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　　助教：

学院	计算机科学与技术	专业（方向）	计算机科学与技术
学号	191220029	姓名	傅小龙
Email	1830970417@qq.com	开始/完成日期	2021/5/21 - 2021/5/22

1.实验名称

Lab6 Reliable Communication

2.实验目的

模拟TCP协议构建一个简单的可靠传输模型。通过ACK回复和超时重传机制保证接收方能够收到所有的数据包。

3. 实验内容

3.1 Coding

3.1.1 Middlebox

网络中运输层一下的协议被抽象为一个黑盒`Middlebox`. `Middlebox`负责转发`Blastee`和`Blaster`发出的包。由于网络中只有这两个设备需要建立连接，根据实验手册要求，`Middlebox`的转发机制采用硬编码实现。

`Middlebox`还需要模拟网络中的丢包现象，丢包率由参数`droprate`给出. `Middlebox`仅可能会丢弃`Blaster`发出的包，对于`Blastee`发出的ACK信息保证传输的可靠性。

结合代码框架中已经给出的结构，`Middlebox`的执行逻辑如下：

```
                Received one packet. Where does it come?
                    /                   \
                   / blaster             \ blastee
                  ↙                       ↘
        drop it or not?             transfer it to blaster
    ↙yes        |no
do nothing      ↓
        transfer it to blastee
```

其中对于是否丢弃`blaster`发来的包的判断，将在区间`[0, 1]`之间生成一随机数与给定的参数`droprate`比较, 若该随机数小于丢包率，那么将之丢弃，否则将该包转发给`blastee`.

相关代码实现如下：

```
...
if fromIface == "middlebox-eth0":
    log_debug("Received from blaster")
    x = randint(0, 10)
```

```
    if float(x) / 10 >= self.dropRate:
        packet[0].src = mac_middlebox_ee
        packet[0].dst = mac_blastee
        self.net.send_packet("middlebox-eth1", packet)
        else :
            print(f"Drop a packet{packet}")
     elif fromIface == "middlebox-eth1":
        log_debug("Received from blastee")
        packet[0].src = mac_middlebox_er
        packet[0].dst = mac_blaster
        self.net.send_packet("middlebox-eth0", packet)
...
```

3.1.2 Blastee

接收方`Blastee`负责在收到`Blaster`发送来的数据包后回复ACK信息. 与TCP协议不同的是，这里ACK的序列号以包的数量而不是字节数为粒度.

根据实验手册描述，接收方不设滑动窗口，只是简单根据对收到的包的序列号回复ACK，ACK信息包括收到的包的序列号和一个8字节长的载荷（所收到包的前8字节的有效载荷）。

根据手册给出的`Blaster`所发包的结构，序列号位于第4个包头的前4个字节`Sequence number`字段，故序列号用以下方式取出：

```
sequence =  packet[3].to_bytes()[:4]
```

这里取出的序列号已经是大端存储格式（如果`Blaster`发来的包格式正确的话），可以直接将之拼接到ACK中。

收到包的有效载荷长度由第4个包头的后2个字节`Length`字段给出。若有效载荷长度小于8字节，则用空字符右补齐8字节，否则截取原有效载荷前8字节作为ACK的载荷. 代码实现如下：

```
...'''received one packet from blaster'''
ether = Ethernet(
    src = mac_blastee,
    dst = mac_middlebox,
    ethertype = EtherType.IPv4
)
ipv4 = IPv4(
    src = IPv4Address("192.168.200.1"),
    dst = self.blasterIp,
    ttl = 64,
    protocol = IPProtocol.UDP
)

sequence =  packet[3].to_bytes()[:4]
len_payload = int.from_bytes(packet[3].to_bytes()[4:6], 'big')
if len_payload >= 8:
    ack = ether + ipv4 + UDP()+ sequence + packet[3].to_bytes()[6:8]
    else:
        ack = ether + ipv4 + UDP() + sequence + packet[3].to_bytes()[6:] +
(0).to_bytes(8 - len_payload, "big")
self.net.send_packet(fromIface, ack)
print(f"ACK {int.from_bytes(sequence, 'big')} sent")
```

在发送完所有包的ACK后，`Blastee`应调用成员函数`shutdown()`。这里用字典`acked_table`记录，以序列号为键值，每发送一个ACK就令`acked_table[sequence_number] = True`，这样可以让重复ACK只有一条记录。当字典的键值队列长度达到`num`时，`Blastee`就可以调用`shutdown()`退出了：

```
while len(self.acked_table.keys()) < self.num:
    ...
self.shutdown()
```

3.1.3 Blaster

发送方`Blaster`需维护一个滑动窗口。这里采用`(List) window`来记录已发送包的状态，序列号从1开始，故第0项初始化为-1，其余num项初始化为`UNSENT`. 对于序列号为`i`的包，若`window[i] == UNSENT(0)`，那么说明该包已经加入窗口但是没有发送；若`window[i] == SENT(1)`,那么说明该包已经发送但是未收到ACK；若`window[i] == ACKED(2)`，那么说明该包已经收到ACK。

滑动窗口的大小由参数`senderWindow`给出，待发送的包总数由`num`给出,故窗口的左右端应满足`RHS - LHS + 1 ≤ SW && RHS < num + 1`

因而在`RHS - LHS < SW && RHS < num`时，可扩充滑动窗口，将RHS右移直至不满足这一条件。同时将新加入窗口的序列号的状态设为`UNSENT(0)`。这一操作在`Blaster`没有收到包时的"空闲时间"处理，相关代码实现如下：

```
if self.RHS - self.LHS < self.senderWindow and self.RHS <= self.num:
    # RHS can move so more pkts can send
    self.RHS = min(self.senderWindow + self.LHS, self.num + 1)
```

`Blaster`没有收到包时，还需要检查窗口中是否有已经超时需要重传的包。向`Blaster`类中添加成员`time`记录窗口上一次发生变化的时间，成员`state`记录`Blaster`的状态，分为正常传输`TRANSMIT(0)`和超时重传`RETRANSMIT(1)`两种状态。超时重传的时间由参数`timeout`给出。故当`time.time() - self.time > self.timeout`时，需要将`Blaster`的状态置为重传`RETRANSMIT`状态。相关代码的实现如下：

```
if time.time() - self.time > self.timeout:
    # timeout detected! need to retransmit every unacked packet in SW
    self.state = RETRANSMIT
    self.coarse_to += 1
    for i in range(self.LHS, self.RHS, 1) :
        if self.window[i] != ACKED :
            self.window[i] = UNSENT
    self.time = time.time() #reset self.time
```

在超时重传状态，需要将窗口中已发送但未收到ACK的包(`SENT(1)`)重新发送，变量`rtsm_ptr`记录本次要重传的包的序号。当队列中所有需要重传的包完成重传后，则将状态置为正常传输`TRANSMIT(0)`；在正常传输状态，将窗口中未发送的包(`UNSENT(0)`)发送。相关代码的实现如下：

```
if self.state == TRANSMIT :
    for i in range(self.LHS, self.RHS) :
        if(self.window[i] == UNSENT) :
            self.net.send_packet('blaster-eth0',self.make_pkt(pkt, i))
            self.window[i] = SENT
            self.throughput += self.length
```

```
            self.goodput += self.length
            break
elif self.state == RETRANSMIT:
    for i in range(self.rtsm_ptr, self.RHS) :
        if self.window[i] == SENT :
            self.net.send_packet('blaster-eth0',self.make_pkt(pkt, i))
            print(f"Resending Packet {i}")
            self.throughput +=  self.length
            self.num_retsmt += 1
            self.rtsm_ptr = i + 1
            break
    i = self.rtsm_ptr
    for i in range(self.rtsm_ptr, self.RHS):
        if self.window[i] == SENT:
            break
    if i == self.RHS - 1 and self.window[self.RHS - 1] != SENT:
        self.state = TRANSMIT
```

发送的包由`Blaster`的成员函数`make_pkt(...)`生成，根据手册给出的包结构，生成包的方法类似`Blastee`中的实现。这里暂用`hello blastee`作为发送的信息。相关代码实现如下：

```
pkt = Ethernet() + IPv4() + UDP()
pkt[1].protocol = IPProtocol.UDP
# Do other things here and send packet
pkt[0].src = mac_blaster
pkt[0].dst = mac_middlebox
pkt[1].src = IPv4Address('192.168.100.1')
pkt[1].dst = self.blasteeIp
pkt[1].ttl = 64
def make_pkt(self, pkt, sequence):
        my_header = sequence.to_bytes(4, 'big') + self.length.to_bytes(2, 'big')
        packet = pkt + my_header + 'hello blastee'.ljust(self.length, '
').encode(encoding='UTF-8')
        return packet
```

`Blaster`在收到ACK后，需要提取ACK中的序列号修改窗口中对应项的状态，并检查：若此时窗口左端对应的包状态为`ACKED`，那么将左端右移直到碰到`SENT`状态后者窗口右端。相关代码实现如下：

```
sequence = int.from_bytes(packet[3].to_bytes()[:4], 'big')
print(f"ACK {sequence} received")
if self.window[sequence] == SENT:
    self.window[sequence] = ACKED
    print(f"window[] = {self.window}")
    self.num_acked += 1
    for i in range(self.LHS, self.RHS - 1, 1):
        if self.window[i] == ACKED:
            self.LHS += 1
            self.time = time.time()
            else :
                break
```

在完成所有包的传输和收到对应的ACK后，需要打印相关的信息及其记录方法：

· 总耗时：在开始传输前记录一下时间，结束后用当前时间减去之即可得到。

· 重传次数：每重传一个包即重传次数+1

· 超时次数：每检测到一次超时即超时次数+1

· Throughput：每发送一个包即将总流量+length, 最后除以总耗时

· Goodput：仅在每个包首次发送时（即扩大窗口时）记录流量，最后除以总耗时

框架中`Blaster`的成员函数需要将`while`循环条件改为`self.num_acked < self.num`以使在传输结束后能正确的退出循环打印相关信息。

3.2 Deploying

3.2.1 Droprate = 0.19

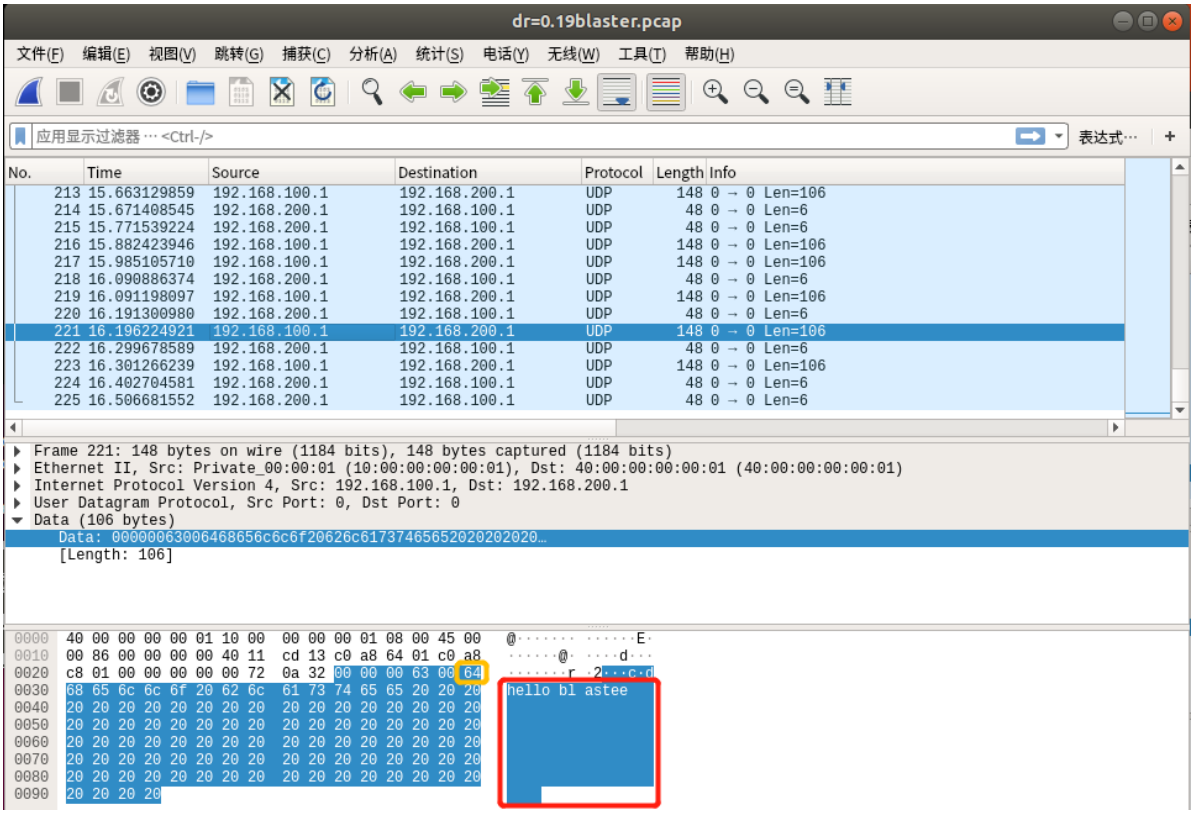
`middlebox`以0.19的丢包率运行，发送100个包，各参数如下所示：

```
middlebox# swyard middlebox.py -g 'dropRate=0.19'
blastee# swyard blastee.py -g 'blasterIp=192.168.100.1 num=100'
blaster# swyard blaster.py -g 'blasteeIp=192.168.200.1 num=100 length=100
senderWindow=5 timeout=500 recvTimeout=100'
```

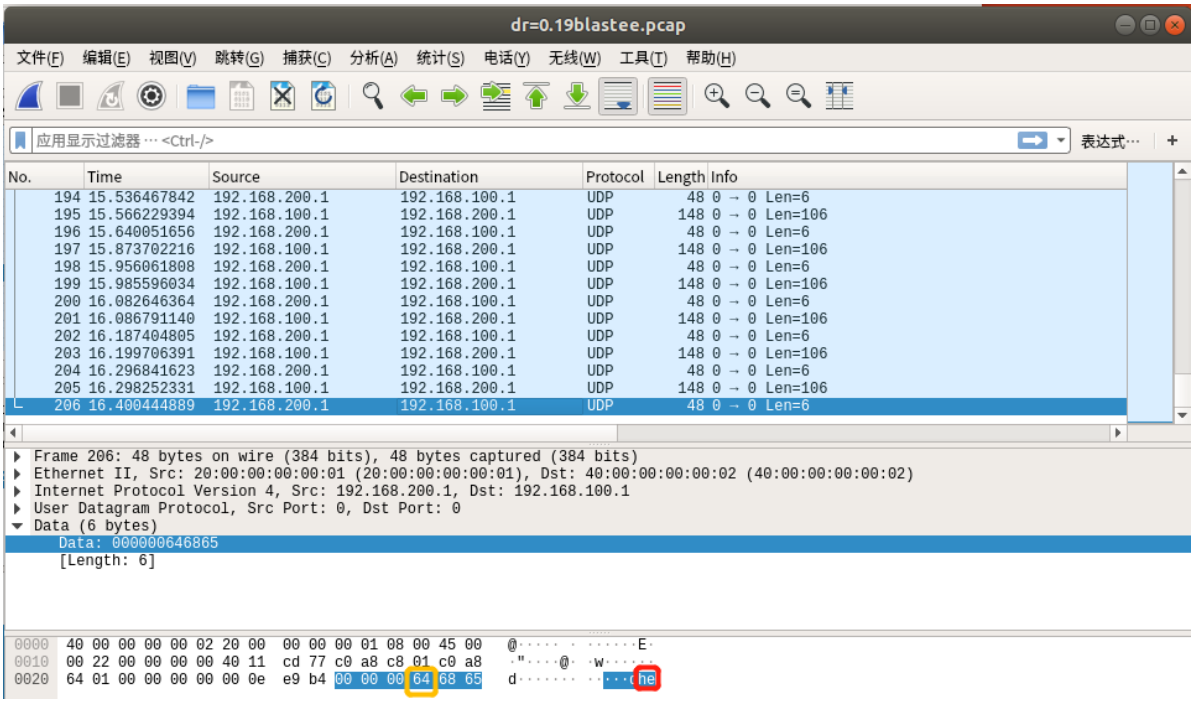
完成传输后`blaster`统计到的各项信息如下：

共发生了12次超时，与`droprate × num = 19`相近。超时造成的重传共有22个包。

使用wireshark在`blaster`和`blastee`的端口监听，得到如下结果：



上图是`blaster`端口处的监听结果。选中的第221个包为`blaster`发出的序列号为100的包，可以看到序列号的编码在图片下方黄色方框处(`0x64`)，有效载荷`hello blastee`在图片红色方框处，对应左侧的十六进制码。



上图是`blastee`端口处的监听结果。选中的第206个包为`blastee`发出的序列号为100的ACK，可以看到序列号的编码在图片下方黄色方框处(`0x64`)，有效载荷的前2个字节`he`在红色方框处。

3.2.2 Droprate = 0.00

将丢包率设置为0再次发送100个包，参数设置如下所示：

```
middlebox# swyard middlebox.py -g 'dropRate=0.19'
blastee# swyard blastee.py -g 'blasterIp=192.168.100.1 num=100'
blaster# swyard blaster.py -g 'blasteeIp=192.168.200.1 num=100 length=100
senderWindow=5 timeout=400 recvTimeout=100'
```

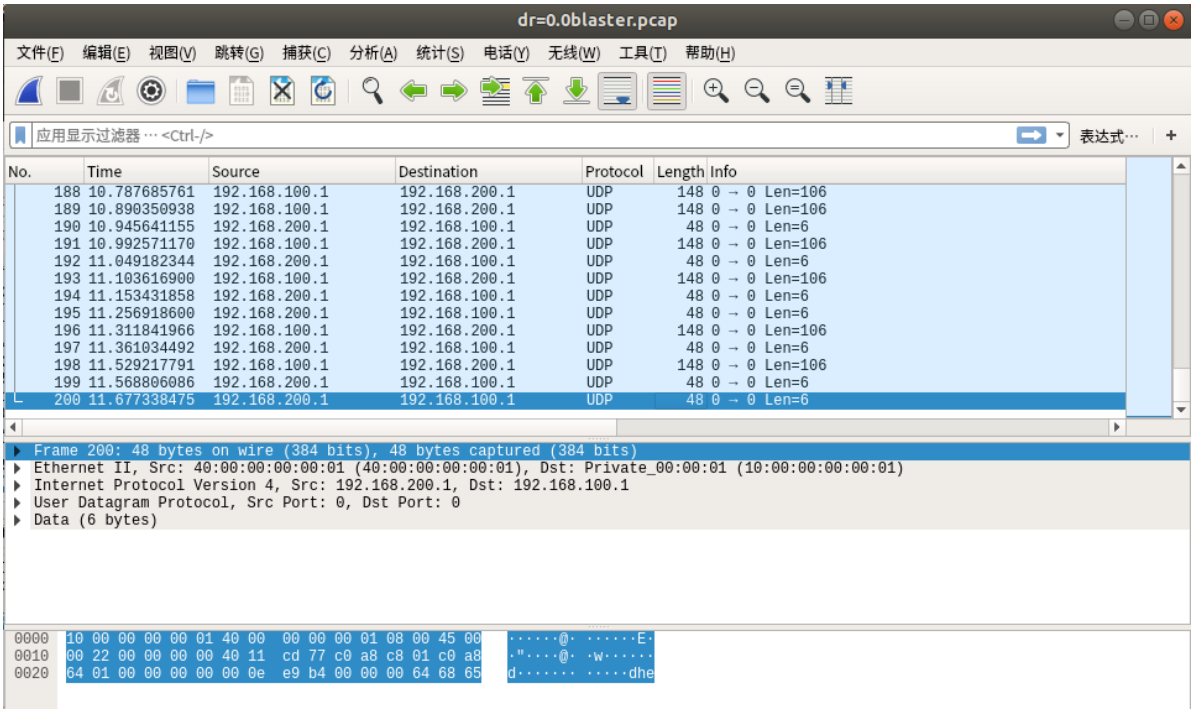
运行结果如下：

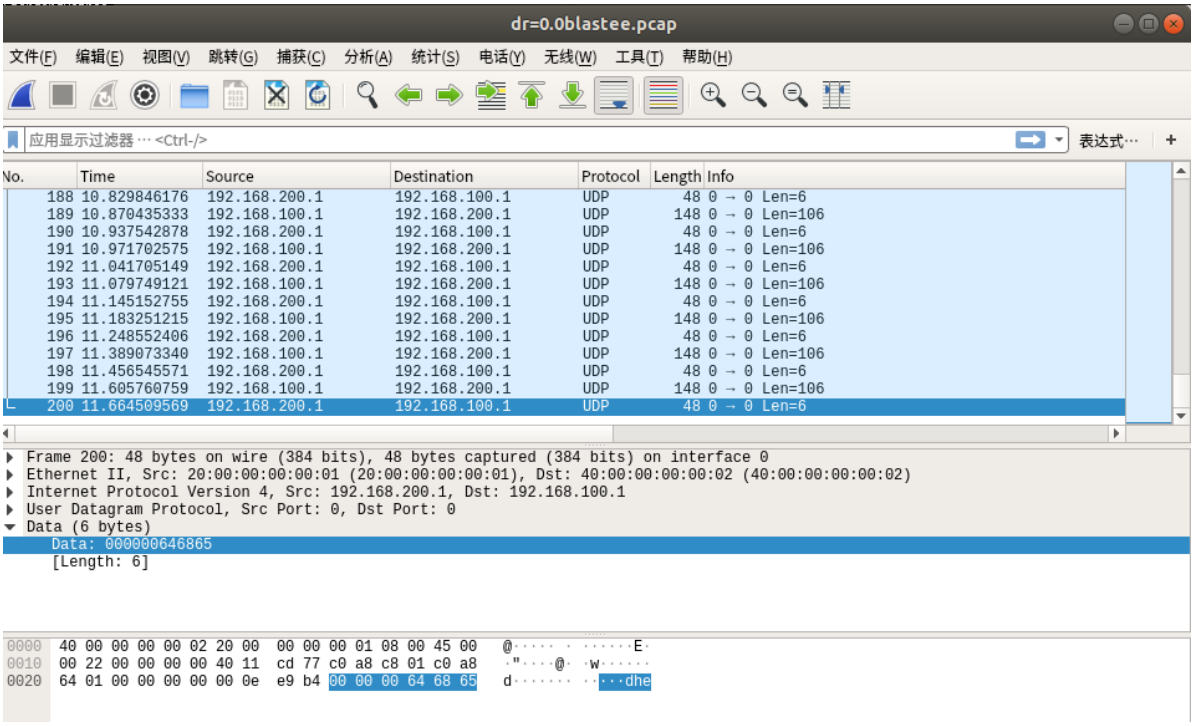
```
Total TX time = 11.646630048751831
Number of reTX = 0
Number of coarse TOs = 0
Throughput(Bps) = 858.6174677259281
Goodput(Bps) = 858.6174677259281
09:12:47 2021/05/24     INFO Restoring saved iptables state

(syenv) root@njucs-VirtualBox:~/cnLab/cnLab06/lab-6-191220029#
```

可以看到丢包率为0时没有发生超时，`Goodput = Throughput`

使用wireshark在`blaster`和`blastee`的端口监听，得到如下结果：





可以看到两边收到的包数量相同且都是`num * 2 =200`个包，说明没有丢包也没有重复发送包。

但是将`timeout`参数值调小100ms后，出现了超时重传：(`timeout = 300ms`)

这里出现超时的原因并不是网络中的丢包，而是发送方未来得及收到接收方的ACK就已经判断发生超时，导致重传的发生。

4.实验总结与感想

本次实验实现了一个类TCP协议，通过滑动窗口、ACK回复和超时重传机制保证了传输的可靠性。实验中设计的传输机制对实际的TCP协议作出了很大的简化，包括去除了传输速率控制，去除了接收方的滑动窗口等。