计算机网络大作业报告

~~~~~

1. 结合代码和 LOG 文件分析针对每个项目举例说明解决效果。（16 分）
2. 未完全完成的项目，说明完成中遇到的关键困难，以及可能的解决方式。（2 分）
3. 说明在实验过程中采用迭代开发的优点或问题。(优点或问题合理：1 分)
4. 总结完成大作业过程中已经解决的主要问题和自己采取的相应解决方法(1 分)
5. 对于实验系统提出问题或建议(1 分)

目录
~~~~~

RDT1.0

RDT1.0 是在非常可靠的信道上实现可靠数据传输的协议，它假设底层信道不会出现位错和数据丢失。所以我们设置 eFlag 为 0 表示信道完全可靠。

```
2 个用法
@Override
//不可靠发送：将打包好的TCP数据报通过不可靠传输信道发送；仅需修改错误标志
public void udt_send(TCP_PACKET stcpPack) {
    //设置错误控制标志
    tcpH.setTh_eflag((byte) 0);
    //System.out.println("to send: "+stcpPack.getTcpH().getTh_seq());
    //发送数据报
    client.send(stcpPack);
}
```

通过查看日志，接收方和发送方的传输成功率均为 100%。

发送方：

```
CLIENT HOST TOTAL   SUC_RATIO   NORMAL  WRONG   LOSS    DELAY
172.27.176.1:9001   1000    100.00% 1000    0   0   0
    2024-01-02 11:54:31:731 CST DATA_seq: 1     ACKed
    2024-01-02 11:54:31:757 CST DATA_seq: 101       ACKed
    2024-01-02 11:54:31:769 CST DATA_seq: 201       ACKed
    2024-01-02 11:54:31:785 CST DATA_seq: 301       ACKed
    2024-01-02 11:54:31:802 CST DATA_seq: 401       ACKed
```

接收方：

```
172.27.176.1:9002   1000    100.00% 1000    0   0   0
    2024-01-02 11:54:31:737 CST ACK_ack: 1
    2024-01-02 11:54:31:757 CST ACK_ack: 101
    2024-01-02 11:54:31:769 CST ACK_ack: 201
    2024-01-02 11:54:31:790 CST ACK_ack: 301
    2024-01-02 11:54:31:802 CST ACK_ack: 401
    2024-01-02 11:54:31:818 CST ACK_ack: 501
    2024-01-02 11:54:31:837 CST ACK_ack: 601
```

RDT2.0

RDT2.0 是在可能出现位错的信道上进行可靠数据传输的协议，它在 RDT1.0 的基础上增加了错误检测机制。我们使用校验码 checkSum 来完成。使用 CRC 进行校验，在代码中，使用 CRC32 完成。

```java
/*计算TCP报文段校验和：只需校验TCP首部中的seq、ack和sum，以及TCP数据字段*/
5 个用法
public static short computeChkSum(TCP_PACKET tcpPack) {
    CRC32 crc32 = new CRC32();
    TCP_HEADER header = tcpPack.getTcpH();  // 获取原TCP报文头部
    crc32.update(header.getTh_seq());  // 添加seq进行校验
    crc32.update(header.getTh_ack());  // 添加ack进行校验
    // 添加 TCP 数据字段进行校验
    for (int i = 0; i < tcpPack.getTcpS().getData().length; i++) {
        crc32.update(tcpPack.getTcpS().getData()[i]);
    }
    // 获取校验码，并返回
    return (short) crc32.getValue();
}
```

发送方需要计算校验码，将校验码附在数据后，进行发送，然后监听接收方发送的信号，如果是 NAK 则重发该包。

处理 ack 包，如果是对当前发送包的确认，则从缓存中清楚，否则重发。NCK 的 ack 值为-1，所以不可能是对当前发送包的确认【即 ack!=当前发送包的 seq】，所以也需要重发。

```java
@Override
//需要修改
public void waitACK() {
    //循环检查ackQueue
    //循环检查确认号对列中是否有新收到的ACK
    if (!ackQueue.isEmpty()) {
        int currentAck = ackQueue.poll();
        // System.out.println("CurrentAck: "+currentAck);
        if (currentAck == tcpPack.getTcpH().getTh_seq()) {
            System.out.println("Clear: " + tcpPack.getTcpH().getTh_seq());
            flag = 1;
            //break;
        } else { // NAK
            System.out.println("Retransmit: " + tcpPack.getTcpH().getTh_seq());
            udt_send(tcpPack);
            flag = 0;
        }
    }
}
```

接收方需要将 checkSum 重新计算一遍，如果和发送来的校验码相同则发送 ACK，否则发送 NAK（代码中我们将 ack 设置为-1 实现）。

```java
@Override
//接收到数据报：检查校验和，设置回复的ACK报文段
public void rdt_recv(TCP_PACKET recvPack) {
    //检查校验码，生成ACK
    if(CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
        //生成ACK报文段（设置确认号）
        tcpH.setTh_ack(recvPack.getTcpH().getTh_seq());
        ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());
        tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));
        //回复ACK报文段
        reply(ackPack);

        //将接收到的正确有序的数据插入data队列，准备交付
        dataQueue.add(recvPack.getTcpS().getData());
        sequence++;
    }else{
        System.out.println("Recieve Computed: "+CheckSum.computeChkSum(recvPack));
        System.out.println("Recieved Packet"+recvPack.getTcpH().getTh_sum());
        System.out.println("Problem: Packet Number: "+recvPack.getTcpH().getTh_seq()+" + InnerSeq:  "+sequence);
        tcpH.setTh_ack(-1);
        ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());
        tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));
        //回复ACK报文段
        reply(ackPack);
    }
```

为了验证我们实现了在可能出现位错的信道上进行可靠数据传输，我们需要将信道设置为可能出现位错，即将 eFlag 设置为 1。

```java
@Override
//不可靠发送：将打包好的TCP数据报通过不可靠传输信道发送；仅需修改错误标志
public void udt_send(TCP_PACKET stcpPack) {
    //设置错误控制标志
    tcpH.setTh_eflag((byte) 1);
    //System.out.println("to send: "+stcpPack.getTcpH().getTh_seq());
    //发送数据报
    client.send(stcpPack);
}
```

同时需要让 ACK/NAK 包发生位错的可能。

```java
@Override
//回复ACK报文段
public void reply(TCP_PACKET replyPack) {
    //设置错误控制标志
    tcpH.setTh_eflag((byte)1);  //eFlag=0，信道无错误

    //发送数据报
    client.send(replyPack);
}
```

查看日志：

```
2024-01-03 14:38:22:273 CST DATA_seq: 25201     ACKed
2024-01-03 14:38:22:287 CST DATA_seq: 25301     ACKed
2024-01-03 14:38:22:300 CST DATA_seq: 25401 WRONG   NO_ACK
2024-01-03 14:38:22:301 CST *Re: DATA_seq: 25401        ACKed
2024-01-03 14:38:22:313 CST DATA_seq: 25501     NO_ACK
2024-01-03 14:38:22:314 CST *Re: DATA_seq: 25501        ACKed
```

可以发现 25401 号包发生位错，没有 ACK，收到了 NAK，重发了当前数据包。

```
2024-01-03 14:38:22:288 CST ACK_ack: 25301
2024-01-03 14:38:22:301 CST ACK_ack: -1          NAK
2024-01-03 14:38:22:301 CST ACK_ack: 25401
2024-01-03 14:38:22:314 CST ACK_ack: -1733765829    WRONG
2024-01-03 14:38:22:315 CST ACK_ack: 25501
```

查看接收方日志，可以发现是发送了一个 ACK 为-1 的包，即 NAK。在重传后，对 25401 包进行 ACK 了。

同时可以发现，RDT2.0 有个非常严重的问题，无法对错误的 ACK 包进行处理，但由于代码中逻辑为当前 ack != seq 值则重发，最终还是重发了该包。

```
if (!ackQueue.isEmpty()) {
    int currentAck = ackQueue.poll();
    // System.out.println("CurrentAck: "+currentAck);å
    if (currentAck == tcpPack.getTcpH().getTh_seq()) {
        System.out.println("Clear: " + tcpPack.getTcpH().getTh_seq());
        flag = 1;
        //break;
    } else { // NAK 或 对其他包的ACK
        System.out.println("Retransmit: " + tcpPack.getTcpH().getTh_seq());
        udt_send(tcpPack);
        flag = 0;
    }
}
```

RDT2.1

RDT2.1 在 RDT2.0 的基础上考虑了 ACK 和 NAK 出错的情况，针对这个情况进行了处理。

发送方在接受到 ACK/NAK 包后，需要先校验 ACK/NAK 包是否发生位错，如果 ACK/NAK 包发生位错，则应让发送方重传，我这里使用向 ACK 队列中添加一个-1 来完成（等价于收到了一个 NAK 包）。

发送发如果收到重复的 ACK 同样交由 waitACK 进行处理，如果不是对当前包的 ack 确认就重发数据包，知道得到对当前包的确认。

```java
@Override
//接收到ACK报文：检查校验和，将确认号插入ack队列;NACK的确认号为－1；不需要修改
public void recv(TCP_PACKET recvPack) {
    if(CheckSum.computeChkSum(recvPack) ==  recvPack.getTcpH().getTh_sum()){
        System.out.println("Receive ACK Number:  " + recvPack.getTcpH().getTh_ack());
        ackQueue.add(recvPack.getTcpH().getTh_ack());
    }else{
        System.out.println("Receive corrupt ACK:  " + recvPack.getTcpH().getTh_ack());
        ackQueue.add(-1);
    }
    System.out.println();

    //处理ACK报文
    waitACK();
}
```

作为接受方则需要考虑重复包的问题，如果为重复包则不应该重复交付数据，应只返回 ack。

为了解决这个问题，我们需要记录上一个包的序号。观察源代码可以发现，sequence 记录的是当前待接受的是第几个包，而不是包中的 seq。

```
2 个用法
int sequence=1;//用于记录当前待接收的包序号，注意包序号不完全是
```

通过观察 Log 文件，可以发现一个包的大小为 100，所以不难得出

(seq - 1) / 100 = sequence - 1

```java
@Override
//接收到数据报：检查校验和，设置回复的ACK报文段
public void rdt_recv(TCP_PACKET recvPack) {
    //检查校验码，生成ACK
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
        //生成ACK报文段（设置确认号）
        tcpH.setTh_ack(recvPack.getTcpH().getTh_seq());
        ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());
        tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));
        //回复ACK报文段
        reply(ackPack);
        int seq = (recvPack.getTcpH().getTh_seq() - 1) / 100;
        if (seq != (sequence - 1)) {
            //将接收到的正确有序的数据插入data队列，准备交付
            dataQueue.add(recvPack.getTcpS().getData());
            sequence++;
        }
```

查看日志：

```
2024-01-02 21:42:53:972 CST ACK_ack: 13901
2024-01-02 21:42:53:985 CST ACK_ack: 14001
2024-01-02 21:42:53:999 CST ACK_ack: -423114388 WRONG
2024-01-02 21:42:54:000 CST ACK_ack: 14101
2024-01-02 21:42:54:014 CST ACK_ack: 14201
```

我们发现在第一次 ACK14101 时发生了位错，观察发送方日志，可以发现第一次没有 ACK，并且重发了 14101 号数据包，在收到接受方重发的 ACK14101 包后，完成了 ACK。

```
2024-01-02 21:42:53:985 CST DATA_seq: 14001     ACKed
2024-01-02 21:42:53:998 CST DATA_seq: 14101     NO_ACK
2024-01-02 21:42:54:000 CST *Re: DATA_seq: 14101        ACKed
2024-01-02 21:42:54:013 CST DATA_seq: 14201     ACKed
2024-01-02 21:42:54:026 CST DATA_seq: 14301     ACKed
```

RDT2.2

RDT2.2 在 RDT2.1 的基础上优化掉了 NAK，只使用 ACK 完成了该功能。即接受方收到了位错的数据包，不再发送 NAK 包，改为发送对上一个包的 ACK。发送方收到的是对上一个包的 ACK，则重新发送当前数据包。

```
} else {
    System.out.println("Receive Computed: " + CheckSum.computeChkSum(recvPack));
    System.out.println("Received Packet" + recvPack.getTcpH().getTh_sum());
    System.out.println("Problem: Packet Number: " + recvPack.getTcpH().getTh_seq() + " + InnerSeq:  " + sequence);
    tcpH.setTh_ack((sequence - 1) * 100 + 1);
    ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());
    tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));
    //回复ACK报文段
    reply(ackPack);
}
```

接受方修改如上图所示。

```
if (currentAck == tcpPack.getTcpH().getTh_seq()) {
    System.out.println("Clear: " + tcpPack.getTcpH().getTh_seq());
    flag = 1;
    //break;
} else { // NAK 或 ACK上一个包
    System.out.println("Retransmit: " + tcpPack.getTcpH().getTh_seq());
    udt_send(tcpPack);
    flag = 0;
}
```

发送方相关代码如上图所示。

查看日志：

```
2024-01-02 22:16:05:049 CST DATA_seq: 24201     ACKed
2024-01-02 22:16:05:061 CST DATA_seq: 24301     ACKed
2024-01-02 22:16:05:074 CST DATA_seq: 24401 WRONG   NO_ACK
2024-01-02 22:16:05:074 CST *Re: DATA_seq: 24401        ACKed
2024-01-02 22:16:05:087 CST DATA_seq: 24501     ACKed
2024-01-02 22:16:05:100 CST DATA_seq: 24601     ACKed
```

发送方在发送 24401 号包时，发生了位错。

```
2024-01-02 22:16:05:049 CST ACK_ack: 24201
2024-01-02 22:16:05:061 CST ACK_ack: 24301
2024-01-02 22:16:05:074 CST ACK_ack: 24301
2024-01-02 22:16:05:075 CST ACK_ack: 24401
2024-01-02 22:16:05:087 CST ACK_ack: 24501
2024-01-02 22:16:05:101 CST ACK_ack: 24601
```

查看接收方日志，第一次接受 24401 包发现位错，发送了对上一个包 24301 的确认，发送方收到对上一个包的第二次 ACK，重发当前包，接收方第二次正确接收，发送了对当前包 24401 包的确认。

RDT3.0

RDT3.0 在 RDT2.2 的基础上考虑了信道可能丢包的问题，为了解决该问题，RDT3.0 采用了超时计时器解决该问题，如果到了超时计时器所设置的重传时间而发送方仍收不到接收方的任何 ACK 或 NAK,则重传原来的数据分组。也就是超时重传。

```
private UDT_Timer timer; // 超时计时器

private UDT_RetransTask retransTask; // 超时重传任务
```

除此之外，还需要确定超时重传时间 RTO，我们观察 Log 文件，可以发现由于是发送方和接收方都在本地，RTT 基本上在十几 ms 左右，所以我们设置 RTO 的值为 100ms。同时设置 flag 为 0，使用 while 循环一直等到 flag 为 1，来阻塞应用层调用。

```
//可靠发送（应用层调用）：封装应用层数据，产生TCP数据报；需要修改
public void rdt_send(int dataIndex, int[] appData) {

    //生成TCP数据报（设置序号和数据字段/校验和),注意打包的顺序
    tcpH.setTh_seq(dataIndex * appData.length + 1);//包序号设置为字节流号:
    tcpS.setData(appData);
    tcpPack = new TCP_PACKET(tcpH, tcpS, destinAddr);
    //更新带有checksum的TCP 报文头
    tcpH.setTh_sum(CheckSum.computeChkSum(tcpPack));
    tcpPack.setTcpH(tcpH);

    timer = new UDT_Timer();
    retransTask = new UDT_RetransTask(client, tcpPack);
    timer.schedule(retransTask, delay: 100, period: 100);
    //发送TCP数据报
    udt_send(tcpPack);
    flag = 0;

    //等待ACK报文
    //waitACK();
    while (flag == 0) ;
}
```

作为发送方，在发送时启用超时计时器后，还需要在正确接受到 ACK 包后清除当前的超时计时器。

```
public void waitACK() {
    //循环检查ackQueue
    //循环检查确认号对列中是否有新收到的ACK
    if (!ackQueue.isEmpty()) {
        int currentAck = ackQueue.poll();
        // System.out.println("CurrentAck: "+currentAck);â
        if (currentAck == tcpPack.getTcpH().getTh_seq()) {
            System.out.println("Clear: " + tcpPack.getTcpH().getTh_seq());
            timer.cancel();
            flag = 1;
            //break;
        } else { // NAK 或 ACK上一个包
            System.out.println("Retransmit: " + tcpPack.getTcpH().getTh_seq());
            udt_send(tcpPack);
            flag = 0;
        }
    }
}
```

发送方收到重复的 ACK 包后，发送下一个数据包。接受方在接收到重复的数据包后，判断是不是按序到达，如果是则，则对当前包进行 ACK，并交付数据，如果不是则发送对上一个包的 ACK。

为了模拟信道丢包的情况，需要修改 eFlag 的值为 4。【出错/丢包】

发送方：

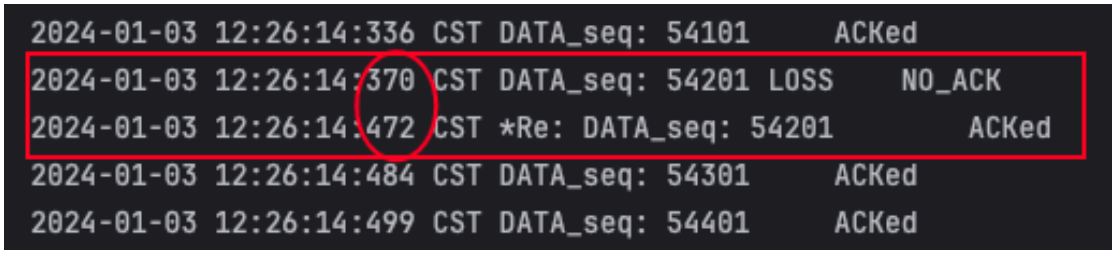
```
@Override
//不可靠发送：将打包好的TCP数据报通过不可靠传输信道发送；仅需修改错误标志
public void udt_send(TCP_PACKET stcpPack) {
    //设置错误控制标志
    tcpH.setTh_eflag((byte) 4);
    //System.out.println("to send: "+stcpPack.getTcpH().getTh_seq());
    //发送数据报
    client.send(stcpPack);
}
```

接收方：

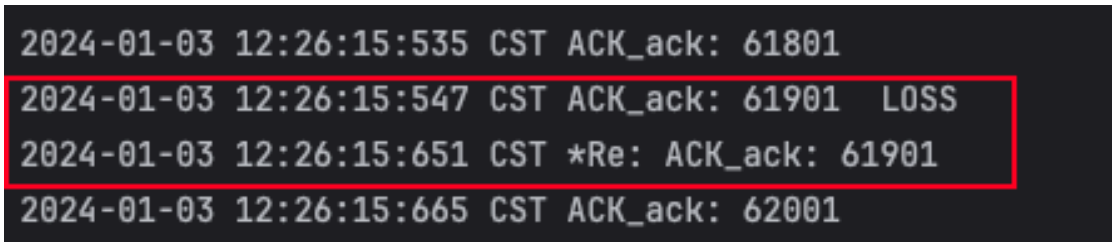
```
@Override
//回复ACK报文段
public void reply(TCP_PACKET replyPack) {
    //设置错误控制标志
    tcpH.setTh_eflag((byte) 4);     //eFlag=0, 信道无错误

    //发送数据报
    client.send(replyPack);
}
```

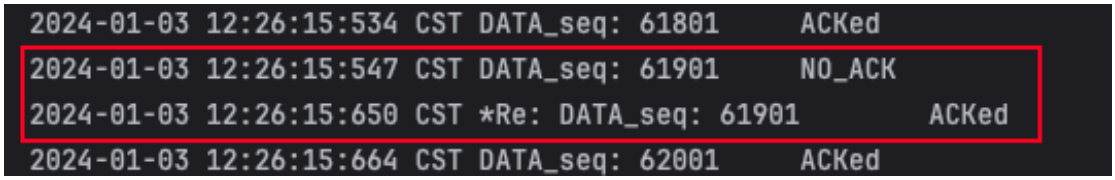
查看日志：



可以发现发送方在发送 54201 包的时候丢包了，没有收到 ACK，在 102ms 后进行了重发，其中的 2ms 可能是因为软件运行已经写日志花费的时间。



查看接收方日志，可以发现 ACK61901 包发生了丢失，发送方没有收到，在 100ms 后重发了 61901 号包。



GoBackN

要实现 GoBackN 协议，需要发送方维护一个滑动窗口。表达窗口的数据结构为 SlideWindow，具体见 SlideWindow.java 中，如图所示。

```java
public class SlideWindow {
    2 个用法
    int size = 5; // 窗口大小
    6 个用法
    private final LinkedBlockingQueue<TCP_PACKET> window = new LinkedBlockingQueue<>(this.size); // 窗口
    6 个用法
    private UDT_Timer timer; // 超时计时器
    2 个用法
    TCP_Sender sender; // 保存发送者的引用，用于发送数据
```

作为发送方，需要完成以下几件事情：

1. 维护一个滑动窗口。
2. 如果滑动窗口已满，阻塞应用层调用。
3. 在滑动窗口中，使用 1 个超时计时器，超时没有收到正确 ack 包，则重新发送窗口内所有数据。
4. 完成累积确认后，将窗口进行滑动。

为了维护一个滑动窗口，在构造发送方时，对滑动窗口进行初始化。

```java
/*构造函数*/
0 个用法  bai 1
public TCP_Sender() {
    super();     //调用超类构造函数
    super.initTCP_Sender( tcpSender: this);          //初始化TCP发送端
    this.slideWindow = new SlideWindow( sender: this);
}
```

在应用层调用 rdt_send 的时候，如果滑动窗口已经满了，则对应用层调用进行阻塞，即将 flag 设置为 0（这里为了方便讲 flag 设置为了 boolean，如果滑动窗口满则为 0），等待滑动窗口有空间后，将包放入滑动窗口内。

```java
//可靠发送（应用层调用）：封装应用层数据，产生TCP数据报；需要修改
public void rdt_send(int dataIndex, int[] appData) {
    //设置错误控制标志
    tcpH.setTh_eflag((byte) 7);
    //生成TCP数据报（设置序号和数据字段/校验和),注意打包的顺序
    tcpH.setTh_seq(dataIndex * appData.length + 1);//包序号设置为字节流号:
    tcpS.setData(appData);
    tcpPack = new TCP_PACKET(tcpH, tcpS, destinAddr);
    //更新带有checksum的TCP 报文头
    tcpH.setTh_sum(CheckSum.computeChkSum(tcpPack));
    tcpPack.setTcpH(tcpH);

    // 如果滑动窗口已满，阻塞应用层调用
    flag = !slideWindow.isFull();
    while (!flag) ;

    // 将数据包加入滑动窗口
    try {
        slideWindow.add(tcpPack.clone());
    } catch (CloneNotSupportedException e) {
        throw new RuntimeException(e);
    }

}
```

对于滑动窗口来说，需要维护一个超时计时器，重传滑动窗口内所有数据。我这里编写了 start_timer 函数来开启计时器，如果需要结束使用 timer.cancel()。

```java
private void start_timer() {
    timer = new UDT_Timer();
    白
    timer.schedule(new TimerTask() {
        白
        @Override
        public void run() {
            send();
        }
    }, delay: 100, period: 100);
}
```

send 函数发送窗口内所有包。

```
public void send() {
    for (TCP_PACKET packet : window) {
        sender.udt_send(packet);
    }
}
```

在将包放入滑动窗口内时，需要重启超时计时器。

```
public void add(TCP_PACKET packet) {
    window.add(packet);
    // 取消之前的计时器
    if (timer != null) {
        timer.cancel();
    }
    // 重启计时器
    start_timer();
}
```

发送方还需要完成累积确认，此时应该放弃前面使用的停止等待协议的 waitACK 代码，交给 SlideWindow 进行处理。

```
//接收到ACK报文：检查校验和，将确认号插入ack队列;NACK的确认号为－1；不需要修改
public void recv(TCP_PACKET recvPack) {
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
        System.out.println("Receive ACK Number:  " + recvPack.getTcpH().getTh_ack());
        slideWindow.recv(recvPack);
        flag = !slideWindow.isFull();
    }
}
```

在 SlideWindow 中完成累积确认，首先需要将已经确认的包（即 seq <= ack）移出滑动窗口中。在滑动窗口更新后，如果滑动窗口内没有数据了，应该清除计时器，防止空转。

```
public void recv(TCP_PACKET packet) {
    for (TCP_PACKET p : window) {
        // 对累积确认的处理
        if (p.getTcpH().getTh_seq() <= packet.getTcpH().getTh_ack()) {
            window.poll();
        } else {
            break;
        }
    }
    if(window.isEmpty()) {
        timer.cancel();
    }
}
```

作为接收方，主要任务在如何响应 ack。通过 seq 对包进行编号，如果收到的是待接收的数据包，则对当前包进行确认，否则对按序接受到的最后一个包进行确认。【延迟和重复都是这么处理。】

```
//检查校验码，生成ACK
if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
    int seq = (recvPack.getTcpH().getTh_seq() - 1) / 100;
    if (seq == sequence) {
        // 将接收到的正确有序的数据插入data队列，准备交付
        dataQueue.add(recvPack.getTcpS().getData());
        sequence++;
        sendACK(recvPack.getTcpH().getTh_seq(), recvPack);
    }else {
        // 非所需数据包，发送对上一个包的ack
        sendACK( seq: (sequence - 1) * 100 + 1, recvPack);
    }
```

如何使用计时器跟踪窗口左沿？使用 window.peek()即可跟踪到左沿。

为了模拟真实情况，我们还需要将信道的 eFlag 设置为 7，模拟 出错/丢包/延迟都存在的情况。

接收方：

```
//回复ACK报文段
public void reply(TCP_PACKET replyPack) {
    //设置错误控制标志
    tcpH.setTh_eflag((byte) 7);     //eFlag=0，信道无错误
    //发送数据报
    client.send(replyPack);
}
```

发送方：

```
public void rdt_send(int dataIndex, int[] appData) {
    //设置错误控制标志
    tcpH.setTh_eflag((byte) 7);
    //生成TCP数据报（设置序号和数据字段/校验和),注意打包的顺序
    tcpH.setTh_seq(dataIndex * appData.length + 1);//包序号设置为字节流号:
    tcpS.setData(appData);
    tcpPack = new TCP_PACKET(tcpH, tcpS, destinAddr);
    //更新带有checksum的TCP 报文头
    tcpH.setTh_sum(CheckSum.computeChkSum(tcpPack));
    tcpPack.setTcpH(tcpH);
```

查看日志：

```
2024-01-04 14:09:00:727 CST ACK_ack: 18601
2024-01-04 14:09:00:727 CST ACK_ack: 18701  LOSS
2024-01-04 14:09:00:727 CST ACK_ack: 18801
2024-01-04 14:09:00:727 CST ACK_ack: 18901
```

可以发现 ack8201 延迟到达。

```
2024-01-04 14:09:00:723 CST DATA_seq: 18601     ACKed
2024-01-04 14:09:00:724 CST DATA_seq: 18701     NO_ACK
2024-01-04 14:09:00:724 CST DATA_seq: 18801     ACKed
2024-01-04 14:09:00:724 CST DATA_seq: 18901     ACKed
2024-01-04 14:09:00:876 CST DATA_seq: 19001     ACKed
```

但是在发送方日志中，没有对 8201 进行重发，实现了累积确认。

```
2024-01-04 14:08:55:754 CST DATA_seq: 3501      ACKed
2024-01-04 14:08:55:754 CST DATA_seq: 3601  LOSS    NO_ACK
2024-01-04 14:08:55:754 CST DATA_seq: 3701      NO_ACK
2024-01-04 14:08:55:755 CST DATA_seq: 3801      NO_ACK
2024-01-04 14:08:55:756 CST DATA_seq: 3901      NO_ACK
2024-01-04 14:08:55:859 CST *Re: DATA_seq: 3601     ACKed
2024-01-04 14:08:55:859 CST *Re: DATA_seq: 3701     ACKed
2024-01-04 14:08:55:859 CST *Re: DATA_seq: 3801     ACKed
2024-01-04 14:08:55:860 CST *Re: DATA_seq: 3901     ACKed
2024-01-04 14:08:55:860 CST DATA_seq: 4001      ACKed
```

查看发送方日志，3601 号包丢包，后面的 3701、3801、3901 也没有办法进行 ack，回退了 4 帧。同时可以发现，第一次发送了 3501 – 3901，超时重传发送了 3601 – 4001，实现了滑动窗口。

```
2024-01-04 14:08:55:600 CST ACK_ack: 3401
2024-01-04 14:08:55:755 CST ACK_ack: 3501
2024-01-04 14:08:55:757 CST ACK_ack: 3501
2024-01-04 14:08:55:758 CST ACK_ack: 3501
2024-01-04 14:08:55:758 CST ACK_ack: 3501
2024-01-04 14:08:55:860 CST ACK_ack: 3601
2024-01-04 14:08:55:862 CST ACK_ack: 3701
```

查看接收方日志，可以发现对 3501 发送了四次 ack，第一次是 3501 正确到达，后面三次分别是 3701、3801、3901 到达接收方，但由于 3501 包还没到，而发送对 3501 的 ack。

SR

SR 在 GoBackN 协议的基础上，扩大了接收方的窗口，让其不再为 1，同时接收方需要对已接收的数据包进行缓存，并发送 ack，同时将窗口滑动到第一个没有被 ack 的包。发送方收到 ack 后，对窗口内数据包进行标记，如果没有被 ack，则等待超时重传，同时窗口滑动到第一个没有被 ack 的包。
表示窗口的数据结构为 ReceiverSlideWindow，具体见 ReceiverSlideWindow.java 文件中。

```
public class ReceiverSlideWindow {
    4 个用法
    int start = 0; // 窗口起始序号
    2 个用法
    int size = 4; // 窗口大小
    8 个用法
    private final ArrayList<TCP_PACKET> window = new ArrayList<>(Collections.nCopies(size, o: null)); // 窗口
    3 个用法
    private TCP_Receiver receiver; // 保存发送者的引用
    4 个用法
    protected TCP_HEADER tcpH; // 保存TCP报文头的引用，用于构建ACK包
    2 个用法
    protected TCP_SEGMENT tcpS; // 保存TCP报文段的引用，用于构建ACK包
    3 个用法
    protected Queue<int[]> dataQueue; // 保存对接受端接收数据的队列的引用，用于交付数据
```

发送方：
建立了一个 TCP_PACKET_WITH_CHECK 的队列窗口，TCP_PACKET_WITH_CHECK 在 TCP_PACKET 的基础上添加了是否被 ack 标志以及超时定时器。

```
public class TCP_PACKET_WITH_CHECK {
    5 个用法
    public boolean acked; // 当前包是否被ack
    1 个用法
    private TCP_Sender sender; // 保存发送者的引用，用于发送数据
    3 个用法
    private UDT_Timer timer; // 超时定时器
    2 个用法
    private TCP_PACKET packet; // 包
```

主要修改对收到 ack 包后的处理，此时不能使用 GoBackN 的累积确认，而应该对窗口内每个包都进行确认。在确认完成后，窗口滑动到第一个没有被 ack 包的位置，并清除已移出窗口的数据包。

```
public void recv(TCP_PACKET packet) {
    for (TCP_PACKET_WITH_CHECK p : window) {
        // 对包进行ack
        if (!p.acked && p.get_seq() == packet.getTcpH().getTh_ack()) {
            p.ack();
        }
    }
    // 滑动窗口
    while(!window.isEmpty() && window.peek().acked){
        window.poll();
    }
}
```

Ack 函数主要功能为标志该包已经被确认，并取消重传计时器。

```
public void ack() {
    this.acked = true;
    this.timer.cancel();
}
```

同时为了方便，将阻塞应用层调用的 flag 标志改为了 bool 值，表示窗口非慢，如果满，则阻塞调用。

```
// 如果滑动窗口已满，阻塞应用层调用
flag = !slideWindow.isFull();
while (!flag) ;
```

计时器数组如何生成，如何去除？计时器数组包括 TCP_PACKET_WITH_CHECK 队列中，在构造函数中生成了计时器。在包被移出窗口的时候，清除了计时器。

接收方：

为了方便，接收方收到数据包后，只校验包是否错误，正确则交给窗口进行处理。

```
//接收到数据报：检查校验和，设置回复的ACK报文段
public void rdt_recv(TCP_PACKET recvPack) {
    //检查校验码，生成ACK
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
        try {
            window.recv(recvPack.clone());
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException(e);
        }
    }
    System.out.println();
}
```

窗口处理 ack 包的函数为 recv，首先计算当前包的序号，如果小于发送窗口前沿则对该包进行 ack。【主要目的为处理 ack 丢失后，发送方重传数据包后，无法 ack 的问题】。如果数据包落在窗口内，则还需要将数据包放入窗口。【start 为后延，end 为前沿】

```
public void recv(TCP_PACKET recvPack) {
    int end = start + size - 1;
    int seq = (recvPack.getTcpH().getTh_seq() - 1) / 100;
    int index = seq - start;
    if(seq <= end) {
        // 发送ack
        tcpH.setTh_ack(recvPack.getTcpH().getTh_seq());
        TCP_PACKET ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());
        tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));
        receiver.reply(ackPack);
        if(seq >= start){
            // 如果落在窗口内，更新窗口
            window.set(index, recvPack);
        }
    }
```

然后需要滑动窗口，在滑动窗口前，需要将已按序接受到的数据放入缓冲中，等待交付上层应用。之后滑动窗口，将窗口后移到第一个没有被 ack 的包上，并其后的位置设置为 null。并且每按序收到 20 个包，交付数据给上层应用。

```
// 数据交到缓存区
int i = 0;
for(; i < window.size() && window.get(i) != null; ++i){
    dataQueue.add(recvPack.getTcpS().getData());
    start++;
}
// 滑动窗口
for(int j = 0;j < window.size() - i; j++){
    window.set(j, window.get(j + i));
}
while (i > 0) {
    window.set(window.size() - i, null);
    i--;
}

// 交付数据
if (dataQueue.size() == 20)
    receiver.deliver_data();
```

查看日志文件：

```
2024-01-04 22:37:09:164 CST DATA_seq: 6101      ACKed
2024-01-04 22:37:09:176 CST DATA_seq: 6201  LOSS    NO_ACK
2024-01-04 22:37:09:190 CST DATA_seq: 6301      ACKed
2024-01-04 22:37:09:202 CST DATA_seq: 6401      ACKed
2024-01-04 22:37:09:214 CST DATA_seq: 6501      ACKed
2024-01-04 22:37:09:278 CST *Re: DATA_seq: 6201     ACKed
2024-01-04 22:37:09:279 CST DATA_seq: 6601      ACKed
```

窗口大小为 4，此时在窗口内的包应为 6201，6301，6401，6501。其中 6201 包丢包，在 100ms 后，只重传了 6201 包，完成了选择重传。

```
2024-01-04 22:37:16:996 CST ACK_ack: 68801
2024-01-04 22:37:17:006 CST ACK_ack: 68901
2024-01-04 22:37:17:018 CST ACK_ack: 69001  LOSS
2024-01-04 22:37:17:030 CST ACK_ack: 69101
2024-01-04 22:37:17:042 CST ACK_ack: 69201
```

Ack69001 发生了丢包。

```
2024-01-04 22:37:17:005 CST DATA_seq: 68901     ACKed
2024-01-04 22:37:17:017 CST DATA_seq: 69001     NO_ACK
2024-01-04 22:37:17:029 CST DATA_seq: 69101     ACKed
2024-01-04 22:37:17:041 CST DATA_seq: 69201     ACKed
2024-01-04 22:37:17:052 CST DATA_seq: 69301     ACKed
2024-01-04 22:37:17:119 CST *Re: DATA_seq: 69001        ACKed
2024-01-04 22:37:17:119 CST DATA_seq: 69401     ACKed
```

发送方日志中，可以发现，只对窗口内的 69001 包进行了重传。

TCP

TCP 和 GBN 类似，使用累积确认。但在接收端与 SR 类似，有缓存来正确接受失序的分组。同时采用快重传机制，在超时事件发生前就对报文段进行重传。
发送端的窗口采用 GBN 的窗口，在此基础上进行修改。
需要在处理 ACK 包的基础上加上快重传机制。
last_acked 记录上一个 ack 包的 ack 号，ack_count 已连续收到多少个这个 ack 包。
resend_threshold 为重发阈值，即连续收到多少个 ack 包立即重发。

在代码中，设置了发送方和接收方窗口大小均为 8。Resend_threshold 为 3。
在到达阈值之前，对 ack_count+1，到达阈值后，对 ack_count 进行重置。

```
public void recv(TCP_PACKET packet) {
    if (packet.getTcpH().getTh_ack() == last_acked) {
        // 数量 + 1
        ack_count++;
        // 累积数量到达阈值，立即重发
        if (ack_count == resend_threshold) {
            ack_count = 0;
            resend();
            timer.cancel();
            start_timer();
        }
    } else {
        last_acked = packet.getTcpH().getTh_ack();
        ack_count = 1;
    }

    for(TCP_PACKET p : window){
        if(p.getTcpH().getTh_seq() <= packet.getTcpH().getTh_ack()){
            window.poll();
        }
    }
```

其中 resend 只重发窗口左沿的一个包。

```
    public void resend() {
        for (TCP_PACKET packet : window) {
            if (packet.getTcpH().getTh_seq() == last_acked + 100) {
//                System.out.println("快重传" + packet.getTcpH().getTh_seq() + "~~~~~~~~~~
                sender.udt_send(packet);
            } else if (packet.getTcpH().getTh_seq() > last_acked + 100) {
                break;
            }
        }
    }
```

接收方的窗口采用 SR 的窗口，在此技术上，修改对 ack 包的处理。判断是否落在窗口内，如果落在窗口内，则将数据放入缓存，等待交付应用层。之后再返回响应的 ACK 包，即对最后一个按序收到的数据包进行 ACK。【滑动窗口后，获取窗口后沿即可实现。】，如果不落在窗口内，则对最后一个按序收到的数据包进行 ACK。

```java
public void recv(TCP_PACKET recvPack) {
    int end = start + size - 1;
    int seq = (recvPack.getTcpH().getTh_seq() - 1) / 100;
    int index = seq - start;
    if(seq >= start && seq <= end) {
        window.set(index, recvPack);
    }

    // 数据交到缓存区
    int i = 0;
    for(; i < window.size() && window.get(i) != null; ++i){
        dataQueue.add(recvPack.getTcpS().getData());
//          System.out.println("交付数据" + (start + i) + "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        start++;
    }
    TCP_PACKET ackPack = new TCP_PACKET(tcpH, tcpS, recvPack.getSourceAddr());
    // 由于程序中ACK是对当前包的确认，所以需要 -1
    tcpH.setTh_ack((start - 1) * 100 + 1);
    tcpH.setTh_sum(CheckSum.computeChkSum(ackPack));
    receiver.reply(ackPack);
```

查看日志：

```
2024-01-05 04:25:20:344 CST DATA_seq: 13501     ACKed
2024-01-05 04:25:20:537 CST DATA_seq: 13601 LOSS    NO_ACK
2024-01-05 04:25:20:538 CST DATA_seq: 13701     NO_ACK
2024-01-05 04:25:20:538 CST DATA_seq: 13801     NO_ACK
2024-01-05 04:25:20:538 CST DATA_seq: 13901     NO_ACK
2024-01-05 04:25:20:538 CST DATA_seq: 14001     NO_ACK
2024-01-05 04:25:20:538 CST DATA_seq: 14101     NO_ACK
2024-01-05 04:25:20:538 CST DATA_seq: 14201     NO_ACK
2024-01-05 04:25:20:538 CST DATA_seq: 14301     ACKed
2024-01-05 04:25:20:540 CST *Re: DATA_seq: 13601        NO_ACK
2024-01-05 04:25:20:541 CST *Re: DATA_seq: 13601        NO_ACK
2024-01-05 04:25:20:726 CST DATA_seq: 14401     ACKed
```

13601 号包丢失，但在几 ms 后，14301 号包收到了 ACK。在这段期间，发送方收到 3 个连续的 ACK 确认，发生了快重传。其中 14301 的时间 538 其实是发送该包的时间，接收方发送 ACK14301 包的时间为 542，快重传 13601 包到达接受方的时间为 541，在确认 14301 包之前，13601 已经达到了接收方，所以可以进行 ACK。【查看控制台输出可以知道，具体看下图】

```
-> 2024-01-05 04:25:20:541 CST
** TCP_Receiver
   Receive packet from: [198.18.0.1:9001]
   Packet data: 147137 147139 147151 147163 147179 147197 147209 147211 147221 147227 147229 147253 147263 147283 147289
   PACKET_TYPE: DATA_SEQ_13601

-> 2024-01-05 04:25:20:542 CST
** TCP_Sender
   Receive packet from: [198.18.0.1:9002]
   Packet data:
   PACKET_TYPE: ACK_14301
Receive ACK Number:  14301
```

TCP Tahoe

Tahoe 是 TCP 的最早版本，其主要有三个算法去控制数据流和拥塞窗口。慢开始、拥塞避免、快重传【上面已经实现】。
下面假设接收方的窗口无限大。

```
//    int size = 8; // 窗口大小（假设无限大，以保证能正常测试拥塞控制）
    11 个用法
    private final ArrayList<TCP_PACKET> window = new ArrayList<>(); // 窗口
```

只要大于窗口后沿，则放入窗口内。

```
//        int end = start + size - 1;
        int seq = (recvPack.getTcpH().getTh_seq() - 1) / 100;
        int index = seq - start;
        if(seq >= start) {
            if(window.size() <= index + 1){
                // set null
                for(int i = window.size(); i <= index; i++){
                    window.add(null);
                }
            }
            window.set(index, recvPack);
        }
```

下面介绍一些慢启动和拥塞避免都是怎么实现的。

首先，为发送方窗口新增了几个变量。将 size 改为 cwnd，为拥塞窗口大小。ssThresh 为慢开始门限值，超过该值使用拥塞避免算法。ca_next 记录拥塞避免算法下一次执行窗口增大所需的收到的 ack 包。【用于确保至少是一个 RTT】

```
9 个用法
int cwnd = 1; // 窗口大小
2 个用法
int ssThresh = 16; // 慢启动阈值
2 个用法
int ca_next = -1; // 拥塞避免阶段下一次增大窗口的ack包的值
```

慢开始：

对于慢开始算法来说，完成非常简单，只需要在收到 ack 包的时候，如果拥塞窗口小于阈值，则拥塞窗口+1。假设初始窗口大小为 1，在收到一个 ack 包后，窗口扩大到 2，发送两个包，之后再收到两个 ack，窗口+2，变成了 4。。。即完成了窗口的指数级增长。

```
public void recv(TCP_PACKET packet) {
    System.out.println("当前拥塞窗口大小为: " + cwnd);
    if (cwnd < ssThresh) {
        // 慢开始算法
        cwnd++;
        System.out.println("慢开始算法,  窗口大小+1");
```

拥塞避免：

需要维护 ca_next 的值，以确保是一个 RTT 之后，才执行的拥塞避免算法。

【ca_next 的作用上面介绍过了】

```
} else {
    if(ca_next == -1){
        ca_next = packet.getTcpH().getTh_ack();
    }
    // 拥塞避免
    if(packet.getTcpH().getTh_ack() >= ca_next){ // 不用等于，因为对应的那个ack包可能丢了
        System.out.println("拥塞避免算法，  窗口大小+1");
        ca_next = ca_next + cwnd * 100;
        cwnd++;
    }
}
```

同时需要在发生快重传的时候，慢开始门限设置为窗口大小的一半，窗口大小减为 1。同时快重传重发是对窗口左沿的重发，和上述的 TCP 一样。

```
if (packet.getTcpH().getTh_ack() == last_acked) {
    // 快重传
    // 累积数量到达阈值，立即重发
    if (ack_count == resend_threshold) {
        // 拥塞避免
        ssThresh = cwnd / 2;
        cwnd = 1;

        ack_count = 0;
        resend();
        timer.cancel();
        start_timer();
    }
    // 数量 + 1
    ack_count++;
```

发送方是否仅使用 1 个超时计时器？沿用的 TCP 代码，TCP 中只使用了一个超时计时器。

查看日志：

```
2024-01-05 08:15:51:294 CST DATA_seq: 1     ACKed      1
2024-01-05 08:15:51:430 CST DATA_seq: 101       ACKed
2024-01-05 08:15:51:430 CST DATA_seq: 201       ACKed  2
2024-01-05 08:15:51:581 CST DATA_seq: 301       ACKed
2024-01-05 08:15:51:581 CST DATA_seq: 401       ACKed
2024-01-05 08:15:51:582 CST DATA_seq: 501       ACKed  4
2024-01-05 08:15:51:582 CST DATA_seq: 601       ACKed
2024-01-05 08:15:51:796 CST DATA_seq: 701       ACKed
2024-01-05 08:15:51:797 CST DATA_seq: 801       ACKed
2024-01-05 08:15:51:825 CST DATA_seq: 901       ACKed
2024-01-05 08:15:51:837 CST DATA_seq: 1001      ACKed  8
2024-01-05 08:15:51:839 CST DATA_seq: 1101      ACKed
2024-01-05 08:15:51:842 CST DATA_seq: 1201      ACKed
2024-01-05 08:15:51:876 CST DATA_seq: 1301      ACKed
2024-01-05 08:15:51:879 CST DATA_seq: 1401      ACKed
2024-01-05 08:15:52:222 CST DATA_seq: 1501      ACKed
2024-01-05 08:15:52:223 CST DATA_seq: 1601      ACKed
```

第一次发包数量为 1，第二次为 2，第三次为 4，第四次为 8，慢开始完成。

从上图可以发现实现了拥塞避免，但是有个非常严重的问题，在 9301 包丢失后，后面多个包，都会重复发送对 9301 的确认，导致多次触发快重传，使慢开始门限降为 1，所以导致重传后执行的还是拥塞避免算法。

TCP Reno

快重传算法在 TCP 部分已经描述过了。

Reno 在 Tahoe 的基础上，多了一个快恢复算法，即在快重传结束后，不执行慢开始算法，而是将 ssthresh 的值设置为当前 cwnd 的一半，将 cwnd 设置为 ssThrsh + 3，执行拥塞避免算法，窗口进行线性扩张。

```
if (packet.getTcpH().getTh_ack() == last_acked) {
    // 快重传
    // 累积数量到达阈值，立即重发
    if (ack_count == resend_threshold) {
        // 拥塞避免
        ssThresh = cwnd / 2;
        cwnd = ssThresh + 3;

        ack_count = 0;
        resend();
        timer.cancel();
        start_timer();
    }
    // 数量 + 1
    ack_count++;
```

、

查看日志：

```
2024-01-05 09:12:39:524 CST DATA_seq: 95001     ACKed
2024-01-05 09:12:39:835 CST DATA_seq: 95101     ACKed
2024-01-05 09:12:39:835 CST DATA_seq: 95201     ACKed
2024-01-05 09:12:39:835 CST DATA_seq: 95301     ACKed
2024-01-05 09:12:39:835 CST DATA_seq: 95401     ACKed
2024-01-05 09:12:39:835 CST DATA_seq: 95501     ACKed
2024-01-05 09:12:39:836 CST DATA_seq: 95601     ACKed
2024-01-05 09:12:39:836 CST DATA_seq: 95701     ACKed
2024-01-05 09:12:39:836 CST DATA_seq: 95801     ACKed
2024-01-05 09:12:39:836 CST DATA_seq: 95901     ACKed
2024-01-05 09:12:39:836 CST DATA_seq: 96001     ACKed
2024-01-05 09:12:39:836 CST DATA_seq: 96101     ACKed
2024-01-05 09:12:39:836 CST DATA_seq: 96201     ACKed
2024-01-05 09:12:39:836 CST DATA_seq: 96301     ACKed
2024-01-05 09:12:39:836 CST DATA_seq: 96401     ACKed
2024-01-05 09:12:39:836 CST DATA_seq: 96501 DELAY   NO_ACK
2024-01-05 09:12:39:836 CST DATA_seq: 96601     NO_ACK
2024-01-05 09:12:39:836 CST DATA_seq: 96701     NO_ACK
2024-01-05 09:12:39:836 CST DATA_seq: 96801     ACKed
2024-01-05 09:12:39:842 CST *Re: DATA_seq: 96501        NO_ACK
2024-01-05 09:12:40:072 CST DATA_seq: 96901     ACKed
2024-01-05 09:12:40:072 CST DATA_seq: 97001     ACKed
2024-01-05 09:12:40:072 CST DATA_seq: 97101     ACKed
2024-01-05 09:12:40:073 CST DATA_seq: 97201     ACKed
2024-01-05 09:12:40:073 CST DATA_seq: 97301     ACKed
2024-01-05 09:12:40:073 CST DATA_seq: 97401     ACKed
2024-01-05 09:12:40:073 CST DATA_seq: 97501     ACKed
2024-01-05 09:12:40:073 CST DATA_seq: 97601     ACKed
2024-01-05 09:12:40:073 CST DATA_seq: 97701     ACKed
2024-01-05 09:12:40:073 CST DATA_seq: 97801     ACKed
2024-01-05 09:12:40:073 CST DATA_seq: 97901     ACKed
2024-01-05 09:12:40:073 CST DATA_seq: 98001     ACKed
2024-01-05 09:12:40:314 CST DATA_seq: 98101     ACKed
2024-01-05 09:12:40:314 CST DATA_seq: 98201     ACKed
```

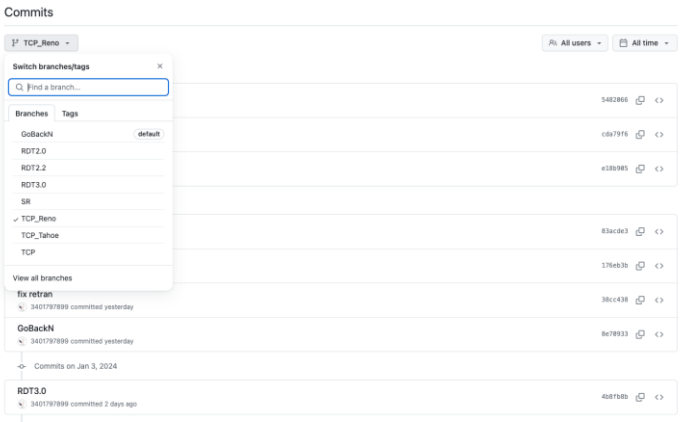
触发快重传之前，窗口大小为18

触发快重传后执行快恢复，窗口大小为12 = 18 / 2 + 3

可以发现，快恢复执行正确。

未完成关键困难

所有实验均已完成，并使用 git 的不同分支进行储存各个版本的代码。



迭代开发

优点：

1. TCP 的协议是在原来提出的协议的基础上一步步优化来的。迭代开发可以更深刻地体会到其中差别以及进步。
2. 对于实验来说，由易到难，容易上手，并且有助于激发继续完成的兴趣。
3. 利于代码重用，如 TCP 的完成需要用到 GBN 和 SR 两个协议的部分代码，迭代开发降低了难度，也利于原来的代码重用。

缺点：

1. 容易忘记 WRONG 和 LOSS 是否处理过，有时候还得回去看看。
2. 对内容没有那么熟悉的情况下，容易纠结这个东西到底在这个协议中有没有。

已解决主要问题

1. 课程中讲解的 ACK 的值确认是不包括当前包的，而实验中的是包括的。例如：ACK101，应该是对 1 号包的确认，而实验中是对 101 号包的确认。
   解决方案：
   想到这个问题主要是在快重传的时候，发现 ack 的值不对，解决方案就是理清 ack 数值之间的关系，发送 ack 时发送对上一个包的确认，快重传时+100。
2. 发送方和接收方的成员变量基本上为 protected，抽象出窗口很麻烦。
   解决方案：
   通过构造函数传入相关参数，将对相关变量的引用进行保存。
3. 快重传时，会有 99901 号包一直在窗口内，移出不掉的情况。
   解决方案：
   研究一番过后，发现是对 ack 的值含义不清楚，调整过后，解决了。
4. 接收方窗口使用 ArrayList 设置无限大时，放入窗口会报错。
   解决方案：
   预先使用 null 进行填满，再进行设置。

```
if(seq >= start) {
    if(window.size() <= index + 1){
        // set null
        for(int i = window.size(); i <= index; i++){
            window.add(null);
        }
    }
    window.set(index, recvPack);
}
```

实验系统建议

1. 发包太快了，容易看不清控制台输出，感觉可以稍微调慢一点。
2. 看了一下 delay 的时间为 20～40s，感觉太长了，按正常速度已经跑完了。

```
public void run() {
    Random timeRand = new Random();
    long delay = (long) (timeRand.nextInt( bound: 20001) + 20000);

    try {
        Thread.sleep(delay);
        Server.forwardMessage(this.delayPack);
    } catch (InterruptedException var5) {
        var5.printStackTrace();
    }

}
```

3. 对于触发快重传的日志里可以顺带显示一下 ACK 的时间。

```
2024-01-05 04:25:20:344 CST DATA_seq: 13501     ACKed
2024-01-05 04:25:20:537 CST DATA_seq: 13601 LOSS    NO_ACK
2024-01-05 04:25:20:538 CST DATA_seq: 13701     NO_ACK
2024-01-05 04:25:20:538 CST DATA_seq: 13801     NO_ACK
2024-01-05 04:25:20:538 CST DATA_seq: 13901     NO_ACK
2024-01-05 04:25:20:538 CST DATA_seq: 14001     NO_ACK
2024-01-05 04:25:20:538 CST DATA_seq: 14101     NO_ACK
2024-01-05 04:25:20:538 CST DATA_seq: 14201     NO_ACK
2024-01-05 04:25:20:538 CST DATA_seq: 14301     ACKed
2024-01-05 04:25:20:540 CST *Re: DATA_seq: 13601        NO_ACK
2024-01-05 04:25:20:541 CST *Re: DATA_seq: 13601        NO_ACK
2024-01-05 04:25:20:726 CST DATA_seq: 14401     ACKed
```

图中，对 13601 的重发时间在 14301 之后，但是在接受窗口滑动后，会对 14301 进行 ACK。初看以为是有问题的，但实际是对的。

4. 建议 TCP_Receiver 可以开放一下成员变量，不要全部 protected 了，要自己设置窗口数据结构并进行发包的话，需要全部传一遍参数。