Lab 6: reliable communication

191220138 杨飞洋

实验目的

学习路由器的工作原理，并通过代码实现其中的一部分。

具体功能为：接收并回复ICMP包，并对一些异常情况做出处理。

背景知识

interface接口

class `switchyard.lib.interface.`**`Interface`**(*name*, *ethaddr*, *ipaddr=None*, *netmask=None*, *ifnum=None*, *iftype=<InterfaceType.Unknown: 1>*) ¶

`ethaddr`
Get the Ethernet address associated with the interface

`ifnum`
Get the interface number (integer) associated with the interface

`iftype`
Get the type of the interface as a value from the InterfaceType enumeration.

`ipaddr`
Get the IPv4 address associated with the interface

`ipinterface`
Returns the address assigned to this interface as an IPInterface object. (see documentation for the built-in ipaddress module).

`name`
Get the name of the interface

`netmask`
Get the IPv4 subnet mask associated with the interface

packet接口

`get_header`(*hdrclass*, *returnval=None*)
Return the first header object that is of class hdrclass, or None if the header class isn't found.

`get_header_by_name`(*hdrname*)
Return the header object that has the given (string) header class name. Returns None if no such header exists.

`get_header_index`(*hdrclass*, *startidx=0*)
Return the first index of the header class hdrclass starting at startidx (default=0), or -1 if the header class isn't found in the list of headers.

`has_header`(*hdrclass*)
Return True if the packet has a header of the given hdrclass, False otherwise.

UDP接口

UDP (user datagram protocol) header

class switchyard.lib.packet.**UDP**(***kwargs*) [source]

The UDP header contains just source and destination port fields.

dst

length

src

To construct a packet that includes an UDP header as well as some application data, the same pattern of packet construction can be followed:

```
>>> p = Ethernet() + IPv4(protocol=IPProtocol.UDP) + UDP()
>>> p[UDP].src = 4444
>>> p[UDP].dst = 5555
>>> p += b'These are some application data bytes'
>>> print (p)
Ethernet 00:00:00:00:00:00->00:00:00:00:00:00 IP | IPv4 0.0.0.0->0.0.0.0 UDP | UDP 4444->5555 | RawPacketContents (37 bytes) b'These are '...
>>>
```

python byte操作

[资料链接]

实现逻辑

middlebox

`middlebox`类似一个路由器，只不过它只有两个端口，从这个端口进入，就从另一个端口发出，不用转发等操作，非常方便。

我们可以从`start_mininet.py`中读到关于端口的信息，以及`blastee`、`blaster`的`max`、`ip`地址。

```
def setup_addressing(net):
    '''
    * reset_macs call sets the MAC address of the nodes in the network
    * blaster and blastee has a single port, hence the MAC address ends with :01
    * middlebox has two ports, MAC address ends with :01 and :02 respectively,
that are connected to the blaster and blastee.
    '''
    reset_macs(net, 'blaster', '10:00:00:00:00:{:02x}')
    reset_macs(net, 'blastee', '20:00:00:00:00:{:02x}')
    reset_macs(net, 'middlebox', '40:00:00:00:00:{:02x}')
    '''
    * set_ip_pair call assigns IP addresses of the interfaces
    * convention is same as MAC address
    * middlebox has two IP addresses: 192.168.100.2 and 192.168.200.2 - connected
to blaster and blastee respectively
    '''
    set_ip_pair(net,
'blaster','middlebox','192.168.100.1/30','192.168.100.2/30')
    set_ip_pair(net,
'blastee','middlebox','192.168.200.1/30','192.168.200.2/30')
    set_route(net, 'blaster', '192.168.200.0/24', '192.168.100.2')
    set_route(net, 'blastee', '192.168.100.0/24', '192.168.200.2')
```

从上可以看到`middlebox` 和`blastee`相连的端口mac地址是`40:00:00:00:00:02`,`middlebox` 和`blaster`相连的端口mac地址是`40:00:00:00:00:01`， `blaster`的mac地址是`10:00:00:00:00:01`， `blastee`的mac地址是`20:00:00:00:00:01`。

需要处理一个特殊情况就是当`blaster`发给`blastee`时，可能会丢包，这是一个概率事件，概率是`dropRate="0.19"`

那可以用`randint(1,100)`函数来生成一个1到100的随机数来判断是否丢包，如果丢包了，就打印一下序列号，如果没丢包，就配置一下mac地址，从另一个端口发出去。

```
        if fromIface == "middlebox-eth0":
            _drop = randint(1,100)
            if _drop >= 100 * self.dropRate:
                eth_idx = packet.get_header_index(Ethernet)
                packet[eth_idx].src = '40:00:00:00:00:02'
                packet[eth_idx].dst = '20:00:00:00:00:01'
                self.net.send_packet("middlebox-eth1", packet)
            else:
                sequence = packet[3].to_bytes()[:4]
                seq = int.from_bytes(sequence,'big')
                print(seq," packet dropped")
```

如果是从`blastee`发给`blaster`的则不用考虑丢包，配置一下地址直接从另一个端口发就行了：

```
        elif fromIface == "middlebox-eth1":
            eth_idx = packet.get_header_index(Ethernet)
            packet[eth_idx].src = '40:00:00:00:00:01'
            packet[eth_idx].dst = '10:00:00:00:00:01'
            self.net.send_packet("middlebox-eth0", packet)
```

这就完成了`middlebox`的工作了。

关于python中`int`和`byte`之间的相互转换，并且用大端方式，在背景知识

blastee

`blastee`需要回发一个ack包，ack包的格式为：

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

-------------------------------------------------------------------------------
---------------------------

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

-------------------------------------------------------------------------------
--------------------------
```

首先肯定是设置包头，目的mac地址是`middlebox`与之相连的端口mac，目的ip是`blaster`的ip地址，ip层协议为UDP

`blastee`接收到的来自`blaster`的数据包的格式如下：

```
------- 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
-------------------------------------------------------------------------------
---------------------------
```

首先从packet获取序列号，是数据的前四个字节，长度是数据区的第5、6个字节，数据区的剩余就是数据。

序列号不变，直接放进ack包中。

ack包不用字节来指示长度。

当读取的packet包的`length < 8`时，说明数据不够8个字节，需要补0来对齐。反之，当读取的packet包的`length >= 8`时，说明此时数据字节数肯定够，则读`[6:14]`这前8个字节就行了。

这里需要python中int和byte相互转化，并且用大端方式，可在背景知识：python-byte操作可以找到教程。

实现代码如下：

```
        ack = Ethernet() + IPv4(protocol=IPProtocol.UDP) + UDP()
        ack[0].src = '20:00:00:00:00:01'
        ack[0].dst = '40:00:00:00:00:02'
        ack[1].src = '192.168.200.1'
        ack[1].dst = '192.168.100.1'
        ack[1].ttl = 10
        sequence = packet[3].to_bytes()[:4]
        #payload = packet[3].to_bytes[6:14]
        ack += sequence
        length = int.from_bytes(packet[3].to_bytes()[4:6], byteorder = 'big')
        if length < 8:
            ack += packet[3].to_bytes()[6:]
            ack += (0).to_bytes(8 - length, byteorder = "big")
        else:
            ack += packet[3].to_bytes()[6:14]
        self.net.send_packet(fromIface,ack)
```

blaster

主要实现一个滑动窗口功能。

滑动窗口的原理在手册中十分清晰，这里就不过多赘述了，主要来谈如何实现的。

在`blaster`类中定义一个队列`queue`来存放。

```
self.queue = []
for i in range(0,self.num+1):
    self.queue.append(mission())
```

定义一个`mission`类来存放各个序列号对应的包的状态

```
class mission:
    def __init__(self) -> None:
        self.is_acked = 0
        self.is_sent = 0
```

`is_acked`指示是否收到了ack包，`is_sent`指示有没有进行重发，来防止多余的重发。

首先来看下`__init__()`函数：

除了必要的参数之外，还初始化了这个队列，并且初始化了`timer`这个用于指示重发的时间变量，还有一系列的需要在最后打印的数值。

还有很重要的`rhs,lhs`来指示队列的左右指针。

```
        self.net = net
        # TODO: store the parameters
        self.blasterIp = blasterIp
        self.num = int(num)
        self.length = int(length)
        self.sw = int(senderWindow)
        self.timeout = int(timeout)
        self.recv_to = int(recvTimeout)
        self.lhs = self.rhs = 1
        self.begintime = self.endtime = -1
        self.ack_num = 0 //收到ack的数目
        self.timer = time.time()
        self.queue = []
        self.re_tranmit_num = 0 // 重传次数
        self.num_timeout = 0 //超时次数
        self.output_byte = 0 //总长度
        self.good_put = 0 //有效长度
        self.last_seq = -1 //上一次重传的seq
        for i in range(0,self.num+1):
            self.queue.append(mission())
```

接下来看`blaster`的工作逻辑：

`handle_packet`

如果收到了包，则执行`handle_packet()`函数，在这个实验中，很显然收到的包一定是从`blastee`发来的ack包，那只需要识别出序列号，将对应序列号的`is_acked`置为1，并且如果序列号是`self.lhs`，将`self.lhs`后移，并在此时更新`blaster`的时间。

但也是有情况是`self.lhs`不会移动，比如：

window：3 4 5 6 7

滑动窗口是 3 4 5 6 7，但是在3的包丢包了，收到了来自4 5 6 7的ack包，此时就不会移动ack，直到重发再收到3的ack包。

如果这个序列号的ack包此前没有被收到，则表示这是第一次收到，`goodput+=length`且`ack_num++`，如果`ack_num == num`，表示收到了所有的ack包，此时就可以设置`endtime`了。

代码如下：

```python
        sequence = packet[3].to_bytes()[:4]
        seq = int.from_bytes(sequence,'big')
        print("ack ",seq)
        if self.queue[seq].is_acked == 0:
            self.good_put += self.length
            self.queue[seq].is_acked = 1
            self.ack_num += 1
        if self.ack_num == self.num:
            self.endtime = time.time()
        i = 1
        while i < self.num+1 and self.queue[i].is_acked == 1:
            i += 1
        if i > self.lhs:
            self.lhs = i
            if self.lhs > self.rhs:
                self.rhs = self.lhs
            self.timer = time.time()
```

`handle_no_packet`

当没有收到ack包时，就执行`handle_no_packet()`函数。

在这个函数中，我会优先考虑是否超时了，如果超时了，代表序列号为`self.lhs`的包没有收到ack，即在`middlebox`上丢包了，此时会重发这个包。

`re_tranmit_num++`即重发数目自增。

比较上次重发的seq即`last_seq`，如果不一样则`num_timeout++`。

用`process_pkt()`函数来包装一下这个包，从端口发出，同时`output_byte += length`。

```python
        if time.time() - self.timer > self.timeout:
            self.re_tranmit_num += 1
            if self.last_seq != self.lhs:
                self.last_seq = self.lhs
                self.num_timeout += 1
            self.process_pkt(pkt,self.lhs)
            self.output_byte += self.length
            self.net.send_packet("blaster-eth0",pkt)
            print(self.lhs," timeout resend")
```

说下`process_pkt()`函数的实现，因为之后还会复用这个函数。

首先来看下数据包的结构：

```
------- 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  -----------------------------------------------------------------------
---------------------------------------
```

`sequence_number`自然就是序列号，`length`不用多说是在`blaster`中统一的，剩下的数据部分随意发挥。

再设置一下收发地址就行了，目的ip是`blastee`的ip，目的mac我选择的是`middlebox`与之相连的端口的mac地址

`process_pkt()`函数的实现如下：

```
    def process_pkt(self,pkt,seq):
        pkt[0].src = '10:00:00:00:00:01'
        pkt[0].dst = '40:00:00:00:00:01'
        pkt[1].src = '192.168.100.1'
        pkt[1].dst = '192.168.200.1'
        pkt[1].ttl = 10
        data = seq.to_bytes(4, 'big')
        data += self.length.to_bytes(2, 'big')
        pkt+= data
        payload = b'data ddata dddata ddddata dddddata'
        payload = payload[0:self.length-1]
        pkt += payload
```

处理完超时情况，现在需要对`rhs`做出处理，不可能只关注`lhs`吧，`rhs`也得往前推进。

如果`rhs - lhs + 1 <= sw`并且`rhs <= num`时，这项工作才有可能进行。

此时rhs、lhs都是合法的，那检查rhs是否发过去包了，如果没发，那就用`process_pkt()`包装一下，发出去，并且如果是第一次发，记录一下`begintime`，将`rhs`置为已发过包的状态，如果rhs和lhs之间的距离还能拉长，则rhs往右移动。

代码如下：

```
        if self.rhs - self.lhs < 5 and self.rhs <= self.num:
            if self.queue[self.rhs].is_sent == 0:
                if self.rhs == 1 and self.begintime == -1:
                    self.begintime = time.time()
                self.process_pkt(pkt,self.rhs)
                print("send ",self.rhs)
                self.output_byte += self.length
                self.net.send_packet("blaster-eth0",pkt)
                self.queue[self.rhs].is_sent = 1
                #self.queue[self.rhs].timer = time.time()
                if self.rhs - self.lhs < 4 and self.rhs < self.num:
                    self.rhs += 1
            elif self.rhs - self.lhs < 4 and self.rhs < self.num:
                self.rhs += 1
```

在while循环中，需要一个条件来结束循环，我选择判断`ack_num`是否和`num`相等。

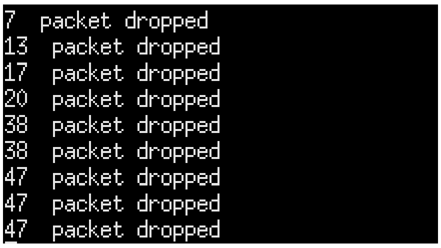
```python
        while True:
            try:
                recv = self.net.recv_packet(timeout=1.0)
            except NoPackets:
                self.handle_no_packet()
                continue
            except Shutdown:
                break

            self.handle_packet(recv)
            if self.ack_num == self.num:
                break
```

完成了`handle_packet()`和`handle_no_packet()`函数的编写，就基本上完成了`blaster`的构建了。

测试结果

现在来看下测试结果，我在`middle_box`丢包时打印了一下，在`blaster`收到`ack`包时打印，在`blaster`发包时打印，并且在`blaster`重发时打印，将`timeout`置为10，`num`置为50，加快测试速度，结果如下：

这些包是在middlebox上丢掉的。



```
7  packet dropped
13  packet dropped
17  packet dropped
20  packet dropped
38  packet dropped
38  packet dropped
47  packet dropped
47  packet dropped
47  packet dropped
```

能够看到比如7、13是丢了一次后重发接收到了的

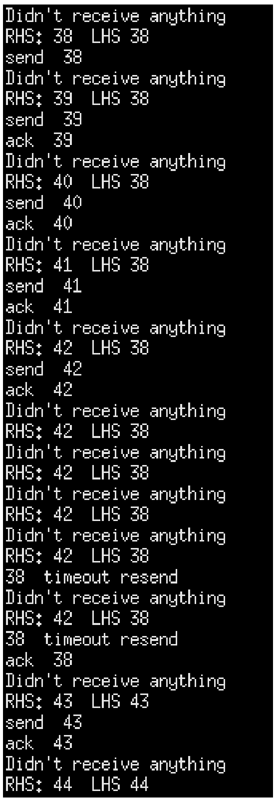
可以在`blaster`的打印信息中看到相应的解释：

```
Didn't receive anything
RHS: 7  LHS 7
send  7
Didn't receive anything
RHS: 8  LHS 7
send  8
ack  8
Didn't receive anything
RHS: 9  LHS 7
send  9
ack  9
Didn't receive anything
RHS: 10  LHS 7
send  10
ack  10
Didn't receive anything
RHS: 11  LHS 7
send  11
ack  11
Didn't receive anything
RHS: 11  LHS 7
Didn't receive anything
RHS: 11  LHS 7
Didn't receive anything
RHS: 11  LHS 7
Didn't receive anything
RHS: 11  LHS 7
7  timeout resend
ack  7
Didn't receive anything
RHS: 12  LHS 12
```

可以看到在丢失了7之后，blaster是继续之后的发包的，不过到了11就停下了，等到发现超时，就重发了7的包，可以看到打印了一行

`7 : timeout send`，就是重发了7这个包，立马收到了7的ack，然后就将rhs和lhs都更新为12了。

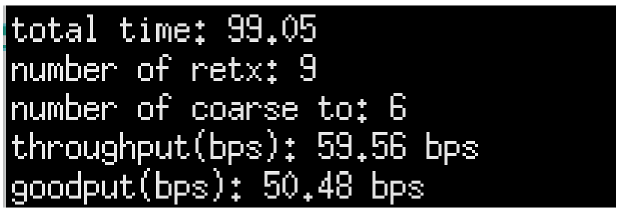
但比如38、47是重发的包依然丢了，以38为例。



```
Didn't receive anything
RHS: 38  LHS 38
send  38
Didn't receive anything
RHS: 39  LHS 38
send  39
ack  39
Didn't receive anything
RHS: 40  LHS 38
send  40
ack  40
Didn't receive anything
RHS: 41  LHS 38
send  41
ack  41
Didn't receive anything
RHS: 42  LHS 38
send  42
ack  42
Didn't receive anything
RHS: 42  LHS 38
Didn't receive anything
RHS: 42  LHS 38
Didn't receive anything
RHS: 42  LHS 38
Didn't receive anything
RHS: 42  LHS 38
38  timeout resend
Didn't receive anything
RHS: 42  LHS 38
38  timeout resend
ack  38
Didn't receive anything
RHS: 43  LHS 43
send  43
ack  43
Didn't receive anything
RHS: 44  LHS 44
```

可以看看到打印了两行的`38 : timeout send`，才正常工作。

在看下最后要求打印的数据：



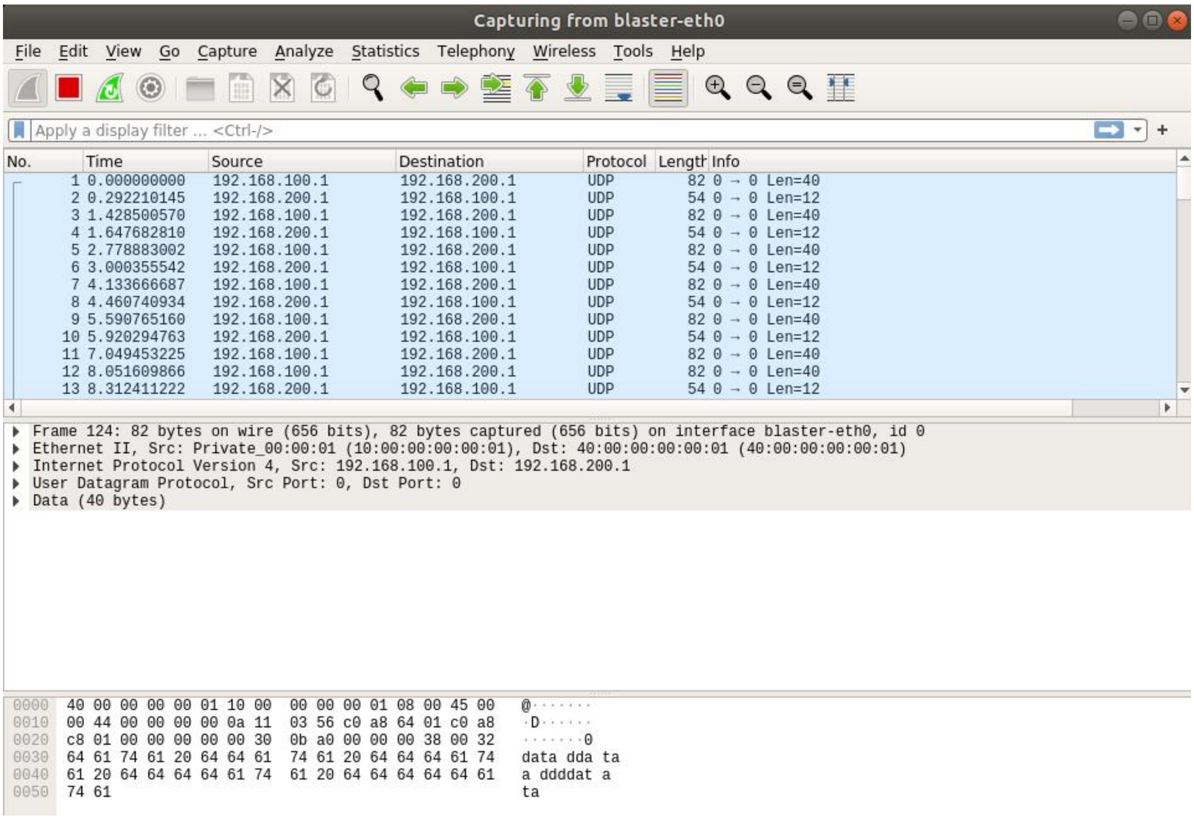
```
total time: 99.05
number of retx: 9
number of coarse to: 6
throughput(bps): 59.56 bps
goodput(bps): 50.48 bps
```

可以看到一共重发了9次，一共重发了6个不同的包，整个过程持续了99.05秒

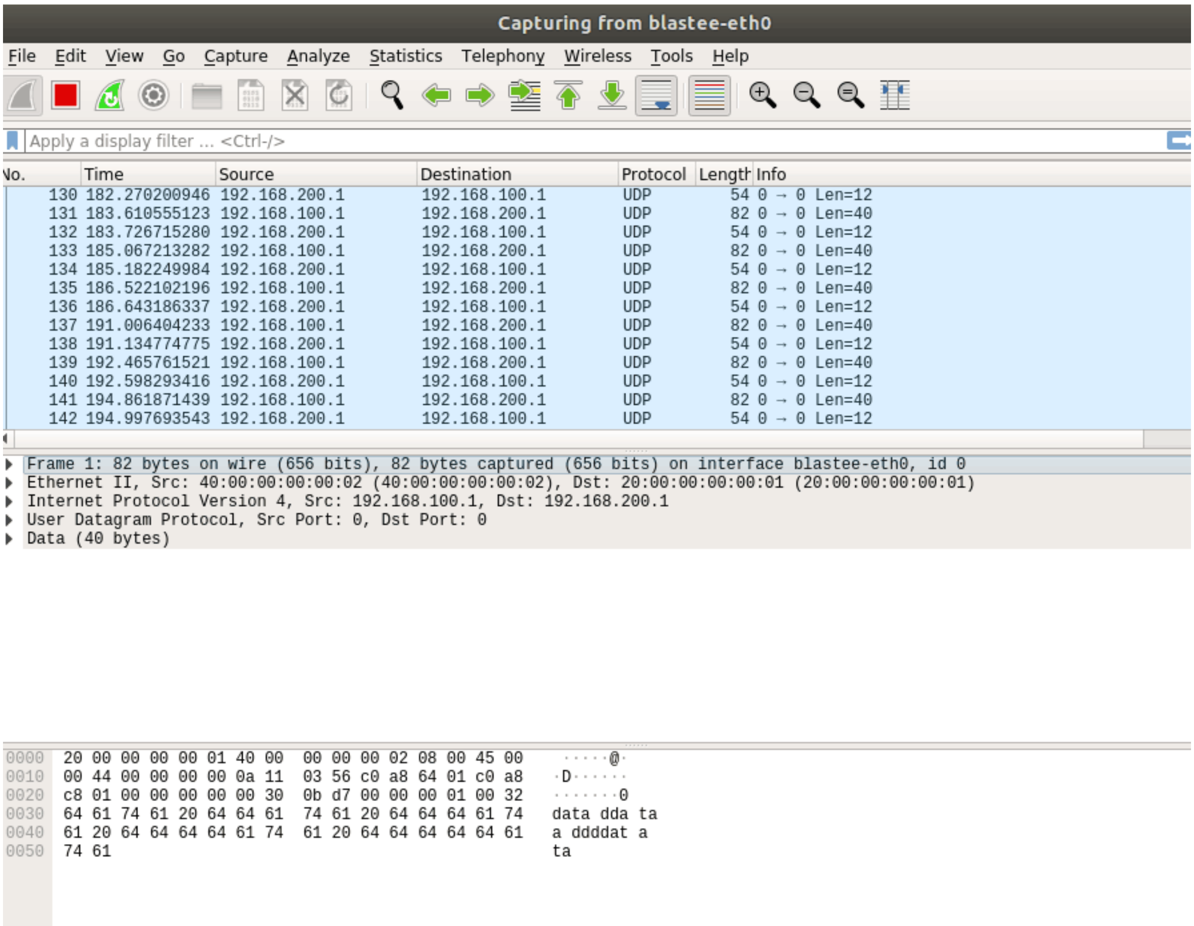
`throughput`是要略微比`goodput`大的，这是源于丢包的缘故。

重新开了一次测试，用wireshark监测。

blaster:



blastee:



可以从两张截图的下方观察到，blaster发出的数据，blastee收到了，工作顺利。

实验心得

通过代码复现滑动窗口协议，加深了理解。

也对python语言有了更多的掌握，尤其是main函数中`**kwargs`是很神奇。

对传输层的工作原理加深了理解。