

# 计算机网络实验 Lab6实验报告

工程管理学院 计算机金融实验班 211275032 汪科

Email: [211275032@smail.nju.edu.cn](mailto:211275032@smail.nju.edu.cn)

## 计算机网络实验 Lab6实验报告

工程管理学院 计算机金融实验班 211275032 汪科

Email: [211275032@smail.nju.edu.cn](mailto:211275032@smail.nju.edu.cn)

一、实验名称: Reliable Communication

二、实验目的

三、实验内容

Middlebox

Blastee

Blaster

四、实验结果

五、总结与感想

---

## 一、实验名称: Reliable Communication

---

## 二、实验目的

构造一个可靠通信传输，由3个代理组成。

可靠通信库将实现以下功能：

1. 每个成功接收的数据包的 ACK 机制。
  2. 冲击波上的固定大小滑动窗口。
  3. 冲击波上的粗略超时，重新发送非 ACK 数据包。
-

### 三、实验内容

#### *Middlebox*

“硬编码”:



```
1  BLASTER_MAC = "10:00:00:00:00:01"
2  BLASTEE_MAC = "20:00:00:00:00:01"
```

在收到来自blaster的包时，以一定概率丢包，这里使用random模块模拟[0,1]内随机数:



```
1  if fromIface == "middlebox-eth0":
2      log_info("Received from blaster")
3      current_random = random.random()
4      log_info(f'current_random: {current_random}')
5      if current_random < self.dropRate:
6          log_info("Dropping packet")
7          return
8      packet[Ethernet].src = "40:00:00:00:00:02"
9      packet[Ethernet].dst = BLASTEE_MAC
10     self.net.send_packet("middlebox-eth1", packet)
```

在收到来自blastee的包时，不丢包:

```
1 elif fromIface == "middlebox-eth1":
2     log_info("Received from blasteer")
3     packet[Ethernet].src = "40:00:00:00:00:01"
4     packet[Ethernet].dst = BLASTER_MAC
5     self.net.send_packet("middlebox-eth0", packet)
```


## *Blastee*

在初始化函数中:

```
1 class Blastee:
2     def __init__(self, net: switchyard.llnetbase.LLNetBase, blasterIp, num):
3         self.net = net
4         self.blasterIp = blasterIp
5         self.total_pkt_num = int(num) #总共要接受的包的数量
6         self.current_num=0 #当前已经收到的包的数量
7         self.pkt_received = set() #用集合来存储已经已经收到的包序列号， 以避免对重发包的重计数。
```

- 前面是将传进来的参数存储为实例变量
- `current_num` 存储已经收到的包的数量（当这个包是第一次收到时才自增）
- `pkt_received` 是一个集合，存储已经收到的包，这是为了避免收到重复的包导致 `current_num` 计数错误。

基于上述变量，对于包的处理，我是这样做的：




```

1  del packet[Ethernet]
2  del packet[IPv4]
3  del packet[UDP]
4  #去除包头
5
6  if int.from_bytes(packet[0].to_bytes()[4:], 'big') not in self.pkt_received:
7      self.pkt_received.add(int.from_bytes(packet[0].to_bytes()[4:], 'big'))
8      self.current_num+=1
9      log_info(f"Received packet {int.from_bytes(packet[0].to_bytes()[4:], 'big')}")
10     #如果是第一次收到这个包，就把包的序列号加入集合，并且计数器加一

```

接下来是构造包和发包的过程：




```

1  pkt=Packet()
2  EthHeader=Ethernet()
3  EthHeader.src='20:00:00:00:00:01'
4  EthHeader.dst='40:00:00:00:00:02'
5  IPv4Header=IPv4()
6  IPv4Header.src='192.168.200.1'
7  IPv4Header.dst=self.blasterIp
8  IPv4Header.protocol=IPProtocol.UDP
9  IPv4Header.ttl=64
10 UDPHeader=UDP()
11 UDPHeader.src=114
12 UDPHeader.dst=514
13 #UDP随便设的
14 pkt=EthHeader+IPv4Header+UDPHeader
15 SequencePart=packet[0].to_bytes()[4:]
16 pkt+=RawPacketContents(SequencePart)
17 ACKPart=packet[0].to_bytes()[6:]+bytes(8)
18 ACKPart=ACKPart[:8]
19 pkt+=RawPacketContents(ACKPart)
20 #构造ACK包
21 log_info(f'Sending ACK packet {int.from_bytes(SequencePart,"big")}')
22 self.net.send_packet(fromIface,pkt)

```


## Blaster

初始化函数中的几个变量是比较重要的，分别简单介绍下：



```
1 self.net = net
2 self.blasteeIp = blasteeIp
3 self.num = int(num)
4 self.length = int(length)
5 self.senderWindow = int(senderWindow)
6 self.timeout = int(timeout)/1000
7 self.recvTimeout = int(recvTimeout)/1000
```

这部分主要存储接受的参数。



```
1 self.LHS=1
2 self.RHS=self.LHS+self.senderWindow-1
3 self.time=time.time()
4 self.ACKs=[False]*(self.num+1)
5 self.payloads=[None]*(self.num+1)
6 self.outport=self.net.interfaces()[0].name
7
8 self.payload_init() #初始化payloads
9 self.Retransmit_Queue=queue.Queue() #用于存储需要重传的包的序号
```

这部分存储和滑动窗口有关的变量，以及提前准备payload填充内容。

- self.ACKs是用来记录每个编号的包是否已ACK。
- self.Retransmit\_Queue是一个队列，用来存储需要重传的包的序号，它的使用将会在后面详细描述。



```
1  # 统计量
2  self.firstsendtime=1e20
3  self.lastackdtime=0
4  self.FirstSend=[True]*(self.num+1)
5  self.Retransmit_Count=0
6  self.CoarseTimeout_Count=0
7  self.Throughput=0
8  self.Goodput=0
```

这部分主要存储一些统计量。

---

功能部分的大致逻辑如下：

- 所有的发包工作都在recvTimeout超时后，由handle\_no\_packet完成。
- handle\_packet 将会在接收到 ACK 时被调用。它的作用是维护滑动窗口，并将需要发送的包添加至 self.Retransmit\_Queue。

handle\_no\_packet的代码：



```
1  def handle_no_packet(self):
2
3      if time.time()-self.time>self.timeout: #超时了
4          self.CoarseTimeout_Count+=1
5          self.time=time.time() #更新计时器
6          for NCKnum in range(self.LHS,self.RHS+1):
7              if self.ACKs[NCKnum]: #在这个区间内，如果ACK已经收到，就不用再发了
8                  continue
9              self.Retransmit_Queue.put(NCKnum) #将需要重传的包的序号加入等待队列
10         self.Retransmit_single_packet()
11     else:
12         self.Retransmit_single_packet()
```

其中，self.Retransmit\_single\_packet()的作用是取出self.Retransmit\_Queue的队首，并重传该包。

每次handle\_no\_packet被调用，self.Retransmit\_single\_packet()只会被调用一次，它最多发一个包，这就控制了发送速率。

当该函数被调用，它会首先检查是否出现了coarse\_timeout，如果没有，就重传一个包；如果超时，它首先重置coarse\_timeout的计时器，并将窗口内的所有unACKd序号加入self.Retransmit\_Queue。并重传队列里的一个包。

handle\_packet的代码：



```
1 def handle_packet(self, recv: switchyard.llnetbase.ReceivedPacket):
2     _, fromIface, packet = recv
3     self.time=time.time() #更新计时器
4     del packet[Ethernet]
5     del packet[IPv4]
6     del packet[UDP]
7     #去除包头
8
9     ACKnum=int.from_bytes(packet[0].to_bytes()[4:], 'big')
10    log_info("got a ACK packet with ACKnum: {}".format(ACKnum))
11    #获取ACK包的序列号
12
13    #遍历等待队列，如果ACKnum在等待队列中，就将其从等待队列中删除
14    for i in range(self.Retransmit_Queue.qsize()):
15        tmp=self.Retransmit_Queue.get()
16        if tmp!=ACKnum:
17            self.Retransmit_Queue.put(tmp)
18        else:
19            pass
20
21    self.ACKs[ACKnum]=True
22    # ACKnum对应的ACK标记为True
23
24    while self.ACKs[self.LHS] and self.LHS<=self.RHS: #移动LHS
25        self.LHS+=1
26        self.time=time.time() #LHS更新，重置计时器
27        if self.LHS==self.num+1:
28            break
29    # 如果LHS对应的ACK为True，就右移LHS，直到LHS对应的ACK为False或者LHS>RHS
30
31    while self.RHS<self.num and self.RHS-self.LHS+1<self.senderWindow: #移动RHS
32        self.RHS+=1
33        self.Retransmit_Queue.put(self.RHS) #将RHS+1加入等待队列
34
35    if self.LHS==self.num+1:
36        log_info("All packets have been sent")
37        self.lastackdtime=time.time()
38        self.shutdown()
```

当它被调用，首先循环队列，去掉队列中仍存在的该序号的待发包，这就避免了重发不必要的包，同时记录对应序号收到了ACK。

然后，它首先移动LHS，使得它到达下一个unACKd的序号处，每移动一次就更新计时器。

然后移动RHS，每移动一次，它必定会到达一个unACKd的序号，这时将该序号加入self.Retransmit\_Queue，直到窗口长度到达指定长度or到尾部了。



接下来说一下各类统计量的实现机制。

- Total TX time : 在 init 里 初始化了 初始发包时间 1e20 , 随后在发包时执行 `self.firstsendtime=min(self.firstsendtime,time.time())`, 这样可以避免复杂逻辑判断。在 shutdown 时记录结束时间并相减即可。
- Number of reTX: 首先初始化 `self.FirstSend=[True]*(self.num+1)`, 在重传包的时候 (注意: 首次发包也是由重传函数 `Retransmit_single_packet` 完成的), 进行判断:

```
1  if self.FirstSend[current_num]==True:
2      self.Goodput+=len(Payload)
3      self.FirstSend[current_num]=False
4  else:
5      self.Retransmit_Count+=1
6  self.Throughput+=len(Payload)
```

如果标志位为 True (初始化为 True), 就置为 False, 同时增加 Goodput。如果首次发送标志位为 False, 就将重传次数+1。

不管哪种情况都会增加 Throughput。

其余部分较为简单, 就不赘述了。

---

## 四、实验结果

数据:

```
07:48:33 2023/05/30 INFO All packets have been sent
07:48:34 2023/05/30 INFO -----
07:48:34 2023/05/30 INFO Total TX time: 17.32140612602234
07:48:34 2023/05/30 INFO Number of reTX:: 25
07:48:34 2023/05/30 INFO Number of coarse T0s: 16
07:48:34 2023/05/30 INFO Throughput: 721.650419663157
07:48:34 2023/05/30 INFO Goodput: 577.3203357305256
07:48:34 2023/05/30 INFO Restoring saved iptables state
```

wireshark抓包 (blaster) :

- [illegible]