



哈尔滨工业大学
Harbin Institute of Technology

计算机网络 课程实验报告

实验名称	GBN 协议的设计与实现					
姓名	张儒		院系	软件学院		
班级	2137101		学号	2021112678		
任课教师	李全龙		指导教师	李全龙		
实验地点	格物 207		实验时间	10.23		
实验课表现	出勤、表现得分(10)		实验报告 得分(40)		实验总分	
	操作结果得分(50)					
教师评语						



哈尔滨工业大学计算学部
FACULTY OF COMPUTING, HIT

实验目的：

1. 理解可靠数据传输的基本原理；掌握停等协议的工作原理；掌握基于 UDP 设计并实现一个停等协议的过程与技术。
2. 理解滑动窗口协议的基本原理。掌握 GBN 的工作原理并掌握基于 UDP 设计并实现一个 GBN 协议的过程与技术。

实验内容：

1. 基于UDP设计一个简单的停等协议，并实现支持双向数据传输，并模拟引入数据包的丢失，验证所设计协议的有效性。
2. 进一步进行改进和优化，实现一个简单的GBN协议，实现双向可靠数据传输（服务器到客户、客户到服务器的数据传输），并模拟引入数据包的丢失，验证所设计协议的有效性。
3. 最后将我们所设计的GBN协议改进为SR协议，并验证可行性。

实验过程：**1. 确定GBN协议数据分组的格式、确认分组的格式、各个域的作用**

- GBN协议数据分组格式

采用的如下图所示的分组格式：

Seq	Data	0
-----	------	---

Seq表示分组的序列号，通过序号，我们查看是否有分组丢失，失序或重复。

Data用来储存我们传输的数据，其中Data长度为1024个字节

0为EOF0表示数据的结尾

- 确认分组的格式

采用的如下图所示的分组格式：

ACK	0
-----	---

ACK数据帧不含有任何数据，只需发送ACK

ACK字段为一个字节，表示序列号的数值

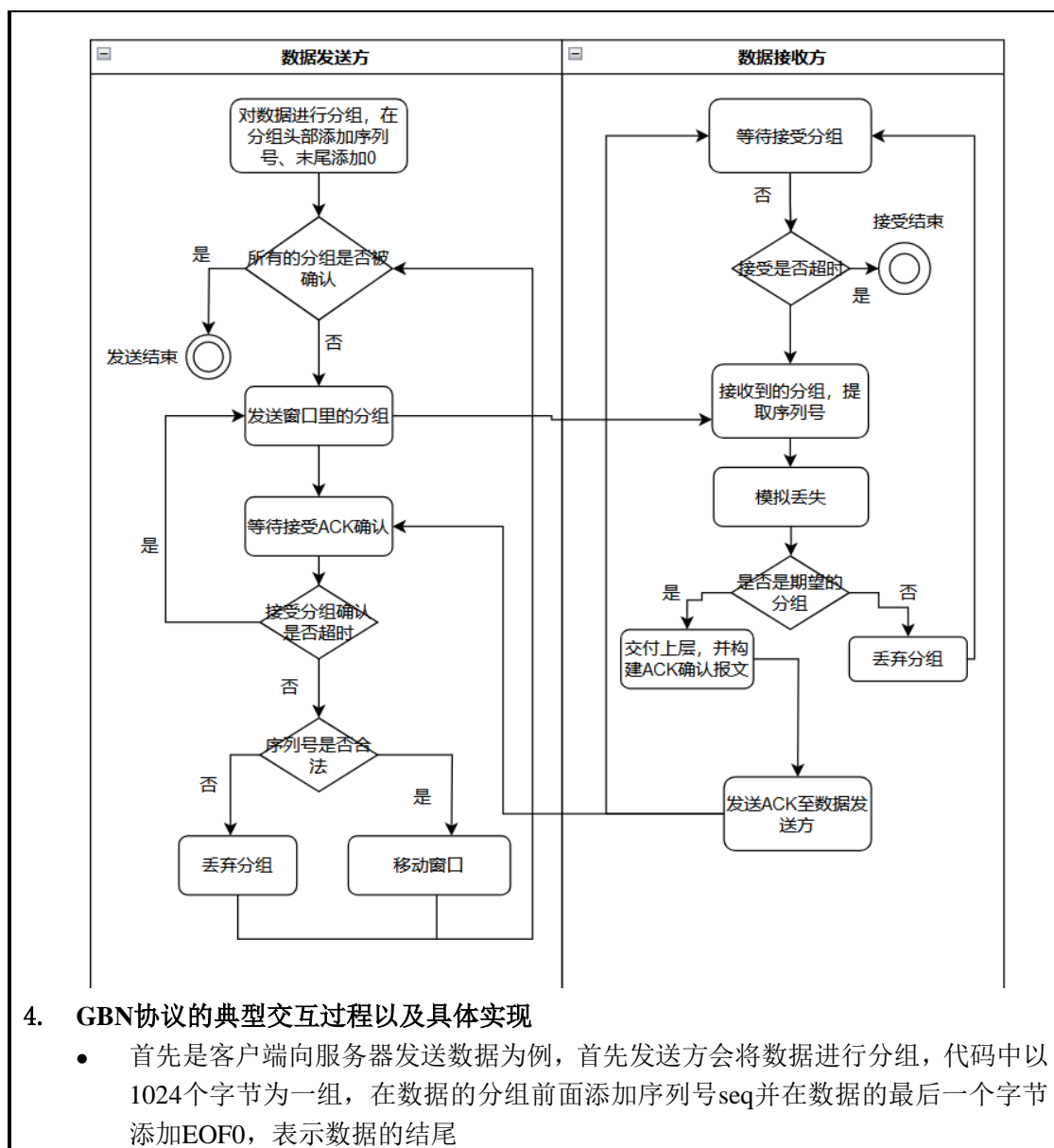
0放在末尾，表示数据结束

2. 数据分组丢失验证模拟方法

客户端对接收的数据帧进行计数，然后对总数进行模N运算，若规定求模运算结果为零，那么就丢失对应的分组。其余的分组正常接受。

```
// 模拟丢包
if (count % LOSS == 0) {
    System.out.println("丢弃此分组~: "+seq);
    continue;
}
```

3. GBN协议两端程序流程图



4. GBN协议的典型交互过程以及具体实现

- 首先是客户端向服务器发送数据为例，首先发送方会将数据进行分组，代码中以1024个字节为一组，在数据的分组前面添加序列号seq并在数据的最后一个字节添加EOF0，表示数据的结尾

```
private byte[][] splitData(ByteOutputStream dataStream, int size) {
    byte[] data = dataStream.toByteArray();
    //分组的个数
    int numPackets = (int) Math.ceil((double) data.length / size);
    byte[][] result = new byte[numPackets][size];
    long currentSeq = 0;
    //将数据按照seq + data的形式进行拼接
    int dataStartIndex = 0;
    //在每个分组前面都加上分组编号
    for (int i = 0; i < numPackets; i++) {
        ByteOutputStream temp = new ByteOutputStream();
        //将分组号写入到数据
        ByteBuffer longBuffer = ByteBuffer.allocate(Long.BYTES);
        longBuffer.putLong(currentSeq);
        byte[] longTemp = longBuffer.array();
        temp.write(longTemp, 0, Long.BYTES);
        //确定结束下标, 防止超过数据总长度
        int len = size - Long.BYTES;
        if (dataStartIndex + len > data.length) {
            len = data.length - dataStartIndex;
        }
        //将数据写入分组
        temp.write(data, dataStartIndex, len);
        //下一个分组在data的开始下标
        dataStartIndex = dataStartIndex + len;
        result[i] = temp.toByteArray();
        currentSeq++;
    }
    return result;
}
```

- 对数据处理完之后, 开始进行数据的发送。如果所有的分组没有被全部确认, 就一直重复以下步骤, 直到分组被完全确认。
- 发送窗口里面的全部分组:

```
//发送窗口里面所有的分组
for (int i = windowIndex; i < windowIndex + WINDOW_SIZE && i < packetNum; i++) {
    DatagramPacket sendPacket = new DatagramPacket(dataGroup[i], offset: 0, dataGroup[i]
    //发送分组
    try {
        clientSocket.send(sendPacket);
    } catch (IOException e) {
        System.out.println("发送分组异常.....");
        throw new RuntimeException(e);
    }
}
```

- 发送完窗口里的分组之后, 开始等待接受ACK确认报文
- 如果接收方返回来的确认分组的序号是合法的, 那么就移动发送方的窗口

```
//从receivePacket中拿到确认的编号
long receiveAckSeqNum = getSeqNum(receivePacket);
System.out.println("接收者已确认分组编号:" + receiveAckSeqNum);
//如果接收方返回来的确认分组序号是合法的, 更新发送方最新的确认分组号
if (receiveAckSeqNum >= sendAckSeqNum && receiveAckSeqNum <= sendAckSeqNum + WINDOW_SIZE) {
    sendAckSeqNum = receiveAckSeqNum;
    //窗口移动
    windowIndex = (int) sendAckSeqNum + 1;
} else {
    break;
}
```

- 如果某个发送的分组超时了(规定时间内没有收到ACK确认), 就重新发送窗口里面所有的分组。这里对于每个分组的超时检测是采用的UDP协议的

setSoTimeout方法

```
catch (SocketTimeoutException e) {
    //当sendAckSeqNum < packetNum - 1 而且出现SocketTimeoutException的时候
    //说明确认编号错误, 重新发送窗口里所有的分组
    if(sendAckSeqNum < packetNum - 1){
        //说明传送socket超时了, 那么重传窗口里的分组
        //发送窗口里面所有的分组
        for (int i = windowIndex; i < windowIndex + WINDOW_SIZE && i < packetNum; i++) {
            DatagramPacket sendPacket = new DatagramPacket(dataGroup[i], offset: 0, dataGroup[i]
            //发送分组
            try {
                clientSocket.send(sendPacket);
            } catch (IOException ex) {
                throw new RuntimeException(ex);
            }
        }
    }
}
```

- 发送方的操作介绍完毕, 下面来介绍接收方的操作
- 接收方等待接受数据, 收到了分组之后, 会随机进行模拟丢失, 若不丢失, 会从分组中提取分组号, 检测分组序号是否是期望的分组

```
//接受一个分组
byte[] receive = new byte[GROUP_SIZE];
receivePacket = new DatagramPacket(receive, receive.length, targetHost, targetPort);
datagramSocket.receive(receivePacket);
//从接受的数据中提取分组号
long seq = getSeqNum(receivePacket);
// 若不是期望接收的分组, 则丢弃
if(expectSeq != seq) continue;

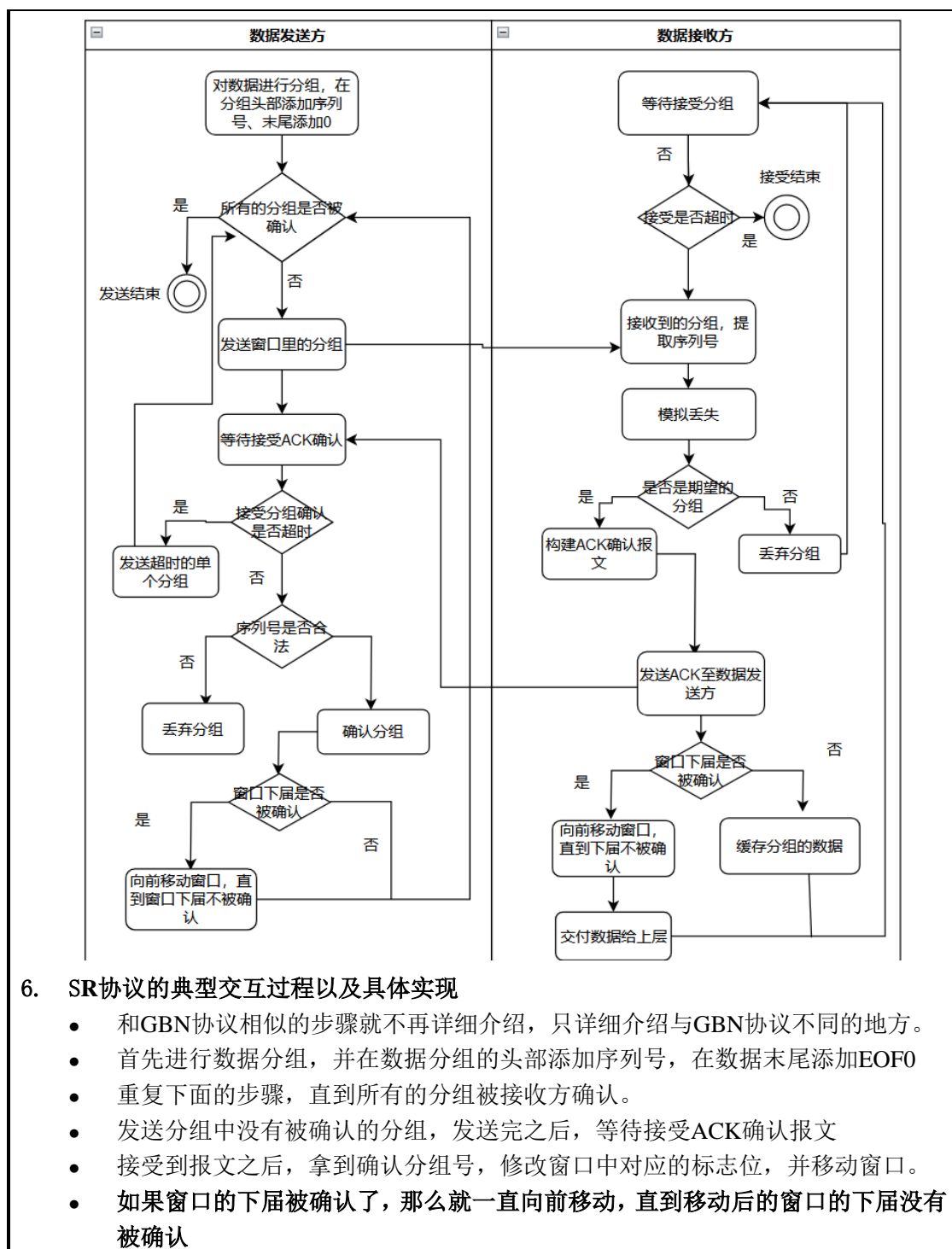
// 模拟丢包
if(count % LOSS == 0) {
    System.out.println("丢弃此分组~:" + seq);
    continue;
}
```

- 如果没有被模拟丢弃, 而且分组号为期望的。那么就会写入到结果中去, 并构建一个ACK确认报文, 其中含有确认的分组号, 返回给发送方。

```
//将收到的分组写入结果中去
result.write(receive, off: 8, len: receivePacket.getLength() - 8);
expectSeq++;
//创建一个ack报文, 含有确认的分组号
ByteArrayOutputStream temp = new ByteArrayOutputStream();
ByteBuffer longBuffer = ByteBuffer.allocate(Long.BYTES);
longBuffer.putLong(seq);
byte[] longTemp = longBuffer.array();
temp.write(longTemp, off: 0, Long.BYTES);
byte[] seqPacket = temp.toByteArray();
//发送ack确认分组
receivePacket = new DatagramPacket(seqPacket, seqPacket.length, targetHost, targetPort);
datagramSocket.send(receivePacket);
System.out.println("接收到分组: seq " + seq);
```

- 最后如果长时间没有收到分组, 表明接收完毕, 关闭链接, 将接受到的数据返回给上层。

5. SR协议两端程序流程图



```
//从receivePacket中拿到确认的编号
long receiveAckSeqNum = getSeqNum(receivePacket);
System.out.println("接收者已确认分组编号:" + receiveAckSeqNum);
//如果接收方返回来的确认分组序号是合法的, 确认对应分组
if (receiveAckSeqNum >= windowIndex && receiveAckSeqNum < allPacket.size() &&
    //确认对应分组
    allPacket.set((int) receiveAckSeqNum, true);
    //如果下届被确认了, 那么向前移动窗口
    while (windowIndex < allPacket.size() && allPacket.get(windowIndex)) {
        windowIndex++;
        sendAckSeqNum++;
    }
} else {
    break;
}
```

- 如果接受分组的ACK确认报文超时了, 那么就重发超时的那一个分组
- 发送方操作介绍完毕, 下面介绍接收方操作
- 接收方会缓存接收方发来的分组 (因为有可能是乱序的)

```
//缓存发送方发过来的乱序的分组数据
HashMap<Integer, ByteArrayOutputStream> receiveCache = new HashMap<>();
//接收方的窗口
LinkedHashMap<Integer, Boolean> receiveWindow = new LinkedHashMap<>();
//初始化接收方窗口
//W+1<=2L 我们序号数直接选择2倍窗口大小
for (int i = 0; i < SEQ_NUM; i++) {
    receiveWindow.put(i, false);
}
//期望接受的分组, 也可以看做是下届
long receiveBase = 0;
```

- 如果接收到了分组, 进行模拟丢失和检验合法性。如果没被丢失且合法, 就提取分组号, 构建ACK确认报文, 并在窗口中确认该分组。
- 检验窗口的下届是否被确认, 如果下届被确认, 就向前移动窗口, 移动窗口的同时, 将缓存中对应的数据交付上层。直到窗口的下届不被确认。如果下届没被确认, 就将收到的分组先缓存起来。

```
//如果序号等于下届, 那么就传输数据
if (seq == receiveBase) {
    int begin = (int) seq;
    //如果下届始终被确认, 那么就一直发送
    while (receiveWindow.containsKey(begin % SEQ_NUM) && receiveWindow.get(begin % SEQ_NUM)) {
        result.write(receiveCache.get(begin).toByteArray());
        //发送完就删了
        receiveCache.remove(begin);
        //窗口向前滚动
        receiveWindow.replace(begin % SEQ_NUM, false);
        begin++;
        receiveBase++;
    }
} else {
    //将分组先缓存起来
    receiveCache.put((int) seq, receive);
}
```

- 如果长时间没有收到分组就表明接收完毕，关闭连接。

实验结果：

客户端要传输给服务器的文件为：

此电脑 > 桌面 > Java > NetWorkLab02 > clientSendFile



服务器要传输给客户端的文件为：

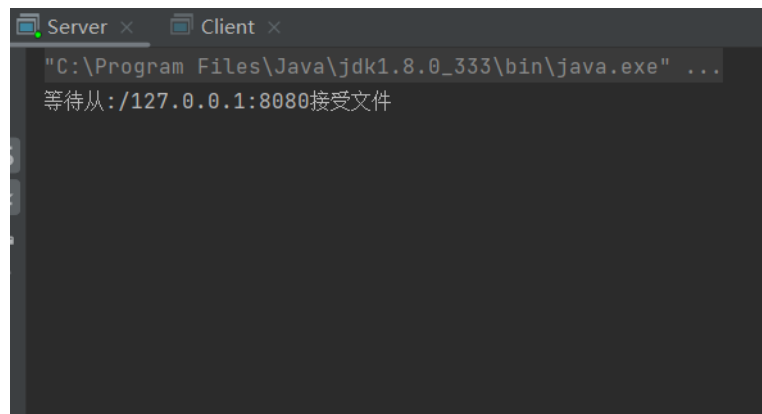
此电脑 > 桌面 > Java > NetWorkLab02 > serverSendFile



下面所演示的各种协议的双向数据传输均是以上述两个文件为例：

1. GBN协议的设计与实现的结果

- 首先运行服务端，服务器开启后等待接受客户端发来的数据



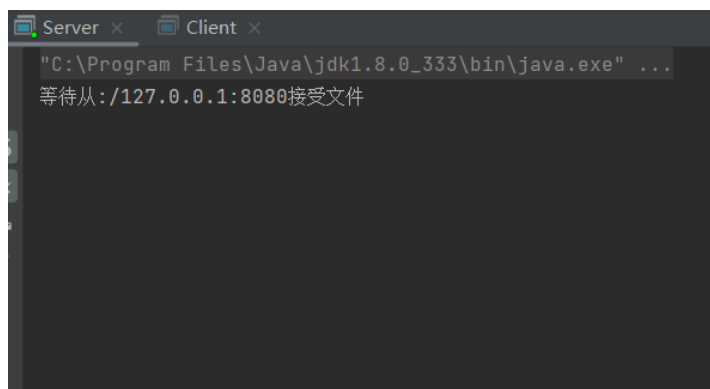
- 运行客户端，客户端就会开始向服务器发送数据分组，服务端接收到分组，并会丢弃一些分组


```
等待从: /127.0.0.1:8080接受文件
接收到分组: seq 0
接收到分组: seq 1
丢弃此分组~:2
接收到分组: seq 2
接收到分组: seq 3
接收到分组: seq 4
接收到分组: seq 5
接收到分组: seq 6
接收到分组: seq 7
接收到分组: seq 8
接收到分组: seq 9
丢弃此分组~:10
接收到分组: seq 10
接收到分组: seq 11
接收到分组: seq 12
接收到分组: seq 13
接收到文件!保存路径为:serverReceiveFile\file1.jpg
```

- 服务器收到分组后会返回ACK确认报文，并对超时的分组进行重发，如此反复，知道所有的分组都被服务器接受

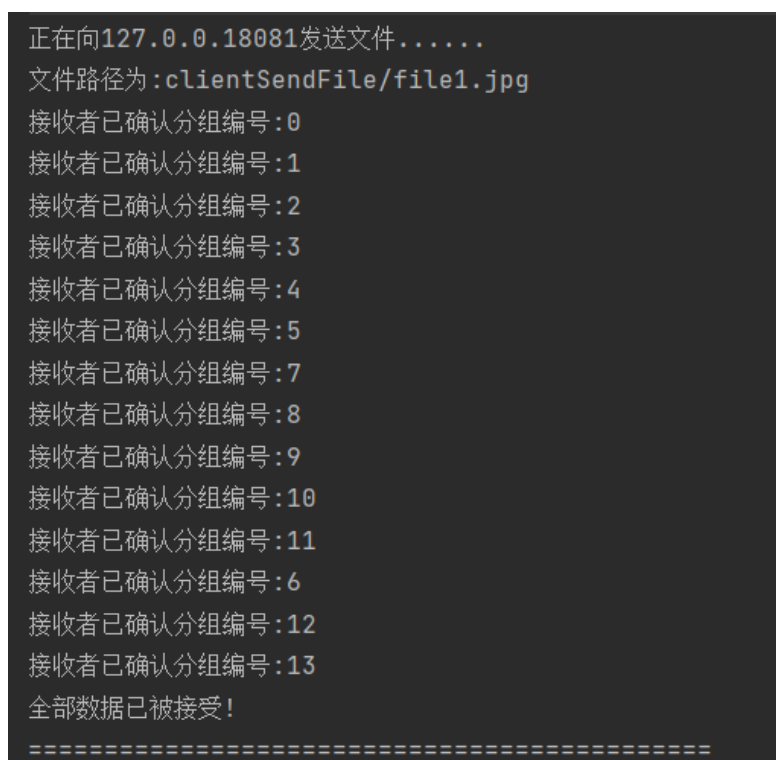
```
正在向127.0.0.18081发送文件.....
文件路径为:clientSendFile/file1.jpg
接收者已确认分组编号:0
接收者已确认分组编号:1
接收者已确认分组编号:2
接收者已确认分组编号:3
接收者已确认分组编号:4
接收者已确认分组编号:5
接收者已确认分组编号:6
接收者已确认分组编号:7
接收者已确认分组编号:8
接收者已确认分组编号:9
接收者已确认分组编号:10
接收者已确认分组编号:11
接收者已确认分组编号:12
接收者已确认分组编号:13
全部数据已被接受!
```

- 可以在相应文件夹看到服务器和客户端接收到文件
2. SR协议的实现与结果
- 首先运行服务端，开启后会等待接受数据



```
Server x Client x
"C:\Program Files\Java\jdk1.8.0_333\bin\java.exe" ...
等待从:/127.0.0.1:8080接受文件
```

- 运行客户端，向服务器发送数据。同时客户端也会受到服务器返回来的ACK确认报文



```
Client x
"C:\Program Files\Java\jdk1.8.0_333\bin\java.exe" ...
正在向127.0.0.18081发送文件.....
文件路径为:c:\clientSendFile/file1.jpg
接收者已确认分组编号:0
接收者已确认分组编号:1
接收者已确认分组编号:2
接收者已确认分组编号:3
接收者已确认分组编号:4
接收者已确认分组编号:5
接收者已确认分组编号:7
接收者已确认分组编号:8
接收者已确认分组编号:9
接收者已确认分组编号:10
接收者已确认分组编号:11
接收者已确认分组编号:6
接收者已确认分组编号:12
接收者已确认分组编号:13
全部数据已被接受!
=====
```

- 我们可以看到服务器收到了分组，并模拟丢弃了一些分组。丢弃之后，还会再次收到这个分组

```
等待从:/127.0.0.1:8080接受文件
接收到分组: seq 0
接收到分组: seq 1
接收到分组: seq 2
接收到分组: seq 3
接收到分组: seq 4
接收到分组: seq 5
丢弃此分组~: 6
接收到分组: seq 7
接收到分组: seq 8
接收到分组: seq 9
接收到分组: seq 10
接收到分组: seq 11
接收到分组: seq 6
接收到分组: seq 12
接收到分组: seq 13
接收到文件!保存路径为:serverReceiveFile\file1.jpg
```

- 客户端发送完文件之后，会开始等待从服务器接收数据。服务器收到数据之后，会想客户端发送数据。具体流程与客户端向服务器发送数据相同。
- 双向数据传输完毕后，可以在各自的receive文件夹看到接受到的文件

窗 > 桌面 > Java > NetWorkLab02 > serverReceiveFile



问题讨论:

1. 关于分组的序列号设计问题，在一开始，我的分组的序列号是把数据分成了多少个分组，那么就有多少序列号，在封装到分组的时候，会导致序列号占用的位数较多，产生了浪费。
之后，我采用了固定位数的序列号，再原来实现的基础上对原有的分组序列号进行了取余，由此实现了分组序列号的范围缩小。并同步在接收端对序号进行取余，再不大改原有代码的情况下实现了优化。
2. 接收方缓存数据的问题，在一开始，我是采用一个byte的二维数组来缓存接收方接收到的数据，但是最后查看接收到的图片的时候，图片无法正常加载。
经过打印相关的接收到的序列号可得知，当我们模拟了数据分组的丢失之后，接收方收到的数据不一定是有序的，有可能是乱序的，如果按照顺序来放入byte二维数据，那么交付上层的时候肯定也是乱序的。因此，我便改用了有一个HashMap来存储，其中，key为分组的序列号，Value为数据，通过这样，可以实现有序的向上层提交数据。

心得体会:

通过自己实现GBN（Go-Back-N）协议和SR（Selective Repeat）协议，我深入理解了滑动窗口协议的基本原理和工作机制。滑动窗口协议是一种在数据通信中用于流量控制和可靠数据传输的关键概念，通过实际编码实践，巩固了GBN协议的工作原理、SR协议的原理、网络编程等知识，使我能够更好地理解和处理网络通信中的问题，确保数据的可靠传输。这是在计算机网络和通信领域中重要的知识和技能。

注释代码：

Server的代码：

```
/**
 * 服务器
 */
public class Server {
    //服务器的端口号
    private static final int PORT = 8081;
    //目的主机（客户端）的IP地址
    private static final String TARGET_IP = "127.0.0.1";
    //目的主机（客户端）的端口号
    private static final int TARGET_PORT = 8080;
    //接受文件的路径
    private static final String RECEIVE_FILE_PATH = "serverReceiveFile/file1.jpg";
    //发送的文件的路径
    private static final String SEND_FILE_PATH = "serverSendFile/file1.jpg";
    //目的主机（客户端）的地址
    private static final InetAddress TARGET_HOST;
    static {
        try {
            TARGET_HOST = InetAddress.getByName(TARGET_IP);
        } catch (UnknownHostException e) {
            throw new RuntimeException(e);
        }
    }
    public static void main(String[] args) throws InterruptedException {
        Thread.sleep(1000);
        // TransferData GBN = new GBNTransferData(Server.PORT, TARGET_HOST, TARGET_PORT);
        // transfer(GBN);

        TransferData SR = new SRTransferData(Server.PORT, TARGET_HOST, TARGET_PORT);
        transfer(SR);
    }
    /**
     * 发送和接收文件
     * @param protocol 协议的类型
     * @throws IOException
     */
    private static void transfer(TransferData protocol) {
```

```

//1. 接受文件
System.out.println("等待从:" + TARGET_HOST + ":" + TARGET_PORT + "接受文件");
//2. 将接受的文件写入硬盘
ByteArrayOutputStream byteArrayOutputStream;
try {
    if((byteArrayOutputStream = protocol.receiveData()).size() != 0) {
        File file = new File(RECEIVE_FILE_PATH);
        writeDataToFile(byteArrayOutputStream, file);
        System.out.println("接收到文件! 保存路径为:" + file.getPath());
    }
} catch (IOException e) {
    throw new RuntimeException("文件IO出现错误!");
}
System.out.println("=====");
//3. 将要发送的文件从硬盘写入到内存
ByteArrayOutputStream fileData;
try {
    //将文件写入到字节输出流
    fileData = Client.transferFileToStream(SEND_FILE_PATH);
} catch (IOException e) {
    throw new RuntimeException("文件IO出现错误!");
}
System.out.println("正在向" + TARGET_IP + TARGET_PORT + "发送文件.....");
System.out.println("文件路径为:" + SEND_FILE_PATH);

//4. 发送数据
protocol.sendData(fileData);
}

/**
 * 将字节输出流中的数据写入到文件中
 * @param data 数据
 * @param file 文件
 */
public static void writeDataToFile(ByteArrayOutputStream data, File file) {
    FileOutputStream fileOutputStream;
    try {
        fileOutputStream = new FileOutputStream(file);
        fileOutputStream.write(data.toByteArray(), 0, data.size());
        fileOutputStream.close();
    } catch (IOException e) {
        System.out.println("写入文件出现错误!");
        throw new RuntimeException(e);
    }
}
}

```

```

}
Client代码:
/**
 * 客户端
 */
public class Client {
    //客户端的端口号
    private static final int PORT = 8080;
    //目的主机（服务器）的IP地址
    private static final String TARGET_IP = "127.0.0.1";
    //目的主机（服务器）的端口号
    private static final int TARGET_PORT = 8081;
    //发送的文件的路径
    private static final String SEND_FILE_PATH = "clientSendFile/file1.jpg";
    //接受的文件的路径
    private static final String RECEIVE_FILE_PATH = "clientReceiveFile/file1.jpg";
    //目的主机（服务器）的地址
    public static final InetAddress TARGET_HOST;
    static {
        try {
            TARGET_HOST = InetAddress.getByName(TARGET_IP);
        } catch (UnknownHostException e) {
            throw new RuntimeException(e);
        }
    }
    public static void main(String[] args) {
        // TransferData GBN = new GBNTransferData(Client.PORT, TARGET_HOST, TARGET_PORT);
        // transfer(GBN);

        TransferData SR = new SRTransferData(Client.PORT, TARGET_HOST, TARGET_PORT);
        transfer(SR);
    }
    public static void transfer(TransferData protocol){
        //1.先将文件写入到字节输出流中去
        ByteArrayOutputStream fileData;
        try {
            //将文件写入到字节输出流
            fileData = transferFileToStream(SEND_FILE_PATH);
        } catch (IOException e) {
            throw new RuntimeException("文件IO出现错误!");
        }
        //打印日志信息
        System.out.println("正在向"+ TARGET_IP + TARGET_PORT+"发送文件.....");
        System.out.println("文件路径为:"+SEND_FILE_PATH);
    }
}

```

```

//2.发送数据
//protocol = new GBNTTransferData(Client.Port, TargetHost, TargetPort);
protocol.sendData(fileData);
System.out.println("=====");
//3.接收数据
System.out.println("等待从:"+ TARGET_HOST +":"+ TARGET_PORT +"接受文件");
//4.将数据写入到文件中去
ByteArrayOutputStream byteArrayOutputStream;
try {
    if((byteArrayOutputStream = protocol.receiveData()).size() != 0) {
        File file = new File(RECEIVE_FILE_PATH);
        Server.writeDataToFile(byteArrayOutputStream,file);
        System.out.println("接收到文件!保存路径为:"+file.getPath());
    }
} catch (IOException e) {
    throw new RuntimeException(e);
}
}
/**
 * 将文件数据写入到ByteOutputStream 中去
 * @param url 文件路径
 * @return 返回含有数据的ByteOutputStream
 * @throws IOException IO异常
 */
public static ByteArrayOutputStream transferFileToStream(String url) throws IOException {
    File file = new File(url);
    if(!file.exists()){
        throw new RuntimeException("文件不存在!");
    }
    //读取文件基本操作
    ByteArrayOutputStream data = new ByteArrayOutputStream();
    FileInputStream fileInputStream = new FileInputStream(file);
    byte[] buffer = new byte[1024];
    int length;
    while ((length = fileInputStream.read(buffer)) != -1) {
        data.write(buffer, 0, length);
    }
    fileInputStream.close();
    return data;
}
}

```

GBNTTransferData代码:

```

/**
 * 使用GBN进行数据传输
 */
public class GBNTransferData implements TransferData {
    //目的主机的端口号
    private final int port;
    //目的主机的地址
    private final InetAddress targetHost;
    //目的主机的端口号
    private final int targetPort;
    //发送窗口的大小
    private static final int WINDOW_SIZE = 6;
    //分组的最大数据长度（字节）
    private static final int GROUP_SIZE = 1024;
    //超时时间
    private static final int TIMEOUT = 800;
    //如果超时了,重新尝试的次数
    private static final int OUT_TIME_TRY_TIMES = 3;
    //进行模loss运算，来模拟数据丢失
    private static final int LOSS = 8;
    //窗口的起始位置
    private int windowIndex = 0;
    //已经确认的最新数据序号,比如：1,2,3都确认，那就是3
    private long sendAckSeqNum = 0;

    public GBNTransferData(int port, InetAddress targetHost, int targetPort) {
        this.port = port;
        this.targetHost = targetHost;
        this.targetPort = targetPort;
    }
    /**
     * 发送数据
     * @param data 数据
     */
    @Override
    public void sendData(ByteArrayOutputStream data){
        //1.首先将数据按照seq + data进行拆分
        byte[][] dataGroup = splitData(data, GROUP_SIZE);
        //分组的总数
        int packetNum = dataGroup.length;
        DatagramSocket clientSocket;
        try {
            clientSocket = new DatagramSocket(port);
        } catch (SocketException e) {

```



```

        throw new RuntimeException(e);
    }
    //当最后一个分组packetNum - 1没有确认时,就循环操作
    while(sendAckSeqNum < packetNum - 1) {
        //发送窗口里面所有的分组
        for (int i = windowIndex; i < windowIndex + WINDOW_SIZE && i < packetNum; i++) {
            DatagramPacket sendPacket = new
            DatagramPacket(dataGroup[i],0,dataGroup[i].length,targetHost,targetPort);
            //发送分组
            try {
                clientSocket.send(sendPacket);
            } catch (IOException e) {
                System.out.println("发送分组异常.....");
                throw new RuntimeException(e);
            }
        }

        try {
            //设置超时时间
            clientSocket.setSoTimeout(TIMEOUT);
            //发送完之后等待接受ack
            while (true) {
                byte[] receiveData = new byte[GROUP_SIZE];
                DatagramPacket receivePacket = new DatagramPacket(receiveData,
                receiveData.length);

                clientSocket.receive(receivePacket);
                //从receivePacket中拿到确认的编号
                long receiveAckSeqNum = getSeqNum(receivePacket);
                System.out.println("接收者已确认分组编号:"+receiveAckSeqNum);
                //如果接收方返回来的确认分组序号是合法的,更新发送方最新的确认分组号
                if (receiveAckSeqNum >= sendAckSeqNum && receiveAckSeqNum <=
                sendAckSeqNum + WINDOW_SIZE) {
                    sendAckSeqNum = receiveAckSeqNum;
                    //窗口移动
                    windowIndex = (int)sendAckSeqNum + 1;
                } else {
                    break;
                }
            }
        } catch (SocketTimeoutException e) {
            //当sendAckSeqNum < packetNum - 1 而且出现SocketTimeoutException的时候
            //说明确认编号错误,重新发送窗口里所有的分组
            if(sendAckSeqNum < packetNum - 1){
                //说明传送socket超时了,那么重传窗口里的分组
            }
        }
    }
}

```

```

        //发送窗口里面所有的分组
        for (int i = windowIndex; i < windowIndex + WINDOW_SIZE && i < packetNum; i++)
        {
            DatagramPacket sendPacket = new DatagramPacket(dataGroup[i], 0,
dataGroup[i].length, targetHost, targetPort);
            //发送分组
            try {
                clientSocket.send(sendPacket);
            } catch (IOException ex) {
                throw new RuntimeException(ex);
            }
        }
    }
    catch (IOException e) {
        throw new RuntimeException(e);
    }
}

System.out.println("全部数据已被接受!");
//关闭连接, 恢复窗口起始位置以及期望分组编号
clientSocket.close();
windowIndex = 0;
sendAckSeqNum = 0;
}

/**
 * 接受数据
 * @return 返回数据所在的字节输出流
 * @throws IOException
 */
@Override
public ByteArrayOutputStream receiveData() throws IOException {
    //重新尝试次数
    int time = 0;
    //计数, 模拟丢失分组
    int count = 0;
    // 期望接收到的分组
    long expectSeq = 0;
    //存储最后接收到的数据, 交付给上层
    ByteArrayOutputStream result = new ByteArrayOutputStream();
    DatagramSocket datagramSocket = new DatagramSocket(port);
    DatagramPacket receivePacket;
    //为接收设置超时时间
    datagramSocket.setSoTimeout(TIMEOUT);

```

```
while (true) {
    count++;
    try {
        //接受一个分组
        byte[] receive = new byte[GROUP_SIZE];
        receivePacket = new DatagramPacket(receive, receive.length, targetHost, targetPort);
        datagramSocket.receive(receivePacket);
        //从接受的数据中提取分组号
        long seq = getSeqNum(receivePacket);
        // 若不是期望接收的分组，则丢弃
        if(expectSeq != seq) continue;
        // 模拟丢包
        if(count % LOSS == 0) {
            System.out.println("丢弃此分组~:"+seq);
            continue;
        }
        //将收到的分组写入结果中去
        result.write(receive, 8, receivePacket.getLength() - 8);
        expectSeq++;
        //创建一个ack报文，含有确认的分组号
        ByteArrayOutputStream temp = new ByteArrayOutputStream();
        ByteBuffer longBuffer = ByteBuffer.allocate(Long.BYTES);
        longBuffer.putLong(seq);
        byte[] longTemp = longBuffer.array();
        temp.write(longTemp, 0, Long.BYTES);
        byte[] seqPacket = temp.toByteArray();
        //发送ack确认分组
        receivePacket = new DatagramPacket(seqPacket, seqPacket.length, targetHost, targetPort);
        datagramSocket.send(receivePacket);
        System.out.println("接收到分组: seq " + seq);
        //如果收到了数据，计数置为0
        time = 0;
    } catch (SocketTimeoutException e) {
        //超时一次，time++
        time ++;
    }
    // 超出最大接收时间，则接收结束，写出数据
    if(time > OUT_TIME_TRY_TIMES) {
        break;
    }
}
//关闭连接
datagramSocket.close();
return result;
```

```
}  
/**  
 * 从发送的分组里面提取出来seqNum  
 * @param receivePacket  
 * @return  
 */  
private long getSeqNum(DatagramPacket receivePacket) {  
    byte[] data = receivePacket.getData();  
    // 填充byteArray，确保前8个字节能够构成一个long值  
    // 从字节数组中提取long值  
    ByteBuffer buffer = ByteBuffer.wrap(data);  
    return buffer.getLong();  
}  
/**  
 * 将数据拆分,并拼接成seq + data形式  
 * @param dataStream 要拆分的数据  
 * @param size 分组的字节数  
 * @return  
 */  
private byte[][] splitData(ByteArrayOutputStream dataStream, int size) {  
    byte[] data = dataStream.toByteArray();  
    //分组的个数  
    int numPackets = (int) Math.ceil(((double) data.length / size);  
    byte[][] result = new byte[numPackets][size];  
    long currentSeq = 0;  
    //将数据按照seq + data的形式进行拼接  
    int dataStartIndex = 0;  
    //在每个分组前面都加上分组编号  
    for (int i = 0; i < numPackets; i++) {  
        ByteArrayOutputStream temp = new ByteArrayOutputStream();  
        //将分组号写入到数据  
        ByteBuffer longBuffer = ByteBuffer.allocate(Long.BYTES);  
        longBuffer.putLong(currentSeq);  
        byte[] longTemp = longBuffer.array();  
        temp.write(longTemp, 0, Long.BYTES);  
        //确定结束下标，防止超过数据总长度  
        int len = size - Long.BYTES;  
        if(dataStartIndex + len > data.length) {  
            len = data.length - dataStartIndex;  
        }  
        //将数据写入分组  
        temp.write(data, dataStartIndex, len);  
        //下一个分组在data的开始下标  
        dataStartIndex = dataStartIndex + len;
```

```

        result[i] = temp.toByteArray();
        currentSeq++;
    }
    return result;
}
}

```

SRTransferData代码:

```

public class SRTransferData implements TransferData {
    private final int port;
    //目的主机的地址
    private final InetAddress targetHost;
    //目的主机的端口号
    private final int targetPort;
    //窗口的大小
    private static final int WINDOW_SIZE = 6;
    //序号的数目
    private static final int SEQ_NUM = 2 * WINDOW_SIZE;
    //分组的最大数据长度（字节）
    private static final int GROUP_SIZE = 1024;
    //超时时间
    private static final int TIMEOUT = 800;
    //如果超时了,重新尝试的次数
    private static final int OUT_TIME_TRY_TIMES = 3;
    //进行模loss运算，来模拟数据丢失
    private static final int LOSS = 8;
    //窗口里面的元素是否已被确认
    private final List<Boolean> allPacket = new ArrayList<>();

    //窗口的起始位置
    private int windowIndex = 0;
    //已经确认的最新数据序号,比如：1,2,3都确认，那就是3
    private long sendAckSeqNum = 0;

    public SRTransferData(int port, InetAddress targetHost, int targetPort) {
        this.port = port;
        this.targetHost = targetHost;
        this.targetPort = targetPort;
    }

    @Override
    public void sendData(ByteArrayOutputStream data) {
        //1.首先先将数据进行拆分成 seq + data 的形式
    }
}

```

```

byte[][] dataGroup = splitData(data, GROUP_SIZE);
//分组的总数
int packetNum = dataGroup.length;
//将所有分组，全部设置为未验证
for (int i = 0; i < packetNum; i++) {
    allPacket.add(false);
}
DatagramSocket clientSocket;
try {
    clientSocket = new DatagramSocket(port);
} catch (SocketException e) {
    throw new RuntimeException(e);
}
//直到所有分组全被确认
while (!windowAllACK(allPacket)) {
    //发送窗口里没有被确认的分组
    for (int i = windowIndex; i < windowIndex + WINDOW_SIZE && i < packetNum; i++) {
        if (!allPacket.get(i)) {
            DatagramPacket sendPacket = new DatagramPacket(dataGroup[i], 0,
dataGroup[i].length, targetHost, targetPort);
            try {
                clientSocket.send(sendPacket);
            } catch (IOException e) {
                System.out.println("发送分组异常.....");
                throw new RuntimeException(e);
            }
        }
    }
    try {
        //设置超时时间
        clientSocket.setSoTimeout(TIMEOUT);
        //发送完之后等待接受ack
        while (true) {
            byte[] receiveData = new byte[GROUP_SIZE];
            DatagramPacket receivePacket = new DatagramPacket(receiveData,
receiveData.length);
            clientSocket.receive(receivePacket);
            //从receivePacket中拿到确认的编号
            long receiveAckSeqNum = getSeqNum(receivePacket);
            System.out.println("接收者已确认分组编号:" + receiveAckSeqNum);
            //如果接收方返回来的确认分组序号是合法的，确认对应分组
            if (receiveAckSeqNum >= windowIndex && receiveAckSeqNum < allPacket.size() &&
receiveAckSeqNum <= windowIndex + WINDOW_SIZE) {
                //确认对应分组
            }
        }
    }
}
    
```

```

        allPacket.set((int) receiveAckSeqNum, true);
        //如果下届被确认了，那么向前移动窗口
        while (windowIndex < allPacket.size() && allPacket.get(windowIndex)) {
            windowIndex++;
            sendAckSeqNum++;
        }
    } else {
        break;
    }
}
}
//如果接收超时，重新发送窗口里没有被确认的分组
catch (SocketTimeoutException e) {
    if (sendAckSeqNum < packetNum - 1) {
        //重传窗口里没有被确认的分组
        for (int i = windowIndex; i <= windowIndex + WINDOW_SIZE && i < packetNum; i++)
        {
            if (!allPacket.get(i)) {
                DatagramPacket sendPacket = new DatagramPacket(dataGroup[i], 0,
dataGroup[i].length, targetHost, targetPort);
                //发送分组
                try {
                    clientSocket.send(sendPacket);
                } catch (IOException ex) {
                    throw new RuntimeException(ex);
                }
            }
        }
    }
} catch (IOException ex) {
    throw new RuntimeException(ex);
}
}
System.out.println("全部数据已被接受!");
//关闭连接，将窗口位置以及确认号设置为0，为下次发送数据做准备
clientSocket.close();
windowIndex = 0;
sendAckSeqNum = 0;
//清空保存信息的集合
allPacket.clear();
}

/**
 * 判断窗口中所有的分组是否全部已被确认

```

```

    *@param window 窗口
    *@return 如果全被确认, 返回true, 否则返回false
    */
    private boolean windowAllACK(List<Boolean> window) {
        for (Boolean a : window) {
            if (!a) return false;
        }
        return true;
    }

    @Override
    public ByteArrayOutputStream receiveData() throws IOException {
        //缓存发送方发过来的乱序的分组数据
        HashMap<Integer, ByteArrayOutputStream> receiveCache = new HashMap<>();
        //接收方的窗口
        LinkedHashMap<Integer, Boolean> receiveWindow = new LinkedHashMap<>();
        //初始化接收方窗口
        //W+1<=2L 我们序号数直接选择2倍窗口大小
        for (int i = 0; i < SEQ_NUM; i++) {
            receiveWindow.put(i, false);
        }
        //期望接受的分组,也可以看做是下届
        long receiveBase = 0;
        //超时和计数模拟丢失
        int time = 0;
        int count = 0;
        // 按序输出流
        ByteArrayOutputStream result = new ByteArrayOutputStream();
        // server监听socket
        DatagramSocket datagramSocket = new DatagramSocket(port);
        DatagramPacket receivePacket;

        //为接收设置超时时间
        datagramSocket.setSoTimeout(TIMEOUT);
        while (true) {
            count++;
            try {
                //接收一个分组到receive中去
                ByteArrayOutputStream receive = new ByteArrayOutputStream();
                byte[] recv = new byte[1024];
                receivePacket = new DatagramPacket(recv, recv.length, targetHost, targetPort);
                datagramSocket.receive(receivePacket);
                //获取数据中的分组号
                long seq = getSeqNum(receivePacket);
            }
        }
    }

```



```
// 检测发回来的分组是不是期望的
if (seq < receiveBase || seq > receiveBase + WINDOW_SIZE - 1) continue;
//如果已被确认了continue
if (receiveWindow.containsKey((int)seq % SEQ_NUM) && receiveWindow.get((int) seq %
SEQ_NUM)) continue;

// 模拟丢包
if (count % LOSS == 0) {
    System.out.println("丢弃此分组~: "+seq);
    continue;
}
//构建ACK报文，确认收到某分组
receive.write(recv, 8, receivePacket.getLength() - 8);
ByteArrayOutputStream temp = new ByteArrayOutputStream();
ByteBuffer longBuffer = ByteBuffer.allocate(Long.BYTES);
longBuffer.putLong(seq);
byte[] longTemp = longBuffer.array();
temp.write(longTemp, 0, Long.BYTES);
byte[] seqPacket = temp.toByteArray();
//发送ack确认报文
receivePacket = new DatagramPacket(seqPacket, seqPacket.length, targetHost, targetPort);
datagramSocket.send(receivePacket);
System.out.println("接收到分组: seq " + seq);
receiveCache.put((int) seq, receive);
//窗口中确认分组
receiveWindow.replace((int)seq % SEQ_NUM, true);
//如果序号等于下届，那么就传输数据
if (seq == receiveBase) {
    int begin = (int) seq;
    //如果下届始终被确认，那么就一直发送
    while (receiveWindow.containsKey(begin % SEQ_NUM) &&
receiveWindow.get(begin % SEQ_NUM)){
        result.write(receiveCache.get(begin).toByteArray());
        //发送完就删了
        receiveCache.remove(begin);
        //窗口向前滚动
        receiveWindow.replace(begin % SEQ_NUM, false);
        begin++;
        receiveBase++;
    }
} else {
    //将分组先缓存起来
    receiveCache.put((int) seq, receive);
}
```

```
    }  
    time = 0;  
    } catch (SocketTimeoutException e) {  
        time++;  
    }  
    // 如果超时了，接收结束  
    if (time > OUT_TIME_TRY_TIMES) {  
        break;  
    }  
}  
datagramSocket.close();  
return result;  
}  
/**  
 * 从发送的分组里面提取出来seqNum  
 *  
 * @param receivePacket  
 * @return  
 */  
private long getSeqNum(DatagramPacket receivePacket) {  
    byte[] data = receivePacket.getData();  
    // 填充byteArray，确保前8个字节能够构成一个long值  
    // 从字节数组中提取long值  
    ByteBuffer buffer = ByteBuffer.wrap(data);  
    long seq = buffer.getLong();  
    return seq;  
}  
/**  
 * 将数据拆分,并拼接成seq + data形式  
 *  
 * @param dataStream 要拆分的数据  
 * @param size       分组的字节数  
 * @return  
 */  
private byte[][] splitData(ByteArrayOutputStream dataStream, int size) {  
    byte[] data = dataStream.toByteArray();  
    //得到数据分组的个数  
    int numPackets = (int) Math.ceil(((double) data.length / size));  
    byte[][] result = new byte[numPackets][size];  
    long currentSeq = 0;  
    //将数据按照seq + data的形式进行拼接  
    int dataStartIndex = 0;  
    for (int i = 0; i < numPackets; i++) {
```

```
        ByteArrayOutputStream temp = new ByteArrayOutputStream();
        ByteBuffer longBuffer = ByteBuffer.allocate(Long.BYTES);
        longBuffer.putLong(currentSeq);
        byte[] longTemp = longBuffer.array();
        temp.write(longTemp, 0, Long.BYTES);
        int len = size - Long.BYTES;
        if (dataStartIndex + len > data.length) {
            len = data.length - dataStartIndex;
        }
        temp.write(data, dataStartIndex, len);
        dataStartIndex = dataStartIndex + len;
        result[i] = temp.toByteArray();
        currentSeq++;
    }
    return result;
}
```