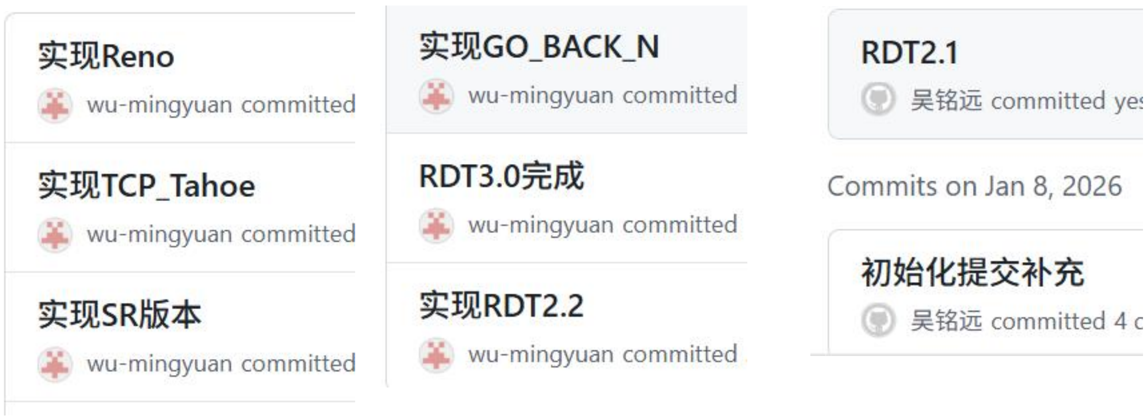
计算机网络大作业报告

学号：__23090012041__ 姓名：__吴铭远__ 专业：__计算机科学与技术__ 年级：__23 级__

Github 提交历史截图


实现Reno
wu-mingyuan committed
实现TCP_Tahoe
wu-mingyuan committed
实现SR版本
wu-mingyuan committed
实现GO_BACK_N
wu-mingyuan committed
RDT3.0完成
wu-mingyuan committed
实现RDT2.2
wu-mingyuan committed
RDT2.1
吴铭远 committed yes
Commits on Jan 8, 2026
初始化提交补充
吴铭远 committed 4 c

一、结合代码和 LOG 文件分析针对每个项目举例说明解决效果。

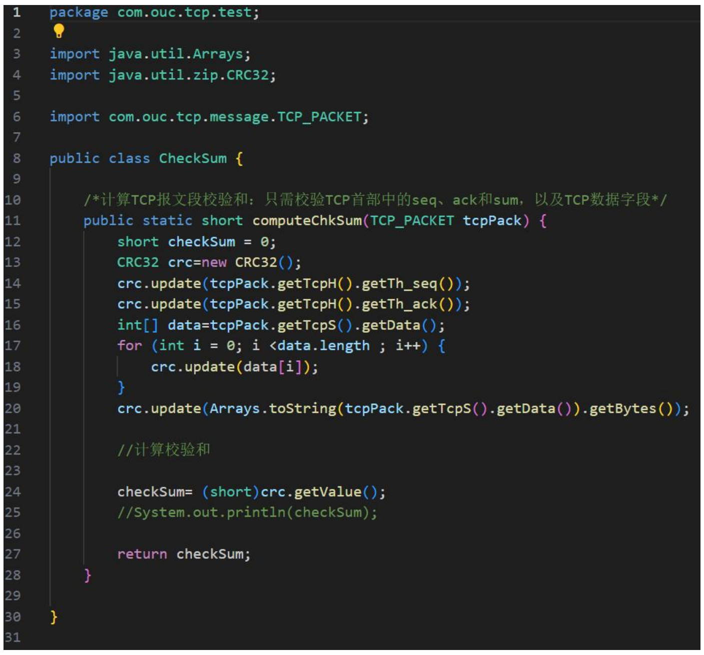
（1）RDT2.1

RDT2.1 相较于 1.0 来说可以通过 ACK/NAK 来进行差错检验。如果 ACK/NAK 出现错误，则进行重传。

具体实现方式：

1.CheckSum.java（校验和检验）

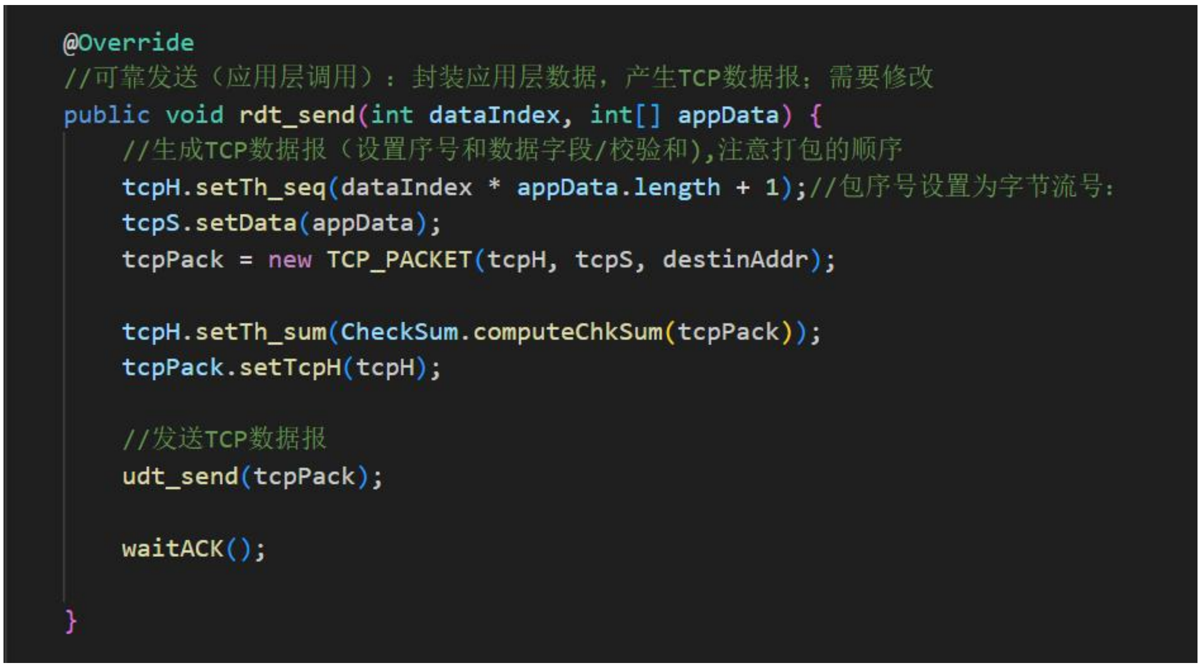
首先设置一个校验和并初始化为 0，之后每次传输之后先通过数据报中的 seq 和 ack 更新校验和。具体实现方法如下：



```java
package com.ouc.tcp.test;

import java.util.Arrays;
import java.util.zip.CRC32;

import com.ouc.tcp.message.TCP_PACKET;

public class CheckSum {

    /*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*/
    public static short computeChkSum(TCP_PACKET tcpPack) {
        short checkSum = 0;
        CRC32 crc=new CRC32();
        crc.update(tcpPack.getTcpH().getTh_seq());
        crc.update(tcpPack.getTcpH().getTh_ack());
        int[] data=tcpPack.getTcpS().getData();
        for (int i = 0; i <data.length ; i++) {
            crc.update(data[i]);
        }
        crc.update(Arrays.toString(tcpPack.getTcpS().getData()).getBytes());

        //计算校验和

        checkSum= (short)crc.getValue();
        //System.out.println(checkSum);

        return checkSum;
    }

}

```

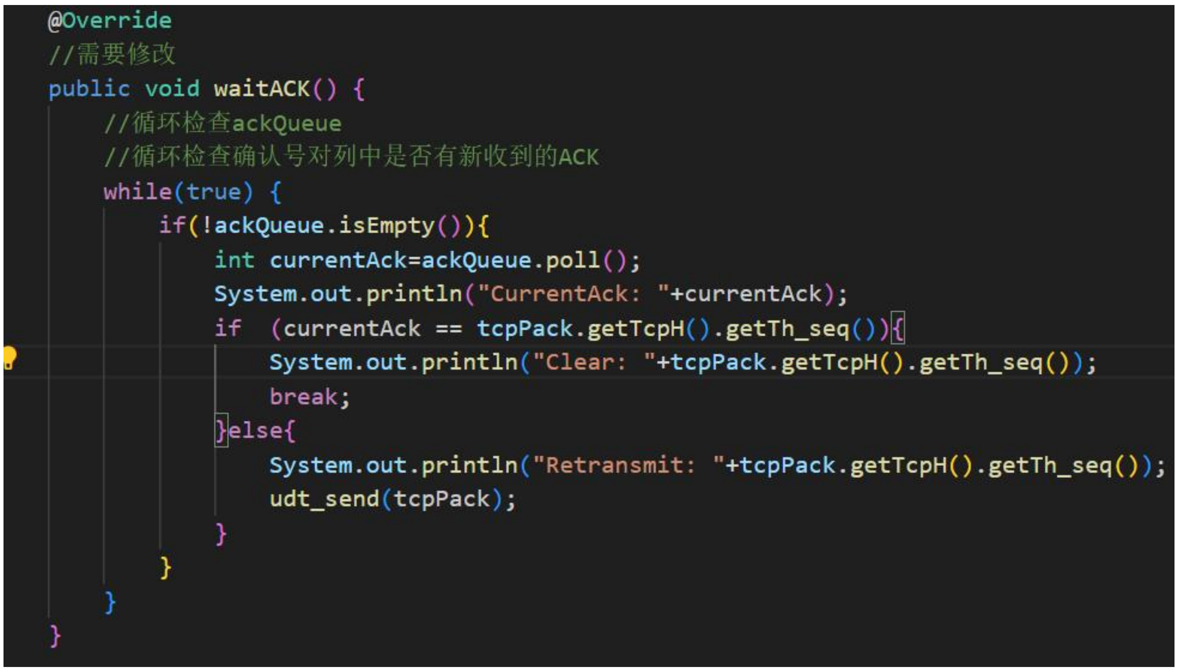
2.　Sender.java（发送端）

发送方每发送出一个数据包，循环检查队列中是否有新收到的 ACK。在 waitACK 函数中，若新收到的 ACK 等于刚刚发送包的 seq，则结束检索，开 始发送下一个数据包；若不等，则重发刚才的数据包，并继续进行 waitACK。



```java
@Override
//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);

    tcpH.setTh_sum(CheckSum.computeChkSum(tcpPack));
    tcpPack.setTcpH(tcpH);

    //发送TCP数据报
    udt_send(tcpPack);

    waitACK();

}
```

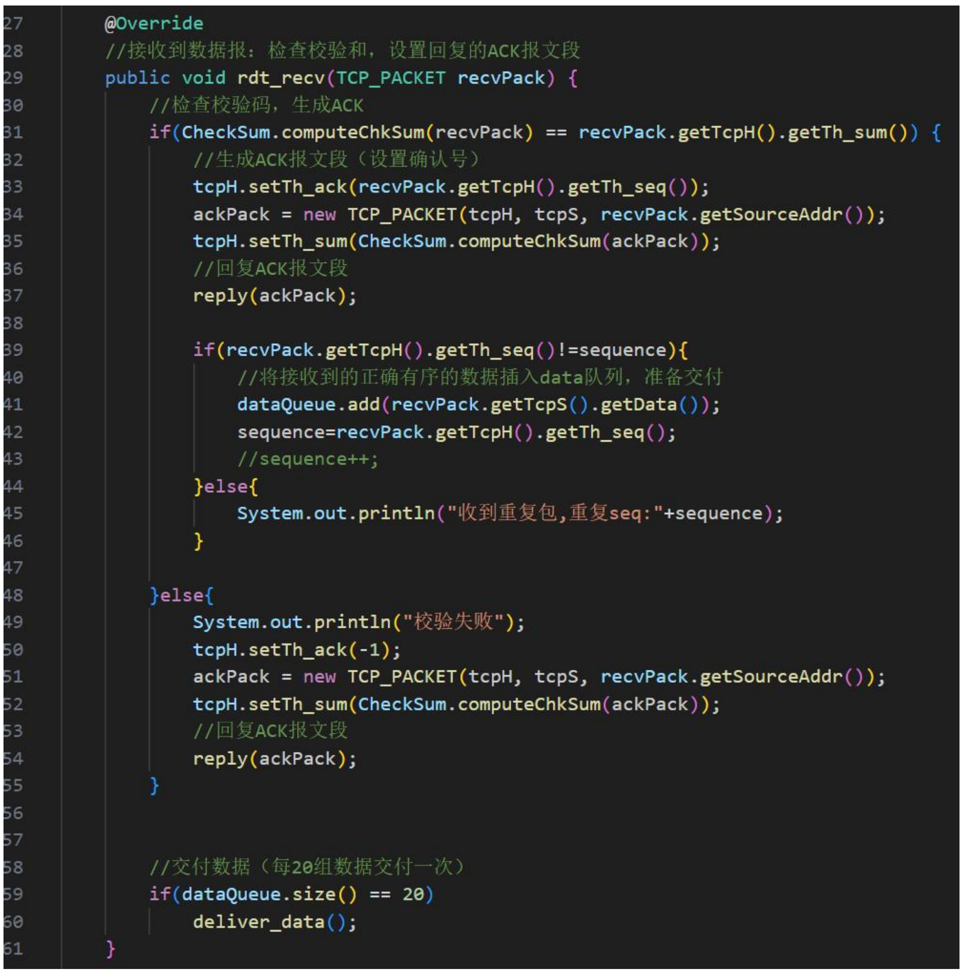
waitACK：



```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
    while(true) {
        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());
                break;
            }else{
                System.out.println("Retransmit: "+tcpPack.getTcpH().getTh_seq());
                udt_send(tcpPack);
            }
        }
    }
}
```

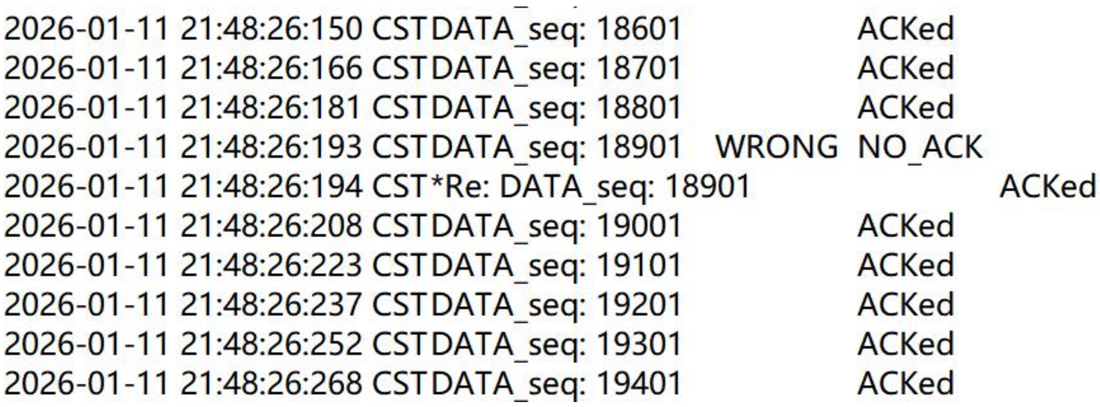
3.Receiver.java（接收端）

每接收到一个包，便计算校验和



```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);

            if(recvPack.getTcpH().getTh_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=recvPack.getTcpH().getTh_seq();
                //sequence++;
            }else{
                System.out.println("收到重复包,重复seq:"+sequence);
            }

        }else{
            System.out.println("校验失败");
            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);
        }


        //交付数据（每20组数据交付一次）
        if(dataQueue.size() == 20)
            deliver_data();
    }
```

日志文件：

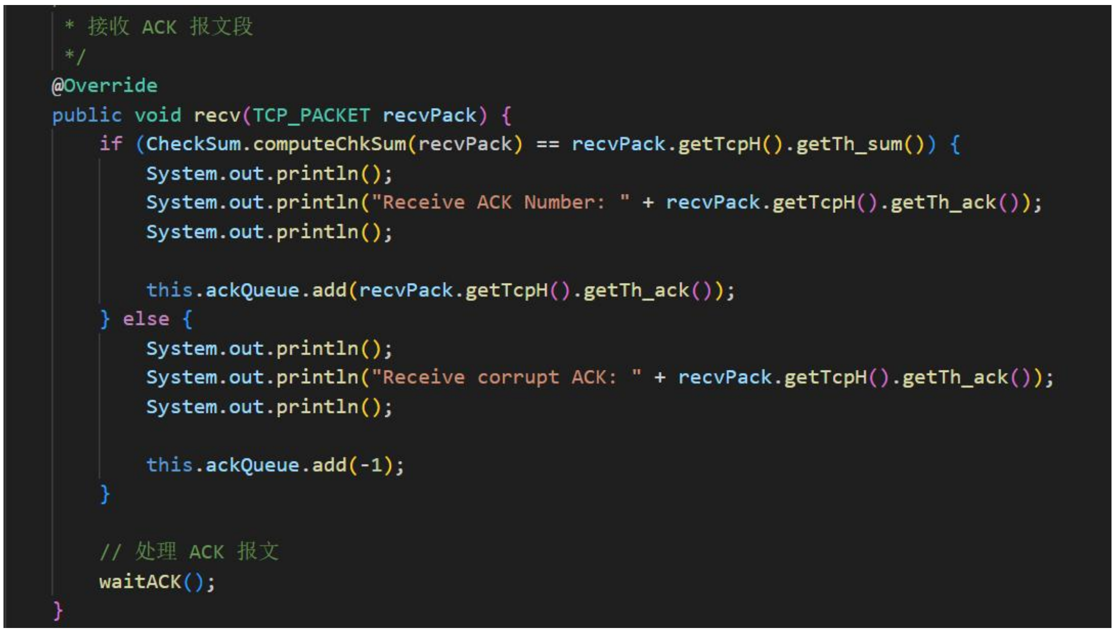


```
2026-01-11 21:48:26:150 CST DATA_seq: 18601            ACKed
2026-01-11 21:48:26:166 CST DATA_seq: 18701            ACKed
2026-01-11 21:48:26:181 CST DATA_seq: 18801            ACKed
2026-01-11 21:48:26:193 CST DATA_seq: 18901   WRONG   NO_ACK
2026-01-11 21:48:26:194 CST *Re: DATA_seq: 18901                ACKed
2026-01-11 21:48:26:208 CST DATA_seq: 19001            ACKed
2026-01-11 21:48:26:223 CST DATA_seq: 19101            ACKed
2026-01-11 21:48:26:237 CST DATA_seq: 19201            ACKed
2026-01-11 21:48:26:252 CST DATA_seq: 19301            ACKed
2026-01-11 21:48:26:268 CST DATA_seq: 19401            ACKed
```

（2）RDT2.2

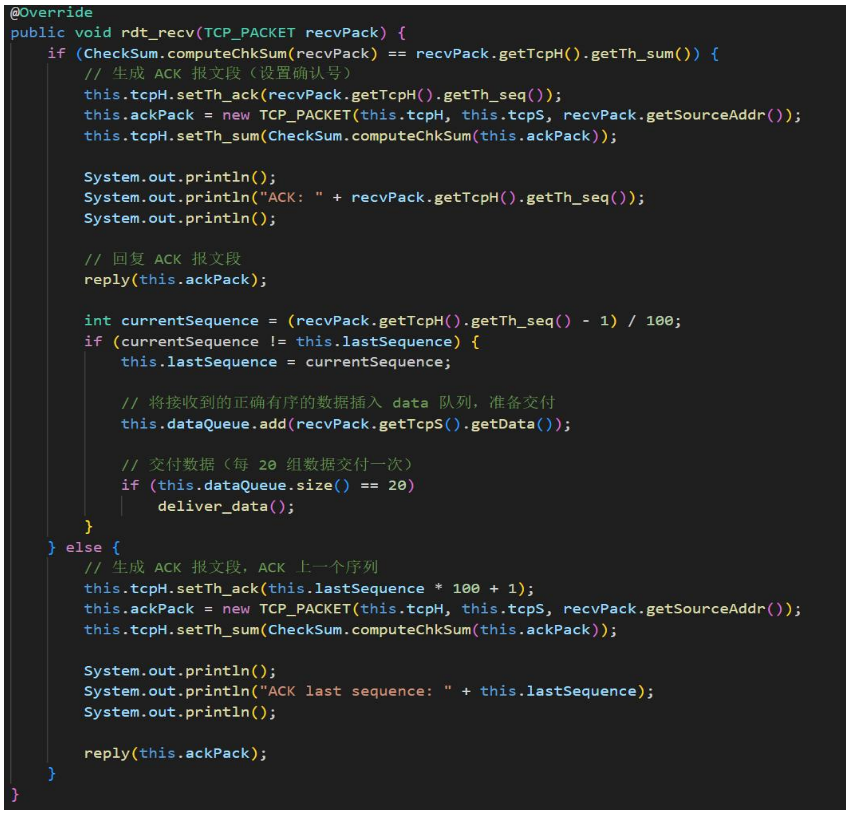
使用 ACK 来完成 2.1 中的 ACK/NAK 的功能。修改 waitACK 函数，只有新收到的 ACK 包的确认字段等于发送数据包 的 seq 字段时，才会接受确认为 ACK；否则，则重发错误的 ACK 序列号。

1. Sender.java（发送端修改）



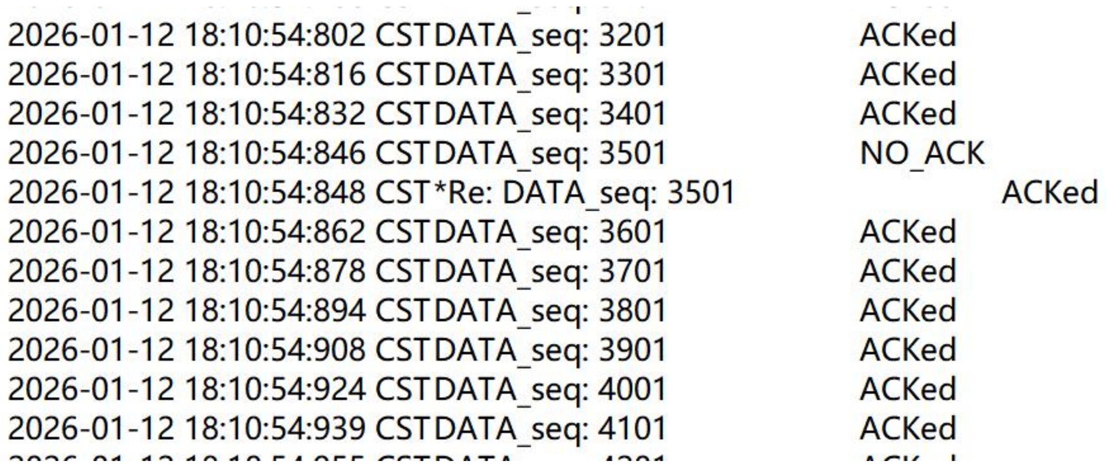
```java
 * 接收 ACK 报文段
 */
@Override
public void recv(TCP_PACKET recvPack) {
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
        System.out.println();
        System.out.println("Receive ACK Number: " + recvPack.getTcpH().getTh_ack());
        System.out.println();

        this.ackQueue.add(recvPack.getTcpH().getTh_ack());
    } else {
        System.out.println();
        System.out.println("Receive corrupt ACK: " + recvPack.getTcpH().getTh_ack());
        System.out.println();

        this.ackQueue.add(-1);
    }

    // 处理 ACK 报文
    waitACK();
}
```

2.Receiver.java（接收端修改）



```java
@Override
public void rdt_recv(TCP_PACKET recvPack) {
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
        // 生成 ACK 报文段（设置确认号）
        this.tcpH.setTh_ack(recvPack.getTcpH().getTh_seq());
        this.ackPack = new TCP_PACKET(this.tcpH, this.tcpS, recvPack.getSourceAddr());
        this.tcpH.setTh_sum(CheckSum.computeChkSum(this.ackPack));

        System.out.println();
        System.out.println("ACK: " + recvPack.getTcpH().getTh_seq());
        System.out.println();

        // 回复 ACK 报文段
        reply(this.ackPack);

        int currentSequence = (recvPack.getTcpH().getTh_seq() - 1) / 100;
        if (currentSequence != this.lastSequence) {
            this.lastSequence = currentSequence;

            // 将接收到的正确有序的数据插入 data 队列，准备交付
            this.dataQueue.add(recvPack.getTcpS().getData());

            // 交付数据（每 20 组数据交付一次）
            if (this.dataQueue.size() == 20)
                deliver_data();
        }
    } else {
        // 生成 ACK 报文段，ACK 上一个序列
        this.tcpH.setTh_ack(this.lastSequence * 100 + 1);
        this.ackPack = new TCP_PACKET(this.tcpH, this.tcpS, recvPack.getSourceAddr());
        this.tcpH.setTh_sum(CheckSum.computeChkSum(this.ackPack));

        System.out.println();
        System.out.println("ACK last sequence: " + this.lastSequence);
        System.out.println();

        reply(this.ackPack);
    }
}
```

3.Log 文件



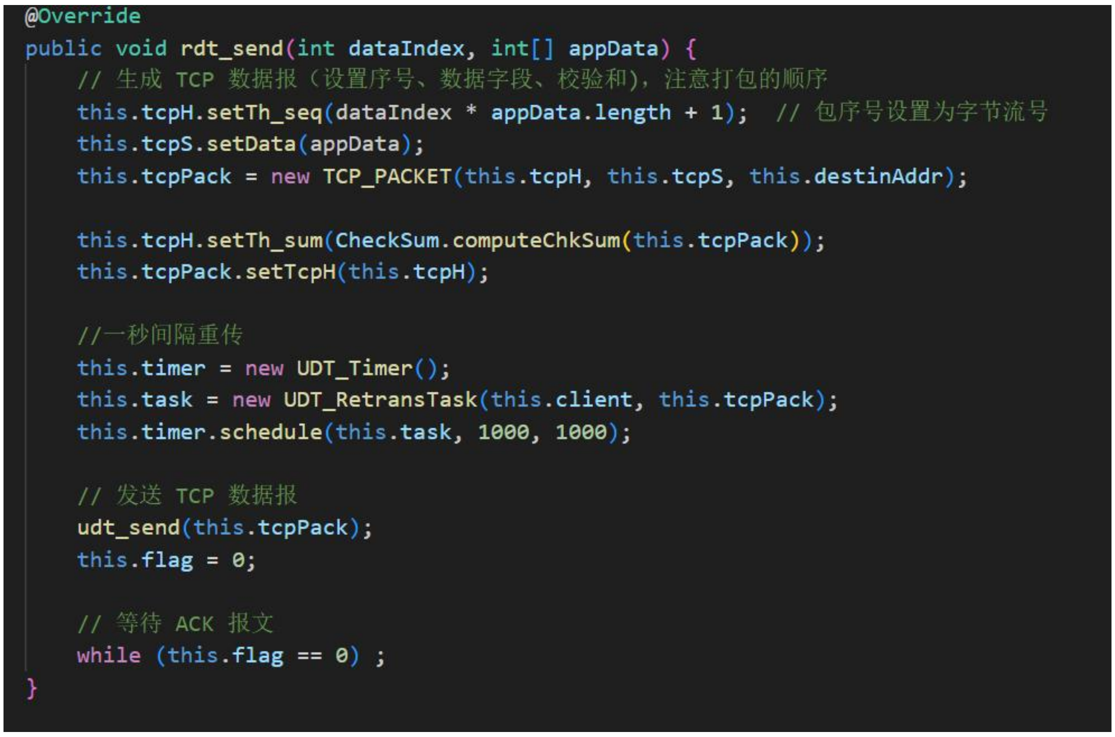
```
2026-01-12 18:10:54:802 CSTDATA_seq: 3201             ACKed
2026-01-12 18:10:54:816 CSTDATA_seq: 3301             ACKed
2026-01-12 18:10:54:832 CSTDATA_seq: 3401             ACKed
2026-01-12 18:10:54:846 CSTDATA_seq: 3501             NO_ACK
2026-01-12 18:10:54:848 CST*Re: DATA_seq: 3501                  ACKed
2026-01-12 18:10:54:862 CSTDATA_seq: 3601             ACKed
2026-01-12 18:10:54:878 CSTDATA_seq: 3701             ACKed
2026-01-12 18:10:54:894 CSTDATA_seq: 3801             ACKed
2026-01-12 18:10:54:908 CSTDATA_seq: 3901             ACKed
2026-01-12 18:10:54:924 CSTDATA_seq: 4001             ACKed
2026-01-12 18:10:54:939 CSTDATA_seq: 4101             ACKed
```

(3) RDT3.0

rdt3.0 在 rdt2.2 的基础之上处理了数据 包丢失的情况，增加了计时器的机制， 如果在 RTT 时间段内， 发送方没有接 收到反馈信息， 那么发送方默认数据包已经丢失了， 会自动重传。

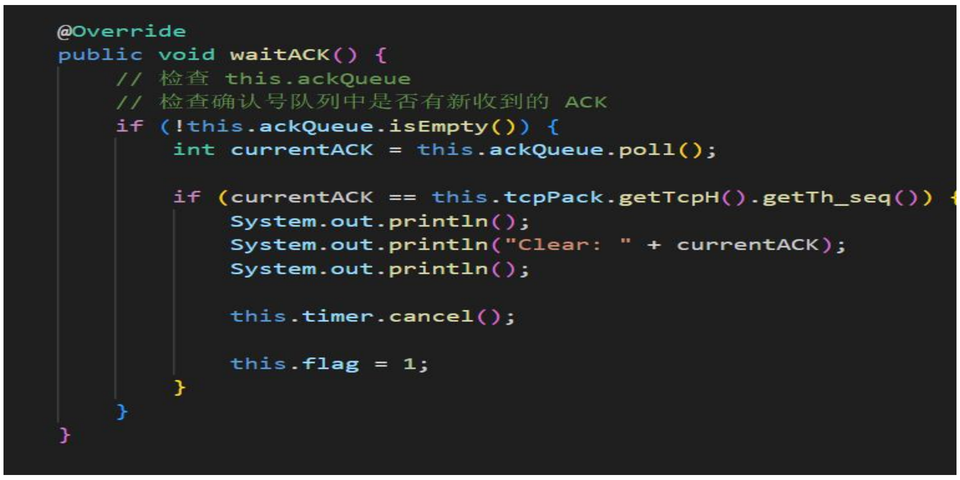
1.Sender.java（发送端修改）

首先生成一个 timer，通过修改 rdt_sender 函数来加入一个定时器以便于设置重传。



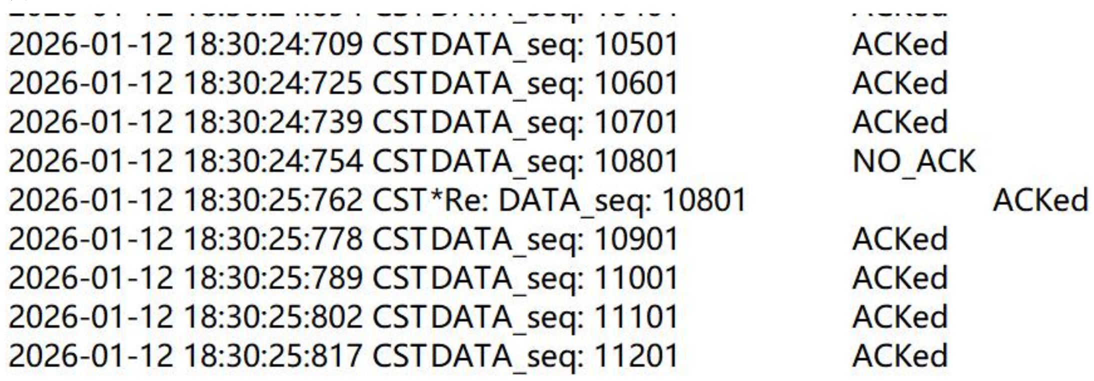
```java
@Override
public void rdt_send(int dataIndex, int[] appData) {
    // 生成 TCP 数据报（设置序号、数据字段、校验和)，注意打包的顺序
    this.tcpH.setTh_seq(dataIndex * appData.length + 1);  // 包序号设置为字节流号
    this.tcpS.setData(appData);
    this.tcpPack = new TCP_PACKET(this.tcpH, this.tcpS, this.destinAddr);

    this.tcpH.setTh_sum(CheckSum.computeChkSum(this.tcpPack));
    this.tcpPack.setTcpH(this.tcpH);

    //一秒间隔重传
    this.timer = new UDT_Timer();
    this.task = new UDT_RetransTask(this.client, this.tcpPack);
    this.timer.schedule(this.task, 1000, 1000);

    // 发送 TCP 数据报
    udt_send(this.tcpPack);
    this.flag = 0;

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

在 waitACK 函数中，如果收到了正确的 ACK 则，关闭计时器。如图：



```
@Override
public void waitACK() {
    // 检查 this.ackQueue
    // 检查确认号队列中是否有新收到的 ACK
    if (!this.ackQueue.isEmpty()) {
        int currentACK = this.ackQueue.poll();

        if (currentACK == this.tcpPack.getTcpH().getTh_seq()) {
            System.out.println();
            System.out.println("Clear: " + currentACK);
            System.out.println();

            this.timer.cancel();

            this.flag = 1;
        }
    }
}
```

日志：

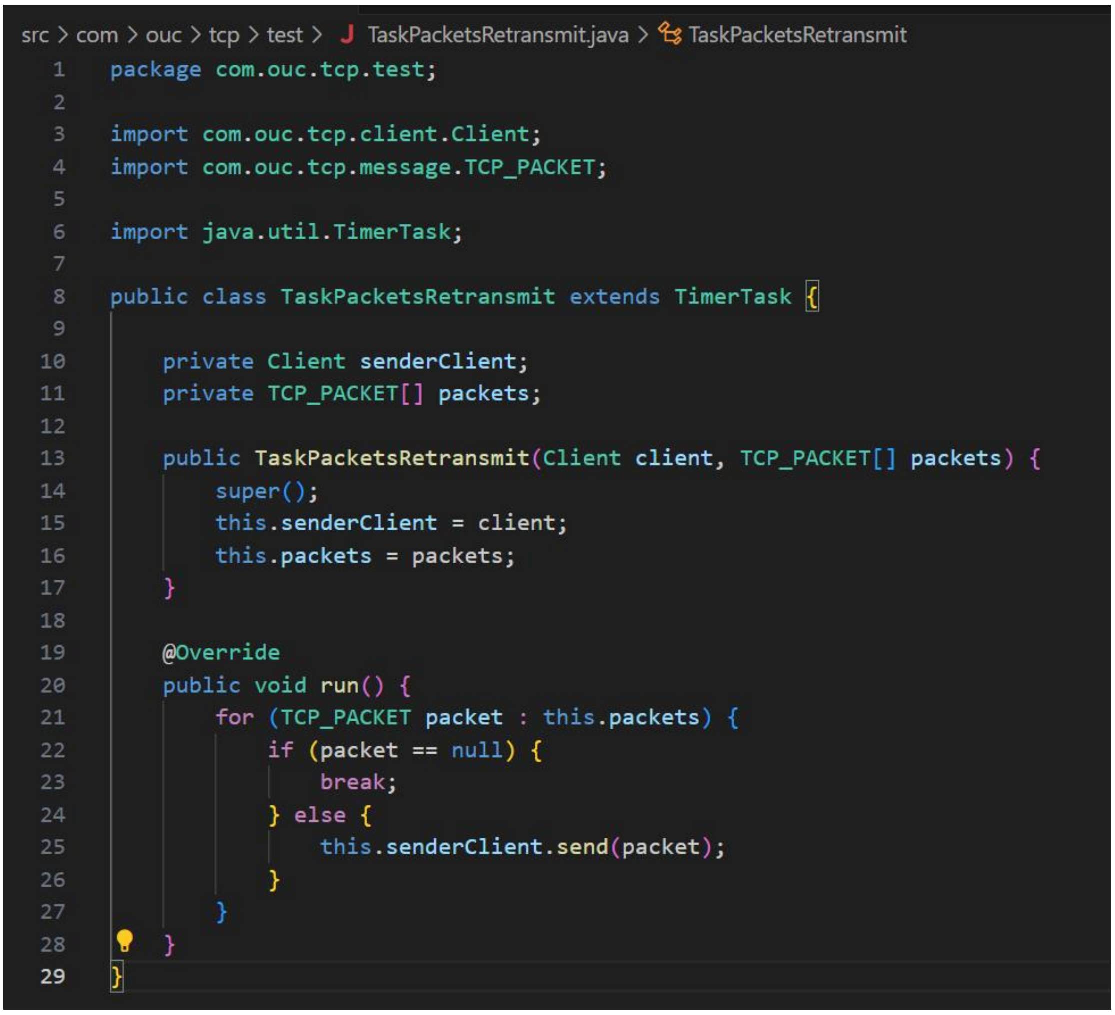


```
2026-01-12 18:30:24:709 CSTDATA_seq: 10501          ACKed
2026-01-12 18:30:24:725 CSTDATA_seq: 10601          ACKed
2026-01-12 18:30:24:739 CSTDATA_seq: 10701          ACKed
2026-01-12 18:30:24:754 CSTDATA_seq: 10801          NO_ACK
2026-01-12 18:30:25:762 CST*Re: DATA_seq: 10801             ACKed
2026-01-12 18:30:25:778 CSTDATA_seq: 10901          ACKed
2026-01-12 18:30:25:789 CSTDATA_seq: 11001          ACKed
2026-01-12 18:30:25:802 CSTDATA_seq: 11101          ACKed
2026-01-12 18:30:25:817 CSTDATA_seq: 11201          ACKed
```

如上图所示，当没有收到 ACK 时，会在间隔一秒后进行重传。

（4）GO_BACK_N

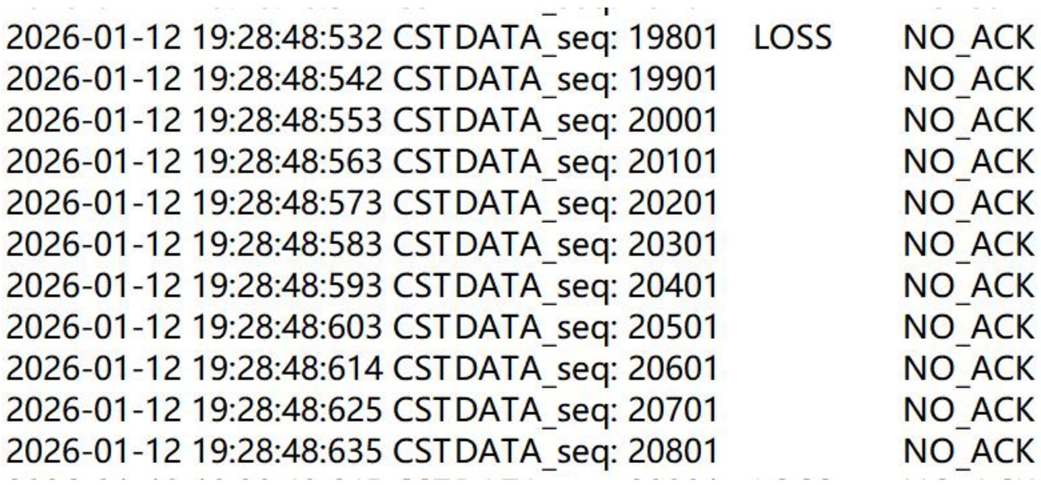
相较于 RDT3.0，GO_BACK_N 采用流水线传输，发送方可以连续发送多个数据包，而不需要等待每个数据包的确认。并且发送方可以连续发送多个数据包，而不需要等待每个数据包的确认。发送方可以连续发送多个数据包，减少了等待时间。



src > com > ouc > tcp > test > TaskPacketsRetransmit.java > TaskPacketsRetransmit

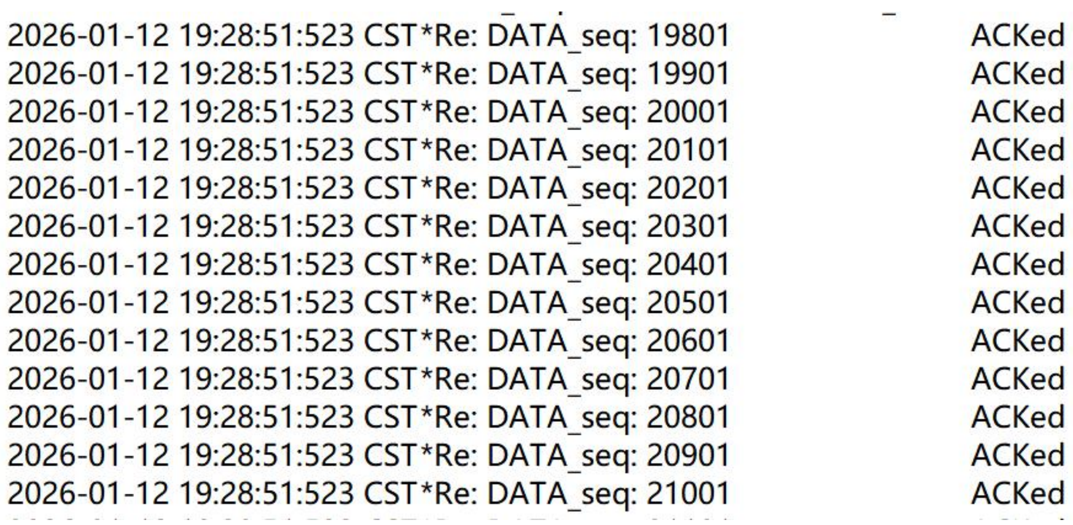
```java
package com.ouc.tcp.test;

import com.ouc.tcp.client.Client;
import com.ouc.tcp.message.TCP_PACKET;

import java.util.TimerTask;

public class TaskPacketsRetransmit extends TimerTask {

    private Client senderClient;
    private TCP_PACKET[] packets;

    public TaskPacketsRetransmit(Client client, TCP_PACKET[] packets) {
        super();
        this.senderClient = client;
        this.packets = packets;
    }

    @Override
    public void run() {
        for (TCP_PACKET packet : this.packets) {
            if (packet == null) {
                break;
            } else {
                this.senderClient.send(packet);
            }
        }
    }
}
```

日志文件：



```
2026-01-12 19:28:48:532 CSTDATA_seq: 19801   LOSS      NO_ACK
2026-01-12 19:28:48:542 CSTDATA_seq: 19901             NO_ACK
2026-01-12 19:28:48:553 CSTDATA_seq: 20001             NO_ACK
2026-01-12 19:28:48:563 CSTDATA_seq: 20101             NO_ACK
2026-01-12 19:28:48:573 CSTDATA_seq: 20201             NO_ACK
2026-01-12 19:28:48:583 CSTDATA_seq: 20301             NO_ACK
2026-01-12 19:28:48:593 CSTDATA_seq: 20401             NO_ACK
2026-01-12 19:28:48:603 CSTDATA_seq: 20501             NO_ACK
2026-01-12 19:28:48:614 CSTDATA_seq: 20601             NO_ACK
2026-01-12 19:28:48:625 CSTDATA_seq: 20701             NO_ACK
2026-01-12 19:28:48:635 CSTDATA_seq: 20801             NO_ACK
```

19801 丢包时，这个窗口的内的数据包都没有收到 ACK



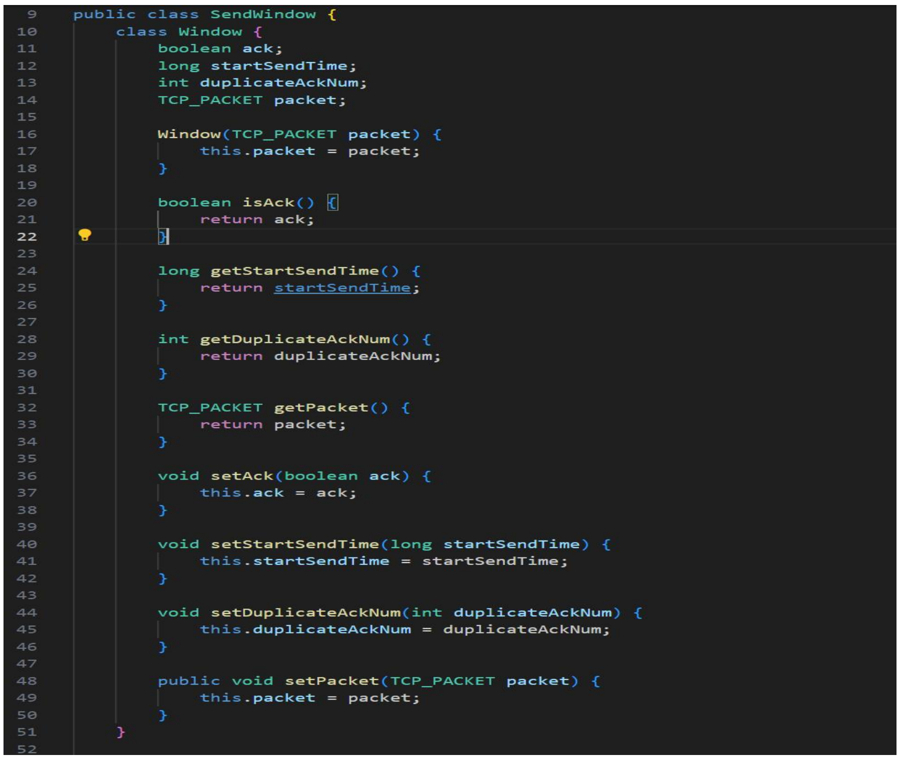
```
2026-01-12 19:28:51:523 CST*Re: DATA_seq: 19801          ACKed
2026-01-12 19:28:51:523 CST*Re: DATA_seq: 19901          ACKed
2026-01-12 19:28:51:523 CST*Re: DATA_seq: 20001          ACKed
2026-01-12 19:28:51:523 CST*Re: DATA_seq: 20101          ACKed
2026-01-12 19:28:51:523 CST*Re: DATA_seq: 20201          ACKed
2026-01-12 19:28:51:523 CST*Re: DATA_seq: 20301          ACKed
2026-01-12 19:28:51:523 CST*Re: DATA_seq: 20401          ACKed
2026-01-12 19:28:51:523 CST*Re: DATA_seq: 20501          ACKed
2026-01-12 19:28:51:523 CST*Re: DATA_seq: 20601          ACKed
2026-01-12 19:28:51:523 CST*Re: DATA_seq: 20701          ACKed
2026-01-12 19:28:51:523 CST*Re: DATA_seq: 20801          ACKed
2026-01-12 19:28:51:523 CST*Re: DATA_seq: 20901          ACKed
2026-01-12 19:28:51:523 CST*Re: DATA_seq: 21001          ACKed
```

后续全部重发

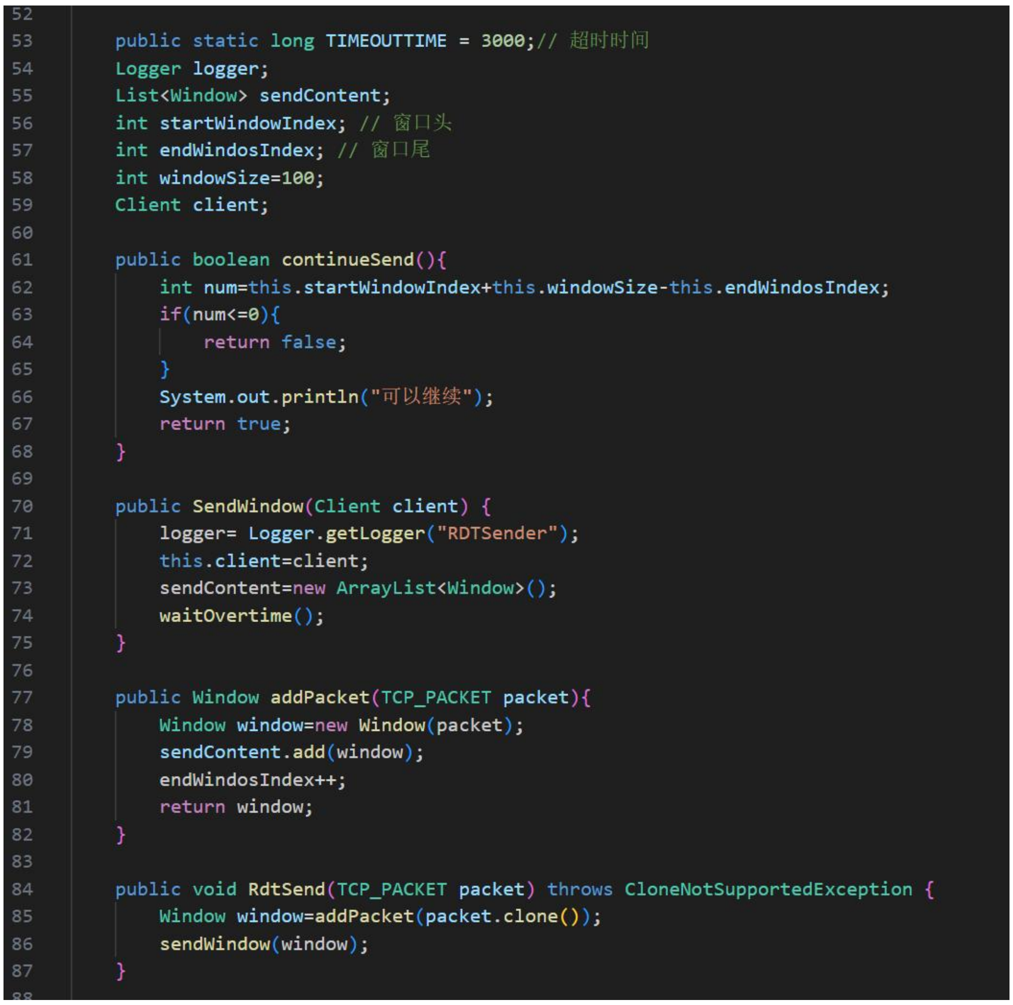
（5）SR

SR 协议只会重传那些发送方认为在 接收方出现差错的分组，避免了不必要的重传，极大的提升了效率。所以需 要建立接收缓冲区。如果一个序号为 n 的分组被正确接收到，并且按序（即上次交付给上层的数据的序号为 n – 1），则接收方为分组 n 发送一个 ACK，并将数据部分交付给上层。

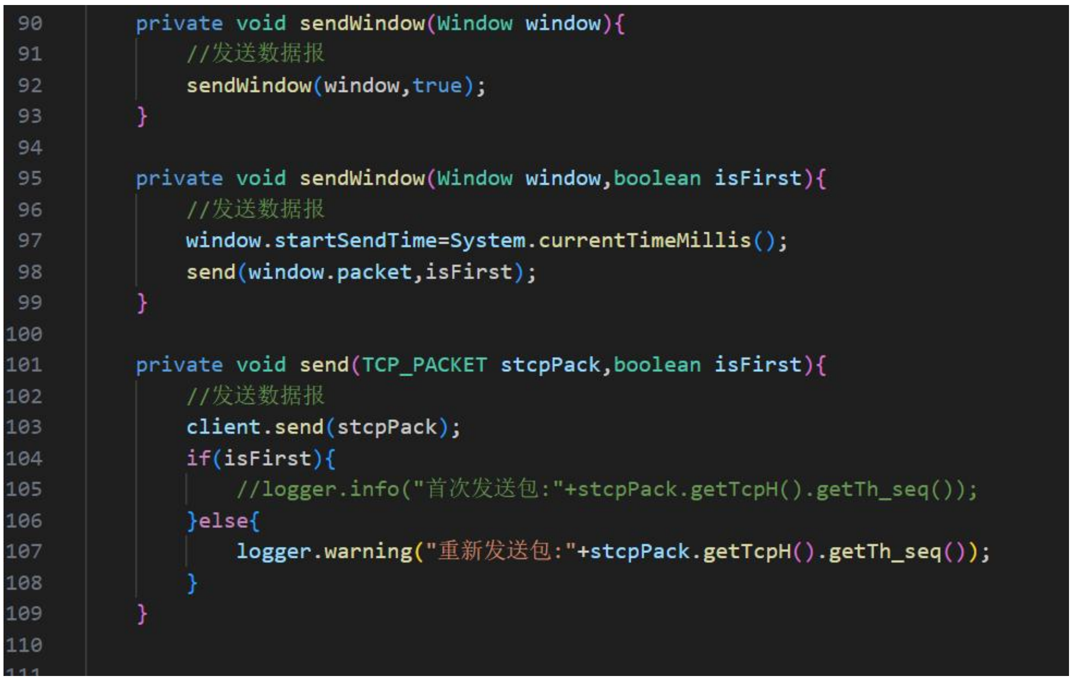
具体实现如下：



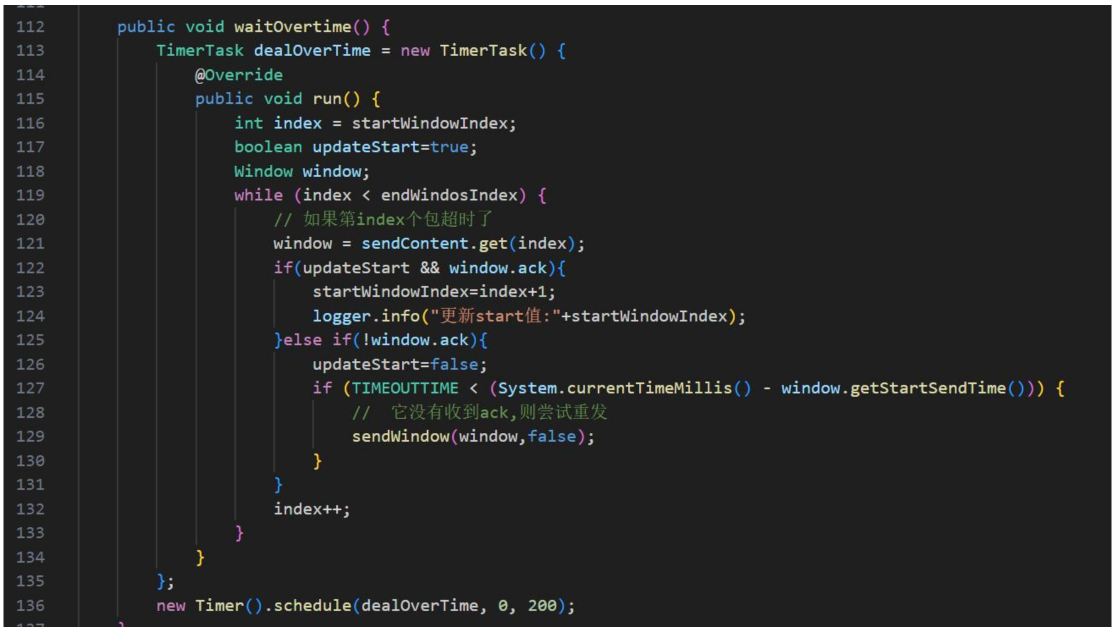
```java
public class SendWindow {
    class Window {
        boolean ack;
        long startSendTime;
        int duplicateAckNum;
        TCP_PACKET packet;

        Window(TCP_PACKET packet) {
            this.packet = packet;
        }

        boolean isAck() {
            return ack;
        }

        long getStartSendTime() {
            return startSendTime;
        }

        int getDuplicateAckNum() {
            return duplicateAckNum;
        }

        TCP_PACKET getPacket() {
            return packet;
        }

        void setAck(boolean ack) {
            this.ack = ack;
        }

        void setStartSendTime(long startSendTime) {
            this.startSendTime = startSendTime;
        }

        void setDuplicateAckNum(int duplicateAckNum) {
            this.duplicateAckNum = duplicateAckNum;
        }

        public void setPacket(TCP_PACKET packet) {
            this.packet = packet;
        }
    }

```



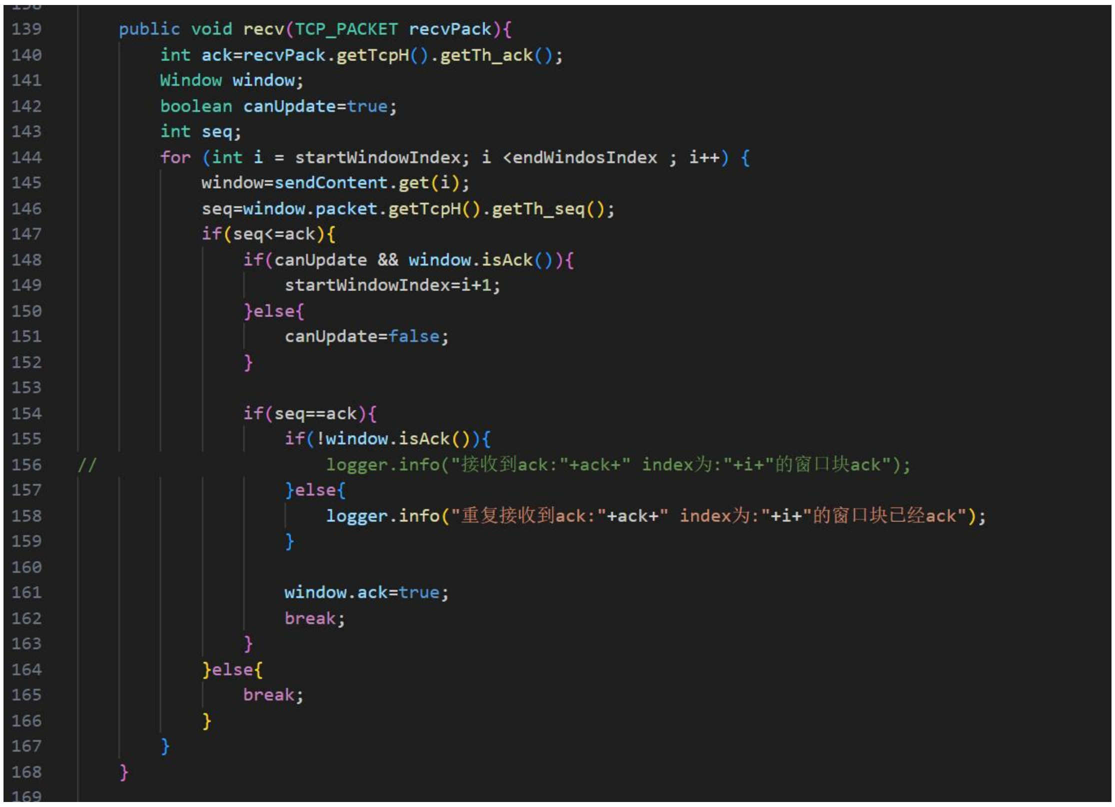
```java
    public static long TIMEOUTTIME = 3000;// 超时时间
    Logger logger;
    List<Window> sendContent;
    int startWindowIndex; // 窗口头
    int endWindosIndex; // 窗口尾
    int windowSize=100;
    Client client;

    public boolean continueSend(){
        int num=this.startWindowIndex+this.windowSize-this.endWindosIndex;
        if(num<=0){
            return false;
        }
        System.out.println("可以继续");
        return true;
    }

    public SendWindow(Client client) {
        logger= Logger.getLogger("RDTSender");
        this.client=client;
        sendContent=new ArrayList<Window>();
        waitOvertime();
    }

    public Window addPacket(TCP_PACKET packet){
        Window window=new Window(packet);
        sendContent.add(window);
        endWindosIndex++;
        return window;
    }

    public void RdtSend(TCP_PACKET packet) throws CloneNotSupportedException {
        Window window=addPacket(packet.clone());
        sendWindow(window);
    }
```



```java
    private void sendWindow(Window window){
        //发送数据报
        sendWindow(window,true);
    }

    private void sendWindow(Window window,boolean isFirst){
        //发送数据报
        window.startSendTime=System.currentTimeMillis();
        send(window.packet,isFirst);
    }

    private void send(TCP_PACKET stcpPack,boolean isFirst){
        //发送数据报
        client.send(stcpPack);
        if(isFirst){
            //logger.info("首次发送包:"+stcpPack.getTcpH().getTh_seq());
        }else{
            logger.warning("重新发送包:"+stcpPack.getTcpH().getTh_seq());
        }
    }
```



```java
    public void waitOvertime() {
        TimerTask dealOverTime = new TimerTask() {
            @Override
            public void run() {
                int index = startWindowIndex;
                boolean updateStart=true;
                Window window;
                while (index < endWindosIndex) {
                    // 如果第index个包超时了
                    window = sendContent.get(index);
                    if(updateStart && window.ack){
                        startWindowIndex=index+1;
                        logger.info("更新start值:"+startWindowIndex);
                    }else if(!window.ack){
                        updateStart=false;
                        if (TIMEOUTTIME < (System.currentTimeMillis() - window.getStartSendTime())) {
                            //  它没有收到ack,则尝试重发
                            sendWindow(window,false);
                        }
                    }
                    index++;
                }
            }
        };
        new Timer().schedule(dealOverTime, 0, 200);
```

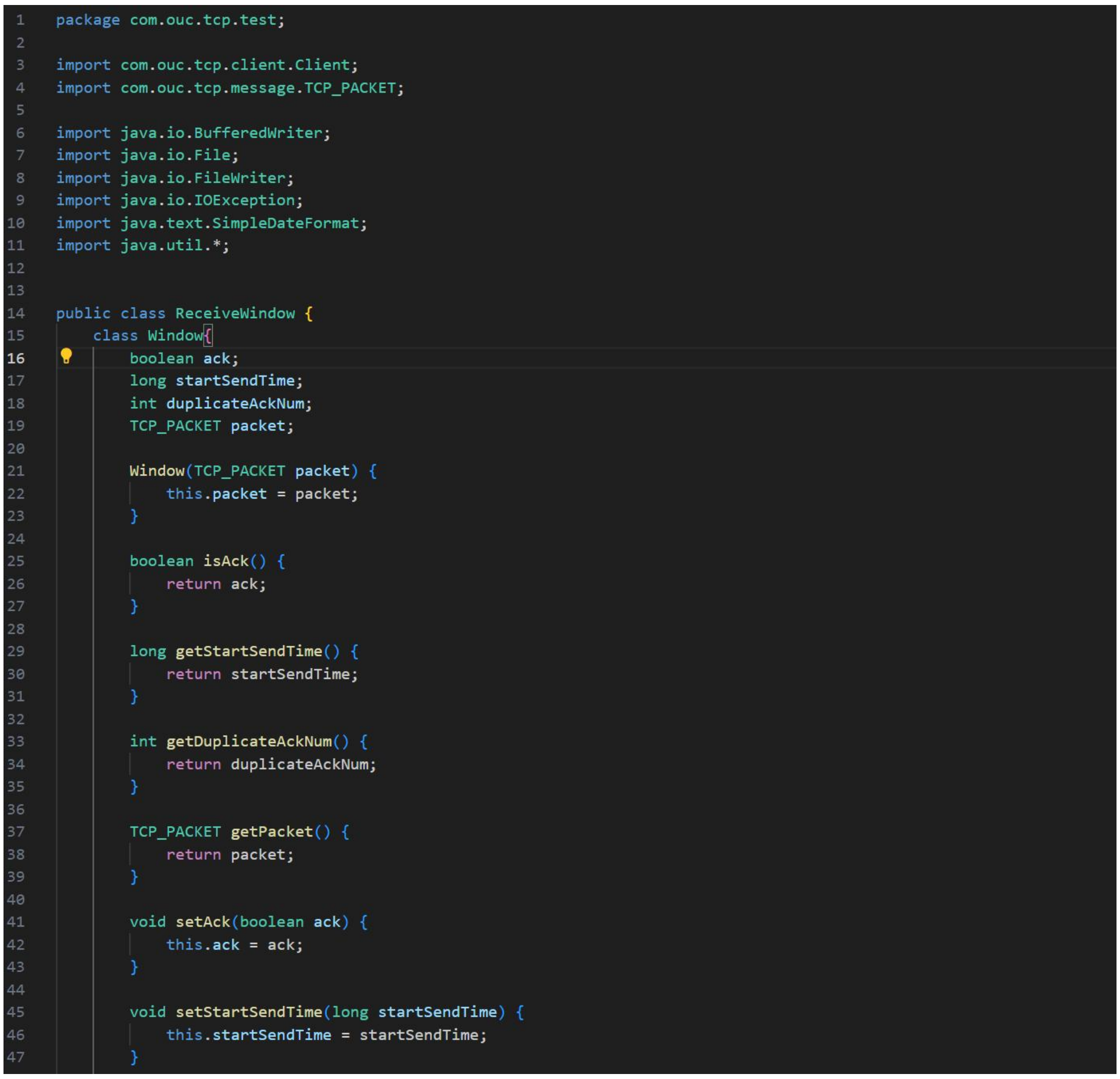


```java
    public void recv(TCP_PACKET recvPack){
        int ack=recvPack.getTcpH().getTh_ack();
        Window window;
        boolean canUpdate=true;
        int seq;
        for (int i = startWindowIndex; i <endWindosIndex ; i++) {
            window=sendContent.get(i);
            seq=window.packet.getTcpH().getTh_seq();
            if(seq<=ack){
                if(canUpdate && window.isAck()){
                    startWindowIndex=i+1;
                }else{
                    canUpdate=false;
                }

                if(seq==ack){
                    if(!window.isAck()){
//                      logger.info("接收到ack:"+ack+" index为:"+i+"的窗口块ack");
                    }else{
                        logger.info("重复接收到ack:"+ack+" index为:"+i+"的窗口块已经ack");
                    }

                    window.ack=true;
                    break;
                }
            }else{
                break;
            }
        }
    }
```

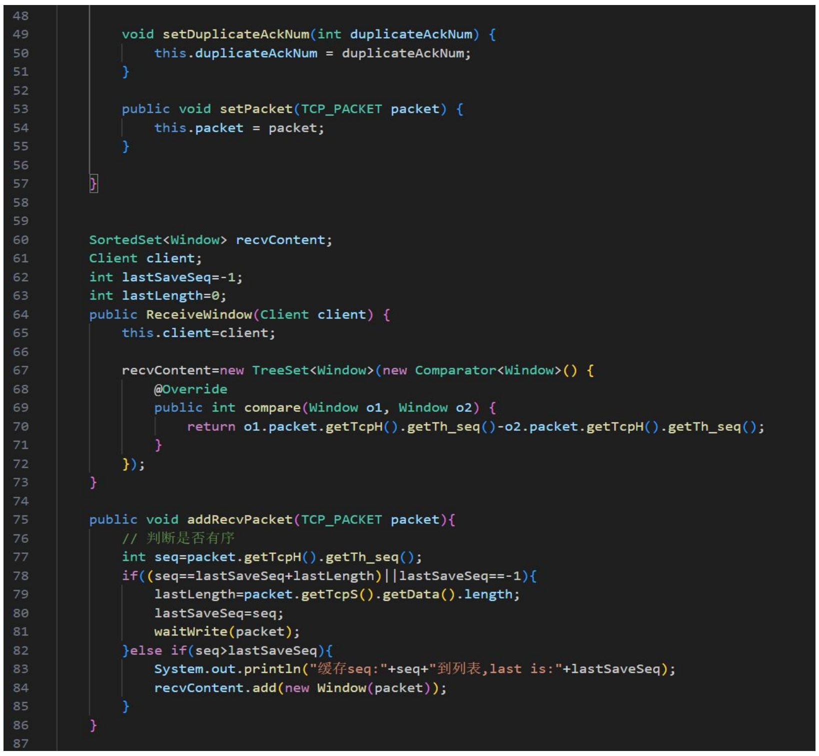
1.接收端滑动窗口

首先建立窗口，然后通过 ReceiveWindow 类来完成对数据包，ACK 的确认，初始化等操作。然后将包排序，判断包是否按序到达，如果未按序到达，则缓存靠后的包，同时向发送端进行请求，如果出错还要将错误的堆栈信息打印下来，如果无误就释放资源。

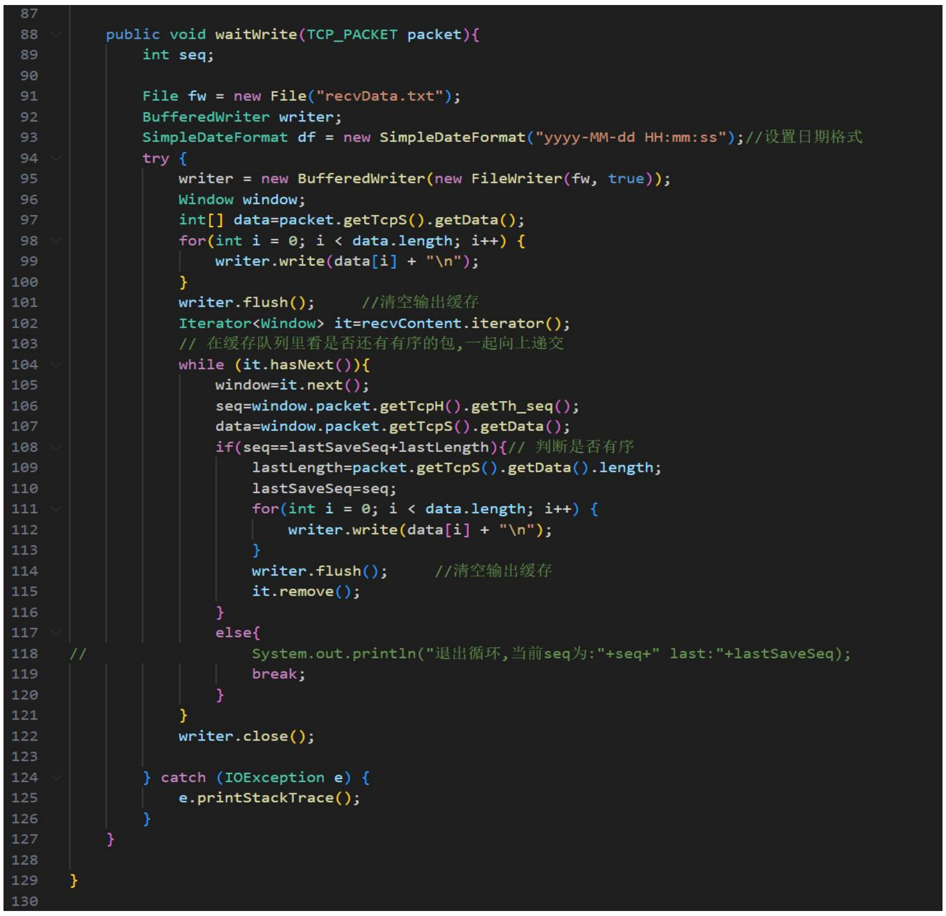
实现代码如下：



```java
package com.ouc.tcp.test;

import com.ouc.tcp.client.Client;
import com.ouc.tcp.message.TCP_PACKET;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.*;


public class ReceiveWindow {
    class Window{
        boolean ack;
        long startSendTime;
        int duplicateAckNum;
        TCP_PACKET packet;

        Window(TCP_PACKET packet) {
            this.packet = packet;
        }

        boolean isAck() {
            return ack;
        }

        long getStartSendTime() {
            return startSendTime;
        }

        int getDuplicateAckNum() {
            return duplicateAckNum;
        }

        TCP_PACKET getPacket() {
            return packet;
        }

        void setAck(boolean ack) {
            this.ack = ack;
        }

        void setStartSendTime(long startSendTime) {
            this.startSendTime = startSendTime;
        }
```



```java

        void setDuplicateAckNum(int duplicateAckNum) {
            this.duplicateAckNum = duplicateAckNum;
        }

        public void setPacket(TCP_PACKET packet) {
            this.packet = packet;
        }

    }


    SortedSet<Window> recvContent;
    Client client;
    int lastSaveSeq=-1;
    int lastLength=0;
    public ReceiveWindow(Client client) {
        this.client=client;

        recvContent=new TreeSet<Window>(new Comparator<Window>() {
            @Override
            public int compare(Window o1, Window o2) {
                return o1.packet.getTcpH().getTh_seq()-o2.packet.getTcpH().getTh_seq();
            }
        });
    }

    public void addRecvPacket(TCP_PACKET packet){
        // 判断是否有序
        int seq=packet.getTcpH().getTh_seq();
        if((seq==lastSaveSeq+lastLength)||lastSaveSeq==-1){
            lastLength=packet.getTcpS().getData().length;
            lastSaveSeq=seq;
            waitWrite(packet);
        }else if(seq>lastSaveSeq){
            System.out.println("缓存seq:"+seq+"到列表,last is:"+lastSaveSeq);
            recvContent.add(new Window(packet));
        }
    }

```

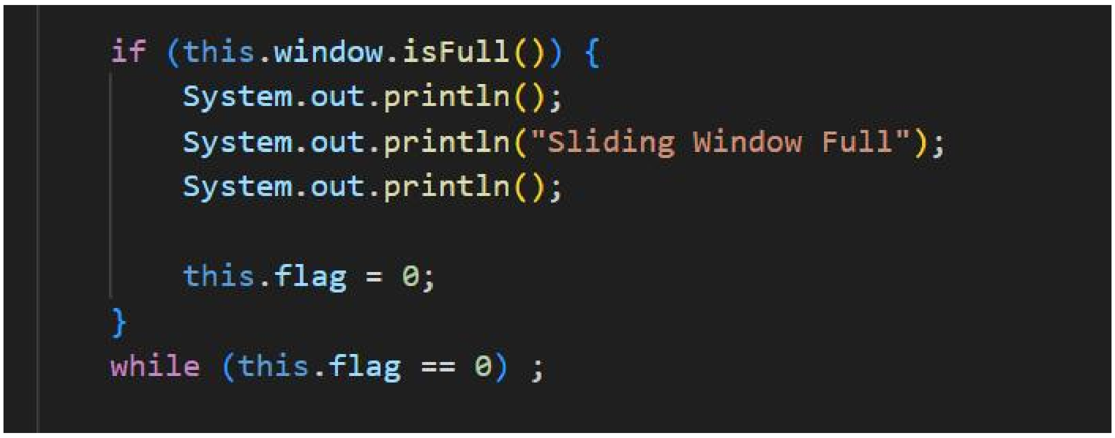


```java

    public void waitWrite(TCP_PACKET packet){
        int seq;

        File fw = new File("recvData.txt");
        BufferedWriter writer;
        SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");//设置日期格式
        try {
            writer = new BufferedWriter(new FileWriter(fw, true));
            Window window;
            int[] data=packet.getTcpS().getData();
            for(int i = 0; i < data.length; i++) {
                writer.write(data[i] + "\n");
            }
            writer.flush();     //清空输出缓存
            Iterator<Window> it=recvContent.iterator();
            // 在缓存队列里看是否还有有序的包,一起向上递交
            while (it.hasNext()){
                window=it.next();
                seq=window.packet.getTcpH().getTh_seq();
                data=window.packet.getTcpS().getData();
                if(seq==lastSaveSeq+lastLength){// 判断是否有序
                    lastLength=packet.getTcpS().getData().length;
                    lastSaveSeq=seq;
                    for(int i = 0; i < data.length; i++) {
                        writer.write(data[i] + "\n");
                    }
                    writer.flush();     //清空输出缓存
                    it.remove();
                }
                else{
//                    System.out.println("退出循环,当前seq为:"+seq+" last:"+lastSaveSeq);
                    break;
                }
            }
            writer.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}

```

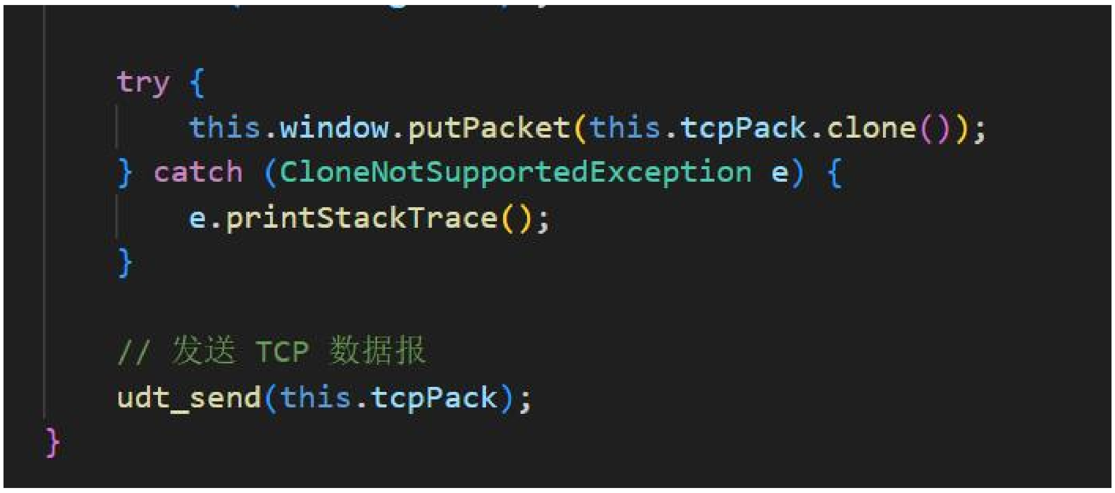
2.Sender.java（发送端修改）

首先，在 rdt_send()函数中，我们要判断发送窗口是不是已满的，调用 sendwindow 中的 isFull()函数，如果满了则将 flag 置为 0，同时打印 Sliding Window Full

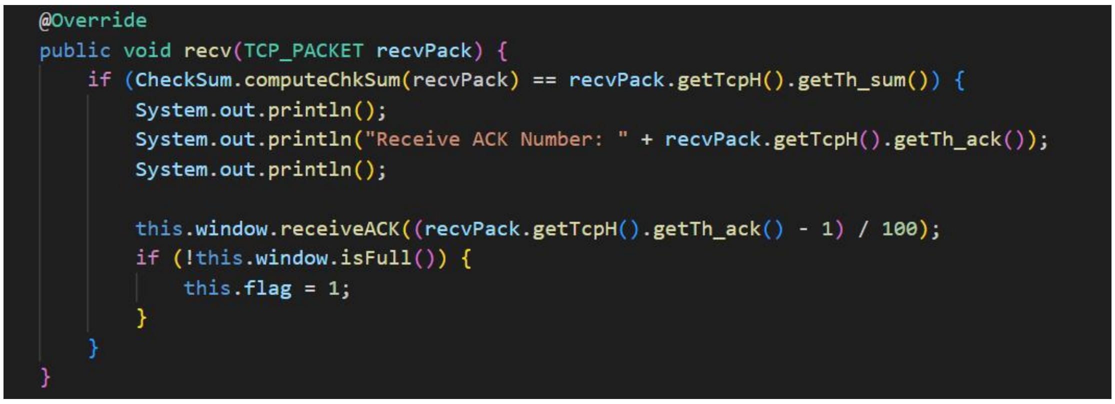
具体代码如下图：



```java
if (this.window.isFull()) {
    System.out.println();
    System.out.println("Sliding Window Full");
    System.out.println();

    this.flag = 0;
}
while (this.flag == 0) ;
```



```java
    try {
        this.window.putPacket(this.tcpPack.clone());
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }

    // 发送 TCP 数据报
    udt_send(this.tcpPack);
}
```



```java
@Override
public void recv(TCP_PACKET recvPack) {
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
        System.out.println();
        System.out.println("Receive ACK Number: " + recvPack.getTcpH().getTh_ack());
        System.out.println();

        this.window.receiveACK((recvPack.getTcpH().getTh_ack() - 1) / 100);
        if (!this.window.isFull()) {
            this.flag = 1;
        }
    }
}
```

日志分析：

可以看到 10101 丢失后，后续只重传 10101

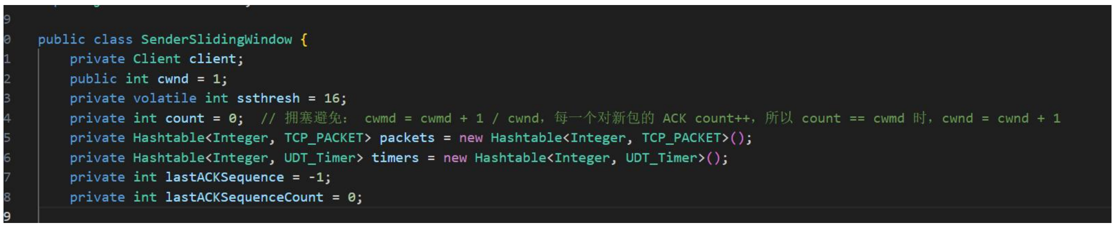
```
2026-01-12 20:20:18:984 CSTDATA_seq: 10001            ACKed
2026-01-12 20:20:18:998 CSTDATA_seq: 10101   LOSS     NO_ACK
2026-01-12 20:20:19:010 CSTDATA_seq: 10201            ACKed
2026-01-12 20:20:19:026 CSTDATA_seq: 10301            ACKed
2026-01-12 20:20:19:037 CSTDATA_seq: 10401            ACKed
2026-01-12 20:20:19:048 CSTDATA_seq: 10501            ACKed
2026-01-12 20:20:19:068 CSTDATA_seq: 10601            ACKed
2026-01-12 20:20:19:093 CSTDATA_seq: 10701            ACKed
2026-01-12 20:20:19:109 CSTDATA_seq: 10801            ACKed
2026-01-12 20:20:19:124 CSTDATA_seq: 10901            ACKed
2026-01-12 20:20:19:140 CSTDATA_seq: 11001            ACKed
2026-01-12 20:20:19:151 CSTDATA_seq: 11101            ACKed
2026-01-12 20:20:19:167 CSTDATA_seq: 11201            ACKed
2026-01-12 20:20:19:182 CSTDATA_seq: 11301            ACKed
2026-01-12 20:20:19:195 CSTDATA_seq: 11401            NO_ACK
2026-01-12 20:20:19:211 CSTDATA_seq: 11501            ACKed
2026-01-12 20:20:19:227 CSTDATA_seq: 11601            ACKed
2026-01-12 20:20:22:011 CST*Re: DATA_seq: 10101                 ACKed
2026-01-12 20:20:22:014 CSTDATA_seq: 11701            ACKed
2026-01-12 20:20:22:027 CSTDATA_seq: 11801            ACKed
```

（6）TCP_Tahoe

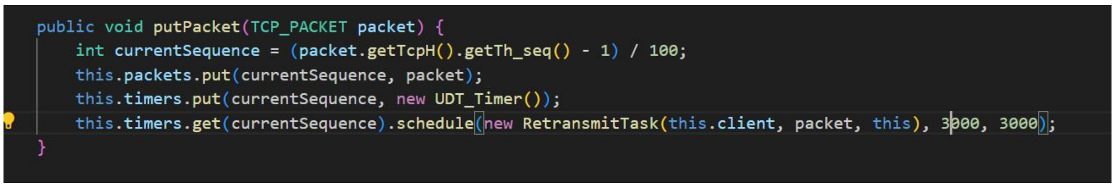
TCP_Tahoe 实现了流量控制和拥塞窗口，主要有三个机制: slow start (慢开始), congestion avoidance (拥塞避免), and fast retransmit(快重传)。

1. 发送窗口修改

通过哈希表来储存待发送的 TCP 数据包，来用于快速查找和管理发送的包，因为每个包都可以通过其序列号进行索引。

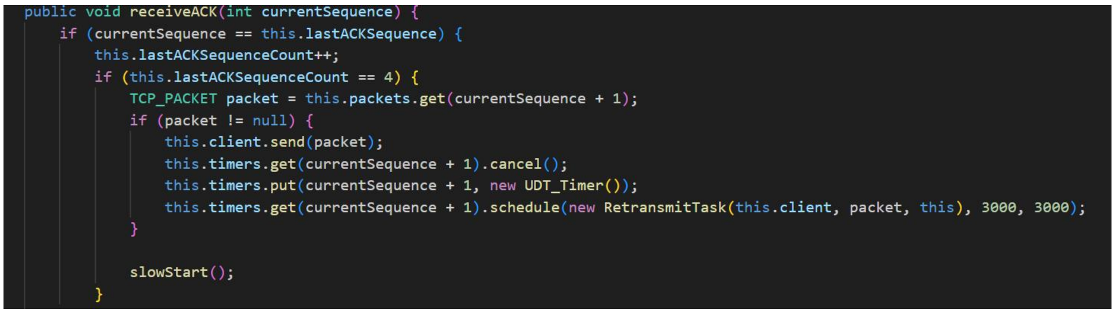


```java
public class SenderSlidingWindow {
    private Client client;
    public int cwnd = 1;
    private volatile int ssthresh = 16;
    private int count = 0;  // 拥塞避免:  cwmd = cwmd + 1 / cwnd, 每一个对新包的 ACK count++, 所以 count == cwmd 时, cwnd = cwnd + 1
    private Hashtable<Integer, TCP_PACKET> packets = new Hashtable<Integer, TCP_PACKET>();
    private Hashtable<Integer, UDT_Timer> timers = new Hashtable<Integer, UDT_Timer>();
    private int lastACKSequence = -1;
    private int lastACKSequenceCount = 0;
```



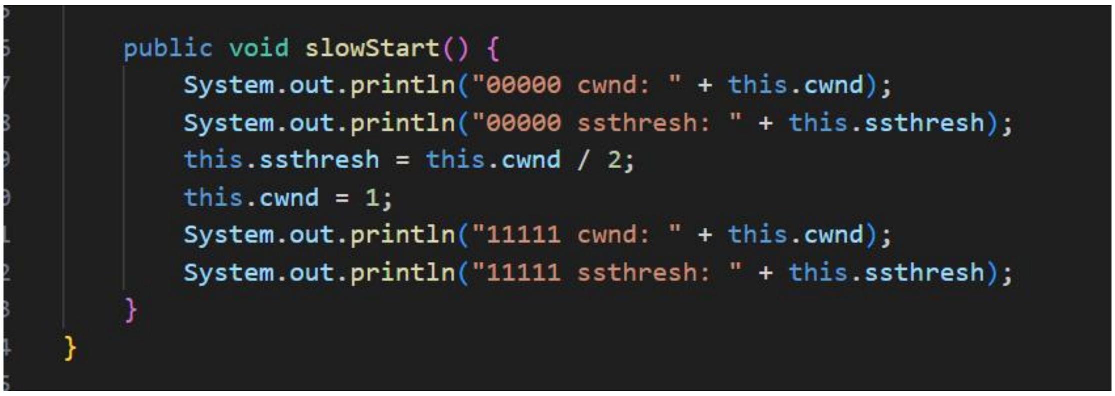
```java
public void putPacket(TCP_PACKET packet) {
    int currentSequence = (packet.getTcpH().getTh_seq() - 1) / 100;
    this.packets.put(currentSequence, packet);
    this.timers.put(currentSequence, new UDT_Timer());
    this.timers.get(currentSequence).schedule(new RetransmitTask(this.client, packet, this), 3000, 3000);
}
```

快重传的实现方式：声明一个 lastACKSequenceCount，如果收到重复 ACK，则 lastACKSequenceCount++， 如果 lastACKSequenceCount= 4（初值为 1），代表连续收到 3 个重复 ACK，执行快重传。



```java
public void receiveACK(int currentSequence) {
    if (currentSequence == this.lastACKSequence) {
        this.lastACKSequenceCount++;
        if (this.lastACKSequenceCount == 4) {
            TCP_PACKET packet = this.packets.get(currentSequence + 1);
            if (packet != null) {
                this.client.send(packet);
                this.timers.get(currentSequence + 1).cancel();
                this.timers.put(currentSequence + 1, new UDT_Timer());
                this.timers.get(currentSequence + 1).schedule(new RetransmitTask(this.client, packet, this), 3000, 3000);
            }

            slowStart();
        }
```

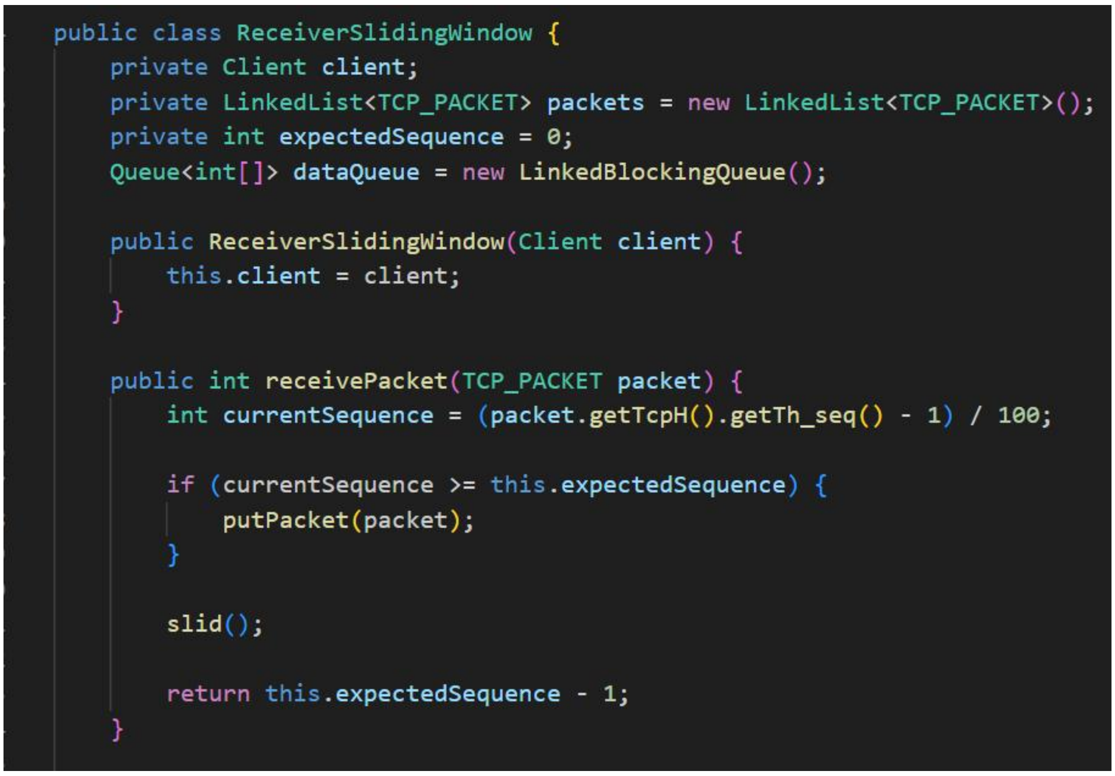
慢开始，快恢复实现方式：每一次接收到 ACK，就使 cwnd ++。快恢复为连续收到 3 个重复 ACK 后，要进行快重传，同时，发送方知道现在只是丢失了个别的报文段，于是调整门限 ssthresh = cwnd / 2，同时设置 cwnd =1，并开始执行拥塞避免阶段。



```java
    public void slowStart() {
        System.out.println("00000 cwnd: " + this.cwnd);
        System.out.println("00000 ssthresh: " + this.ssthresh);
        this.ssthresh = this.cwnd / 2;
        this.cwnd = 1;
        System.out.println("11111 cwnd: " + this.cwnd);
        System.out.println("11111 ssthresh: " + this.ssthresh);
    }
}
```

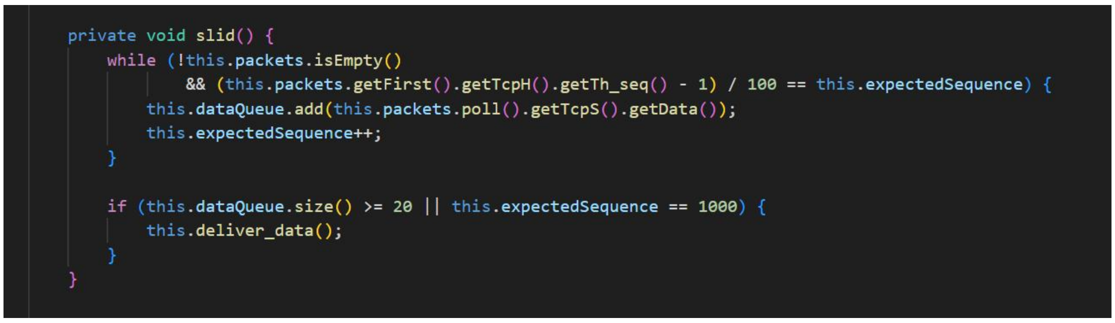
2. 接收方窗口修改

接收方窗口要将失序变成有序，所以每收到一个包，就要根据他的 ACK 号，插入对应位置。



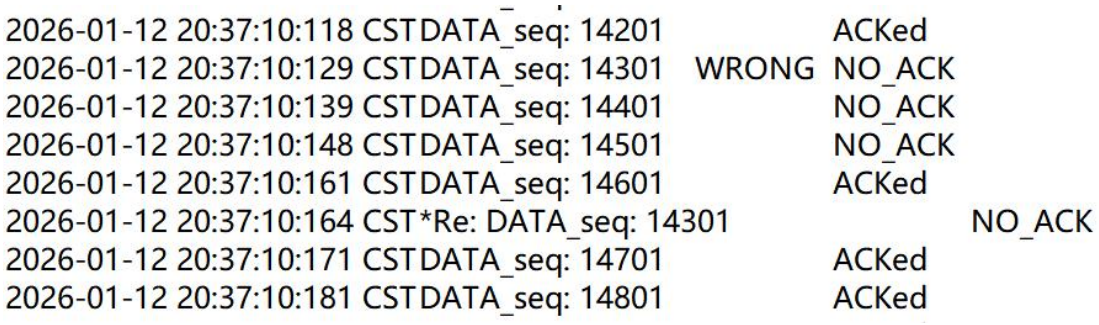
```java
public class ReceiverSlidingWindow {
    private Client client;
    private LinkedList<TCP_PACKET> packets = new LinkedList<TCP_PACKET>();
    private int expectedSequence = 0;
    Queue<int[]> dataQueue = new LinkedBlockingQueue();

    public ReceiverSlidingWindow(Client client) {
        this.client = client;
    }

    public int receivePacket(TCP_PACKET packet) {
        int currentSequence = (packet.getTcpH().getTh_seq() - 1) / 100;

        if (currentSequence >= this.expectedSequence) {
            putPacket(packet);
        }

        slid();

        return this.expectedSequence - 1;
    }
```

修改 slid()滑动窗口函数：正常情况为收到一个包，加入链表，从链表删除，加入 data 队列；非正常情况则需要缓冲未发送的包，到时候一次全部发送。



```java
private void slid() {
    while (!this.packets.isEmpty()
            && (this.packets.getFirst().getTcpH().getTh_seq() - 1) / 100 == this.expectedSequence) {
        this.dataQueue.add(this.packets.poll().getTcpS().getData());
        this.expectedSequence++;
    }

    if (this.dataQueue.size() >= 20 || this.expectedSequence == 1000) {
        this.deliver_data();
    }
}
```

日志文件：

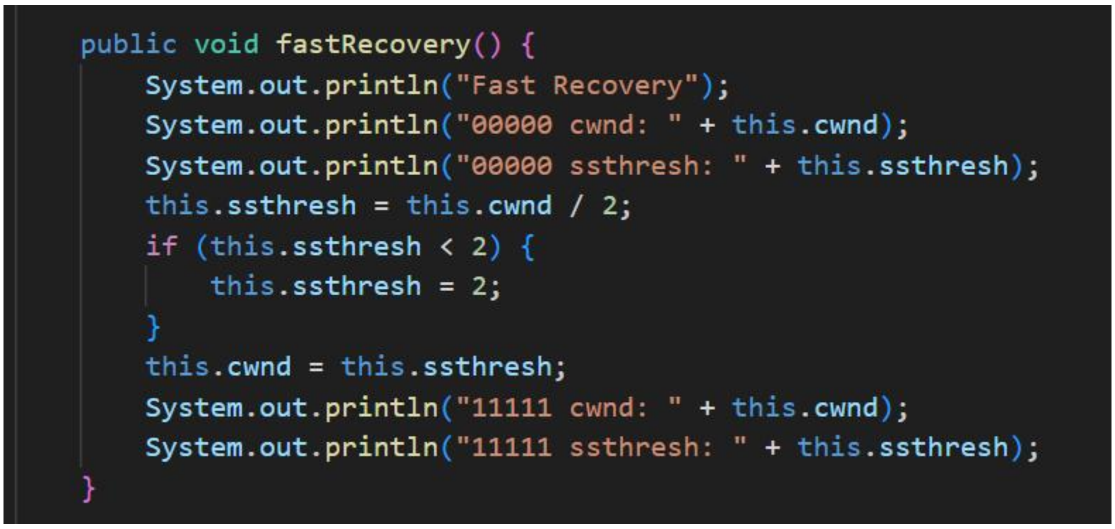


```
2026-01-12 20:37:10:118 CSTDATA_seq: 14201          ACKed
2026-01-12 20:37:10:129 CSTDATA_seq: 14301   WRONG  NO_ACK
2026-01-12 20:37:10:139 CSTDATA_seq: 14401          NO_ACK
2026-01-12 20:37:10:148 CSTDATA_seq: 14501          NO_ACK
2026-01-12 20:37:10:161 CSTDATA_seq: 14601          ACKed
2026-01-12 20:37:10:164 CST*Re: DATA_seq: 14301               NO_ACK
2026-01-12 20:37:10:171 CSTDATA_seq: 14701          ACKed
2026-01-12 20:37:10:181 CSTDATA_seq: 14801          ACKed
```

（7）TCP_Reno

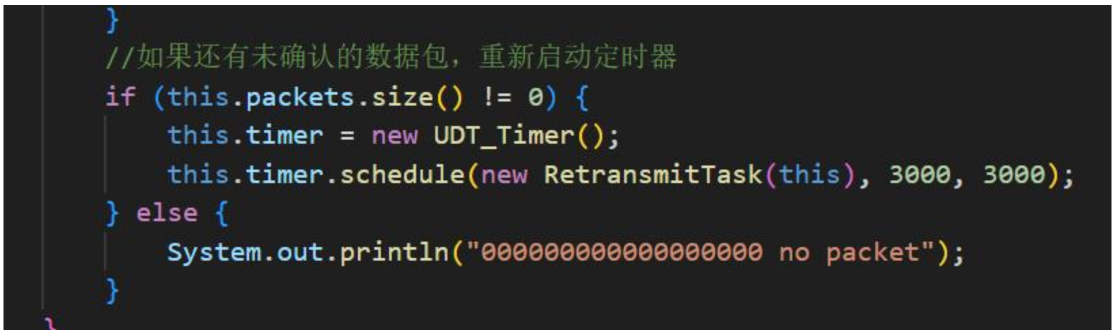
TCP_Reno 需要在 TCP_Tahoe 的基础上增加快恢复（快重传在上一个版本已经完成），以及只使用一个计时器。

下面是快恢复的实现代码：



```java
public void fastRecovery() {
    System.out.println("Fast Recovery");
    System.out.println("00000 cwnd: " + this.cwnd);
    System.out.println("00000 ssthresh: " + this.ssthresh);
    this.ssthresh = this.cwnd / 2;
    if (this.ssthresh < 2) {
        this.ssthresh = 2;
    }
    this.cwnd = this.ssthresh;
    System.out.println("11111 cwnd: " + this.cwnd);
    System.out.println("11111 ssthresh: " + this.ssthresh);
}
```

定时器：

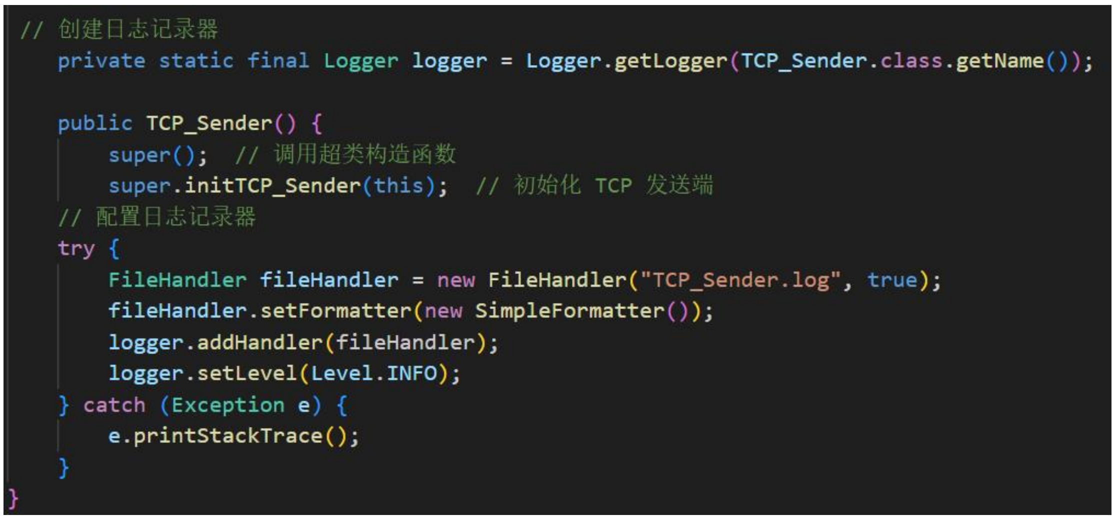


```java
    }
    //如果还有未确认的数据包，重新启动定时器
    if (this.packets.size() != 0) {
        this.timer = new UDT_Timer();
        this.timer.schedule(new RetransmitTask(this), 3000, 3000);
    } else {
        System.out.println("000000000000000000 no packet");
    }
```

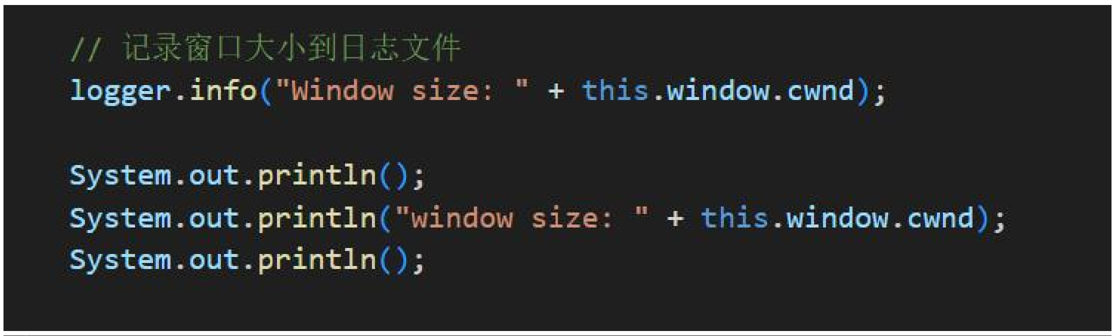
为了方便观察窗口大小变化，我创建了两个新的 log 文件来记录

具体实现函数如下：

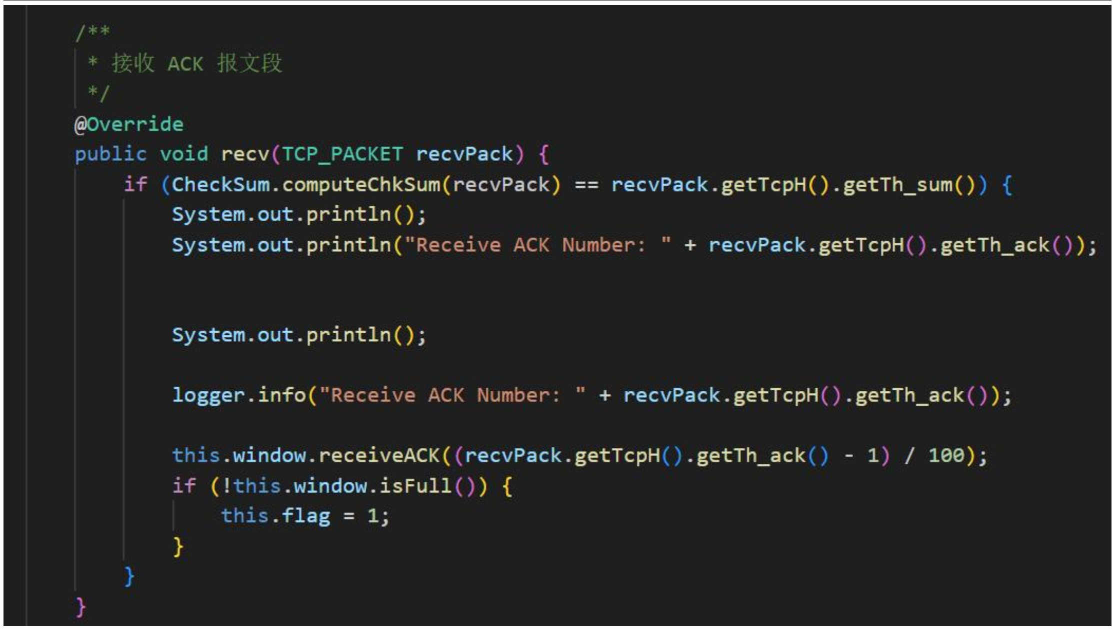
在 TCP_Sender.java 文件中



```java
// 创建日志记录器
    private static final Logger logger = Logger.getLogger(TCP_Sender.class.getName());

    public TCP_Sender() {
        super();  // 调用超类构造函数
        super.initTCP_Sender(this);  // 初始化 TCP 发送端
    // 配置日志记录器
    try {
        FileHandler fileHandler = new FileHandler("TCP_Sender.log", true);
        fileHandler.setFormatter(new SimpleFormatter());
        logger.addHandler(fileHandler);
        logger.setLevel(Level.INFO);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

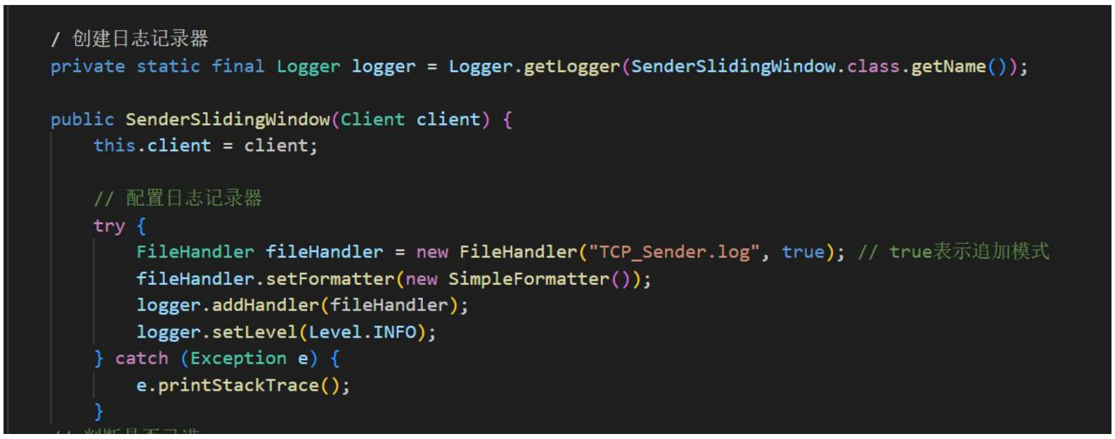


```java
// 记录窗口大小到日志文件
logger.info("Window size: " + this.window.cwnd);

System.out.println();
System.out.println("window size: " + this.window.cwnd);
System.out.println();
```

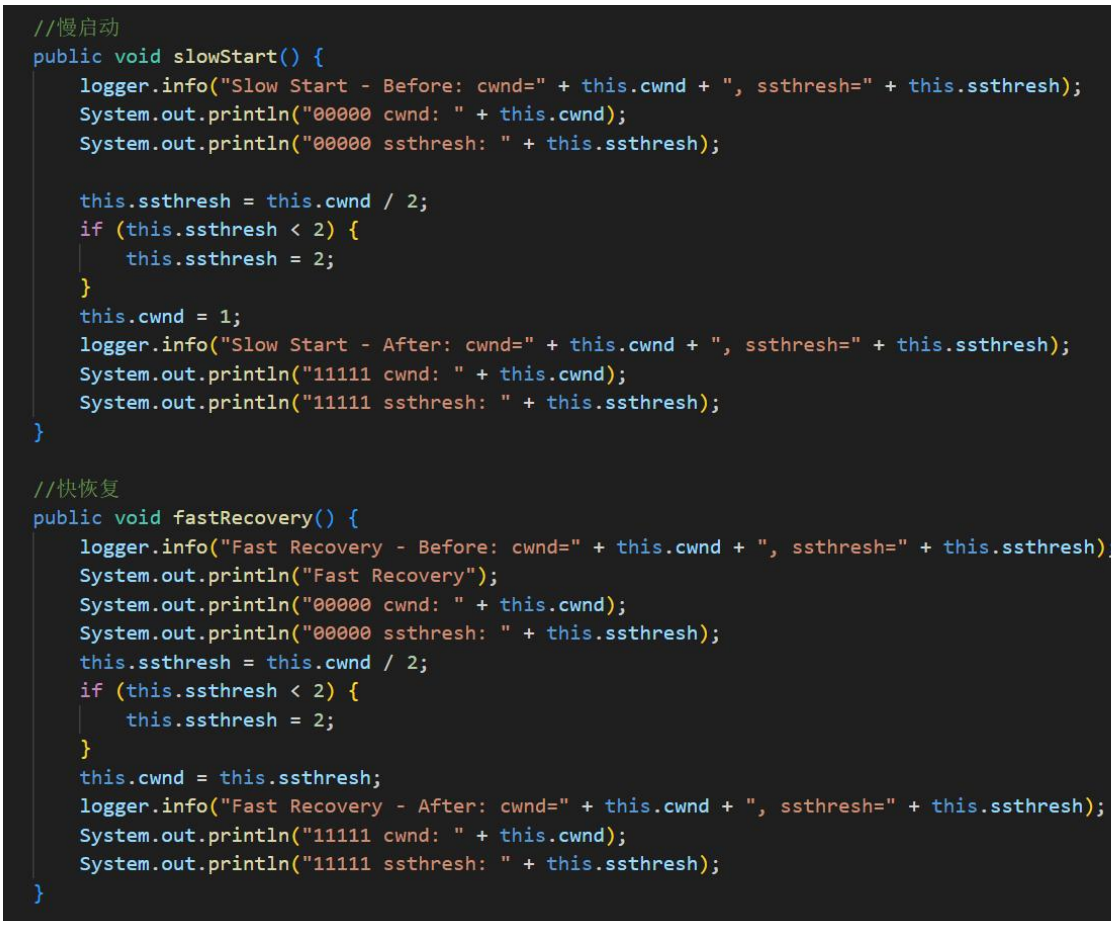


```java
/**
 * 接收 ACK 报文段
 */
@Override
public void recv(TCP_PACKET recvPack) {
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
        System.out.println();
        System.out.println("Receive ACK Number: " + recvPack.getTcpH().getTh_ack());


        System.out.println();

        logger.info("Receive ACK Number: " + recvPack.getTcpH().getTh_ack());

        this.window.receiveACK((recvPack.getTcpH().getTh_ack() - 1) / 100);
        if (!this.window.isFull()) {
            this.flag = 1;
        }
    }
}
```

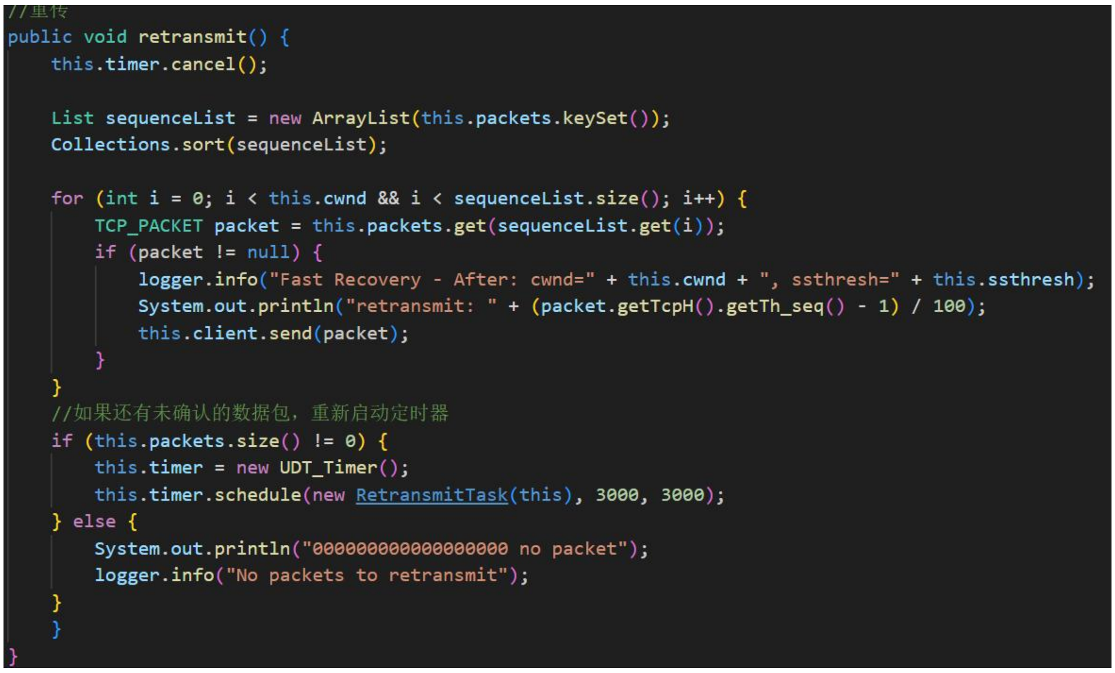
在发送端滑动窗口实现方法如下：
无论是在控制台还是在 log 日志文件中都记录下来。



```java
/ 创建日志记录器
private static final Logger logger = Logger.getLogger(SenderSlidingWindow.class.getName());

public SenderSlidingWindow(Client client) {
    this.client = client;

    // 配置日志记录器
    try {
        FileHandler fileHandler = new FileHandler("TCP_Sender.log", true); // true表示追加模式
        fileHandler.setFormatter(new SimpleFormatter());
        logger.addHandler(fileHandler);
        logger.setLevel(Level.INFO);
    } catch (Exception e) {
        e.printStackTrace();
    }
```

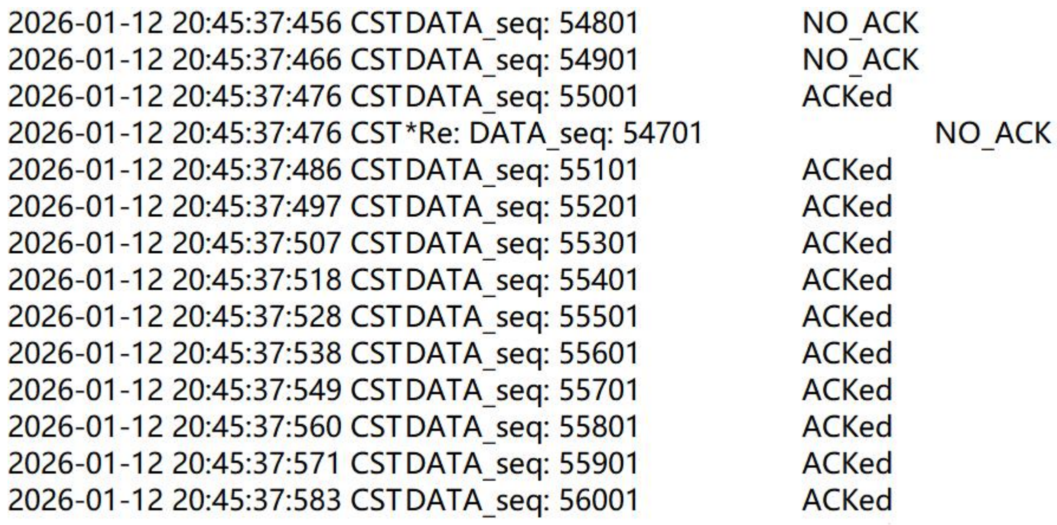


```java
//慢启动
public void slowStart() {
    logger.info("Slow Start - Before: cwnd=" + this.cwnd + ", ssthresh=" + this.ssthresh);
    System.out.println("00000 cwnd: " + this.cwnd);
    System.out.println("00000 ssthresh: " + this.ssthresh);

    this.ssthresh = this.cwnd / 2;
    if (this.ssthresh < 2) {
        this.ssthresh = 2;
    }
    this.cwnd = 1;
    logger.info("Slow Start - After: cwnd=" + this.cwnd + ", ssthresh=" + this.ssthresh);
    System.out.println("11111 cwnd: " + this.cwnd);
    System.out.println("11111 ssthresh: " + this.ssthresh);
}

//快恢复
public void fastRecovery() {
    logger.info("Fast Recovery - Before: cwnd=" + this.cwnd + ", ssthresh=" + this.ssthresh)
    System.out.println("Fast Recovery");
    System.out.println("00000 cwnd: " + this.cwnd);
    System.out.println("00000 ssthresh: " + this.ssthresh);
    this.ssthresh = this.cwnd / 2;
    if (this.ssthresh < 2) {
        this.ssthresh = 2;
    }
    this.cwnd = this.ssthresh;
    logger.info("Fast Recovery - After: cwnd=" + this.cwnd + ", ssthresh=" + this.ssthresh);
    System.out.println("11111 cwnd: " + this.cwnd);
    System.out.println("11111 ssthresh: " + this.ssthresh);
}
```



```java
//重传
public void retransmit() {
    this.timer.cancel();

    List sequenceList = new ArrayList(this.packets.keySet());
    Collections.sort(sequenceList);

    for (int i = 0; i < this.cwnd && i < sequenceList.size(); i++) {
        TCP_PACKET packet = this.packets.get(sequenceList.get(i));
        if (packet != null) {
            logger.info("Fast Recovery - After: cwnd=" + this.cwnd + ", ssthresh=" + this.ssthresh);
            System.out.println("retransmit: " + (packet.getTcpH().getTh_seq() - 1) / 100);
            this.client.send(packet);
        }
    }
    //如果还有未确认的数据包，重新启动定时器
    if (this.packets.size() != 0) {
        this.timer = new UDT_Timer();
        this.timer.schedule(new RetransmitTask(this), 3000, 3000);
    } else {
        System.out.println("000000000000000000 no packet");
        logger.info("No packets to retransmit");
    }
    }
}
```

日志截图



```
2026-01-12 20:45:37:456 CSTDATA_seq: 54801              NO_ACK
2026-01-12 20:45:37:466 CSTDATA_seq: 54901              NO_ACK
2026-01-12 20:45:37:476 CSTDATA_seq: 55001              ACKed
2026-01-12 20:45:37:476 CST*Re: DATA_seq: 54701                  NO_ACK
2026-01-12 20:45:37:486 CSTDATA_seq: 55101              ACKed
2026-01-12 20:45:37:497 CSTDATA_seq: 55201              ACKed
2026-01-12 20:45:37:507 CSTDATA_seq: 55301              ACKed
2026-01-12 20:45:37:518 CSTDATA_seq: 55401              ACKed
2026-01-12 20:45:37:528 CSTDATA_seq: 55501              ACKed
2026-01-12 20:45:37:538 CSTDATA_seq: 55601              ACKed
2026-01-12 20:45:37:549 CSTDATA_seq: 55701              ACKed
2026-01-12 20:45:37:560 CSTDATA_seq: 55801              ACKed
2026-01-12 20:45:37:571 CSTDATA_seq: 55901              ACKed
2026-01-12 20:45:37:583 CSTDATA_seq: 56001              ACKed
```

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）

所有内容均已完成

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)

（1） 可以灵活的修改，并且如果出现问题可以随时退回上一个版本，进行重写。

（2） 难度由易到难，可以让学生更加有成就感，在不断的迭代的过程中，学生可以获得更多的成就感，激发学生的学习兴趣。

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)

（1）只使用一个定时器的问题：

解决方法：一开始我不知道如何使用一个定时器完成实验，后来发现只需要每次发送完重新设置定时器即可，将所有的超时事件统一管理即可。

（2）窗口变化的可视化问题。控制台的信息有限而且比较复杂，每次运行之后信息的可视化都不太好

解决方法：我在发送端分别创建了两个 log 日志来记录窗口的变化和特殊情况下窗口的

变化。

（3）数据报缓存处理问题：我在一开始对于这个问题是没有头绪的，因为接收端需要按照数据包的序列号顺序处理数据包。如果数据包乱序到达，则需要缓存乱序的数据包，直到收到缺失的数据包。

解决方法：使用 LinkedList<TCP_PACKET> 缓存接收到的数据包。在接收到数据包时，检查其序列号是否与序列号匹配。如果匹配，则处理该数据包，并递增序列号；如果不匹配，则将数据包缓存到 LinkedList 中，等待缺失的数据包到达。

5. 对于实验系统提出问题或建议(1 分)

（1）jdk 版本过于老旧，希望可以使用更新的 jdk21、jdk17
（2）最后两个版本难度有些大，但是占比分数很多，或许可以把最后一两个版本当成附加题的形式？分数占比低一些