南京大学本科生实验报告

课程名称：**计算机网络**　　任课教师：**李文中**

学院	**计算机科学与技术系**	**专业（方向）**	**计算机科学与技术系**
学号	181860109	**姓名**	**吴润泽**
Email	181860109@smail.nju.edu.cn	**开始/完成日期**	2020/5/6-2020/5/25

1. 实验名称： Reliable Communication

2. 实验目的

1. 在Switchyard中实现有3个节点的可靠通信机制；
2. 学会在switchyard获取数据包的内容；
3. 理解并掌握滑动窗口机制的实际应用。

---

3. 实验过程

Task 2 Middlebox

a. 实现原理

1. 作为中间方，转发blaster和blastee之间的数据包；
2. 如果为blatsee发往blaster的ACK包，则按一定概率丢弃。否则直接转发即可；
3. 同时本次实验中不处理ARP包，直接忽略即可。

b. 代码编写

```
def switchy_main(net):
    my_intf = net.interfaces()
    mymacs = [intf.ethaddr for intf in my_intf]
    myips = [intf.ipaddr for intf in my_intf]
    ip_mac = ... #ip地址与mac地址的映射
    port_mac = ... #端口与mac地址的映射
    while True:
        try:
            timestamp, dev, pkt = net.recv_packet()
        except NoPackets: # 没有收到数据包
        except Shutdown:
        if gotpkt:
            if pkt[Ethernet].ethertype!=EtherType.IPv4: #非IPv4类型的包不处理
                continue
            if dev == "middlebox-eth0": #Received from blaster
                drop_rate = ... #读取middlebox_params.txt 中的丢包率
                if randint(0, 100) < drop_rate * 100:  #随机数落在该范围则丢弃
                    seq, = unpack('>i', pkt[RawPacketContents].to_bytes()[:4])
                else:  #否则将ACK发送
                    pkt[Ethernet].src = port_mac[dev]
                    pkt[Ethernet].dst = "20:00:00:00:00:01"
                    net.send_packet("middlebox-eth1", pkt)
            elif dev == "middlebox-eth1": #Received from blastee
```

```
                pkt[Ethernet].src = port_mac[dev]
                pkt[Ethernet].dst = "10:00:00:00:00:01"
                net.send_packet("middlebox-eth0", pkt)
            else: #非法的端口
                log_debug("Oops :))")
    net.shutdown()
```

Task3 Blastee

a. 实现原理

1. 判断该包目的地址的合法性，即是否发给blatsee；
2. 提取数据包中的序列号以及额外的payload，额外注意到构造ACK数据包的payload长度为固定的8字节，如果原数据包的payload不足8字节，则需要补齐；
3. 构造该序列号的ACK包，并发送给blatser即可。

b. 代码编写

```
def switchy_main(net):
    my_intf mymacs myips ip_mac port_mac #对其同样进行相同的初始化和硬
编码
    while True:
        try:
            timestamp, dev, pkt = net.recv_packet()
        except NoPackets: # 没有收到数据包
        except Shutdown:
        if gotpkt:
            if pkt[Ethernet].ethertype!=EtherType.IPv4: #非IPv4类型的包
不处理
                continue
            if str(pkt[IPv4].dst) != "192.168.200.1": #error dst isn't
blastee
                continue
            blaster_ip, num = ... #读取blaster的地址和num参数
            #构建以太网包头
            eth_header = Ethernet(src=port_mac["blastee-eth0"],
                                  dst=port_mac["middlebox-eth0"],
                                  ethertype=EtherType.IPv4)
             #构建IP包头，ttl不能为0，否则wireshark显示为红色
            ip_header = IPv4(src="192.168.200.1",
                             dst=blaster_ip,
                             protocol=IPProtocol.UDP,
                             ttl=10)
            #构建udp包头，设置源和目的端口号
            udp_header = UDP(src=7777, dst=6666)
            #获取接收包中的序列号和额外的payload信息
            seq_num =
RawPacketContents(pkt[RawPacketContents].to_bytes()[:4])
            #根据原数据包额外payload长度来构造额外的payload
            payload_len = unpack(">H",
                                 pkt[RawPacketContents].to_bytes()
[4:6])[0]
            add_payload = RawPacketContents(
                pkt[RawPacketContents].to_bytes()[6:14] +
```

```
                (bytes(8 - payload_len) if payload_len < 8 else
bytes(0)))
            ack_packet = eth_header + ip_header +
                        udp_header + seq_num + add_payload
            net.send_packet("blastee-eth0", ack_packet)
    net.shutdown()
```

Task4 Blaster

a. 实现原理

1. 根据文件中的参数，做出相应的初始化；
2. pkt_fifo 记录当前所有需要发送的数据包，$send\_list$ 记录等待ACK的数据包，LHS, RHS 记录当前窗口的具体位置，当 $RHS - LHS + 1 \leq sender\_window$ 时才可以进行发包；
3. 处理ACK机制：提取ACK包中的序列号，将对应序号从 $send\_list$ 和 pkt_fifo 中移除，

   目的是在重传队列加入pkt_fifo 之后又收到了其中的ACK号后，不需要再次发送该数据包。

   移动 LHS：重启计时器，并分三种情况移动 LHS：

   - 如果等待ACK队列 $send\_list$ 不为空，则 LHS 移动到待ACK队列的下一序列号；
   - 如果 $send\_list$ 为空，说明发出数据包均已得到ACK，LHS 移动到 $pkt\_fifo[0]$；
   - 否则 pkt_fifo 也为空，说明均已发送完毕，$LHS = num + 1$ 标志完成。
4. 处理超时机制：每次进入循环体都进行超时的判断。如果发送超时，将所有等待ACK的数据包加入等待发送队列中，重启计时器。
5. 发送数据包机制：在窗口可以发送数据包的情况下，每次均发·送 $pkt\_fifo[0]$。

   - 根据 $pkt\_fifo[0]$ 构造对应序列号的数据包并发送；
   - 如果 $pkt\_fifo[0]$ 不在 $send\_list$ 即尚未发过，则将 RHS 移至 $pkt\_fifo[0]$；
   - 如果 $pkt\_fifo[0]$ 已经在 $send\_list$ 即其为重传数据包，不需移动 RHS；
   - 将 $pkt\_fifo[0]$ 从发送队列 pkt_fifo 移除并加入待ACK队列 $send\_list$。

b. 代码编写

```
def switchy_main(net):
    my_intf mymacs myips ip_mac port_mac #对其同样进行相同的初始化和硬
编码
    begin_time = timer = time.time()  #记录整个的运行开始时间以及LHS的
计时器
    LHS = RHS = 1  #窗口的左端点和右端点
    blastee_ip, num, length, sender_window, timeout, recv_timeout #读取
相应参数
    send_list = set()  #已发送的等待接收ack的集合
    pkt_fifo = list(range(1, num + 1)) #所有需要发送的数据包
    pkt_send_count = [0] * (num + 1) #记录每个数据包发送的次数
    re_sent = once_sent = timeout_count = 0 #记录重传和超时的次数
```

```
    while True:
        try:
            timestamp, dev, pkt = net.recv_packet(timeout=
(recv_timeout) /
                                                  1000) #时间由毫秒转化
为秒
        except NoPackets: # 没有收到数据包
        except Shutdown:

        if gotpkt:
            if pkt[Ethernet].ethertype!=EtherType.IPv4: #非IPv4类型的包
不处理
                continue
            #提取ack包中的序列号
            ack_seq, = unpack('>i', pkt[RawPacketContents].to_bytes()
[:4])
            if ack_seq in send_list: #将该序列号从待ACK队列删除
                send_list.remove(ack_seq)
            if ack_seq in pkt_fifo: #从需要发送队列删除
                pkt_fifo.remove(ack_seq)
            if ack_seq == LHS: #移动LHS到合理的位置
                timer = time.time()  #restart the timer
                if len(send_list) != 0: #待ACK队列不为空
                    LHS = sorted(list(send_list))[0] #移动到待ACK队列的
下一序列号
                elif len(pkt_fifo) != 0: #ACK队列已空。准备发送
pkt_fifo[0]
                    LHS = pkt_fifo[0]
                else: #全部发送完成，LHS移动到最右端加1位置
                    LHS = num + 1
        else: #没有收到包

        if time.time() - timer >= (timeout) / 1000: #判断是否发生超时
                timeout_count += 1
                pkt_fifo.extend(send_list)  #将等待ack队列加入pkt_fifo
准备重传
                pkt_fifo = sorted(list(set(pkt_fifo)))
                timer = time.time()  #restart the timer

        if LHS == num + 1: #发送的所有包均收到ACK，打印结果信息

            #Total TX time Number of reTX
            #Number of coarse TOs
            #Throughput，Goodput
            break

        if len(pkt_fifo) == 0: continue
        #每次均发送pkt_fifo[0]
        if pkt_fifo[0] not in send_list: #判断是否为重传数据包
            if RHS - LHS + 1 <= sender_window:
                RHS = pkt_fifo[0]
            else: #window is full
```

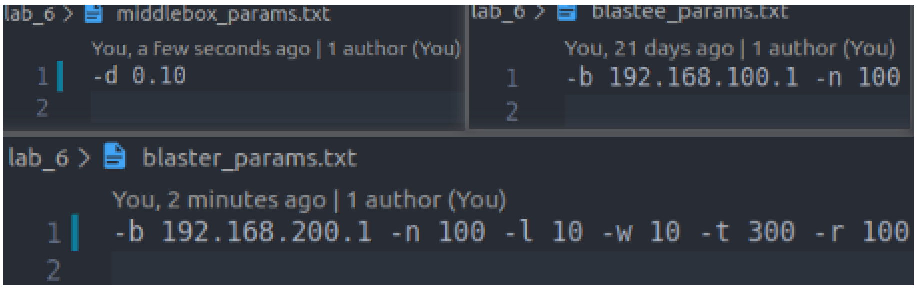
```
                continue
            #sent pkt_fifo[0]
        else: #resent pkt_fifo[0]
        pkt = create_seq_packet(pkt_fifo[0], port_mac, length) #构造对应的数据包
        send_list.add(pkt_fifo[0]) #将pkt_fifo[0]加入待接收队列
        pkt_send_count[pkt_fifo[0]] += 1 #pkt_fifo[0]的发送次数加1
        pkt_fifo.pop(0) #从等待发送队列删除pkt_fifo[0]
        net.send_packet("blaster-eth0", pkt)
    net.shutdown()
```

Task5 实现测试

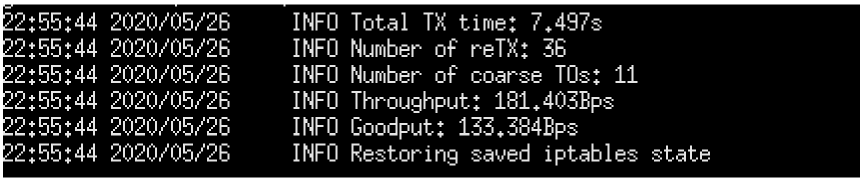
简单的性能测试

测试丢包率，窗口大小，超时时延设置对于重传率和吞吐量的影响。

初始的各项参数设置如下

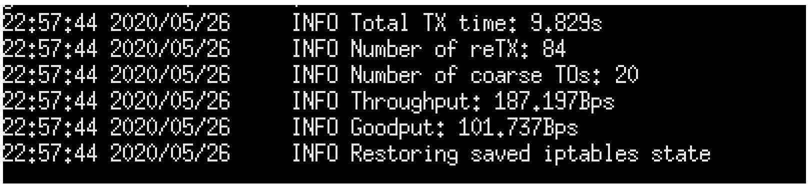


得到的测试结果如下

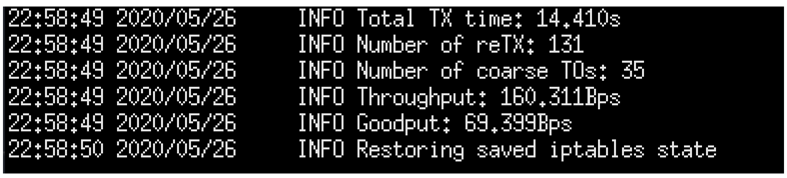


不同丢包率的性能测试

提高丢包率为0.2，进行测试得到结果



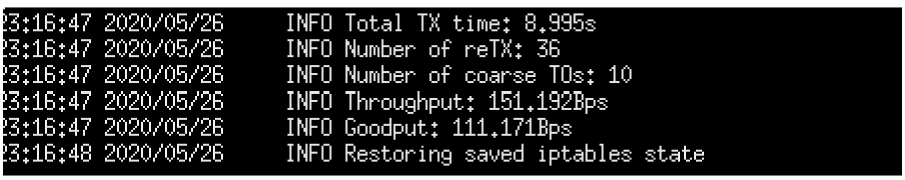
丢包率提升至0.4，进行测试得到结果



可以看到，随着丢包率的增加，超时频录增大，重传的数据包的次数增加，在总吞吐量几乎不变的情况下，*Goodput* 的吞吐量显著下降。

不同窗口的性能测试

将窗口改为5，进行测试得到结果



将窗口改为20，进行测试得到结果

```
23:18:13 2020/05/26     INFO Total TX time: 5.146s
23:18:13 2020/05/26     INFO Number of reTX: 70
23:18:13 2020/05/26     INFO Number of coarse TOs: 9
23:18:13 2020/05/26     INFO Throughput: 330.358Bps
23:18:13 2020/05/26     INFO Goodput: 194.328Bps
23:18:13 2020/05/26     INFO Restoring saved iptables state
```

将窗口改为30，进行测试得到结果

```
23:19:34 2020/05/26     INFO Total TX time: 6.138s
23:19:34 2020/05/26     INFO Number of reTX: 78
23:19:34 2020/05/26     INFO Number of coarse TOs: 12
23:19:34 2020/05/26     INFO Throughput: 290.012Bps
23:19:34 2020/05/26     INFO Goodput: 162.928Bps
23:19:34 2020/05/26     INFO Restoring saved iptables state
```

窗口大小的提高，一方面使得 $blaster$ 的总吞吐量有一定的提高，因为窗口的限制减弱，但是过高后，窗口已经对于数据包的传输数量不起决定性的限制，在三者间的传输延迟影响更大。同时，三者之间的流量增大，超时频率不变的情况下，重传的数据包增多。

H6 不同超时时延的性能测试

将超时时延改为200，进行测试得到结果

```
23:24:39 2020/05/26     INFO Total TX time: 8.150s
23:24:39 2020/05/26     INFO Number of reTX: 158
23:24:39 2020/05/26     INFO Number of coarse TOs: 31
23:24:39 2020/05/26     INFO Throughput: 316.554Bps
23:24:39 2020/05/26     INFO Goodput: 122.695Bps
23:24:39 2020/05/26     INFO Restoring saved iptables state
```

将超时时延改为500，进行测试得到结果

```
23:26:20 2020/05/26     INFO Total TX time: 8.388s
23:26:20 2020/05/26     INFO Number of reTX: 17
23:26:20 2020/05/26     INFO Number of coarse TOs: 5
23:26:20 2020/05/26     INFO Throughput: 139.485Bps
23:26:20 2020/05/26     INFO Goodput: 119.218Bps
23:26:21 2020/05/26     INFO Restoring saved iptables state
```

超时时延的提高，会明显降低超时发生的频率，但由于窗口大小的限制，重传的频录降低，窗口中等待 ACK 的数据包增多，也会一定程度上使得总吞吐量下降。

经过上述简单的测试分析，丢包率、窗口大小、超时时延三者对于吞吐量的影响是相互制约的，而非简单的线性关系。

H5 抓包测试

H6 无丢包测试

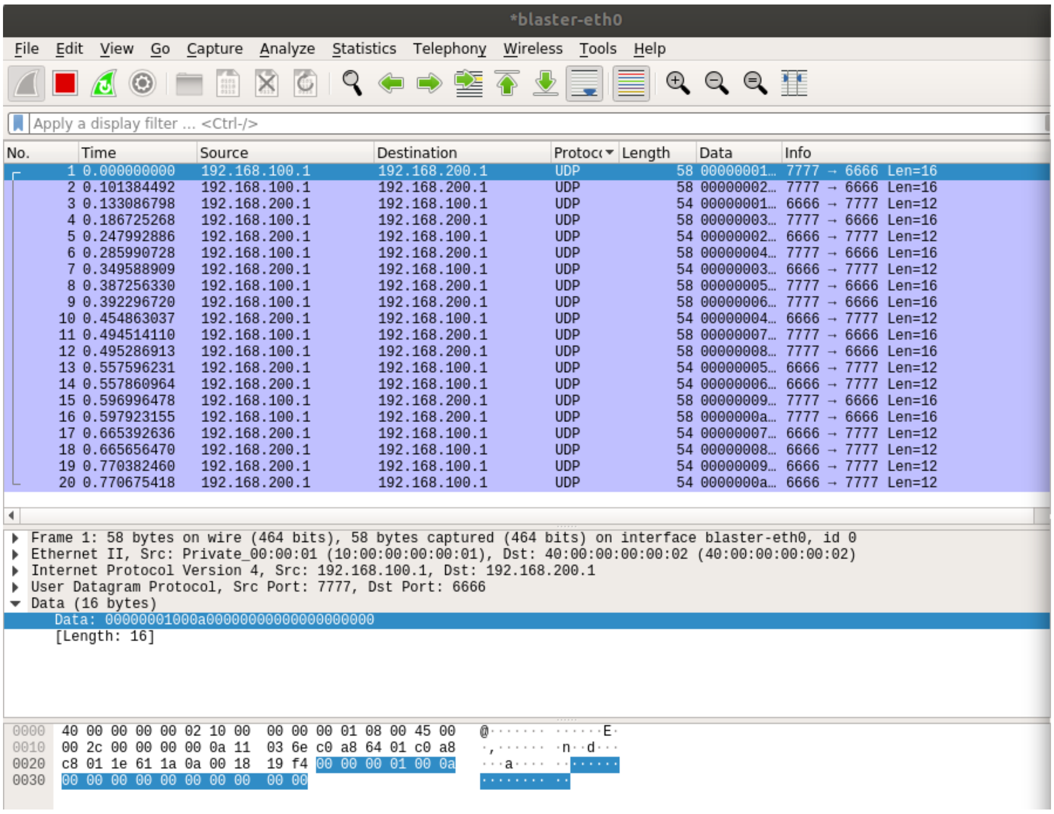
验证在丢包率为0并且超时重传延迟较大（测试中设为$500ms$），保证blaster不会发生重传的情况。设待发送的队列为10，窗口大小为5，观察 $blaster$ 和 $blastee$ 能否正确通信。

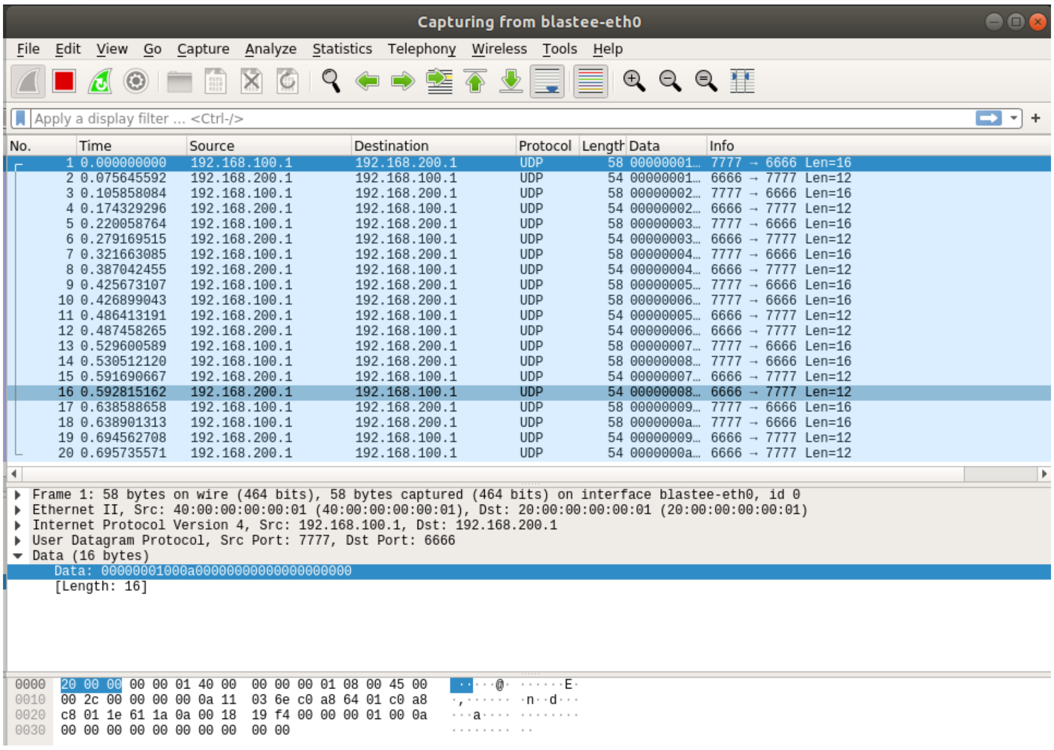
```
lab_6 > blastee_params.txt
You, 18 days ago | 1 author (You)
-b 192.168.100.1 -n 100
```

```
lab_6 > middlebox_params.txt
You, 33 minutes ago | 1 author (You)
-d 0.0
```

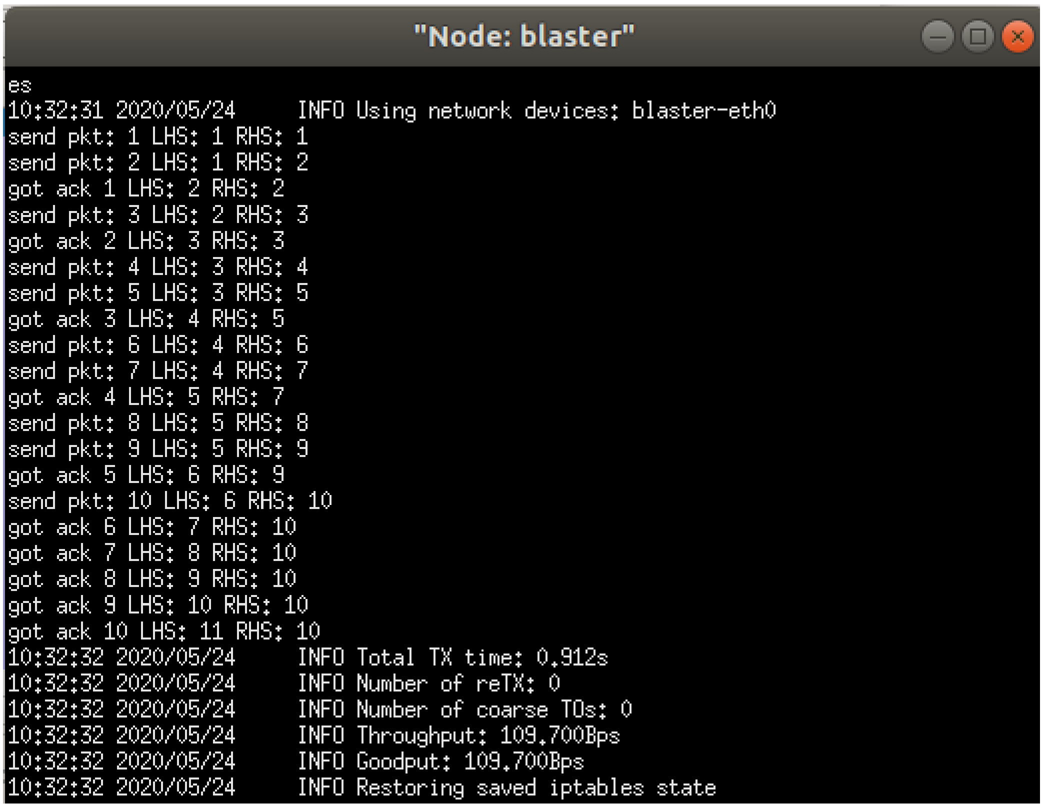
```
lab_6 > blaster_params.txt
You, 33 minutes ago | 1 author (You)
-b 192.168.200.1 -n 10 -l 10 -w 5 -t 500 -r 100
```

下图 $blaster$ 和 $blastee$ 的抓包情况，为了便于观察将data中的序列号展示。



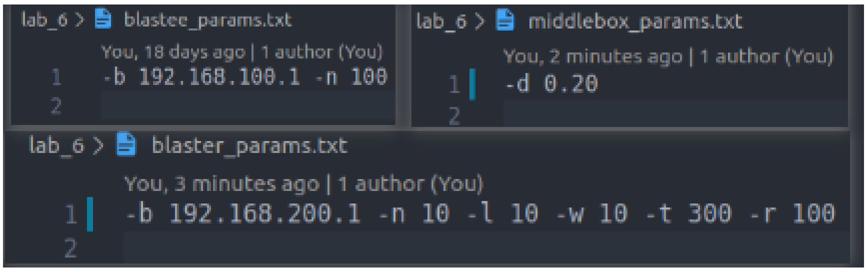


因为没有丢包，*blatser* 发送10个包，*blastee* 发送10个对应的ACK，通过在命令行中输出的调试信息，同样可以验证其正确性，没有重传的情况发生，LHS与RHS正确移动。

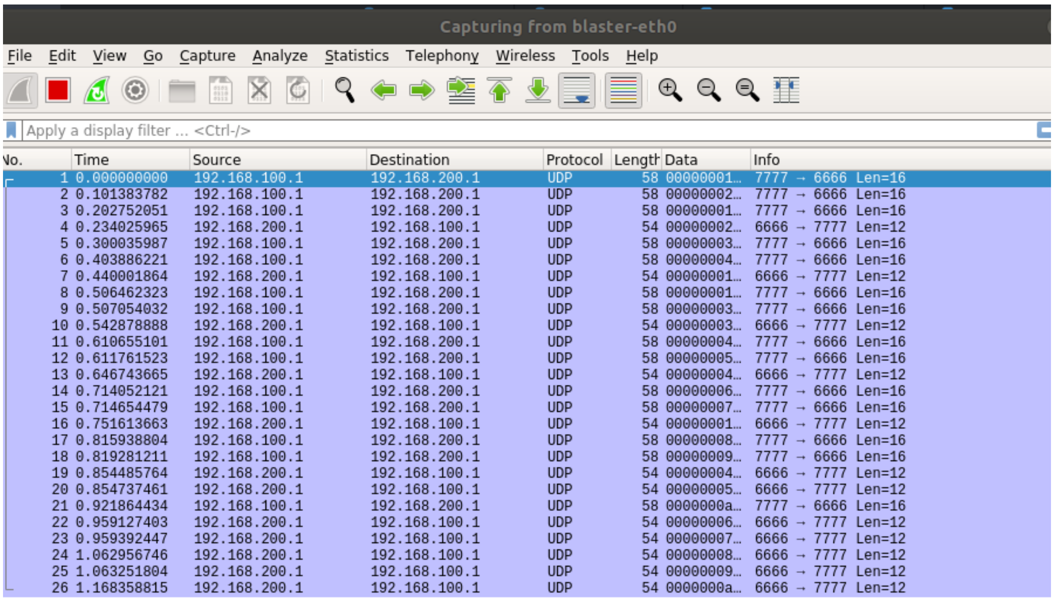


H6 有丢包测试

提高丢包率至 0.2，并将超时时间改为$300ms$，修改窗口为10，即保证不会发生窗口满的情况，仅观察超时重传机制的正确性。



下图 *blaster* 和 *blastee* 的抓包情况



No.	Time	Source	Destination	Protocol	Length	Data	Info
1	0.000000000	192.168.100.1	192.168.200.1	UDP	58	00000001…	7777 → 6666 Len=16
2	0.101383782	192.168.100.1	192.168.200.1	UDP	58	00000002…	7777 → 6666 Len=16
3	0.202752051	192.168.100.1	192.168.200.1	UDP	58	00000001…	7777 → 6666 Len=16
4	0.234025965	192.168.200.1	192.168.100.1	UDP	54	00000002…	6666 → 7777 Len=12
5	0.300035987	192.168.100.1	192.168.200.1	UDP	58	00000003…	7777 → 6666 Len=16
6	0.403886221	192.168.100.1	192.168.200.1	UDP	58	00000004…	7777 → 6666 Len=16
7	0.440001864	192.168.200.1	192.168.100.1	UDP	54	00000001…	6666 → 7777 Len=12
8	0.506462323	192.168.100.1	192.168.200.1	UDP	58	00000001…	7777 → 6666 Len=16
9	0.507054032	192.168.100.1	192.168.200.1	UDP	58	00000003…	7777 → 6666 Len=16
10	0.542878888	192.168.200.1	192.168.100.1	UDP	54	00000003…	6666 → 7777 Len=12
11	0.610655101	192.168.100.1	192.168.200.1	UDP	58	00000004…	7777 → 6666 Len=16
12	0.611761523	192.168.100.1	192.168.200.1	UDP	58	00000005…	7777 → 6666 Len=16
13	0.646743665	192.168.200.1	192.168.100.1	UDP	54	00000004…	6666 → 7777 Len=12
14	0.714052121	192.168.100.1	192.168.200.1	UDP	58	00000006…	7777 → 6666 Len=16
15	0.714654479	192.168.100.1	192.168.200.1	UDP	58	00000007…	7777 → 6666 Len=16
16	0.751613663	192.168.200.1	192.168.100.1	UDP	54	00000001…	6666 → 7777 Len=12
17	0.815938804	192.168.100.1	192.168.200.1	UDP	58	00000008…	7777 → 6666 Len=16
18	0.819281211	192.168.100.1	192.168.200.1	UDP	58	00000009…	7777 → 6666 Len=16
19	0.854485764	192.168.200.1	192.168.100.1	UDP	54	00000004…	6666 → 7777 Len=12
20	0.854737461	192.168.200.1	192.168.100.1	UDP	54	00000005…	6666 → 7777 Len=12
21	0.921864434	192.168.100.1	192.168.200.1	UDP	58	0000000a…	7777 → 6666 Len=16
22	0.959127403	192.168.200.1	192.168.100.1	UDP	54	00000006…	6666 → 7777 Len=12
23	0.959392447	192.168.200.1	192.168.100.1	UDP	54	00000007…	6666 → 7777 Len=12
24	1.062956746	192.168.200.1	192.168.100.1	UDP	54	00000008…	6666 → 7777 Len=12
25	1.063251804	192.168.200.1	192.168.100.1	UDP	54	00000009…	6666 → 7777 Len=12
26	1.168358815	192.168.200.1	192.168.100.1	UDP	54	0000000a…	6666 → 7777 Len=12

Capturing from blastee-eth0

No.	Time	Source	Destination	Protocol	Length	Data	Info
1	0.000000000	192.168.100.1	192.168.200.1	UDP	58	00000002…	7777 → 6666 Len=16
2	0.008676429	192.168.200.1	192.168.100.1	UDP	54	00000002…	6666 → 7777 Len=12
3	0.106257063	192.168.100.1	192.168.200.1	UDP	58	00000001…	7777 → 6666 Len=16
4	0.208445477	192.168.100.1	192.168.200.1	UDP	58	00000003…	7777 → 6666 Len=16
5	0.209419318	192.168.200.1	192.168.100.1	UDP	54	00000001…	6666 → 7777 Len=12
6	0.313429146	192.168.100.1	192.168.200.1	UDP	58	00000004…	7777 → 6666 Len=16
7	0.313802154	192.168.200.1	192.168.100.1	UDP	54	00000003…	6666 → 7777 Len=12
8	0.415891683	192.168.100.1	192.168.200.1	UDP	58	00000001…	7777 → 6666 Len=16
9	0.415900343	192.168.200.1	192.168.100.1	UDP	54	00000004…	6666 → 7777 Len=12
10	0.519690642	192.168.100.1	192.168.200.1	UDP	58	00000004…	7777 → 6666 Len=16
11	0.520023789	192.168.200.1	192.168.100.1	UDP	54	00000001…	6666 → 7777 Len=12
12	0.520142717	192.168.100.1	192.168.200.1	UDP	58	00000005…	7777 → 6666 Len=16
13	0.624663706	192.168.100.1	192.168.200.1	UDP	58	00000006…	7777 → 6666 Len=16
14	0.624924414	192.168.200.1	192.168.100.1	UDP	54	00000004…	6666 → 7777 Len=12
15	0.625116260	192.168.100.1	192.168.200.1	UDP	58	00000007…	7777 → 6666 Len=16
16	0.626134681	192.168.200.1	192.168.100.1	UDP	54	00000005…	6666 → 7777 Len=12
17	0.727639780	192.168.200.1	192.168.100.1	UDP	54	00000006…	6666 → 7777 Len=12
18	0.727874120	192.168.100.1	192.168.200.1	UDP	58	00000008…	7777 → 6666 Len=16
19	0.728317718	192.168.100.1	192.168.200.1	UDP	58	00000009…	7777 → 6666 Len=16
20	0.728840221	192.168.200.1	192.168.100.1	UDP	54	00000007…	6666 → 7777 Len=12
21	0.832470345	192.168.200.1	192.168.100.1	UDP	54	00000008…	6666 → 7777 Len=12
22	0.832471236	192.168.100.1	192.168.200.1	UDP	58	0000000a…	7777 → 6666 Len=16
23	0.833517106	192.168.200.1	192.168.100.1	UDP	54	00000009…	6666 → 7777 Len=12
24	0.936363291	192.168.200.1	192.168.100.1	UDP	54	0000000a…	6666 → 7777 Len=12

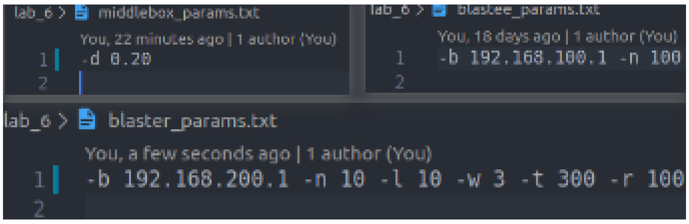
由于middlebox丢包的原因，$blaster$ 和 $blastee$ 的抓包数量不再相同，同样可以验证 $blatser$ 调试信息，与上述抓包情况的逻辑一致性。当发送了$1,2$ 包后，只得到了2的ACK，LHS仍为1，产生了超时，重传1，因为2已经获得ack，故不需再次发送。之后发送了$3,4$，因为仍未收到1的ACK，再次产生超时，重传1，3，4，重传机制实现正确。

```
(syenv) root@njucs-VirtualBox:~/switchyard/lab_6# swyard blaster.py
12:53:05 2020/05/24     INFO Saving iptables state and installing switchyard rul
es
12:53:05 2020/05/24     INFO Using network devices: blaster-eth0
send pkt: 1 LHS: 1 RHS: 1
send pkt: 2 LHS: 1 RHS: 2
timeout meet 1590295985.548255 1590295985.2422779
resend pkt: 1
got ack 2 LHS: 1 RHS: 2
send pkt: 3 LHS: 1 RHS: 3
send pkt: 4 LHS: 1 RHS: 4
timeout meet 1590295985.851418 1590295985.5483003
resend pkt: 1
got ack 1 LHS: 3 RHS: 4
resend pkt: 3
resend pkt: 4
got ack 3 LHS: 4 RHS: 4
send pkt: 5 LHS: 4 RHS: 5
send pkt: 6 LHS: 4 RHS: 6
got ack 4 LHS: 5 RHS: 6
send pkt: 7 LHS: 5 RHS: 7
send pkt: 8 LHS: 5 RHS: 8
got ack 1 LHS: 5 RHS: 8
send pkt: 9 LHS: 5 RHS: 9
send pkt: 10 LHS: 5 RHS: 10
got ack 4 LHS: 5 RHS: 10
got ack 5 LHS: 6 RHS: 10
got ack 6 LHS: 7 RHS: 10
got ack 7 LHS: 8 RHS: 10
got ack 8 LHS: 9 RHS: 10
got ack 9 LHS: 10 RHS: 10
got ack 10 LHS: 11 RHS: 10
12:53:06 2020/05/24     INFO Total TX time: 1.340s
12:53:06 2020/05/24     INFO Number of reTX: 4
12:53:06 2020/05/24     INFO Number of coarse TOs: 2
12:53:06 2020/05/24     INFO Throughput: 104.448Bps
12:53:06 2020/05/24     INFO Goodput: 74.605Bps
12:53:06 2020/05/24     INFO Restoring saved iptables state
```

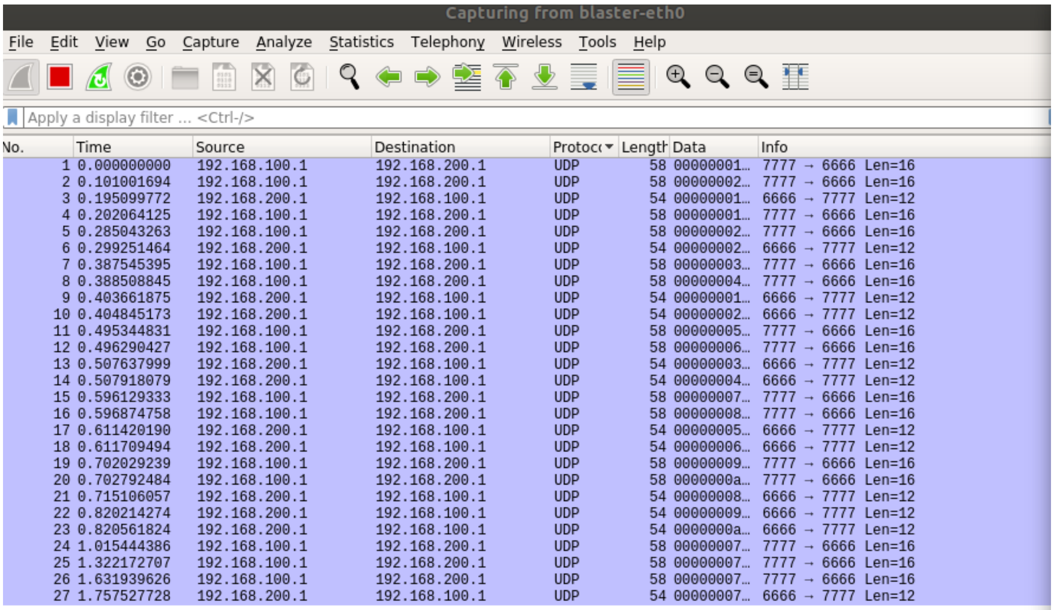
丢包和缓存测试

测试当丢包和窗口大小有限时的处理机制，观察是否符合逻辑。

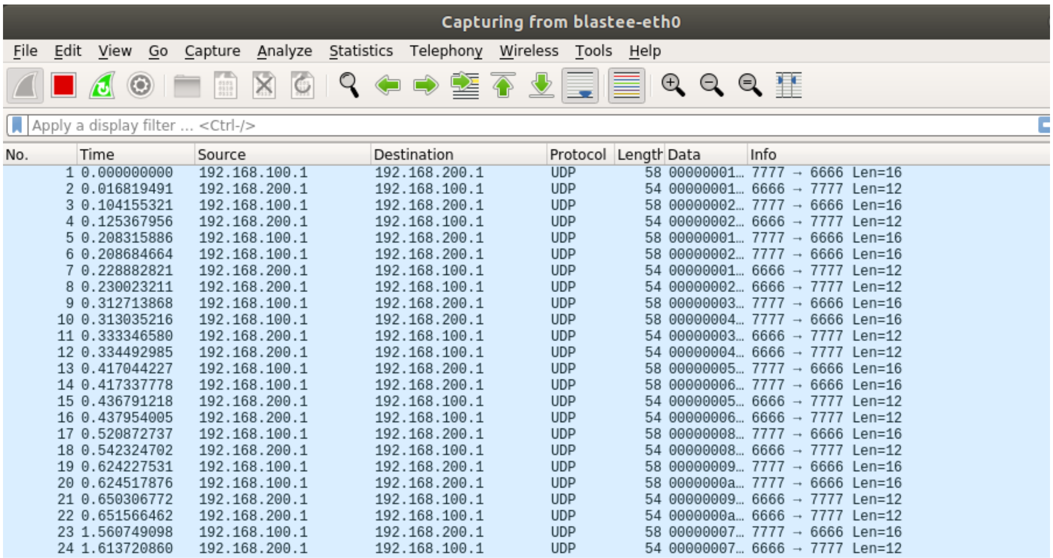
窗口参数改为3，其余与上一个测试参数相同。



下图 $blaster$ 和 $blastee$ 的抓包情况

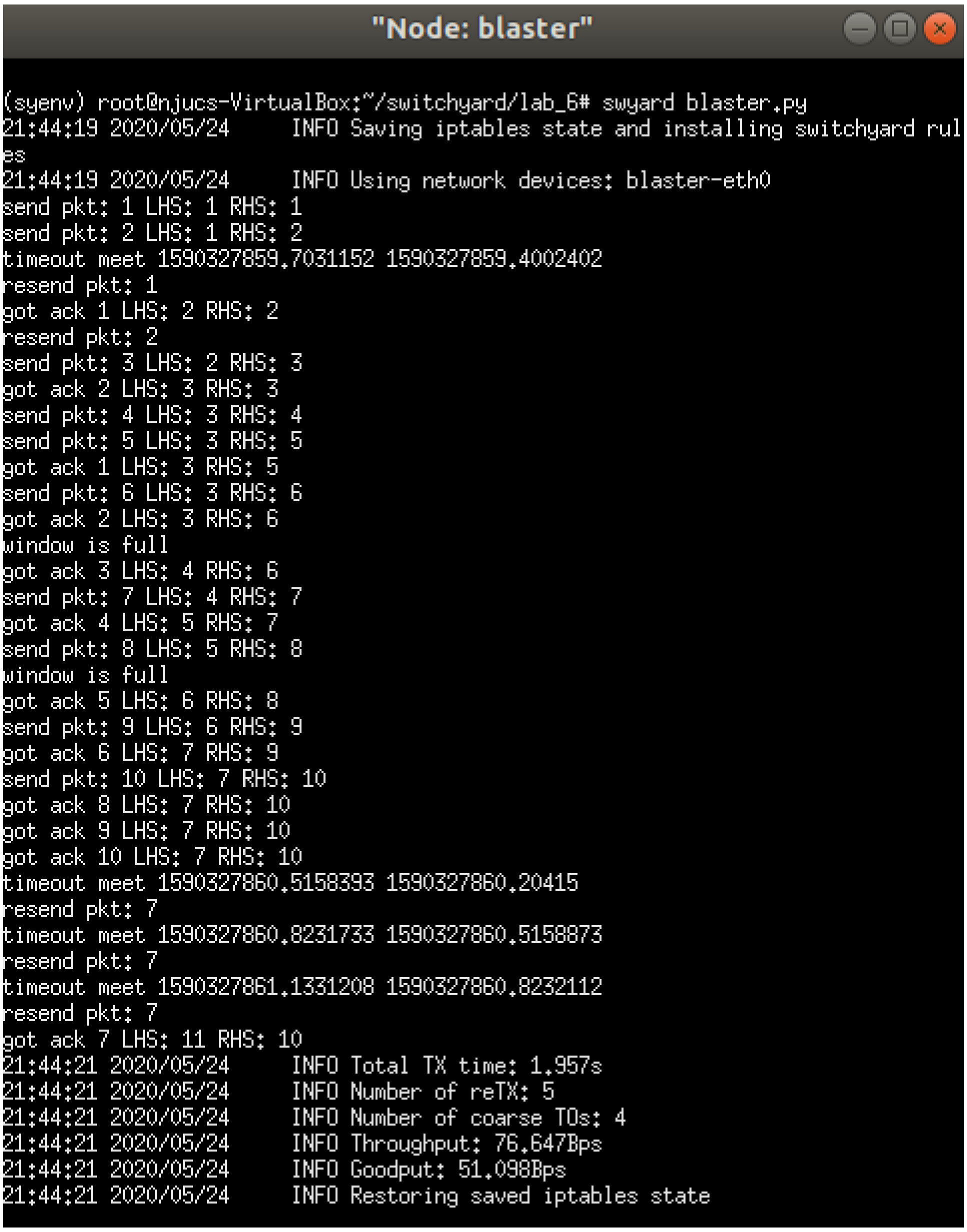


No.	Time	Source	Destination	Protocol	Length	Data	Info
1	0.000000000	192.168.100.1	192.168.200.1	UDP	58	00000001…	7777 → 6666 Len=16
2	0.101001694	192.168.100.1	192.168.200.1	UDP	58	00000002…	7777 → 6666 Len=16
3	0.195099772	192.168.200.1	192.168.100.1	UDP	54	00000001…	6666 → 7777 Len=12
4	0.202064125	192.168.100.1	192.168.200.1	UDP	58	00000001…	7777 → 6666 Len=16
5	0.285043263	192.168.100.1	192.168.200.1	UDP	58	00000002…	7777 → 6666 Len=16
6	0.299251464	192.168.200.1	192.168.100.1	UDP	54	00000002…	6666 → 7777 Len=12
7	0.387545395	192.168.100.1	192.168.200.1	UDP	58	00000003…	7777 → 6666 Len=16
8	0.388508845	192.168.100.1	192.168.200.1	UDP	58	00000004…	7777 → 6666 Len=16
9	0.403661875	192.168.200.1	192.168.100.1	UDP	54	00000001…	6666 → 7777 Len=12
10	0.404845173	192.168.200.1	192.168.100.1	UDP	54	00000002…	6666 → 7777 Len=12
11	0.495344831	192.168.100.1	192.168.200.1	UDP	58	00000005…	7777 → 6666 Len=16
12	0.496290427	192.168.100.1	192.168.200.1	UDP	58	00000006…	7777 → 6666 Len=16
13	0.507637999	192.168.200.1	192.168.100.1	UDP	54	00000003…	6666 → 7777 Len=12
14	0.507918079	192.168.200.1	192.168.100.1	UDP	54	00000004…	6666 → 7777 Len=12
15	0.596129333	192.168.100.1	192.168.200.1	UDP	58	00000007…	7777 → 6666 Len=16
16	0.596874758	192.168.100.1	192.168.200.1	UDP	58	00000008…	7777 → 6666 Len=16
17	0.611420190	192.168.200.1	192.168.100.1	UDP	54	00000005…	6666 → 7777 Len=12
18	0.611709494	192.168.200.1	192.168.100.1	UDP	54	00000006…	6666 → 7777 Len=12
19	0.702029239	192.168.100.1	192.168.200.1	UDP	58	00000009…	7777 → 6666 Len=16
20	0.702792484	192.168.100.1	192.168.200.1	UDP	58	0000000a…	7777 → 6666 Len=16
21	0.715106057	192.168.200.1	192.168.100.1	UDP	54	00000008…	6666 → 7777 Len=12
22	0.820214274	192.168.200.1	192.168.100.1	UDP	54	00000009…	6666 → 7777 Len=12
23	0.820561824	192.168.200.1	192.168.100.1	UDP	54	0000000a…	6666 → 7777 Len=12
24	1.015444386	192.168.100.1	192.168.200.1	UDP	58	00000007…	7777 → 6666 Len=16
25	1.322172707	192.168.100.1	192.168.200.1	UDP	58	00000007…	7777 → 6666 Len=16
26	1.631939626	192.168.100.1	192.168.200.1	UDP	58	00000007…	7777 → 6666 Len=16
27	1.757527728	192.168.200.1	192.168.100.1	UDP	54	00000007…	6666 → 7777 Len=12



No.	Time	Source	Destination	Protocol	Length	Data	Info
1	0.000000000	192.168.100.1	192.168.200.1	UDP	58	00000001…	7777 → 6666 Len=16
2	0.016819491	192.168.200.1	192.168.100.1	UDP	54	00000001…	6666 → 7777 Len=12
3	0.104155321	192.168.100.1	192.168.200.1	UDP	58	00000002…	7777 → 6666 Len=16
4	0.125367956	192.168.200.1	192.168.100.1	UDP	54	00000002…	6666 → 7777 Len=12
5	0.208315886	192.168.100.1	192.168.200.1	UDP	58	00000001…	7777 → 6666 Len=16
6	0.208684664	192.168.100.1	192.168.200.1	UDP	58	00000002…	7777 → 6666 Len=16
7	0.228882821	192.168.200.1	192.168.100.1	UDP	54	00000001…	6666 → 7777 Len=12
8	0.230023211	192.168.200.1	192.168.100.1	UDP	54	00000002…	6666 → 7777 Len=12
9	0.312713868	192.168.100.1	192.168.200.1	UDP	58	00000003…	7777 → 6666 Len=16
10	0.313035216	192.168.100.1	192.168.200.1	UDP	58	00000004…	7777 → 6666 Len=16
11	0.333346580	192.168.200.1	192.168.100.1	UDP	54	00000003…	6666 → 7777 Len=12
12	0.334492985	192.168.200.1	192.168.100.1	UDP	54	00000004…	6666 → 7777 Len=12
13	0.417044227	192.168.100.1	192.168.200.1	UDP	58	00000005…	7777 → 6666 Len=16
14	0.417337778	192.168.100.1	192.168.200.1	UDP	58	00000006…	7777 → 6666 Len=16
15	0.436791218	192.168.200.1	192.168.100.1	UDP	54	00000005…	6666 → 7777 Len=12
16	0.437954005	192.168.200.1	192.168.100.1	UDP	54	00000006…	6666 → 7777 Len=12
17	0.520872737	192.168.100.1	192.168.200.1	UDP	58	00000008…	7777 → 6666 Len=16
18	0.542324702	192.168.200.1	192.168.100.1	UDP	54	00000008…	6666 → 7777 Len=12
19	0.624227531	192.168.100.1	192.168.200.1	UDP	58	00000009…	7777 → 6666 Len=16
20	0.624517876	192.168.100.1	192.168.200.1	UDP	58	0000000a…	7777 → 6666 Len=16
21	0.650306772	192.168.200.1	192.168.100.1	UDP	54	00000009…	6666 → 7777 Len=12
22	0.651566462	192.168.200.1	192.168.100.1	UDP	54	0000000a…	6666 → 7777 Len=12
23	1.560749098	192.168.100.1	192.168.200.1	UDP	58	00000007…	7777 → 6666 Len=16
24	1.613720860	192.168.200.1	192.168.100.1	UDP	54	00000007…	6666 → 7777 Len=12

可以观察到，RHS与LHS限制的大小始终没有超过窗口大小，如在blaster抓到的第10号包，即对于2的ACK，但是窗口已满，$blaster$ 不再发送新的数据包。同时超时机制也运行正常。除此之外，经过若干次抓包测试发现，由于窗口大小的限制，等待ack的数据包数量减少，超时发生的频率也有些许下降。



"Node: blaster"

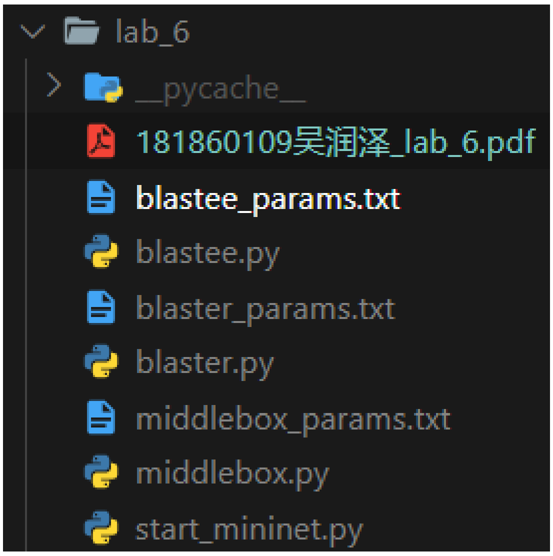
```
(syenv) root@njucs-VirtualBox:~/switchyard/lab_6# swyard blaster.py
21:44:19 2020/05/24     INFO Saving iptables state and installing switchyard rul
es
21:44:19 2020/05/24     INFO Using network devices: blaster-eth0
send pkt: 1 LHS: 1 RHS: 1
send pkt: 2 LHS: 1 RHS: 2
timeout meet 1590327859.7031152 1590327859.4002402
resend pkt: 1
got ack 1 LHS: 2 RHS: 2
resend pkt: 2
send pkt: 3 LHS: 2 RHS: 3
got ack 2 LHS: 3 RHS: 3
send pkt: 4 LHS: 3 RHS: 4
send pkt: 5 LHS: 3 RHS: 5
got ack 1 LHS: 3 RHS: 5
send pkt: 6 LHS: 3 RHS: 6
got ack 2 LHS: 3 RHS: 6
window is full
got ack 3 LHS: 4 RHS: 6
send pkt: 7 LHS: 4 RHS: 7
got ack 4 LHS: 5 RHS: 7
send pkt: 8 LHS: 5 RHS: 8
window is full
got ack 5 LHS: 6 RHS: 8
send pkt: 9 LHS: 6 RHS: 9
got ack 6 LHS: 7 RHS: 9
send pkt: 10 LHS: 7 RHS: 10
got ack 8 LHS: 7 RHS: 10
got ack 9 LHS: 7 RHS: 10
got ack 10 LHS: 7 RHS: 10
timeout meet 1590327860.5158393 1590327860.20415
resend pkt: 7
timeout meet 1590327860.8231733 1590327860.5158873
resend pkt: 7
timeout meet 1590327861.1331208 1590327860.8232112
resend pkt: 7
got ack 7 LHS: 11 RHS: 10
21:44:21 2020/05/24     INFO Total TX time: 1.957s
21:44:21 2020/05/24     INFO Number of reTX: 5
21:44:21 2020/05/24     INFO Number of coarse TOs: 4
21:44:21 2020/05/24     INFO Throughput: 76.647Bps
21:44:21 2020/05/24     INFO Goodput: 51.098Bps
21:44:21 2020/05/24     INFO Restoring saved iptables state
```

4. 总结与感想

这次实验总体难度不大，主要是一次对于滑动窗口机制的具体实现。在实现过程中，提取数据包的序列号成为很大的阻力，通过查阅相关资料，通过 $to\_bytes/from\_bytes, pack/unpack$ 四个python处理字节函数较为轻松的实现相关要求。除此之外，通过纯抓包来验证实现逻辑的正确性很有难度，可以通过输出调试信息，来帮助自己理清发包之间的具体关系。

除此之外，本次实验还暴露出我的阅读理解水平的严重不足。例如，最开始忽略了对于 $payload$ 不足8字节进行补充的要求；超时检测机制理解不清，最初实现为，没有收到包才进行超时检测。诸如此类的问题，在临近ddl前爆发（庆幸发现），重构代码，重新抓包，修改报告，给自己带来不小的困扰。本次实验让我深刻体会到认真读手册的重要性。

5. 文档结构


lab_6
__pycache__
181860109吴润泽_lab_6.pdf
blastee_params.txt
blastee.py
blaster_params.txt
blaster.py
middlebox_params.txt
middlebox.py
start_mininet.py