哈爾濱工業大學
Harbin Institute of Technology

计算机网络
课程实验报告

实验名称	可靠数据传输协议—GBN 协议的设计与实现					
姓名	田一间		院系	计算机学院		
班级	1636101		学号	1160300617		
任课教师	李全龙		指导教师	李全龙		
实验地点	格物楼 213		实验时间	2018 年 11 月 3 日		
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						

计算机科学与技术学院 SINCE 1956...
School of Computer Science and Technology

实验目的：

理解滑动窗口协议的基本原理；掌握 GBN 的工作原理；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GBN协议，实现单向可靠数据传输（服务器到客户的数据传输）。

2) 模拟引入数据包的丢失，验证所设计协议的有效性。

3) 改进所设计的 GBN 协议，支持双向数据传输。

4) 将所设计的 GBN 协议改进为 SR 协议。

实验过程：

1) 设计GBN协议数据分组格式

发送端：

Seq	Data	0

共1026字节，其中Seq是序列号，1个字节，取值为0~255；Data为1024字节，存储数据；最末尾为0，表示结束。

接收端：

ACK	0

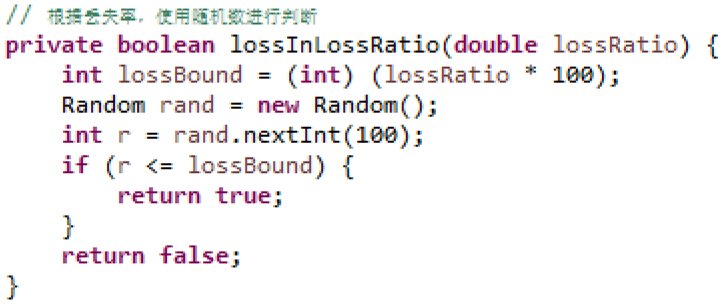
接收端为ACK的数据帧，不需附带任何数据，ACK字段为1个字节，表示序列号数值，其余字节为0。

2) 设计滑动窗口

发送窗口大小W为10，序列号个数N为20，满足W+1 <= N。

3) 设计数据分组丢失验证模拟方法

接收端采用一定概率值进行丢包与丢失ACK的模拟，每次收到包时或要发送ACK时，生成随机数，判断其是否在概率范围内以决定是否发生丢失。



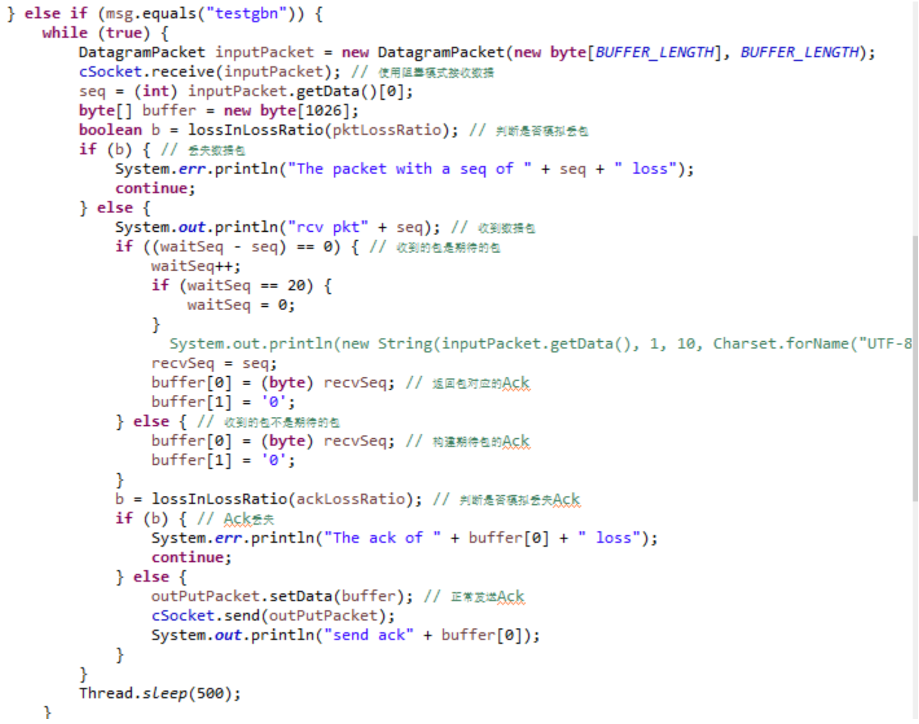
```
// 根据丢失率，使用随机数进行判断
private boolean lossInLossRatio(double lossRatio) {
    int lossBound = (int) (lossRatio * 100);
    Random rand = new Random();
    int r = rand.nextInt(100);
    if (r <= lossBound) {
        return true;
    }
    return false;
}
```

4) GBN协议实现

客户端：

使用阻塞模式接收数据包，当接收到一个包时，使用随机概率判断该包是否需要模拟丢失。若不需丢失，则判断该包是否是期待接收的数据包，若是，则构建该序列号的Ack数据帧，若不是，则构建期待序列号的数据帧。接着使用随机概率判断该包是否需要模拟Ack丢失，若是，直接返回，若不是，则返回该Ack数据帧。

关键代码：



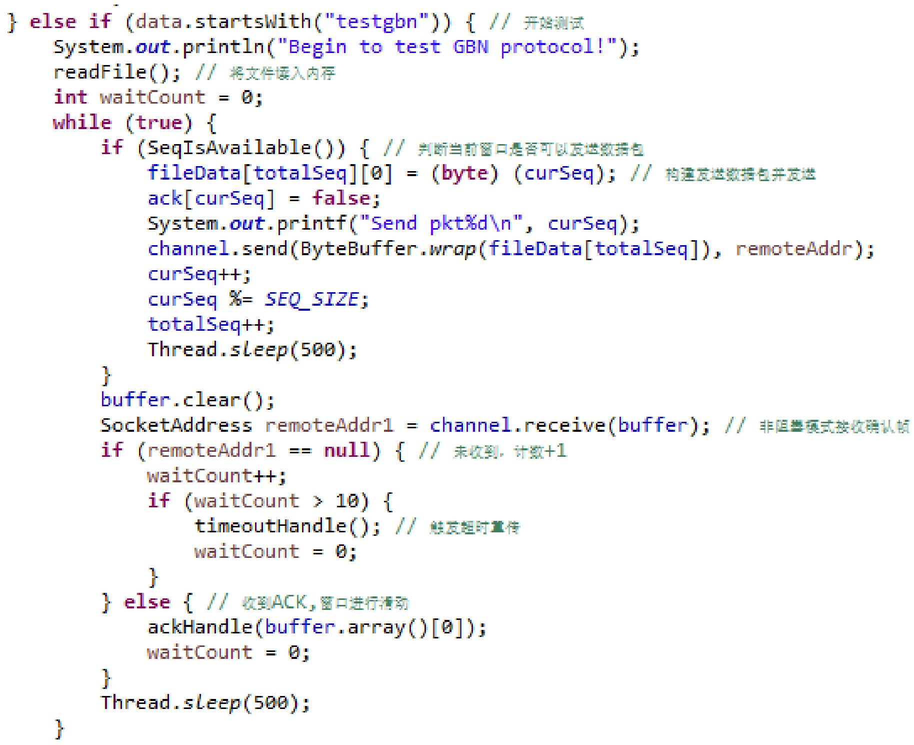
```
} else if (msg.equals("testgbn")) {
    while (true) {
        DatagramPacket inputPacket = new DatagramPacket(new byte[BUFFER_LENGTH], BUFFER_LENGTH);
        cSocket.receive(inputPacket); // 使用阻塞模式接收数据
        seq = (int) inputPacket.getData()[0];
        byte[] buffer = new byte[1026];
        boolean b = lossInLossRatio(pktLossRatio); // 判断是否模拟丢包
        if (b) { // 丢失数据包
            System.err.println("The packet with a seq of " + seq + " loss");
            continue;
        } else {
            System.out.println("rcv pkt" + seq); // 收到数据包
            if ((waitSeq - seq) == 0) { // 收到的包是期待的包
                waitSeq++;
                if (waitSeq == 20) {
                    waitSeq = 0;
                }
                  System.out.println(new String(inputPacket.getData(), 1, 10, Charset.forName("UTF-8
                recvSeq = seq;
                buffer[0] = (byte) recvSeq; // 返回包对应的Ack
                buffer[1] = '0';
            } else { // 收到的包不是期待的包
                buffer[0] = (byte) recvSeq; // 构建期待包的Ack
                buffer[1] = '0';
            }
            b = lossInLossRatio(ackLossRatio); // 判断是否模拟丢失Ack
            if (b) { // Ack丢失
                System.err.println("The ack of " + buffer[0] + " loss");
                continue;
            } else {
                outPutPacket.setData(buffer); // 正常发送Ack
                cSocket.send(outPutPacket);
                System.out.println("send ack" + buffer[0]);
            }
        }
        Thread.sleep(500);
    }
```

服务端：

当接收到客户端的协议测试指令后，读取文件构建分组数据包。判断当前窗口是否可以继续发送，若可以发送，则发送相应数据包，窗口内部进行相应下标调整。

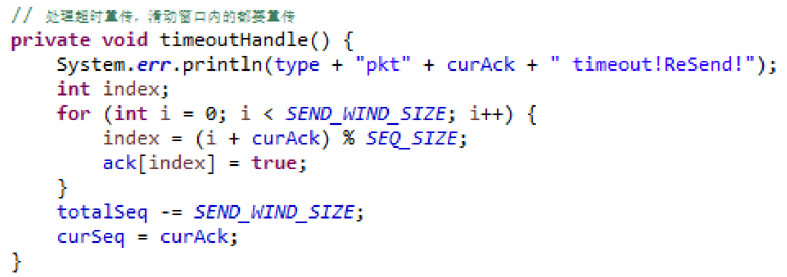
使用非阻塞方式接收客户端返回的Ack，线程延迟作为计时，则可通过一个计数器来作为计时器。没有接收到确认Ack，则计数器加1，超时时触发超时重传事件。接收到确认的Ack，则窗口进行滑动，进入下一轮。

关键代码：



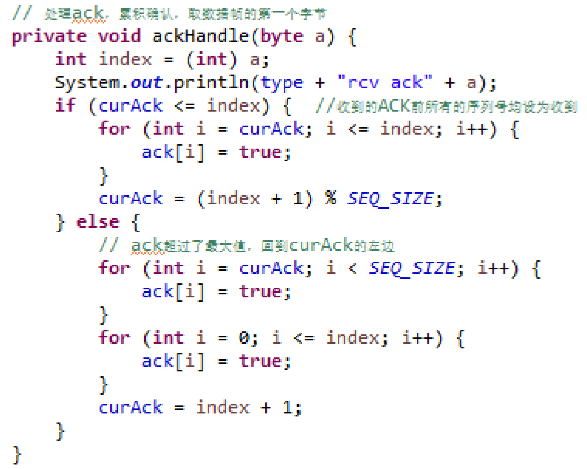
```
} else if (data.startsWith("testgbn")) { // 开始测试
    System.out.println("Begin to test GBN protocol!");
    readFile(); // 将文件读入内存
    int waitCount = 0;
    while (true) {
        if (SeqIsAvailable()) { // 判断当前窗口是否可以发送数据包
            fileData[totalSeq][0] = (byte) (curSeq); // 构建发送数据包并发送
            ack[curSeq] = false;
            System.out.printf("Send pkt%d\n", curSeq);
            channel.send(ByteBuffer.wrap(fileData[totalSeq]), remoteAddr);
            curSeq++;
            curSeq %= SEQ_SIZE;
            totalSeq++;
            Thread.sleep(500);
        }
        buffer.clear();
        SocketAddress remoteAddr1 = channel.receive(buffer); // 非阻塞模式接收确认帧
        if (remoteAddr1 == null) { // 未收到，计数+1
            waitCount++;
            if (waitCount > 10) {
                timeoutHandle(); // 触发超时重传
                waitCount = 0;
            }
        } else { // 收到ACK,窗口进行滑动
            ackHandle(buffer.array()[0]);
            waitCount = 0;
        }
        Thread.sleep(500);
    }
```

超时重传函数：



```
// 处理超时重传，滑动窗口内的都要重传
private void timeoutHandle() {
    System.err.println(type + "pkt" + curAck + " timeout!ReSend!");
    int index;
    for (int i = 0; i < SEND_WIND_SIZE; i++) {
        index = (i + curAck) % SEQ_SIZE;
        ack[index] = true;
    }
    totalSeq -= SEND_WIND_SIZE;
    curSeq = curAck;
}
```

Ack处理：

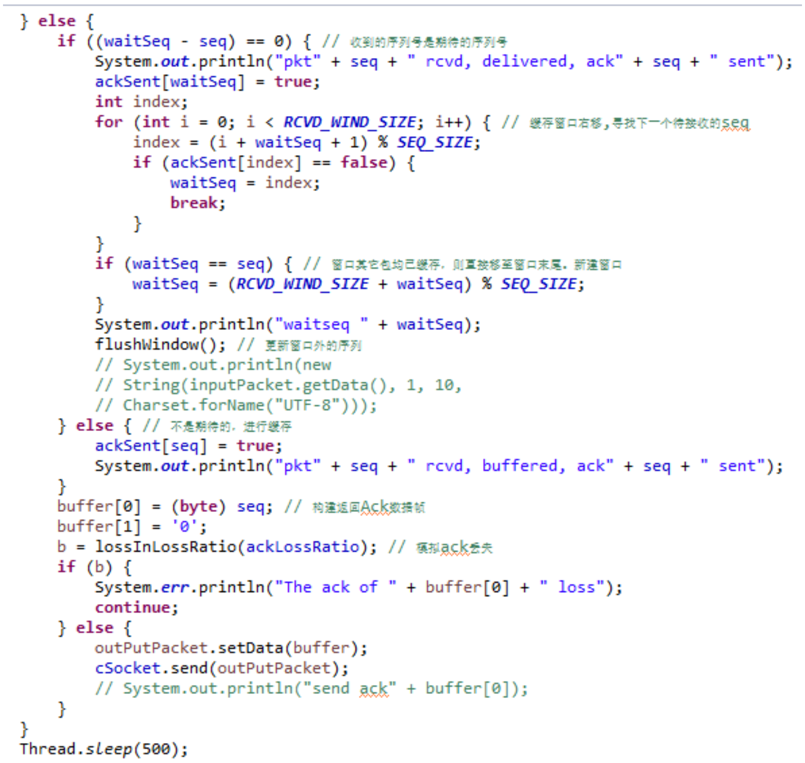


```
// 处理ack，累积确认，取数据帧的第一个字节
private void ackHandle(byte a) {
    int index = (int) a;
    System.out.println(type + "rcv ack" + a);
    if (curAck <= index) {  //收到的ACK前所有的序列号均设为收到
        for (int i = curAck; i <= index; i++) {
            ack[i] = true;
        }
        curAck = (index + 1) % SEQ_SIZE;
    } else {
        // ack超过了最大值，回到curAck的左边
        for (int i = curAck; i < SEQ_SIZE; i++) {
            ack[i] = true;
        }
        for (int i = 0; i <= index; i++) {
            ack[i] = true;
        }
        curAck = index + 1;
    }
}
```

5) SR协议实现

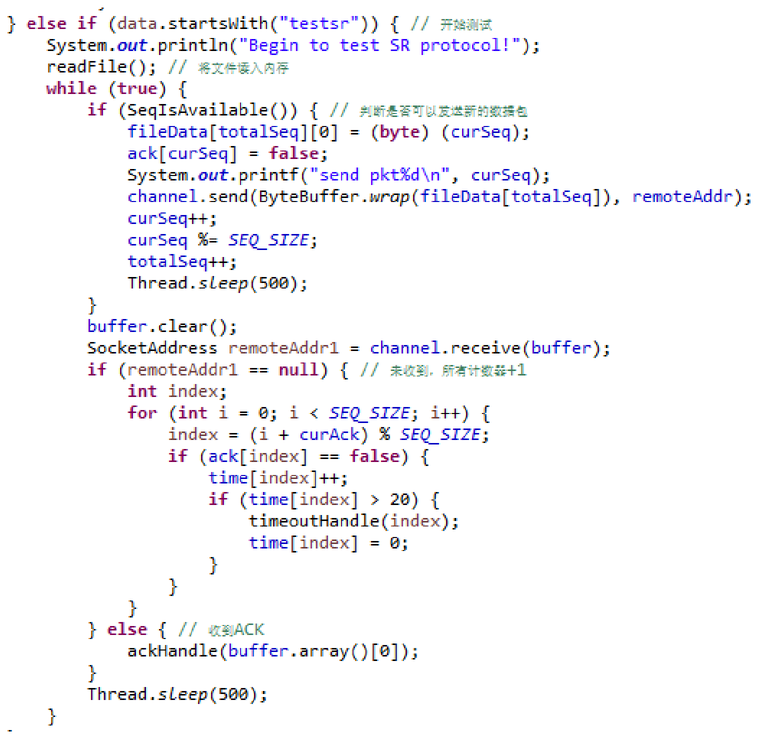
SR协议在GBN协议的基础上进行设计，接收方对每个分组单独进行确认，设置缓存机制，缓存乱序到达的分组。发送方为每个分组设置定时器，只重传那些没有收到ACK的分组。

客户端：



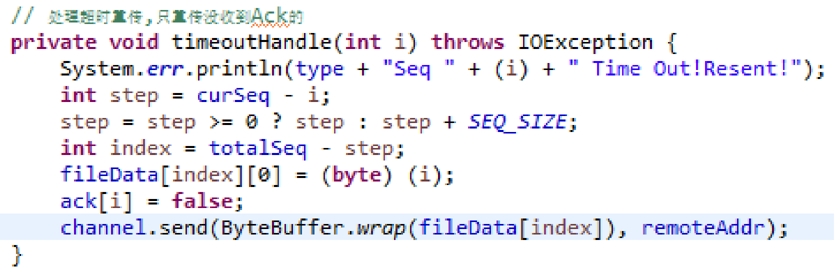
```
} else {
    if ((waitSeq - seq) == 0) { // 收到的序列号是期待的序列号
        System.out.println("pkt" + seq + " rcvd, delivered, ack" + seq + " sent");
        ackSent[waitSeq] = true;
        int index;
        for (int i = 0; i < RCVD_WIND_SIZE; i++) { // 缓存窗口右移,寻找下一个待接收的seq
            index = (i + waitSeq + 1) % SEQ_SIZE;
            if (ackSent[index] == false) {
                waitSeq = index;
                break;
            }
        }
        if (waitSeq == seq) { // 窗口其它包均已缓存，则直接移至窗口末尾，新建窗口
            waitSeq = (RCVD_WIND_SIZE + waitSeq) % SEQ_SIZE;
        }
        System.out.println("waitseq " + waitSeq);
        flushWindow(); // 更新窗口外的序列
        // System.out.println(new
        // String(inputPacket.getData(), 1, 10,
        // Charset.forName("UTF-8")));
    } else { // 不是期待的，进行缓存
        ackSent[seq] = true;
        System.out.println("pkt" + seq + " rcvd, buffered, ack" + seq + " sent");
    }
    buffer[0] = (byte) seq; // 构建返回Ack数据帧
    buffer[1] = '0';
    b = lossInLossRatio(ackLossRatio); // 模拟ack丢失
    if (b) {
        System.err.println("The ack of " + buffer[0] + " loss");
        continue;
    } else {
        outPutPacket.setData(buffer);
        cSocket.send(outPutPacket);
        // System.out.println("send ack" + buffer[0]);
    }
}
Thread.sleep(500);
```

服务端：



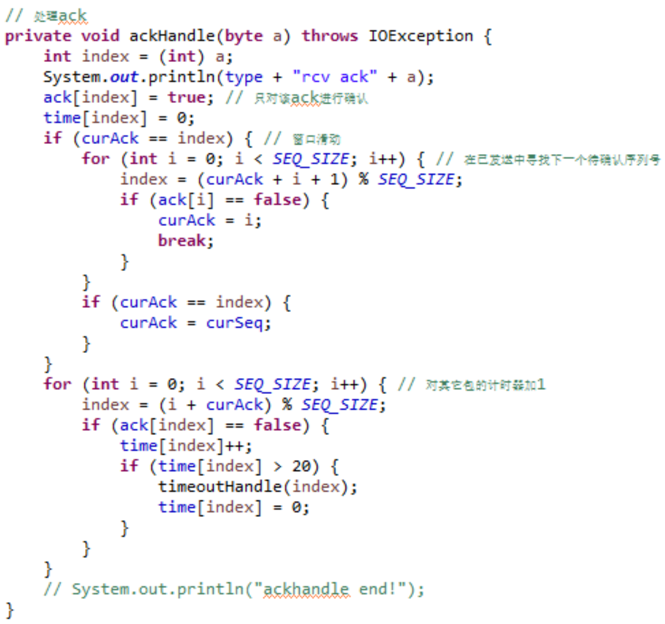
```
} else if (data.startsWith("testsr")) { // 开始测试
    System.out.println("Begin to test SR protocol!");
    readFile(); // 将文件读入内存
    while (true) {
        if (SeqIsAvailable()) { // 判断是否可以发送新的数据包
            fileData[totalSeq][0] = (byte) (curSeq);
            ack[curSeq] = false;
            System.out.printf("send pkt%d\n", curSeq);
            channel.send(ByteBuffer.wrap(fileData[totalSeq]), remoteAddr);
            curSeq++;
            curSeq %= SEQ_SIZE;
            totalSeq++;
            Thread.sleep(500);
        }
        buffer.clear();
        SocketAddress remoteAddr1 = channel.receive(buffer);
        if (remoteAddr1 == null) { // 未收到，所有计数器+1
            int index;
            for (int i = 0; i < SEQ_SIZE; i++) {
                index = (i + curAck) % SEQ_SIZE;
                if (ack[index] == false) {
                    time[index]++;
                    if (time[index] > 20) {
                        timeoutHandle(index);
                        time[index] = 0;
                    }
                }
            }
        } else { // 收到ACK
            ackHandle(buffer.array()[0]);
        }
        Thread.sleep(500);
    }
```

超时重传函数：



```
// 处理超时重传,只重传没收到Ack的
private void timeoutHandle(int i) throws IOException {
    System.err.println(type + "Seq " + (i) + " Time Out!Resent!");
    int step = curSeq - i;
    step = step >= 0 ? step : step + SEQ_SIZE;
    int index = totalSeq - step;
    fileData[index][0] = (byte) (i);
    ack[i] = false;
    channel.send(ByteBuffer.wrap(fileData[index]), remoteAddr);
}
```

Ack处理：

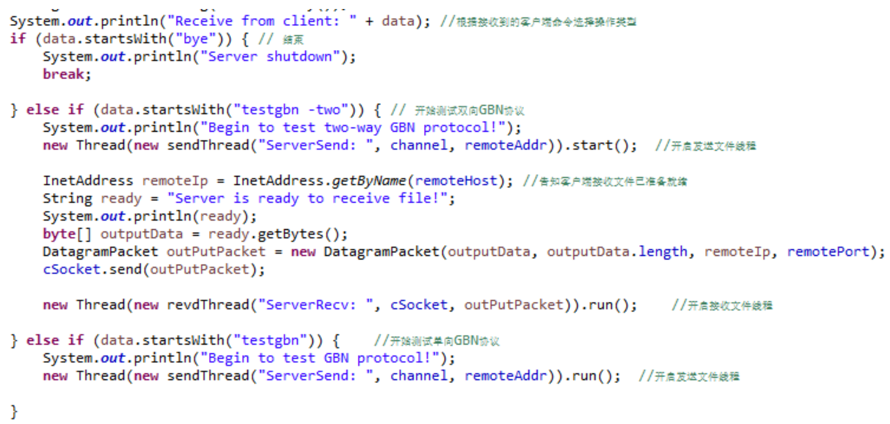


```
// 处理ack
private void ackHandle(byte a) throws IOException {
    int index = (int) a;
    System.out.println(type + "rcv ack" + a);
    ack[index] = true; // 只对该ack进行确认
    time[index] = 0;
    if (curAck == index) { // 窗口滑动
        for (int i = 0; i < SEQ_SIZE; i++) { // 在已发送中寻找下一个待确认序列号
            index = (curAck + i + 1) % SEQ_SIZE;
            if (ack[i] == false) {
                curAck = i;
                break;
            }
        }
        if (curAck == index) {
            curAck = curSeq;
        }
    }
    for (int i = 0; i < SEQ_SIZE; i++) { // 对其它包的计时器加1
        index = (i + curAck) % SEQ_SIZE;
        if (ack[index] == false) {
            time[index]++;
            if (time[index] > 20) {
                timeoutHandle(index);
                time[index] = 0;
            }
        }
    }
    // System.out.println("ackhandle end!");
}
```

6) 双向传输实现

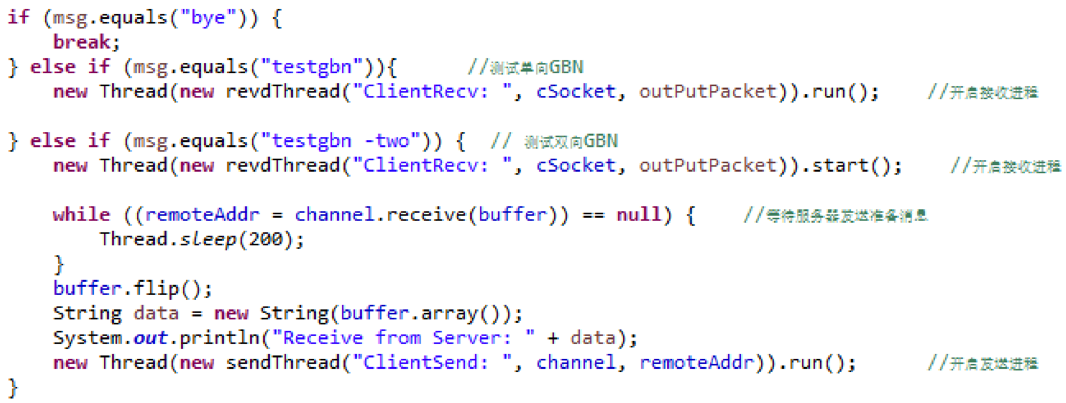
客户端和服务器使用两个端口进行传输和接收，双线程并发进行，以达到全双工双向文件传输的效果。

服务器端：



```
System.out.println("Receive from client: " + data); //根据接收到的客户端命令选择操作类型
if (data.startsWith("bye")) { // 结束
    System.out.println("Server shutdown");
    break;

} else if (data.startsWith("testgbn -two")) { // 开始测试双向GBN协议
    System.out.println("Begin to test two-way GBN protocol!");
    new Thread(new sendThread("ServerSend: ", channel, remoteAddr)).start();  //开启发送文件线程

    InetAddress remoteIp = InetAddress.getByName(remoteHost); //告知客户端接收文件已准备就绪
    String ready = "Server is ready to receive file!";
    System.out.println(ready);
    byte[] outputData = ready.getBytes();
    DatagramPacket outPutPacket = new DatagramPacket(outputData, outputData.length, remoteIp, remotePort);
    cSocket.send(outPutPacket);

    new Thread(new revdThread("ServerRecv: ", cSocket, outPutPacket)).run();   //开启接收文件线程

} else if (data.startsWith("testgbn")) {    //开始测试单向GBN协议
    System.out.println("Begin to test GBN protocol!");
    new Thread(new sendThread("ServerSend: ", channel, remoteAddr)).run();  //开启发送文件线程

}
```

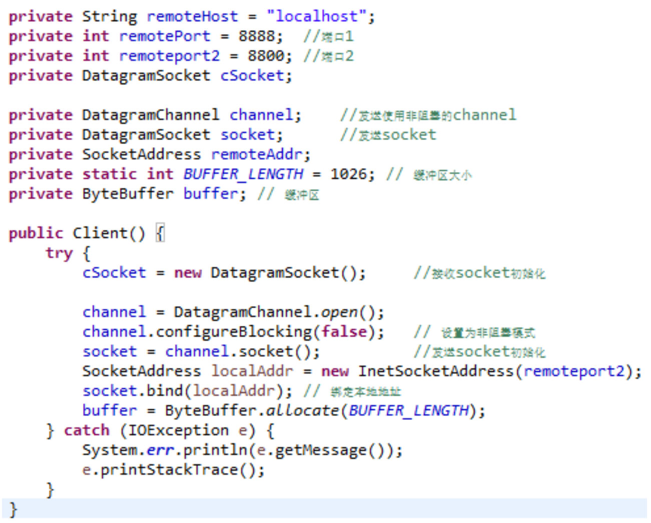
客户端：



```
if (msg.equals("bye")) {
    break;
} else if (msg.equals("testgbn")){      //测试单向GBN
    new Thread(new revdThread("ClientRecv: ", cSocket, outPutPacket)).run();     //开启接收进程

} else if (msg.equals("testgbn -two")) {  // 测试双向GBN
    new Thread(new revdThread("ClientRecv: ", cSocket, outPutPacket)).start();     //开启接收进程

    while ((remoteAddr = channel.receive(buffer)) == null) {     //等待服务器发送准备消息
        Thread.sleep(200);
    }
    buffer.flip();
    String data = new String(buffer.array());
    System.out.println("Receive from Server: " + data);
    new Thread(new sendThread("ClientSend: ", channel, remoteAddr)).run();      //开启发送进程
}
```

其中的 revdThread 和 sendThread 即为接收和发送进程，客户端和服务端各自构建接收和发送的socket，发送使用非阻塞的DatagramChannel，接收使用阻塞的DataSocket，初始化后，根据用户命令选择运行相应的线程即可。

初始化示例(客户端)：

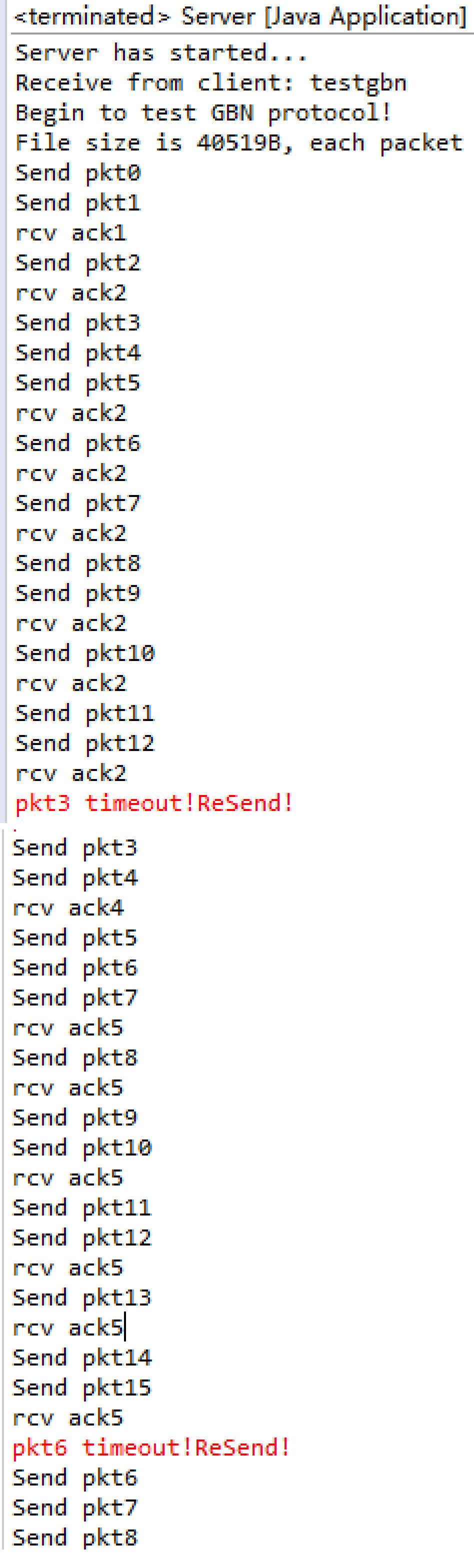


```
private String remoteHost = "localhost";
private int remotePort = 8888;  //端口1
private int remoteport2 = 8800; //端口2
private DatagramSocket cSocket;

private DatagramChannel channel;    //发送使用非阻塞的channel
private DatagramSocket socket;      //发送socket
private SocketAddress remoteAddr;
private static int BUFFER_LENGTH = 1026; // 缓冲区大小
private ByteBuffer buffer; // 缓冲区

public Client() {
    try {
        cSocket = new DatagramSocket();     //接收socket初始化

        channel = DatagramChannel.open();
        channel.configureBlocking(false);   // 设置为非阻塞模式
        socket = channel.socket();          //发送socket初始化
        SocketAddress localAddr = new InetSocketAddress(remoteport2);
        socket.bind(localAddr); // 绑定本地地址
        buffer = ByteBuffer.allocate(BUFFER_LENGTH);
    } catch (IOException e) {
        System.err.println(e.getMessage());
        e.printStackTrace();
    }
}
```

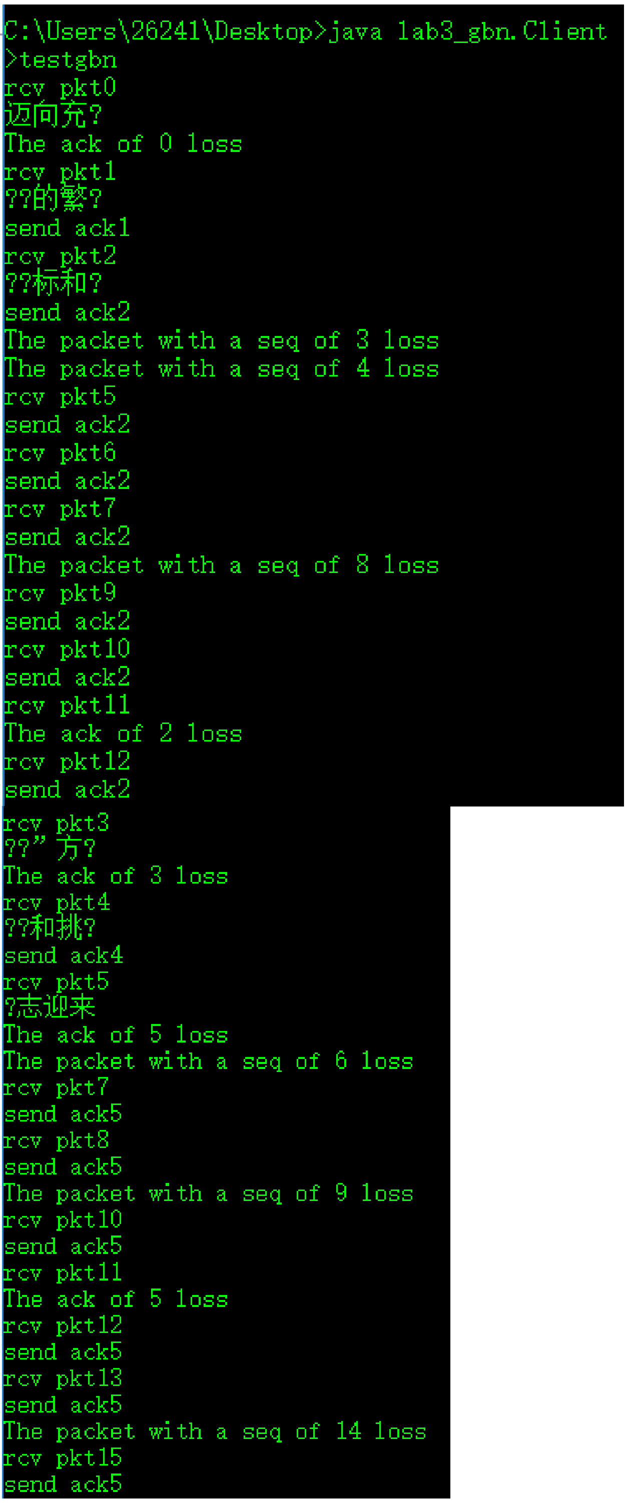
实验结果：

采用演示截图、文字说明等方式，给出本次实验的实验结果。

GBN：



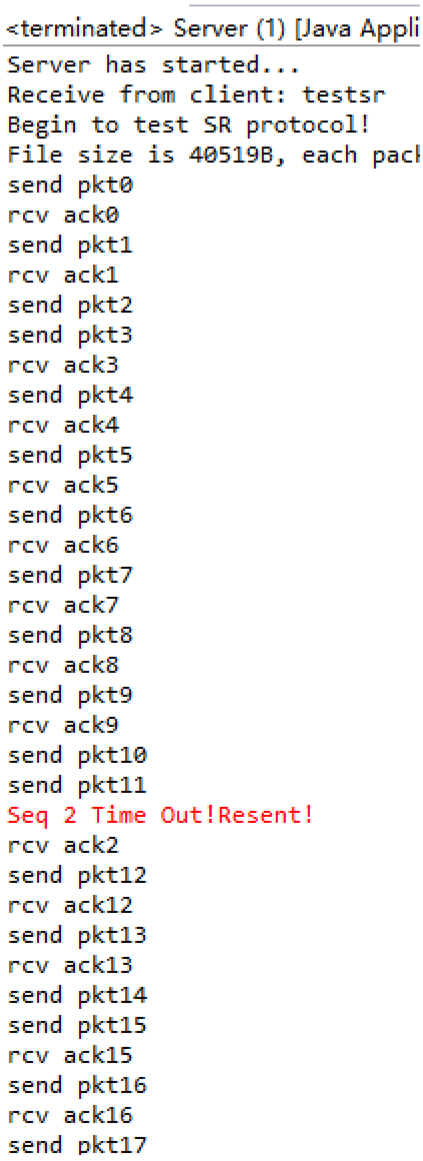
```
<terminated> Server [Java Application]
Server has started...
Receive from client: testgbn
Begin to test GBN protocol!
File size is 40519B, each packet
Send pkt0
Send pkt1
rcv ack1
Send pkt2
rcv ack2
Send pkt3
Send pkt4
Send pkt5
rcv ack2
Send pkt6
rcv ack2
Send pkt7
rcv ack2
Send pkt8
Send pkt9
rcv ack2
Send pkt10
rcv ack2
Send pkt11
Send pkt12
rcv ack2
pkt3 timeout!ReSend!
Send pkt3
Send pkt4
rcv ack4
Send pkt5
Send pkt6
Send pkt7
rcv ack5
Send pkt8
rcv ack5
Send pkt9
Send pkt10
rcv ack5
Send pkt11
Send pkt12
rcv ack5
Send pkt13
rcv ack5
Send pkt14
Send pkt15
rcv ack5
pkt6 timeout!ReSend!
Send pkt6
Send pkt7
Send pkt8
```



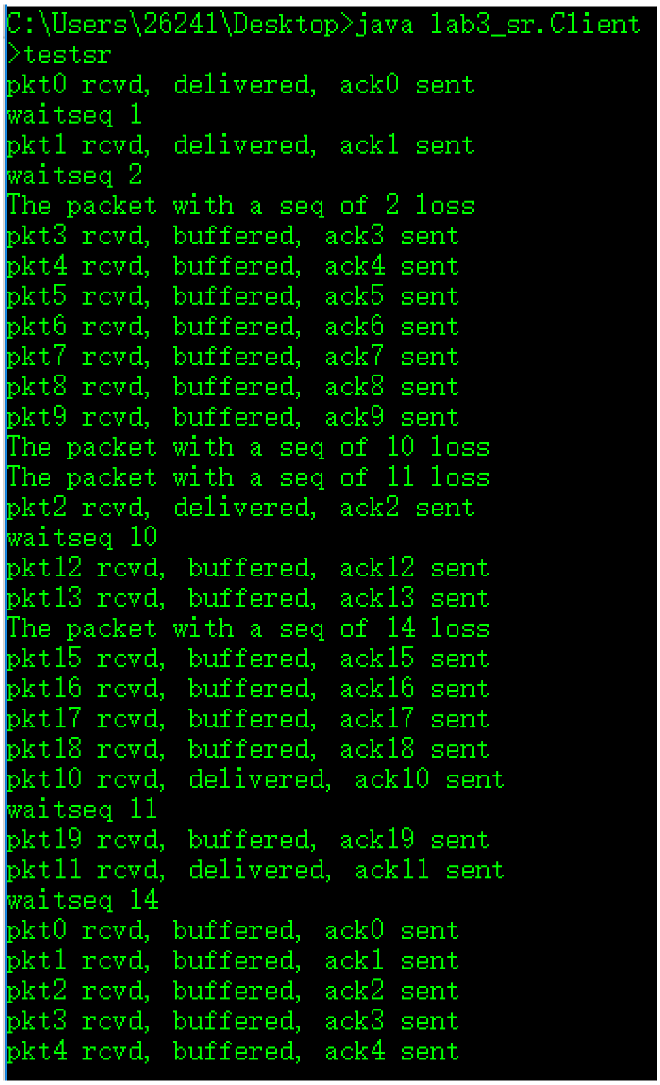
```
C:\Users\26241\Desktop>java lab3_gbn.Client
>testgbn
rcv pkt0
迈向充?
The ack of 0 loss
rcv pkt1
??的繁?
send ack1
rcv pkt2
??标和?
send ack2
The packet with a seq of 3 loss
The packet with a seq of 4 loss
rcv pkt5
send ack2
rcv pkt6
send ack2
rcv pkt7
send ack2
The packet with a seq of 8 loss
rcv pkt9
send ack2
rcv pkt10
send ack2
rcv pkt11
The ack of 2 loss
rcv pkt12
send ack2
rcv pkt3
??” 方?
The ack of 3 loss
rcv pkt4
??和挑?
send ack4
rcv pkt5
?志迎来
The ack of 5 loss
The packet with a seq of 6 loss
rcv pkt7
send ack5
rcv pkt8
send ack5
The packet with a seq of 9 loss
rcv pkt10
send ack5
rcv pkt11
The ack of 5 loss
rcv pkt12
send ack5
rcv pkt13
send ack5
The packet with a seq of 14 loss
rcv pkt15
send ack5
```

从运行结果可以看出，接收方pkt3发生丢失，于是一直重复发送ack2，服务器端等到 pkt3超时，进行重传，从pkt3依次重传窗口中的 pkt。稍后，接收方又发生了pkt6丢失，一直发送ack5，等到服务器pkt6超时，再次重传。

SR:



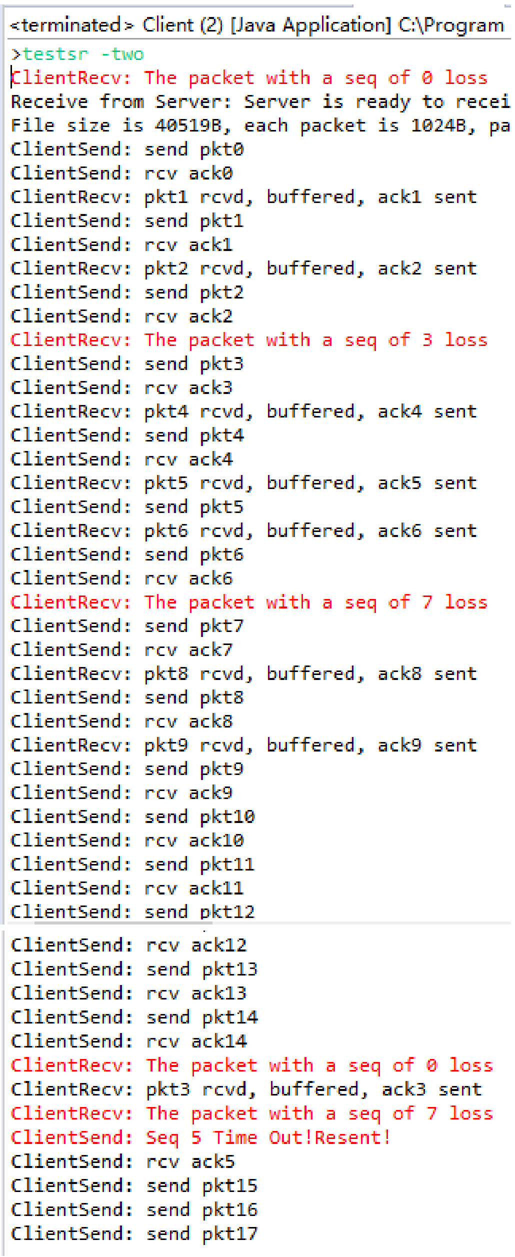
```
<terminated> Server (1) [Java Appli
Server has started...
Receive from client: testsr
Begin to test SR protocol!
File size is 40519B, each pack
send pkt0
rcv ack0
send pkt1
rcv ack1
send pkt2
send pkt3
rcv ack3
send pkt4
rcv ack4
send pkt5
rcv ack5
send pkt6
rcv ack6
send pkt7
rcv ack7
send pkt8
rcv ack8
send pkt9
rcv ack9
send pkt10
send pkt11
Seq 2 Time Out!Resent!
rcv ack2
send pkt12
rcv ack12
send pkt13
rcv ack13
send pkt14
send pkt15
rcv ack15
send pkt16
rcv ack16
send pkt17
```



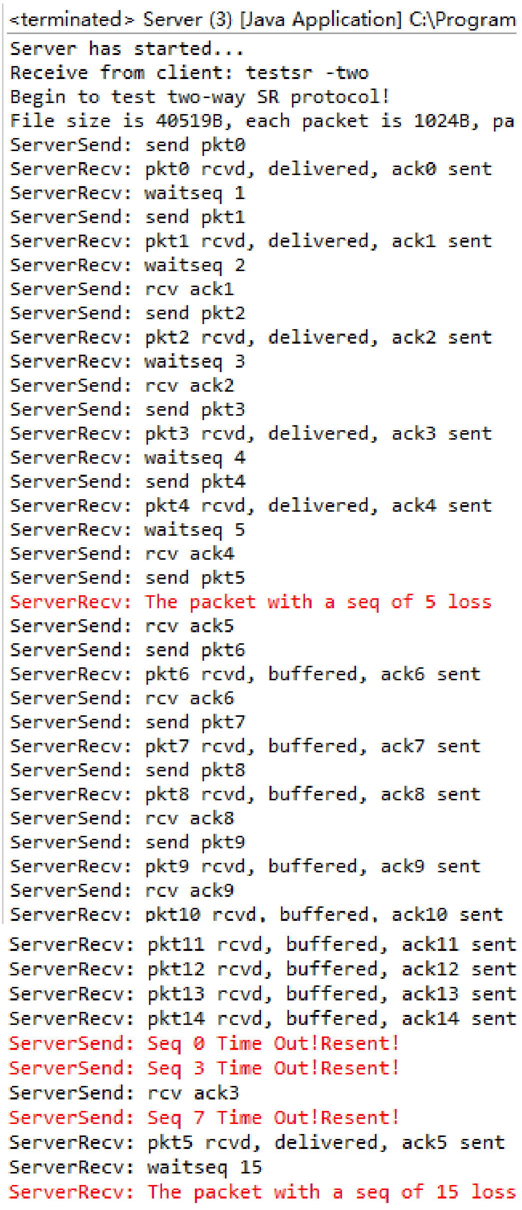
```
C:\Users\26241\Desktop>java lab3_sr.Client
>testsr
pkt0 rcvd, delivered, ack0 sent
waitseq 1
pkt1 rcvd, delivered, ack1 sent
waitseq 2
The packet with a seq of 2 loss
pkt3 rcvd, buffered, ack3 sent
pkt4 rcvd, buffered, ack4 sent
pkt5 rcvd, buffered, ack5 sent
pkt6 rcvd, buffered, ack6 sent
pkt7 rcvd, buffered, ack7 sent
pkt8 rcvd, buffered, ack8 sent
pkt9 rcvd, buffered, ack9 sent
The packet with a seq of 10 loss
The packet with a seq of 11 loss
pkt2 rcvd, delivered, ack2 sent
waitseq 10
pkt12 rcvd, buffered, ack12 sent
pkt13 rcvd, buffered, ack13 sent
The packet with a seq of 14 loss
pkt15 rcvd, buffered, ack15 sent
pkt16 rcvd, buffered, ack16 sent
pkt17 rcvd, buffered, ack17 sent
pkt18 rcvd, buffered, ack18 sent
pkt10 rcvd, delivered, ack10 sent
waitseq 11
pkt19 rcvd, buffered, ack19 sent
pkt11 rcvd, delivered, ack11 sent
waitseq 14
pkt0 rcvd, buffered, ack0 sent
pkt1 rcvd, buffered, ack1 sent
pkt2 rcvd, buffered, ack2 sent
pkt3 rcvd, buffered, ack3 sent
pkt4 rcvd, buffered, ack4 sent
```

从结果可以看到，接收方pkt2丢失，再收到来自发送方的包时，进行了缓存。发送方等到pkt2超时时，只重新发送了该包，而接收方收到该包后，窗口直接滑动到pkt10，因为pkt10也发生了丢包事件。

双向SR：



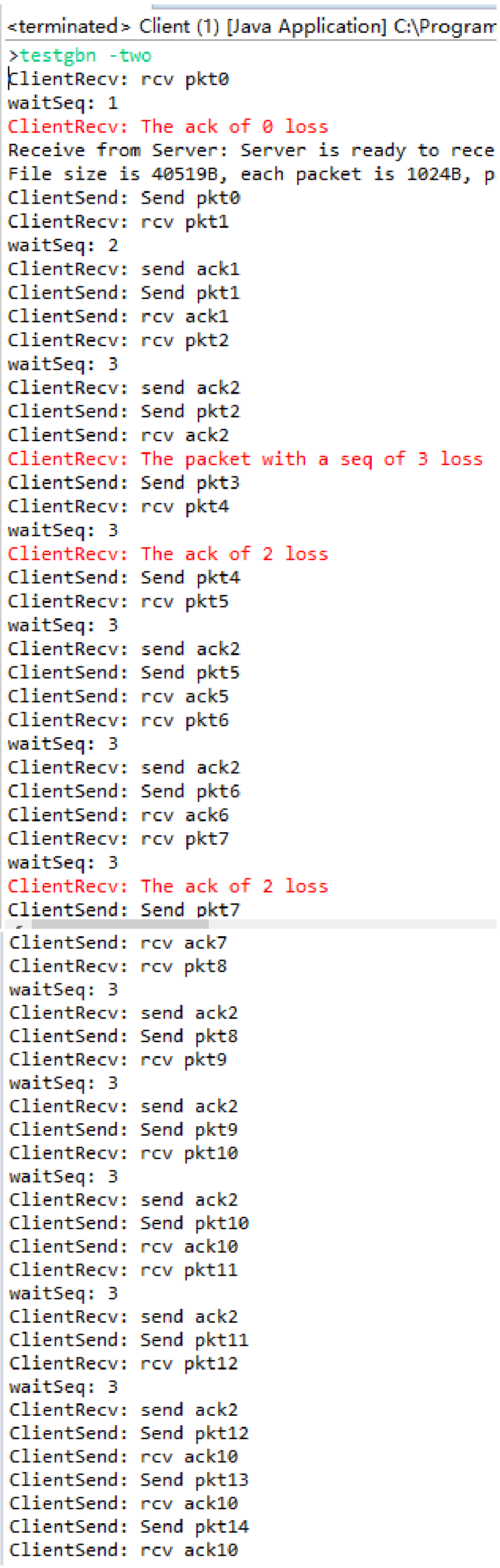
```
<terminated> Client (2) [Java Application] C:\Program
>testsr -two
ClientRecv: The packet with a seq of 0 loss
Receive from Server: Server is ready to recei
File size is 40519B, each packet is 1024B, pa
ClientSend: send pkt0
ClientSend: rcv ack0
ClientRecv: pkt1 rcvd, buffered, ack1 sent
ClientSend: send pkt1
ClientSend: rcv ack1
ClientRecv: pkt2 rcvd, buffered, ack2 sent
ClientSend: send pkt2
ClientSend: rcv ack2
ClientRecv: The packet with a seq of 3 loss
ClientSend: send pkt3
ClientSend: rcv ack3
ClientRecv: pkt4 rcvd, buffered, ack4 sent
ClientSend: send pkt4
ClientSend: rcv ack4
ClientRecv: pkt5 rcvd, buffered, ack5 sent
ClientSend: send pkt5
ClientRecv: pkt6 rcvd, buffered, ack6 sent
ClientSend: send pkt6
ClientSend: rcv ack6
ClientRecv: The packet with a seq of 7 loss
ClientSend: send pkt7
ClientSend: rcv ack7
ClientRecv: pkt8 rcvd, buffered, ack8 sent
ClientSend: send pkt8
ClientSend: rcv ack8
ClientRecv: pkt9 rcvd, buffered, ack9 sent
ClientSend: send pkt9
ClientSend: rcv ack9
ClientSend: send pkt10
ClientSend: rcv ack10
ClientSend: send pkt11
ClientSend: rcv ack11
ClientSend: send pkt12
ClientSend: rcv ack12
ClientSend: send pkt13
ClientSend: rcv ack13
ClientSend: send pkt14
ClientSend: rcv ack14
ClientRecv: The packet with a seq of 0 loss
ClientRecv: pkt3 rcvd, buffered, ack3 sent
ClientRecv: The packet with a seq of 7 loss
ClientSend: Seq 5 Time Out!Resent!
ClientSend: rcv ack5
ClientSend: send pkt15
ClientSend: send pkt16
ClientSend: send pkt17
```



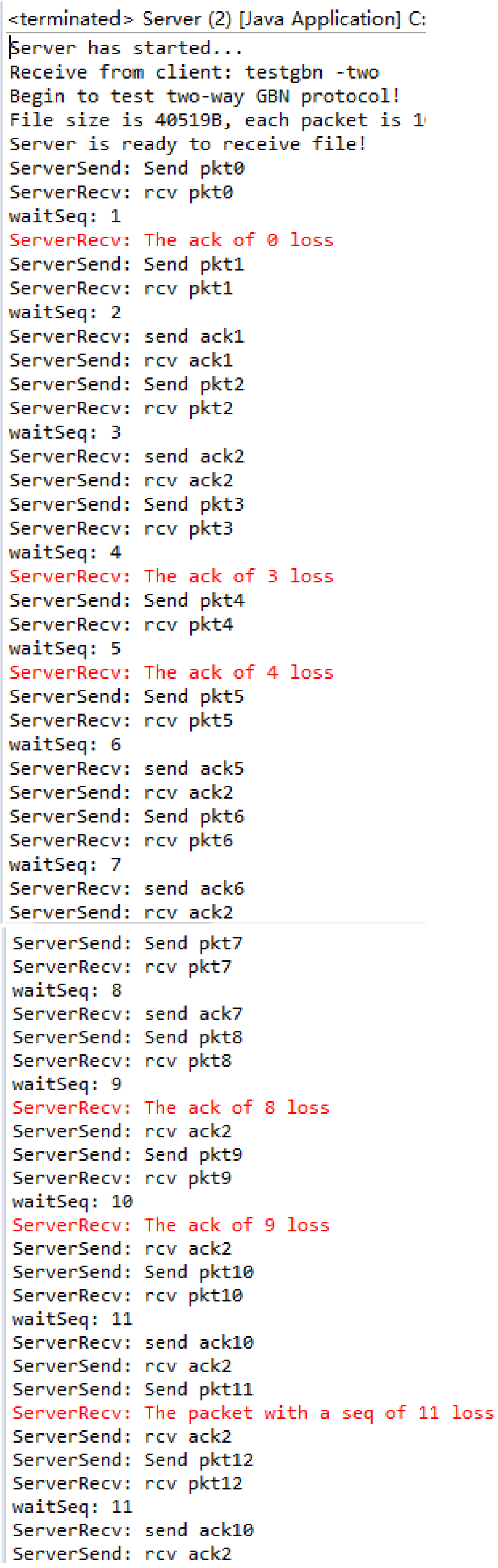
```
<terminated> Server (3) [Java Application] C:\Program
Server has started...
Receive from client: testsr -two
Begin to test two-way SR protocol!
File size is 40519B, each packet is 1024B, pa
ServerSend: send pkt0
ServerRecv: pkt0 rcvd, delivered, ack0 sent
ServerRecv: waitseq 1
ServerSend: send pkt1
ServerRecv: pkt1 rcvd, delivered, ack1 sent
ServerRecv: waitseq 2
ServerSend: rcv ack1
ServerSend: send pkt2
ServerRecv: pkt2 rcvd, delivered, ack2 sent
ServerRecv: waitseq 3
ServerSend: rcv ack2
ServerSend: send pkt3
ServerRecv: pkt3 rcvd, delivered, ack3 sent
ServerRecv: waitseq 4
ServerSend: send pkt4
ServerRecv: pkt4 rcvd, delivered, ack4 sent
ServerRecv: waitseq 5
ServerSend: rcv ack4
ServerSend: send pkt5
ServerRecv: The packet with a seq of 5 loss
ServerSend: rcv ack5
ServerSend: send pkt6
ServerRecv: pkt6 rcvd, buffered, ack6 sent
ServerSend: rcv ack6
ServerSend: send pkt7
ServerRecv: pkt7 rcvd, buffered, ack7 sent
ServerSend: send pkt8
ServerRecv: pkt8 rcvd, buffered, ack8 sent
ServerSend: rcv ack8
ServerSend: send pkt9
ServerRecv: pkt9 rcvd, buffered, ack9 sent
ServerSend: rcv ack9
ServerRecv: pkt10 rcvd, buffered, ack10 sent
ServerRecv: pkt11 rcvd, buffered, ack11 sent
ServerRecv: pkt12 rcvd, buffered, ack12 sent
ServerRecv: pkt13 rcvd, buffered, ack13 sent
ServerRecv: pkt14 rcvd, buffered, ack14 sent
ServerSend: Seq 0 Time Out!Resent!
ServerSend: Seq 3 Time Out!Resent!
ServerSend: rcv ack3
ServerSend: Seq 7 Time Out!Resent!
ServerRecv: pkt5 rcvd, delivered, ack5 sent
ServerRecv: waitseq 15
ServerRecv: The packet with a seq of 15 loss
```

双向SR协议的传输如上图，可以看到实现效果为全双工，服务器与客户端同时进行文件的发送与接收，且均符合SR协议的原理，在这里不再进行分析。

双向GBN协议：



```
<terminated> Client (1) [Java Application] C:\Program
>testgbn -two
ClientRecv: rcv pkt0
waitSeq: 1
ClientRecv: The ack of 0 loss
Receive from Server: Server is ready to rece
File size is 40519B, each packet is 1024B, p
ClientSend: Send pkt0
ClientRecv: rcv pkt1
waitSeq: 2
ClientRecv: send ack1
ClientSend: Send pkt1
ClientSend: rcv ack1
ClientRecv: rcv pkt2
waitSeq: 3
ClientRecv: send ack2
ClientSend: Send pkt2
ClientSend: rcv ack2
ClientRecv: The packet with a seq of 3 loss
ClientSend: Send pkt3
ClientRecv: rcv pkt4
waitSeq: 3
ClientRecv: The ack of 2 loss
ClientSend: Send pkt4
ClientRecv: rcv pkt5
waitSeq: 3
ClientRecv: send ack2
ClientSend: Send pkt5
ClientSend: rcv ack5
ClientRecv: rcv pkt6
waitSeq: 3
ClientRecv: send ack2
ClientSend: Send pkt6
ClientSend: rcv ack6
ClientRecv: rcv pkt7
waitSeq: 3
ClientRecv: The ack of 2 loss
ClientSend: Send pkt7
ClientSend: rcv ack7
ClientRecv: rcv pkt8
waitSeq: 3
ClientRecv: send ack2
ClientSend: Send pkt8
ClientRecv: rcv pkt9
waitSeq: 3
ClientRecv: send ack2
ClientSend: Send pkt9
ClientRecv: rcv pkt10
waitSeq: 3
ClientRecv: send ack2
ClientSend: Send pkt10
ClientSend: rcv ack10
ClientRecv: rcv pkt11
waitSeq: 3
ClientRecv: send ack2
ClientSend: Send pkt11
ClientRecv: rcv pkt12
waitSeq: 3
ClientRecv: send ack2
ClientSend: Send pkt12
ClientSend: rcv ack10
ClientSend: Send pkt13
ClientSend: rcv ack10
ClientSend: Send pkt14
ClientSend: rcv ack10
```



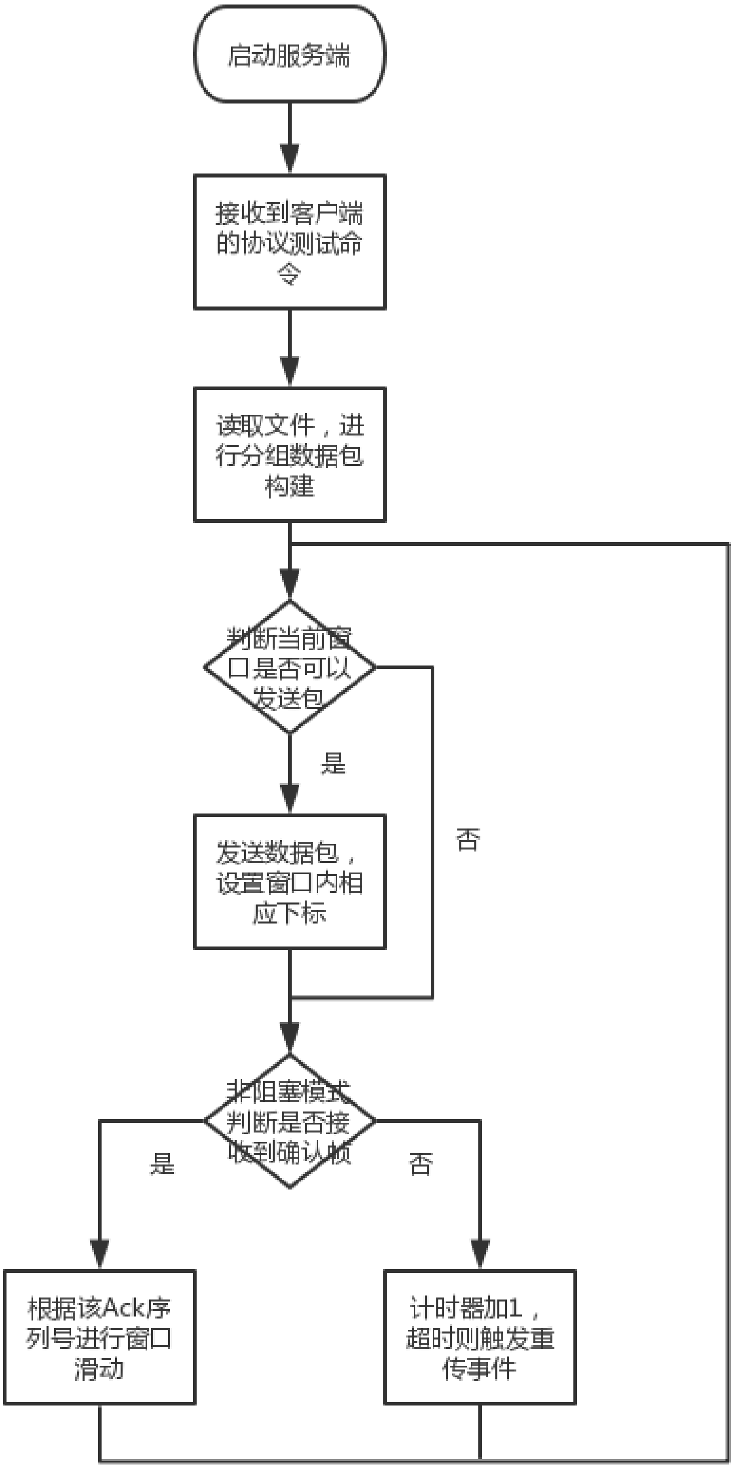
```
<terminated> Server (2) [Java Application] C:
Server has started...
Receive from client: testgbn -two
Begin to test two-way GBN protocol!
File size is 40519B, each packet is 1(
Server is ready to receive file!
ServerSend: Send pkt0
ServerRecv: rcv pkt0
waitSeq: 1
ServerRecv: The ack of 0 loss
ServerSend: Send pkt1
ServerRecv: rcv pkt1
waitSeq: 2
ServerRecv: send ack1
ServerSend: rcv ack1
ServerSend: Send pkt2
ServerRecv: rcv pkt2
waitSeq: 3
ServerRecv: send ack2
ServerSend: rcv ack2
ServerSend: Send pkt3
ServerRecv: rcv pkt3
waitSeq: 4
ServerRecv: The ack of 3 loss
ServerSend: Send pkt4
ServerRecv: rcv pkt4
waitSeq: 5
ServerRecv: The ack of 4 loss
ServerSend: Send pkt5
ServerRecv: rcv pkt5
waitSeq: 6
ServerRecv: send ack5
ServerSend: rcv ack2
ServerSend: Send pkt6
ServerRecv: rcv pkt6
waitSeq: 7
ServerRecv: send ack6
ServerSend: rcv ack2
ServerSend: Send pkt7
ServerRecv: rcv pkt7
waitSeq: 8
ServerRecv: send ack7
ServerSend: Send pkt8
ServerRecv: rcv pkt8
waitSeq: 9
ServerRecv: The ack of 8 loss
ServerSend: rcv ack2
ServerSend: Send pkt9
ServerRecv: rcv pkt9
waitSeq: 10
ServerRecv: The ack of 9 loss
ServerSend: rcv ack2
ServerSend: Send pkt10
ServerRecv: rcv pkt10
waitSeq: 11
ServerRecv: send ack10
ServerSend: rcv ack2
ServerSend: Send pkt11
ServerRecv: The packet with a seq of 11 loss
ServerSend: rcv ack2
ServerSend: Send pkt12
ServerRecv: rcv pkt12
waitSeq: 11
ServerRecv: send ack10
ServerSend: rcv ack2
```

其效果展示如上图，在这里不再进行具体的原理分析。

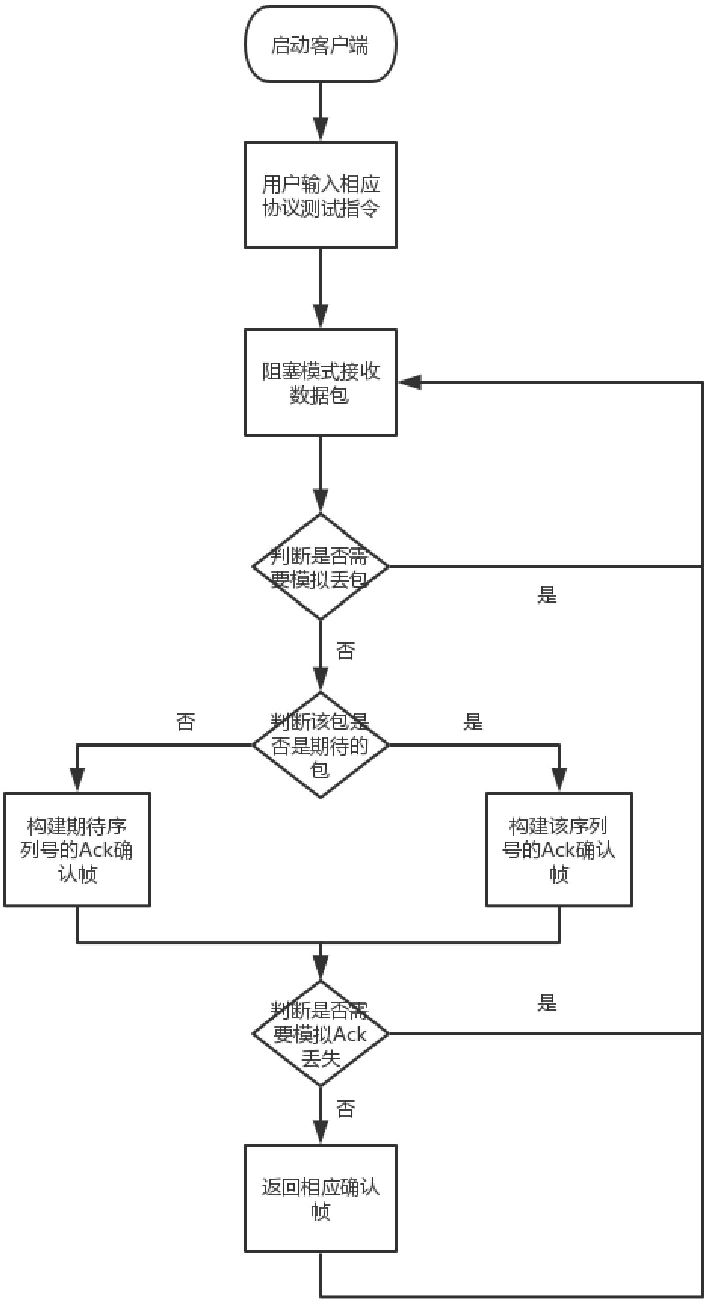
问题讨论：

对实验过程中的思考问题进行讨论或回答。

1) 发送端程序流程图


启动服务端
接收到客户端的协议测试命令
读取文件，进行分组数据包构建
判断当前窗口是否可以发送包
是
否
发送数据包，设置窗口内相应下标
非阻塞模式判断是否接收到确认帧
是
否
根据该Ack序列号进行窗口滑动
计时器加1，超时则触发重传事件

2) 接收端程序流程图



3) 协议典型交互过程

GBN：

接收方某一数据包丢失时，会重复发送期待的Ack序列号。发送方在该丢失包超时时，重传窗口内从该包开始的所有包。

SR：

接收方某一数据包丢失后，不发送该包Ack，会缓冲发送方发来的后续数据包。发送方在该丢失包超时时，仅重传该数据包。

心得体会：

结合实验过程和结果给出实验的体会和收获。

通过这次实验，自己对于GBN协议与SR协议有了更清晰的认识。一直认为自己这块掌握的还行，但是在真正代码实现时，才发现自己还是没有真正的理解其过程。

所幸这次实验帮助自己弥补了这个不足，尽管过程十分艰难，但是能够正确的实现这两个协议确实让自己受益匪浅。

不仅如此，在单向传输的基础上，成功实现GBN协议与SR协议的客户端与服务器的双向同时传输，增加了自己的自信心。

在这里，感谢老师的知识教导，也感谢助教的指点，感谢帮助我的人。