南京大学计算机网络实验报告

任课教师:田臣

实验六 Reliable Communication

计算机科学与技术系

181860055 **刘国涛**

邮箱: 181860055@smail.nju.edu.cn

2020年5月13日

实验目的

- 构建可靠的通信库
- 实现通信库中的窗口机制

实验内容

TASK 2 Middlebox

任务概述 构建一个具有两个端口 `Middlebox`，实现简单的转发功能并模拟丢包机制

任务实现

1、初始化

任务要求通过读取文件 `middlebox_params.txt` 获取丢包率 `drop_rate` 的初始值

```
def init_drop_rate(filepath):
    f = open(filepath,"r")
    line = f.readline()
    mode, param = line.strip().split(' ')
    print("drop rate:{}".format(param))
    f.close()
    return float(param)
```

这里通过 `init_drop_rate` 函数来实现

2、实现转发功能

再 `Middlebox` 中要求实现的转发功能比较简单，只需要实现：

1. 将从一个接口收到的包从另一个接口发出
2. 修改接收到的包的Ethernet header
3. 根据 `drop_rate` 随机丢包

首先，转发机制的实现很简单，只需要如下结构即可：

```python
if dev == "middlebox-eth0":
    # send packet on "middlebox-eth1"
    net.send_packet('middlebox-eth1',pkt)
elif dev == "middlebox-eth1":
    # send packet on "middlebox-eth0"
    net.send_packet('middlebox-eth0',pkt)
```

然后对于包头的修改，则通过 `modify_packet_header` 方法实现：

```python
def modify_packet_header(packet,sourceMAC,nextMAC):
    eth = packet.get_header(Ethernet)
    eth.dst = nextMAC
    eth.src = sourceMAC
```

例如，将从blaster发来的包修改源MAC为 `eth1` 的MAC（连接blastee的端口），修改目的MAC为blastee的MAC，此处都是通过硬编码实现

最后，实现随机丢包的方法如下：

```python
def is_lucky_packet(drop_rate):
    ran = random.random()
    if ran > drop_rate:
        return True
    else:
        return False
```

在主循环中的结构如下：

```python
if dev == "middlebox-eth0":
    if is_lucky_packet(drop_rate):
        modify_packet_header(pkt,mymacs[1],blastee_mac)
        net.send_packet("middlebox-eth1",pkt)
    else:   # drop
        pass
elif dev == "mimddlebox-eth1":
    # ...
```

至此，middlebox就完成了

TASK 3 Blastee

任务概述 接收Blaster发来的数据包，并返回确认ACK

任务实现

1、初始化

Blastee也需要通过文件 `blastee_params.txt` 获取Blaster的IP地址和将要发送的数据包的数量

```
def init(filepath):
    f = open(filepath,'r')
    line = f.readline()
    mode_b,blaster_ip,mode_n,num = line.strip().split(' ')
    f.close()
    return blaster_ip,int(num)
```

2、实现ACK

ACK的结构：

```
<-Switchyard headers-> <-Your packet header(raw bytes)-> <-Payload in raw bytes->

|ETH Hd|IP Hdr|UDP Hdr|    Sequence number(32 bits)     |Payload  (8 bytes)  |
```

首先封装一个 `mk_ack` 方法，该方法用于根据收到的数据包的 `sequence number` 和 `payload` 来构造ACK

```
def mk_ACK(pkt,dst_mac,src_mac,dst_ip,src_ip):
    eth = Ethernet()
    eth.dst = dst_mac
    eth.src = src_mac

    ip = IPv4(protocol=IPProtocol.UDP)
    ip.dst = dst_ip
    ip.src = src_ip

    udp = UDP()  # not use port

    bs = pkt[3].to_bytes()
    sequence_num = bs[:32]
```

```python

        con = RawPacketContents(sequence_num)
        payload = RawPacketContents(bs[48:56])

        return eth + ip + udp + con + payload
```

然后在主循环中实现ACK的发送即可：

```python
if gotpkt:
    ack = mk_ACK(pkt,middlebox_mac,mymacs[0],blaster_ip,myips[0])
    net.send_packet('blastee-eth0',ack)
```

至此Blastee完成

Blaster

任务概述 在Blaster中实现窗口机制

任务实现

设计窗口类：

`Sender_Window` 类的设计思想为：

1. 通过窗口机制控制发包速率
2. 应与包的创建分离，仅管理包的发送与超时重发
3. 对传输数据进行统计，在发送结束后输出

首先在 `__init__` 中对窗口机制实现所需变量进行初始化：

```python
class Sender_Window():
    def __init__(self,size,timeout,num,length):
        self.rhs = -1
        self.lhs = 0
        self.size = size
        self.length = length
        self.window = []
        self.timeout = timeout
        self.num = num
```

然后在 `start` 和 `end` 中对统计变量分别进行初始化和计算输出

```python
    def start(self):
        self.startTime = time.time()
        self.reNum = 0
        self.toNum = 0
        self.packet_count = 0
        self.update_time = self.startTime
    def end(self):
        self.endTime = time.time()
        total_time = self.endTime - self.startTime
        total_through_bytes = self.length*self.packet_count
        total_good_bytes = self.length*(self.packet_count-self.reNum)
        throughput = total_through_bytes / total_time
        goodput = total_good_bytes / total_time
        print("transmission statistics:\ntotal TX time:{}\nNumber of reTX:{}\nNumber of coarse TOs:{}\nThroughput:{}\ngoodput:{}\n" \
.format(total_time,self.reNum,self.toNum,throughput,goodput))
```

接下来实现窗口机制

SW中的窗口通过list实现，lhs,rhs分别作为当前窗口的左右指针。

`window` 中每一个item的结构是

```
{
    'packet':packet # 数据包
    'state':state # 数据包状态: 0->待发送 1->已发送 2->ACKd
}
```

首先考虑SW在主循环中的**未收到包**时运行的框架

```
if sw.need_load(): # 判断是否需要装载数据包
    index = sw.load_packet( \
mk_pkt(seq,middlebox_mac,mymacs[0],blastee_ip,myips[0],length) \
    )
    seq += 1       #  sequence number 自增
    sw.send_packet(net,index) # 发送
else:  # 不需要装载数据包 则进行超时检测
    index = sw.check_timeout()
    sw.send_packet(net,index)
```

接下来对SW的各个方法进行实现：

```
    def load_packet(self,packet):
        self.rhs += 1
        self.window.append({
            'packet':packet,
            'state':0 # to send
        })
        return self.rhs

    def send_packet(self,net,index):
        if index >= self.lhs and index <= self.rhs:
            net.send_packet('blaster-eth0',self.window[index]['packet'])
            self.window[index]['state'] = 1
            self.packet_count += 1

    def need_load(self):
        if self.rhs >= self.num - 1: # no packet need to send
            return False
        if self.rhs - self.lhs + 1 >= self.size:
            return False

        return True

    def check_timeout(self):
        now = time.time()
```

```
        if now - self.update_time > self.timeout: # 超时判断
            self.toNum += 1
        else:
            return -1

        for i in range(self.rhs-self.lhs+1):
            item = self.window[self.lhs + i]
            if item['state'] == 1:
                self.reNum += 1
                print('renum:',self.reNum)
                # timeout , resend it
                return self.lhs + i

        return -1
```

接着考虑SW在主循环中**收到数据包**的运行结构：

```
if gotpkt:
    sw.dealACK(pkt)
```

在收到ACK后，sw解析ACK里的sequence number，然后将 `window` 中对应item的状态设为2

```
    def dealACK(self,ack):
        seq = ack[3].to_bytes()[:31]
        sequence_num = int(seq)
        self.window[sequence_num]['state'] = 2 # ack
        log_info("ack {}".format(sequence_num))
```

最后，实现对窗口的更新（更新 `lhs` 和 `update_time` ）

```python
    def update_window(self):
        if self.lhs == self.rhs and self.rhs == self.num - 1:
            return 0  # done !

        if self.rhs>=self.lhs and self.window[self.lhs]['state'] == 2:
            self.lhs += 1
            while self.lhs<self.rhs and self.window[self.lhs]['state'] == 2 :
                self.lhs += 1
            self.update_time = time.time()

        return 1
```

所有都完成后，主循环的结构如下

```python
sw = Sender_Window(...)
sw.start()
while True:
    # recv packet and set flag 'gotpkt'

    if gotpkt:
        sw.dealACK()
    else:
        if sw.need_load():
            index = sw.load_packet(mk_pkt(seq))
            seq += 1
            sw.send_packet(net,index)
        else:
            index = sw.check_timeout()
            sw.send_packet(net,index)
    ret = sw.update_window()
    if ret == 0: # 发送结束
        sw.end()
        break
```

超时机制的实现：

在Blaster中，超时的定义为：

> LHS一段时间不改变

在SW中，只在 `update_window` 方法中对LHS进行更新，所以

在LHS更新同时，更新 `update_time`

```python
self.update_time = time.time()
```

然后再超时检测方法 `check_timeout` 中，检测当前时间与 `update_time` 的差值是否大于 `timeout` 即可

```python
now = time.time()
if now - self.update_time > self.timeout: # 超时判断
    pass # do something
```

实验结果

Deploying:

为了证明blastee,blaster,middlebox能够正常运行，使用下面三组参数进行测试：

	丢包率drop_rate	窗口大小W
1	0.2	8
2	0.2	5
3	0.1	5

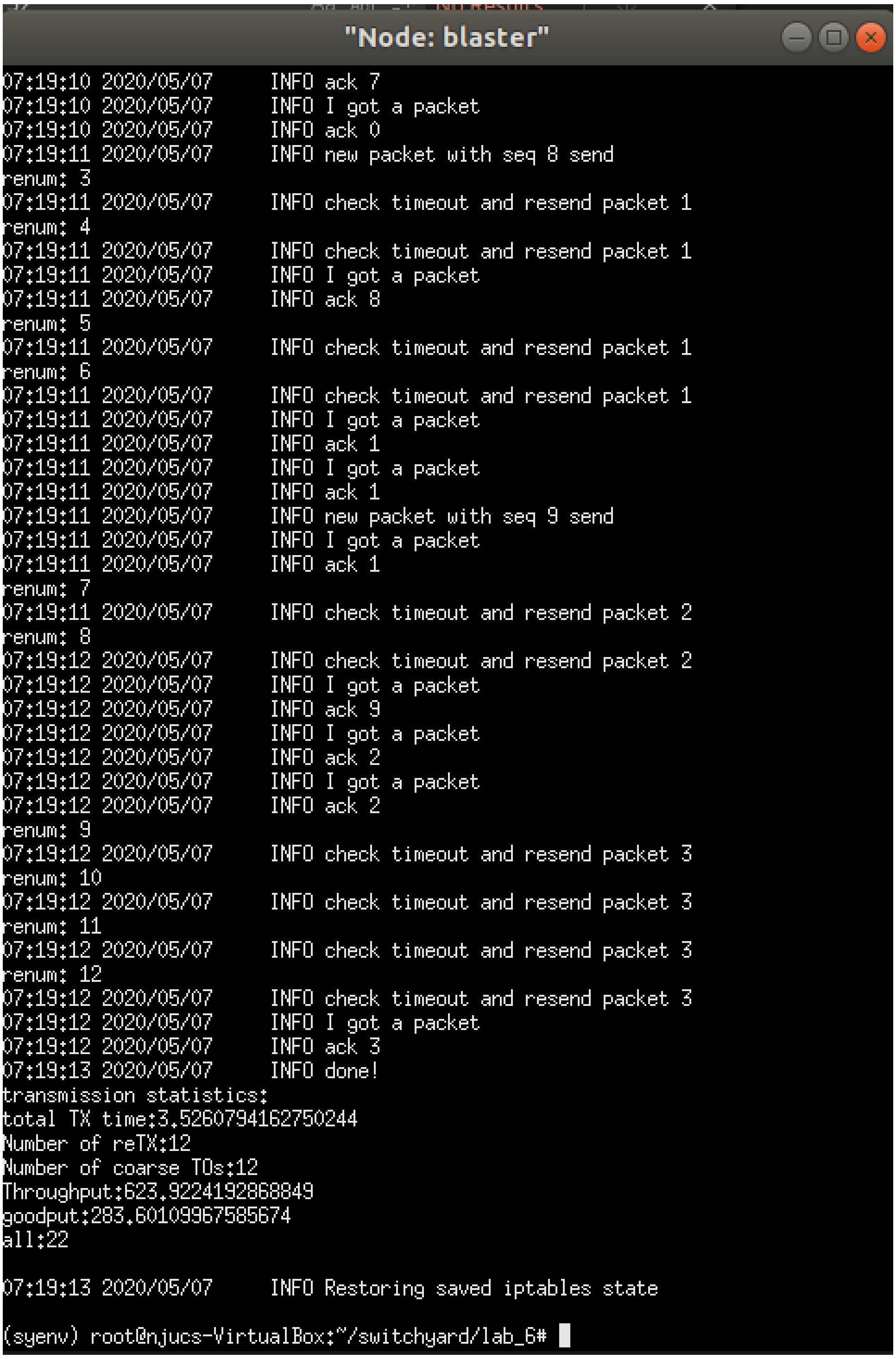
保持以下参数不变：

- length = 100 bytes
- recv_timeout = 100 ms
- timeout = 300 ms
- num = 10

得到的三组数据如下（附截图）：

	1	2	3
total TX time(seconds)	3.52	1.87	2.71
Number of reTX	12	4	8
Number of coarse TOs	12	4	8
Throughput(bps)	623.9	747.7	662.9
goodput(bps)	283.6	534.1	368.3

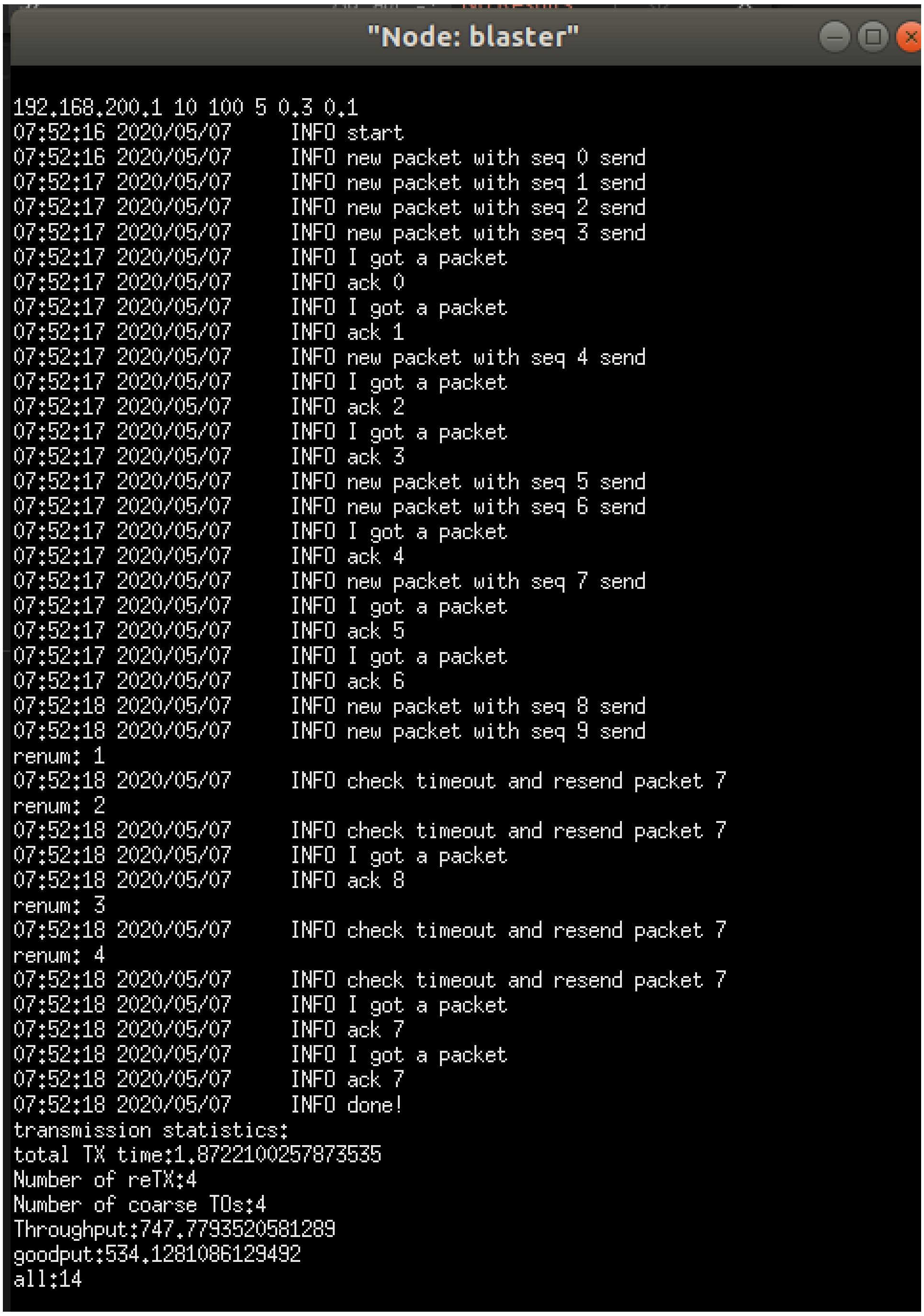
三次的Blaster的log截图如下：




```
07:19:10 2020/05/07     INFO ack 7
07:19:10 2020/05/07     INFO I got a packet
07:19:10 2020/05/07     INFO ack 0
07:19:11 2020/05/07     INFO new packet with seq 8 send
renum: 3
07:19:11 2020/05/07     INFO check timeout and resend packet 1
renum: 4
07:19:11 2020/05/07     INFO check timeout and resend packet 1
07:19:11 2020/05/07     INFO I got a packet
07:19:11 2020/05/07     INFO ack 8
renum: 5
07:19:11 2020/05/07     INFO check timeout and resend packet 1
renum: 6
07:19:11 2020/05/07     INFO check timeout and resend packet 1
07:19:11 2020/05/07     INFO I got a packet
07:19:11 2020/05/07     INFO ack 1
07:19:11 2020/05/07     INFO I got a packet
07:19:11 2020/05/07     INFO ack 1
07:19:11 2020/05/07     INFO new packet with seq 9 send
07:19:11 2020/05/07     INFO I got a packet
07:19:11 2020/05/07     INFO ack 1
renum: 7
07:19:11 2020/05/07     INFO check timeout and resend packet 2
renum: 8
07:19:12 2020/05/07     INFO check timeout and resend packet 2
07:19:12 2020/05/07     INFO I got a packet
07:19:12 2020/05/07     INFO ack 9
07:19:12 2020/05/07     INFO I got a packet
07:19:12 2020/05/07     INFO ack 2
07:19:12 2020/05/07     INFO I got a packet
07:19:12 2020/05/07     INFO ack 2
renum: 9
07:19:12 2020/05/07     INFO check timeout and resend packet 3
renum: 10
07:19:12 2020/05/07     INFO check timeout and resend packet 3
renum: 11
07:19:12 2020/05/07     INFO check timeout and resend packet 3
renum: 12
07:19:12 2020/05/07     INFO check timeout and resend packet 3
07:19:12 2020/05/07     INFO I got a packet
07:19:12 2020/05/07     INFO ack 3
07:19:13 2020/05/07     INFO done!
transmission statistics:
total TX time:3.5260794162750244
Number of reTX:12
Number of coarse TOs:12
Throughput:623.9224192868849
goodput:283.60109967585674
all:22

07:19:13 2020/05/07     INFO Restoring saved iptables state

(syenv) root@njucs-VirtualBox:~/switchyard/lab_6#
```



"Node: blaster"

```
192.168.200.1 10 100 5 0.3 0.1
07:52:16 2020/05/07     INFO start
07:52:16 2020/05/07     INFO new packet with seq 0 send
07:52:17 2020/05/07     INFO new packet with seq 1 send
07:52:17 2020/05/07     INFO new packet with seq 2 send
07:52:17 2020/05/07     INFO new packet with seq 3 send
07:52:17 2020/05/07     INFO I got a packet
07:52:17 2020/05/07     INFO ack 0
07:52:17 2020/05/07     INFO I got a packet
07:52:17 2020/05/07     INFO ack 1
07:52:17 2020/05/07     INFO new packet with seq 4 send
07:52:17 2020/05/07     INFO I got a packet
07:52:17 2020/05/07     INFO ack 2
07:52:17 2020/05/07     INFO I got a packet
07:52:17 2020/05/07     INFO ack 3
07:52:17 2020/05/07     INFO new packet with seq 5 send
07:52:17 2020/05/07     INFO new packet with seq 6 send
07:52:17 2020/05/07     INFO I got a packet
07:52:17 2020/05/07     INFO ack 4
07:52:17 2020/05/07     INFO new packet with seq 7 send
07:52:17 2020/05/07     INFO I got a packet
07:52:17 2020/05/07     INFO ack 5
07:52:17 2020/05/07     INFO I got a packet
07:52:17 2020/05/07     INFO ack 6
07:52:18 2020/05/07     INFO new packet with seq 8 send
07:52:18 2020/05/07     INFO new packet with seq 9 send
renum: 1
07:52:18 2020/05/07     INFO check timeout and resend packet 7
renum: 2
07:52:18 2020/05/07     INFO check timeout and resend packet 7
07:52:18 2020/05/07     INFO I got a packet
07:52:18 2020/05/07     INFO ack 8
renum: 3
07:52:18 2020/05/07     INFO check timeout and resend packet 7
renum: 4
07:52:18 2020/05/07     INFO check timeout and resend packet 7
07:52:18 2020/05/07     INFO I got a packet
07:52:18 2020/05/07     INFO ack 7
07:52:18 2020/05/07     INFO I got a packet
07:52:18 2020/05/07     INFO ack 7
07:52:18 2020/05/07     INFO done!
transmission statistics:
total TX time:1.8722100257873535
Number of reTX:4
Number of coarse TOs:4
Throughput:747.7793520581289
goodput:534.1281086129492
all:14
```

```
"Node: blaster"
07:54:35 2020/05/07     INFO ack 3
07:54:35 2020/05/07     INFO I got a packet
07:54:35 2020/05/07     INFO ack 4
renum: 1
07:54:35 2020/05/07     INFO check timeout and resend packet 1
07:54:35 2020/05/07     INFO I got a packet
07:54:35 2020/05/07     INFO ack 5
renum: 2
07:54:35 2020/05/07     INFO check timeout and resend packet 1
renum: 3
07:54:35 2020/05/07     INFO check timeout and resend packet 1
renum: 4
07:54:35 2020/05/07     INFO check timeout and resend packet 1
07:54:35 2020/05/07     INFO I got a packet
07:54:35 2020/05/07     INFO ack 1
07:54:35 2020/05/07     INFO new packet with seq 6 send
07:54:36 2020/05/07     INFO I got a packet
07:54:36 2020/05/07     INFO ack 1
07:54:36 2020/05/07     INFO new packet with seq 7 send
07:54:36 2020/05/07     INFO I got a packet
07:54:36 2020/05/07     INFO ack 1
07:54:36 2020/05/07     INFO new packet with seq 8 send
07:54:36 2020/05/07     INFO I got a packet
07:54:36 2020/05/07     INFO ack 6
07:54:36 2020/05/07     INFO new packet with seq 9 send
07:54:36 2020/05/07     INFO I got a packet
07:54:36 2020/05/07     INFO ack 7
07:54:36 2020/05/07     INFO I got a packet
07:54:36 2020/05/07     INFO ack 9
renum: 5
07:54:36 2020/05/07     INFO check timeout and resend packet 8
renum: 6
07:54:36 2020/05/07     INFO check timeout and resend packet 8
renum: 7
07:54:37 2020/05/07     INFO check timeout and resend packet 8
renum: 8
07:54:37 2020/05/07     INFO check timeout and resend packet 8
07:54:37 2020/05/07     INFO I got a packet
07:54:37 2020/05/07     INFO ack 8
07:54:37 2020/05/07     INFO I got a packet
07:54:37 2020/05/07     INFO ack 8
07:54:37 2020/05/07     INFO done!
transmission statistics:
total TX time:2.7150661945343018
Number of reTX:8
Number of coarse TOs:8
Throughput:662.9672615804282
goodput:368.31514532246007
all:18

07:54:37 2020/05/07     INFO Restoring saved iptables state

(syenv) root@njucs-VirtualBox:~/switchyard/lab_6#
```

分析数据包结构是否符合要求：

```
31 2.936881554   192.168.200.1   192.168.100.1   UDP    82 0 → 0 Len=40
32 3.035699620   192.168.100.1   192.168.200.1   UDP   190 0 → 0 Len=148
33 3.038065811   192.168.200.1   192.168.100.1   UDP    82 0 → 0 Len=40
34 3.142511934   192.168.200.1   192.168.100.1   UDP    82 0 → 0 Len=40

▸ Frame 32: 190 bytes on wire (1520 bits), 190 bytes captured (1520 bits) on interface 0
▸ Ethernet II, Src: 40:00:00:00:00:02 (40:00:00:00:00:02), Dst: 20:00:00:00:00:01 (20:00:00:00:00:01)
▸ Internet Protocol Version 4, Src: 192.168.100.1, Dst: 192.168.200.1
▸ User Datagram Protocol, Src Port: 0, Dst Port: 0
▾ Data (148 bytes)
    Data: 33202020202020202020202020202020202020202020202020…
    [Length: 148]

0020  c8 01 00 00 00 00 00 9c  c4 08 33 20 20 20 20 20   ·········  ·3
0030  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
0040  20 20 20 20 20 20 20 20  20 20 31 30 30 20 20 20             100
0050  20 20 20 20 20 20 20 20  20 20 30 20 20 20 20 20             0
0060  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
0070  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
0080  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
0090  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
00a0  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
00b0  20 20 20 20 20 20 20 20  20 20 20 20 20 20

Data (data.data), 148 bytes
```

可以看到，前32位（从 `b'33'` 开始）是sequence number = 3 空位填充20（空格符）的结果

紧接着的16位是length = 100 空位填充20的结果

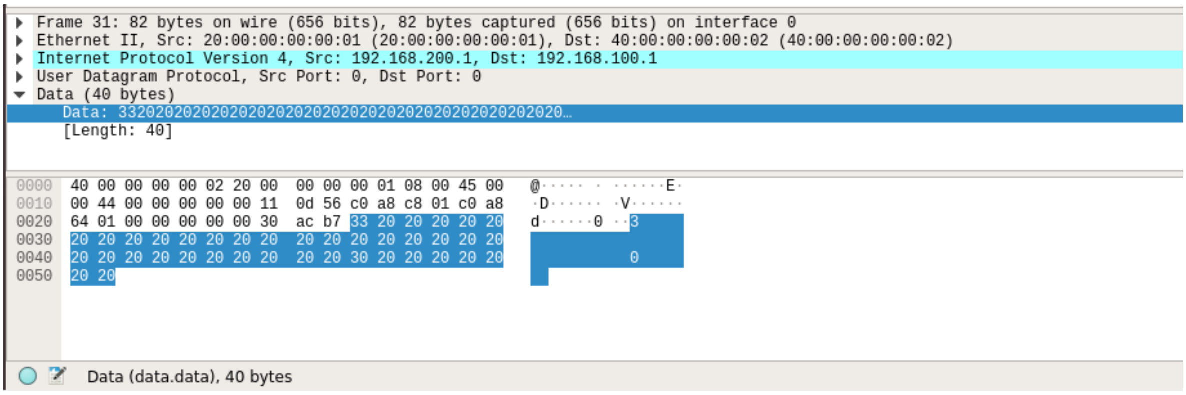
对应代码里的

```
        seq = str(sequence_num)
        length = str(payload_length)
        p += RawPacketContents(seq.ljust(32)+length.ljust(16))
```

最后100位（从 `b'30'` 开始）则是payload

对应代码里的

```
        payload = '0'.ljust(payload_length)
        p += RawPacketContents(payload)
```

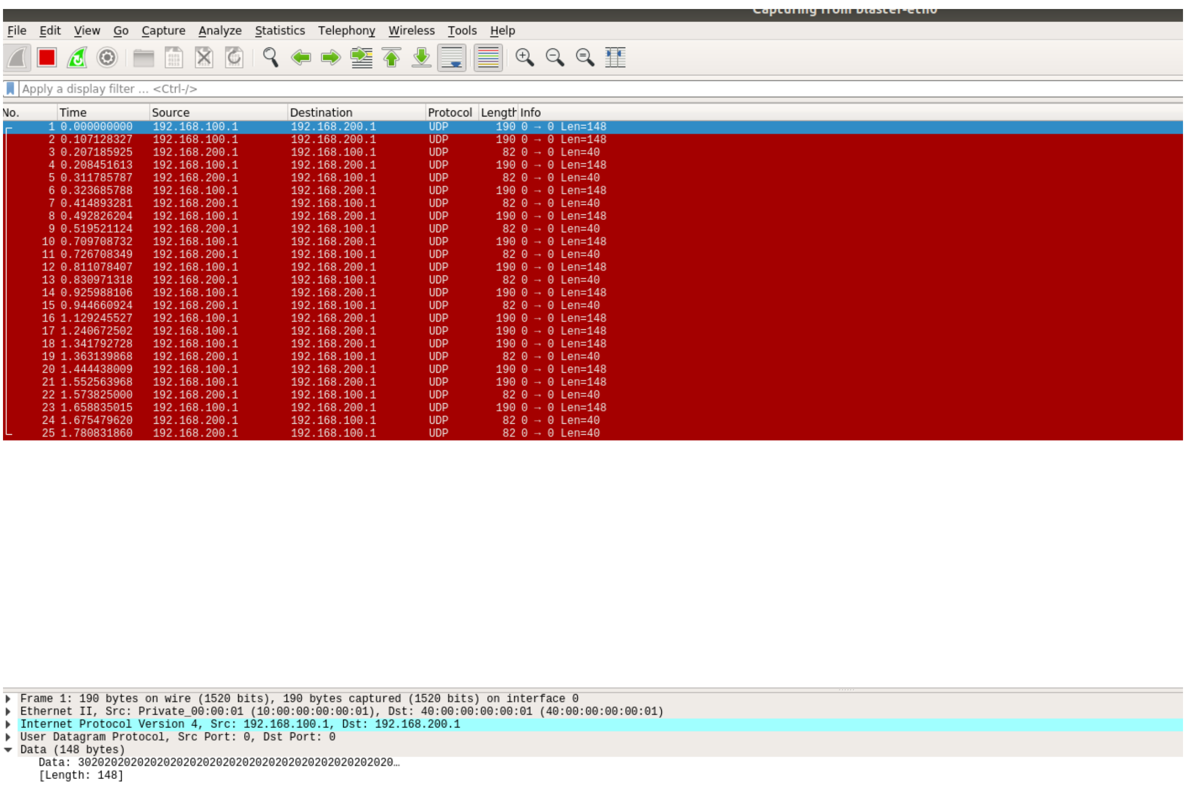


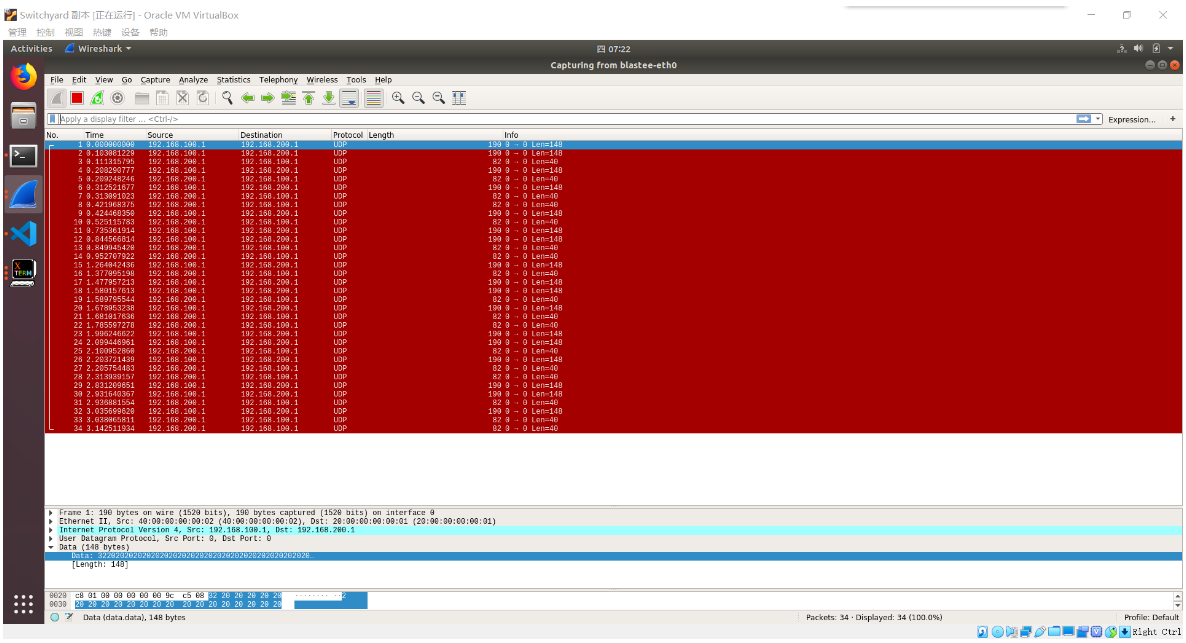
上图是ACK数据包的内容

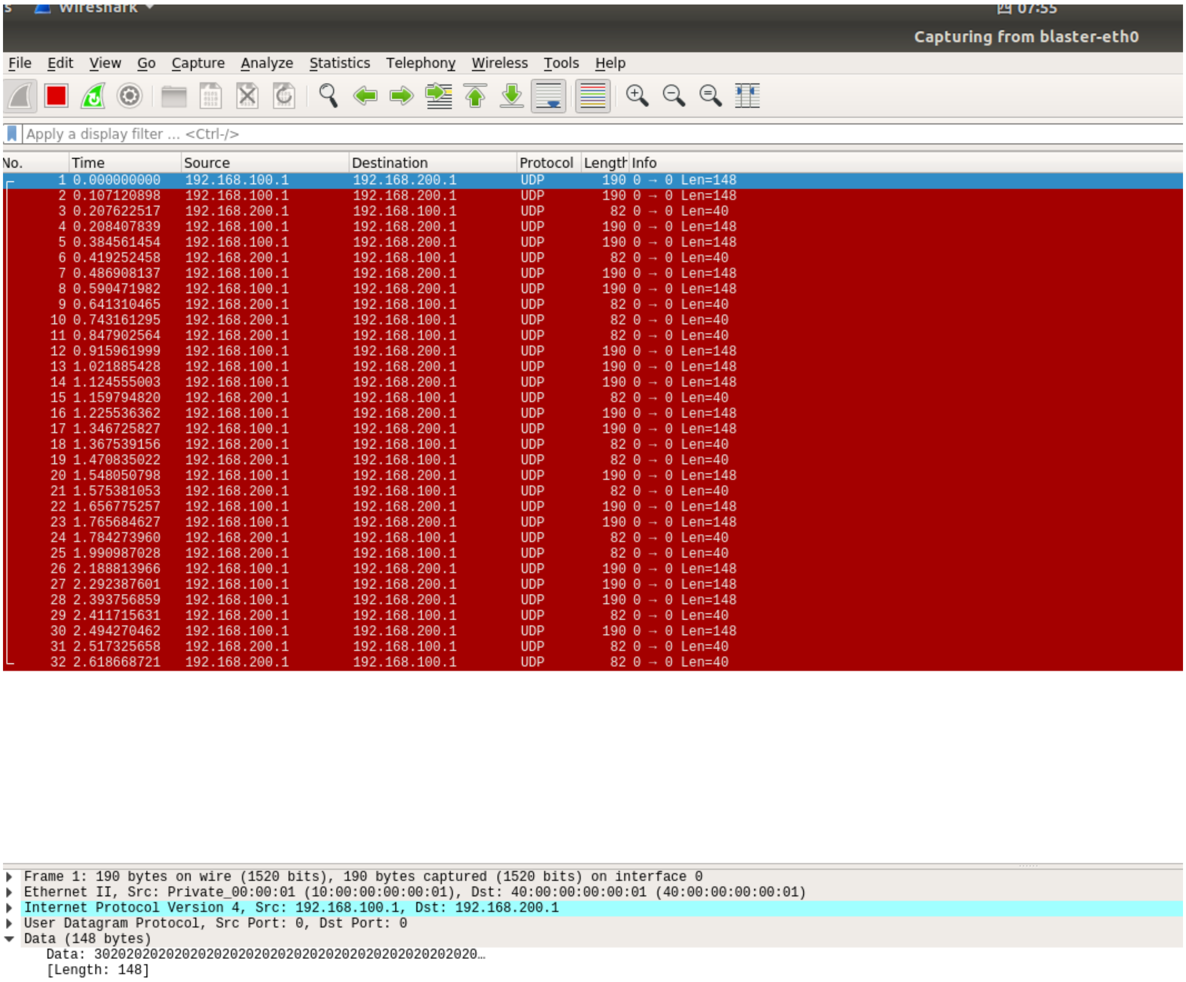
可以看到前32位为sequence number，空位填空格(20)，33对应的就是‘3’，就是该ACK的序列号

最后8位是 payload，从30开始，可以看到和数据包的payload前八位相同

数据包结构符合要求


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
No. Time Source Destination Protocol Length Info
1 0.000000000 192.168.100.1 192.168.200.1 UDP 190 0 → 0 Len=148
2 0.107128327 192.168.100.1 192.168.200.1 UDP 190 0 → 0 Len=148
3 0.207185925 192.168.200.1 192.168.100.1 UDP 82 0 → 0 Len=40
4 0.208451613 192.168.100.1 192.168.200.1 UDP 190 0 → 0 Len=148
5 0.311785787 192.168.200.1 192.168.100.1 UDP 82 0 → 0 Len=40
6 0.323685788 192.168.100.1 192.168.200.1 UDP 190 0 → 0 Len=148
7 0.414893281 192.168.200.1 192.168.100.1 UDP 82 0 → 0 Len=40
8 0.492826204 192.168.100.1 192.168.200.1 UDP 190 0 → 0 Len=148
9 0.519521124 192.168.200.1 192.168.100.1 UDP 82 0 → 0 Len=40
10 0.709708732 192.168.100.1 192.168.200.1 UDP 190 0 → 0 Len=148
11 0.726708349 192.168.200.1 192.168.100.1 UDP 82 0 → 0 Len=40
12 0.811078407 192.168.100.1 192.168.200.1 UDP 190 0 → 0 Len=148
13 0.830971318 192.168.200.1 192.168.100.1 UDP 82 0 → 0 Len=40
14 0.925988106 192.168.100.1 192.168.200.1 UDP 190 0 → 0 Len=148
15 0.944660924 192.168.200.1 192.168.100.1 UDP 82 0 → 0 Len=40
16 1.129245527 192.168.100.1 192.168.200.1 UDP 190 0 → 0 Len=148
17 1.240672502 192.168.100.1 192.168.200.1 UDP 190 0 → 0 Len=148
18 1.341792728 192.168.100.1 192.168.200.1 UDP 190 0 → 0 Len=148
19 1.363139868 192.168.200.1 192.168.100.1 UDP 82 0 → 0 Len=40
20 1.444438009 192.168.100.1 192.168.200.1 UDP 190 0 → 0 Len=148
21 1.552563968 192.168.100.1 192.168.200.1 UDP 190 0 → 0 Len=148
22 1.573825000 192.168.200.1 192.168.100.1 UDP 82 0 → 0 Len=40
23 1.658835015 192.168.100.1 192.168.200.1 UDP 190 0 → 0 Len=148
24 1.675479620 192.168.200.1 192.168.100.1 UDP 82 0 → 0 Len=40
25 1.780831860 192.168.200.1 192.168.100.1 UDP 82 0 → 0 Len=40
Frame 1: 190 bytes on wire (1520 bits), 190 bytes captured (1520 bits) on interface 0
Ethernet II, Src: Private_00:00:01 (10:00:00:00:00:01), Dst: 40:00:00:00:00:01 (40:00:00:00:00:01)
Internet Protocol Version 4, Src: 192.168.100.1, Dst: 192.168.200.1
User Datagram Protocol, Src Port: 0, Dst Port: 0
Data (148 bytes)
Data: 302020202020202020202020202020202020202020202020…
[Length: 148]


Switchyard 副本 [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
Activities Wireshark
07:22
Capturing from blastee-eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
Frame 1: 190 bytes on wire (1520 bits), 190 bytes captured (1520 bits) on interface 0
User Datagram Protocol, Src Port: 0, Dst Port: 0
Data (148 bytes)
Data (data.data), 148 bytes
Packets: 34 · Displayed: 34 (100.0%)
Profile: Default
Right Ctrl



总结与感想

本次实验让我理解了网络传输中的窗口机制，不过我觉得实验难度可以略微增加，这次实验的窗口大小是固定的，如果能够加入拥塞控制，根据网络状况调整窗口大小就更好了