

南京大学本科生实验报告

- 课程名称：计算机网络
助教：

任课教师：田臣/李文中

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

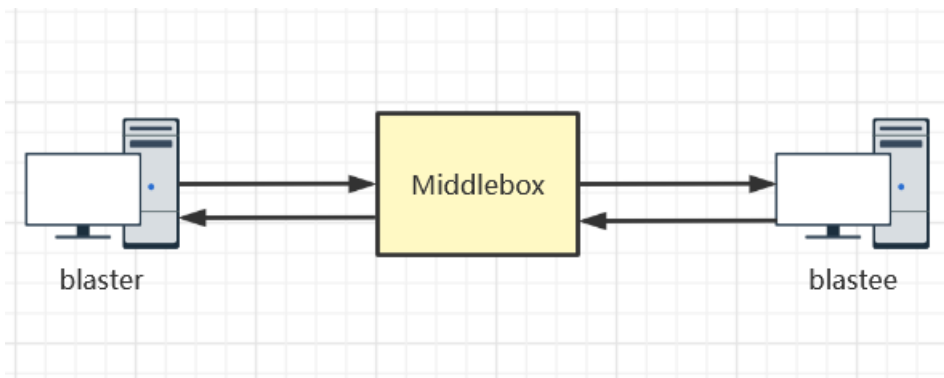
- 进一步熟悉Switchyard框架。
通过自行学习Switchyard相关API的实现和接口进一步熟悉lab提供的Switchyard框架。
- 学习端到端可靠通信机制
了解端到端通信通过等待确认方式或滑动窗口分组确认的方式来实现IP尽力而为服务上的可靠通信机制。
- 学习端到端通信发送端滑动窗口工作方式并简要实现
深入学习滑动窗口的工作细节并在Switchyard框架中实现简单的基于滑动窗口的端到端可靠通信。

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。

```

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

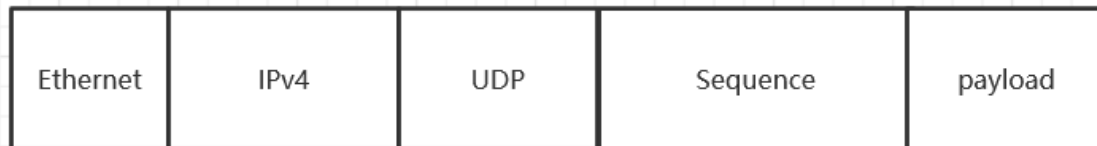
- **载荷**

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：

```

1 | 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：

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

◦ 构造发送分组

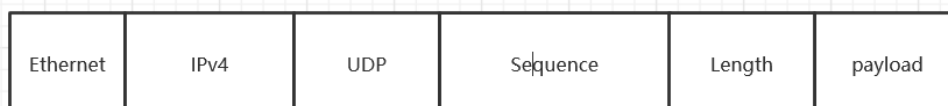
- 构造转发包头：这一部分和task3重合，不赘述。

- 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。

```
1 SequenceNum_bytes = SequenceNum.to_bytes(4,byteorder='big')
2 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：

```
1 #class window:
2     self.SW = dict()#
3     self.Maxsize = size
4     self.LHS = 1
5     self.RHS = 0
6     self.ResendHead = -1
7     self.ResendPos = 0
8     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。

```
1 ...
2         if self.window.SW[idx][1] == 'Ack':
3             self.window.LHS += 1;
4             self.window.timer = time.time()
5             self.window.ResendHead = -1
6         else:
7             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**。

```
1 if self.window.RHS - self.window.LHS + 1 < 5:
2     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。

```
1 middlebox# swyard middlebox.py -g 'dropRate=0'
2 blasteer# swyard blasteer.py -g 'blasterIp=192.168.100.1 num=10'
3 blaster# swyard blaster.py -g 'blasteerIp=192.168.200.1 num=10 length=100
senderwindow=5 timeout=300 rcvTimeout=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。

```
23:19:56 2022/05/18      INFO Ack Sequence:10
Ack Packet Sequence 10
Total TX time:1.3253860473632812
Number of reTX:0
Coarse Timeouts:0
Throughput(Bps):754.4979678240913
Goodput(Bps):754.4975606526921
```

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。

1 0.000000000	192.168.100.1	192.168.200.1	UDP	148 4321 → 1234	Len=106
2 0.102975582	192.168.100.1	192.168.200.1	UDP	148 4321 → 1234	Len=106
3 0.104904364	192.168.200.1	192.168.100.1	UDP	54 1234 → 4321	Len=12
4 0.218351535	192.168.200.1	192.168.100.1	UDP	54 1234 → 4321	Len=12
5 0.279184677	192.168.100.1	192.168.200.1	UDP	148 4321 → 1234	Len=106
6 0.383777979	192.168.100.1	192.168.200.1	UDP	148 4321 → 1234	Len=106
7 0.420177237	192.168.200.1	192.168.100.1	UDP	54 1234 → 4321	Len=12
8 0.485118231	192.168.100.1	192.168.200.1	UDP	148 4321 → 1234	Len=106
9 0.587455050	192.168.100.1	192.168.200.1	UDP	148 4321 → 1234	Len=106
10 0.624743133	192.168.200.1	192.168.100.1	UDP	54 1234 → 4321	Len=12
11 0.624880528	192.168.200.1	192.168.100.1	UDP	54 1234 → 4321	Len=12
12 0.689709132	192.168.100.1	192.168.200.1	UDP	148 4321 → 1234	Len=106

- 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。

```
1 middlebox# swyard middlebox.py -g 'dropRate=0.19'
2 blasteer# swyard blasteer.py -g 'blasterIp=192.168.100.1 num=10'
3 blaster# swyard blaster.py -g 'blasteerIp=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。

```
23:28:07 2022/05/18      INFO Ack Sequence:8
Ack Packet Sequence 8
Total TX time:1.7635812759399414
Number of reTX:3
Coarse Timeouts:2
Throughput(Bps):737.1377222308068
Goodput(Bps):567.0285571602175
```

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。

```

#28:28:05 2022/05/18      [INFO Using network device: blaster-eth0]
#28:28:05 2022/05/18      [INFO Create a new packet to be sent; Ethernet 10:00:00:00:00:01->40:00:00:00:00:01 IP | IPv4 192.168.100.1->192.168.200.1 UDP | UDP 4321->1234 | RawPacketContents (106 bytes) b'\x00\x00\x00\x01\x00\x00\x00\x00'
#28:28:05 2022/05/18      [INFO Create a new packet to be sent; Ethernet 10:00:00:00:00:01->40:00:00:00:00:01 IP | IPv4 192.168.100.1->192.168.200.1 UDP | UDP 4321->1234 | RawPacketContents (106 bytes) b'\x00\x00\x00\x01\x00\x00\x00\x00'
#28:28:05 2022/05/18      [INFO Create a new packet to be sent; Ethernet 10:00:00:00:00:01->40:00:00:00:00:01 IP | IPv4 192.168.100.1->192.168.200.1 UDP | UDP 4321->1234 | RawPacketContents (106 bytes) b'\x00\x00\x00\x01\x00\x00\x00\x00'
#28:28:05 2022/05/18      [INFO Resend the packet; Ethernet 10:00:00:00:00:01->40:00:00:00:00:01 IP | IPv4 192.168.100.1->192.168.200.1 UDP | UDP 4321->1234 | RawPacketContents (106 bytes) b'\x00\x00\x00\x01\x00\x00\x00\x00'
#28:28:05 2022/05/18      [INFO Ack Sequence=1]
#28:28:05 2022/05/18      [INFO Packet Sequence=1]
Total TX time: 339012216/68867
Number of reTxs: 1
Close Timeout(s): 1
throughput(Bps)=1104, 976133876316
throughput(Bps)=1201, 454672614989

```

- 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。

```
1 middlebox# swyard middlebox.py -g 'dropRate=0.66'
2 blastee# swyard blastee.py -g 'blasterIp=192.168.100.1 num=10'
3 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。**

```
Ack Packet Sequence 10
Total TX time:13.723323345184326
Number of reTX:13
Coarse Timeouts:7
Throughput(Bps):167.59793102040445
Goodput(Bps):72.8686555333261
```

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

```
1  #middlebox.py/handle_packet
2  #此处以middlebox-eth0接收到分组为例
3  ...#收到分组并即将转发
4      IPv4 = packet[1]
5      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
6      Ethernet = packet[0]
7      intf = self.net.interface_by_name("middlebox-eth1")
8      Ethernet.src = intf.ethaddr#将源mac地址修改为发出分组的端口地址
9      Ethernet.dst = '20:00:00:00:00:01'#目的端口地址
10     self.net.send_packet("middlebox-eth1", packet)
```

- 实现随机丢包

```
1  import random#利用python随机数库
2  #middlebox.py/handle_packet
3  rand = random.uniform(0,1);#生成在[0,1]之间的随机数
4      if (rand > self.dropRate):
5          #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

```
1      def Create_ACK_packet(self, recv_packet, intf_name):
2          intf = self.net.interface_by_name(intf_name);
3          #Ethernet
4          recv_Ethernet_header = recv_packet.get_header_by_name("Ethernet")
5          ACK_Ethernet_header = Ethernet()
6          ACK_Ethernet_header.ethertype = EtherType.IPv4
7          ACK_Ethernet_header.src = intf.ethaddr
8          ACK_Ethernet_header.dst = recv_Ethernet_header.src
9          #IPv4
10         ACK_IPv4_header = IPv4()
11         ACK_IPv4_header.protocol = IPProtocol.UDP
12         ACK_IPv4_header.src = intf.ipaddr
13         ACK_IPv4_header.dst = self.blastIp
14         ACK_IPv4_header.ttl = 64
15         #UDP
```

```

16     ACK_UDP_header = UDP()
17     ACK_UDP_header.src = 1234
18     ACK_UDP_header.dst = 4321
19     #RawPacketContent
20     recv_RawPckectContent_header =
recv_packet.get_header_by_name("RawPacketContents")
21     recv_RawPacketContent = recv_RawPckectContent_header.to_bytes()
22     #从收到的分组中获得确认序号
23     recv_SequenceNum_bytes = recv_RawPacketContent[0:4];
24     recv_SequenceNum = int.from_bytes(recv_SequenceNum_bytes,'big')
25     #截取收到的分组载荷的前八个字节
26     recv_payload = recv_RawPacketContent[7:15]
27
28     ACK_SequenceNum = recv_SequenceNum;
29     ACK_SequenceNum_bytes = ACK_SequenceNum.to_bytes(4,byteorder='big')
30     ACK_RawPacketContents = ACK_SequenceNum_bytes + recv_payload;
31
32
33     ACK_packet = Packet()
34     ACK_packet += ACK_Ethernet_header
35     ACK_packet += ACK_IPv4_header
36     ACK_packet += ACK_UDP_header
37     ACK_packet += ACK_RawPacketContents;
38     log_info(f"create a new ACK:{ACK_packet}")
39     return ACK_packet
40

```

task 4:Blaster

- 处理ACK确认分组

```

1  def handle_packet(self, recv: switchyard.llnetbase.ReceivedPacket):
2      _, fromIface, packet = recv
3      log_debug("I got a packet")
4      #或许ACK确认号
5      recv_RawPacketContent_header =
packet.get_header_by_name("RawPacketContents")
6      recv_RawPacketContent = recv_RawPacketContent_header.to_bytes()
7      recv_SequenceNum_bytes = recv_RawPacketContent[0:4]
8      recv_SequenceNum = int.from_bytes(recv_SequenceNum_bytes,'big')
9      log_info(f"Ack Sequence:{recv_SequenceNum}")
10
11     #如果确认分组在滑动窗口中并且处于未确认状态
12     if recv_SequenceNum in range(self.Window.LHS, self.Window.RHS +
1):
13         if self.window.SW[recv_SequenceNum][1] == 'NotAck':
14             #计算输出内容
15             self.TotalSuccSendbytes += self.length
16             Throughput = self.TotalSendbytes / (time.time() -
self.Starttime)
17             Goodput = self.TotalSuccSendbytes / (time.time() -
self.Starttime)
18             #Goodput =
19             print(f"Ack Packet Sequence {recv_SequenceNum} \nTotal
TX time:{time.time() - self.Starttime} \nNumber of reTX:{self.reTX}
\nCoarse Timeouts:{self.coarseTO}")
20             print(f"Throughput(Bps):{Throughput}")

```

```

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

```

- 构造发送分组

```

1  def Create_Packet(self ,SequenceNum):
2      pkt = Ethernet() + IPv4() + UDP()
3      pkt[0].ethertype = EtherType.IPv4
4      pkt[0].src = '10:00:00:00:00:01'
5      pkt[0].dst = '40:00:00:00:00:01'
6
7      pkt[1].protocol = IPProtocol.UDP
8      pkt[1].src = IPv4Address('192.168.100.1')
9      pkt[1].dst = self.blasteeIp
10     pkt[1].ttl = 64
11
12     pkt[2].src = 4321
13     pkt[2].dst = 1234
14     SequenceNum_bytes = SequenceNum.to_bytes(4,byteorder='big')
15     #log_info(f"{SequenceNum_bytes}")
16     #Test_SequenceNum = int.from_bytes(SequenceNum_bytes,'big')
17     #log_info(f"{Test_SequenceNum}")
18     length_bytes = self.length.to_bytes(2,byteorder='big')
19     payload = 0
20     payload_bytes = payload.to_bytes(self.length,byteorder="big")
21     RawPacketContents_bytes = SequenceNum_bytes + length_bytes +
payload_bytes
22
23     pkt += RawPacketContents_bytes
24     log_info(f"Create a new Packet to be sent:{pkt} Sequence =
{SequenceNum}")
25     #log_info(f"RawPacket class:{type(RawPacketContents_bytes)}")
26
27
28     return pkt

```

- 维护滑动窗口

```

1  #根据ACK状态收缩滑动窗口
2  list = range(self.window.LHS, self.window.RHS + 1)
3      for idx in list:
4          if self.window.SW[idx][1] == 'Ack':
5              self.window.LHS += 1;
6              self.window.timer = time.time()
7              self.window.ResendHead = -1
8          else:
9              break
10         #if self.window.LHS == self.num + 1:
11             # self.net.shutdown()
12
13     #滑动窗口长度小于最大长度，扩展滑动窗口
14     if self.window.RHS - self.window.LHS + 1 < 5:
15         if self.window.RHS < self.num:
16             #while self.window.RHS - self.window.LHS + 1 < 5 :
17                 # if self.window.RHS < self.num:

```



```

18         self.window.RHS += 1
19         send_packet = self.Create_Packet(self.window.RHS)
20         self.TotalSendbytes += self.length
21         intf =
self.net.interface_by_ipaddr(IPv4Address('192.168.100.1'))
22         self.net.send_packet(intf.name, send_packet)
23         self.window.SW[self.window.RHS] = [send_packet,
'NotAck',time.time(),0]

```

- 超时重传机制

```

1         #超时，刷新超时重传状态到出生点 (x)
2         if self.window.timer + self.timeout < time.time() and
self.window.LHS <= self.num:#timeout
3             self.coarseTO += 1
4             self.window.ResendHead = self.window.LHS
5             self.window.ResendPos = self.window.ResendHead
6             send_packet = self.window.SW[self.window.ResendPos][0]
7             intf =
self.net.interface_by_ipaddr(IPv4Address('192.168.100.1'))
8             intf =
self.net.interface_by_ipaddr(IPv4Address('192.168.100.1'))
9             RawPacketContent_header =
send_packet.get_header_by_name("RawPacketContents")
10
11             RawPacketContent = RawPacketContent_header.to_bytes()
12             SequenceNum_bytes = RawPacketContent[0:4];
13             #log_info(f"{SequenceNum_bytes}")
14             SequenceNum = int.from_bytes(SequenceNum_bytes, 'big')
15             log_info(f"Resend the packet: {send_packet}, SequenceNum =
{SequenceNum}")
16             #self.window.SW[idx][3] += 1
17             self.reTX += 1
18             self.TotalSendbytes += self.length
19             self.net.send_packet(intf.name, send_packet)
20             if self.window.ResendPos == self.window.RHS:
21                 self.window.ResendPos = self.window.ResendHead
22             else:
23                 self.window.ResendPos = self.window.ResendPos + 1
24                 while self.window.SW[self.window.ResendPos][1] == 'Ack' and
self.window.ResendPos != self.window.ResendHead:
25                     if self.window.ResendPos == self.window.RHS:
26                         self.window.ResendPos = self.window.ResendHead
27                     else:
28                         self.window.ResendPos = self.window.ResendPos + 1
29                 #list = range(self.window.LHS, self.window.RHS + 1)
30                 #log_info(f"{list}")
31                 #for idx in list:
32
33
34             self.window.timer = time.time()
35
36             # Creating the headers for the packet
37             #正在处于一轮未被打断的超时重传中
38             elif self.window.ResendHead != -1:
39                 if self.window.ResendPos == self.window.ResendHead:#结束了一
轮重传，打断重传重新计时

```

```

40         self.window.ResendHead = -1
41     else: #否则按顺序重传分组，每次传送一个分组
42         send_packet = self.window.SW[self.window.ResendPos][0]
43         intf =
self.net.interface_by_ipaddr(IPv4Address('192.168.100.1'))
44         intf =
self.net.interface_by_ipaddr(IPv4Address('192.168.100.1'))
45         RawPacketContent_header =
send_packet.get_header_by_name("RawPacketContents")
46
47         RawPacketContent = RawPacketContent_header.to_bytes()
48         SequenceNum_bytes = RawPacketContent[0:4];
49         #log_info(f"{SequenceNum_bytes}")
50         SequenceNum = int.from_bytes(SequenceNum_bytes, 'big')
51         log_info(f"Resend the packet: {send_packet}, SequenceNum
= {SequenceNum}")
52         #self.window.SW[idx][3] += 1
53         self.reTX += 1
54         self.TotalSendbytes += self.length
55         self.net.send_packet(intf.name, send_packet)
56         if self.window.ResendPos == self.window.RHS:
57             self.window.ResendPos = self.window.ResendHead
58         else:
59             self.window.ResendPos = self.window.ResendPos + 1
60         while self.window.SW[self.window.ResendPos][1] == 'Ack'
and self.window.ResendPos != self.window.ResendHead:
61             if self.window.ResendPos == self.window.RHS:
62                 self.window.ResendPos = self.window.ResendHead
63             else:
64                 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。