南京大学本科生实验报告

- 课程名称：**计算机网络** 任课教师：田臣/李文中
  助教：

学院	计算机科学与技术系	专业（方向）	计算机科学与技术
学号	201220102	姓名	武雅琛
Email	201220102@smail.nju.edu.cn	开始/完成日期	2022.5.18

1. 实验名称

Reliable Communication

2. 实验目的

- **进一步熟悉Switchyard框架。**

  通过自行学习Switchyard相关**API的实现和接口**进一步熟悉lab提供的Switchyard框架。
- **学习端到端可靠通信机制**

  了解端到端到端通信通过等待确认方式或**滑动窗口分组确认的方式**来实现IP尽力而为服务上的可靠通信机制。
- **学习端到端通信发送端滑动窗口工作方式并简要实现**

  深入学习滑动窗口的工作细节并在Switchyard框架中实现简单的基于滑动窗口的端到端可靠通信。

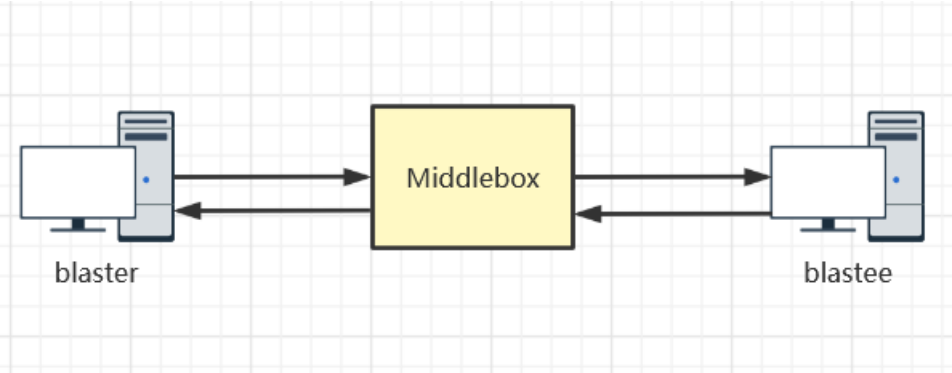
3. 实验内容

task 1:Preparation

task 2:Middlebox

- 这一步要求实现一条链接端到端之间的**逻辑链路**，这条链路封装了三层及以下的通信细节（包括网络层路由器IP通信、链路层交换机通信以及物理层通信等），只需要提供上层需要的工作功能。相当于对使用者展示一条从blaster端到blastee端的虚拟电路。
- 在本实验中，只需要实现以下两个功能：**传递分组**和按照一定概率**模拟真实电路**中的丢包情形。
- **传递分组**
  - **端口转发逻辑**

    Middlebox提供两个端口分别与blaster和blastee连接，可以接收来自两侧的分组并且**转发到另一个端口**，这种转发逻辑非常简单，无需查找转发表，我们只需要硬编码在Middlebox.py即可。



  - **发送分组**

    在网络层转发分组时需要**修改以太网包头指向下一跳的mac地址。**同样，虽然平时需要通过发送ARP请求获取目的端口的mac地址，在这里终端地址也是固定的，通过硬编码在Middlebox.py即可。

- **模拟真实电路丢包**
  - 逻辑上，实验要求实现从blaster到blastee方向链路按照**一定概率丢包**，也就是Middlebox在某些情形下**收到报文不做转发。**

    可以在每次转发之前获得一个**大小在[0,1]的随机数**，通过和droprate比较判断如果大于等于droprate做分组转发即可。
  - 实现中可以调用python提供的`random`库中`random.uniform()`函数。

task 3:Blastee

- 在本实验中简化了端到端通信接收方提供的服务，**只保留了构造ACK分组并发送确认**的功能，不通过接收端滑动窗口实现累积确认。
- 具体实现以下两种功能：根据接收分组**构造ACK分组**内容和**构造转发包头。**
  - **构造转发包头**

    以太网和IPv4包头同task2中的实现方式一致，采用**硬编码**的方式，不赘述。

    UDP包头只需要**随意填写端口号**，注意**封装在以太网和IPv4包头**之后即可。
  - **构造ACK分组**

    这一部分以字节流的形式编码到报文中，分为两个部分：确认序号和载荷。
    - 确认序号

      本实验中确认序号以**字节串大端方式**存储。

      要填写正确的确认序号，需要解析接收到的报文序号。阅读blaster的报文结构，可以了解到Sequence在字节串的前四个字节。利用python提供的API`int.from_bytes`解码即可。

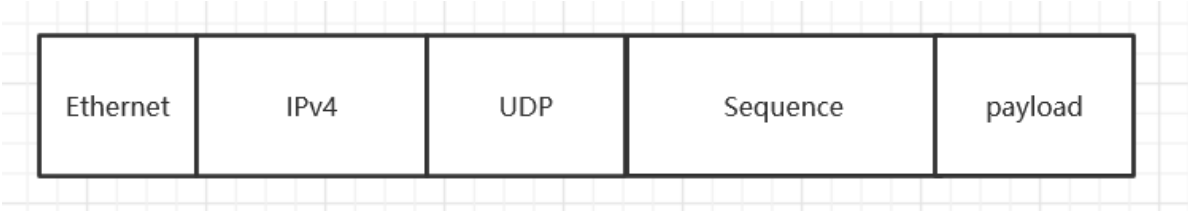
      ```
      recv_SequenceNum_bytes =  recv_RawPacketContent[0:4];
      recv_SequenceNum = int.from_bytes(recv_SequenceNum_bytes,'big')
      ```

    - 载荷

      接收方只需要取出发送端报文载荷的前八个字节的内容填写在ACK报文中即可。结合blaster报文结构：

      ```
      recv_payload =  recv_RawPacketContent[7:16]
      ```

最后按照以下顺序组成分组即可：



task 4:Blaster

- 本实验中发送方通过以滑动窗口为基础，实现粗粒度的**超时重传机制**和**伪发送速率。**
- 主要包括四个部分：**处理ACK确认分组，构造发送分组，维护滑动窗口，超时重传机制。**
  - **处理ACK确认分组**

    从接收到的分组得到确认号并解码在task3中有涉及到。

    只需要根据确认号检查是否**在滑动窗口中并且尚未确认**，否则，视作重复或无效分组丢弃。

    如果在窗口中应该修改状态：

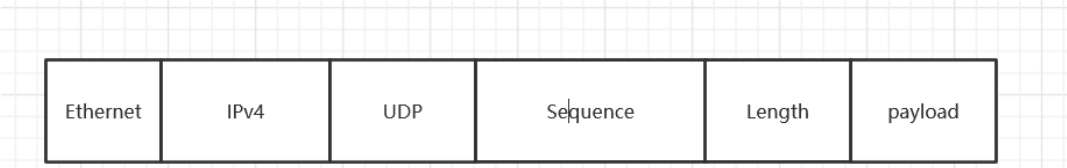
    ```
    self.window.SW[recv_SequenceNum][1] = 'Ack'
    ```

  - **构造发送分组**
    - 构造转发包头：这一部分和task3重合，不赘述。
    - 构造字节串分组头部和载荷：、

      和task3的实现方式基本一致，只需要在序号和载荷中插入两个字节的载荷长度即可。

      ```
      SequenceNum_bytes = SequenceNum.to_bytes(4,byteorder='big')
      length_bytes = self.length.to_bytes(2,byteorder='big')
      ```

    - 最后按照以下顺序组成分组即可：

  - **维护滑动窗口**

    维护滑动窗口主要维护的是滑动窗口的左右边界，已发送分组和计时器。

    在实现时包括以下成员：

    ```
    #class Window:
            self.SW = dict()#
            self.Maxsize = size
            self.LHS = 1
            self.RHS = 0
            self.ResendHead = -1
            self.ResendPos = 0
            self.timer = time.time()
    ```

    特别的，我采用**字典**来**存储滑动窗口对应序号的分组内容、确认状态、重传次数和开始发送时间**，方便通过序号直接访问到对应的信息及表项的增删。

- 滑动窗口的左边界会在左边界指向的分组成功ACK之后收缩，这个过程我放在了`handle_no_packet()`函数中。

  实现逻辑为通过判断序号对应的分组是否被ACK不断将LHS向右收缩，直到遇到第一个未被ACK的分组或者是RHS。

```
...
            if self.Window.SW[idx][1] == 'Ack':
                self.Window.LHS += 1;
                self.Window.timer = time.time()
                self.Window.ResendHead = -1
            else:
                break
```

- 滑动窗口的右边界在滑动窗口的大小收缩至**小于最大长度时**拓展，在拓展时构造一个新的分组从端口发出。

  由于有控制发送速率的需求，在每次进入handle_no_packet函数中**至多可以拓展一个单位。**

```
if self.Window.RHS - self.Window.LHS + 1 < 5:
    if self.Window.RHS < self.num:
```

- 计时器在两种情况下出现更新：**LHS收缩**之后和**LHS指向的分组重发**之后。

- **超时重传机制**

  为实现超时重传机制，我在滑动窗口上增加了两个标志：`ResendHead`和`ResendPos`。

  - ResendHead：表示是**是否进入重传**状态，如果为-1则未进入超时重传状态，否则指向超时重传的LHS的元素。方便**检查一次重传是否结束。**只需要再进入超时重传状态时将它置为LHS的值即可，在任何需要打断重传的情境下置为-1即可中断。
  - ResendPos：表明如果在重传状态，**重传的分组序号。**每次重传结束后，沿序号遍历滑动串口，找到**下一个`NotAck`的元素**，将序号赋值给ResendPos。
  - **特别的，在*LHS*收缩后如果仍旧在重传中需要打断重传；在一次重传了滑动窗口中所有未确认分组后打断重传，等待下一次超时；在滑动窗口中的元素尚未完全发送完毕时出现新的超时刷新重传：即回到*LHS*的位置开始新的一轮重传。**

4.实验结果

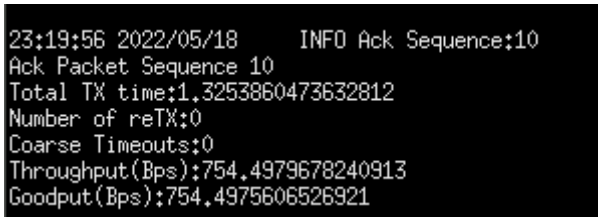
Deploying

当然我也做了num更大的相关测试，但是因为测试内容较繁杂不易展示，这里均以num=10为例讨论典型现象。

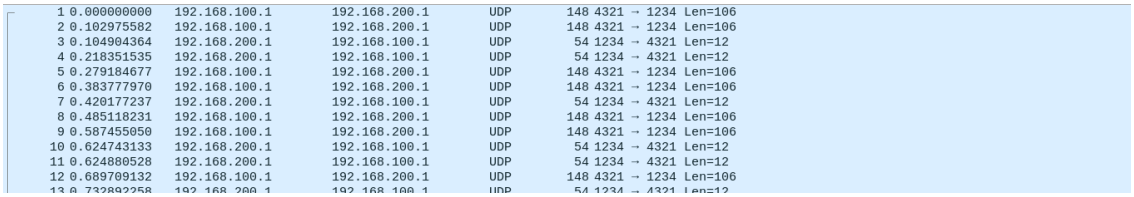
- 第一组参数：将丢包率置为0，检验是否可以实现基本的分组发送和确认机制。

```
middlebox# swyard middlebox.py -g 'dropRate=0'
blastee# swyard blastee.py -g 'blasterIp=192.168.100.1 num=10'
blaster# swyard blaster.py -g 'blasteeIp=192.168.200.1 num=10 length=100
senderWindow=5 timeout=300 recvTimeout=100'
```

可以看出在没有重传的情况下顺利的完成了十个分组的传送。



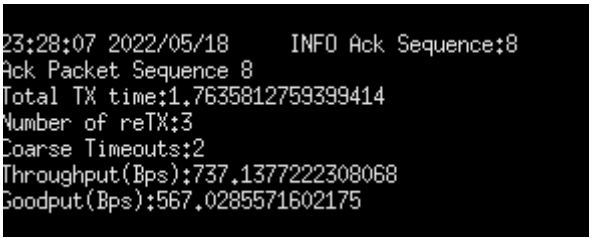
wireshark在blaster的端口抓包也得到了预期效果，可以看到二者建立了双向的链路。



- 第二组参数：采用实验手册中的数据，为了方便分析，在这里将规模缩小到10个分组分析。

```
middlebox# swyard middlebox.py -g 'dropRate=0.19'
blastee# swyard blastee.py -g 'blasterIp=192.168.100.1 num=10'
blaster# swyard blaster.py -g 'blasteeIp=192.168.200.1 num=10 length=100
senderWindow=5 timeout=300 recvTimeout=100'
```

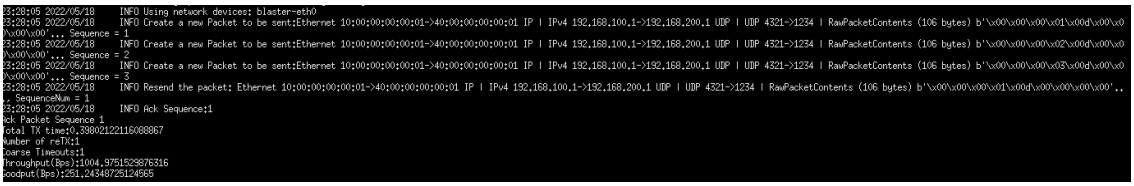
发生了两次超时，重传三个分组，虽然基数较小不具有代表性但基本呼应了0.19的丢包率。



这里和**超时时间为发送一个分组的传输时间的三倍**对应，在没有收到序号1的分组的确认信息的情况下，**没有去发送序号为4的分组而是重传了序号1的分组**，并且通过重传收到了确认。

  - **但是结合wireshark的抓包内容发现并！不！是！被middlebox丢包**，而是**ACK没有到达先发生了超时，所以产生了冗余ack。**

    并且在序号10的分组传送时也发生了类似的情况。

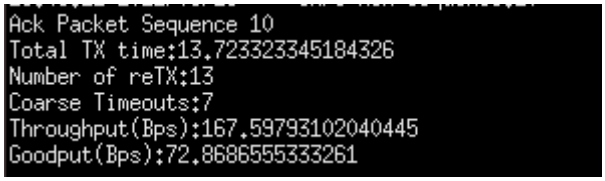


  - 在wireshark中从blaster端统计得到**13个向blastee发送**的分组，而**反过来统计到12个分组**，说明在middlebox中出现了一次丢包，查看日志后发现**序号8分组发送了两次分组仅仅收到了一次ACK**，说明出现了丢包。

- 第三组参数：增大了Timeout的值和丢包率，使丢包的情况更加突出而不容易出现冗余ACK。

```
middlebox# swyard middlebox.py -g 'dropRate=0.66'
blastee# swyard blastee.py -g 'blasterIp=192.168.100.1 num=10'
blaster# swyard blaster.py -g 'blasteeIp=192.168.200.1 num=10
length=100 senderWindow=5 timeout=1500 recvTimeout=100'
```

虽然只发送了十个分组，依赖惊人的丢包率和超时时间，得到了**惊人的重传率（130%）和网速。**并且在wireshark中捕获到了多至35个分组。**可见拥塞状态下的网络通信体验感极差。**



另外，也如我所料，将**超时时间拉长后，冗余ACK就可以消除**，在wireshark中发现收到了**恰好10个回复分组**。而在发送的分组中，**只有10/25 = 40%的分组到达了接收端**，和设定的丢包率0.66接近。

5.核心代码

task 2:Middlebox

- **修改以太网包头并且转发到另一个端口**

```
#middlebox.py/handle_packet
#此处以middlebox-eth0接收到分组为例
...#收到分组并即将转发
    IPv4 = packet[1]
    IPv4.ttl = IPv4.ttl - 1#跳数减一？意思意思，实际上封装的可能不止一个
     Ethernet = packet[0]
     intf = self.net.interface_by_name("middlebox-eth1")
     Ethernet.src = intf.ethaddr#将源mac地址修改为发出分组的端口地址
     Ethernet.dst = '20:00:00:00:00:01'#目的端口地址
     self.net.send_packet("middlebox-eth1", packet)
```

- **实现随机丢包**

```
impoart random#利用python随机数库
#middlebox.py/handle_packet
rand = random.uniform(0,1);#生成在[0,1]之间的随机数
   if (rand > self.dropRate):
        #转发分组
```

task 3:Blastee

主函数逻辑较为简单，构造对应的ACK分组并发送，略。

- **构造ACK分组**

```
    def Create_ACK_packet(self, recv_packet, intf_name):
        intf = self.net.interface_by_name(intf_name);
        #Ethernet
        recv_Ethernet_header = recv_packet.get_header_by_name("Ethernet")
        ACK_Ethernet_header = Ethernet()
        ACK_Ethernet_header.ethertype = EtherType.IPv4
        ACK_Ethernet_header.src = intf.ethaddr
        ACK_Ethernet_header.dst = recv_Ethernet_header.src
        #IPv4
        ACK_IPv4_header = IPv4()
        ACK_IPv4_header.protocol = IPProtocol.UDP
        ACK_IPv4_header.src = intf.ipaddr
        ACK_IPv4_header.dst = self.blasterIp
        ACK_IPv4_header.ttl = 64
        #UDP
```

```
        ACK_UDP_header = UDP()
        ACK_UDP_header.src = 1234
        ACK_UDP_header.dst = 4321
        #RawPacketContent
        recv_RawPckectContent_header = recv_packet.get_header_by_name("RawPacketContents")
        recv_RawPacketContent = recv_RawPckectContent_header.to_bytes()
        #从收到的分组中获得确认序号
        recv_SequenceNum_bytes =  recv_RawPacketContent[0:4];
        recv_SequenceNum = int.from_bytes(recv_SequenceNum_bytes,'big')
        #截取收到的分组载荷的前八个字节
        recv_payload =  recv_RawPacketContent[7:15]

        ACK_SequenceNum = recv_SequenceNum;
        ACK_SequenceNum_bytes = ACK_SequenceNum.to_bytes(4,byteorder='big')
        ACK_RawPacketContents = ACK_SequenceNum_bytes + recv_payload;


        ACK_packet = Packet()
        ACK_packet += ACK_Ethernet_header
        ACK_packet += ACK_IPv4_header
        ACK_packet += ACK_UDP_header
        ACK_packet += ACK_RawPacketContents;
        log_info(f"create a new ACK:{ACK_packet}")
        return ACK_packet

```

task 4:Blaster

- **处理ACK确认分组**

```
def handle_packet(self, recv: switchyard.llnetbase.ReceivedPacket):
        _, fromIface, packet = recv
        log_debug("I got a packet")
        #或许ACK确认号
        recv_RawPacketContent_header = packet.get_header_by_name("RawPacketContents")
        recv_RawPacketContent = recv_RawPacketContent_header.to_bytes()
        recv_SequenceNum_bytes =  recv_RawPacketContent[0:4]
        recv_SequenceNum = int.from_bytes(recv_SequenceNum_bytes,'big')
        log_info(f"Ack Sequence:{recv_SequenceNum}")

        #如果确认分组在滑动窗口中并且处于未确认状态
        if recv_SequenceNum in range(self.Window.LHS, self.Window.RHS + 1):
            if self.Window.SW[recv_SequenceNum][1] == 'NotAck':
                #计算输出内容
                self.TotalSuccSendbytes += self.length
                Throughput = self.TotalSendbytes / (time.time() - self.Starttime)
                Goodput = self.TotalSuccSendbytes / (time.time() - self.Starttime)
                #Goodput =
                print(f"Ack Packet Sequence {recv_SequenceNum} \nTotal TX time:{time.time() - self.Starttime} \nNumber of reTX:{self.reTX} \nCoarse Timeouts:{self.coarseTO}")
                print(f"Throughput(Bps):{Throughput}")
```

```
                print(f"Goodput(Bps):{Goodput}\n")
                #修改状态
                self.Window.SW[recv_SequenceNum][1] = 'Ack'
```

- **构造发送分组**

```
def Create_Packet(self ,SequenceNum):
        pkt = Ethernet() + IPv4() + UDP()
        pkt[0].ethertype = EtherType.IPv4
        pkt[0].src = '10:00:00:00:00:01'
        pkt[0].dst = '40:00:00:00:00:01'

        pkt[1].protocol = IPProtocol.UDP
        pkt[1].src = IPv4Address('192.168.100.1')
        pkt[1].dst = self.blasteeIp
        pkt[1].ttl = 64

        pkt[2].src = 4321
        pkt[2].dst = 1234
        SequenceNum_bytes = SequenceNum.to_bytes(4,byteorder='big')
        #log_info(f"{SequenceNum_bytes}")
        #Test_SequenceNum = int.from_bytes(SequenceNum_bytes,'big')
        #log_info(f"{Test_SequenceNum}")
        length_bytes = self.length.to_bytes(2,byteorder='big')
        payload = 0
        payload_bytes = payload.to_bytes(self.length,byteorder="big")
        RawPacketContents_bytes = SequenceNum_bytes + length_bytes +
payload_bytes

        pkt += RawPacketContents_bytes
        log_info(f"Create a new Packet to be sent:{pkt} Sequence =
{SequenceNum}")
        #log_info(f"RawPacket class:{type(RawPacketContents_bytes)}")


        return pkt
```

- **维护滑动窗口**

```
#根据ACK状态收缩滑动窗口
list = range(self.Window.LHS, self.Window.RHS + 1)
        for idx in list:
            if self.Window.SW[idx][1] == 'Ack':
                self.Window.LHS += 1;
                self.Window.timer = time.time()
                self.Window.ResendHead = -1
            else:
                break
            #if self.Window.LHS == self.num + 1:
             #   self.net.shutdown()

#滑动窗口长度小于最大长度，扩展滑动窗口
        if self.Window.RHS - self.Window.LHS + 1 < 5:
            if self.Window.RHS < self.num:
                #while self.Window.RHS - self.Window.LHS + 1 < 5 :
                    # if self.Window.RHS < self.num:
```

```
                self.Window.RHS += 1
                send_packet = self.Create_Packet(self.Window.RHS)
                self.TotalSendbytes += self.length
                intf =
self.net.interface_by_ipaddr(IPv4Address('192.168.100.1'))
                self.net.send_packet(intf.name, send_packet)
                self.Window.SW[self.Window.RHS] = [send_packet,
'NotAck',time.time(),0]
```

- **超时重传机制**

```
        #超时，刷新超时重传状态到出生点（x）
        if self.Window.timer + self.timeout < time.time() and
self.Window.LHS <= self.num:#timeout
            self.coarseTO += 1
            self.Window.ResendHead = self.Window.LHS
            self.Window.ResendPos = self.Window.ResendHead
            send_packet = self.Window.SW[self.Window.ResendPos][0]
            intf =
self.net.interface_by_ipaddr(IPv4Address('192.168.100.1'))
            intf =
self.net.interface_by_ipaddr(IPv4Address('192.168.100.1'))
            RawPackectContent_header =
send_packet.get_header_by_name("RawPacketContents")

            RawPackectContent = RawPackectContent_header.to_bytes()
            SequenceNum_bytes =  RawPackectContent[0:4];
                #log_info(f"{SequenceNum_bytes}")
            SequenceNum = int.from_bytes(SequenceNum_bytes,'big')
            log_info(f"Resend the packet: {send_packet}, SequenceNum =
{SequenceNum}")
                #self.Window.SW[idx][3] += 1
            self.reTX += 1
            self.TotalSendbytes += self.length
            self.net.send_packet(intf.name, send_packet)
            if self.Window.ResendPos == self.Window.RHS:
                self.Window.ResendPos = self.Window.ResendHead
            else:
                self.Window.ResendPos = self.Window.ResendPos + 1
            while self.Window.SW[self.Window.ResendPos][1] =='Ack' and
self.Window.ResendPos != self.Window.ResendHead:
                if self.Window.ResendPos == self.Window.RHS:
                    self.Window.ResendPos = self.Window.ResendHead
                else:
                    self.Window.ResendPos = self.Window.ResendPos + 1
            #list = range(self.Window.LHS, self.Window.RHS + 1)
            #log_info(f"{list}")
            #for idx in list:


            self.Window.timer = time.time()

        # Creating the headers for the packet
        #正在处于一轮未被打断的超时重传中
        elif self.Window.ResendHead != -1:
            if self.Window.ResendPos == self.Window.ResendHead:#结束了一
轮重传，打断重传重新记时
```

```
                    self.Window.ResendHead = -1
            else:#否则按顺序重传分组，每次传送一个分组
                send_packet = self.Window.SW[self.Window.ResendPos][0]
                intf =
self.net.interface_by_ipaddr(IPv4Address('192.168.100.1'))
                intf =
self.net.interface_by_ipaddr(IPv4Address('192.168.100.1'))
                RawPackectContent_header =
send_packet.get_header_by_name("RawPacketContents")

                RawPackectContent = RawPackectContent_header.to_bytes()
                SequenceNum_bytes =  RawPackectContent[0:4];
                #log_info(f"{SequenceNum_bytes}")
                SequenceNum = int.from_bytes(SequenceNum_bytes,'big')
                log_info(f"Resend the packet: {send_packet}, SequenceNum
= {SequenceNum}")
                #self.Window.SW[idx][3] += 1
                self.reTX += 1
                self.TotalSendbytes += self.length
                self.net.send_packet(intf.name, send_packet)
                if self.Window.ResendPos == self.Window.RHS:
                    self.Window.ResendPos = self.Window.ResendHead
                else:
                    self.Window.ResendPos = self.Window.ResendPos + 1
                while self.Window.SW[self.Window.ResendPos][1] =='Ack'
and self.Window.ResendPos != self.Window.ResendHead:
                    if self.Window.ResendPos == self.Window.RHS:
                        self.Window.ResendPos = self.Window.ResendHead
                    else:
                        self.Window.ResendPos = self.Window.ResendPos +
1
```

6.总结与感想

- 希望下一届的实现手册对于重传时间，计时器更新的细节有更好的规约。写完以后改来改去真的有点痛苦QAQ。