**计算机网络大作业报告**

**学号：20090022058** **姓名：朱甲文** **专业：机械设计制造及其自动化** **年级：2020**

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）

说明：目标版本号及得分对照

5.1 (Reno)：17

5.0 (Tahoe)：15

4.0 SR/GB/TCP：12

3.0：9

2.2: 6

2.0: 3

1. 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）
2. 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)
3. 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)
4. 对于实验系统提出问题或建议(1分)

**目录**

**[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 2](#_Toc2699)**

[1、RDT1.0 2](#_Toc8030)

[2、RDT2.0 2](#_Toc17826)

[3、RDT2.1 4](#_Toc29163)

[4、RDT2.2 5](#_Toc15871)

[5、RDT3.0 6](#_Toc7271)

[6、RDT4.0 Secletive Response 8](#_Toc22771)

[7、TCP Tahoe&Reno 11](#_Toc31988)

**[二、未完全完成的项目，完成中遇到的关键困难，以及可能的解决方式 1](#_Toc31576)9**

**[三、说明在实验过程中采用迭代开发的优点或问题 1](#_Toc31576)9**

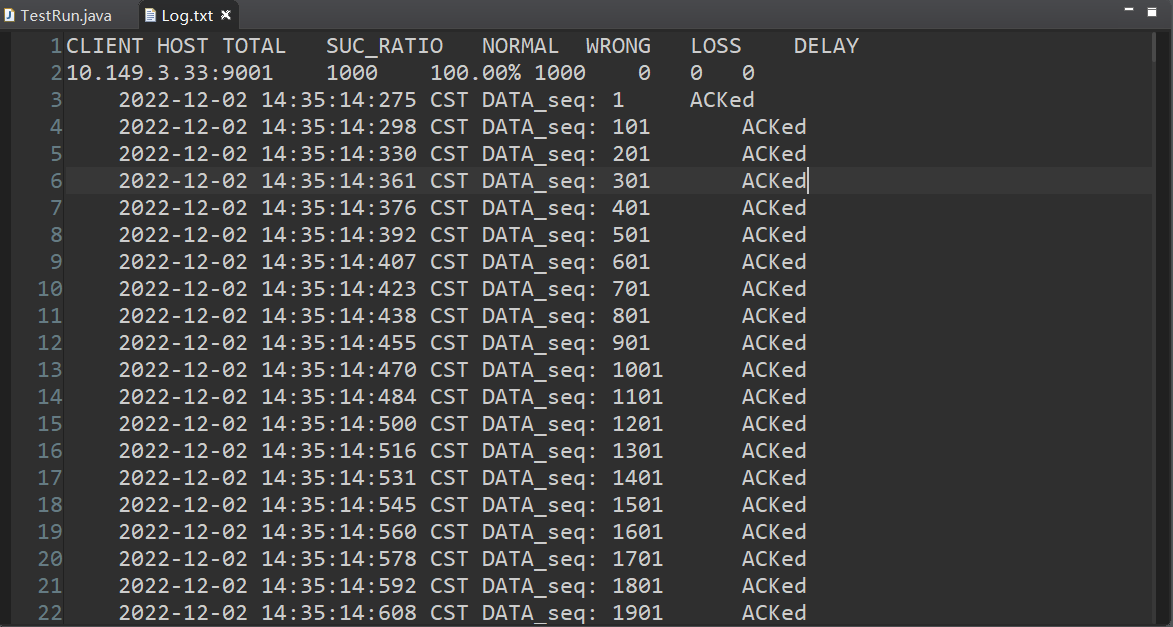
**[四、完成大作业过程中已经解决的主要问题和自己采取的相应解决方法 1](#_Toc31576)[9](#_Toc7960)**

**[五、对于实验系统提出问题或建议 2](#_Toc7960)0**

**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**

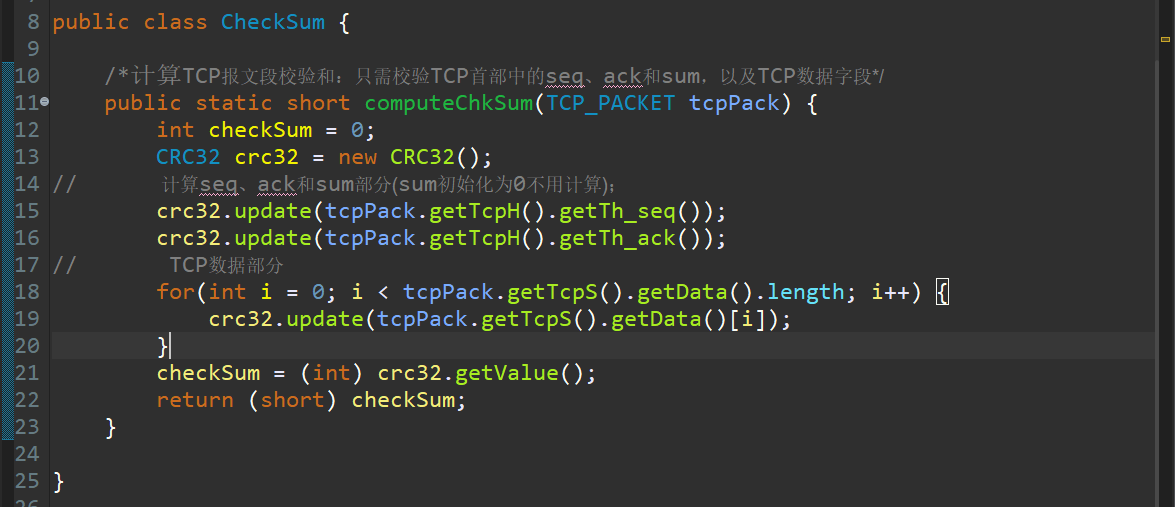
**1、RDT1.0**

RDT1.0是在可靠信道上进行可靠的数据传输。可以看到，数据在可靠信道上传输的成功率是100%。



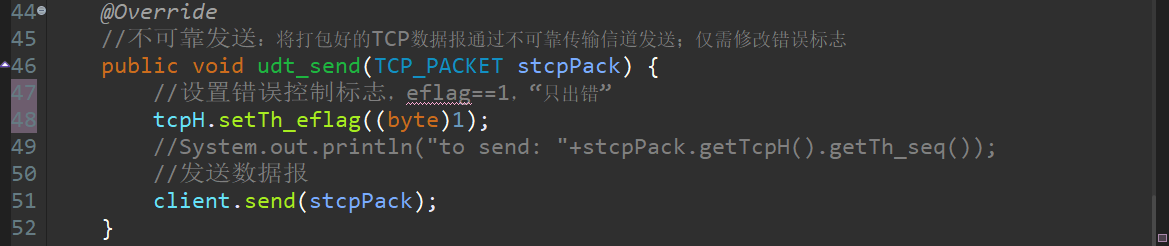
**2、RDT2.0**

RDT2.0假设底层信道传输过程中，个别数据包的某些字节可能发生位错，所以我们要利用校验和算法检查数据包的正确性。所以我们要先完成CheckSum类中的ComputeChkSum()方法，我们使用CRC32来计算校验码。

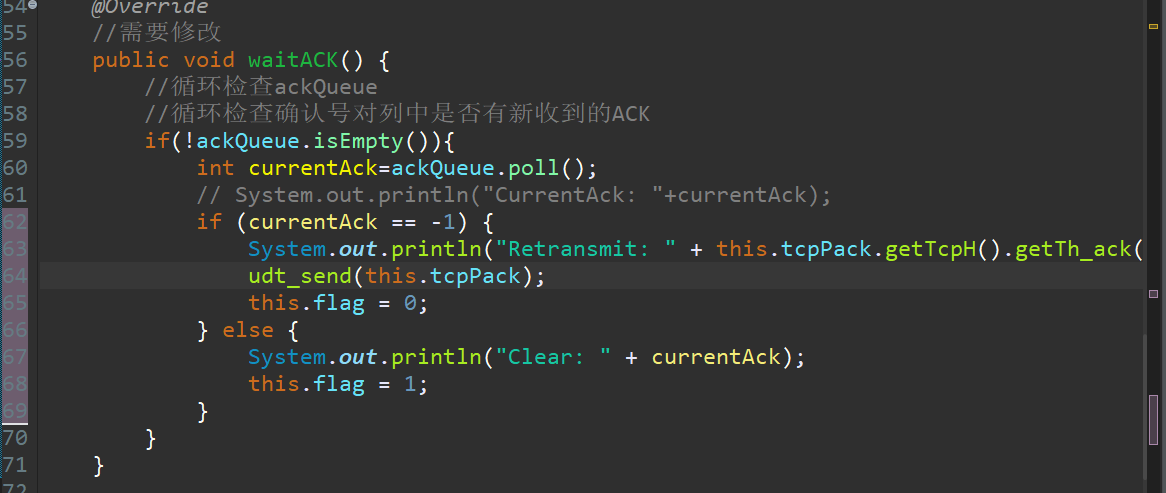


向该函数传入TCP包，拿出头部并获取其中的seq和ack字段，再和数据字段一同计算校验和，通过getValue()方法得到checkSum并返回。

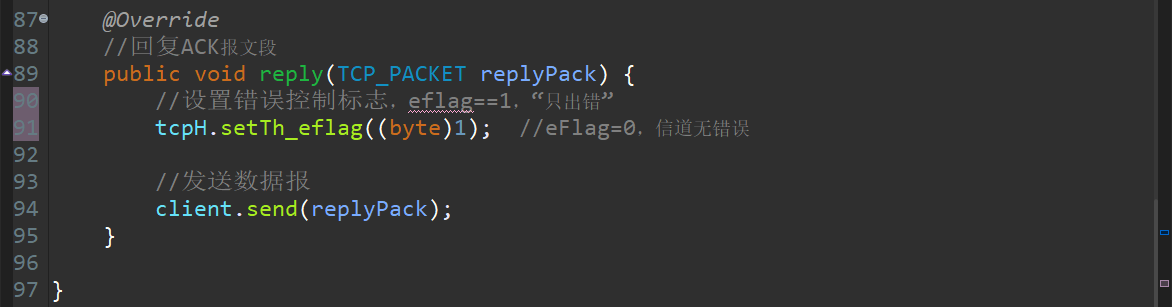
发送方：首先将TCP\_Sender中的udt\_eflag置为1，也就是“只出错”。



在waitACK状态，检查接收方发来的确认号队列，判定它是ACK还是NACK。如果是ACK，则切换状态等待应用层调用；如果是NACK，则继续等待ACK并且重新发送。



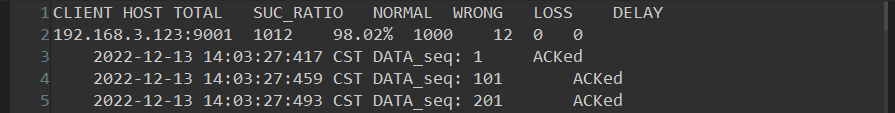
接收方：首先将TCP\_Receiver中的reply()中的eflag置为1。



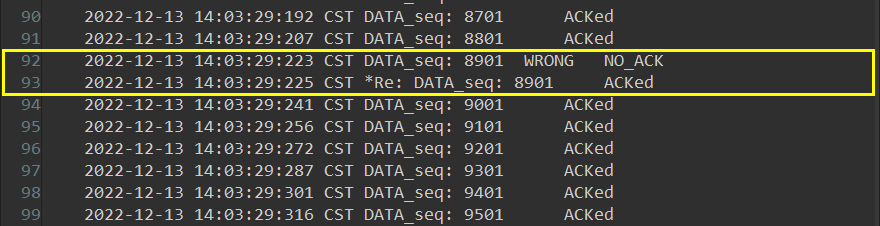
接受方接受到一个包时，首先将这个包送入ComputeChkSum()中计算校验和，并与所收到的包的头部中的校验和进行比较，如果相等，则证明TCP分组正确，没有出错，此时发送ACK给发送方；如果不相等，则表明TCP分组出错，此时发送NACK给发送方。

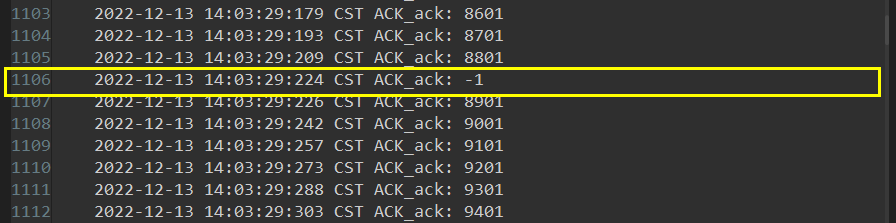
**Log分析**

可以看出，本次传输共发生了12次位错误。

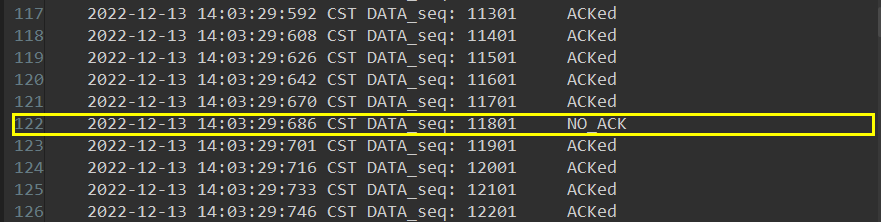


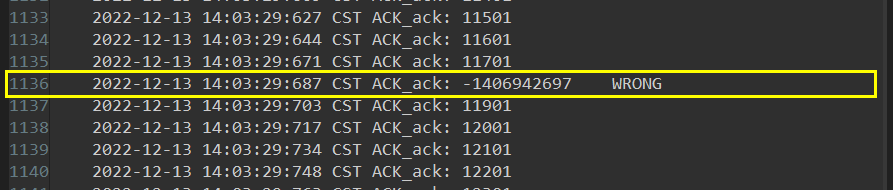
如下图，发送方发送了一个DATA\_seq为8901的包并出现了wrong，我们接着查看接收方的log，发现接收方回复了一个ACK\_ack为-1的包，此时发送方收到了这个ack为-1的包，于是进行了重传。





但是，RDT2.0无法处理出错的ACK/NACK，如下图所示。发送方发送了一个DATA\_seq为11801的包，而接收方回复了一个错误的ACK，可以看出发送方在收到接收方错误的ACK后并没有重传这个包。

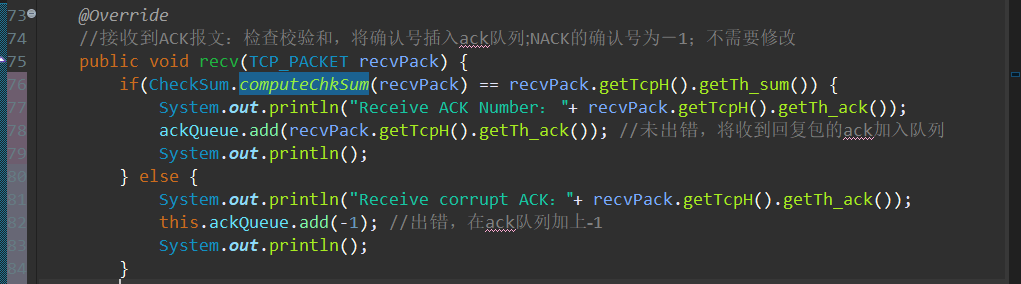




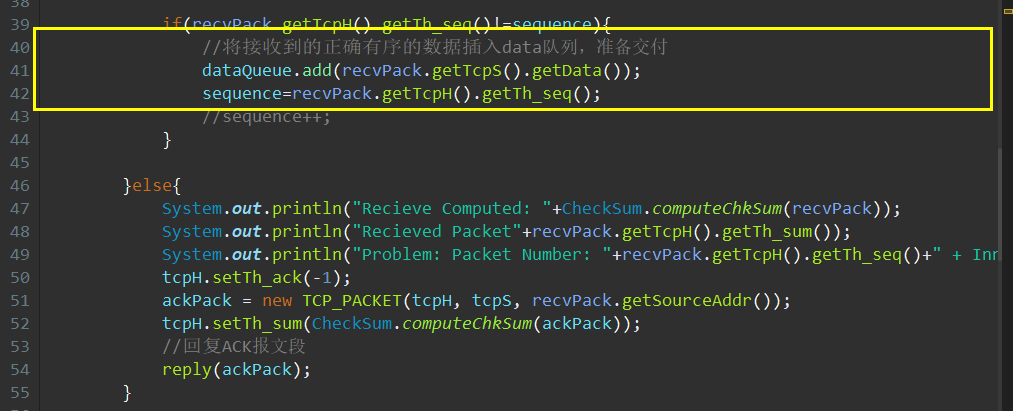
**3、RDT2.1**

在RDT2.1中，我们对RDT2.0的缺陷进行处理，即无法处理ACK/NACK出错的情况。

发送方：修改recv函数，将收到的回复包的ack加入ackQueue，如果出错，则在ackQueue中加上-1。

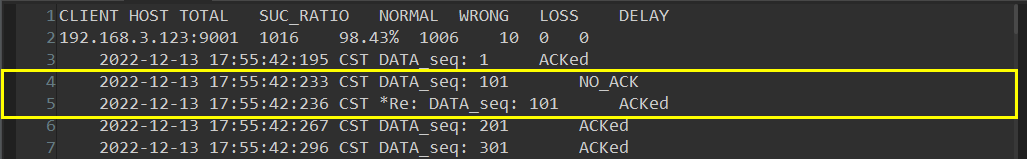


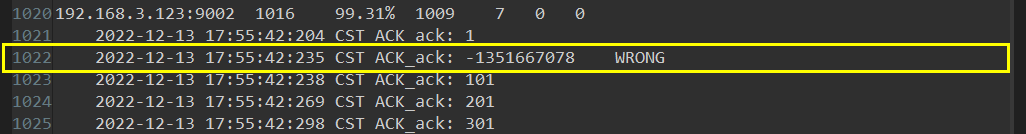
接收方：修改rdt\_recv函数，在计算校验和正确后，计算当前收到的包的seq，接收方需要记录上次接收的包的seq值,若与本次接收的相同,则不能将它插入data队列。



**Log分析**

如下图可以看出，在传输DATA\_seq为101的包时，接收方回复了一个错误的ACK，发送方收到这个错误的ACK之后进行了重传，这个重传的包被接收方收到后，接收方回复了正确的ACK，说明RDT2.1具备了检查ACK/NACK出错的能力，并能够在出错时对这个包进行重传。

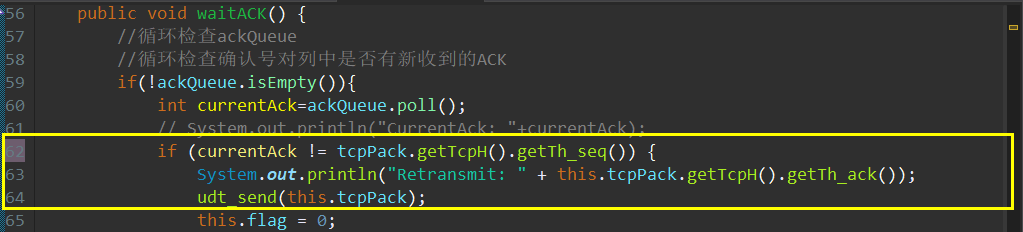




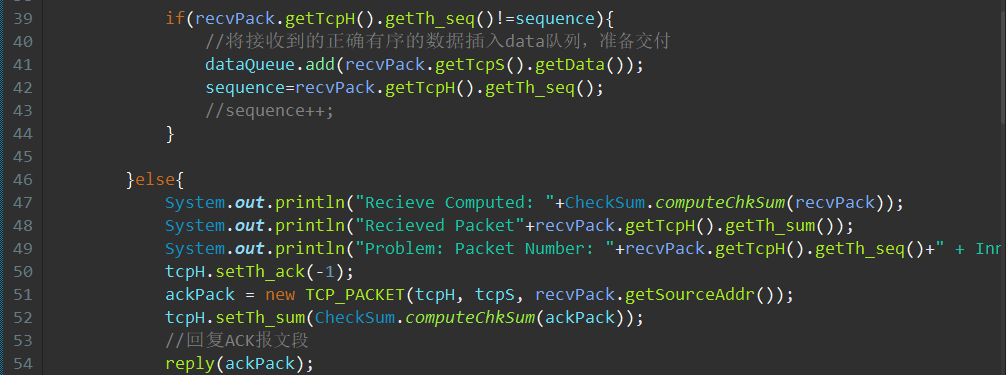
**4、RDT2.2**

RDT2.2和RDT2.1的功能相同，但是仅使用ACK。接收方正确接收一个包后，发送ACK，在ACK包中，接收方必须通过信号指明是对哪个数据包的确认。重复的ACK包对发送方来说，和收到NACK的效果一样。

发送端：修改waitACK函数，使其接收到错误的ACK后进行重发。

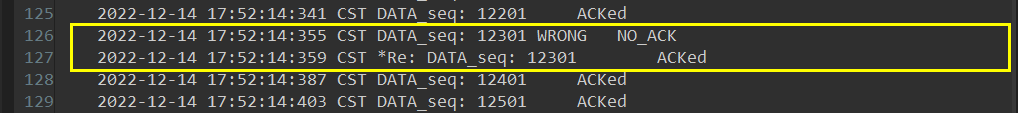


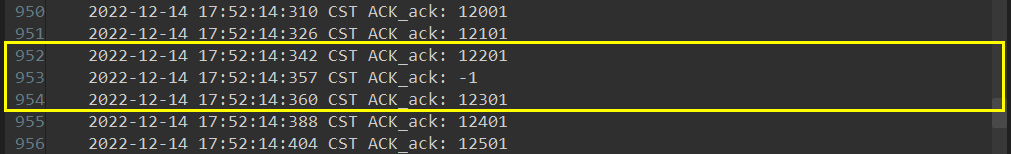
接收端：修改rdt\_recv函数，记录上次接收的包的seq值,若与本次接收的相同,则不能将它插入data队列，发送方就会重发数据包。



**LOG分析**

如下图，发送方发送了一个DATA\_seq为12301的包，但是发生了wrong并且是NO\_ACK，也就是没有收到正确的ACK，而接收方对DATA\_seq为12301的包出错后，回复了错误的ACK，此时发送方便会进行重发，于是接收方收到了正确的DATA\_seq为12301的包，并回复了正确的ACK。

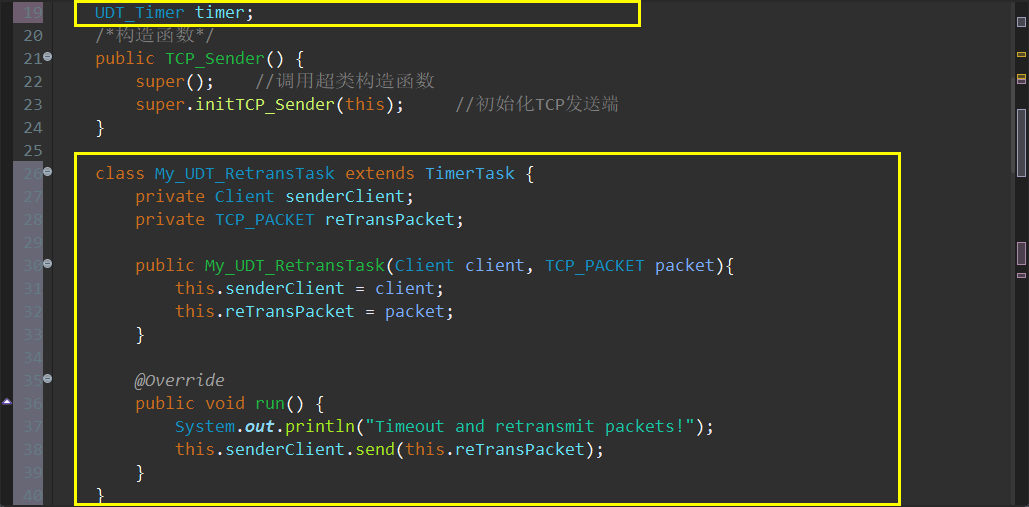


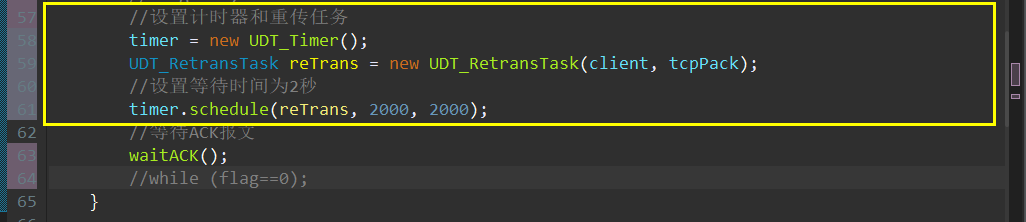


**5、RDT3.0**

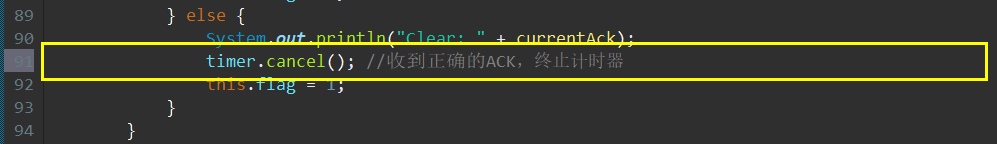
RDT3.0假设底层信道可能丢包，这时需要设置让发送方等待一段时间，当某些数据或ACK丢失时，也就是发送方在这段等待的时间内没有收到ACK，就进行重传。如果ACK仅仅被延迟而没有丢失，这时重传会导致包重复，我们可以通过seq进行控制，也就是接收方必须指明被ACK分组所确认的seq。

发送端：修改rdt\_send函数，通过实例化一个UDT\_Timer对象来作为一个计时器，再设置一个UDT\_RetransTask对象，作为重传任务，之后我们将计时器加入重传任务，也就是如果超过我们设定的时间发送方还没有收到回复，就进行重传。具体的实现方式如下：

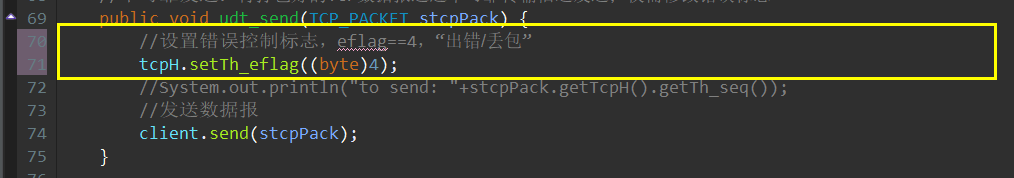




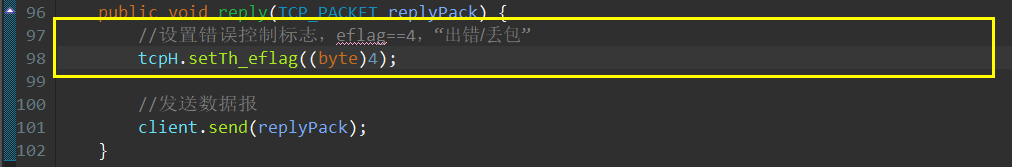
在这里，我们设置等待时间为2秒。同时，我们还要修改waitACK函数，如果收到正确的ACK，就终止计时器。如果收到错误的ACK，那么等待重传即可。具体的实现方式如下：



我们还需要修改udt\_send函数，将错误控制标志eflag修改为4，也就是“出错/丢包”的状态。

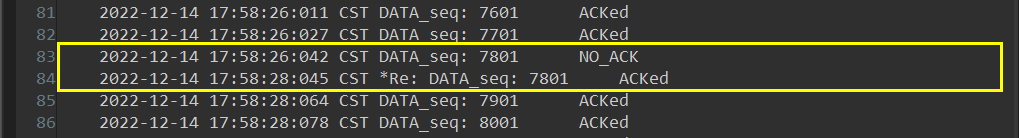


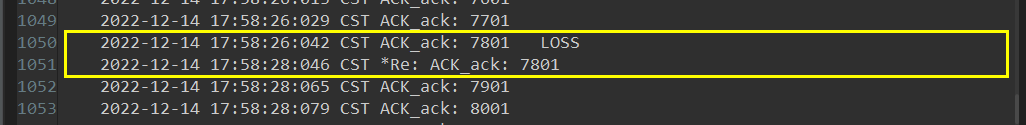
接收端：只需要修改reply函数，将错误控制标志eflag修改为4。



**Log分析**

可以看到，发送端发送了一个DATA\_seq为7801的包但是并没有收到ACK，而该包是丢失的状态，在等待2秒后，发送端重传了该数据包，接收端也回复了正确的ACK。

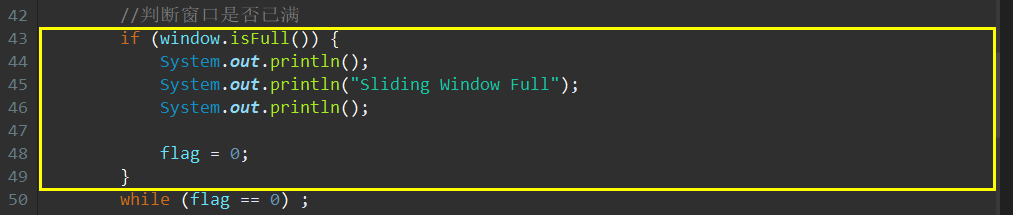




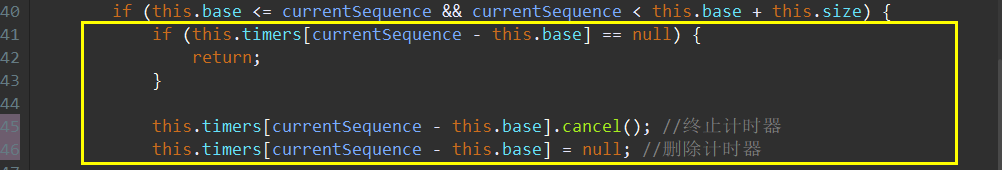
**6、RDT4.0 Secletive Response**

用来表达窗口的数据结构是数组，具体实现窗口的代码写在Sender\_Window.java和Receiver\_Window.java里。

在发送端中实现窗口满阻塞应用层的调用。具体实现方法：在TCP\_Sender.java的rdt\_send()函数中增加以下代码，用来判断窗口是否已满。如果已满，那么让flag=0，并且执行while循环，从而实现窗口满阻塞应用层的调用。



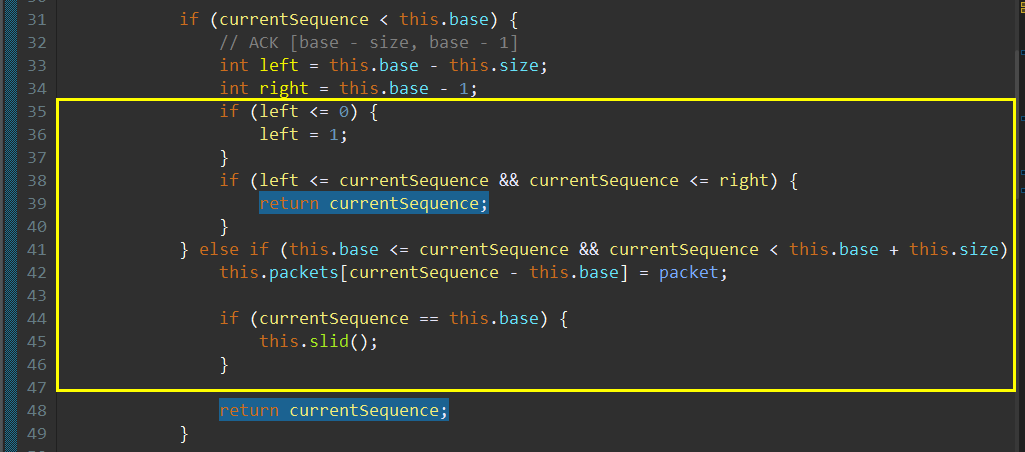
窗口在确认后如何移动：对于窗口外的确认不做响应。对于窗口内的ACK，我们要将其在窗口的对应的位置处的计时器停止并删除，具体的实现方法如下：在Sender\_Window.java中的receiveACK()函数中书写如下代码，从而实现计时器的终止和删除。



如果接收到的ACK位于窗口左沿，这时候要进行窗口移动，具体的实现方法如下：在Sender\_Window.java中的receiveACK()函数中书写如下代码，首先计算创口左沿应该移动到的位置，然后将窗口内的包移动，移动后清空已经左移的包原来所在位置处的包和计时器，最后更新窗口左沿的值和下一个包的插入位置。

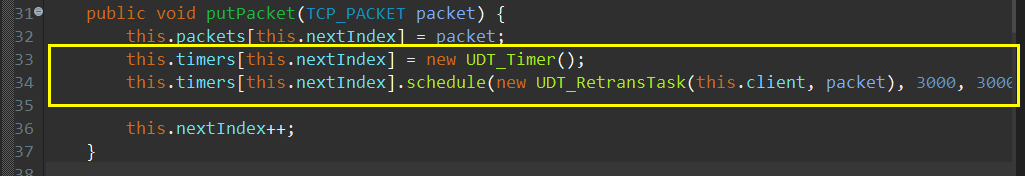


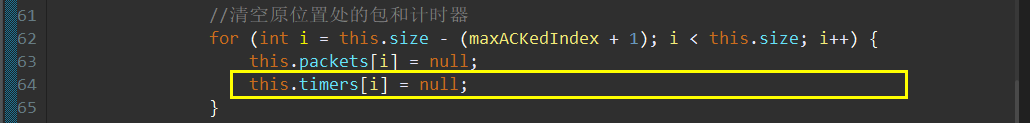
确认后的数据包是否去除：是。具体过程如下：接收方要维护一个接收窗口，使用数组来缓存正确接收的包，对于失序数组，要回复ACK，对于正确序号的数组，要将其加入到缓存窗口中，如果这个位置正好是窗口左沿的话，还要进行窗口滑动，最后交付数据。具体的代码如下：



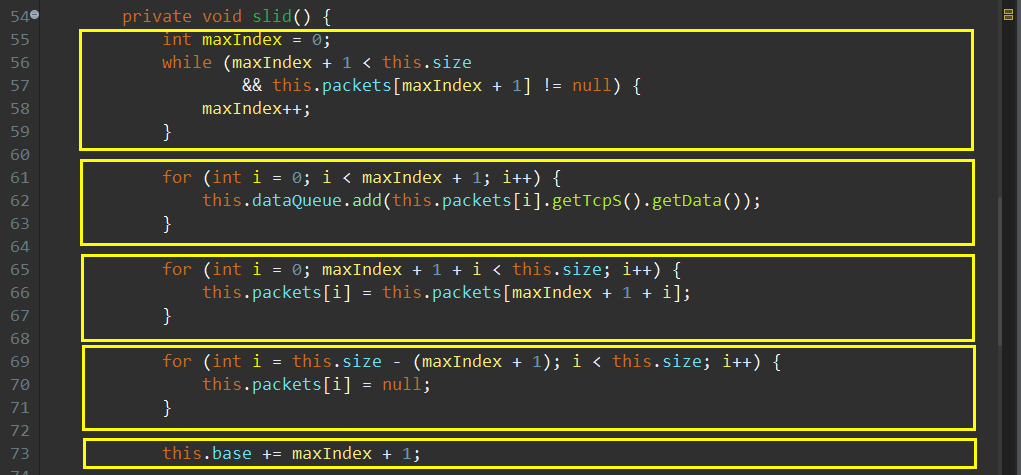
计时器数组如何声明和去除：在Sender\_Window.java中进行声明，并在putPacket()函数中使用，之后在receiveACK()函数中清除，具体的实现如下：



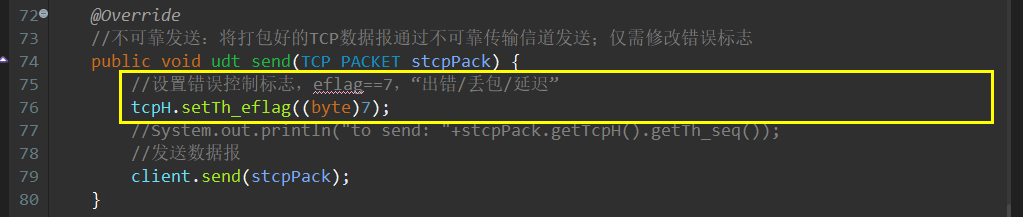


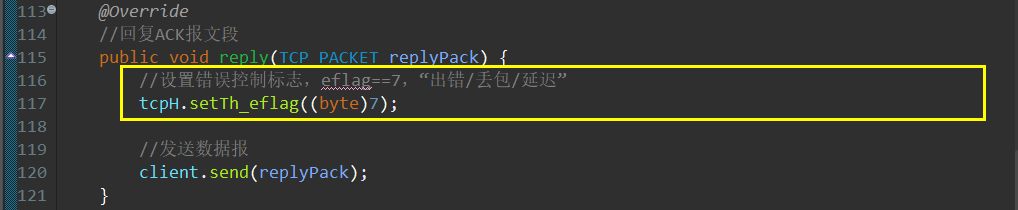


累计确认后的窗口如何移动：首先要计算窗口左沿应该移动到的位置，也就是最小未收到数据包处，之后将已接收到的分组加入到交付队列中，将剩余位置的包左移，再将左移的包的原来的位置处置空，最后移动窗口左沿。具体代码的实现在Receiver\_Window.java中的slid()函数中实现，如下：



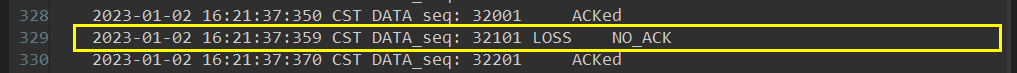
在传输之前，我们要将错误控制标志eflag修改为7，也就是“出错/丢包/延迟”的状态，如下。

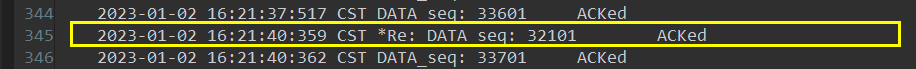


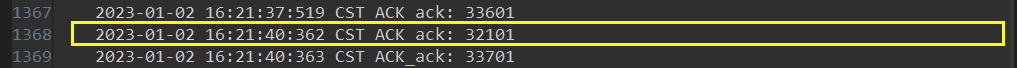


**Log分析**

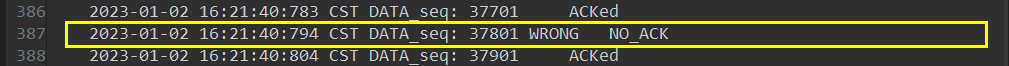
对于丢包的情况，seq为32101的包出现了LOSS，发送方没有收到该包的ACK，由于我们设定的计时器等待时间为3000ms，所以3000ms之后，发送方重新发送了这个包并收到了正确的ACK。

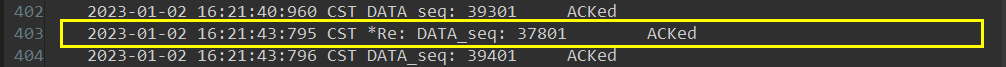


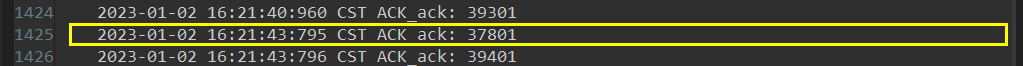




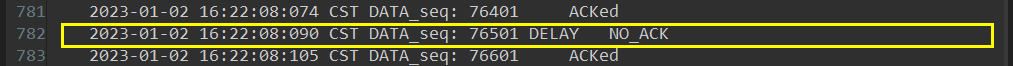
对于出错的情况，seq为37801的包出现了WRONG，发送方没有收到该包的ACK，所以3000ms之后，发送方重新发送了这个包并收到了正确的ACK。

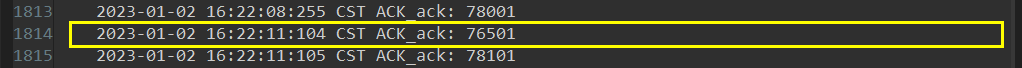
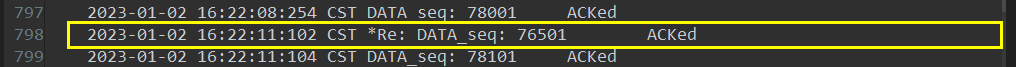






对于延迟的情况，seq为76501的包出现了DELAY，发送方没有收到该包的ACK，所以3000ms之后，发送方重新发送了这个包并收到了正确的ACK。

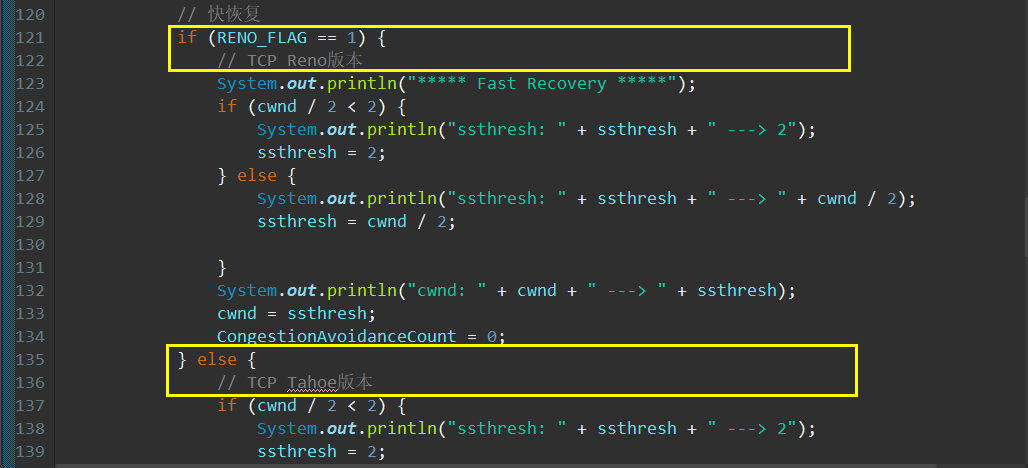




**7、TCP Tahoe&Reno**

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，分别是慢启动、拥塞避免和快重传。而Reno除了包含Tahoe的三个算法，还多了一个Fast Recovery（快速恢复）算法。当收到三个重复的ACK或是超过了RTO时间且尚未收到某个数据包的ACK，Reno就会认为丢包了，并认定网络中发生了拥塞。Reno会把当前的ssthresh的值设置为当前cwnd的一半，但是并不会回到slow start阶段，而是将cwnd设置为（更新后的）ssthresh+3MSS，之后cwnd呈线性增长。

在实验代码中，由于Reno版本仅比Tahoe多了快速恢复算法，其他均类似，所以我们在Sender\_Window.java中定义了变量RENO\_FLAG，当该变量的值为0时，代码为Tahoe版本，当变量的值为1时，代码为Reno版本。该功能的实现写在Sender\_Window.java的receiveAck()函数中，通过一个if else分支结构来实现，如下：



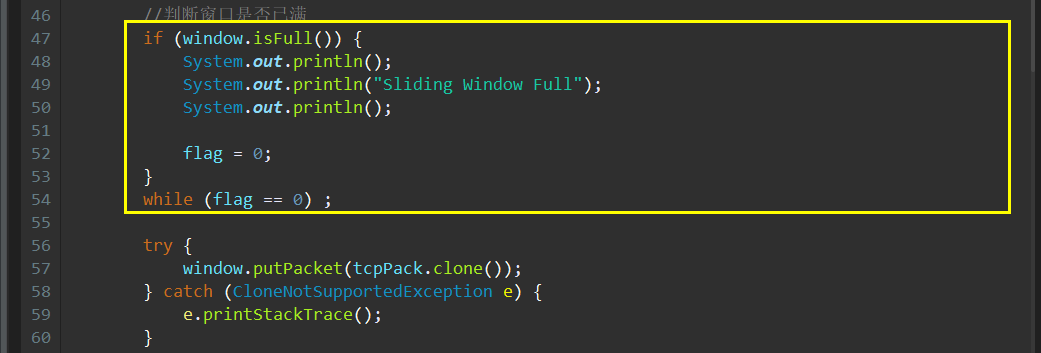
在该版本中，我们设接收方的窗口大小为无限大。

在该版本中，发送方的滑动窗口编写在**Sender\_Window.java**中，**慢开始、拥塞避免、快重传和快恢复**均在此代码中实现，对于该代码的介绍在P14。

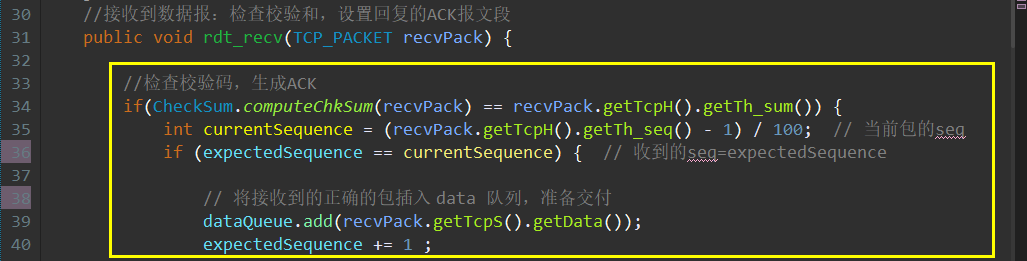
超时重传的内容编写在TCP\_Retry.java中。

发送端**TCP\_Sender.java**和接收端**TCP\_Receiver.java**的具体介绍如下。

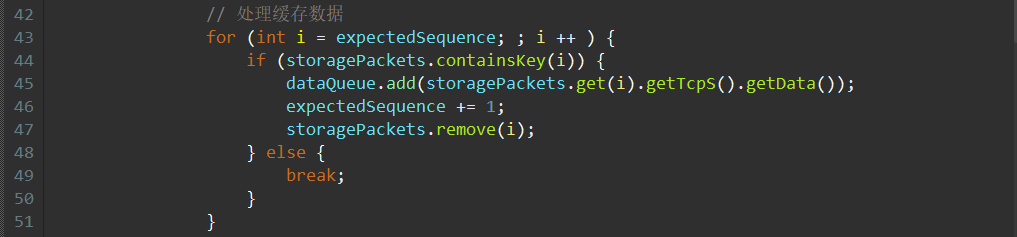
窗口满阻塞应用层调用的实现：与Selective Response版本类似，在发送端中实现窗口满阻塞应用层的调用。具体实现方法：在TCP\_Sender.java的rdt\_send()函数中增加以下代码，用来判断窗口是否已满。如果已满，那么让flag=0，并且执行while循环，从而实现窗口满阻塞应用层的调用。其中，putPacket()函数的作用是处理发送方的滑动窗口。



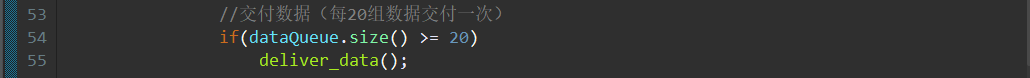
在接收端TCP\_Receiver.java中，我们使用HashTable来缓存失序的分组，并且定义一个私有变量expectedSequence来表示我们期望收到的包的seq，这样如果我们对收到的分组计算校验和，如果正确并且收到的seq就是我们期待的expectedSequence，我们就将接收到的正确的包插入准备交付的data序列，并且随着正确的包的接收将expectedSequence进行自加1操作。具体的代码实现如下：



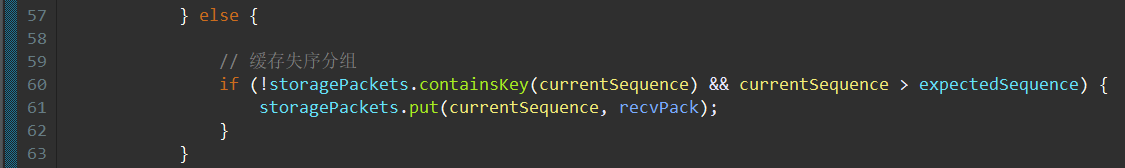
接下来，我们判断用来缓存失序包的哈希表中是否有数据，也就是从expectedSequence开始进行循环加1来检查，如果有数据就需要插入data队列中，同时在哈希表中删除交付的数据，具体的实现如下：



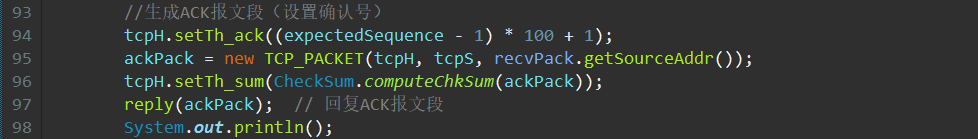
之后交付数据即可，如下：



如果出现seq不等于expectedSequence的情况，则进行缓存操作。如果要缓存的分组的seq小于expectedSequence，此时该seq已经交付，因此无需缓存，如果要缓存的分组的seq大于expectedSequence，则将其缓存，具体的实现如下：

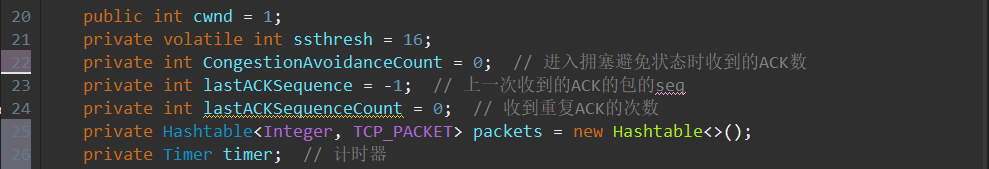


最后回复ACK即可。

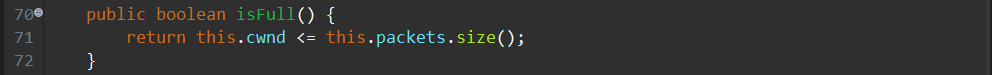


在Sender\_Window.java中实现了慢开始、拥塞避免、快重传和快恢复。

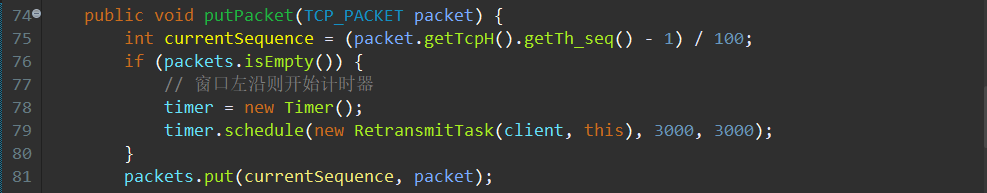
首先我们定义了拥塞窗口cwnd和慢开始的门限ssttresh两个变量，初始值分别是1和16。定义了变量CongestionAvoidanceCount，记录进入拥塞避免状态收到的ACK的数量，初始值是0。定义了变量lastACKSequence和lastACKSequenceCount，分别记录上一次收到的ACK的seq和重复ACK的数量，初始值分别是-1和0，这两个变量主要用于快重传的条件的判断。定义了一个哈希表，用来表示窗口，key是分组序号seq，value是TCP的分组类型。最后我们定义一个计时器，如下：



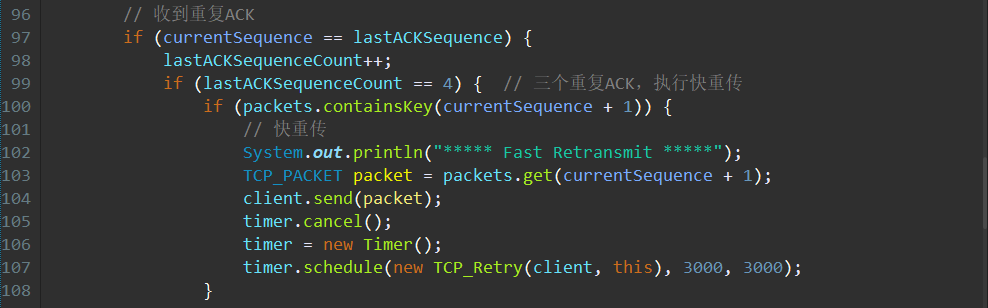
我们通过isFull()函数来判断发送窗口是否已满，也就是当cwnd与packets相等时，发送窗口已满。



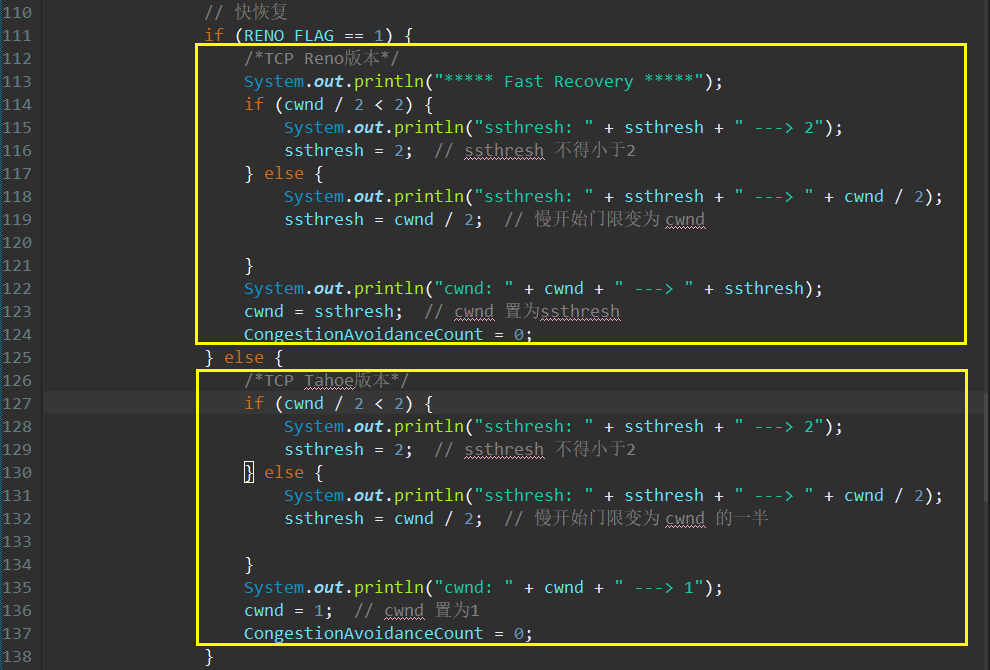
接下来，我们编写了一个putPacket()函数，其作用是将分组放入滑动窗口中。首先要计算当前的seq，如果加入的分组在窗口左沿，也就是当packet为空时，开启计时器，我们设定的时间为3000ms。之后在packets中加入这个分组。



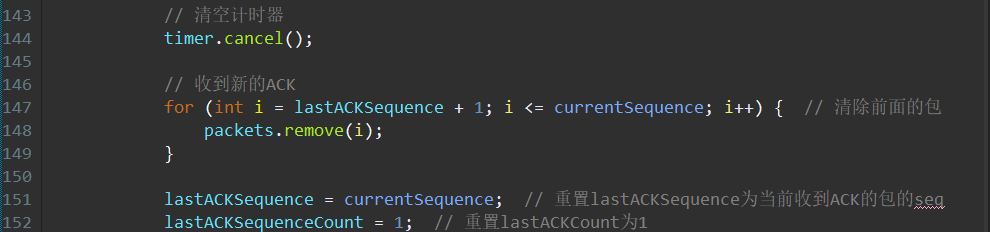
我们还编写了receiveACK()函数，判断是否有重复的ACK，进一步判断是否是三个重复的ACK，如果是的话启动快重传，重传的分组是上一个ACK对应的seq值加1，并且重启计时器，具体的实现如下：



接下来是快恢复阶段，将ssttresh变为cwnd的1/2，cwnd变为ssttresh，如果cwnd大于等于ssttresh，则进入拥塞避免阶段。对于Tahoe版本，这里cwnd不是变为ssttresh，而是置为1。



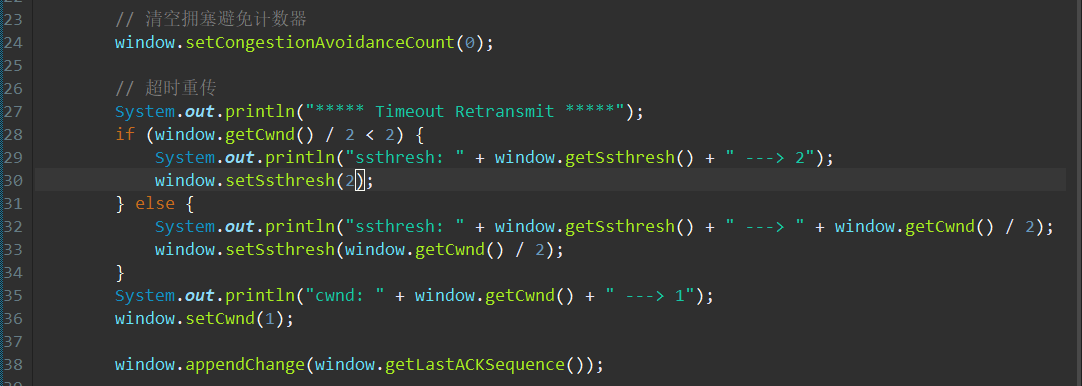
接下来是清空拥塞避免的计时器。如果收到新的ACK，那么直接清空计时器，还要对当前的seq之前的分组进行删除，并且更新lastACKSequence为当前的ACK对应的seq值，并且更新lsatACKSequenceCount为1。



如果窗口中仍然有分组，那么需要重新开启计时器，并且更新窗口的大小cwnd和慢开始的门限ssttresh，如果cwnd小于ssttresh的话，那么说明这时处于慢开始阶段，每收到一个ACK，窗口的大小cwnd就要加1。如果cwnd大于等于ssttresh的话，说明此时处于拥塞避免阶段，计数器CongestionAvoidanceCount要加1，如果计数器的值等于窗口大小，此时要使cwnd加1并清空计数器，如下：

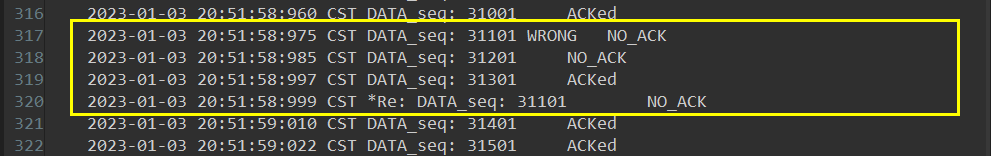


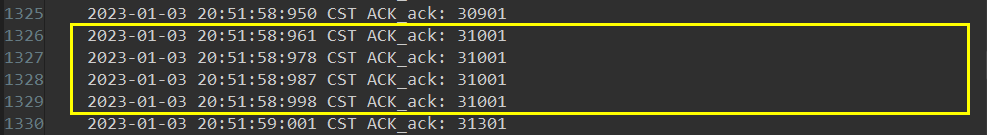
在TCP\_Retry.java中实现了超时重传。首先要清空拥塞避免的计时器，然后立刻重传超时的分组。在超时重传窗口中，ssttresh变为cwnd的一半，cwnd置1。



**Log分析**

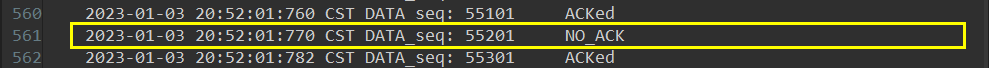
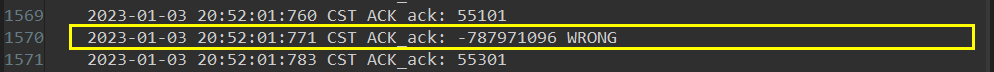
对于出错的情况，对于seq为31101的分组出现了WRONG，于是接收方回复了已确认序号的最大的ACK，也就是31001，同时，发送方发送的seq为31201同样是NO\_ACK的状态，因为接收方回复的也是已确认序号的最大的ACK，也就是31001，于是接收方同样缓存了31201，但是继续往下可以发现，发送方发送的seq为31301的分组收到了ACK，但是接收方回复的不是31301，而同样是最大ACK的seq，也就是31001，这是因为发送方连续收到三个重复的ACK之后，执行了快重传，立刻重传了分组31101，重传之后还要取消并且重开计时器，



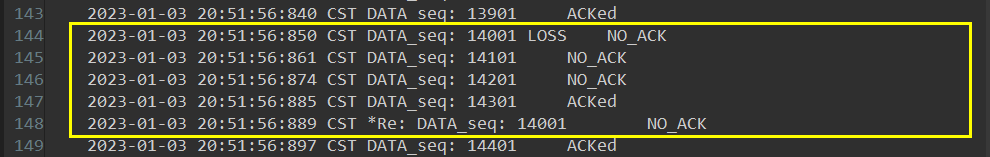


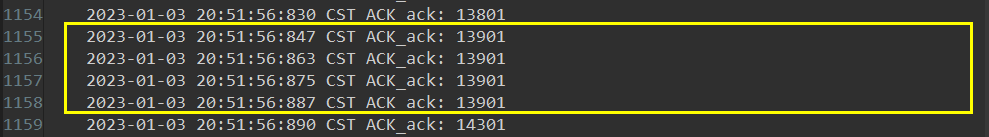
我们继续观察，接收方收到了这个重传的分组，同时缓存了失序分组31201和31301，所以分组31301才会收到ACK。

接下来，我们考虑ACK出错的情况。可以看到，seq为55201的分组的ACK出错了，发送方对于seq为55201分组是NO\_ACK状态，但是并未采取重传等措施，这是因为发送方已经接受到了后面的seq的ACK，于是发送方认为该分组已经被接收端正确的接收。所以发送方对于出错的ACK不做任何处理。

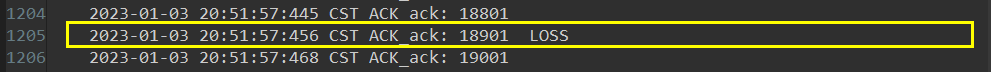


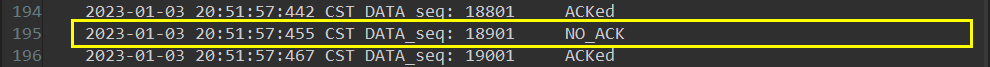
我们考虑丢包的情况，可以看到，seq为14001的分组发生了LOSS，由于LOSS的分组并没有ACK，于是接下来发送方要连续发送三个分组之后才会启动快重传。在接收方我们也可以看出，接收方缓存了14101和14201，重传之后，14001,14101和14201都成功到达了接收方。



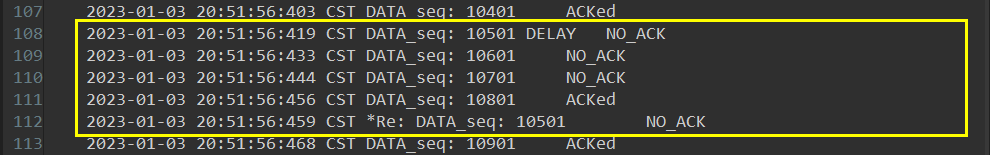


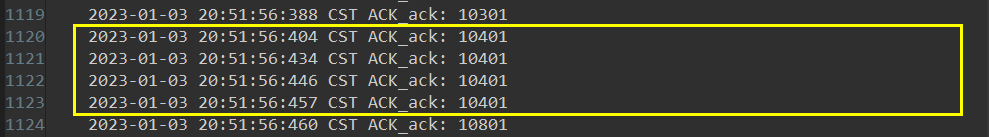
接下来，我们考虑接收方发送ACK丢失的情况。可以看到，ACK为18901发生了LOSS。发送方对于seq为18901分组是NO\_ACK状态，但是并未采取重传等措施，这是因为发送方已经接受到了后面的seq的ACK，于是发送方认为该分组已经被接收端正确的接收。



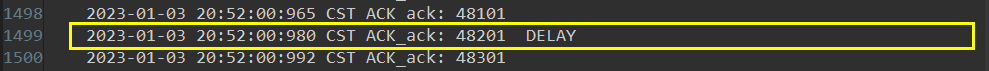


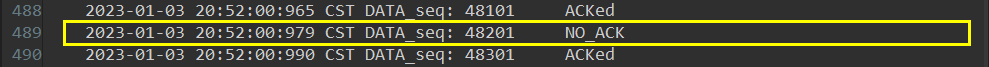
我们考虑延迟的情况，可以看到，seq为10501的分组发生了DELAY，由于DELAY的分组并没有及时得到ACK，于是接下来发送方要连续发送三个分组之后才会启动快重传，但是DELAY的分组最终会到达接收方。在接收方我们也可以看出，接收方缓存了10601和10701，重传之后，10501,10601和10701都成功到达了接收方。





接下来，我们考虑接收方发送ACK延迟的情况。可以看到，ACK为48201发生了DELAY。发送方对于seq为48201分组是NO\_ACK状态，但是并未采取重传等措施，这可能是因为ACK到达发送方之前TCP连接已经释放。





**二、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。**

RDT 2.0、RDT 2.2、RDT 3.0、RDT 4.0 SR、RDT 5.0 (Tahoe)、RDT 5.1 (Reno)版本均已经完成，除了前三个版本外，其他版本在完成时或多或少遇到了一些困难，不过均已经通过自己查资料、和同学交流方法等途径解决，最终完成了Reno版本。

1. **说明在实验过程中采用迭代开发的优点或问题。**

我认为迭代开发的优点是显然的，迭代开发从易到难，可以兼顾不同层次的学生，最简单的RDT 2.0版本，其核心仅需要完善校验和函数，这对于刚接触计算机网络这门课程的我们还是比较友好的，上手较为容易。不过随后的几个版本难度都在逐渐增大，尤其到了拥塞控制阶段，需要我们不断的去翻阅课本，回顾课堂知识，这样可以加深我们对于课堂知识的理解。

至于迭代开发容易出现的问题，我也深有体会。比如我在做RDT 3.0版本时，意外的发现自己之前书写的一个函数有问题，这下我不仅需要重新改动代码，还要把之前已经使用过这部分代码的RDT 2.2版本也重新推翻，重新再进行分析等。所以迭代开发要求我们更加细心，每完成一个版本都尽量不要留下错误，以免后面填坑更加麻烦。

1. **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**

在我实现低版本的过程中并未遇到太大的问题，主要问题是在Reno版本的实现中，如何实现拥塞避免。我采用的方法是使用拥塞避免计数器，也就是在拥塞避免阶段收到正确的ACK就让计数器加1，这样收到16个ACK之后就可以让窗口的大小直接变成17，但是退出拥塞避免时一定要清空计时器。

1. **对于实验系统提出问题或建议**

说一点自己的在实验过程中深有体会的建议（可能无关紧要），就是我在进行Log文件分析时，鼠标上下不断滑动，有一种“乱花渐欲迷人眼”的感觉。所以我认为如果Log文件中发送方和接收方分列左右两栏可能看起来会更加的简便。当然，上下布局的话并不会有什么大的问题，也只是一点无关紧要的小建议，这个TCP实验的意义远大于我在实验前想象到的，这个实验使我的课堂理论知识更加巩固，对知识的理解也更加深刻，在这里要说一句老师辛苦了，助教学长学姐辛苦了！