计算机网络实验报告

实验3-2：基于滑动窗口的流量控制机制

姓名：谢雯菲 学号：2110803

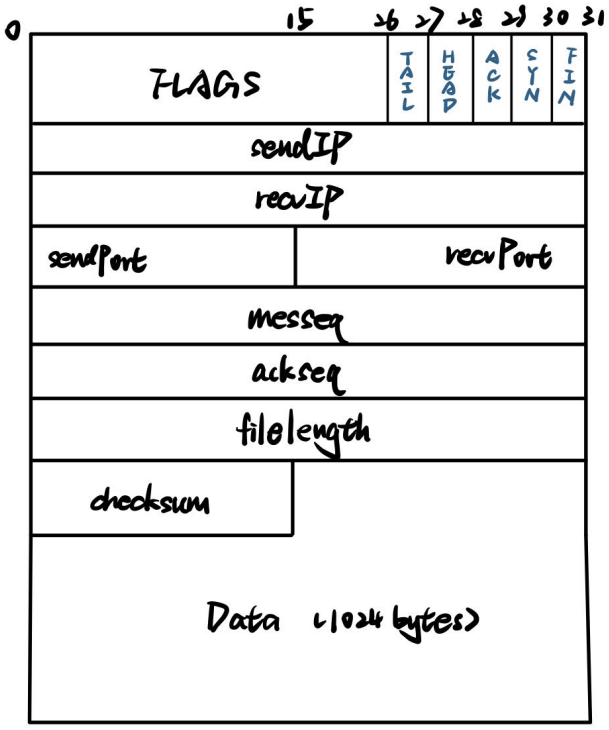
实验要求

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，支持累积确认，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：

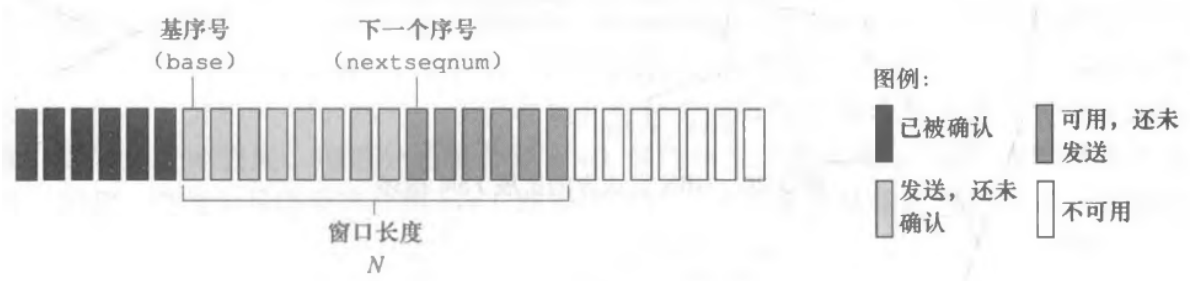


- 32位FLAGS位（后五个比特位分别为：TAIL（最后一个文件标志位）、HEAD（头文件标志位）、ACK（确认位）、SYN（同步位）、FIN（结束位））
- 32位发送端IP地址（没用到）
- 32位接收端IP地址（没用到）
- 16位发送端端口号，16位接收端端口号（没用到）
- 32位发送序列号（实际只用了1位）
- 32位确认序列号（实际只用了1位）
- 32位文件长度
- 16位校验和

数据包格式的代码和3-1相同，在此不过多赘述。

GBN滑动窗口

本次实验设计滑动窗口参照教材，滑动窗口示意图如下所示：



代码中变量名与图中名称对应如下：

- 基序号：sendbase
- 下一个序号：nextseqnum
- 窗口长度：windowsize

在GBN协议中，发送方可以发送不超过窗口大小数量的数据包而不需等待确认。但是，如果窗口开始的包超过一定时间还未收到相应的ACK，或者收到了之前已经收到过的ACK的话，就会重传窗口内所有未收到确认的包，这一步就是`Go Back N`回退N步。

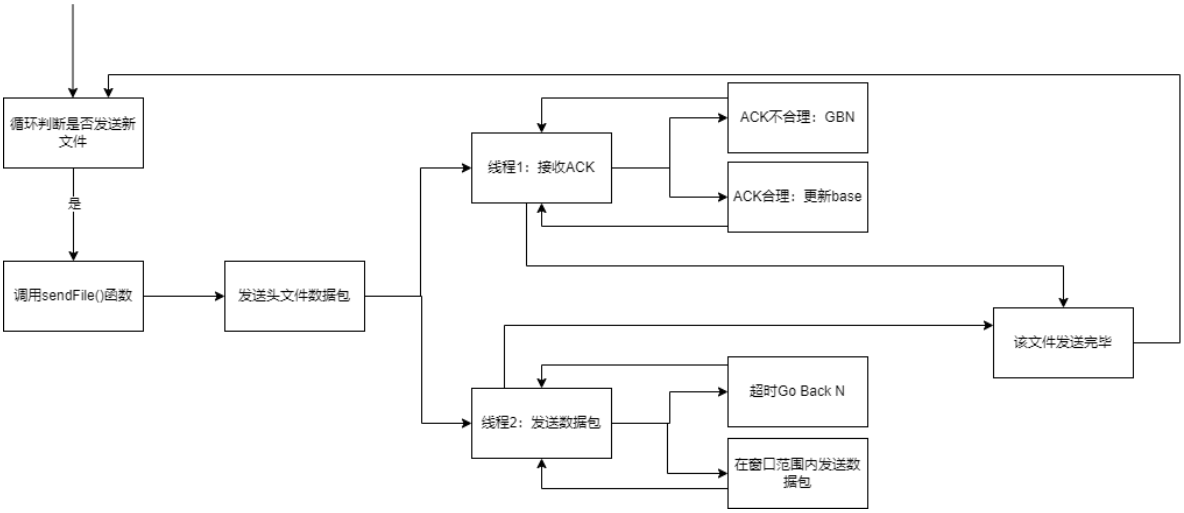
累积确认

本次实验发送端采用累积确认的方式确认接收端正确接收分组。

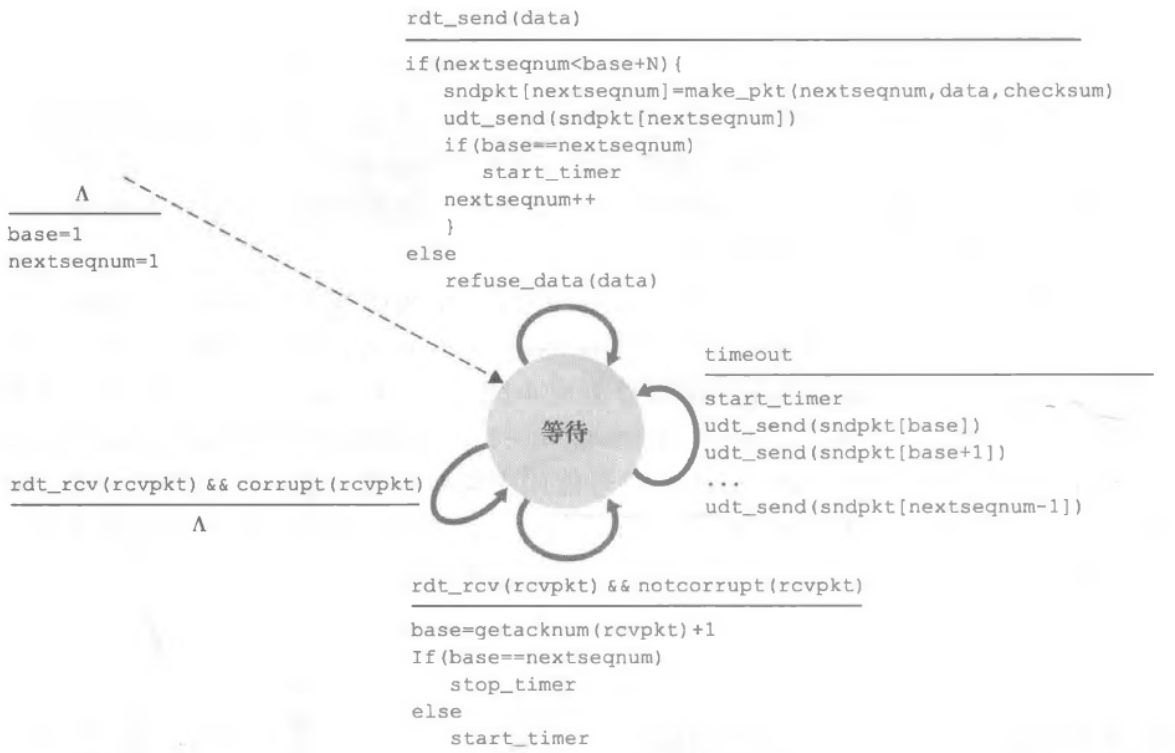
接收端接收到正确序号的包后就会发回相应的ACK，如果收到序号不正确的数据包都会返回上一个正确接收包的ACK，但是ACK包可能会发生丢包。发送端忽略中间丢掉的ACK包，每次收到新的ACK时都会将窗口滑动到相应的位置，一次滑动的位置取决于收到的ACK序号，这就是累积确认。

发送端交互

在发送端的主函数有一个`while`循环接收指令，决定是否开始发送新的文件。之后，调用了`sendFile()`函数发送数据，`sendFile()`函数流程示意图如下所示：



其中，主要的设计思路参照书上的GBN发送端FSM：



由于发送端需要能够响应以下三种类型的事件：

1. 上层调用并发送数据包
2. 处理收到的ACK
3. 超时时间

因此在设计发送文件的代码时，使程序发送完头文件包后进入两个线程：

- 线程1：负责处理接收端发来的ACK，如果为新的ACK，更新sendbase的值；如果是之前收到过的ACK，重传窗口内包。
- 线程2：负责为接收端发送数据包，并且重传窗口内的未确认的包。

线程中需要对线程共享的数据上锁，否则会产生冲突。实验中在未加锁时出现过窗口正在滑动而发送窗口内数据包同时正在发送的情况。

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
clock_t StartTimer; //定时器
bool EndTimer = false;
bool needResend = false; //用来判断是否收到错误的ACK，如果是，重传

//重传函数
void resendMes() {
    cout << "----重传窗口内包----" << endl;
    for (int i = sendbase; i < nextseqnum; i++) {
        int index = i % windowsize;
        sendto(clisock, (char*)sendMes[index], BUFFER_SIZE, 0,
(SOCKADDR*)&seraddr, seraddr_len);
        printTime();
```

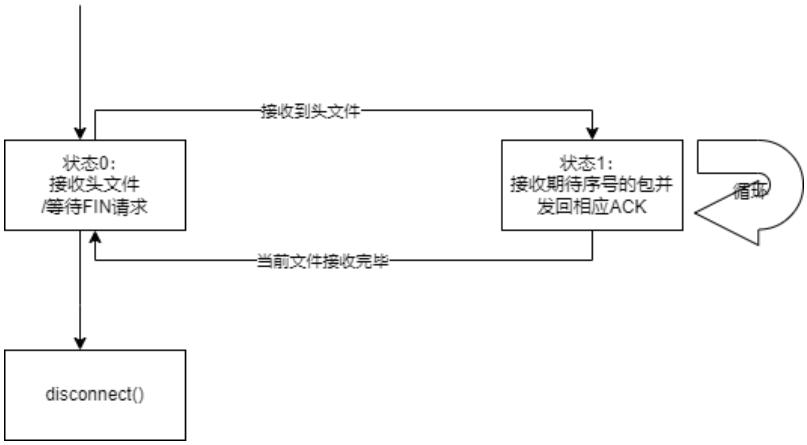
```
        cout << "【重传数据包】";
        printSendMessage(sendMes[index]);
        lostnum++;
        allsendnum++;
    }
}

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)) {
                cout << "收到ACK：" << recvMes->ackseq << endl;
                if ((recvMes->ackseq + 1) <= (sendbase)) {
                    cout << "窗口大小未改变，立即重传";
                    cout << recvMes->ackseq << "::" << sendbase << endl;
                    {
                        std::lock_guard<std::mutex> mylockguard(mylock);
                        needResend = true;
                    }
                }
                else {
                    {
                        std::lock_guard<std::mutex> mylockguard(mylock);
                        sendbase = recvMes->ackseq + 1;
                        cout << "窗口边界base变为：" << sendbase << endl;
                        if (sendbase == nextseqnum) {
                            //KillTimer(NULL, timer);
                            EndTimer = true;
                            cout << "----停止计时----" << endl;
                        }
                        else {
                            //timer = SetTimer(NULL, 0, timeout, (TIMERPROC)resendMes);
                            EndTimer = false;
                            StartTimer = clock();
                        }
                    }
                }
            }
            else {
                cout << "接收的包已损坏或不是ACK包" << endl;
            }
        }
    }
}

//发送文件
void sendFile() {
    sendMes = new message * [windowsize];
    sendbase = 0;
    nextseqnum = 0; //下一个要发送包的序号
    needResend = 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);
    StartTimer = clock();
    EndTimer = false;

    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
        clock_t StopTimer = clock();
        double intival = (StopTimer - StartTimer) * 1000 / CLOCKS_PER_SEC;
        cout << "超时时间为：" << intival << endl;
        if ( (intival > timeout && !EndTimer) || needResend) { //如果超时或者接收错误ACK，重传
            if (intival > timeout && !EndTimer) {
                cout << "设置超时时间为：" << timeout << endl;
                cout << "超时时间为：" << intival << endl;
            }
            {
                std::lock_guard<std::mutex> mylockguard(mylock);
                needResend = false;
                resendMes();
                StartTimer = clock();
                EndTimer = false;
            }
        }
        if (nextseqnum < sendbase + windowsize && nextseqnum <= packetNum) { //如果窗口没满且没有发送完毕
            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;
                sendMes[index]->clearMes();
                if (nextseqnum == packetNum) { //如果是最后一个文件，设置文件尾
```

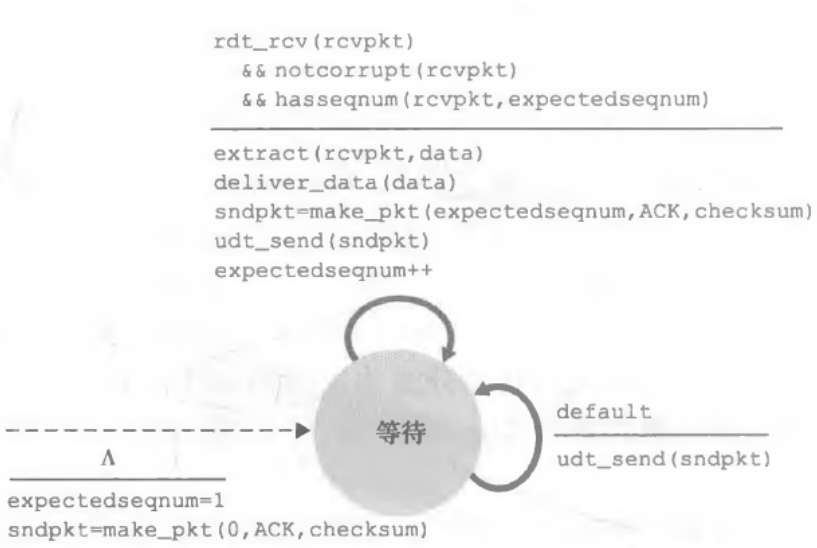
```
                    sendMes[index]->FLAGS = TAIL;
                }
                sendMes[index]->fillData(nextseqnum, (fileSize -
(nextseqnum - 1) * 1024) > 1024 ? 1024 : fileSize - (nextseqnum - 1) *
1024, fileBuffer + (nextseqnum - 1) * 1024);
                sendMes[index]->setchecksum((uint16_t*)sendMes[index]);
                printSendMessage(sendMes[index]);
                res = sendto(clisock, (char*)sendMes[index], BUFFER_SIZE,
0, (SOCKADDR*)&seraddr, seraddr_len);
                if (sendbase == nextseqnum) {
                    EndTimer = false;
                    StartTimer = clock();
                }
                if (res > 0) {
                    allsendnum++;
                    cout << "数据包发送成功！" << endl;
                    //printSendMessage(sendMes);
                    nextseqnum++;
                }
                else {
                    cout << "数据包发送失败！" << endl;
                }
            }
        }
        else {
            cout << "窗口已满或该文件已发送完毕，等待确认。" << endl;
            Sleep(20);
        }
    }
    EndTimer = true;
    //timerThread.join();
    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) / CLOCKS_PER_SEC << "秒" <<
endl;
    cout << "吞吐率为：" << (float)fileSize / ((end - start) * 1000 /
CLOCKS_PER_SEC) << "比特/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;
}
```

接收端交互

接收端调用`receiveFile()`函数流程如下图所示：



主要设计思路参照书上的GBN接收端的FSM：



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，如果发送序号正确，返回相应的ACK包；否则，返回已经确认的最大序号的ACK

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
    int expectedseqnum = 0; //期待接收的下一个包的序号


    message* 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) {
                        cout << "当前文件已接收完毕，等待发送下一文件" << endl;
                        sendACK(expectedseqnum - 1);
                    }
                    else if (!isCorrupt(recvMes) && recvMes->messeq == 0) {
                        if (recvMes->FLAGS == HEAD) {
                            expectedseqnum = 1;
                            cout << "接收数据包头成功！" << endl;
                            //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;
                            cout << "文件大小为：" << fileSize << "比特，总共需要接收" << packetnum << "个数据包" << endl;
                            cout << "等待发送文件数据包……" << endl;
                            //发送ACK_0数据包
                            sendACK(0);
                            state = 1;
                        }
                        else {
                            cout << "收到的数据包不是文件头，等待发送端重传……" << endl;
                        }
                    }
                }
            }
            break;
```

```
        case 1:
            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);
                    //累积确认
                    if (!isCorrupt(recvMes) && recvMes->messeq == expectedseqnum) {
                        //收到了目标的包
                        memcpy(fileBuffer + recvSize, recvMes->data, recvMes->filelen);
                        recvSize += recvMes->filelen;
                        sendACK(expectedseqnum);
                        expectedseqnum++;
                        if (recvMes->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;
                            //保存文件
                            saveFile();
                        }
                        else {
                            cout << endl << "文件包接收成功！" << endl;
                        }
                    }
                    else {
                        cout << endl << "包损坏或序号错误" << endl;
                        sendACK(expectedseqnum - 1);
                    }
                }
            }
            else {
                cout << "接受包错误！" << endl;
            }
            break;
        }
    }
    delete recvMes;
}
```

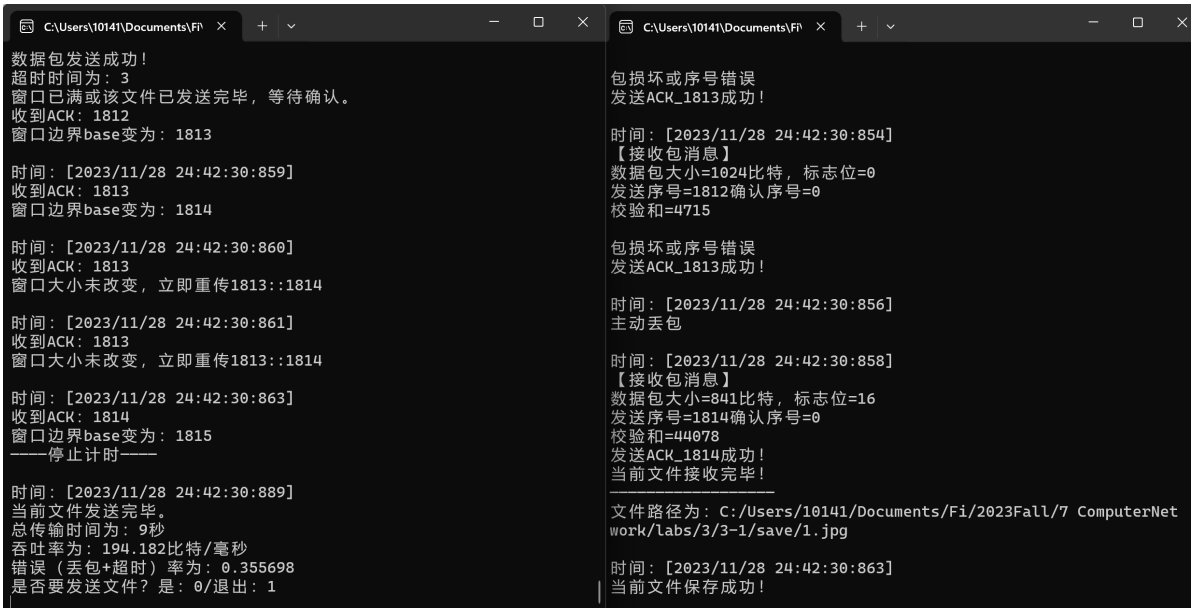
运行结果

丢包率0.1，窗口大小4

使用自己写的基于随机数的丢包，设置丢包率为0.1，在窗口大小为4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）：

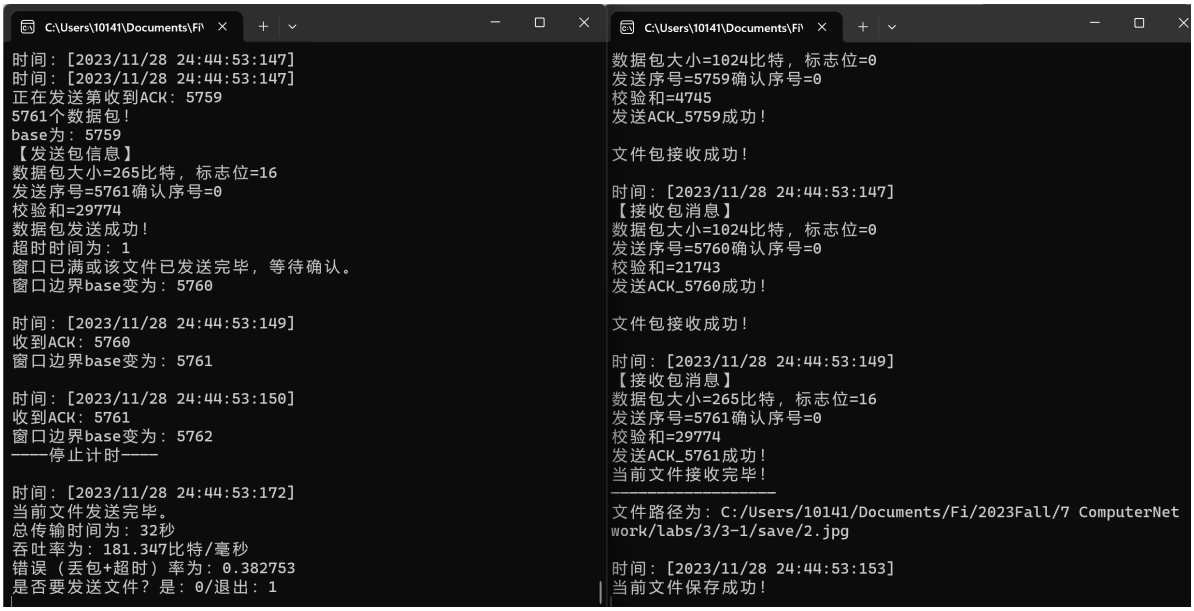
`1.jpg`：

传输时间为9秒，吞吐率为194比特/毫秒



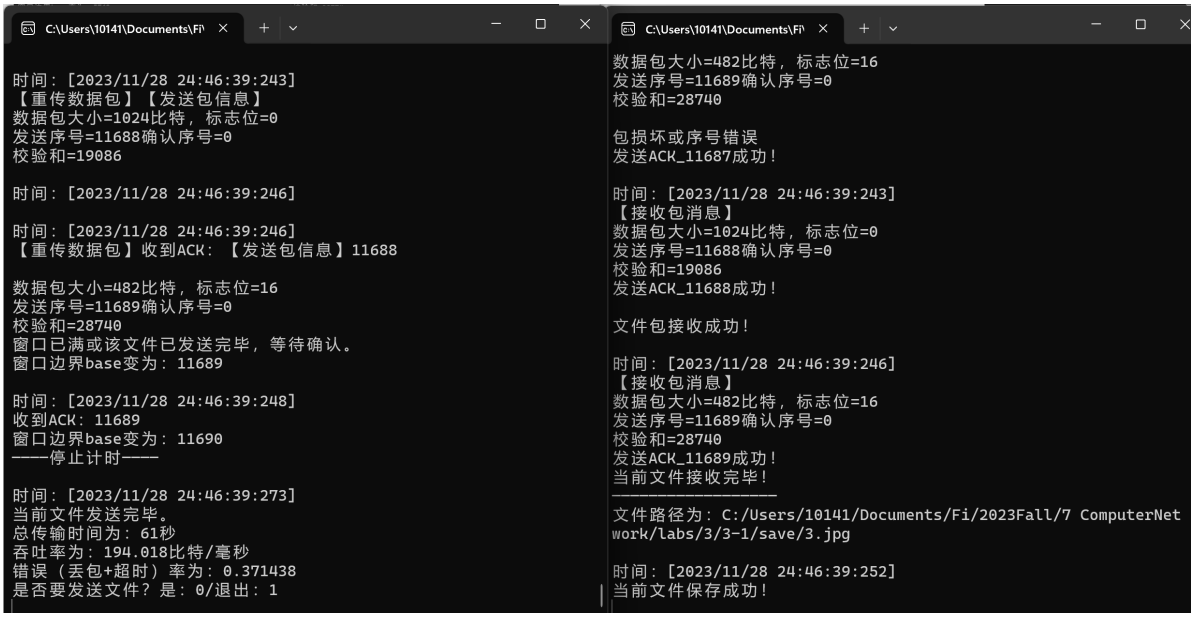
`2.jpg`：

传输时间为32秒，吞吐率为181比特/毫秒



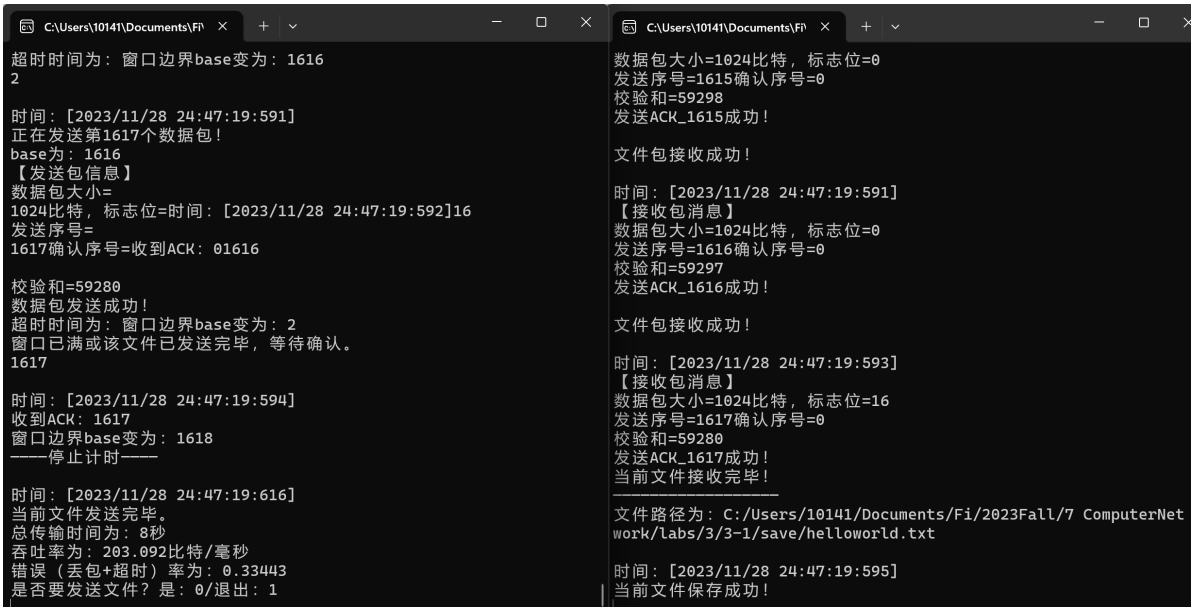
`3.jpg`：

传输时间为61秒，吞吐率为194比特/毫秒



`helloworld.txt`：

传输时间为8秒，吞吐率为203比特/毫秒

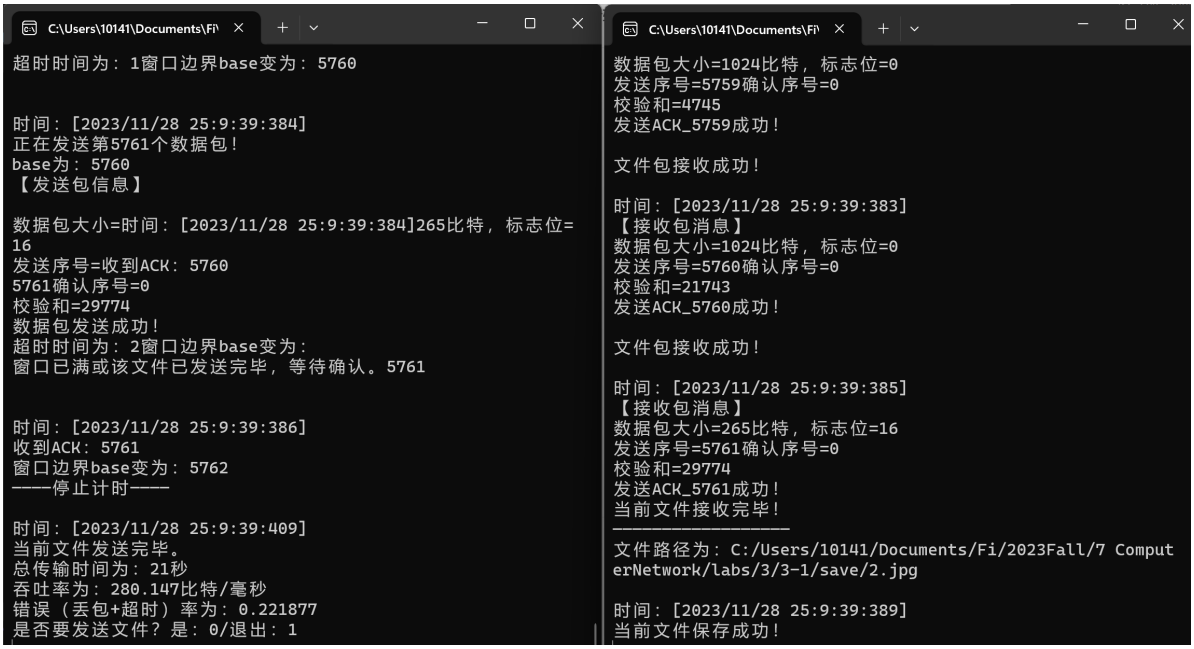


不同窗口大小吞吐率比较

在丢包率为0.05，滑动窗口大小不同的情况下进行测试传输`2.jpg`文件：

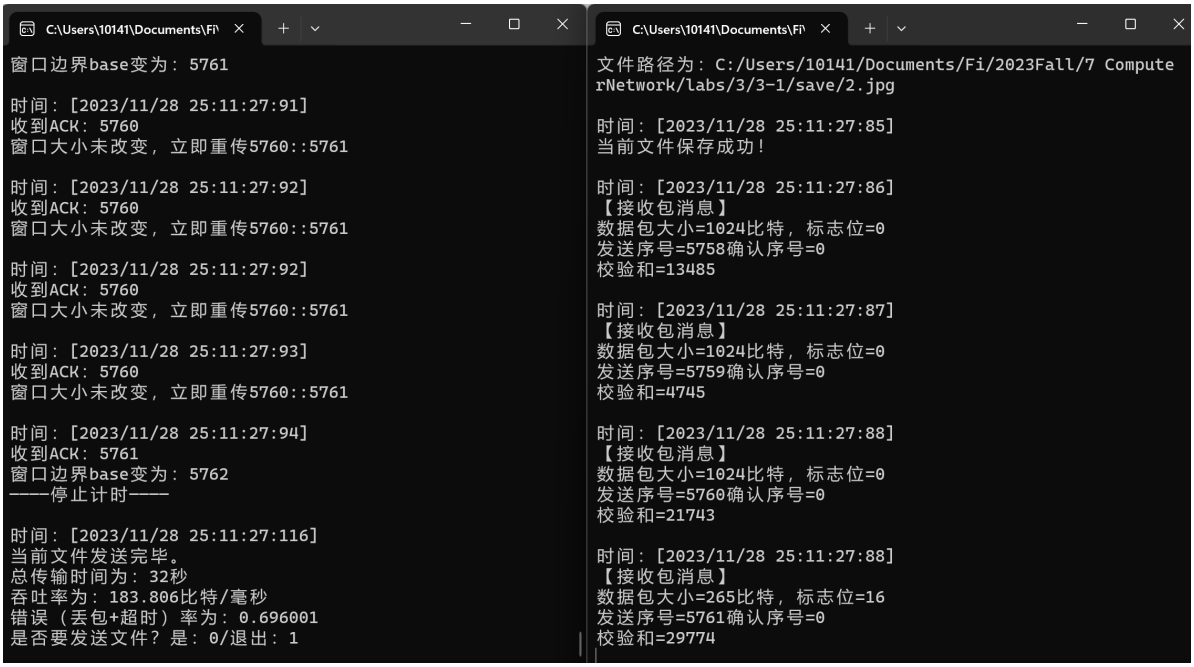
窗口大小为4时：

传输时间为21秒，吞吐率为280比特/毫秒



窗口大小为8时：

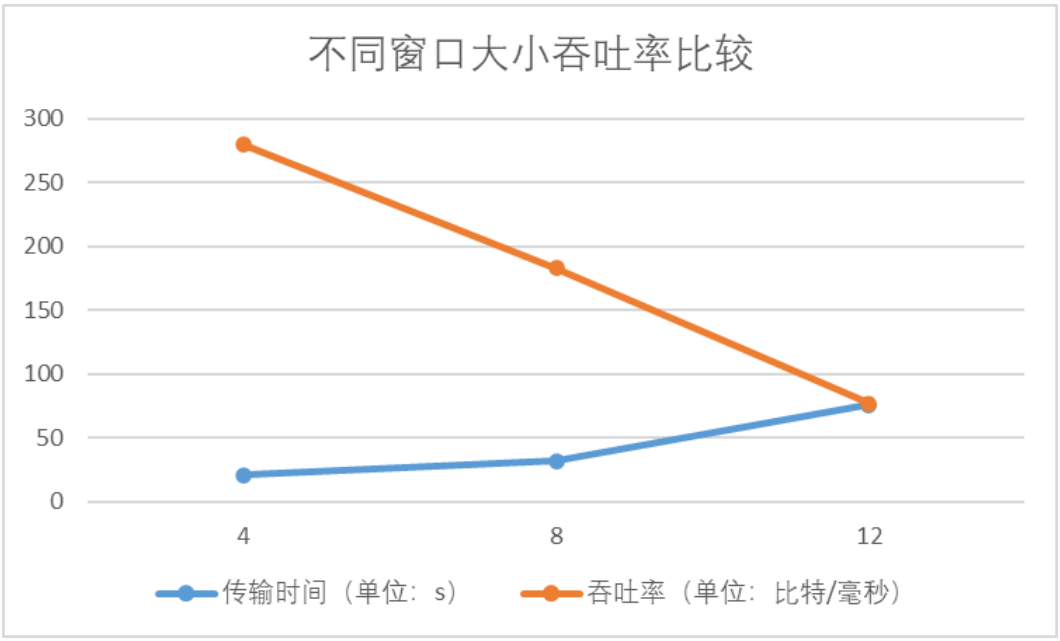
传输时间为32秒，吞吐率为183比特/毫秒



窗口大小为16时：

传输时间为76秒，吞吐率为77比特/毫秒

比较折线图如下所示：



发现窗口越大时，传输速率反而越低，猜测是因为窗口越大时，重传的时间消耗越大，在后续实验中完善协议后应当有所改善。

实验总结

在本次实验中，基于之前实现的传输协议，编写程序实现了基于滑动窗口的流量控制机制。虽然在吞吐率方面还未十分完善，但为后续实验实现更多功能奠定了基础。