

# LAB6实验报告

| 专业        | 学号        | 姓名 | 开始/结束日期   | Email             |
|-----------|-----------|----|-----------|-------------------|
| 计算机科学与技术系 | 191220156 | 张桓 | 5.29~5.30 | 2659428453@qq.com |

## LAB6实验报告

实验名称

实验目的

实验内容

Task 2: Middlebox

实现逻辑

核心代码

Task3: Blastee

实现逻辑

核心代码

Task4: Blaster

准备工作

窗口的实现

窗口的动作

判断结束

Task5: Running your code

总结感想

## 实验名称

*Reliable Communication*

## 实验目的

本实验中需要建立一个端到端的**可靠通信**，包含三个部分，整体来说就是一个`blaster`通过`middlebox`发送数据包到`blastee`。由于网络层的IP服务是个“尽力而为”的服务(*best-effort service*)，这意味着在数据包在网络中会发生各种各样的错误：丢包，延迟，冗余等等。本实验就是实现一些额外的担保保证通信的可靠：

- 对每个正确收到的包`blastee`需要发送`ACK`
- 在`blaster`处维护一个大小固定的滑动窗口
- `blaster`要对未收到`ACK`的包超时重传

# 实验内容

## Task 2: Middlebox

*Middlebox*的结构很简单，只有两个端口，它的工作逻辑大概由两部分组成：

- 转发：从一个端口收到一个数据包那么只需要简单从另一个端口发出去，甚至不用查询转发表。
- 单向丢包：当从*blaster*收到发向*blastee*的数据包需要以 $P$ 的概率丢包( $0 < P < 1$ )，这个丢包是单向的，不会丢掉ACK包。

### 实现逻辑

在转发包的时候需要改写包头的MAC地址，由于*Middlebox*根本没有ARP机制，怎么知道目的MAC呢？后来根据FAQ可以知道直接查询 `start_mininet.py` 硬编码。查询可知，*Middlebox*的 `eth0` 是连接 *balster*的，MAC为 `40:00:00:00:00:01`，`eth1` 和*blastee*连接MAC为 `40:00:00:00:00:02`。  
*balster*的MAC为 `10:00:00:00:00:01`，*blastee*的MAC为 `20:00:00:00:00:01`。

实现单向的 $P$ 概率丢包时，用 `random` 库中的 `random.random()` 生成 `0-1` 之间的随机浮点数，如果随机值小于 `dropRate` 就丢包，同样根据FAQ，一个包不处理就是丢。

### 核心代码

```
1  if fromIface == "middlebox-eth0":
2      log_debug("Received from blaster")
3      if random.random() > self.dropRate:
4          packet[Ethernet].src = '40:00:00:00:00:02'
5          packet[Ethernet].dst = '20:00:00:00:00:01'
6          self.net.send_packet("middlebox-eth1", packet)
7  elif fromIface == "middlebox-eth1":
8      log_debug("Received from blastee")
9      packet[Ethernet].src = '40:00:00:00:00:01'
10     packet[Ethernet].dst = '10:00:00:00:00:01'
11     self.net.send_packet("middlebox-eth0", packet)
12 else:
13     log_debug("Oops :))")
```

参考文章：[python随机数生成方法](#)

## Task3: Blastee

*Blastee*作为接收方，在收到来自*Blaster*的数据包之后需要立即确认，他需要从包中提取序列号信息，并创建一个具有该序列号的ACK包，整体工作如下：

- 收到数据包后提取序列号信息`sequence`和有效负载`payload`作为ACK包的一部分内容，如果`payload`的长度不足8字节，需要自己补齐。
- 构造ACK包，并且指定目的IP为*blaster*发出。

## 实现逻辑

根据手册使用 `RawPacketContents` 类型，根据源码发现该类型初始化时数据都已经处理成了 `bytes` 类型，其中的 `data` 函数返回的也是 `bytes` 类型，如下：

```
class RawPacketContents(PacketHeaderBase):
    __slots__ = ['_raw']

    def __init__(self, raw=None):
        if isinstance(raw, str):
            raw = bytes(raw, 'utf8')
        elif isinstance(raw, bytes):
            pass
        else:
            raise TypeError("RawPacketContents must be initialized with either str or bytes. You gave me {}".format(raw.__class__.__name__))
        self._raw = raw

    def to_bytes(self):
        return self._raw

    @property
    def data(self):
        return self._raw
```

要从原始包中读取 `seqnum` 字段，首先需要知道原始包的结构，在 `Task4` 中可以找到原始包的结构如下：

```
<----- Switchyard headers -----> <----- Your packet header(raw bytes) -----> <-- Payload in raw bytes --->
|  ETH Hdr |  IP Hdr  |  UDP Hdr  | Sequence number(32 bits) | Length(16 bits) | Variable length payload |
```

我们用 `get_header(RawPacketContent)` 得到 `raw bytes` 之后的部分，前4个字节为 `seqnum` 字段，而 `payload` 字段从第6个字节之后开始。因此可以用 `data[:4]` 和 `data[6:]` 分别获取这两部分，如果读出的 `payload` 字段长度 < 8 字节，就填充 `'\x00'` 字节至长度为 8 字节，否则直接截取其前 8 个字节 `payload[0:8]`。

之后就是构造 `ACK` 包的结构，需要填好 `Eth` 头，`IP` 头和 `UDP` 头，以及序列号和有效负载，`ACK` 包的结构如下图，其中源和目的的 `MAC` 地址以及 `IP` 地址同之前的实现，都是通过查询 `start.mininet.py` 硬编码的。

```
<----- Switchyard headers -----> <----- Your packet header(raw bytes) -----> <-- Payload in raw bytes --->
|  ETH Hdr |  IP Hdr  |  UDP Hdr  | Sequence number(32 bits) | Payload (8 bytes) |
```

## 核心代码

为了让输出更清楚，我用 `Python` 的 `from_bytes` 内置函数将字节类型转换成整型输出方便观察，参数 `'big'` 表示用大端读取。

```
1 _, fromIface, packet = recv
2 log_debug(f"I got a packet from {fromIface}")
3 log_debug(f"Pkt: {packet}")
4 raw = packet.get_header(RawPacketContents)
5 seqnum = raw.data[:4]
6 print(f"the seqnum is: {int.from_bytes(seqnum, 'big')}")
7 payload = raw.data[6:]
8 if len(payload) < 8:
9     payload += b'\x00'*(8-len(payload))
10 else:
11     payload = payload[0:8]
12 eth = Ethernet()
13 eth.src = "10:00:00:00:00:01"
14 eth.dst = "40:00:00:00:00:01"
15 ip = IPv4(protocol = IPProtocol.UDP)
```

```

16 ip.src = '192.168.200.2/30'
17 ip.dst = '192.168.200.1/30'
18 ip.ttl = 64
19 udp = UDP()
20 ACK = eth + ip + udp + seqnum + payload
21 self.net.send_packet(fromIface, ACK)

```

参考文章: [int.from\\_bytes和int.to\\_bytes函数介绍](#)

## Task4: Blaster

*Blaster*向*Blastee*发送数据包, 并接收来自对方的*ACK*, 它需要维护一个**大小固定**的滑动窗口, 所谓发送者窗口*sender window*(*SW*)。为了解释窗口的性质, 用*LHS*和*RHS*举例, 需要满足:

- $RHS - LHS + 1 < SW$ , 即当前窗口大小不能超过窗口最大大小
- *LHS*之前的包一定都是被*ACK*过的, 即*LHS*总指向第一个未被*ACK*的包
- 为整个窗口维护一个粗糙的计时器, 当*LHS*移动时更新计时器, 当*LHS*在超时时间内没有改变就重发当前窗口中未被*ACK*的包

### 准备工作

- 可以看到窗口中的包不仅只是一个数据包, 还需要有**包的序号**, **是否确认**等等附加信息, 我们构造窗口中的包类如下: 其中 `pkt` 为数据包本身, `seqnum` 为该包的序号, `ACKed` 表示该包是否被确认, `recounts` 表示该包的重发次数, `sendTime` 表示该包的发送时间, 下面的 `pkt_win` 类就是**实际存在窗口中的数据结构**:

```

1 class pkt_win:
2     def __init__(self, seqnum, pkt):
3         self.pkt = pkt
4         self.seqnum = seqnum
5         self.ACKed = False
6         self.recounts = 0
7         self.sendTime = time()

```

- 由于发送方发送的数据包**有效负载随意**, 我们**只关注发送包的序号**, 因此构造数据包可以单独封装成一个函数 `make_packet(self, seqnum)`, 其参数也只有一个 `seqnum`, 具体函数如下。其中有效负载部分使用 `os.urandom(self.length)` 来生成随机的 `length` 长度的字节, 源和目的的*MAC*以及*IP*地址是**硬编码**进去的, 构造包的结构在已在[上文提到](#)。

```

1  def make_packet(self, seqnum):
2      eth = Ethernet()
3      eth.src = '20:00:00:00:00:01'
4      eth.dst = '40:00:00:00:00:02'
5      ip = IPv4(protocol = IPPROTO_UDP)
6      ip.src = '192.168.100.1/30'
7      ip.dst = self.blasteeIP
8      ip.ttl = 64
9      udp = UDP()
10     pkt = eth + ip + udp + seqnum.to_bytes(4, 'big') + self.length.to_bytes(2,
        'big') + os.urandom(self.length)
11     return pkt

```

- 根据手册的要求，在整个通信过程结束后还需要输出一些信息：
  - 从第一个包被发出到最后一个包被ACK之间的时间 $TX$ ，只需要在代码中记录这两个时间 `self.firstSend` 和 `self.lastACK` 即可。
  - 重发的包的数量 $reTX$ ，记录为 `self.allre_count`
  - 超时次数 $Number\ of\ coarse\ TOs$ ，记录为 `self.TOs`
  - 吞吐量，记录总时间 $TX$ ，还需要再记录总的发出包的数量(包括重发) `self.allsend_count`，用总包数\*长度/ $TX$ 即可
  - 有效吞吐量，只需用(总包数-重发数)\*长度/ $TX$ 即可

这些信息的记录更新放在下文代码中。

## 窗口的实现

接下来就是窗口的实现了，由于窗口中的包都是左出右进，这里用 `deque` 来实现，`deque` 可以设置最大长度 `maxlen = self.maxsize`，这样就保证满足了  $C_1: RHS - LHS + 1 < SW$ 。

还需要满足  $C_2: Every\ packet\ with\ sequence\ number\ S_j < S_i\ has\ been\ successfully\ ACKd$ ，可以看到

- 当窗口大小  $< maxlen$  时，可以直接 `deque.append()` 新的包到窗口，即  $RHS$  右移
- 当窗口大小  $= maxlen$  时，且  $LHS$  指向的包被ACK了，那么  $LHS$  和  $RHS$  都需要右移，同样这种情况用 `deque.append()` 可直接实现，右侧新的包的进入会挤掉最左侧已经ACK的包，这是 `deque` 的定长特性导致的。
- 当窗口最左侧的包被ACK时，可以用 `deque.popleft()` 来移动  $LHS$
- 为整个窗口维护的粗糙计时器 `window.timer`，当移动  $LHS$  时或者超时重发时，重置计时器。如果  $LHS$  值在超时时间内没有改变，那么您将重新传输该窗口中的每个非ACK包。

## 窗口的动作

下面讨论在各种情况下窗口的具体动作：

- 当收到一个包的时候，这必定是一个ACK包，我们需要检查包中的 `seqnum` 来确定ACK的是哪一个包，类似上面Blastee的实现，同样用到 `packet.get_header(RawPacketContents)` 找到包的序号字段为前4个字节，然后再遍历窗口中的包看ACK的是其中哪一个，更新其信息即可，然后调用接下来的 `window_send()` 函数重传或发一个新数据包，代码如下：

```

1  def handle_packet(self, recv: switchyard.llnetbase.ReceivedPacket):
2      _, fromIface, packet = recv
3      log_debug("I got a packet")
4      #first complete the ACK's work
5      raw = packet.get_header(RawPacketContents)
6      this_seqnum = int.from_bytes(raw[:4], 'big')
7      print(f"the ACK's seqnum is: {this_seqnum}")
8      for i, v in enumerate(self.window):
9          if v.seqnum == this_seqnum:
10             v.ACKed = True
11             print(f"the packet({i + 1}) in current window is ACKed!")
12             self.lastACK = max(self.lastACK, time.time())
13     #then send one pkt in this loop
14     self.window_send()

```

- 重传和发包，根据FAQ的要求可以知道，发包应该在每次 `recv_timeout` 循环发送单个数据包，这个包可以是新的数据包，也可以是窗口内超时重传的数据包。在每个循环内，不管是收到包还是没收到包，都需要重传或者发一个新的包，因此可以单独抽象出一个函数 `window_send()`，里面包含了两部分：
  - 首先检测当前窗口的计时器是否超时，然后重传一个未被ACK的包，更新窗口信息，这里面涉及到窗口计时器 `window.timer` 的维护，重发时需要重置计时器，该部分代码如下：

```

1  #first if need resend a packet
2  RESEND = False
3      if len(self.window) > 0:
4          if time.time() - self.timer > self.timeout:
5              #the window's timer is timeout
6              self.TOs += 1
7              print("the window's timer is timeout!")
8              for v in self.window:
9                  if v.ACKed == False:
10                     RESEND = True
11                     print(f"resend a unACKed packet whose seqnum is
12 {v.seqnum}")
13
14             self.net.send_packet(self.net.interface_by_name("blaster-eth0"), v.pkt)
15             v.sendTime = time.time()
16             self.timer = v.sendTime #update window's timer -> resend
17             v.recounts += 1
18             self.allre_count += 1
19             self.allsend_count += 1
20             break #only send one in a loop

```

- 如果不能重传，那就尝试发一个新的数据包，并且更新窗口信息。在还有包可发时，只有当前窗口未滿，或者窗口最左侧的包被ACK后，才能向窗口增加新的包或者滑动窗口，不同的情况对于窗

口信息的更新方法不同，具体见代码：

```
1  #if don't need resend, then send a next pkt in this loop
2  if RESEND == False and self.allcount > 0 and (len(self.window) <
    self.maxsize or self.window[0].Aked == True):
3      next_seq = 1
4      if len(self.window) > 0: #get the last seq in window
5          next_seq = self.window[-1].seqnum + 1
6          pkt = self.make_packet(next_seq) #create next pkt
7          pktWindow = pkt_win(next_seq, pkt)
8          self.net.send_packet(self.net.interface_by_name('blaster-eth0'),
    pkt)
9          print(f"send the packet whose seqnum is {next_seq}")
10
11         if len(self.window) < self.maxsize and self.window[0].Aked ==
    True: #LHS change
12             self.window.popleft() #pop the left ACKed pkt
13             self.window.append(pktWindow) #add pkt in window
14             self.timer = pktWindow.sendTime #update window's timer
15         elif len(self.window) < self.maxsize: #LHS not change
16             self.window.append(pktWindow)
17         elif self.window[0].Aked == True: #LHS change
18             self.window.popleft()
19             self.window.append(pktWindow)
20             self.timer = pktWindow.sendTime
21
22         if self.firstSend == -1.0: #the first send pkt's time
23             self.firstSend = pktWindow.sendTime
24         self.allsend_count += 1
25         self.allcount -= 1
```

## 判断结束

当没有可发的包并且所有发出去的包都收到了ACK，那么整个过程结束，需要输出之前记录的各种信息，代码如下：

```
1  def check_over(self):
2      if self.allcount > 0:
3          return False
4      for i, v in enumerate(self.window):
5          if v.Aked == False:
6              return False
7      totalTXtime = self.lastACK - self.firstSend
8      print(f"= " * 80)
9      print(f"Total TX time (in seconds): {totalTXtime}")
10     print(f"Number of reTX: {self.allre_count}")
11     print(f"Number of coarse T0s: {self.T0s}")
```

```

12     print(f"Throughput (Bps): {self.allsend_count * self.length / totalTXtime}")
13     print(f"Goodput (Bps): {(self.allsend_count - self.allre_count) * self.length
    / totalTXtime}")
14     print("=" * 80)

```

## Task5: Running your code

为了让观察更直观，我设置**blastee**发出的ACK的 $t_{tl} = 64$ ，**blaster**发出的包的 $t_{tl} = 0$ ，这样在**wireshark**抓包文件中，**blaster**发出的数据包颜色为红色，**blastee**发出的ACK包颜色为蓝色。

- 在**mininet**上运行给好的拓扑 `start_mininet.py`，然后在**blastee** **middlebox** **blaster**各自的 *xterm*上运行上文中编写好的代码，按照**手册**中给出的参数进行测试：

```

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

```

得到如下结果

- 从**blastee**视角：

```

receive a pkt with seqnum: 1
send a ACK for seqnum:1
receive a pkt with seqnum: 2
send a ACK for seqnum:2
receive a pkt with seqnum: 3
send a ACK for seqnum:3
receive a pkt with seqnum: 4
send a ACK for seqnum:4
receive a pkt with seqnum: 5
send a ACK for seqnum:5
receive a pkt with seqnum: 6
send a ACK for seqnum:6
receive a pkt with seqnum: 7
send a ACK for seqnum:7
receive a pkt with seqnum: 8
send a ACK for seqnum:8
receive a pkt with seqnum: 9
send a ACK for seqnum:9
receive a pkt with seqnum: 10
send a ACK for seqnum:10
receive a pkt with seqnum: 11

```

可以看到**blastee**的工作非常简单，就是简单收到一个包之后对其回复即可，从**wireshark**的抓包文件看到也是如此：



正在捕获 blastee-eth0

| No. | Time        | Source        | Destination   | Protocol | Length | Info          |
|-----|-------------|---------------|---------------|----------|--------|---------------|
| 1   | 0.000000000 | 192.168.100.1 | 192.168.200.1 | UDP      | 148    | 0 → 0 Len=106 |
| 2   | 0.052445239 | 192.168.200.1 | 192.168.100.1 | UDP      | 54     | 0 → 0 Len=12  |
| 3   | 0.101826957 | 192.168.100.1 | 192.168.200.1 | UDP      | 148    | 0 → 0 Len=106 |
| 4   | 0.154772884 | 192.168.200.1 | 192.168.100.1 | UDP      | 54     | 0 → 0 Len=12  |
| 5   | 0.205461868 | 192.168.100.1 | 192.168.200.1 | UDP      | 148    | 0 → 0 Len=106 |
| 6   | 0.258331045 | 192.168.200.1 | 192.168.100.1 | UDP      | 54     | 0 → 0 Len=12  |
| 7   | 0.417230174 | 192.168.100.1 | 192.168.200.1 | UDP      | 148    | 0 → 0 Len=106 |
| 8   | 0.477960492 | 192.168.200.1 | 192.168.100.1 | UDP      | 54     | 0 → 0 Len=12  |
| 9   | 0.625380607 | 192.168.100.1 | 192.168.200.1 | UDP      | 148    | 0 → 0 Len=106 |
| 10  | 0.685879050 | 192.168.200.1 | 192.168.100.1 | UDP      | 54     | 0 → 0 Len=12  |
| 11  | 0.941251817 | 192.168.100.1 | 192.168.200.1 | UDP      | 148    | 0 → 0 Len=106 |
| 12  | 1.006057370 | 192.168.200.1 | 192.168.100.1 | UDP      | 54     | 0 → 0 Len=12  |

应用显示过滤器: <Ctrl-/> 表达式...

Frame 1: 148 bytes on wire (1184 bits), 148 bytes captured (1184 bits) on interface 0  
 Ethernet II, Src: 48:00:00:00:00:02 (48:00:00:00:00:02), Dst: 28:00:00:00:00:01 (28:00:00:00:00:01)  
 Internet Protocol Version 4, Src: 192.168.100.1, Dst: 192.168.200.1  
 User Datagram Protocol, Src Port: 0, Dst Port: 0  
 Data (106 bytes)

- 从 *middlebox* 视角:

```

"Node: middlebox"
Received from blaster
Received from blaster
Received ACK from blastee
Received ACK from blastee
Received from blaster
Received ACK from blastee
Received from blaster
this packet is dropped!!!
Received from blaster
Received ACK from blastee
Received from blaster
Received ACK from blastee
Received from blaster
this packet is dropped!!!
Received from blaster
Received ACK from blastee
Received from blaster
Received from blaster
Received from blaster
Received ACK from blastee
Received ACK from blastee
Received ACK from blastee

```

可以看到它负责直接接收来自 *blaster* 的数据包和来自 *blastee* 的 *ACK* 包, 并且以一定的概率丢掉来自 *blaster* 的数据包

- 从 *blaster* 视角:

```
"Node: blaster"
20:56:50 2021/05/30 INFO Saving iptables state and installing switchyard rules
20:56:50 2021/05/30 INFO Using network devices: blaster-eth0
send the packet whose seqnum is 1
send the packet whose seqnum is 2
send the packet whose seqnum is 3
receive the ACK's seqnum is: 2
pkt with seqnum 2 located in current window[2] is ACKed!
send the packet whose seqnum is 4
receive the ACK's seqnum is: 3
pkt with seqnum 3 located in current window[3] is ACKed!
the window's timer is timeout!
resend a unACKed packet whose seqnum is 1
send the packet whose seqnum is 5
receive the ACK's seqnum is: 4
pkt with seqnum 4 located in current window[4] is ACKed!
the window's timer is timeout!
resend a unACKed packet whose seqnum is 1
receive the ACK's seqnum is: 5
pkt with seqnum 5 located in current window[5] is ACKed!
receive the ACK's seqnum is: 1
pkt with seqnum 1 located in current window[1] is ACKed!
send the packet whose seqnum is 6
send the packet whose seqnum is 7
send the packet whose seqnum is 8
send the packet whose seqnum is 9
receive the ACK's seqnum is: 7
pkt with seqnum 7 located in current window[3] is ACKed!
send the packet whose seqnum is 10
receive the ACK's seqnum is: 8
pkt with seqnum 8 located in current window[3] is ACKed!
receive the ACK's seqnum is: 9
pkt with seqnum 9 located in current window[4] is ACKed!
receive the ACK's seqnum is: 10
pkt with seqnum 10 located in current window[5] is ACKed!
the window's timer is timeout!
```

此时窗口为1234,2被确认

超时重传窗口最左端的1

此时12345全被确认

窗口右滑此时窗口为56789

此时窗口为6 7 8 9 10

我们观察传输过程中窗口动作滑是否正确，可以看到开始时blaster发送了123三个包，这时窗口大小为3，之后收到了对2的确认，发送4之后又收到了对3的确认，这时窗口大小为4且窗口计时器超时了，需要重传包，由于每个循环只能发一个包，我的代码实现是重传优先，所以当前窗口中未被ACK的包1就被重传了。可以判断重传机制应该被正确实现了。

在图中看到当包1被确认后，此时窗口中12345都被确认，根据我的实现窗口需要右滑，由于一次只能滑动一个包，在发出包6789之后，这时的窗口为56789，且5已经在之前被ACK过了，收到对7的ACK后，根据输出信息看到包7位于当前的window[3]位置，符合我们的预期，并且发出包10后窗口再次右滑(因为最左端的包5之前被ACK过了)，窗口变成了6 7 8 9 10，这之后收到对于包8 9 10的ACK，根据输出信息可以看到分别位于window[3] window[4] window[5]的位置上，符合我们的预期。经过这些分析可以判断窗口的滑动是正确的。

下图抓包文件和上述描述一致，观察抓包文件中的Len项，根据数据包的结构，seqnum的4字节+length的2字节+payload的100字节 = 106字节，正确；根据ACK包的结构，seqnum的4字节+payload的8字节 = 12字节，也正确。

正在捕获 blaster-eth0

文件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)

应用显示过滤器: <Ctrl-/> 表达式: +

| No. | Time        | Source        | Destination   | Protocol | Length | Info          |
|-----|-------------|---------------|---------------|----------|--------|---------------|
| 1   | 0.000000000 | 192.168.100.1 | 192.168.200.1 | UDP      | 148    | 0 → 0 Len=106 |
| 2   | 0.102658574 | 192.168.100.1 | 192.168.200.1 | UDP      | 148    | 0 → 0 Len=106 |
| 3   | 0.206210207 | 192.168.100.1 | 192.168.200.1 | UDP      | 148    | 0 → 0 Len=106 |
| 4   | 0.237047980 | 192.168.100.1 | 192.168.100.1 | UDP      | 54     | 0 → 0 Len=12  |
| 5   | 0.294493726 | 192.168.100.1 | 192.168.200.1 | UDP      | 148    | 0 → 0 Len=106 |
| 6   | 0.340122351 | 192.168.100.1 | 192.168.100.1 | UDP      | 54     | 0 → 0 Len=12  |
| 7   | 0.392335280 | 192.168.100.1 | 192.168.200.1 | UDP      | 148    | 0 → 0 Len=106 |
| 8   | 0.447656009 | 192.168.100.1 | 192.168.100.1 | UDP      | 54     | 0 → 0 Len=12  |
| 9   | 0.495032975 | 192.168.100.1 | 192.168.200.1 | UDP      | 148    | 0 → 0 Len=106 |
| 10  | 0.658410855 | 192.168.100.1 | 192.168.100.1 | UDP      | 54     | 0 → 0 Len=12  |
| 11  | 0.713374732 | 192.168.100.1 | 192.168.200.1 | UDP      | 148    | 0 → 0 Len=106 |
| 12  | 0.868137487 | 192.168.100.1 | 192.168.100.1 | UDP      | 54     | 0 → 0 Len=12  |

Frame 1: 148 bytes on wire (1184 bits), 148 bytes captured (1184 bits) on interface 0  
Ethernet II, Src: 20:00:00:00:00:01 (20:00:00:00:00:01), Dst: 40:00:00:00:00:02 (40:00:00:00:00:02)  
Internet Protocol Version 4, Src: 192.168.100.1, Dst: 192.168.200.1  
User Datagram Protocol, Src Port: 0, Dst Port: 0  
Data (106 bytes)

最后的输出信息如下：

```
the window's timer is timeout!
resend a unACKed packet whose seqnum is 99
receive the ACK's seqnum is: 99
pkt with seqnum 99 located in current window[4] is ACKed!
=====
Total TX time (in seconds): 21.960185289382935
Number of reTX: 39
Number of coarse T0s: 41
Throughput (Bps): 632.9636939229387
Goodput (Bps): 455.36956397333716
=====
the transmission is overed!!!
20:57:12 2021/05/30      INFO Restoring saved iptables state
```

以上抓包文件放在 `/report` 文件夹中，以 `lab_6_1` 开头。

- 我们将参数中窗口长度 `senderWindow` 设置为3，包的总数 `num` 设置为10，再次测验：
  - `blaster`视角：

```
"Node: blaster"
21:37:54 2021/05/30      INFO Saving iptables state and installing switchyard rules
21:37:54 2021/05/30      INFO Using network devices: blaster-eth0
send the packet whose seqnum is 1
send the packet whose seqnum is 2
send the packet whose seqnum is 3
the window's timer is timeout!
resend a unACKed packet whose seqnum is 1
receive the ACK's seqnum is: 1
pkt with seqnum 1 located in current window[1] is ACKed!
send the packet whose seqnum is 4
receive the ACK's seqnum is: 2
pkt with seqnum 2 located in current window[1] is ACKed!
send the packet whose seqnum is 5
receive the ACK's seqnum is: 3
pkt with seqnum 3 located in current window[1] is ACKed!
send the packet whose seqnum is 6
receive the ACK's seqnum is: 1
receive the ACK's seqnum is: 4
pkt with seqnum 4 located in current window[1] is ACKed!
send the packet whose seqnum is 7
receive the ACK's seqnum is: 6
pkt with seqnum 6 located in current window[2] is ACKed!
receive the ACK's seqnum is: 7
pkt with seqnum 7 located in current window[3] is ACKed!
the window's timer is timeout!
resend a unACKed packet whose seqnum is 5
receive the ACK's seqnum is: 5
pkt with seqnum 5 located in current window[1] is ACKed!
send the packet whose seqnum is 8
send the packet whose seqnum is 9
send the packet whose seqnum is 10
receive the ACK's seqnum is: 8
pkt with seqnum 8 located in current window[1] is ACKed!
receive the ACK's seqnum is: 9
pkt with seqnum 9 located in current window[2] is ACKed!
the window's timer is timeout!
resend a unACKed packet whose seqnum is 10
receive the ACK's seqnum is: 10
pkt with seqnum 10 located in current window[3] is ACKed!
```

窗口为123,均未确认,超时重传1

1被确认窗口右滑变为234

右滑变为345

右滑变为456

1的冗余

右滑变为567

5未被确认超时重传

窗口变为8 9 10

10未收到ACK超时重

图中已经标出窗口每一步的动作，所有动作完全符合预期，其中有一点特殊的地方就是有个包1的冗余ACK，我们分析其原因：包1在窗口超时后都没有收到ACK，于是重发了包1，以为包1丢包了，但实际上包1并没有被丢掉，只是因为某种原因被延迟了，因此最终blaster收到了两个包1，因此会发出两个ACK，结合balstee视角：

```
"Node: blastee"
21:36:59 2021/05/30 INFO Saving iptables state and installing switchyard rules
21:36:59 2021/05/30 INFO Using network devices: blastee-eth0
receive a pkt with seqnum: 1
send a ACK for seqnum:1
receive a pkt with seqnum: 2
send a ACK for seqnum:2
receive a pkt with seqnum: 3
send a ACK for seqnum:3
receive a pkt with seqnum: 1
send a ACK for seqnum:1
receive a pkt with seqnum: 4
send a ACK for seqnum:4
receive a pkt with seqnum: 6
send a ACK for seqnum:6
receive a pkt with seqnum: 7
send a ACK for seqnum:7
receive a pkt with seqnum: 5
send a ACK for seqnum:5
receive a pkt with seqnum: 8
send a ACK for seqnum:8
receive a pkt with seqnum: 9
send a ACK for seqnum:9
receive a pkt with seqnum: 10
send a ACK for seqnum:10
```

可以发现`blastee`确实收到了两个包1，只是由于其中的一个`ACK`在途中被延迟了，导致了`blaster`的重传和冗余`ACK`。

最后的输出信息如下：

```
=====
Total TX time (in seconds): 1.6723885536193848
Number of reTX: 3
Number of coarse T0s: 3
Throughput (Bps): 777.3313188412698
Goodput (Bps): 597.9471683394383
=====
the transimtion is overed!!!
21:37:57 2021/05/30 INFO Restoring saved iptables state
```

综上所述，实现的窗口机制应该是正确的。

以上抓包文件放在 `/report` 文件夹中，以 `lab_6_2` 开头。

## 总结感想

这次实验相比前三次实现的路由器来说还是比较简单的，需要关注的地方只有滑动窗口的维护，使用 `deque` 的想法是想到窗口滑动的情形，完全符合 `deque` 的左端出右端入的性质。我认为我做的比较好的地方一个就是用 `deque` 实现窗口，利用其定长特性，完全避免了`LHS`和`RHS`两个指针操作。

再有一个就是对于窗口内数据结构 `pkt_win` 的抽象，每个包不仅仅是一个数据包，还有这个包是否被 `ACK`，包的序号，包的发送时间等等信息，抽象成一个类就相当于一个结构体，操作起来很清晰方便。

至于窗口该怎么维护，我完全按照了手册提供的思路，和课上学的还是有些不同的，本实验中采用的是窗口计时器，当整个窗口的计时器超时后重发窗口内的所有未 $ACK$ 的包，而且在每个循环内只能发一个包，按照助教gg的意思是这样就可以通过 `recv_time` 来实现对速率的一定的控制...总之就是完全按照手册来的，**建议FAQ**的一部分内容可以放在前面的手册中，实际上就为实现代码给了更好的提示。

参考文章：

[python随机数生成方法](#)

[int.from\\_bytes和int.to\\_bytes函数介绍](#)

最后感谢助教的耐心批改 😊😊😊