计算机网络 lab6

181840135 梁俊凯

Task1:

准备工作，复制相应文件到 lab_6 文件夹

Task2:

书写 middlebox 的工作逻辑，首先读入参数 drop_rate，作为包从 blaster 到 blastee 的丢失率，然后若收到 blaster 的包，则获得一个 0-1 的随机数 r，若 r<drop_rate，则丢弃之，否则转发之。不丢弃从 blastee 到 blaster 的 Ack 包。middlebox 的逻辑较简单。

Task3:

书写 blastee 的工作逻辑。首先通过 read_params 读入参数（blaster 的 IP 地址），然后每当收到来自 blaster 的包后，简要检查其是否包含 UDP 头部，若包含，则返回一个 Ack 包。其中 Ack 包的组成由 Eth+ipv4+udp+payload 组成，其中 seqNum 序列也在 payload 中，需要使用大端模式，若不足 8 字节需要补足（这是包的组装细节，与代码整体逻辑无关），部分代码展示如下：

```
if pkt.has_header(IPv4)==False or pkt.has_header(UDP)==False:
    continue
ori=pkt.get_header(RawPacketContents)
seqNum=int.from_bytes(ori.data[:4],'big')
payload=base64.b64decode(base64.b64encode(ori.data[6:]))
payload_byte=ori.data[6:]
if len(payload_byte)<8:
    payload_byte+='\0'.encode()*(8-len(payload_byte))
payload_byte=payload_byte[0:8]
print('\nseqNum:{}\n'.format(seqNum))
Ack=mk_udp(blastee.ethaddr,middleEth,blastee.ipaddr,blaster_ip,seqNum,payload_byte)
print('\nblastee ready to send Ack:{}\n'.format(Ack))
net.send_packet('blastee-eth0',Ack)
```

Task4:

书写 blaster 的代码逻辑。blaster 的执行框架也不繁琐，在每次 recv_packet 后，就进行一次 window 的更新，每次 recv_packet 流中只能发送一个包（或是重发，或是由于 window 更新，可以转发更大的 seqNum 的包）

但是由于在转发过程中要保存较多信息，故建立了一个 blaster 类，将所有操作和变量封装在其中。我将简要介绍其中较核心函数的执行流程。

1>recv_packet(self,pkt)函数：

根据 pkt 的 seqNum 信息更新 window 内的包的 Ack 情况，若 window[0].Ack==True 则意味着 LHS 需要加一，此时还应更新 window 的 time 信息（为重传做准备）

```
self.window[pos].Ack=True
if seqNum==self.num-1:
    self.last_ack_time=time.time()
    print('last_ack_time:{}'.format(self.last_ack_time))
print('set seqNum ack is true')
flag=False
while len(self.window) > 0 and self.window[0].Ack == True:
    self.LHS += 1
    del self.window[0]
    flag=True
if flag:
    self.reset_time()
```

如图，最后的 while 循环即是更新 window 的操作

2>send_pkt(self)函数

判断窗口里的包是否达到最大数量，若没有，则再发一个新包，RHS+=1。代码中还更新了一些信息变量（为了最后的信息打印），并且加入了一些调试信息。

3>resend(self)函数

根据窗口时间和包的 ack 信息，来决定重传。即若时间达到 timeout，则将没有得到 ack 消息的包重传。

```
if curtime-self.window_send_time>self.timeout:
    self.coarseTO+=1
    print('pkt resend: ',end=' ')
    for i,item in enumerate(self.window):
        if item.Ack==False:
            item.resendNum+=1
            self.net.send_packet(self.interface.name,item.pkt)
            print('{}'.format(i+self.LHS),end=' ')
            self.total_resend_pkt+=1
            self.total_send_pkt+=1
            is_resend=True
            #break
```

4>print_stat 函数

若所有文件传输完毕，调用 print_stat 函数，将讲义中所要求的信息打印，执行流结束。

```
def print_stat(self):
    print('\nall the stats\n')
    total_tx_time=self.last_ack_time-self.first_pkt_time
    send_num=self.total_send_pkt-self.total_resend_pkt
    print('Total TX time:{}(seconds)'.format(total_tx_time))
    print('Number of reTx:{}'.format(self.total_resend_pkt))
    print('Throughput(Bps):{}'.format(self.total_send_pkt*self.length/total_tx_time))
    print('Goodput: {}'.format(send_num*self.length/total_tx_time))
```

整体流程：

每次 while 循环：若收到包，调用 recv 函数更新 window 表，不论是否收到包，都进入一次 resend 函数进行一次重传判断，若未重传包，还需要进入 send 函数决定是否要发送新的包（当 window 的长度未达到最大长度时。

Task5:测试代码

我通过一组特定的参数来测试实现的正确性。

各文件配置如下:

middlebox_params.txt:

-d 0.4

blastee_params.txt:

-b 192.168.100.1 -n 20

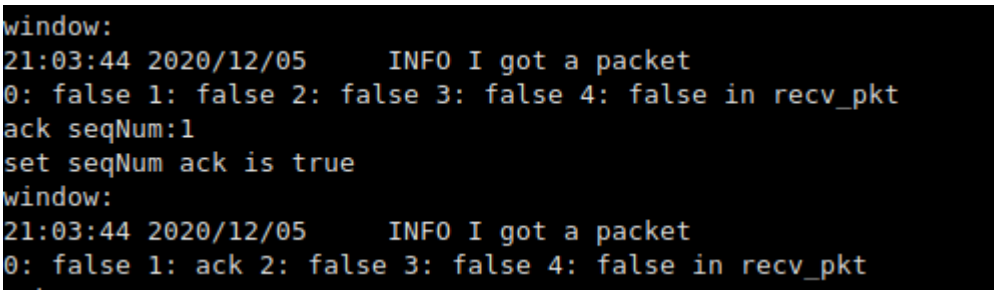
blaster_params.txt:

-b 192.168.200.1 -n 20 -l 100 -w 5 -t 1000 -r 100

考虑到 middlebox 仅仅完成转发和模拟丢包的功能，而 blastee 收到包就发送一条 Ack 回复，两者逻辑较简单，故 wireshark 验证中只展开 blaster 的端口进行详细的分析。其它的验证通过终端的调试信息来展开。

由于分析每一步 blaster 的执行比较繁琐，故在此选取 blaster 执行的侧面来进行分析（即程序特定的状态）

1>首先 blaster 发送包，每当收到包时更新 ack 信息



可以看到，window 有五个包，但都没有收到 ack 回复，此时收到一个 seqNum 为 1 的 ack 包，并按照预想的流程将包 1 的信息更新为 ack（True）

此时包 0 的信息还没有收到回复，故猜想包 0 应该是被 middlebox 丢掉了（因为 blastee 是按收到的顺序进行回复的），查看 middlebox 的终端输出

```
got pkt:0
random:0.06882775053342582
the packet was dropped
```

果然序列为 0 的包被丢弃了，代码执行符合预期。

2>超时重传机制

```
window:
0: false 1: ack 2: ack 3: ack 4: false
pkt resend:  0 4
```

可见此时 0 和 4 的包均未收到 ack 回复，故在收到下一个包前需要不断重传两个包，在重传前会打印调试信息 pkt resend 由图可知重传了包 0 4，这时考虑到如果不更新 window 的 time，则每次循环都会重传两个包，造成浪费，所以应该更新 window 的 time 变量，保证不过多的重传。（但是讲义上仅说明了 LHS 移动时才更新 time 变量，这里可能算是一个补充或改动）

3>LHS 与 RHS 的移动

```
window:
0: false 1: ack 2: ack 3: ack 4: false
pkt resend:  0 4
21:17:09 2020/12/05     INFO I got a packet
in recv_pkt
ack seqNum:0
set seqNum ack is true
window:
4: false
```

在这里看到当收到 0 的 ack 回复后，0,1,2,3 都为 ack，故 window 应仅含 4 一个包，在最下方的输出中证明了这一点。

然后此时 window 的大小为 1，小于 5，可以预料到在一段时间后 window 的长度将重新达到最大（RHS 的移动）

```
window:
5: false 6: false 7: false 8: false
```

此时 window 的长度达到 4，均为收到 ack 回复。

4>打印信息

```
all the stats

Total TX time:7.896686792373657(seconds)
Number of reTx:13
Total Coarse timeouts:7
Throughput(Bps):417.8967821272871
Goodput: 253.27077704684066
```

可以发现 Throughput 比 Goodput 大很多，因为 Throughput 将重传的包的比特也计算在内了。

使用 wireshark 分析结果很难进行，因为序列号在 payload 中用大端存储，而 wireshark 不能解析 payload 中的数据，所以用 wireshark 验证结果是不适宜的。

```
352 1790.1288086… 192.168.100.1  192.168.200.1  UDP  148 0 → 0 Len=106
353 1790.3384325… 192.168.200.1  192.168.100.1  UDP   54 0 → 0 Len=12
354 1790.3895736… 192.168.100.1  192.168.200.1  UDP  148 0 → 0 Len=106
355 1790.4932231… 192.168.100.1  192.168.200.1  UDP  148 0 → 0 Len=106
356 1790.5968889… 192.168.100.1  192.168.200.1  UDP  148 0 → 0 Len=106
357 1790.7008696… 192.168.100.1  192.168.200.1  UDP  148 0 → 0 Len=106
358 1790.7543286… 192.168.200.1  192.168.100.1  UDP   54 0 → 0 Len=12
359 1790.8030003… 192.168.100.1  192.168.200.1  UDP  148 0 → 0 Len=106
360 1790.9625514… 192.168.200.1  192.168.100.1  UDP   54 0 → 0 Len=12
361 1791.0665718… 192.168.200.1  192.168.100.1  UDP   54 0 → 0 Len=12
362 1791.4171640… 192.168.100.1  192.168.200.1  UDP  148 0 → 0 Len=106
363 1791.4174338… 192.168.100.1  192.168.200.1  UDP  148 0 → 0 Len=106
364 1791.6903195… 192.168.200.1  192.168.100.1  UDP   54 0 → 0 Len=12
365 1791.7414477… 192.168.100.1  192.168.200.1  UDP  148 0 → 0 Len=106
366 1792.0023448… 192.168.200.1  192.168.100.1  UDP   54 0 → 0 Len=12
367 1792.7570940… 192.168.100.1  192.168.200.1  UDP  148 0 → 0 Len=106
368 1792.9385197… 192.168.200.1  192.168.100.1  UDP   54 0 → 0 Len=12
```

上图为 wireshark 的抓包结果（在 blaster 上）

通过本次实验我初步了解了在传输层上的保证机制，并看到了软件（重传协议）对传输速率的影响是很大的，硬件允许的最大速率并不等于实际速率，应根据应用选择合适的协议。
后续我会自主探索 TCP 包的传输，感受其与 UDP 的不同（理论上 TCP 的重传信息不应在应用层的数据报上）

本次实验报告到此结束，感谢阅读！