Lab7_Report

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	专业（方向）	计算机
学号	181860077	姓名	佘帅杰
Email	3121416933@qq.com	开始完成日期	2020.6.10

实验名称

计算机网络试验7

实验目的

学习防火墙的实现和原理

实验内容

实现防火墙功能，包括规则解析和发包与否的规则判断

核心代码

由于本次实验的代码修改集中在firewall.py，下面将基于功能划分进行简单的代码解析

数据结构Rule

基于任务构造数据结构

为了能够泛化到所有的规则，所以默认都设置为空，后续可以利用这个做一些判断

mode就是permit或者是deny，netmode保存的是ip,udp,icmp,tcp，其他的变量见名字即可

show_info函数是用于测试文件读写功能设计的，无实际用途

```
class Rule():
    def __init__(self,modes=None,netmodes=None,srcs=None,dsts=None,srcports=None,dstports=None,ratelimits=None,impairs=None):
        self.mode=modes
        self.netmode=netmodes
        self.src=srcs
        self.dst=dsts
        self.srcport=srcports
        self.dstport=dstports
        self.ratelimit=ratelimits
        self.impair=impairs
    def show_info(self):
        print(self.mode,self.netmode,self.src,self.srcport,self.dst,self.dstport,self.ratelimit,self.impair)
```

Posssess file

这个函数是针对文件进行读取解析

读入一个文件，按行读取，读取到了空，就直接直接接着读下去

然后取出末尾的换行符，再用空格进行分割得到一个列表

如果是#开头就是注释不管

非#开头的，创建一个新的数据结构

每一条规则都一定有且位置固定的就是mode，netmode，src直接保存即可

如果规则有srcport就对应的分配填写，没有srcport就不填那些，直接填dst即可

同理可以查看是否有ratelimit和impair，对应设置即可

最后加入列表，函数结束时返回列表

```
def possesse_file():
    Rule_List=[]
    with open("firewall_rules.txt") as file:
        while 1:
            line = file.readline()
            if not line:
                break
            else:
                while line.isspace():
                    line = file.readline()
                line=line.strip('\n')
                d=line.split(" ")
                if d[0]!='#':
                    temp=Rule()
                    temp.mode=d[0]
                    temp.netmode=d[1]
                    temp.src=d[3]
                    if "srcport" in line:
                        temp.srcport=d[5]
                        temp.dst=d[7]
                        temp.dstport=d[9]
                    else:
                        temp.dst=d[5]
                    if "ratelimit" in line:
                        for i in range(len(d)):
                            if d[i]== "ratelimit":
                                temp.ratelimit=d[i+1]
                                break
                    if "impair" in line:
                        temp.impair=True
                    else:
                        temp.impair=False
                    Rule_List.append(temp)

    return Rule_List
```

Match系列函数

这个系列的函数是针对不同的netmode的规则进行匹配，传入的参数就是rule和一个pac包

下面的是最基本的函数，作为分类器，根据netmode调用

```python
def match_rule(rule,packet):
    if rule.netmode =="ip":
        return match_ip(rule,packet)
    if rule.netmode=="udp":
        return match_udp(rule,packet)
    if rule.netmode=="tcp":
        return match_tcp(rule,packet)
    else:
        return match_icmp(rule,packet)
```

对应规则的种类在调用前就已经确定了，下面举个ip的例子

判断有没有ip头，没有ip头直接返回失配。有的话判断是不是any，默认的flag=1，所以any可以跳过

不是any就用逻辑操作判断即可，参考于实验手册

```python
def match_ip(rule,packet):
    flag=1
    if packet.has_header(IPv4):
        if rule.src !="any":
            net1=IPv4Network(rule.src,strict=False)
            if int(net1.network_address) & int(packet[IPv4].src) != int(net1.network_address):
                flag=0
        if rule.dst !="any":
            net1=IPv4Network(rule.dst,strict=False)
            if int(net1.network_address) & int(packet[IPv4].dst) != int(net1.network_address):
                flag=0
    else:
        flag=0
    return flag
```

再举一个udp的例子，udp在ip的基础上多了端口号的匹配，同样是对不匹配这个情况进行匹配判断

any特殊情况处理即可

```python
def match_udp(rule,packet):
    flag=1
    if packet.has_header(IPv4):
        if rule.src !="any":
            net1=IPv4Network(rule.src,strict=False)
            if int(net1.network_address) & int(packet[IPv4].src) != int(net1.network_address):
                flag=0
        if rule.dst !="any":
            net1=IPv4Network(rule.dst,strict=False)
            if int(net1.network_address) & int(packet[IPv4].dst) != int(net1.network_address):
                flag=0
    else:
        flag=0
    if packet.has_header(UDP):
        if rule.srcport !="any" and int(rule.srcport) != packet[UDP].src:
            flag=0
        if rule.dstport !='any' and int (rule.dstport)!=packet[UDP].dst:
            flag=0
    else:
        flag=0
    return flag
```

TCP的处理和UDP基本一样，ICMP和IP的处理比较相似，不再举例

Kernel

预处理

在main函数的头部处理文件得到规则列表，初始化一个token_bucket为后续的限速功能准备

限速只对ratelimit生效，所以用字典，key是下标，初始化为0

设置两个计时器cur_time指示当前的实践，set_time就是上一次添加令牌的时间

```
def main(net):
    # assumes that there are exactly 2 ports
    portnames = [ p.name for p in net.ports() ]
    portpair = dict(zip(portnames, portnames[::-1]))
    rules=possesse_file()
    index=0
    token_bucket={}
    for i in rules:
        if i.ratelimit!=None:
            token_bucket[index]=0
        index+=1
    cur_time=time.time()
    set_time=time.time()
```

循环的开始

上面提到的数据结构，每一次循环的开始检查一下时间以及桶是不是满了

```
cur_time=time.time()
if cur_time-set_time>0.5:
    for i in token_bucket:
        if token_bucket[i]<int(rules[i].ratelimit)*2:
            token_bucket[i]+=int(rules[i].ratelimit)/2
    set_time=time.time()
```

对当前收到的包，判断一下所有规则是不是匹配，如果匹配就停止遍历，进入如果是permit做下一步，反之跳过。对于permit的情况，判断一下这个流是否有损坏标记，如果有则开始随机数，设置为1%的丢包清零send_flag，然后是判断有没有限速，如果有就计算报的大小，判断对应的桶的令牌数，然后看看发不发。

```
if pkt is not None:

    # This is logically where you'd include some  firewall
    # rule tests.  It currently just forwards the packet
    # out the other port, but depending on the firewall rules
    # the packet may be dropped or mutilated.
    #print(str(pkt),input_port)
    index=0
    match_flag=0
    for i in rules:
        index+=1
        if match_rule(i,pkt):
            match_flag=1
            break
    if match_flag:
        print("Match rule index",index)
    else:
        print("No match")
    if match_flag:
        if rules[index-1].mode=="permit":
            send_flag=1
            if rules[index-1].impair:
                temp=randint(1,100)
                if temp<2:
                    send_flag=0
            if send_flag:
                if rules[index-1].ratelimit!=None:
                    use_size=len(pkt)-len(pkt.get_header(Ethernet))
                    if use_size<=token_bucket[index-1]:
                        token_bucket[index-1]-=use_size
                        net.send_packet(portpair[input_port], pkt)
                else:
                    net.send_packet(portpair[input_port], pkt)
    else:
        net.send_packet(portpair[input_port], pkt)
```

测试

测试样例

下面是使用测试样例（群里的新版）的测试结果

可以看到输出的结果和测试样例里的提示信息一模一样

（具体的原理不再赘述，基本就是匹配，唯一特殊的就是最后面的14是arp和IPV6，所以是匹配失败直接发的

```
(syenv) njucs@njucs-VirtualBox:~/switchyard/lab_7$ swyard -t firewalltests.py firewall.py
17:12:38 2020/06/10     INFO Starting test scenario firewalltests.py
Match rule index 3
Match rule index 4
Match rule index 5
Match rule index 6
Match rule index 9
Match rule index 10
Match rule index 7
Match rule index 8
Match rule index 13
Match rule index 13
Match rule index 13
Match rule index 1
Match rule index 1
Match rule index 2
Match rule index 2
Match rule index 14
Match rule index 14
No match
No match

Results for test scenario Firewall tests: 35 passed, 0 failed, 0 pending
```

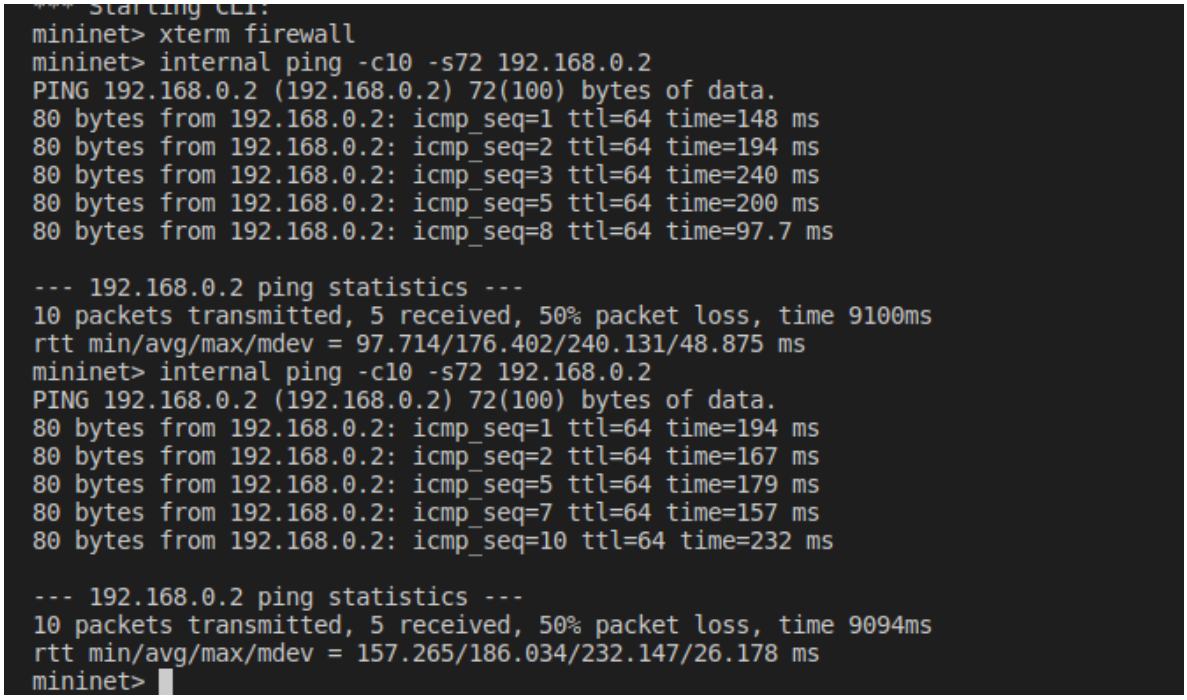
```
Passed:
1   Packet arriving on eth0 should be permitted since it matches
    rule 3.
2   Packet forwarded out eth1; permitted since it matches rule
    3.
3   Packet arriving on eth1 should be permitted since it matches
    rule 4.
4   Packet forwarded out eth0; permitted since it matches rule
    4.
5   Packet arriving on eth0 should be permitted since it matches
    rule 5.
6   Packet forwarded out eth1; permitted since it matches rule
    5.
7   Packet arriving on eth1 should be permitted since it matches
    rule 6.
8   Packet forwarded out eth0; permitted since it matches rule
    6.
9   Packet arriving on eth0 should be permitted since it matches
    rule 9.
10  Packet forwarded out eth1; permitted since it matches rule
    9.
11  Packet arriving on eth1 should be permitted since it matches
    rule 10.
12  Packet forwarded out eth0; permitted since it matches rule
    10.
13  Timeout for 1s
14  Timeout for 1s
15  Timeout for 1s
16  Timeout for 1s
17  Packet arriving on eth0 should be permitted since it matches
    rule 7.
18  Packet forwarded out eth1; permitted since it matches rule
    7.
19  Packet arriving on eth1 should be permitted since it matches
    rule 8.
20  Packet forwarded out eth0; permitted since it matches rule
    8.
21  Packet arriving on eth0 should be permitted since it matches
    rule 13.
22  Packet forwarded out eth1; permitted since it matches rule
    13.
23  Packet arriving on eth1 should be permitted since it matches
    rule 13.
24  Packet forwarded out eth0; permitted since it matches rule
    13.
25  Packet arriving on eth0 should be blocked due to rate limit.
26  Packet arriving on eth0 should be blocked since it matches
    rule 1.
27  Packet arriving on eth1 should be blocked since it matches
    rule 1.
28  Packet arriving on eth0 should be blocked since it matches
    rule 2.
```

mininet

ratelimit

限速设置的是150，同时由于规则的设置的两个any，也就是双向共享规则和令牌桶

会达到限速的边界导致丢失


*** Starting CLI:
mininet> xterm firewall
mininet> internal ping -c10 -s72 192.168.0.2
PING 192.168.0.2 (192.168.0.2) 72(100) bytes of data.
80 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time=148 ms
80 bytes from 192.168.0.2: icmp_seq=2 ttl=64 time=194 ms
80 bytes from 192.168.0.2: icmp_seq=3 ttl=64 time=240 ms
80 bytes from 192.168.0.2: icmp_seq=5 ttl=64 time=200 ms
80 bytes from 192.168.0.2: icmp_seq=8 ttl=64 time=97.7 ms
--- 192.168.0.2 ping statistics ---
10 packets transmitted, 5 received, 50% packet loss, time 9100ms
rtt min/avg/max/mdev = 97.714/176.402/240.131/48.875 ms
mininet> internal ping -c10 -s72 192.168.0.2
PING 192.168.0.2 (192.168.0.2) 72(100) bytes of data.
80 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time=194 ms
80 bytes from 192.168.0.2: icmp_seq=2 ttl=64 time=167 ms
80 bytes from 192.168.0.2: icmp_seq=5 ttl=64 time=179 ms
80 bytes from 192.168.0.2: icmp_seq=7 ttl=64 time=157 ms
80 bytes from 192.168.0.2: icmp_seq=10 ttl=64 time=232 ms
--- 192.168.0.2 ping statistics ---
10 packets transmitted, 5 received, 50% packet loss, time 9094ms
rtt min/avg/max/mdev = 157.265/186.034/232.147/26.178 ms
mininet>

较多的测试

```
mininet> xterm firewall
mininet> internal ping -s72 192.168.0.2
PING 192.168.0.2 (192.168.0.2) 72(100) bytes of data.
80 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time=177 ms
80 bytes from 192.168.0.2: icmp_seq=2 ttl=64 time=221 ms
80 bytes from 192.168.0.2: icmp_seq=5 ttl=64 time=233 ms
80 bytes from 192.168.0.2: icmp_seq=8 ttl=64 time=165 ms
80 bytes from 192.168.0.2: icmp_seq=11 ttl=64 time=181 ms
80 bytes from 192.168.0.2: icmp_seq=13 ttl=64 time=223 ms
80 bytes from 192.168.0.2: icmp_seq=16 ttl=64 time=173 ms
80 bytes from 192.168.0.2: icmp_seq=19 ttl=64 time=227 ms
80 bytes from 192.168.0.2: icmp_seq=22 ttl=64 time=167 ms
80 bytes from 192.168.0.2: icmp_seq=25 ttl=64 time=182 ms
80 bytes from 192.168.0.2: icmp_seq=27 ttl=64 time=152 ms
80 bytes from 192.168.0.2: icmp_seq=29 ttl=64 time=224 ms
80 bytes from 192.168.0.2: icmp_seq=32 ttl=64 time=231 ms
80 bytes from 192.168.0.2: icmp_seq=35 ttl=64 time=185 ms
80 bytes from 192.168.0.2: icmp_seq=38 ttl=64 time=201 ms
80 bytes from 192.168.0.2: icmp_seq=40 ttl=64 time=187 ms
80 bytes from 192.168.0.2: icmp_seq=43 ttl=64 time=184 ms
80 bytes from 192.168.0.2: icmp_seq=45 ttl=64 time=191 ms
80 bytes from 192.168.0.2: icmp_seq=48 ttl=64 time=230 ms
80 bytes from 192.168.0.2: icmp_seq=51 ttl=64 time=274 ms
80 bytes from 192.168.0.2: icmp_seq=54 ttl=64 time=200 ms
80 bytes from 192.168.0.2: icmp_seq=57 ttl=64 time=222 ms
80 bytes from 192.168.0.2: icmp_seq=60 ttl=64 time=159 ms
80 bytes from 192.168.0.2: icmp_seq=63 ttl=64 time=172 ms
80 bytes from 192.168.0.2: icmp_seq=65 ttl=64 time=166 ms
80 bytes from 192.168.0.2: icmp_seq=68 ttl=64 time=140 ms
80 bytes from 192.168.0.2: icmp_seq=70 ttl=64 time=141 ms
80 bytes from 192.168.0.2: icmp_seq=73 ttl=64 time=269 ms
80 bytes from 192.168.0.2: icmp_seq=75 ttl=64 time=137 ms
80 bytes from 192.168.0.2: icmp_seq=80 ttl=64 time=197 ms
80 bytes from 192.168.0.2: icmp_seq=83 ttl=64 time=190 ms
80 bytes from 192.168.0.2: icmp_seq=86 ttl=64 time=204 ms
80 bytes from 192.168.0.2: icmp_seq=89 ttl=64 time=195 ms
80 bytes from 192.168.0.2: icmp_seq=92 ttl=64 time=179 ms
80 bytes from 192.168.0.2: icmp_seq=95 ttl=64 time=209 ms
80 bytes from 192.168.0.2: icmp_seq=97 ttl=64 time=201 ms
80 bytes from 192.168.0.2: icmp_seq=100 ttl=64 time=191 ms
80 bytes from 192.168.0.2: icmp_seq=103 ttl=64 time=193 ms
80 bytes from 192.168.0.2: icmp_seq=106 ttl=64 time=194 ms
^Z
[1]+  Stopped                 sudo python start_mininet.py
```

按照速度的计算12500对应12.5KB/s

最后的瞬时速度和平均速度均符合要求

```
mininet> external ./www/start_webserver.sh
100+0 records in
100+0 records out
102400 bytes (102 kB, 100 KiB) copied, 0.0005799 s, 177 MB/s
mininet> internal wget http://192.168.0.2/bigfile -O /dev/null
--2020-06-10 17:22:45--  http://192.168.0.2/bigfile
Connecting to 192.168.0.2:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 102400 (100K) [application/octet-stream]
Saving to: '/dev/null'

/dev/null           100%[===================>] 100.00K  10.3KB/s    in 8.9s

2020-06-10 17:22:54 (11.2 KB/s) - '/dev/null' saved [102400/102400]

mininet>
```

impair

设置不丢弃的时候

```
mininet> internal wget http://192.168.0.2:8000/bigfile -O /dev/null
--2020-06-10 17:27:45--  http://192.168.0.2:8000/bigfile
Connecting to 192.168.0.2:8000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 102400 (100K) [application/octet-stream]
Saving to: '/dev/null'

/dev/null           100%[===================>] 100.00K   157KB/s    in 0.6s

2020-06-10 17:27:46 (157 KB/s) - '/dev/null' saved [102400/102400]

mininet>
```

设置丢弃1%的时候

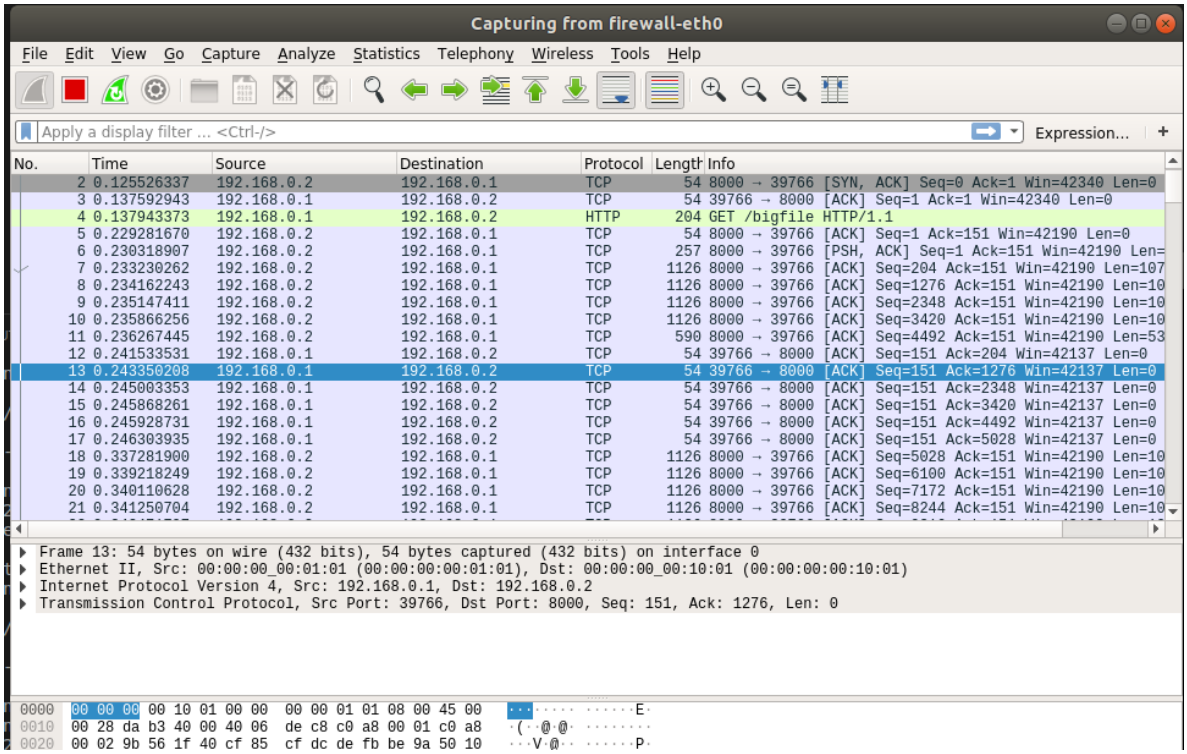
明显看到传输速度的下降

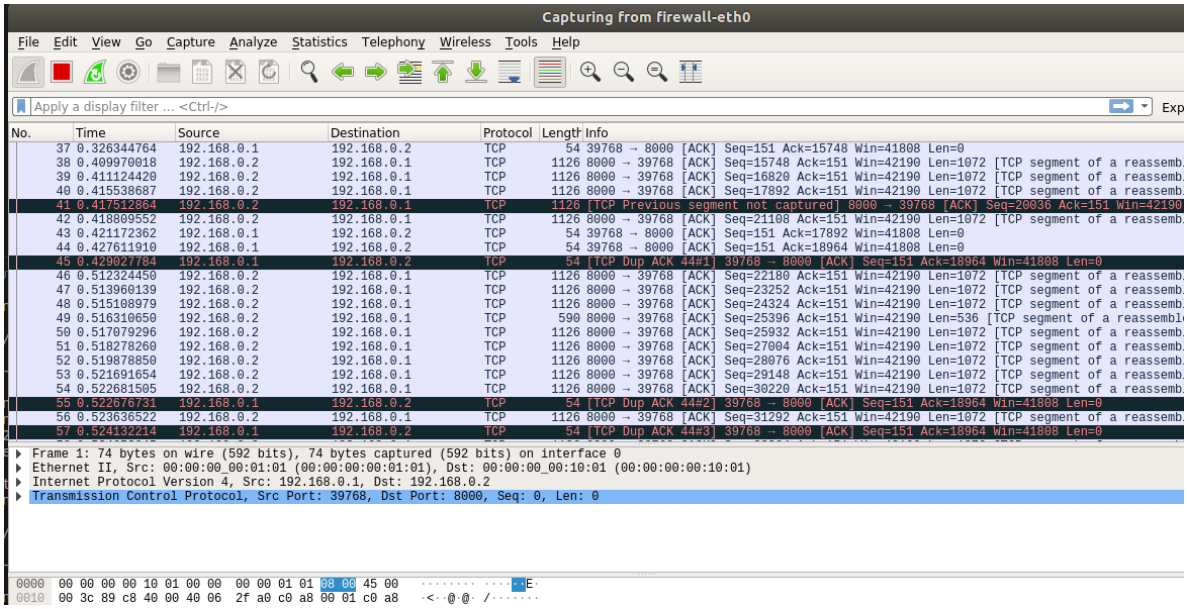
```
mininet> external ./www/start_webserver.sh 8000
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
192.168.0.1 - - [10/Jun/2020 17:22:45] "GET /bigfile HTTP/1.1" 200 -
100+0 records in
100+0 records out
102400 bytes (102 kB, 100 KiB) copied, 0.000544023 s, 188 MB/s
mininet> internal wget http://192.168.0.2:8000/bigfile -O /dev/null
--2020-06-10 17:26:43--  http://192.168.0.2:8000/bigfile
Connecting to 192.168.0.2:8000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 102400 (100K) [application/octet-stream]
Saving to: '/dev/null'

/dev/null           100%[===================>] 100.00K  95.8KB/s    in 1.0s

2020-06-10 17:26:44 (95.8 KB/s) - '/dev/null' saved [102400/102400]

mininet>
```

以firewall为抓包节点，下面的图分别是没有impair和有impair的

可以看到在2图中出现了大量的深色节点Dup Ack。含义见下图3，4的解释。

参考连接https://networkengineering.stackexchange.com/questions/38471/what-does-tcp-dup-ack-mean


Capturing from firewall-eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
No. Time Source Destination Protocol Length Info
2 0.125526337 192.168.0.2 192.168.0.1 TCP 54 8000 → 39766 [SYN, ACK] Seq=0 Ack=1 Win=42340 Len=0
3 0.137592943 192.168.0.1 192.168.0.2 TCP 54 39766 → 8000 [ACK] Seq=1 Ack=1 Win=42340 Len=0
4 0.137943373 192.168.0.1 192.168.0.2 HTTP 204 GET /bigfile HTTP/1.1
5 0.229281670 192.168.0.2 192.168.0.1 TCP 54 8000 → 39766 [ACK] Seq=1 Ack=151 Win=42190 Len=0
6 0.230318907 192.168.0.2 192.168.0.1 TCP 257 8000 → 39766 [PSH, ACK] Seq=1 Ack=151 Win=42190 Len=
7 0.233230262 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39766 [ACK] Seq=204 Ack=151 Win=42190 Len=107
8 0.234162243 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39766 [ACK] Seq=1276 Ack=151 Win=42190 Len=10
9 0.235147411 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39766 [ACK] Seq=2348 Ack=151 Win=42190 Len=10
10 0.235866256 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39766 [ACK] Seq=3420 Ack=151 Win=42190 Len=10
11 0.236267445 192.168.0.2 192.168.0.1 TCP 590 8000 → 39766 [ACK] Seq=4492 Ack=151 Win=42190 Len=53
12 0.241533531 192.168.0.1 192.168.0.2 TCP 54 39766 → 8000 [ACK] Seq=151 Ack=204 Win=42137 Len=0
13 0.243350208 192.168.0.1 192.168.0.2 TCP 54 39766 → 8000 [ACK] Seq=151 Ack=1276 Win=42137 Len=0
14 0.245003353 192.168.0.1 192.168.0.2 TCP 54 39766 → 8000 [ACK] Seq=151 Ack=2348 Win=42137 Len=0
15 0.245868261 192.168.0.1 192.168.0.2 TCP 54 39766 → 8000 [ACK] Seq=151 Ack=3420 Win=42137 Len=0
16 0.245928731 192.168.0.1 192.168.0.2 TCP 54 39766 → 8000 [ACK] Seq=151 Ack=4492 Win=42137 Len=0
17 0.246303935 192.168.0.1 192.168.0.2 TCP 54 39766 → 8000 [ACK] Seq=151 Ack=5028 Win=42137 Len=0
18 0.337281900 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39766 [ACK] Seq=5028 Ack=151 Win=42190 Len=10
19 0.339218249 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39766 [ACK] Seq=6100 Ack=151 Win=42190 Len=10
20 0.340110628 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39766 [ACK] Seq=7172 Ack=151 Win=42190 Len=10
21 0.341250704 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39766 [ACK] Seq=8244 Ack=151 Win=42190 Len=10
Frame 13: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
Ethernet II, Src: 00:00:00_00:01:01 (00:00:00:00:01:01), Dst: 00:00:00_00:10:01 (00:00:00:00:10:01)
Internet Protocol Version 4, Src: 192.168.0.1, Dst: 192.168.0.2
Transmission Control Protocol, Src Port: 39766, Dst Port: 8000, Seq: 151, Ack: 1276, Len: 0
0000 00 00 00 00 10 01 00 00 00 00 01 01 08 00 45 00
0010 00 28 da b3 40 00 40 06 de c8 c0 a8 00 01 c0 a8
0020 00 02 9b 56 1f 40 cf 85 cf dc de fb be 9a 50 10


Capturing from firewall-eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
No. Time Source Destination Protocol Length Info
37 0.326344764 192.168.0.1 192.168.0.2 TCP 54 39768 → 8000 [ACK] Seq=151 Ack=15748 Win=41808 Len=0
38 0.409970018 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39768 [ACK] Seq=15748 Ack=151 Win=42190 Len=1072 [TCP segment of a reassemb
39 0.411124420 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39768 [ACK] Seq=16820 Ack=151 Win=42190 Len=1072 [TCP segment of a reassemb
40 0.415538687 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39768 [ACK] Seq=17892 Ack=151 Win=42190 Len=1072 [TCP segment of a reassemb
41 0.417512864 192.168.0.2 192.168.0.1 TCP 1126 [TCP Previous segment not captured] 8000 → 39768 [ACK] Seq=20036 Ack=151 Win=42190
42 0.418809552 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39768 [ACK] Seq=21108 Ack=151 Win=42190 Len=1072 [TCP segment of a reassemb
43 0.421172362 192.168.0.1 192.168.0.2 TCP 54 39768 → 8000 [ACK] Seq=151 Ack=17892 Win=41808 Len=0
44 0.427611910 192.168.0.1 192.168.0.2 TCP 54 39768 → 8000 [ACK] Seq=151 Ack=18964 Win=41808 Len=0
45 0.429027784 192.168.0.1 192.168.0.2 TCP 54 [TCP Dup ACK 44#1] 39768 → 8000 [ACK] Seq=151 Ack=18964 Win=41808 Len=0
46 0.512324450 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39768 [ACK] Seq=22180 Ack=151 Win=42190 Len=1072 [TCP segment of a reassemb
47 0.513960139 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39768 [ACK] Seq=23252 Ack=151 Win=42190 Len=1072 [TCP segment of a reassemb
48 0.515108979 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39768 [ACK] Seq=24324 Ack=151 Win=42190 Len=1072 [TCP segment of a reassemb
49 0.516310650 192.168.0.2 192.168.0.1 TCP 590 8000 → 39768 [ACK] Seq=25396 Ack=151 Win=42190 Len=536 [TCP segment of a reassembl
50 0.517079296 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39768 [ACK] Seq=25932 Ack=151 Win=42190 Len=1072 [TCP segment of a reassemb
51 0.518278260 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39768 [ACK] Seq=27004 Ack=151 Win=42190 Len=1072 [TCP segment of a reassemb
52 0.519878850 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39768 [ACK] Seq=28076 Ack=151 Win=42190 Len=1072 [TCP segment of a reassemb
53 0.521691654 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39768 [ACK] Seq=29148 Ack=151 Win=42190 Len=1072 [TCP segment of a reassemb
54 0.522681505 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39768 [ACK] Seq=30220 Ack=151 Win=42190 Len=1072 [TCP segment of a reassemb
55 0.522676731 192.168.0.1 192.168.0.2 TCP 54 [TCP Dup ACK 44#2] 39768 → 8000 [ACK] Seq=151 Ack=18964 Win=41808 Len=0
56 0.523636522 192.168.0.2 192.168.0.1 TCP 1126 8000 → 39768 [ACK] Seq=31292 Ack=151 Win=42190 Len=1072 [TCP segment of a reassemb
57 0.524132214 192.168.0.1 192.168.0.2 TCP 54 [TCP Dup ACK 44#3] 39768 → 8000 [ACK] Seq=151 Ack=18964 Win=41808 Len=0
Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
Ethernet II, Src: 00:00:00_00:01:01 (00:00:00:00:01:01), Dst: 00:00:00_00:10:01 (00:00:00:00:10:01)
Internet Protocol Version 4, Src: 192.168.0.1, Dst: 192.168.0.2
Transmission Control Protocol, Src Port: 39768, Dst Port: 8000, Seq: 0, Len: 0
0000 00 00 00 00 10 01 00 00 00 00 01 01 08 00 45 00
0010 00 3c 89 c8 40 00 40 06 2f a0 c0 a8 00 01 c0 a8

What does TCP DUP ACK mean?

Asked 3 years, 4 months ago Active 3 years, 4 months ago Viewed 80k times

10

In Wireshark, I see packets sent from the receiver to the sender. What does it mean? Does it imply packet loss? `TCP duplicate ACK`

Thank you

cisco tcp packet-loss transport-protocol

5

share improve this question follow

asked Jan 29 '17 at 23:54
John T
113 ●1 ●1 ●4

add a comment

1 Answer

Active Oldest Votes

11

There can be several things going on - the most common would be the use of TCP Fast Retransmission which is a mechanism by which a receiver can indicate that it has seen a gap in the received sequence numbers that implies the loss of one or more packets in transit. The repeated acknowledgements at the last known value before the gap signal which packets the sender should retransmit. This can occur without waiting for the acknowledgement timeout for the lost packet to hit on the transmitter - which, as the name implies, means recovering a lot faster.

It's also possible that the same symptom of gaps in sequence numbers might be seen in a situation where packets are being delivered out of order. As above, if the receiver sees (for example) a segment with sequence #5 followed by another with #7 before seeing sequence #6 then it might try to begin to trigger a fast retransmit. Upon seeing #6 arrive, though, it would stop sending the duplicate acknowledgements.

A less common cause would be certain media problems where certain packets might end up being seen more than once. If this is the case, however, you're likely to see other problems on the link (...including other packets showing as dupes in Wireshark).

So - if you're seeing a few random duplicate ACK's but no (or few) actual retransmissions then it's likely packets arriving out of order. If you're seeing a lot more duplicate ACK's followed by actual retransmission then some amount of packet loss is taking place. Both situations are, unfortunately, entirely possible on the global Internet. If you're seeing other kinds of duplicate packets as CRC issues and generally slow performance then it might make sense to look at link issues on your own network.

TCP DUP ACK 是什么意思？

问了 3 年，4 个月前　活跃 3 年，4 个月前　查看 80k 次

10

在Wireshark中，我看到从接收方发送到发送方的数据包。这是什么意思？这是否意味着数据包丢失？`TCP duplicate ACK`

谢谢

思科　Tcp　数据包丢失　传输协议

5

共享　改进此问题　遵循

问1 月 29 '17 在 23：54
约翰T
113 ●1 ●1 ●4

添加注释

1 答案

积极　古老　票

11

可能有几个事情正在发生 - 最常见的是使用TCP快速转播，这是一种机制，接收器可以指示它已经看到了接收的序列号的差距，这意味着在传输中丢失一个或多个数据包。在间隙信号之前的最后一个已知值重复确认，发送方应重新传输数据包。这可能发生，而无需等待丢失的数据包在发射器上命中的确认超时 - 顾名思义，这意味着恢复速度更快。

在数据包无序交付的情况下，也可能看到序列号差距的相同症状。如上所述，如果接收方在看到序列#6之前看到（例如）序列#5的段，然后是另一个带#7的段，则它可能会尝试开始触发快速重新传输。但是，当看到#6到达时，它将停止发送重复的确认。

不太常见的原因是某些媒体问题，其中某些数据包可能最终被多次看到。但是，如果是这种情况，您可能会在链接上看到其他问题（...包括其他在Wireshark中显示为欺骗的数据包）。

所以- 如果你看到一些随机重复的ACK的，但没有（或很少）实际转播，那么它很可能数据包到达顺序。如果您看到更多重复的 ACK，然后是实际重新传输，则发生一些数据包丢失。不幸的是，这两种情况在全球因特网上是完全可能的。如果您看到其他类型的重复数据包作为 CRC 问题，并且性能通常很慢，那么查看您自己的网络上的链路问题可能有意义。

共享　改进此答案　遵循

回答1 月 30 '17 在 5：13
恩克斯克斯
5,324 ●1 ●14 ●19