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。（17 分）

（1）RDT2.1

首先通过计算校验和实现了 RDT2.0 的检验位错机制

```
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());
        crc.update(Arrays.toString(tcpPack.getTcpS().getData()).getBytes());
        //计算校验和
        checkSum = (short) crc.getValue();
        return checkSum;
    }
}
```

```
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 Wrong ACK Number");
        ackQueue.add(-1);
    }
    System.out.println();
    //处理ACK报文
    //waitACK();
}
```

```
-> 2020-12-28 18:55:16:331 CST
** TCP_Sender
   Receive packet from: [192.168.43.98:9002]
   Packet data:
   PACKET_TYPE: ACK_-389257345
Receive Wrong ACK Number：
```

其次发现 log 文件中在出错后重传的数据会重复

```
2020-12-28 18:41:42:884 CST *Re: DATA_seq: 21701        ACKed
2020-12-28 18:41:42:884 CST *Re: DATA_seq: 21701        ACKed
2020-12-28 18:41:42:885 CST *Re: DATA_seq: 21701        ACKed
2020-12-28 18:41:42:885 CST DATA_seq: 21801     ACKed
2020-12-28 18:41:42:885 CST *Re: DATA_seq: 21801        ACKed
2020-12-28 18:41:42:886 CST *Re: DATA_seq: 21801        ACKed
2020-12-28 18:41:42:886 CST *Re: DATA_seq: 21801        ACKed
2020-12-28 18:41:42:887 CST *Re: DATA_seq: 21801        ACKed
2020-12-28 18:41:42:887 CST *Re: DATA_seq: 21801        ACKed
```

解决方法为在发送端为包的每个分组添加序号

```
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();
        } else {
            System.out.println("收到重复包,重复seq:" + 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);
    }
```

并在接收端添加防重复机制

```
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();
        } else {
            System.out.println("收到重复包,重复seq:" + sequence);
        }
```

错误的包能够重新发包，包重复的问题也能够检测到并进行处理，问题得到了

解决

```
2020-12-28 18:55:19:239 CST DATA_seq: 68901     ACKed
2020-12-28 18:55:19:249 CST DATA_seq: 69001 WRONG   NO_ACK
2020-12-28 18:55:19:250 CST *Re: DATA_seq: 69001        ACKed
2020-12-28 18:55:19:260 CST DATA_seq: 69101     ACKed
```

```
** TCP_Receiver
   Receive packet from: [192.168.43.98:9001]
   Packet data: 139907 139921 139939 139943 1
 140191 140197 140207 140221 140227 140237 14
  140449 140453 140473 140477 140521 140527 1
 140717 140729 140731 140741 140759 140761 14
  140989 141023 141041 141061
   PACKET_TYPE: DATA_SEQ_13001
收到重复包,重复seq:13001
```

（2）RDT2.2

接收方正确接收一个包后，发送 ACK，在 ACK 包中，接收方必须通过序号指明是对哪个数据包的确认。
接收方需要记录上次接收的包的 seq 值,若与本次接收的相同,则不能将它插入 data 队列。

```
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();
        } else {
            System.out.println("收到重复包,重复seq:" + 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);
    }
```

发送端收到发生位错误的 ack 包时,认为接收方没有正确收到该包,故重复发送本次包

```java
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 Wrong ACK Number");
        ackQueue.add(-1);
    }
    System.out.println();
    //处理ACK报文
    //waitACK();
}
```

Log 文件得出 WRONG NAK 的包都能正确重发包

```
2020-12-28 18:55:19:239 CST DATA_seq: 68901     ACKed
2020-12-28 18:55:19:249 CST DATA_seq: 69001 WRONG   NO_ACK
2020-12-28 18:55:19:250 CST *Re: DATA_seq: 69001        ACKed
2020-12-28 18:55:19:260 CST DATA_seq: 69101     ACKed
```

（3）RDT3.0

因为通道上除了出错还可能会发生丢包，所以需要一定的方法来处理数据丢失。
计时器 UDT_Timer 控制一个独立于发送方主线程的重传任务线程，以固定周期进行重传任务。

```java
//设置计时器和超时重传任务
UDT_Timer timer = new UDT_Timer();
UDT_RetransTask reTrans = new UDT_RetransTask(client, tcpPack);
//每隔3秒执行重传，直到收到ACK
timer.schedule(reTrans,  delay: 3000,  period: 3000);
```

对应的，已经成功传输数据包的计时器必须手动结束

```java
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());
                flag = 1;
                //停止等待时需关闭计时器
                System.out.println("关闭计时器");
                timer.cancel();
                break;
```

就此，LOSS 丢包皆能重发包，解决了丢包问题

```
2020-12-28 20:37:13:980 CST DATA_seq: 51901     ACKed
2020-12-28 20:37:13:991 CST DATA_seq: 52001 LOSS    NO_ACK
2020-12-28 20:37:16:992 CST *Re: DATA_seq: 52001        ACKed
2020-12-28 20:37:17:003 CST DATA_seq: 52101     ACKed
```

（4）Selective-Response

选择相应协议相比 RDT3.0 有极大变化，它将很多方法的实现都转移至在程序运行时构造的发送或接收窗口进行。包括对发送数据报，接收数据包判断，recvdata 文件写入，超时重传等功能进行进一步改动和封装。

判断数据报是否失序的改动：

```java
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("失序接收，缓存seq:" + seq + "到列表,last is:" + lastSaveSeq);
        recvContent.add(new Window(packet));
    }
}
```

若失序，则缓存在一个有序集合中，若有序，则执行 waitWrite 函数，将此包与缓存的其他有序包一同写入到 recvdata.txt 中

```java
public void waitWrite(TCP_PACKET packet) {
    int seq;
    File fw = new File( pathname: "recvData.txt");
    BufferedWriter writer;
    try {
        writer = new BufferedWriter(new FileWriter(fw,  append: true));
        Window window;
        int[] data = packet.getTcpS().getData();
        for (int i : data) writer.write( str: 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 : data) writer.write( str: i + "\n");
                writer.flush();
                it.remove();
            } else break;
        }
        writer.close();
```

发送端发包改动：

主线程

```java
//发送TCP数据报
try {
    this.sendWindow.RdtSend(tcpPack);
} catch (CloneNotSupportedException e) {
    e.printStackTrace();
}
while (!sendWindow.continueSend()){
    try {
        Thread.sleep( millis: 10);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

窗口

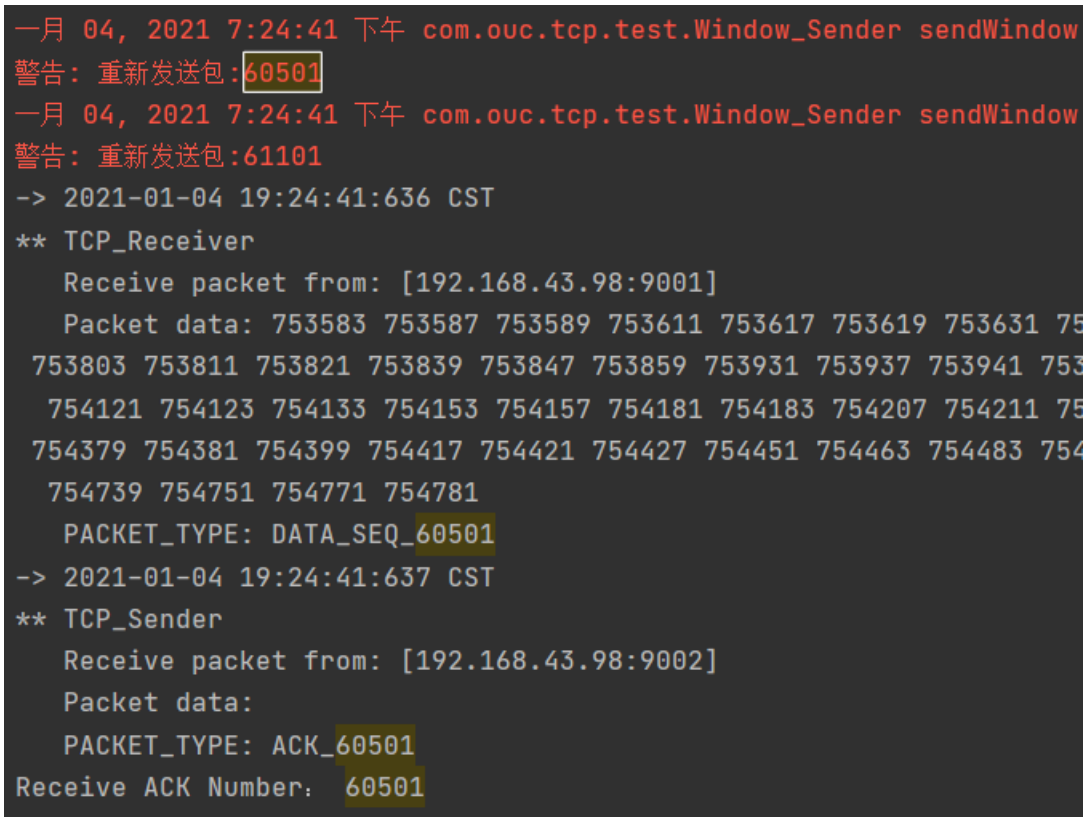
```java
public void RdtSend(TCP_PACKET packet) throws CloneNotSupportedException {
    Window window = addPacket(packet.clone());
    sendWindow(window,  isFirst: true);
}
public Window addPacket(TCP_PACKET packet) {
    Window window = new Window(packet);
    sendContent.add(window);
    endWindowIndex++;
    return window;
}
private void sendWindow(Window window, boolean isFirst) {
    //发送数据报
    window.startSendTime = System.currentTimeMillis();
    client.send(window.packet);
    if (!isFirst) logger.warning( msg: "重新发送包:" + window.packet.getTcpH().getTh_seq());
}
```

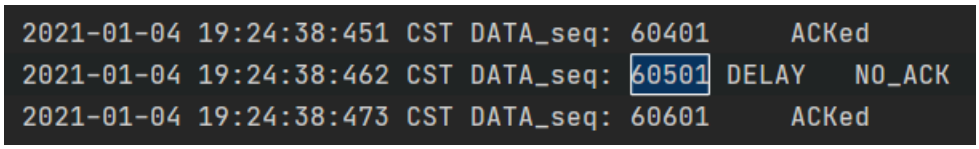
这样封装与滑动窗口机制结合，也可区分初次发包与重传发包，而主线程会判断窗口是否空闲，根据空闲情况决定是否继续发包。

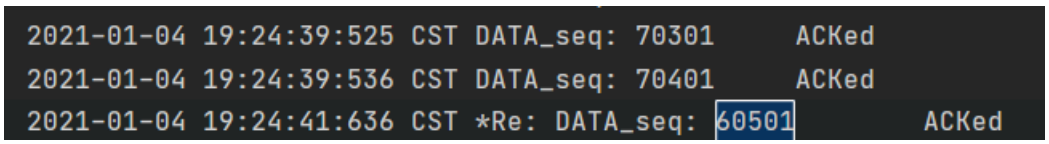
超时重传的改动：

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

结合了滑动窗口机制，从滑动窗口头开始检查超时，一方面若头部的有新的连续 ack, 则更新窗口头部的下标，另一方面重发超时未 ack 的包，







Log 文件分析出对于延时的包也能够被检测到并重新发送。

（5）Congestion Control

未完成

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）

对拥塞控制思想的理解不够透彻，无法将拥塞控制的具体实现写入滑动窗口中。

可能的解决方案为按照不同网络状况设置标识符，再根据标识符实现拥塞控制的不同特性。

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)

因为迭代开发是一个根据前面完成的功能进行进一步功能添加或修改的过程，所以迭代开发相比其他开发方式更加省时省力，免去重新编写部分代码的时间精力，但随着迭代次数增加，每次迭代时积累的诟病会逐渐累加，导致在较后的时期对前期代码的维护十分困难。

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)

实验开始就对实验给的各种接口犯难，后来发现了项目目录下的软件包文档，就可以结合所学知识搞清已知代码的作用与后续编写代码的思路。
发送方分组传输的具体实现不清楚，解决方案为滑动窗口的分组设立开始与结束的索引变量，在两者之间的所有元素算作一组，之后的分组传输与超时分组重传等机制就较为容易实现。

5. 对于实验系统提出问题或建议(1 分)

实验详解对于部分协议的讲解不够清晰，尤其是 Go-Back-N 部分缺乏系统的介绍和原理。