



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	可靠数据传输协议-GBN 协议的设计与实现					
姓名	董琦		院系	数据科学与大数据		
班级	2003501		学号	120L020701		
任课教师	刘亚维		指导教师	刘亚维		
实验地点	G001		实验时间	2022/10/13		
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. 报文结构

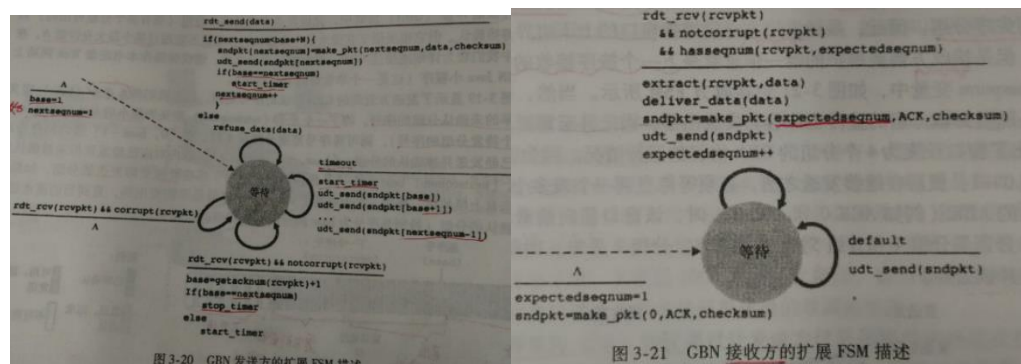
一个MSS共长1500B，第0B是本端发送报文的序号，第1B是确认对方发送报文的序号，2、3B合起来用以标志报文内容的长度，剩下的1496B为报文内容部分。当序号为-1时，代表发送文件结束。

由于考虑到后续扩展为双向通信的协议，故客户端和服务端的发送和接收报文格式一致，均为上述格式。

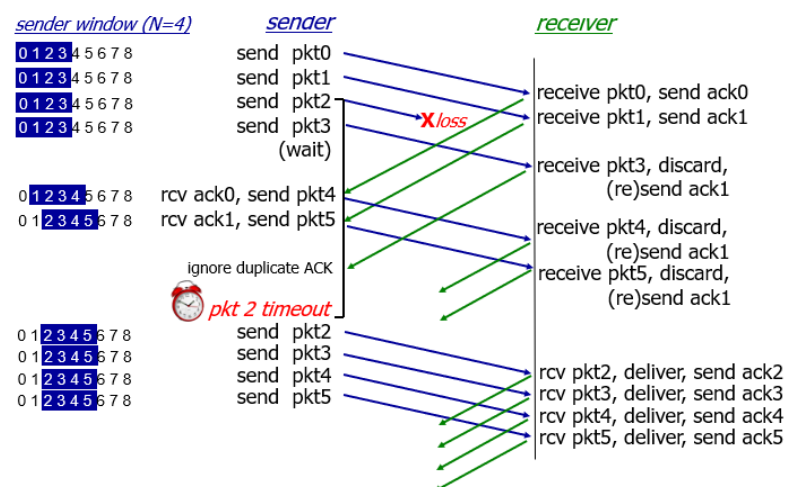
例：若本次发送的包在文件中序号为3，而且接收到对方序号为2的包，且此次包内容大小800B，则报文第0B内容为3，第1B内容为2，第2、3B合并内容为800，余下内容长度为800B，总长度为804B。

2. 报文超时与丢失的实现：在SW协议中已叙述，这里不再赘述。

3. GBN协议FSM的流程图



4. GBN协议的交互示例：



5. GBN协议的实现（写代码时构思不好，没做好模块化）

1) 服务器端:

各属性如下图:

```
private byte mySerial;
//服务器发送报文序号
9 个用法
private byte serial;
//服务器接收报文序号（双向传输）
11 个用法
private final int mss=1496;
//最大报文长度
5 个用法
private final byte size=8;
//窗口大小
9 个用法
private byte base;
//当前窗口的base
12 个用法
private Map<Byte,DatagramPacket> sendBufs;
//报文缓存
13 个用法
private DatagramSocket socket;
//监听端口
11 个用法
private Map<Byte,Timer> timers;
//各报文的定时器
8 个用法
private Map<Byte,Boolean> timeUp;
//标记报文超时
```

当启动一个传输文件任务时，初始化变量、开辟缓存:

```
private void initialize(){
    this.mySerial =0;
    this.serial=0;
    this.timers=new HashMap<>(size);
    this.sendBufs=new HashMap<>(size);
    this.timeUp=new HashMap<>( initialCapacity: 8);
    for(int i=0;i<size;i++){
        sendBufs.put((byte) i,new DatagramPacket(new byte[mss+4], length: mss+4));
    }
    base=0;
}
```

报文构建模块，当要发送新的报文时，执行此部分函数。

```

for(; mySerial < this.size; mySerial++){
    byte number= (byte) (mySerial +base);
    DatagramPacket sendPacket=this.sendBufs.get(number);
    byte[] sendBuf=sendPacket.getData();

    myLength = reader.read(sendBuf, off: 4, mss);

    sendBuf[0] = number;
    sendBuf[2] = (byte) (myLength & 0xFF);
    sendBuf[3] = (byte) (myLength >> 8 & 0xFF);
    //模拟发出后未收到，方法是不发送
    send(sendPacket,number);
    if (myLength<mss) {
        break;
    }
}

```

当文件发送完毕时，发送一个终止报文，序号设置为-1：

```

else if(this.sendBufs.isEmpty()){
    byte[] sendBuf=new byte[mss+4];
    DatagramPacket sendPacket=new DatagramPacket(sendBuf, length: mss+4);
    sendBuf[0]=-1;
    sendBuf[1]=serial;
    System.out.println("文件发送完毕。"+" 期望分组: "+(byte)(serial+(byte) 1));
    this.socket.send(sendPacket);
}

```

发送报文的函数，设置定时器等一系列属性：

```

private void send(DatagramPacket packet,byte number) throws IOException {
    packet.getData()[1]=serial;
    if(Math.random()>0.05) {
        this.socket.send(packet);
        System.out.println("发送分组: "+packet.getData()[0]+" 期望分组: "+(byte)(serial+(byte) 1));
    }else{
        System.out.println("丢失发送分组"+number);
    }

    Timer timer=new Timer( name: number +" packet timer");
    timer.schedule(new TimerTask(this.timeUp,number), delay: 1000);
    this.timeUp.put(number,false);
    this.timers.put(number,timer);
}

```

当接收到ACK报文时，检测其返回的序号是否等于当前的Base，若是，则说明成功接收，base+1，发送新的报文；若否，则判定为冗余分组，不做处理。

```

byte num=recv.getData()[1];
if (num == this.base) {
    if(mySerial!=-1)
        update(num);
    else delete(num);
    this.timers.get(num).cancel();
    this.timers.remove(num);
    ack = true;
} else if(num>=0) {
    System.out.println(num+"冗余分组,不做反应。base="+this.base);
    if(this.timers.containsKey(num)) {
        this.timers.get(num).cancel();
        this.timers.remove(num);
        Timer timer=new Timer( name: num + " packet timer");
        timer.schedule(new timerTask(this.timeUp,num), delay: 1000);
        this.timers.put(num,timer);
    }
    ack = false;
}

```

当接收到冗余分组时，若是已发送未确认的报文，则重启计时器。当接收到正确分组时，则关闭计时器。

实现文件双向传输，要对ACK报文的内容进行解析，步骤类似于停等协议（这里我从简写了）：

```

if(recv.getData()[0]==(byte)(serial+(byte) 1)){
    System.out.println("收到分组"(++serial));
    length = recv.getData()[2]& 0xFF;
    length |= recv.getData()[3] << 8;
    writer.write(recvBuf, off: 4,length);
}else if(recv.getData()[0]==-1){
    serial=-1;
    System.out.println("收到全部文件");
}

```

当有报文超时时，则执行全部重发。

检测是否有报文超时：

```

try {
    this.socket.receive(recv);
}catch (SocketTimeoutException exception){
    if(this.timeUp.containsValue(true)){
        resend();
    }
}

```

重发所有未确认的缓存报文：

```

public void resend() throws IOException {
    System.out.println("重新发送分组"+base);
    for(Map.Entry<Byte,Timer> entry:this.timers.entrySet()){
        entry.getValue().cancel();
    }
    this.timers.clear();
    for(int i=0;i<this.sendBufs.size();i++){
        send(this.sendBufs.get((byte)(i+base)), (byte) (i+base));
    }
}
}

```

定时器类的实现:

```

class timerTask extends TimerTask {
    2 个用法
    private Map<Byte,Boolean> timeUp;
    3 个用法
    private byte num;
    2 个用法  DQ
    public timerTask(Map<Byte,Boolean> timeUp,byte num){
        this.timeUp=timeUp;
        this.num=num;
    }
    DQ
    @Override
    public void run() {
        synchronized (this) {
            System.out.println(num+"超时");
            this.timeUp.put(num, true);
        }
    }
}
}

```

当双方文件传输完毕后，结束任务：

```

}
} while (recvBuf[1]!=-1||serial!=-1);

```

2) 客户端

客户端在收到一个服务器报文时，需检测其头部所带序号。若为期望的序号，则返回ACK并且将报文内容保存，同时更新序号；若不是期望的序号，则依然返回ACK而不保存报文，接收到报文什么序号，则返回的确认报文就是什么信号，交由服务器来判断执行对应动作。

向服务器传输文件的代码则类似SW协议，较为简单，不做赘述。

一次接收任务的参数初始化：

```

byte serial = 0;
byte myserial = 0;
int length;
int count=0;
int myLength;

```

对接收分组的内容解析：

```
if (serial != -1 & recvPacket.getData()[0] == serial) {
    System.out.println("收到正确的分组：" + serial + "    期望分组：" + recvPacket.getData()[1]);

    //收到正确分组，更新序号，保存数据
    serial++;
    count++;

    length = recvPacket.getData()[2] & 0xFF;
    length |= recvPacket.getData()[3] << 8;

    writer.write(recvBuf, off: 4, length);
}else if (recvPacket.getData()[0] == -1) {
    serial = -1;
    System.out.println("文件接收完毕，发送尚未完成。期望分组：" + recvPacket.getData()[1]);
}else if (recvPacket.getData()[0] >= 0) {
    System.out.println("收到冗余分组：" + recvPacket.getData()[0]);
}
```

构建发送的报文，若接收报文显示已经接收到最新的分组，则在发送报文内容中写入新的内容，若否则保持不变进行重发：

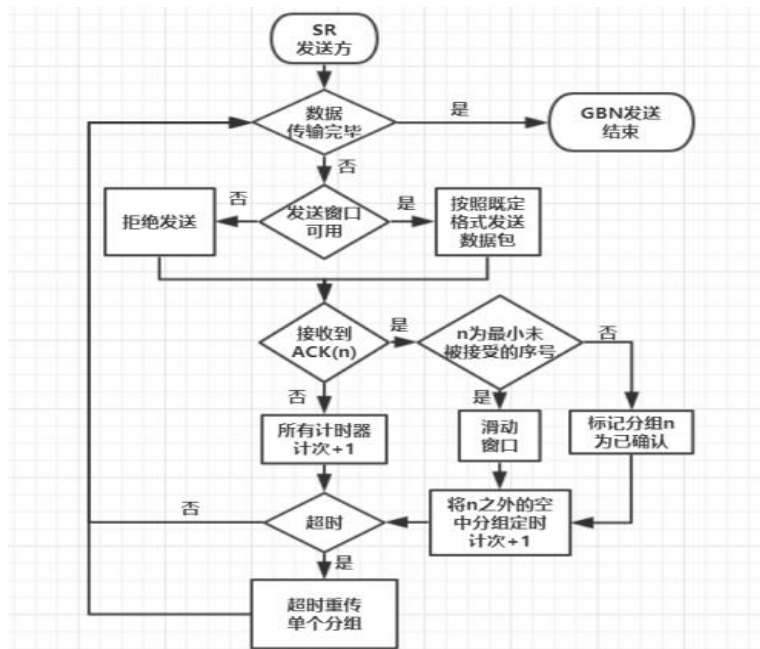
```
//构建发送报文
if (myserial != -1 & recvPacket.getData()[1] == myserial) {
    myLength = reader.read(sendBuf, off: 4, mss);
    System.out.println(myLength + "：" + myserial);
    if (myLength == -1) {
        myserial = -1;
        System.out.println("文件发送完毕");
    } else {
        myserial++;
        sendBuf[2] = (byte) (myLength & 0xFF);
        sendBuf[3] = (byte) (myLength >> 8 & 0xFF);
    }
    sendBuf[0] = myserial;
    sendBuf[1] = recvPacket.getData()[0];
}
```

当双方文件传输结束后，则完成任务：

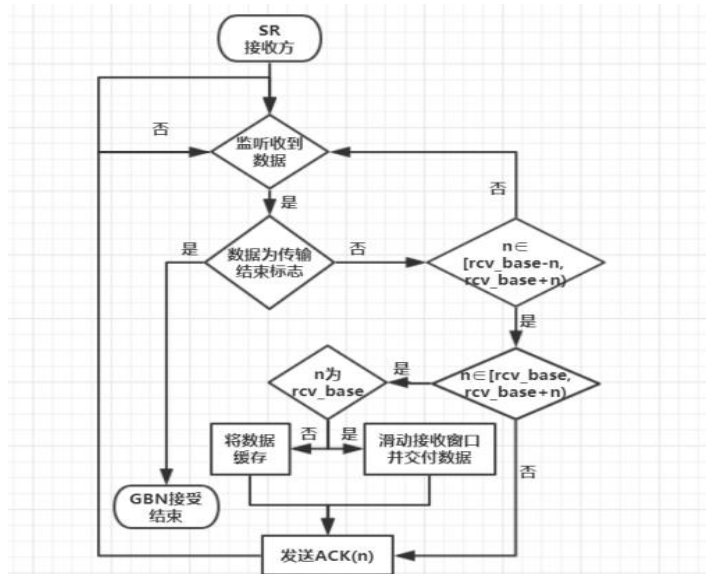
```
if (serial == -1 & recvPacket.getData()[1] == -1)
    break;
```

由于写代码时候双方的循环动作顺序不一样。。。所以客户端和服务端的代码结构不太一样，对称性不太好。

6. SR协议流程:



SR 发送方流程图



SR 接收方流程图

7. SR协议的实现

SR是在GBN协议上进行了更改所得的, 故这里只展示具体改动地方。

1) 服务器端:

重发操作: 只重发超时的报文, 而不是所有缓存报文

```

public void resend() throws IOException {
    System.out.println("重新发送分组"+base);
    for(Map.Entry<Byte, Boolean> entry: this.timeUp.entrySet()){
        if(entry.getValue()) {
            this.timers.get(entry.getKey()).cancel();
            this.timers.remove(entry.getKey());
            send(this.sendBufs.get(entry.getKey()), entry.getKey());
        }
    }
}
    
```


确认收到操作：对于在窗口内的序号报文，收到ACK时则标记已经确认接收。若等于Base，则进行窗口滑动，若不等于，则窗口不变。

```
if (num == base) {
    ack = true;
    updateBase();
    System.out.println(num+"分组成功确认接收. base滑动至: "+base);
} else if(num>base&&num<base+size){
    System.out.println(num+"分组成功确认接收.base不变: "+base);
    ack=false;
}
```

此外还有一些辅助这两个功能的代码，这里就不做赘述了，都是比较简单的逻辑。

2) 客户端

客户端新增了报文缓存，缓存大小等于客户端的窗口大小：

```
private byte base;
3 个用法
private final byte size=8;
5 个用法
private Map<Byte,DatagramPacket> recvBufs;
1 个用法
private Set<Byte> Received;
```

当接受到窗口范围内的报文时，确认接收并且缓存。当报文序号等于base时，则按序号将已经缓存的连续报文保存到磁盘，并且更新序号；而在其他情况下，则单纯返回一个ACK，表示时一个冗余分组：

```
if (received>=base&&received<base+size) {
    System.out.println("收到正确的分组："+ received+" 期望分组: "+recvPacket.getData()[1]);
    recvBufs.put(received,recvPacket);
    if(received==base){
        while(recvBufs.containsKey(base)){
            DatagramPacket packet=recvBufs.get(base);
            length = packet.getData()[2]& 0xFF;
            length |= packet.getData()[3] << 8;
            writer.write(packet.getData(), off: 4, length);
            System.out.println("写入分组: "+base);
            recvBufs.remove(received);
            base++;
            count++;
        }
    }
} else if(received==-1){
    System.out.println("文件接收完毕，发送尚未完成。期望分组: "+recvPacket.getData()[1]);
} else if(received<base&&received>=base-size){
    System.out.println("收到冗余分组："+recvPacket.getData()[0]);
}
```

这样就完成了所有SR协议的改动。更多细节请见代码。

实验结果:

1. GBN协议:

一次文件传输任务中服务器和客户端的输出。由于文件较大，引入丢包、超时等机制，所以流程较为复杂，故不详细解释，只展示大概的输出。具体请助教可以自己运行然后分析一下过程。

```

GBN.Server x
C:\Users\DQ\.jdk\corretto-1.8.0_342\bin>
丢失发送分组0
发送分组: 1 期望分组: 1
发送分组: 2 期望分组: 1
发送分组: 3 期望分组: 1
发送分组: 4 期望分组: 1
发送分组: 5 期望分组: 1
发送分组: 6 期望分组: 1
发送分组: 7 期望分组: 1
1冗余分组, 不做反应。base=0
收到分组1
2冗余分组, 不做反应。base=0
3冗余分组, 不做反应。base=0
4冗余分组, 不做反应。base=0
5冗余分组, 不做反应。base=0
6冗余分组, 不做反应。base=0
7冗余分组, 不做反应。base=0
7冗余分组, 不做反应。base=0
1超时
3超时
5超时
4超时
7冗余分组, 不做反应。base=0
2超时
6超时
0超时
重新发送分组0
发送分组: 0 期望分组: 2
发送分组: 1 期望分组: 2
发送分组: 2 期望分组: 2
发送分组: 3 期望分组: 2
发送分组: 4 期望分组: 2
发送分组: 5 期望分组: 2
发送分组: 6 期望分组: 2
发送分组: 7 期望分组: 2
7冗余分组, 不做反应。base=0
0分组成功确认接收
{1=java.net.DatagramPacket@60e53b93, 2
收到分组2
发送分组: 8 期望分组: 3

GBN.Client x
C:\Users\DQ\.jdk\corretto-1.8.0_342\bin>
2022-10-16 :11:59:04
-quit
Good bye!
收到冗余分组:1
1496:0
成功发送ACK1和分组1
收到冗余分组:2
成功发送ACK2和分组1
收到冗余分组:3
成功发送ACK3和分组1
收到冗余分组:4
成功发送ACK4和分组1
收到冗余分组:5
成功发送ACK5和分组1
收到冗余分组:6
成功发送ACK6和分组1
收到冗余分组:7
成功发送ACK7和分组1
收到正确的分组:0 期望分组: 1
1496:1
成功发送ACK0和分组2
收到正确的分组:1 期望分组: 1
成功发送ACK1和分组2
收到正确的分组:2 期望分组: 1
成功发送ACK2和分组2
收到正确的分组:3 期望分组: 1
成功发送ACK3和分组2
网络发生延迟
收到正确的分组:4 期望分组: 1
成功发送ACK4和分组2
收到正确的分组:5 期望分组: 1
成功发送ACK5和分组2
收到正确的分组:6 期望分组: 1
成功发送ACK6和分组2
收到正确的分组:7 期望分组: 1
成功发送ACK7和分组2
收到正确的分组:8 期望分组: 2
1496:2
成功发送ACK8和分组3
  
```

GBN.Server	GBN.Client
发送分组: 9 期望分组: 3	成功发送ACK8和分组3
2分组成功确认接收	收到正确的分组:9 期望分组: 2
{3=java.net.DatagramPacket@1d44bcfa, 4	成功发送ACK9和分组3
发送分组: 10 期望分组: 3	收到正确的分组:10 期望分组: 2
3分组成功确认接收	成功发送ACK10和分组3
{4=java.net.DatagramPacket@266474c2, 5	收到正确的分组:11 期望分组: 2
发送分组: 11 期望分组: 3	成功发送ACK11和分组3
3冗余分组, 不做反应。base=4	收到冗余分组:13
4分组成功确认接收	成功发送ACK13和分组3
{5=java.net.DatagramPacket@6f94fa3e, 6	收到冗余分组:14
丢失发送分组12	成功发送ACK14和分组3
5分组成功确认接收	收到冗余分组:15
{6=java.net.DatagramPacket@5e481248, 7	成功发送ACK15和分组3
发送分组: 13 期望分组: 3	收到冗余分组:16
6分组成功确认接收	1496:3
{7=java.net.DatagramPacket@66d3c617, 8	成功发送ACK16和分组4
发送分组: 14 期望分组: 3	收到冗余分组:17
7分组成功确认接收	成功发送ACK17和分组4
{8=java.net.DatagramPacket@63947c6b, 9	收到冗余分组:18
发送分组: 15 期望分组: 3	成功发送ACK18和分组4
8分组成功确认接收	收到冗余分组:19
{16=java.net.DatagramPacket@63947c6b,	成功发送ACK19和分组4
收到分组3	收到正确的分组:12 期望分组: 4
发送分组: 16 期望分组: 4	1496:4
9分组成功确认接收	返回的包丢失
{16=java.net.DatagramPacket@63947c6b,	收到正确的分组:13 期望分组: 4
发送分组: 17 期望分组: 4	成功发送ACK13和分组5
10分组成功确认接收	网络发生延迟
{16=java.net.DatagramPacket@63947c6b,	收到正确的分组:14 期望分组: 4
发送分组: 18 期望分组: 4	成功发送ACK14和分组5
11分组成功确认接收	收到正确的分组:15 期望分组: 4
{16=java.net.DatagramPacket@63947c6b,	成功发送ACK15和分组5
发送分组: 19 期望分组: 4	收到正确的分组:16 期望分组: 4
13冗余分组, 不做反应。base=12	成功发送ACK16和分组5
14冗余分组, 不做反应。base=12	收到正确的分组:17 期望分组: 4
15冗余分组, 不做反应。base=12	成功发送ACK17和分组5
16冗余分组, 不做反应。base=12	收到正确的分组:18 期望分组: 4
收到分组4	成功发送ACK18和分组5
17冗余分组, 不做反应。base=12	收到冗余分组:12
18冗余分组, 不做反应。base=12	1496:5

GBN.Server

```

22超时
19超时
20超时
18超时
重新发送分组16
发送分组: 16 期望分组: 10
发送分组: 17 期望分组: 10
发送分组: 18 期望分组: 10
发送分组: 19 期望分组: 10
发送分组: 20 期望分组: 10
发送分组: 21 期望分组: 10
丢失发送分组22
发送分组: 23 期望分组: 10
23冗余分组,不做反应。base=16
16分组成功确认接收
{17=java.net.DatagramPacket@60e53b93,
收到分组10
17分组成功确认接收
{18=java.net.DatagramPacket@5e2de80c,
18分组成功确认接收
{19=java.net.DatagramPacket@1d44bcfa,
19分组成功确认接收
{20=java.net.DatagramPacket@266474c2,
20分组成功确认接收
{21=java.net.DatagramPacket@6f94fa3e,
21分组成功确认接收
{22=java.net.DatagramPacket@5e481248,
23冗余分组,不做反应。base=22
23冗余分组,不做反应。base=22
22超时
23冗余分组,不做反应。base=22
重新发送分组22
发送分组: 22 期望分组: 11
发送分组: 23 期望分组: 11
23冗余分组,不做反应。base=22
22分组成功确认接收
{23=java.net.DatagramPacket@66d3c617}
收到分组11
23分组成功确认接收
收到分组27
文件发送完毕。 期望分组: 28
文件发送完毕。 期望分组: 28
收到全部文件
传输文件结束
                
```

GBN.Client

```

成功发送ACK-1和分组16
文件接收完毕,发送尚未完成。期望分组: 16
1496:16
成功发送ACK-1和分组17
文件接收完毕,发送尚未完成。期望分组: 17
1496:17
成功发送ACK-1和分组18
文件接收完毕,发送尚未完成。期望分组: 18
1496:18
成功发送ACK-1和分组19
文件接收完毕,发送尚未完成。期望分组: 19
1496:19
成功发送ACK-1和分组20
文件接收完毕,发送尚未完成。期望分组: 20
1496:20
成功发送ACK-1和分组21
文件接收完毕,发送尚未完成。期望分组: 21
1496:21
成功发送ACK-1和分组22
网络发生延迟
文件接收完毕,发送尚未完成。期望分组: 22
1496:22
成功发送ACK-1和分组23
文件接收完毕,发送尚未完成。期望分组: 22
成功发送ACK-1和分组23
文件接收完毕,发送尚未完成。期望分组: 23
1496:23
成功发送ACK-1和分组24
文件接收完毕,发送尚未完成。期望分组: 23
成功发送ACK-1和分组24
文件接收完毕,发送尚未完成。期望分组: 24
1496:24
成功发送ACK-1和分组25
文件接收完毕,发送尚未完成。期望分组: 24
成功发送ACK-1和分组25
文件接收完毕,发送尚未完成。期望分组: 25
1496:25
成功发送ACK-1和分组26
文件接收完毕,发送尚未完成。期望分组: 25
成功发送ACK-1和分组26
成功发送ACK-1和分组-1
文件接收完毕,发送尚未完成。期望分组: -1
共收到: 24个包

进程已结束,退出代码0
                
```

最后可以看到文件双向传输成功实现:

GNB_ClientResult.txt

GNB_ClientTest.txt

GNB_ServerResult.txt

GNB_ServerTest.txt

从上面例子中可以看到分组丢失、网络延迟、确认报文丢失等情况，而服务器也相应地做出了重发的操作。

```





丢失发送分组0
发送分组: 1 期望分组: 1
发送分组: 2 期望分组: 1
发送分组: 3 期望分组: 1
发送分组: 4 期望分组: 1
发送分组: 5 期望分组: 1
发送分组: 6 期望分组: 1
发送分组: 7 期望分组: 1
1冗余分组, 不做反应。base=0
收到分组1
2冗余分组, 不做反应。base=0
3冗余分组, 不做反应。base=0
4冗余分组, 不做反应。base=0
5冗余分组, 不做反应。base=0
6冗余分组, 不做反应。base=0
7冗余分组, 不做反应。base=0
1超时
3超时
5超时
4超时
7冗余分组, 不做反应。base=0
2超时
6超时
0超时
重新发送分组0
收到正确的分组:12 期望分组: 4
1496:4
返回的包丢失
收到正确的分组:13 期望分组: 4 等等。
    
```

2. SR协议:

SR.Server	SR.Client
发送分组: 0 期望分组: 1	收到正确的分组:0 期望分组: 0
发送分组: 1 期望分组: 1	写入分组: 0
发送分组: 2 期望分组: 1	1496:0
发送分组: 3 期望分组: 1	成功发送ACK0和分组1
发送分组: 4 期望分组: 1	收到正确的分组:1 期望分组: 0
发送分组: 5 期望分组: 1	写入分组: 1
发送分组: 6 期望分组: 1	成功发送ACK1和分组1
发送分组: 7 期望分组: 1	网络发生延迟
接收分组序号0	收到正确的分组: 2 期望分组: 0
notReceived:[0, 1, 2, 3, 4, 5, 6, 7]	写入分组: 2
0分组成功确认接收。base滑动至: 1	返回的包丢失
{1=java.net.DatagramPacket@60e53b93, 2=java.net.DatagramPacket@60e53b93}	收到正确的分组:3 期望分组: 0
收到分组1	写入分组: 3
发送分组: 8 期望分组: 2	成功发送ACK3和分组1
接收分组序号1	收到正确的分组:4 期望分组: 0
notReceived:[1, 2, 3, 4, 5, 6, 7, 8]	写入分组: 4
1分组成功确认接收。base滑动至: 2	成功发送ACK4和分组1
{2=java.net.DatagramPacket@5e2de80c, 3=java.net.DatagramPacket@5e2de80c}	收到正确的分组:5 期望分组: 0
发送分组: 9 期望分组: 2	写入分组: 5
接收分组序号3	成功发送ACK5和分组1
notReceived:[2, 3, 4, 5, 6, 7, 8, 9]	收到正确的分组:6 期望分组: 0
3分组成功确认接收。base不变: 2	写入分组: 6
{2=java.net.DatagramPacket@5e2de80c, 4=java.net.DatagramPacket@5e2de80c}	成功发送ACK6和分组1
接收分组序号4	收到正确的分组:7 期望分组: 0
notReceived:[2, 4, 5, 6, 7, 8, 9]	写入分组: 7
4分组成功确认接收。base不变: 2	成功发送ACK7和分组1
{2=java.net.DatagramPacket@5e2de80c, 5=java.net.DatagramPacket@5e2de80c}	收到正确的分组:8 期望分组: 1
接收分组序号5	写入分组: 8
	1496:1

SR.Server	SR.Client
{16=java.net.DatagramPacket@63947c6b, 17=java.net.Da	写入分组: 8
2超时	1496:1
重新发送分组2	成功发送ACK8和分组2
发送分组: 2 期望分组: 3	收到正确的分组:9 期望分组: 1
接收分组序号9	写入分组: 9
9冗余分组,不做反应。base不变: 2	成功发送ACK9和分组2
{16=java.net.DatagramPacket@63947c6b, 17=java.net.Da	收到冗余分组:2
接收分组序号2	1496:2
notReceived:[2]	成功发送ACK2和分组3
2分组成功确认接收。base滑动至: 10	收到正确的分组:10 期望分组: 3
{16=java.net.DatagramPacket@63947c6b, 17=java.net.Da	写入分组: 10
收到分组3	1496:3
发送分组: 10 期望分组: 4	成功发送ACK10和分组4
发送分组: 11 期望分组: 4	收到正确的分组:11 期望分组: 3
发送分组: 12 期望分组: 4	写入分组: 11
发送分组: 13 期望分组: 4	成功发送ACK11和分组4
发送分组: 14 期望分组: 4	收到正确的分组:12 期望分组: 3
发送分组: 15 期望分组: 4	写入分组: 12
发送分组: 16 期望分组: 4	成功发送ACK12和分组4
发送分组: 17 期望分组: 4	收到正确的分组:13 期望分组: 3
接收分组序号10	写入分组: 13
notReceived:[16, 17, 10, 11, 12, 13, 14, 15]	成功发送ACK13和分组4
10分组成功确认接收。base滑动至: 11	收到正确的分组:14 期望分组: 3
{16=java.net.DatagramPacket@63947c6b, 17=java.net.Da	写入分组: 14
收到分组4	成功发送ACK14和分组4
发送分组: 18 期望分组: 5	收到正确的分组:15 期望分组: 3
接收分组序号11	写入分组: 15
notReceived:[16, 17, 18, 11, 12, 13, 14, 15]	成功发送ACK15和分组4
11分组成功确认接收。base滑动至: 12	收到正确的分组:16 期望分组: 3
{16=java.net.DatagramPacket@63947c6b, 17=java.net.Da	写入分组: 16
发送分组: 19 期望分组: 5	成功发送ACK16和分组4
接收分组序号12	收到正确的分组:17 期望分组: 3
notReceived:[16, 17, 18, 19, 12, 13, 14, 15]	写入分组: 17
12分组成功确认接收。base滑动至: 13	成功发送ACK17和分组4
{16=java.net.DatagramPacket@63947c6b, 17=java.net.Da	收到正确的分组:18 期望分组: 4
发送分组: 20 期望分组: 5	写入分组: 18
接收分组序号13	1496:4
notReceived:[16, 17, 18, 19, 20, 13, 14, 15]	成功发送ACK18和分组5
13分组成功确认接收。base滑动至: 14	收到正确的分组:19 期望分组: 4
{16=java.net.DatagramPacket@63947c6b, 17=java.net.Da	写入分组: 19
收到分组20	
文件发送完毕。期望分组: 21	文件发送完毕
接收分组序号-1	成功发送ACK-1和分组-1
{}	文件接收完毕,发送尚未完成。期望分组: -1
收到全部文件	共收到: 27个包
传输文件结束	进程已结束,退出代码0

一次传输文件的部分控制台输出，可以看到服务端当前的窗口base位置、尚未确认接收的报文序号集合、窗口滑动变化、期望客户端传输文件报文序号等。也可以看到客户端成功接收报文后的处理，包括写入、base滑动等。最后可以验证，成功完成了文件的双向传输。

 SR_ClientResult.txt
 SR_ClientTest.txt
 SR_ServerResult.txt
 SR_ServerTest.txt

问题讨论：

Q: 相同条件下，GBN协议和SR协议哪个更快一些？

A: SR协议更快一些，SR协议的接收方可以将乱序到达的数据序列缓存起来，直到顺序正确即可向上层交付，而GBN协议要重传base及以后的一系列分组，浪费了大量的时间。

心得体会：

(1) 掌握了滑动窗口协议的基本原理；并通过具体的实践感受到了滑动窗口协议在 GBN 与 SR 中的体现；

(2) 掌握了基于 UDP 设计并实现一个 GBN 协议的过程与技术，同时也体会到了如何实现模拟上层数据到来以及模拟丢包事件的产生；

(3) 掌握了基于 UDP 设计并实现一个 SR 协议的过程与技术。