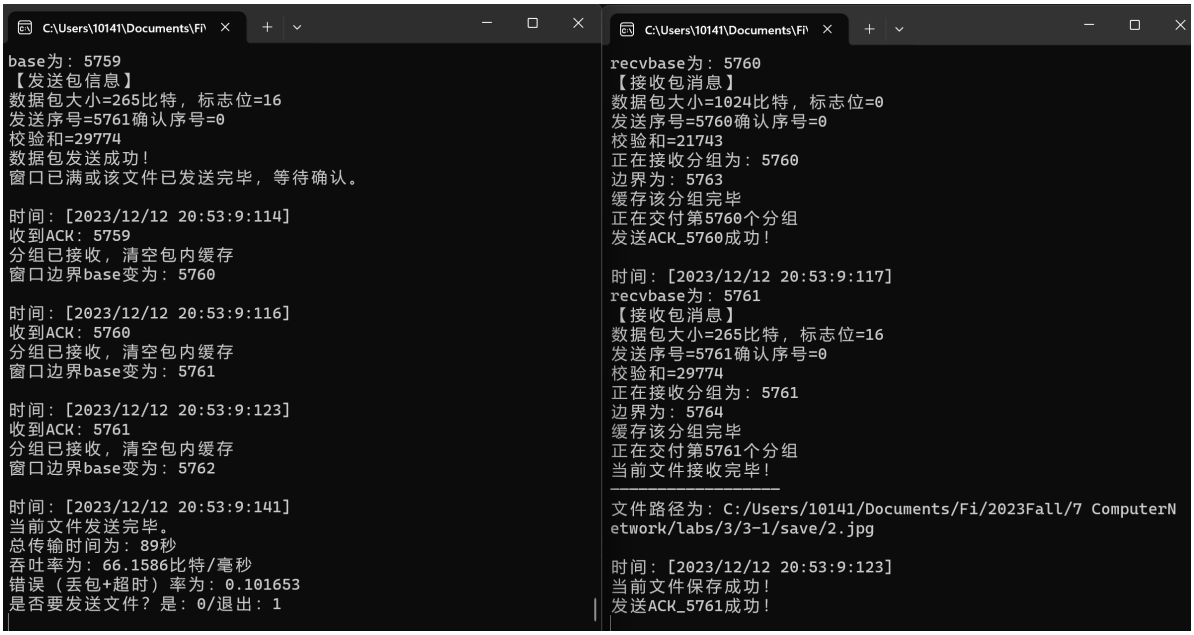


不同窗口大小吞吐率比较

在丢包率为0.1，滑动窗口大小不同的情况下进行测试传输`2.jpg`文件：

窗口大小为4时：

传输时间为89秒，吞吐率为66比特/毫秒

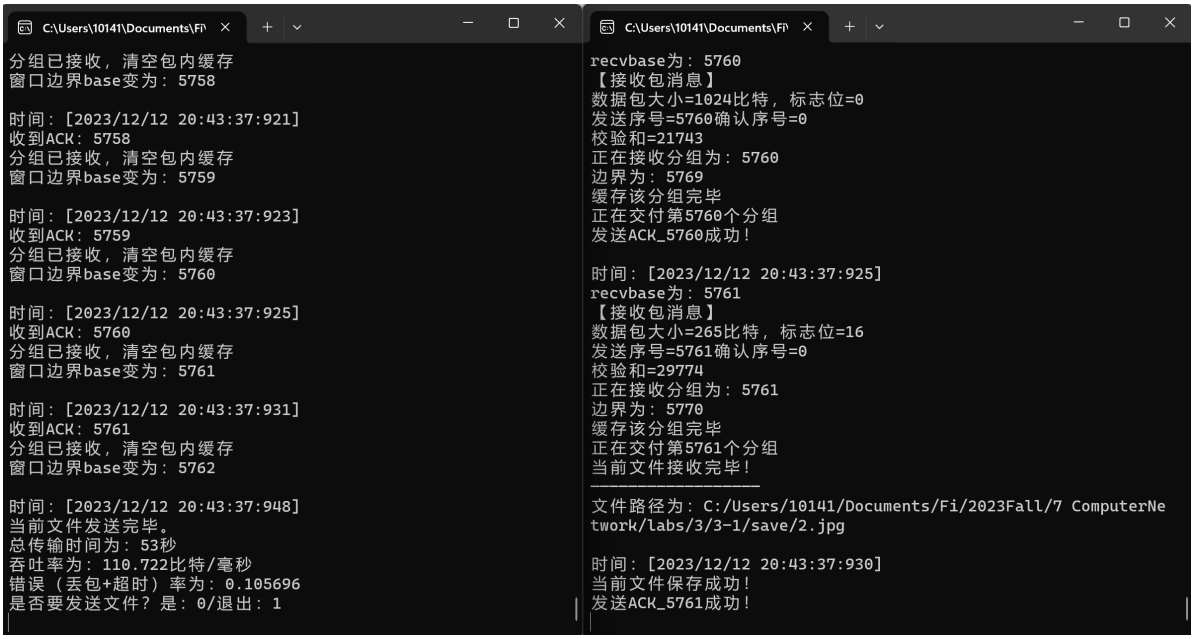


```
base为：5759
【发送包信息】
数据包大小=265比特，标志位=16
发送序号=5761确认序号=0
校验和=29774
数据包发送成功！
窗口已满或该文件已发送完毕，等待确认。

时间：[2023/12/12 20:53:9:114]
收到ACK：5759
分组已接收，清空包内缓存
窗口边界base变为：5760

时间：[2023/12/12 20:53:9:116]
收到ACK：5760
分组已接收，清空包内缓存
窗口边界base变为：5761

时间：[2023/12/12 20:53:9:123]
收到ACK：5761
分组已接收，清空包内缓存
窗口边界base变为：5762

时间：[2023/12/12 20:53:9:141]
当前文件发送完毕。
总传输时间为：89秒
吞吐率为：66.1586比特/毫秒
错误（丢包+超时）率为：0.101653
是否要发送文件？是：0/退出：1
```

```
recvbase为：5760
【接收包消息】
数据包大小=1024比特，标志位=0
发送序号=5760确认序号=0
校验和=21743
正在接收分组为：5760
边界为：5763
缓存该分组完毕
正在交付第5760个分组
发送ACK_5760成功！

时间：[2023/12/12 20:53:9:117]
recvbase为：5761
【接收包消息】
数据包大小=265比特，标志位=16
发送序号=5761确认序号=0
校验和=29774
正在接收分组为：5761
边界为：5764
缓存该分组完毕
正在交付第5761个分组
当前文件接收完毕！
————————————————
文件路径为：C:/Users/10141/Documents/Fi/2023Fall/7 ComputerNetwork/labs/3/3-1/save/2.jpg

时间：[2023/12/12 20:53:9:123]
当前文件保存成功！
发送ACK_5761成功！
```

窗口大小为10时：

传输时间为53秒，吞吐率为111比特/毫秒



```
分组已接收，清空包内缓存
窗口边界base变为：5758

时间：[2023/12/12 20:43:37:921]
收到ACK：5758
分组已接收，清空包内缓存
窗口边界base变为：5759

时间：[2023/12/12 20:43:37:923]
收到ACK：5759
分组已接收，清空包内缓存
窗口边界base变为：5760

时间：[2023/12/12 20:43:37:925]
收到ACK：5760
分组已接收，清空包内缓存
窗口边界base变为：5761

时间：[2023/12/12 20:43:37:931]
收到ACK：5761
分组已接收，清空包内缓存
窗口边界base变为：5762

时间：[2023/12/12 20:43:37:948]
当前文件发送完毕。
总传输时间为：53秒
吞吐率为：110.722比特/毫秒
错误（丢包+超时）率为：0.105696
是否要发送文件？是：0/退出：1
```

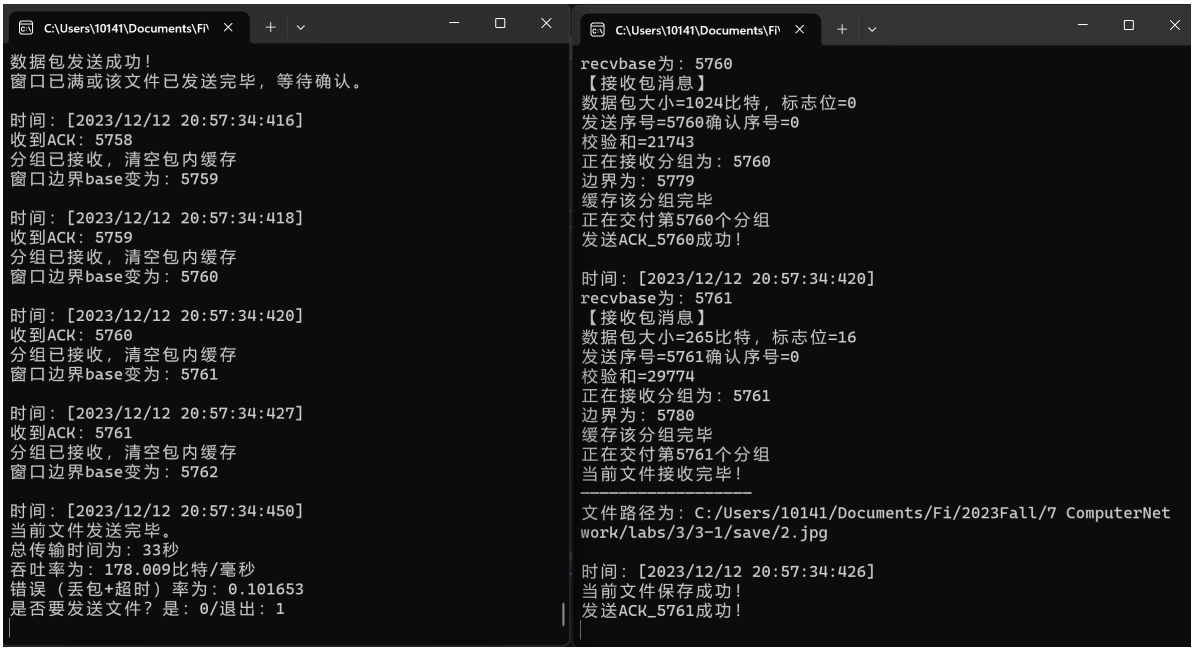
```
recvbase为：5760
【接收包消息】
数据包大小=1024比特，标志位=0
发送序号=5760确认序号=0
校验和=21743
正在接收分组为：5760
边界为：5769
缓存该分组完毕
正在交付第5760个分组
发送ACK_5760成功！

时间：[2023/12/12 20:43:37:925]
recvbase为：5761
【接收包消息】
数据包大小=265比特，标志位=16
发送序号=5761确认序号=0
校验和=29774
正在接收分组为：5761
边界为：5770
缓存该分组完毕
正在交付第5761个分组
当前文件接收完毕！
————————————————
文件路径为：C:/Users/10141/Documents/Fi/2023Fall/7 ComputerNetwork/labs/3/3-1/save/2.jpg

时间：[2023/12/12 20:43:37:930]
当前文件保存成功！
发送ACK_5761成功！
```

窗口大小为16时：

传输时间为39秒，吞吐率为150比特/毫秒

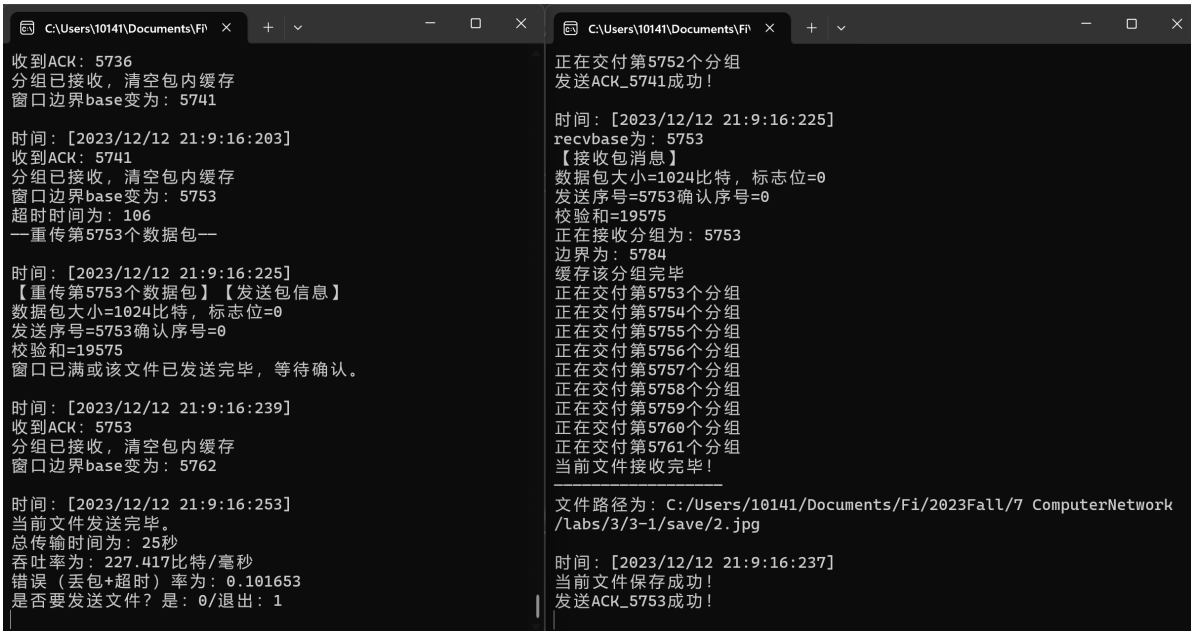
窗口大小为20时：

传输时间为33秒，吞吐率为178比特/毫秒

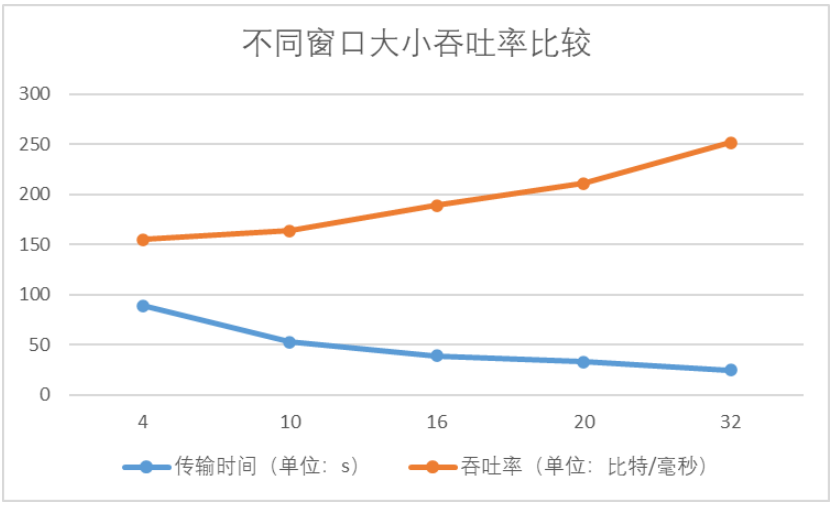


窗口大小为32时：

传输时间为25秒，吞吐率为227比特/毫秒



比较折线图如下所示：



发现在丢包率相同的情况下，滑动窗口越大，吞吐率越高，传输时间越短。这与上次实验3-2的结果相反，可以证明在未实现选择重传时，单个分组的差错就能够引起GBN重传大量分组，增加信道差错率。在实现选择重传机制以后，避免了不必要的重传，提高了数据运输效率。

实验总结

在本次实验中，基于之前实现的流水线可靠数据传输协议，完成了选择确认协议的设计与实现，提高了传输文件的速率。