LAB2实验报告

专业	学号	姓名	开始/结束日期	Email
计算机科学与技术系	191220156	张桓	3.28~4.2	2659428453@qq.com

实验名称

Learning Switch

实验目的

使用`Switchyard`框架实现以太网学习交换机的核心功能，在`Basic Switch`中实现基本的交换机学习功能，除此之外还要实现三种不同机制的从转发表中清除过时的记录，需要在三个单独的`Python`文件中实现：

- `myswitch_to`：10秒后从转发表中删除一条记录
- `myswitch_lru`：从转发表中删除最近最少使用（`LRU`）条目（假设表一次只能容纳5个表项）。如果出现一个新条目，并且转发表已满，则将删除最长时间与以太网帧目的地址不匹配的表项。
- `myswitch_traffic`：删除流量最小的条目（假设表一次只能容纳5个表项）。表项的流量是交换机接收的帧数，其中目的地`MAC`地址==表项目的`MAC`地址。

实验内容

Task2: Basic Switch

> 修改myswitch.py文件，使用Switchyard框架实现交换机的自学习功能逻辑。当以太网帧到达任何端口/接口时，如果交换机知道主机可以通过该端口到达，则交换机将在适当的输出端口上发送帧；如果不知道主机在哪里，则将帧泛洪到所有端口上。

使用一个字典`mytable`来作为交换机表，字典的**表项是<`MAC`：`port`>**。在接受到一个新的`packet`时，处理其帧头`eth`，获取源和目的主机的信息，并且将源主机及接受端口记录到表中。比如一个包从`MAC1`来从`port1`端口接收到，那么就记录下来表项：`mytable[MAC1] = port1`。

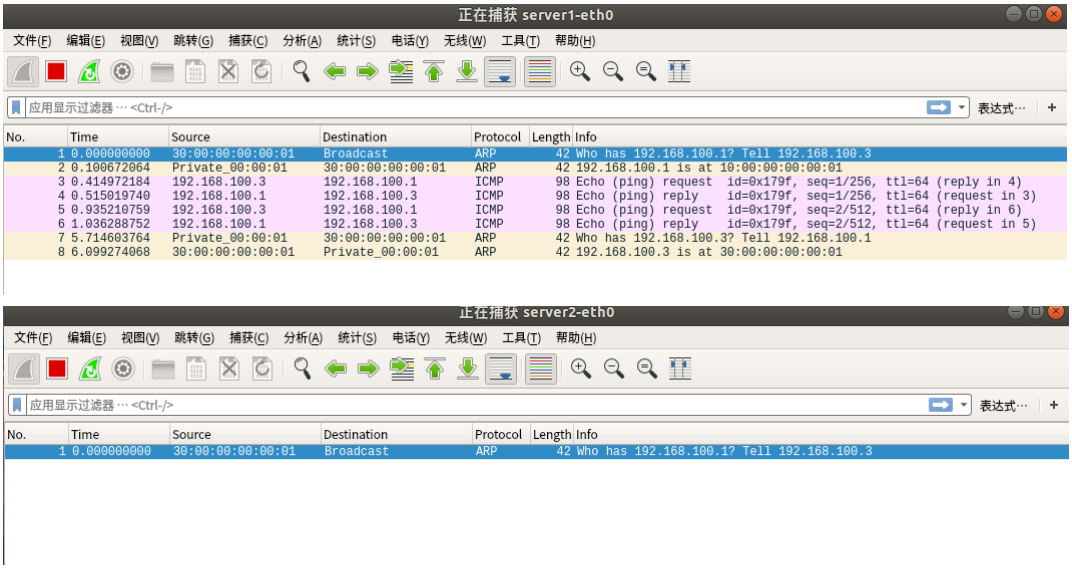
转发帧时，先检测目的主机是否在字典中有记录，如果有直接转发，如果没有则广播。

- 核心代码

```
mytable = {} #创建交换机表,the item is <src's MAC: get port>
while True:
    try:
        _, fromIface, packet = net.recv_packet()
    ...
    eth = packet.get_header(Ethernet)
    mytable[eth.src] = fromIface #TO DO:学习learn the src MAC's port
    ...
    else:
        if eth.dst in mytable.keys(): # 在表中直接转发
            log_info (f"(in table)send packet {packet} to {mytable[eth.dst]}")
            net.send_packet(mytable[eth.dst], packet)
        else: #广播,the dst MAC isn't in table, braodcast the packet
            ...
```

- 实验结果

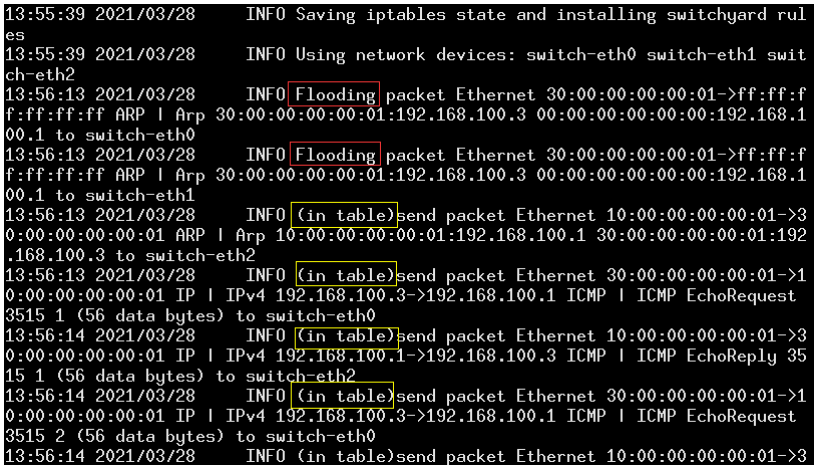
  按照 `Lab Manual` 上的方法测试 `myswitch.py`，即通过 `client ping -c 2 192.168.100.1` 指令，在 `server1` 和 `server2` 处的抓包结果分别如下：



✅分析测试结果

`client` 通过 `switch` 向 `server1` 发送2个数据包，第一次时 `switch` 不知道 `server1` 的 `MAC` 地址，只能通过**广播**来确认，*这时交换机表记录了 `client` 的端口*。可以看到抓包文件中 `server1` 和 `server2` 都捕获到了这个广播包，然后 `server1` 向 `client` 发出了 `ARP` 回应包，*这时交换机表又记录了 `server1` 的端口*。这之后 `client` 向 `server1` 发送 `ICMP` 请求包，因为表中已经记录了 `server1` 的端口号，因此直接转发，同样 `server1` 的 `ICMP` 回应包也是直接转发，因此 `server2` 是不会捕获到 `ARP` 广播之后的包的。

- 为了让输出结果更直观，当目的地址在交换机表查询到时输出 `(in table)` 的字样。可以通过 `switch` 的输出信息来观察判断，如下图：



在广播 `Flooding` 之后交换机表记录了 `client` 和 `server1` 的表项，之后两者交换数据包，它们都可以在交换机表中查询到，因此之后都显示 `in table` 。

综上，可以判断交换机基本的学习功能成功实现了。

Task3: Timeouts

> 用超时机制，使得交换机在表项超过10秒未更新后删除该表项。

同样我们用字典 `mytable` 来表示交换机表，不过这次的**表项变成了** `<MAC : [port, time]>` ，因为每个表项的最近更新时间会不断变化，只能用序列(**序列元素可变**)。在交换机表学习时，只需要多记录每个数据包到达的时间即可，用 `time` 库里的 `time()` 函数来得到交换机表记录每个数据包的准确时间。

> Python中time.time() 返回当前时间的时间戳（1970纪元后经过的浮点秒数）

- 核心代码

```
import time
...
mytable = {} #TO DO: create the table,the item is <MAC:[port, time]>
#TO DO: learn <MAC:[port, time]>, if have been in table, update it's time
mytable[eth.src] = [fromIface, time.time()]
```

删除表中过时的表项时，遍历表中所有**表项的记录时间** `mytable[key][1]` ，用**当前时间** `time.time()` 减去记录时间，**时间差** `time.time()-mytable[key][1]` 超过10s就 `pop` 掉这个表项。

> 由于在迭代字典时会改变字典，因此先用 `list()` 方法将**字典的键**形成序列，再在该序列上迭代。

那么这一部分应该写在代码框架的何处？观察发现函数框架中是通过 `while` 循环不断 `try` 接受数据包，如果接收不到(即 `NoPackets`)则直接 `continue` 再次尝试 `try` 。为了使得交换机表能够在超过 `10s` **及时更新**，将这部分代码放在 `try` **之前**显然更好一些。如果放在 `try` 之后虽然也能正确实现更新的效果，但是更新可能是不在 `10s` 后立即进行的，而是等到接收到下一个包时才进行的更新。

- 核心代码

```
#TO DO: remove the outdata items
for key in list(mytable):
    if time.time() - mytable[key][1] > 10:
        mytable.pop(key)
try:
            _, fromIface, packet = net.recv_packet()
        except NoPackets:
            continue
        ...
```

转发数据包时，如果目的`MAC`地址在表中，则直接转发，不在就广播

- 核心代码

```
if eth.dst in mytable.keys():#TO DO:send it directly
    log_info(f"(in table)send packet {packet} to {mytable[eth.dst][0]}")
    net.send_packet(mytable[eth.dst][0],  packet)
else:#broadcast the packet
    ...
```

- 实验结果

✅展示交换机的测试结果

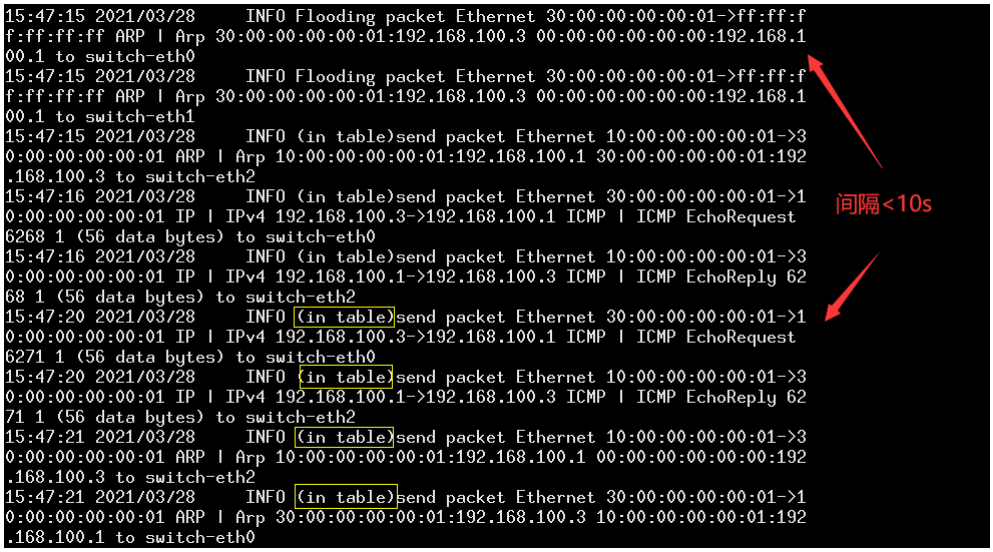
```
Results for test scenario switch tests: 9 passed, 0 failed, 0 pending


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!
```

✅在`mininet`上运行并测试

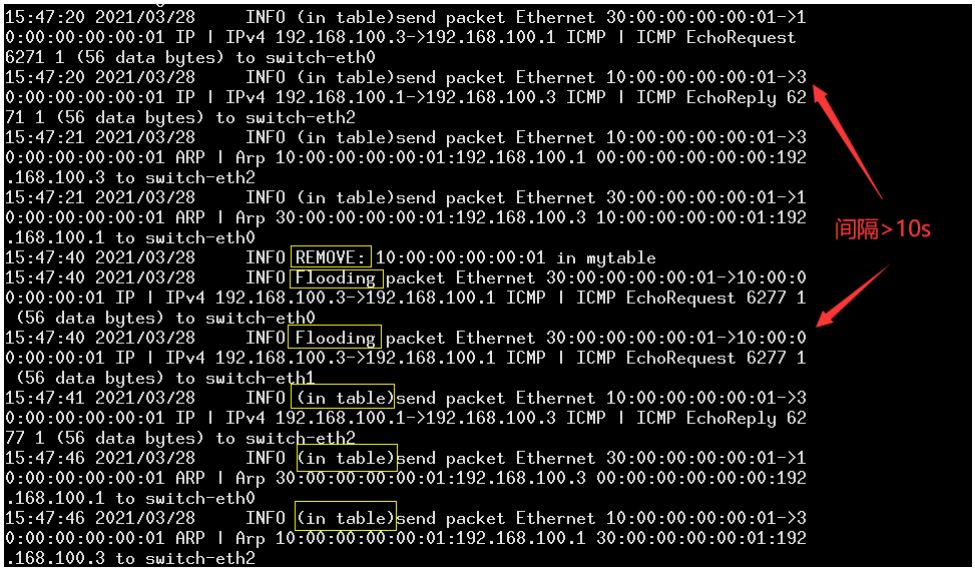
如何检测超时机制正确？肯定需要构造一些流量，这里用 `client ping -c 1 server1` 命令。为了使得输出信息便于判断，如果发生了超时删除，`myswitch_to.py` 代码中增加了**输出**：`log_info(f"REMOVE: {key} in mytable")`。(已注释)

可以通过观察 `switch` 的输出信息，经过不同的时间间隔产生流量，来判断机制是否正确。**以下输出信息均在switch的xterm上**

- 首先两次 `ping` 的时间**间隔小于10s**时，理论上讲不会输出 `REMOVE` 信息，也不会广播，而是直接在 `table` 中转发，得到结果如下，可以看到符合预期



- 如果两次 `ping` 的**间隔大于10s**时，理论上第二次 `ping` 会**输出** `REMOVE` 目的 `server1` 的 `MAC` 表项的信息，并且需要**重新广播**，得到结果如下，可以看到同样符合预期



综上，编写的超时机制应该是正确的。

Task4: Least Recently Used

> 真实的交换机还具有有限的存储空间，无法存储所学习的转发规则。假设交换机表只能存5个表项， 当表满时，使用最近最少使用的规则来删除表项。

使用`LRU`算法时，我们必须维护各个表项的**优先次序**，显然这时候再用字典不合适了。因为字典是**无序**的容器，如果想用字典来维护序关系的话不得不维护更多的信息，因此选用有序的结构比较好。

这里采用**双端队列**`deque`实现`LRU`算法，`deque`的**表项为**`[MAC, port]`。

> 在实现`LRU`算法时我最开始想到的还是用字典，只不过用字典的表项中多加了一个表示次序关系的常数，表项形如`<MAC:[port,counts]>`，`counts`越小表示该表项的使用频率越高。如果某个表项被使用其`counts`变为0，没有被用到的`counts++`。但是这样维护起来太复杂低效，最后抛弃了。

`deuqe`还可以设置最大长度`maxlen`，当表满时，我们在**队尾**用`append`加入新的表项，就会自动将其设置成`MRU`，另一端会**自动抛弃队头**最旧的表项，非常高效。

- 核心代码

```
mytable = deque(maxlen= 5) #TO DO: the item is  [MAC, port]
#learn or update the table
be_in  = False
for i, item in enumerate(mytable):
    if item[0] == eth.src: #更新 if in, only update the item
    mytable[i] = [eth.src, fromIface]
    be_in = True
if be_in == False: #学习 if not in, add it in table and make it MRU
    mytable.append([eth.src, fromIface])
```

在转发数据包时，按照流程，先在`mytable`中查询有无目的`MAC`的表项，如果有则直接转发，并且还需要将该表项设置为`MRU`，我们只需要**先删除**该表项，然后**再**`append`到队尾就可以将其设置为`MRU`。如果没有则广播。

- 核心代码

```
Find = False
for item in list(mytable):# find the dst's mac in table
    if  item[0] == eth.dst:
        Find = True # 直接转发if find, send it directly and make the item MRU
        log_info(f"(in table)send packet {packet} to {item[1]}")
        net.send_packet(item[1],  packet)
        mytable.remove(item)
        mytable.append(item)
if Find == False: #广播
    ...
```

- 实验结果

✅展示交换机的测试结果

```
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!
```

✅在`mininet`上运行并测试

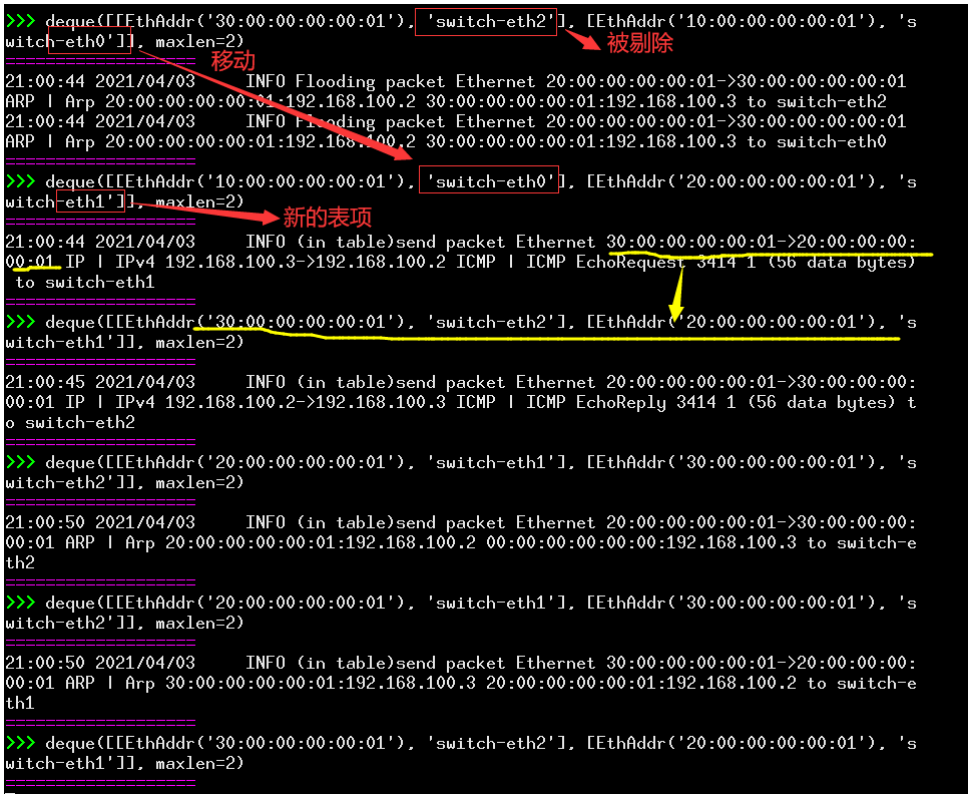
为了方便验证`LRU`机制的正确性，这里临时将交换机表的**最大大小设置为2**即`maxlen = 2`。为了更加直观将**输出交换机表的内容**，结合数据包的转发信息，通过命令`client ping -c 1 server1`和`client ping -c 1 server2`，观察交换机表的内容变化情况，就可以判断`LRU`机制是否正确。**以下输出信息均在switch的xterm上**

键入`client ping -c 1 server1`后得到：

```
20:58:06 2021/04/03     INFO Flooding packet Ethernet 30:00:00:00:00:01->ff:ff:f
f:ff:ff:ff ARP | Arp 30:00:00:00:00:01:192.168.100.3 00:00:00:00:00:00:192.168.1
00.1 to switch-eth1
==========
>>> deque([[EthAddr('30:00:00:00:00:01'), 'switch-eth2']], maxlen=2)
==========
20:58:07 2021/04/03     INFO (in table)send packet Ethernet 10:00:00:00:00:01->3
0:00:00:00:00:01 ARP | Arp 10:00:00:00:00:01:192.168.100.1 30:00:00:00:00:01:192
.168.100.3 to switch-eth2
==========
>>> deque([[EthAddr('10:00:00:00:00:01'), 'switch-eth0'], [EthAddr('30:00:00:00:
00:01'), 'switch-eth2']], maxlen=2)
==========
20:58:07 2021/04/03     INFO (in table)send packet Ethernet 30:00:00:00:00:01->1
0:00:00:00:00:01 IP | IPv4 192.168.100.3->192.168.100.1 ICMP | ICMP EchoRequest
3405 1 (56 data bytes) to switch-eth0
==========
>>> deque([[EthAddr('30:00:00:00:00:01'), 'switch-eth2'], [EthAddr('10:00:00:00:
00:01'), 'switch-eth0']], maxlen=2)
==========
20:58:07 2021/04/03     INFO (in table)send packet Ethernet 10:00:00:00:00:01->3
0:00:00:00:00:01 IP | IPv4 192.168.100.1->192.168.100.3 ICMP | ICMP EchoReply 34
05 1 (56 data bytes) to switch-eth2
==========
>>> deque([[EthAddr('10:00:00:00:00:01'), 'switch-eth0'], [EthAddr('30:00:00:00:
00:01'), 'switch-eth2']], maxlen=2)
==========
20:58:12 2021/04/03     INFO (in table)send packet Ethernet 10:00:00:00:00:01->3
0:00:00:00:00:01 ARP | Arp 10:00:00:00:00:01:192.168.100.1 00:00:00:00:00:00:192
.168.100.3 to switch-eth2
==========
>>> deque([[EthAddr('10:00:00:00:00:01'), 'switch-eth0'], [EthAddr('30:00:00:00:
00:01'), 'switch-eth2']], maxlen=2)
==========
20:58:13 2021/04/03     INFO (in table)send packet Ethernet 30:00:00:00:00:01->1
0:00:00:00:00:01 ARP | Arp 30:00:00:00:00:01:192.168.100.3 10:00:00:00:00:01:192
.168.100.1 to switch-eth0
==========
>>> deque([[EthAddr('30:00:00:00:00:01'), 'switch-eth2'], [EthAddr('10:00:00:00:
00:01'), 'switch-eth0']], maxlen=2)
==========
```

可以看到由于`client`和`server1`之间互相发送数据包，`deque`的表项顺序不断被更新(**最近用到**的表项被正确放到了**队列尾部**)。

在此基础上继续键入`client ping -c 1 server2`后得到：



这时交换机表是满的，看红框框住的部分，当**新的表项**`eth1`到来时，看图可以发现它正确将`LRU`项`eth2`剔除(**队头**的表项是**最不常用的**)，然后将`eth1`加入到表中，并且它被设置为`MRU`项。

再看黄线画出的部分，有一个包是从`30:00...`发到`20:00...`处的，`30:00...`先将原来的`LRU`项`10:00...`剔除掉，并且处于`MRU`的位置，但由于其目的地为`20:00...`，因此数据包从`20:00...`发出去后，会将`20:00...`作为新的`MRU`项。从图中可以看出，这个包发完后，交换机表的内容是`30:00...`，`20:00...`，和预测的完全一致。

综上，可以判断出`LRU`机制是正确的。

Task5: Least Traffic Volume

> 当交换机表满时，也可以使用最少流量法来删除表项。每个表项的流量是将该表项的`MAC`作为目的地址收到的数据包的数量。

这里用字典比较合适，键值仍然是源主机的`MAC`地址，不仅需要记录到达端口，还需要记录以此`MAC`为目的地址的数据包的数量，因此**字典的表项为**`<mac:[port, traffic_volume]>`。接收到数据包时先判断源`MAC`是否在表中，如果在则更新端口，并且不改变流量；如果不在则判断表是否满，若满找到流量最少的表项删掉，然后新增表项(流量初始为0)，若不满则直接新增表项。

- 核心代码

```
if eth.src in mytable:#if src's MAC is in table
    mytable[eth.src][0] = fromIface #keep the same volume count for the host
else:# if not in table
    if len(mytable) < 5: #table is not full, add it directly with volume 0
        mytable[eth.src] = [fromIface, 0]
    else: #table is full
        least  = float('inf')
        for key in list(mytable):
            if mytable[key][1] < least:
                least = mytable[key][1]
                rm_key = key #get the key whose traffic is least and remove it
        mytable.pop(rm_key)
        mytable[eth.src] = [fromIface, 0]
```

转发表项时，先判断目的`MAC`是否在表中，如果在则直接转发，并且对应流量++，不在则广播

- 核心代码

```
if eth.dst in mytable:#if in table, send it directly and volume++
    mytable[eth.dst][1] += 1
    log_info(f"(in table)send packet {packet} to {mytable[eth.dst][0]}")
    net.send_packet(mytable[eth.dst][0],  packet)
else:#broadcast
    ...
```

- 实验结果

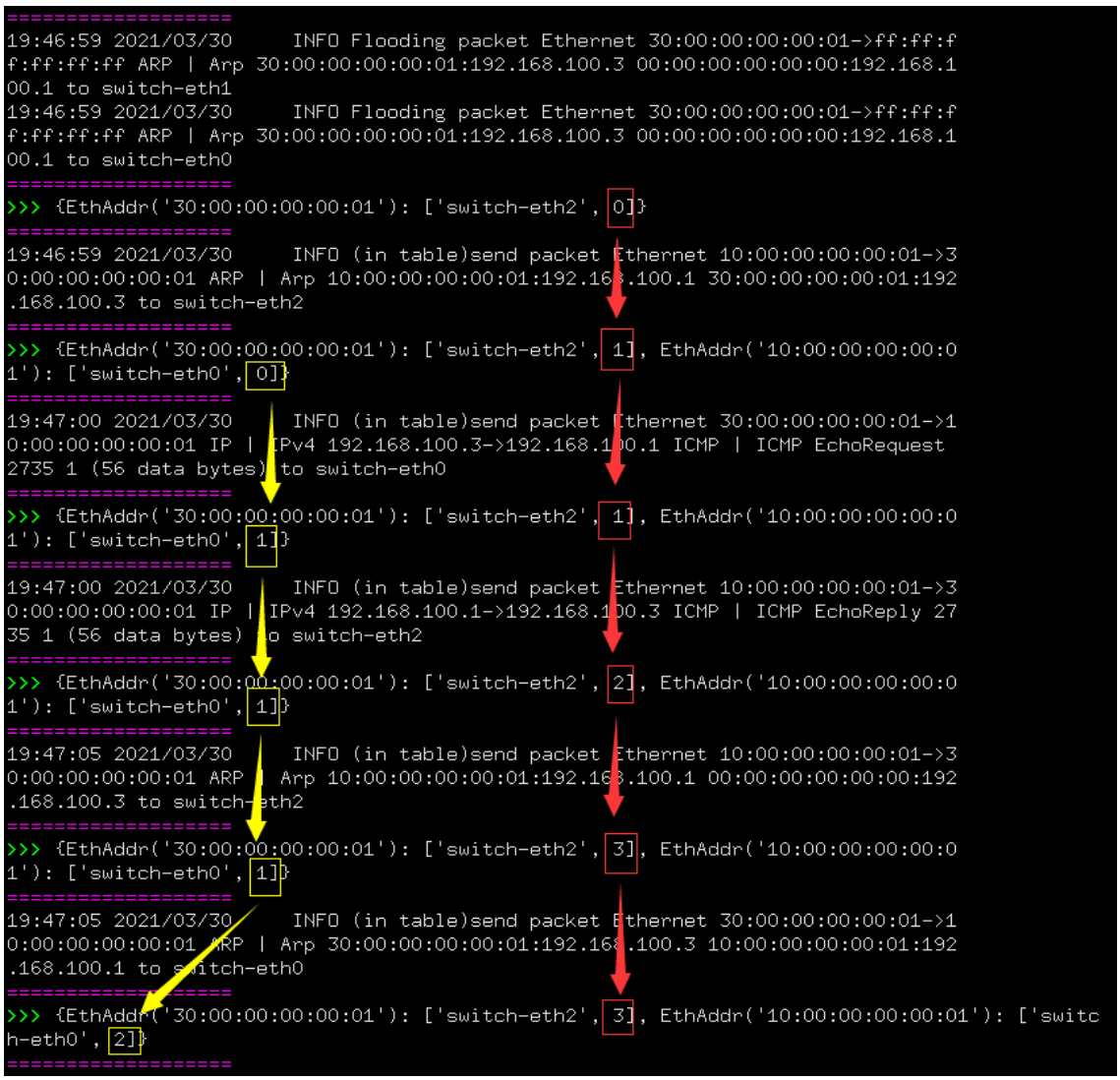
✅展示交换机的测试结果

```
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.


All tests passed!
```

✅在`mininet`上运行并测试

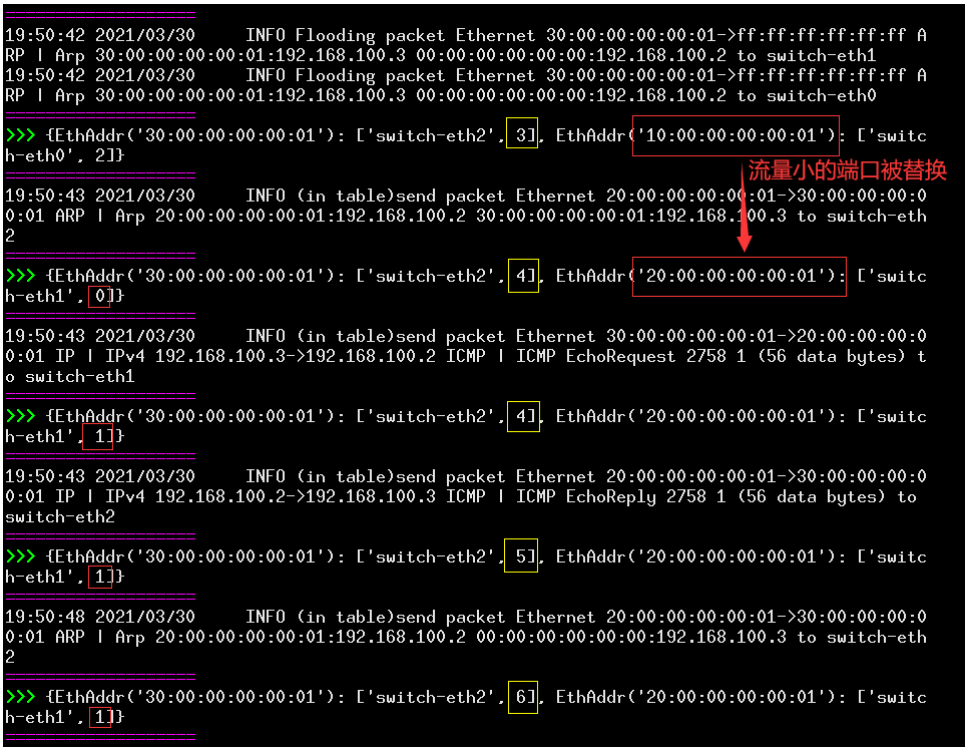
在调试时根据输出信息发现测试用例本身规模比较小，转发表最大时只有2个表项，为了测试替换机制是否正确，索性将转发表**最大大小设置为2**，即修改条件变成 `len(mytable) < 2`.

先构造流量 `client ping -c 1 server1`， `client` 和 `server1` 之间会交换若干数据包，观察交换机表是否可以正确记录端口及其流量。**以下输出信息均在switch的xterm上**：



可以看到交换机表正确记录了表项 `10:00...`：`eth0` 和表项 `30:00...`：`eth2`，并且按照 `INFO` 信息能看到**流量的增加**也正确进行了。

接下来在此基础上，继续构造流量 `client ping -c 1 server2` 来测试交换机表是否可以正确替换表项：



由于交换机表最大能够容纳2个表项，当新的 `MAC` 地址 `20:00...` 传来数据时，交换机表正确选择了删除流量小的 `10:00...` 表项（其流量**只有2**比另一个**流量为3**的要小），替换成新的 `20:00...` 表项，并且初始流量为0。

综上，编写的最小流量机制应该是正确的。

总结与感想

这次实验实现了交换机的学习功能，还有几种替换策略，由于实验 `Manual` 给出了清晰的流程，所以按照流程实现并不困难。主要是学习了一些 `python` 的语法技巧，比如如何在迭代容器的同时修改容器本身，这个在本实验的很多地方都用到了，比如删除字典中超时的表项，更新队列中的表项。

字典是无序的，需要遍历字典时，可以通过 `list[dict]` 得到字典键的序列，我们遍历这个序列相当于遍历每个键，如果此项需要修改可以直接通过键来修改字典本身。

队列是有序的，可以通过下标索引，但是如果不知道队列的大小及其索引，可以通过 `enumrate(deque, start = 0)` 函数，它将返回每个元素和对应的索引（从0开始）。

- 参考文章

  python内置函数

  如何在迭代时修改容器

最后感谢助教的耐心批改😁😁😁