Computer Networking-lab1-Repot

课程名称：**计算机网络** 任课教师：田臣/李文中

学院	Dept. of Computer Science and Technology	专业（方向）	CS
学号	181830044	姓名	董宸郅
Email	pdon#foxmail.com	开始/完成日期	3.19/3.25

实验名称：Leaning Switch

实验目的

- 深入理解***switch***的工作原理
- 理解并实现具有"**学习能力**"的switch
- 进一步熟悉switchyard的代码接口和整体框架
- 加强在实验环境下调试代码的能力

实验内容

理论知识

Switch概念解析

Switch基于mac地址识别，能完成封装转发数据功能的设备。可以记录一定数量的mac地址，放在**内部地址表**中，通过在数据帧的始发者和接收者之间建立临时的交换路径，使数据从源地址到达目的地址。（与之相比，Hub是一种共享设备，本身不能识别目的地址。采用广播的形式传输数据，即向所有端口传送数据。）

ARP

地址解析协议(Address Resolution Protocol)，是根据IP地址获取物理地址的一个TCP/IP协议。主机发送信息时将**包含目标IP地址的ARP请求广播到局域网络上的所有主机**，并接收返回消息，以此确定目标的物理地址；收到返回消息后将该IP地址和物理地址**存入本机ARP缓存中并保留一定时间**，下次请求时直接查询ARP缓存以节约资源。

实验步骤（含结果与关键代码）

注：

①对于之后几个Task中与Task 2基本一致的代码不再专门说明展示，将只说明其特有的关键代码

②Deploy阶段的mininet拓扑结构即为默认提供的结构：中央的switch分别连接server1、server2、client。

Task 2: Basic Switch

Coding

```
# add an empty heapq to store forwarding rules
# each element of tab should be a tuple like: (traffic, host, intf) -->traffic means the number of packets related
# when a heapq consists of tuples, it is organized based on the tuples' first elements
tab = []
```

```
# recording section
# src already recorded
if packet[0].src in [rule[1] for rule in tab]:
    # find the corresbonding iterator for the current host
    cur = iter(tab)
    for rule in tab:
        if rule[1] == packet[0].src:
            cur = rule
            break
    # first delete, then add a new one
    new_rule = (cur[0],cur[1],input_port)
    tab.remove(cur)
    tab.append(new_rule)
    heapq.heapify(tab)
# src not recorded yet
else:
    # forwarding table is full
    if len(tab) == max_rules:
        heapq.heappop(tab)
    tab.append((0, packet[0].src, input_port))
    heapq.heapify(tab)
```

```
# forwarding section
# dst already recorded
if packet[0].dst in [rule[1] for rule in tab]:
    # find the corresbonding iterator for the current host
    cur = iter(tab)
    for rule in tab:
        if rule[1] == packet[0].dst:
            cur = rule
            break
    log_debug ("Flooding packet {} to {}".format(packet, cur[2]))
    net.send_packet(cur[2], packet)
    new_rule = (cur[0]+1,cur[1],cur[2])
    tab.remove(cur)
    tab.append(new_rule)
    heapq.heapify(tab)
# dst not recorded yet
else:
    # do nothing if dst is the switch itself
    if packet[0].dst not in mymacs:
        for intf in my_interfaces:
            if input_port != intf.name:
                log_debug ("Flooding packet {} to {}".format(packet, intf.name))
                net.send_packet(intf.name, packet)
```

Deploying

server1:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	30:00:00:00:00…	Broadcast	ARP	42	Who has 192.168.100.1? Tell 192.168.100.3
2	0.116833379	Private_00:00:…	30:00:00:00:00…	ARP	42	192.168.100.1 is at 10:00:00:00:00:01
3	0.669877596	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x0bec, seq=1/256, ttl=64 (reply in 4)
4	0.793030851	192.168.100.1	192.168.100.3	ICMP	98	Echo (ping) reply id=0x0bec, seq=1/256, ttl=64 (request in 3)
5	0.996696301	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x0bec, seq=2/512, ttl=64 (reply in 6)
6	1.096822978	192.168.100.1	192.168.100.3	ICMP	98	Echo (ping) reply id=0x0bec, seq=2/512, ttl=64 (request in 5)
7	5.934673661	Private_00:00:…	30:00:00:00:00…	ARP	42	Who has 192.168.100.3? Tell 192.168.100.1
8	6.396574253	30:00:00:00:00…	Private_00:00:…	ARP	42	192.168.100.3 is at 30:00:00:00:00:01

server2:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	30:00:00:00:00…	Broadcast	ARP	42	Who has 192.168.100.1? Tell 192.168.100.3

在client的cli中输入以下命令，以向server1发出两次请求：

```
ping -c 2 192.168.100.1
```

根据上述截图，由于server2有且仅有一次收到ARP包询问是否拥有server1的IP地址，可知第一次请求时switch不知道server1的mac地址，第二次则已完成记录，直接发给server1，验证了basic learning的逻辑。

Task 3: Timeouts

Coding

```python
# add an empty dict intended for host-(intf,last_time) pairs
tab={}
```

```python
# when new packet comes, record its information or update recorded information
# since 'dictionary' is used and its keys(hosts' mac addresses) are all unique, the following line can do it all
tab[packet[0].src] = {'intf': input_port, 'last': time.time()}
# delete timeout entries in the forwarding table
for host in list(tab):
    if time.time()-tab[host]['last'] > 10:
        del tab[host]
```

Testing

```
Results for test scenario switch tests: 9 passed, 0 failed, 0 pending


Passed:
1   An Ethernet frame with a broadcast destination address
    should arrive on eth1
2   The Ethernet frame with a broadcast destination address
    should be forwarded out ports eth0 and eth2
3   An Ethernet frame from 20:00:00:00:00:01 to
    30:00:00:00:00:02 should arrive on eth0
4   Ethernet frame destined for 30:00:00:00:00:02 should arrive
    on eth1 after self-learning
5   Timeout for 20s
6   An Ethernet frame from 20:00:00:00:00:01 to
    30:00:00:00:00:02 should arrive on eth0
7   Ethernet frame destined for 30:00:00:00:00:02 should be
    flooded out eth1 and eth2
8   An Ethernet frame should arrive on eth2 with destination
    address the same as eth2's MAC address
9   The hub should not do anything in response to a frame
    arriving with a destination address referring to the hub
    itself.


All tests passed!
```

Deploying

server1:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x0d4f, seq=1/256, ttl=64 (reply in 2)
2	0.109098147	192.168.100.1	192.168.100.3	ICMP	98	Echo (ping) reply id=0x0d4f, seq=1/256, ttl=64 (request in 1)
3	0.939345013	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x0d4f, seq=2/512, ttl=64 (reply in 4)
4	1.039499469	192.168.100.1	192.168.100.3	ICMP	98	Echo (ping) reply id=0x0d4f, seq=2/512, ttl=64 (request in 3)
5	5.062906588	30:00:00:00:00:01	Private_00:00:01	ARP	42	Who has 192.168.100.1? Tell 192.168.100.3
6	5.134184208	Private_00:00:01	30:00:00:00:00:01	ARP	42	Who has 192.168.100.3? Tell 192.168.100.1
7	5.163162672	Private_00:00:01	30:00:00:00:00:01	ARP	42	192.168.100.1 is at 10:00:00:00:00:01
8	5.478011697	30:00:00:00:00:01	Private_00:00:01	ARP	42	192.168.100.3 is at 30:00:00:00:00:01
9	53.643615899	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x0d53, seq=1/256, ttl=64 (reply in 10)
10	53.745554743	192.168.100.1	192.168.100.3	ICMP	98	Echo (ping) reply id=0x0d53, seq=1/256, ttl=64 (request in 9)
11	54.604462003	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x0d53, seq=2/512, ttl=64 (reply in 12)
12	54.705857091	192.168.100.1	192.168.100.3	ICMP	98	Echo (ping) reply id=0x0d53, seq=2/512, ttl=64 (request in 11)
13	58.804501298	30:00:00:00:00:01	Private_00:00:01	ARP	42	Who has 192.168.100.1? Tell 192.168.100.3
14	58.892697087	Private_00:00:01	30:00:00:00:00:01	ARP	42	Who has 192.168.100.3? Tell 192.168.100.1
15	58.904925732	Private_00:00:01	30:00:00:00:00:01	ARP	42	192.168.100.1 is at 10:00:00:00:00:01
16	59.222410076	30:00:00:00:00:01	Private_00:00:01	ARP	42	192.168.100.3 is at 30:00:00:00:00:01

server2:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x0d4f, seq=1/256, ttl=64 (no response found!)
2	53.643611694	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x0d53, seq=1/256, ttl=64 (no response found!)

在client的cli中输入以下命令，以向server1发出四次请求（分两次发送，且**间隔时间大于10s**）：

```
ping -c 2 192.168.100.1
# more than 10s later
ping -c 2 192.168.100.1
```

从上述截图看出，server2收到两次client对server1的请求。

第一次请求时switch不知道server1的地址，进行记录后第二次请求不需要再次广播。

第三次请求时，由于已经过了规定的10s，由**timeout**机制，记录被清除，所以再次广播，到了第四次，就和第二次一样，直接发给server1。

从而，验证了timeout逻辑。

Task 4: Least Recently Used

Coding

```
# add an empty heapq to store forwarding rules
# each element of tab should be a tuple like: (last_time, host, intf)
# when a heapq consists of tuples, it is organized based on the tuples' first elements
# therefore the least recently used rule(its 'last_time' being the 'least' when treated as a number) is always at the front, which is what we expect
tab = []
```

```
# recording section
# src already recorded
if packet[0].src in [rule[1] for rule in tab]:
    # find the corresbonding iterator for the current host
    cur = iter(tab)
    for rule in tab:
        if rule[1] == packet[0].src:
            cur = rule
            break
    # adjust the heapq
    # first delete, then add a new one since it's a tuple
    tab.remove(cur)
# src not recorded yet
else:
    # forwarding table is full
    if len(tab) == max_rules:
        heapq.heappop(tab)
tab.append((time.time(), packet[0].src, input_port))
heapq.heapify(tab)
```

```
# forwarding section
# dst already recorded
if packet[0].dst in [rule[1] for rule in tab]:
    # find the corresbonding iterator for the current host
    cur = iter(tab)
    for rule in tab:
        if rule[1] == packet[0].dst:
            cur = rule
            break
    log_debug ("Flooding packet {} to {}".format(packet, cur[2]))
    net.send_packet(cur[2], packet)
    # adjust the heapq
    # first delete, then add a new one since it's a tuple
    new_rule = (time.time(), cur[1], cur[2])
    tab.remove(cur)
    tab.append(new_rule)
    heapq.heapify(tab)
# dst not recorded yet
else:
    # do nothing if dst is the switch itself
    if packet[0].dst not in mymacs:
        for intf in my_interfaces:
            if input_port != intf.name:
                log_debug ("Flooding packet {} to {}".format(packet, intf.name))
                net.send_packet(intf.name, packet)

```

Testing

```
Passed:
1   An Ethernet frame with a broadcast destination address
    should arrive on eth1
2   The Ethernet frame with a broadcast destination address
    should be forwarded out ports eth0, eth2, eth3 and eth4
3   An Ethernet frame from 20:00:00:00:00:01 to
    30:00:00:00:00:02 should arrive on eth0
4   Ethernet frame destined for 30:00:00:00:00:02 should arrive
    on eth1 after self-learning
5   An Ethernet frame from 20:00:00:00:00:03 to
    30:00:00:00:00:02 should arrive on eth2
6   Ethernet frame destined for 30:00:00:00:00:02 should arrive
    on eth1 after self-learning
7   An Ethernet frame from 30:00:00:00:00:04 to
    20:00:00:00:00:01 should arrive on eth3
8   Ethernet frame destined to 20:00:00:00:00:01 should arrive
    on eth0 after self-learning
9   An Ethernet frame from 20:00:00:00:00:01 to
    30:00:00:00:00:04 should arrive on eth0
10  Ethernet frame destined to 20:00:00:00:00:01 should arrive
    on eth3 after self-learning
11  An Ethernet frame from 40:00:00:00:00:05 to
    20:00:00:00:00:01 should arrive on eth4
12  Ethernet frame destined to 20:00:00:00:00:01 should arrive
    on eth0 after self-learning
13  An Ethernet frame from 30:00:00:00:00:05 to
    20:00:00:00:00:01 should arrive on eth4
14  Ethernet frame destined to 20:00:00:00:00:01 should arrive
    on eth0 after self-learning
15  An Ethernet frame from 20:00:00:00:00:05 to
    30:00:00:00:00:02 should arrive on eth4
16  Ethernet frame destined to 30:00:00:00:00:02 should be
    flooded to eth0, eth1, eth2 and eth3
17  An Ethernet frame should arrive on eth2 with destination
    address the same as eth2's MAC address
18  The hub should not do anything in response to a frame
    arriving with a destination address referring to the hub
    itself.


All tests passed!
```

Deploying

server1:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x0e1d, seq=1/256, ttl=64 (reply in 2)
2	0.103000481	192.168.100.1	192.168.100.3	ICMP	98	Echo (ping) reply id=0x0e1d, seq=1/256, ttl=64 (request in 1)
3	0.941412336	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x0e1d, seq=2/512, ttl=64 (reply in 4)
4	1.041711176	192.168.100.1	192.168.100.3	ICMP	98	Echo (ping) reply id=0x0e1d, seq=2/512, ttl=64 (request in 3)
5	5.082535485	30:00:00:00:00:01	Private_00:00:01	ARP	42	Who has 192.168.100.1? Tell 192.168.100.3
6	5.182721656	Private_00:00:01	30:00:00:00:00:01	ARP	42	192.168.100.1 is at 10:00:00:00:00:01
7	5.192113306	Private_00:00:01	30:00:00:00:00:01	ARP	42	Who has 192.168.100.3? Tell 192.168.100.1
8	5.602972490	30:00:00:00:00:01	Private_00:00:01	ARP	42	192.168.100.3 is at 30:00:00:00:00:01
9	9.099949357	30:00:00:00:00:01	Broadcast	ARP	42	Who has 192.168.100.2? Tell 192.168.100.3
10	29.928844407	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x0e20, seq=1/256, ttl=64 (reply in 11)
11	30.028946144	192.168.100.1	192.168.100.3	ICMP	98	Echo (ping) reply id=0x0e20, seq=1/256, ttl=64 (request in 10)
12	30.881218559	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x0e20, seq=2/512, ttl=64 (reply in 13)
13	30.982214852	192.168.100.1	192.168.100.3	ICMP	98	Echo (ping) reply id=0x0e20, seq=2/512, ttl=64 (request in 12)
14	35.113281397	30:00:00:00:00:01	Private_00:00:01	ARP	42	Who has 192.168.100.1? Tell 192.168.100.3
15	35.215979139	Private_00:00:01	30:00:00:00:00:01	ARP	42	192.168.100.1 is at 10:00:00:00:00:01

server2:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x0e1d, seq=1/256, ttl=64 (no response found!)
2	9.099949461	30:00:00:00:00:01	Broadcast	ARP	42	Who has 192.168.100.2? Tell 192.168.100.3
3	9.205943341	20:00:00:00:00:01	30:00:00:00:00:01	ARP	42	192.168.100.2 is at 20:00:00:00:00:01
4	9.632413597	192.168.100.3	192.168.100.2	ICMP	98	Echo (ping) request id=0x0e1e, seq=1/256, ttl=64 (reply in 5)
5	9.734988380	192.168.100.2	192.168.100.3	ICMP	98	Echo (ping) reply id=0x0e1e, seq=1/256, ttl=64 (request in 4)
6	10.070383175	192.168.100.3	192.168.100.2	ICMP	98	Echo (ping) request id=0x0e1e, seq=2/512, ttl=64 (reply in 7)
7	10.170648244	192.168.100.2	192.168.100.3	ICMP	98	Echo (ping) reply id=0x0e1e, seq=2/512, ttl=64 (request in 6)
8	14.916069766	20:00:00:00:00:01	30:00:00:00:00:01	ARP	42	Who has 192.168.100.3? Tell 192.168.100.2
9	15.316532769	30:00:00:00:00:01	20:00:00:00:00:01	ARP	42	192.168.100.3 is at 30:00:00:00:00:01
10	29.928844717	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x0e20, seq=1/256, ttl=64 (no response found!)

在client的cli中输入以下命令：

```
ping -c 2 192.168.100.1
ping -c 2 192.168.100.2
ping -c 2 192.168.100.2
ping -c 2 192.168.100.1
```

首先，设置**最多容纳的rule数目为2**。

在最开始的client对server1的两次请求中，结果与之前task一致。

之后，进行若干次client对server2的请求，使server2的以达到清除有关server1对应端口信息的目的。

最后，再次进行client对server1的请求，server2的wireshark记录显示其再次被询问，从而验证了**LRU**逻辑。

Task 5: Least Traffic Volume

Coding

```
# add an empty heapq to store forwarding rules
# each element of tab should be a tuple like: (traffic, host, intf) -->traffic means the number of packets related
# when a heapq consists of tuples, it is organized based on the tuples' first elements
tab = []
```

```
# recording section
# src already recorded
if packet[0].src in [rule[1] for rule in tab]:
    # find the corresbonding iterator for the current host
    cur = iter(tab)
    for rule in tab:
        if rule[1] == packet[0].src:
            cur = rule
            break
    # adjust the heapq
    # first delete, then add a new one since it's a tuple
    new_rule = (cur[0],cur[1],input_port)
    tab.remove(cur)
    tab.append(new_rule)
    heapq.heapify(tab)
```

```python
# src not recorded yet
else:
    # forwarding table is full
    if len(tab) == max_rules:
        heapq.heappop(tab)
    tab.append((0, packet[0].src, input_port))
    heapq.heapify(tab)
```

```python
# forwarding section
# dst already recorded
if packet[0].dst in [rule[1] for rule in tab]:
    # find the corresbonding iterator for the current host
    cur = iter(tab)
    for rule in tab:
        if rule[1] == packet[0].dst:
            cur = rule
            break
    log_debug ("Flooding packet {} to {}".format(packet, cur[2]))
    net.send_packet(cur[2], packet)
    # add traffic volume and adjust the heapq
    # first delete, then add a new one since it's a tuple
    new_rule = (cur[0]+1,cur[1],cur[2])
    tab.remove(cur)
    tab.append(new_rule)
    heapq.heapify(tab)
# dst not recorded yet
else:
    # do nothing if dst is the switch itself
    if packet[0].dst not in mymacs:
        for intf in my_interfaces:
            if input_port != intf.name:
                log_debug ("Flooding packet {} to {}".format(packet, intf.name))
                net.send_packet(intf.name, packet)

```

Testing

（此处提供的Test不全面）

```
Results for test scenario switch tests: 8 passed, 0 failed, 0 pending

Passed:
1   An Ethernet frame with a broadcast destination address
    should arrive on eth1
2   The Ethernet frame with a broadcast destination address
    should be forwarded out ports eth0 and eth2
3   An Ethernet frame from 20:00:00:00:00:01 to
    30:00:00:00:00:02 should arrive on eth0
4   Ethernet frame destined for 30:00:00:00:00:02 should arrive
    on eth1 after self-learning
5   An Ethernet frame from 20:00:00:00:00:03 to
    30:00:00:00:00:03 should arrive on eth2
6   Ethernet frame destined for 30:00:00:00:00:03 should be
    flooded on eth0 and eth1
7   An Ethernet frame should arrive on eth2 with destination
    address the same as eth2's MAC address
8   The switch should not do anything in response to a frame
    arriving with a destination address referring to the switch
    itself.
```

Deploying

server1:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	30:00:00:00:00:01	Broadcast	ARP	42	Who has 192.168.100.1? Tell 192.168.100.3
2	0.100596398	Private_00:00:01	30:00:00:00:00:01	ARP	42	192.168.100.1 is at 10:00:00:00:00:01
3	0.523463791	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x120d, seq=1/256, ttl=64 (reply in 4)
4	0.625066689	192.168.100.1	192.168.100.3	ICMP	98	Echo (ping) reply id=0x120d, seq=1/256, ttl=64 (request in 3)
5	1.046257343	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x120d, seq=2/512, ttl=64 (reply in 6)
6	1.149830414	192.168.100.1	192.168.100.3	ICMP	98	Echo (ping) reply id=0x120d, seq=2/512, ttl=64 (request in 5)
7	5.964527079	Private_00:00:01	30:00:00:00:00:01	ARP	42	Who has 192.168.100.3? Tell 192.168.100.1
8	6.380728832	30:00:00:00:00:01	Private_00:00:01	ARP	42	192.168.100.3 is at 30:00:00:00:00:01
9	12.084815042	30:00:00:00:00:01	Broadcast	ARP	42	Who has 192.168.100.2? Tell 192.168.100.3
10	38.318007216	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x1211, seq=1/256, ttl=64 (reply in 11)
11	38.418618476	192.168.100.1	192.168.100.3	ICMP	98	Echo (ping) reply id=0x1211, seq=1/256, ttl=64 (request in 10)
12	39.282417178	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x1211, seq=2/512, ttl=64 (reply in 13)
13	39.382510186	192.168.100.1	192.168.100.3	ICMP	98	Echo (ping) reply id=0x1211, seq=2/512, ttl=64 (request in 12)
14	43.383744011	30:00:00:00:00:01	Private_00:00:01	ARP	42	Who has 192.168.100.1? Tell 192.168.100.3
15	43.484072978	Private_00:00:01	30:00:00:00:00:01	ARP	42	192.168.100.1 is at 10:00:00:00:00:01

server2:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	30:00:00:00:00:01	Broadcast	ARP	42	Who has 192.168.100.1? Tell 192.168.100.3
2	12.084815594	30:00:00:00:00:01	Broadcast	ARP	42	Who has 192.168.100.2? Tell 192.168.100.3
3	12.185314046	20:00:00:00:00:01	30:00:00:00:00:01	ARP	42	192.168.100.2 is at 20:00:00:00:00:01
4	12.614372160	192.168.100.3	192.168.100.2	ICMP	98	Echo (ping) request id=0x120e, seq=1/256, ttl=64 (reply in 5)
5	12.729434589	192.168.100.2	192.168.100.3	ICMP	98	Echo (ping) reply id=0x120e, seq=1/256, ttl=64 (request in 4)
6	13.148472439	192.168.100.3	192.168.100.2	ICMP	98	Echo (ping) request id=0x120e, seq=2/512, ttl=64 (reply in 7)
7	13.249902982	192.168.100.2	192.168.100.3	ICMP	98	Echo (ping) reply id=0x120e, seq=2/512, ttl=64 (request in 6)
8	17.909440732	20:00:00:00:00:01	30:00:00:00:00:01	ARP	42	Who has 192.168.100.3? Tell 192.168.100.2
9	18.429659221	30:00:00:00:00:01	20:00:00:00:00:01	ARP	42	192.168.100.3 is at 30:00:00:00:00:01
10	29.104384162	192.168.100.3	192.168.100.2	ICMP	98	Echo (ping) request id=0x1210, seq=1/256, ttl=64 (reply in 11)
11	29.205198949	192.168.100.2	192.168.100.3	ICMP	98	Echo (ping) reply id=0x1210, seq=1/256, ttl=64 (request in 10)
12	30.061852196	192.168.100.3	192.168.100.2	ICMP	98	Echo (ping) request id=0x1210, seq=2/512, ttl=64 (reply in 13)
13	30.161972749	192.168.100.2	192.168.100.3	ICMP	98	Echo (ping) reply id=0x1210, seq=2/512, ttl=64 (request in 12)
14	38.318007988	192.168.100.3	192.168.100.1	ICMP	98	Echo (ping) request id=0x1211, seq=1/256, ttl=64 (no response found!)

在client的cli中输入以下命令：

```
ping -c 2 192.168.100.1
ping -c 2 192.168.100.2
ping -c 2 192.168.100.2
ping -c 2 192.168.100.1
```

首先，设置**最多容纳的rule数目为2。**

在最开始的client对server1的两次请求中，结果与之前task一致。

之后，进行若干次client对server2的请求，增加server2的traffic_volume，以达到清除有关server1对应端口信息的目的。

最后，再次进行client对server1的请求，server2的wireshark记录显示其再次被询问，从而验证了**Traffic**逻辑。

总结与感想

这次实验总体的代码量其实不算多，整体逻辑也比较清晰，但可能是因为对Python不熟悉的原因，很多时候想要实现一个feature却不知道该用什么合适的内置数据结构和方法，于是还要上网查阅，相对比较费时间。不过，从另一个角度来看，这也许是计网实验给我的另一个机遇，毕竟掌握一门语言最好的方式就是实战运用。

如果说lab_1还是熟悉实验环境，这次就算是小试牛刀了吧。对相关工具的运用也有了进步，本来只是跟着教程重复，现在可以比较深入地理解其内部逻辑，甚至自己主动去探索进一步的用法。

最后，祝愿我们的Q&A小园地能越办越好，方便大家讨论的同时也可能造福以后的学弟学妹！！

总结与感想