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	**191220154**	姓名	**张涵之**
Email	**1683762615@qq.com**	开始/完成日期	**2021/5/30-2021/6/1**

1. 实验名称：Lab 6: Reliable Communication

2. 实验目的：

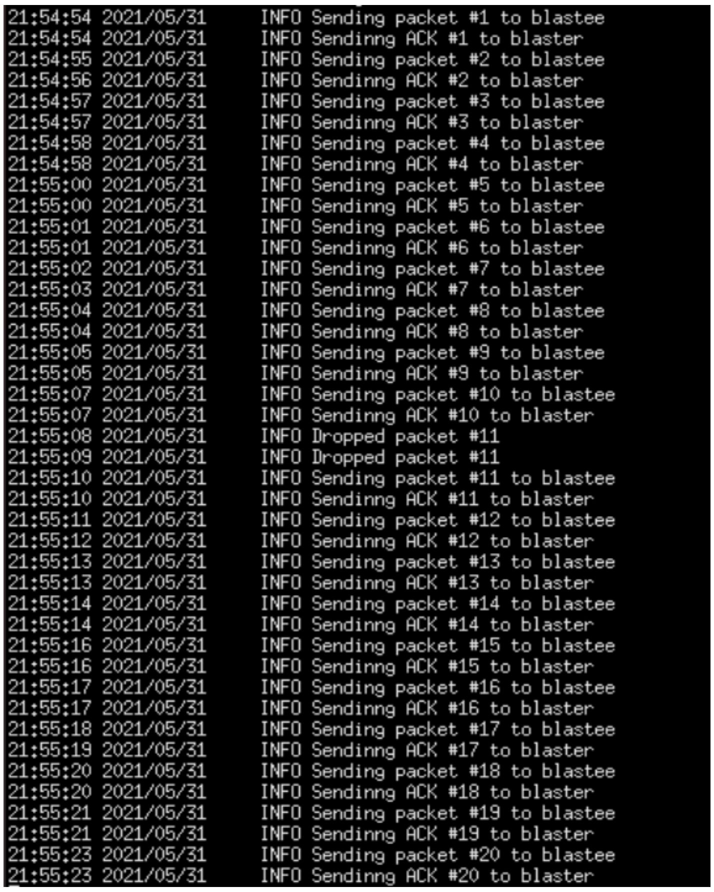
Build a reliable communication library in Switchyard.

3. 实验内容

a) Task 2: Implement the features of middlebox
b) Task 3: Implement the features of blastee
c) Task 4: Implement the features of blaster

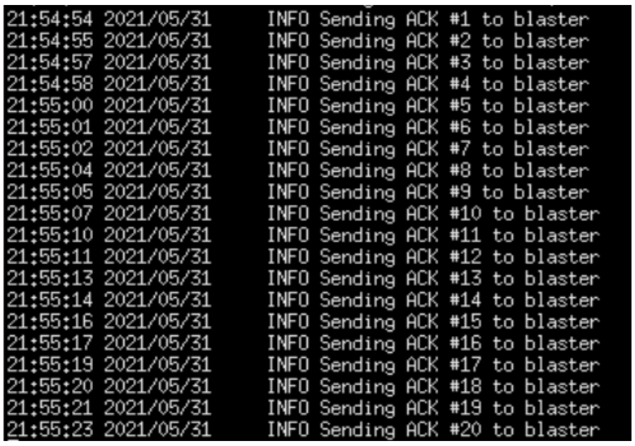
4. 实验结果

a) Running middlebox with the drop rate of 0.19



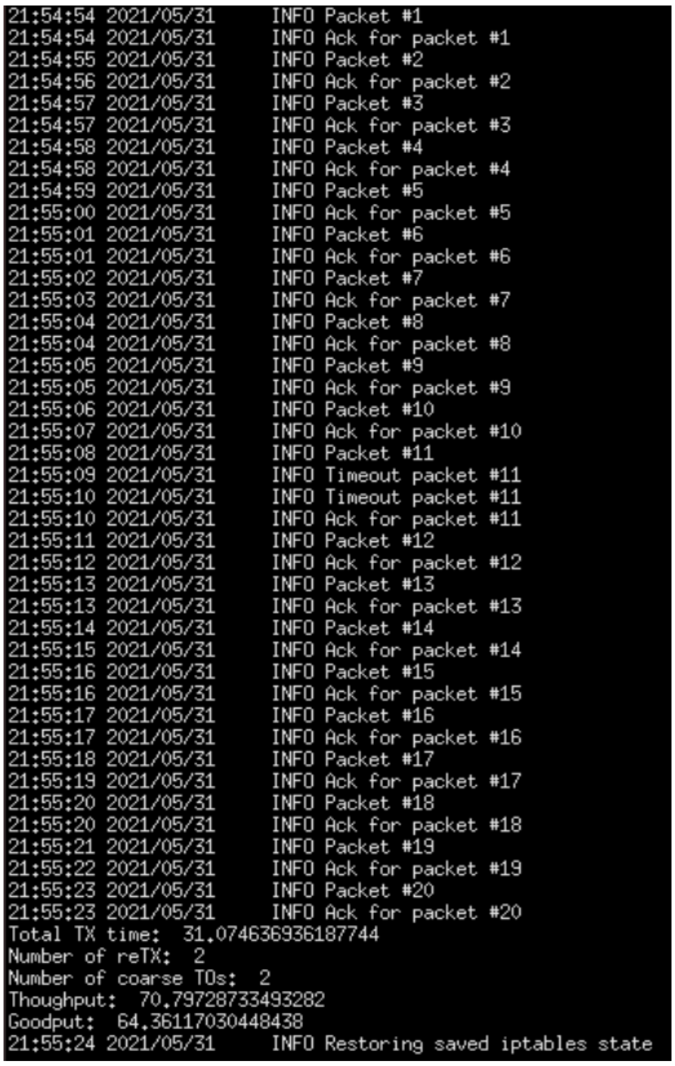
```
21:54:54 2021/05/31     INFO Sending packet #1 to blastee
21:54:54 2021/05/31     INFO Sendinng ACK #1 to blaster
21:54:55 2021/05/31     INFO Sending packet #2 to blastee
21:54:56 2021/05/31     INFO Sendinng ACK #2 to blaster
21:54:57 2021/05/31     INFO Sending packet #3 to blastee
21:54:57 2021/05/31     INFO Sendinng ACK #3 to blaster
21:54:58 2021/05/31     INFO Sending packet #4 to blastee
21:54:58 2021/05/31     INFO Sendinng ACK #4 to blaster
21:55:00 2021/05/31     INFO Sending packet #5 to blastee
21:55:00 2021/05/31     INFO Sendinng ACK #5 to blaster
21:55:01 2021/05/31     INFO Sending packet #6 to blastee
21:55:01 2021/05/31     INFO Sendinng ACK #6 to blaster
21:55:02 2021/05/31     INFO Sending packet #7 to blastee
21:55:03 2021/05/31     INFO Sendinng ACK #7 to blaster
21:55:04 2021/05/31     INFO Sending packet #8 to blastee
21:55:04 2021/05/31     INFO Sendinng ACK #8 to blaster
21:55:05 2021/05/31     INFO Sending packet #9 to blastee
21:55:05 2021/05/31     INFO Sendinng ACK #9 to blaster
21:55:07 2021/05/31     INFO Sending packet #10 to blastee
21:55:07 2021/05/31     INFO Sendinng ACK #10 to blaster
21:55:08 2021/05/31     INFO Dropped packet #11
21:55:09 2021/05/31     INFO Dropped packet #11
21:55:10 2021/05/31     INFO Sending packet #11 to blastee
21:55:10 2021/05/31     INFO Sendinng ACK #11 to blaster
21:55:11 2021/05/31     INFO Sending packet #12 to blastee
21:55:12 2021/05/31     INFO Sendinng ACK #12 to blaster
21:55:13 2021/05/31     INFO Sending packet #13 to blastee
21:55:13 2021/05/31     INFO Sendinng ACK #13 to blaster
21:55:14 2021/05/31     INFO Sending packet #14 to blastee
21:55:14 2021/05/31     INFO Sendinng ACK #14 to blaster
21:55:16 2021/05/31     INFO Sending packet #15 to blastee
21:55:16 2021/05/31     INFO Sendinng ACK #15 to blaster
21:55:17 2021/05/31     INFO Sending packet #16 to blastee
21:55:17 2021/05/31     INFO Sendinng ACK #16 to blaster
21:55:18 2021/05/31     INFO Sending packet #17 to blastee
21:55:19 2021/05/31     INFO Sendinng ACK #17 to blaster
21:55:20 2021/05/31     INFO Sending packet #18 to blastee
21:55:20 2021/05/31     INFO Sendinng ACK #18 to blaster
21:55:21 2021/05/31     INFO Sending packet #19 to blastee
21:55:21 2021/05/31     INFO Sendinng ACK #19 to blaster
21:55:23 2021/05/31     INFO Sending packet #20 to blastee
21:55:23 2021/05/31     INFO Sendinng ACK #20 to blaster
```

b) Running blastee with blaster IP 192.168.100.1 and num 20



```
21:54:54 2021/05/31     INFO Sending ACK #1 to blaster
21:54:55 2021/05/31     INFO Sending ACK #2 to blaster
21:54:57 2021/05/31     INFO Sending ACK #3 to blaster
21:54:58 2021/05/31     INFO Sending ACK #4 to blaster
21:55:00 2021/05/31     INFO Sending ACK #5 to blaster
21:55:01 2021/05/31     INFO Sending ACK #6 to blaster
21:55:02 2021/05/31     INFO Sending ACK #7 to blaster
21:55:04 2021/05/31     INFO Sending ACK #8 to blaster
21:55:05 2021/05/31     INFO Sending ACK #9 to blaster
21:55:07 2021/05/31     INFO Sending ACK #10 to blaster
21:55:10 2021/05/31     INFO Sending ACK #11 to blaster
21:55:11 2021/05/31     INFO Sending ACK #12 to blaster
21:55:13 2021/05/31     INFO Sending ACK #13 to blaster
21:55:14 2021/05/31     INFO Sending ACK #14 to blaster
21:55:16 2021/05/31     INFO Sending ACK #15 to blaster
21:55:17 2021/05/31     INFO Sending ACK #16 to blaster
21:55:19 2021/05/31     INFO Sending ACK #17 to blaster
21:55:20 2021/05/31     INFO Sending ACK #18 to blaster
21:55:21 2021/05/31     INFO Sending ACK #19 to blaster
21:55:23 2021/05/31     INFO Sending ACK #20 to blaster
```

c) Running blaster with blastee IP=192.168.200.1, num=20, length=100, sender window=5, timeout=1000 and receive timeout=100



```
21:54:54 2021/05/31     INFO Packet #1
21:54:54 2021/05/31     INFO Ack for packet #1
21:54:55 2021/05/31     INFO Packet #2
21:54:56 2021/05/31     INFO Ack for packet #2
21:54:57 2021/05/31     INFO Packet #3
21:54:57 2021/05/31     INFO Ack for packet #3
21:54:58 2021/05/31     INFO Packet #4
21:54:58 2021/05/31     INFO Ack for packet #4
21:54:59 2021/05/31     INFO Packet #5
21:55:00 2021/05/31     INFO Ack for packet #5
21:55:01 2021/05/31     INFO Packet #6
21:55:01 2021/05/31     INFO Ack for packet #6
21:55:02 2021/05/31     INFO Packet #7
21:55:03 2021/05/31     INFO Ack for packet #7
21:55:04 2021/05/31     INFO Packet #8
21:55:04 2021/05/31     INFO Ack for packet #8
21:55:05 2021/05/31     INFO Packet #9
21:55:05 2021/05/31     INFO Ack for packet #9
21:55:06 2021/05/31     INFO Packet #10
21:55:07 2021/05/31     INFO Ack for packet #10
21:55:08 2021/05/31     INFO Packet #11
21:55:09 2021/05/31     INFO Timeout packet #11
21:55:10 2021/05/31     INFO Timeout packet #11
21:55:10 2021/05/31     INFO Ack for packet #11
21:55:11 2021/05/31     INFO Packet #12
21:55:12 2021/05/31     INFO Ack for packet #12
21:55:13 2021/05/31     INFO Packet #13
21:55:13 2021/05/31     INFO Ack for packet #13
21:55:14 2021/05/31     INFO Packet #14
21:55:15 2021/05/31     INFO Ack for packet #14
21:55:16 2021/05/31     INFO Packet #15
21:55:16 2021/05/31     INFO Ack for packet #15
21:55:17 2021/05/31     INFO Packet #16
21:55:17 2021/05/31     INFO Ack for packet #16
21:55:18 2021/05/31     INFO Packet #17
21:55:19 2021/05/31     INFO Ack for packet #17
21:55:20 2021/05/31     INFO Packet #18
21:55:20 2021/05/31     INFO Ack for packet #18
21:55:21 2021/05/31     INFO Packet #19
21:55:22 2021/05/31     INFO Ack for packet #19
21:55:23 2021/05/31     INFO Packet #20
21:55:23 2021/05/31     INFO Ack for packet #20
Total TX time:  31.074636936187744
Number of reTX:  2
Number of coarse TOs:  2
Thoughput:  70.79728733493282
Goodput:  64.36117030448438
21:55:24 2021/05/31     INFO Restoring saved iptables state
```

From the log info we can infer that middlebox dropped packet #11 twice, and blaster raised two timeouts for the packet, demanding retransfer of the ack.

5. 核心代码

a) Overview

The middlebox, as its name suggests, serves as a transfer station between the blaster and the blastee, it has some of the basic functions of a hub/switch/router such as receiving and forwarding packets, but is a more simplified version. It only has two interfaces, that is to say, no ARP and no forward table, when the middlebox receives a packet from one interface, it sends it out the other.

The blastee receives packets and sends back ack, simple as that.

The blaster is the most complicated. Suppose it imitates the transfer process of a file, which it breaks down into num pieces. The pieces are put in a queue and wait to be sent in their original order. The sender window can hold at most () such packets, as they except the corresponding ack. Once a packet has received its ack and there are no other packets waiting to be ack'd before it, it can come out of the queue. The blaster continuously test timeout, which occurs when the time past after the last ack packet exceed the set timeout argument. Once the timeout is detected, the first unack'd packet in the window is resent. Strategies are implemented so that one packet is sent in one round. When all num packets have been ack'd successfully, blaster print some statistics and then shutdown.

b) Task 2: Middlebox

Modification in the function handle_packet:

```
sequence = int.from_bytes(packet[3].to_bytes()[:4], 'big')
if fromIface == self.intf1:
    log_debug("Received from blaster")
```

```
    #self.totalCount += 1
    rand = random()
    if rand <= self.dropRate:
        log_info("Dropped packet #{}".format(sequence))
        #self.dropCount += 1
    else:
        log_info("Sending packet #{} to blastee".format(sequence))
        packet[Ethernet].src = '40:00:00:00:00:02'
        packet[Ethernet].dst = '20:00:00:00:00:01'
        self.net.send_packet("middlebox-eth1", packet)
```

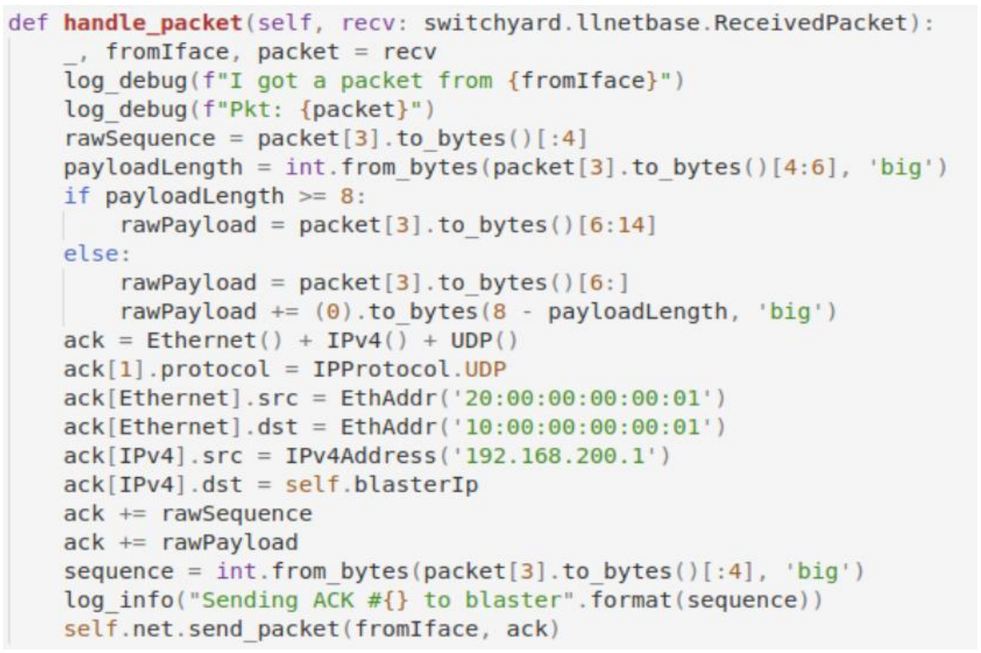
This part deals with packets from blaster to blastee. A random number between 0 and 1 is generated, if it is below drop rate the packet is dropped, otherwise it is sent. This is to simulate the possible packet loss in a real network.

```
elif fromIface == "middlebox-eth1":
    log_debug("Received from blastee")
```

```
    log_info("Sendinng ACK #{} to blaster".format(sequence))
    packet[Ethernet].src = '40:00:00:00:00:01'
    packet[Ethernet].dst = '10:00:00:00:00:01'
    self.net.send_packet("middlebox-eth0", packet)
```

This part deals with ack from blastee to blaster. No ack is dropped here.

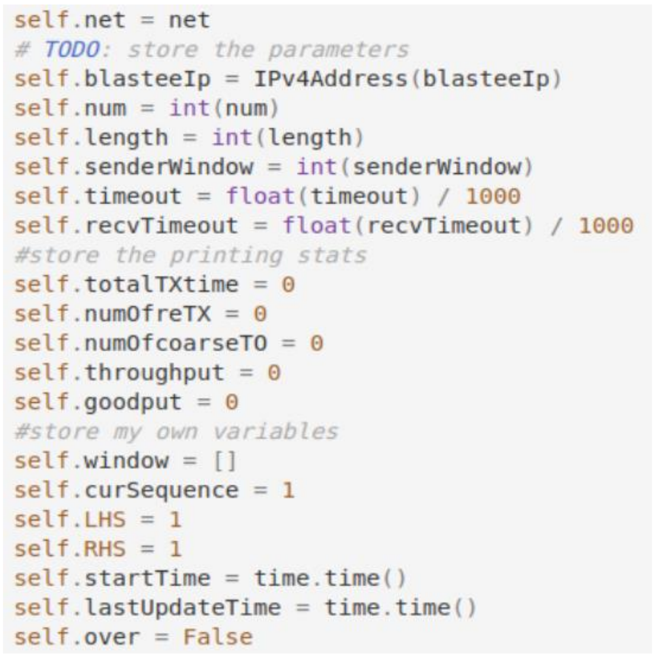
c) Task 3: Blastee
Modification in the function handle_packet:



```
def handle_packet(self, recv: switchyard.llnetbase.ReceivedPacket):
    _, fromIface, packet = recv
    log_debug(f"I got a packet from {fromIface}")
    log_debug(f"Pkt: {packet}")
    rawSequence = packet[3].to_bytes()[:4]
    payloadLength = int.from_bytes(packet[3].to_bytes()[4:6], 'big')
    if payloadLength >= 8:
        rawPayload = packet[3].to_bytes()[6:14]
    else:
        rawPayload = packet[3].to_bytes()[6:]
        rawPayload += (0).to_bytes(8 - payloadLength, 'big')
    ack = Ethernet() + IPv4() + UDP()
    ack[1].protocol = IPProtocol.UDP
    ack[Ethernet].src = EthAddr('20:00:00:00:00:01')
    ack[Ethernet].dst = EthAddr('10:00:00:00:00:01')
    ack[IPv4].src = IPv4Address('192.168.200.1')
    ack[IPv4].dst = self.blasterIp
    ack += rawSequence
    ack += rawPayload
    sequence = int.from_bytes(packet[3].to_bytes()[:4], 'big')
    log_info("Sending ACK #{} to blaster".format(sequence))
    self.net.send_packet(fromIface, ack)
```

The three packet headers are generate using functions, while other data directly attached to the packet. The RawPacketContents header looks complicated and this works, so anyway. The sequence number is encoded (and decoded) using big-endian format, and the payload is set to a fixed size. If the received packet's payload is longer than the length, the first 8 bytes are taken. Otherwise, if it is shorter than the length, some 0s are added to the payload to fill the length.

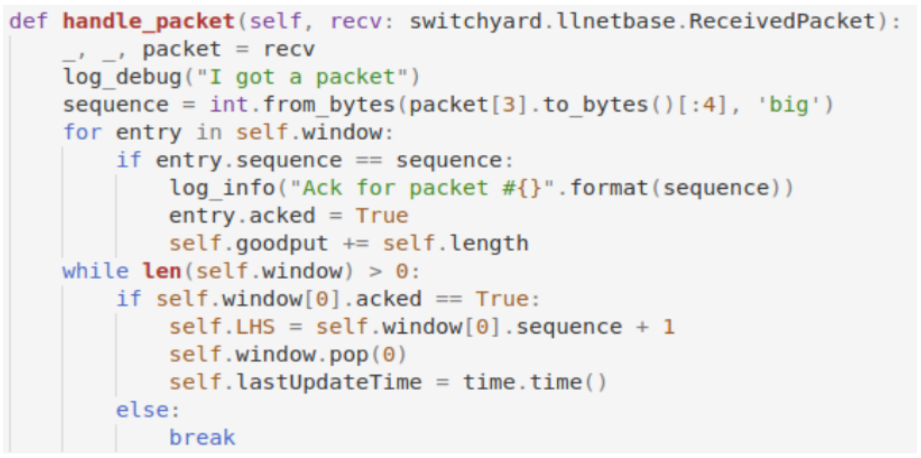
d) Task 4: Blaster
First, we look at the parameters and variables added to the init function:



```
self.net = net
# TODO: store the parameters
self.blasteeIp = IPv4Address(blasteeIp)
self.num = int(num)
self.length = int(length)
self.senderWindow = int(senderWindow)
self.timeout = float(timeout) / 1000
self.recvTimeout = float(recvTimeout) / 1000
#store the printing stats
self.totalTXtime = 0
self.numOfreTX = 0
self.numOfcoarseTO = 0
self.throughput = 0
self.goodput = 0
#store my own variables
self.window = []
self.curSequence = 1
self.LHS = 1
self.RHS = 1
self.startTime = time.time()
self.lastUpdateTime = time.time()
self.over = False
```

As we can see the sequence start from 1, the number of the first packet.

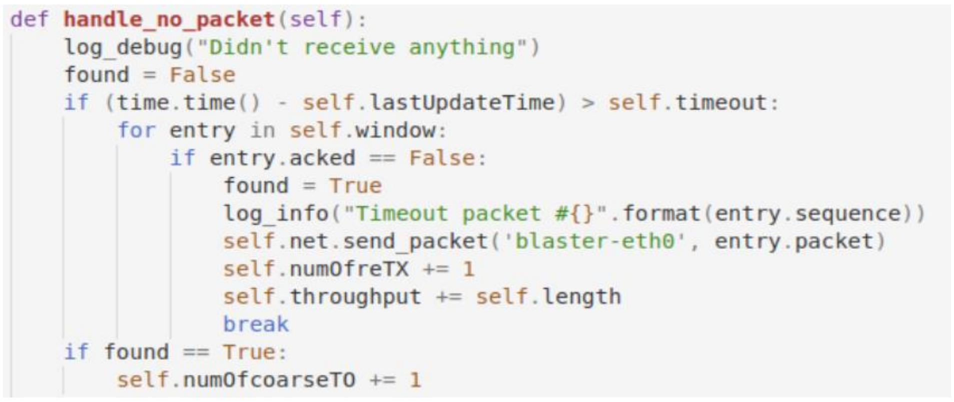
Then we modify the handle_packet function:



```
def handle_packet(self, recv: switchyard.llnetbase.ReceivedPacket):
    _, _, packet = recv
    log_debug("I got a packet")
    sequence = int.from_bytes(packet[3].to_bytes()[:4], 'big')
    for entry in self.window:
        if entry.sequence == sequence:
            log_info("Ack for packet #{}".format(sequence))
            entry.acked = True
            self.goodput += self.length
    while len(self.window) > 0:
        if self.window[0].acked == True:
            self.LHS = self.window[0].sequence + 1
            self.window.pop(0)
            self.lastUpdateTime = time.time()
        else:
            break
```

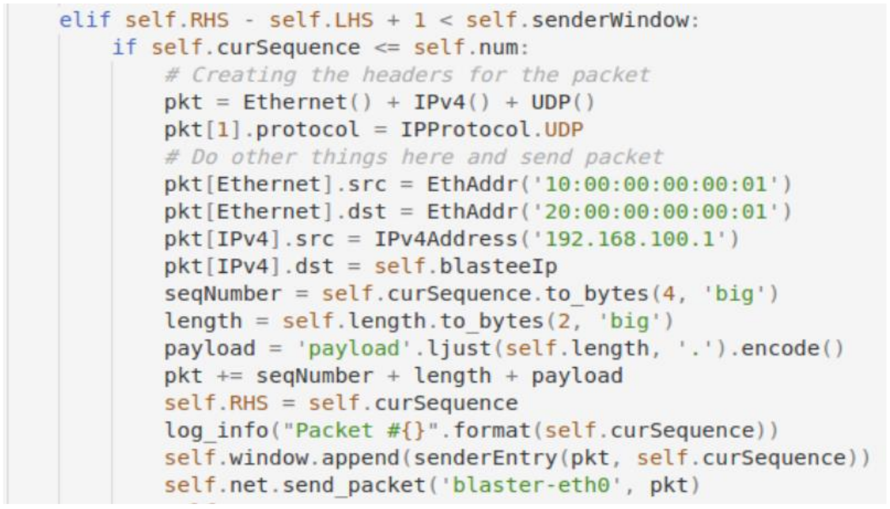
Each time an ack is received, the blaster checks its queue for the corresponding packet. The packet is marked “ack’d”, and length is added to the total goodput. Blaster then checks each packet in the front of the queue, if it has been ack’d it is removed, otherwise the loop stops, for every packet after an unack’d one should be blocked there until each entry before it has been removed.

Then we modify the handle_no_packet function:

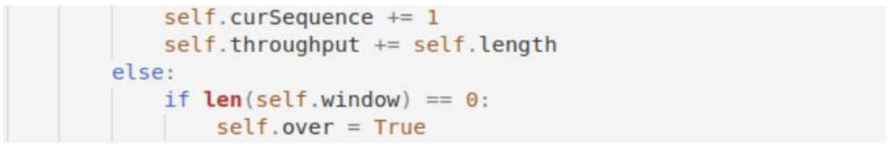


```
def handle_no_packet(self):
    log_debug("Didn't receive anything")
    found = False
    if (time.time() - self.lastUpdateTime) > self.timeout:
        for entry in self.window:
            if entry.acked == False:
                found = True
                log_info("Timeout packet #{}".format(entry.sequence))
                self.net.send_packet('blaster-eth0', entry.packet)
                self.numOfreTX += 1
                self.throughput += self.length
                break
    if found == True:
        self.numOfcoarseTO += 1
```

First, we check if there is a timeout, if so, check the sender window for the first unack’d packet, if found, resend packet, update number of reTX and the total throughput. After the search, also update the number of coarse timeout.



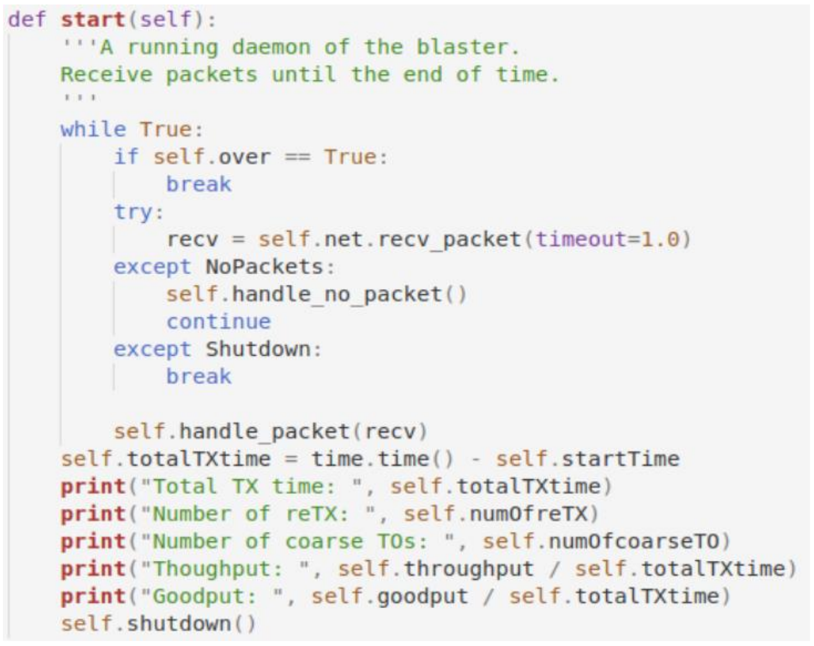
```
    elif self.RHS - self.LHS + 1 < self.senderWindow:
        if self.curSequence <= self.num:
            # Creating the headers for the packet
            pkt = Ethernet() + IPv4() + UDP()
            pkt[1].protocol = IPProtocol.UDP
            # Do other things here and send packet
            pkt[Ethernet].src = EthAddr('10:00:00:00:00:01')
            pkt[Ethernet].dst = EthAddr('20:00:00:00:00:01')
            pkt[IPv4].src = IPv4Address('192.168.100.1')
            pkt[IPv4].dst = self.blasteeIp
            seqNumber = self.curSequence.to_bytes(4, 'big')
            length = self.length.to_bytes(2, 'big')
            payload = 'payload'.ljust(self.length, '.').encode()
            pkt += seqNumber + length + payload
            self.RHS = self.curSequence
            log_info("Packet #{}".format(self.curSequence))
            self.window.append(senderEntry(pkt, self.curSequence))
            self.net.send_packet('blaster-eth0', pkt)
```



```
            self.curSequence += 1
            self.throughput += self.length
        else:
            if len(self.window) == 0:
                self.over = True
```

If there is a timeout unack'd packet and it is resent, we shall not send any other packets in this round. Otherwise, we check the sender window and the current sequence number. If the window is not full yet (the number of packets trapped in a traffic jam blocked by one or more unack'd packet is less than the window size), and the current sequence number is no more than the num of packets to be transferred, a packet is constructed, sent and added to the queue. The RHS is then updated to the next number above the current sequence, the sequence is incremented by 1, and length is added to the total throughput. If the sequence is more than num and the send window is empty (in other words, all the packets sent have been successfully ack'd, the blaster should shutdown.

There are also some modifications in the start function:



```
def start(self):
    '''A running daemon of the blaster.
    Receive packets until the end of time.
    '''
    while True:
        if self.over == True:
            break
        try:
            recv = self.net.recv_packet(timeout=1.0)
        except NoPackets:
            self.handle_no_packet()
            continue
        except Shutdown:
            break

        self.handle_packet(recv)
    self.totalTXtime = time.time() - self.startTime
    print("Total TX time: ", self.totalTXtime)
    print("Number of reTX: ", self.numOfreTX)
    print("Number of coarse TOs: ", self.numOfcoarseTO)
    print("Thoughput: ", self.throughput / self.totalTXtime)
    print("Goodput: ", self.goodput / self.totalTXtime)
    self.shutdown()
```

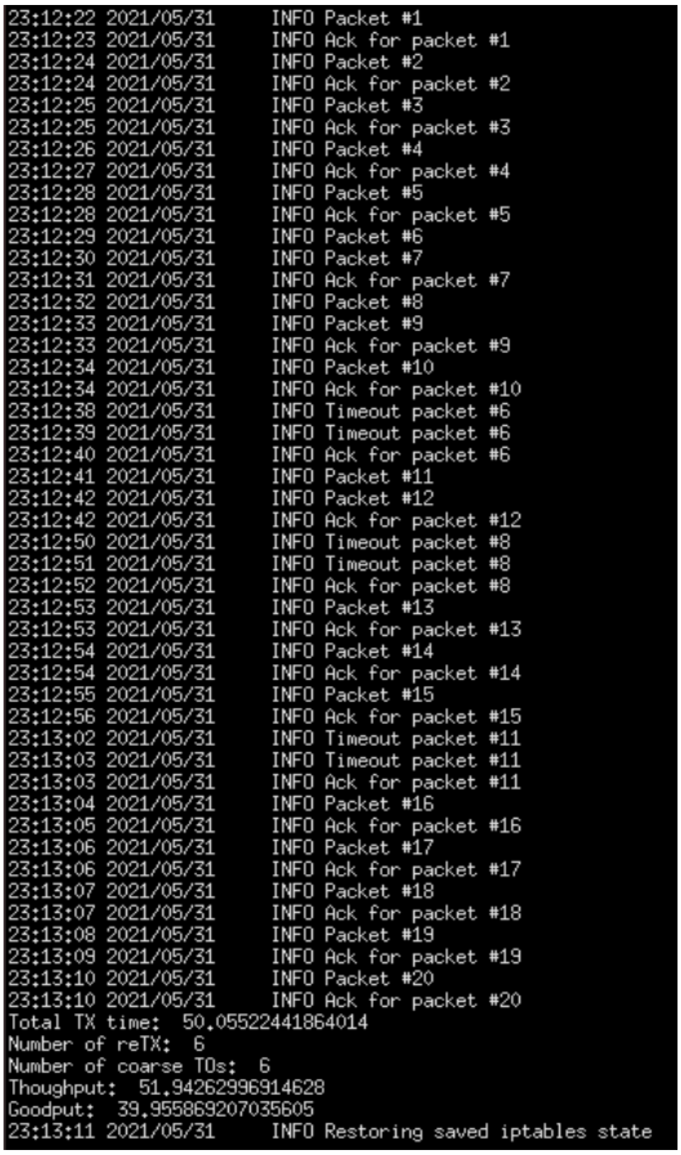
In each loop the blaster check if over==True, that is to say, all num packets are transferred successfully. It then breaks away from the loop. It then calculates and prints all the information required in the manual and calls shutdown.

6. 总结与感想

a) In this lab I still used self-defined classes and queues, like in the previous labs. I also tried to make the logic as simple and efficient as possible. When I went to the TA to test lab 4, he asked me why I put every packet in the wait queue after I receive it, instead of sending the ones than can be matched in forwarding table and save the rest that need ARP. So, this time I send every packet at once upon receiving it, and then put them in a queue for resend.

b) The entire lab appears more confusing than the ones before it. Perhaps because there are three separate modules that shall cooperate. My suggestion is to read through the manual and find out what the three modules respectively does, how they interact and what packets they each send and receive. Once you have an overview of the whole structure, the rest is quite simple.

c) 100 packets are hard to follow, so I tried 20. In the screenshot above it seems that each timeout is directly followed by its corresponding ack. I suppose the time is too short so other acks have not arrived yet, so here I tried an extremely large timeout, and it worked. At most five packets in the window at a time.



```
23:12:22 2021/05/31     INFO Packet #1
23:12:23 2021/05/31     INFO Ack for packet #1
23:12:24 2021/05/31     INFO Packet #2
23:12:24 2021/05/31     INFO Ack for packet #2
23:12:25 2021/05/31     INFO Packet #3
23:12:25 2021/05/31     INFO Ack for packet #3
23:12:26 2021/05/31     INFO Packet #4
23:12:27 2021/05/31     INFO Ack for packet #4
23:12:28 2021/05/31     INFO Packet #5
23:12:28 2021/05/31     INFO Ack for packet #5
23:12:29 2021/05/31     INFO Packet #6
23:12:30 2021/05/31     INFO Packet #7
23:12:31 2021/05/31     INFO Ack for packet #7
23:12:32 2021/05/31     INFO Packet #8
23:12:33 2021/05/31     INFO Packet #9
23:12:33 2021/05/31     INFO Ack for packet #9
23:12:34 2021/05/31     INFO Packet #10
23:12:34 2021/05/31     INFO Ack for packet #10
23:12:38 2021/05/31     INFO Timeout packet #6
23:12:39 2021/05/31     INFO Timeout packet #6
23:12:40 2021/05/31     INFO Ack for packet #6
23:12:41 2021/05/31     INFO Packet #11
23:12:42 2021/05/31     INFO Packet #12
23:12:42 2021/05/31     INFO Ack for packet #12
23:12:50 2021/05/31     INFO Timeout packet #8
23:12:51 2021/05/31     INFO Timeout packet #8
23:12:52 2021/05/31     INFO Ack for packet #8
23:12:53 2021/05/31     INFO Packet #13
23:12:53 2021/05/31     INFO Ack for packet #13
23:12:54 2021/05/31     INFO Packet #14
23:12:54 2021/05/31     INFO Ack for packet #14
23:12:55 2021/05/31     INFO Packet #15
23:12:56 2021/05/31     INFO Ack for packet #15
23:13:02 2021/05/31     INFO Timeout packet #11
23:13:03 2021/05/31     INFO Timeout packet #11
23:13:03 2021/05/31     INFO Ack for packet #11
23:13:04 2021/05/31     INFO Packet #16
23:13:05 2021/05/31     INFO Ack for packet #16
23:13:06 2021/05/31     INFO Packet #17
23:13:06 2021/05/31     INFO Ack for packet #17
23:13:07 2021/05/31     INFO Packet #18
23:13:07 2021/05/31     INFO Ack for packet #18
23:13:08 2021/05/31     INFO Packet #19
23:13:09 2021/05/31     INFO Ack for packet #19
23:13:10 2021/05/31     INFO Packet #20
23:13:10 2021/05/31     INFO Ack for packet #20
Total TX time:  50.05522441864014
Number of reTX:  6
Number of coarse TOs:  6
Thoughput:  51.94262996914628
Goodput:  39.955869207035605
23:13:11 2021/05/31     INFO Restoring saved iptables state
```

One thing that confuses me is the number of reTX, with the same parameters it can vary from 2 to 7, perhaps the random function is not that random.