Computer Networks: lab02

Learning switch

任课教师：田臣　　助教：方毓楚，郑浩等

学院	计算机科学与技术系	专业	计算机科学与技术
学号	201220154	姓名	王紫萁
Email	201220154@smail.nju.edu.cn	开始/完成时间	2022. 3.11/2022.3.12

1. 实验目的

掌握交换机在链路层传输帧和自学习的原理，实现基础的交换机以及分别拥有，Time out替换，LRU替换，traffic volume替换法则的有限表项交换机

2. 实验原理

2.1. 基础交换机

建表：交换机的基本原理就是将数据包通过适当的调度发送到正确的目的地址。由于一台交换机有多个端口因此需要在交换机内部建立起MAC地址到端口号的映射关系。这种映射维持在一张转发表中，并在一个新的帧到来时读取源地址并在转发表中添加这样的映射表项。在本次实验中，虽然基础交换机没有限制转发表的大小，不过我们在这里默认转发表的最大容量为5项，如果转发表已满且需要添加新表项时，就将最后一个表项剔除。

转发：如果在转发表中存在目的地址的表项，直接将当前帧发送到指示的端口，否则向所有端口泛洪发送。

2.2. Time out交换机

在基础交换机的基础上，我们给每一个表项一个时间戳，这样就可以在逻辑上将交换机有效利用，在这里我们依然不限制转发表的大小，但是当某一项过时后我们需要将该项剔除

2.3. LRU 交换机

在基础交换机的基础上，我们限制转发表的最大容量为5个表项，因此选择合适的替换算法可以最大概率地命中目的表项。为此选择合适的数据结构来存储表项使得能在常数时间内找到最不经常用的表项并将他剔除显得尤为重要，在第3.3.2节中我们将详细介绍用到的数据结构

2.4. Traffic Volume 交换机

在基础交换机的基础上，同样限制最大表项个数为5项，但是剔除表项的标准更改为将最小流量的端口表项剔除。具体而言，对于一个数据帧，如果该帧的目的地址在表项中存在，那么我们需要更新它的流量值，在本次实验中我们将流量值定义为帧的大小。，当转发表满时，我们就将拥有最小流量的表项剔除并用新的表项替代。

3. 具体实现与结构优化

3.1. 基础交换机

基础交换机的转发表可以作为所有其他转发表的基类，提供两个重要接口（在之后的转发表数据结构中都提供这两个接口）：

```python
def get_interface(self, mac)
def add_entry(self, mac, interface)
```

其中，`get_interface` 方法负责寻找转发表中目的地址为 `mac` 的表项，如果表项存在直接返回接口名称，否则返回 `None` 。 `add_entry` 方法负责根据转发表的状态以及传入的mac地址-端口名称关系建立新表项或者更新表项内容。

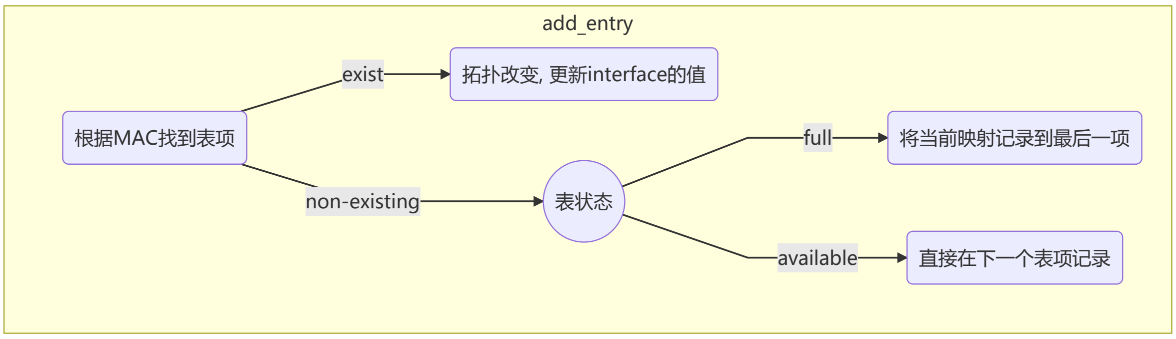
基础交换机的实现方法可以用一下代码概括（对于其他类型的交换机原理实现方法完全相同）

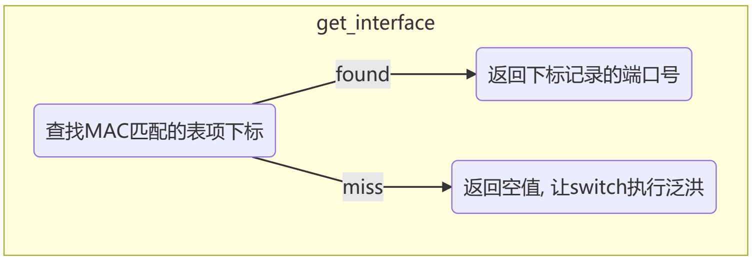
```python
def main(net: switchyard.llnetbase.LLNetBase):
    my_interfaces = net.interfaces()
    mymacs = [intf.ethaddr for intf in my_interfaces]
    table = Table() # the switching table
    # logging.getLogger().setLevel(logging.DEBUG)
    while True:
        try:
            _, fromIface, packet = net.recv_packet()
        except NoPackets:
            continue
        except Shutdown:
            break
        log_debug(f"In {net.name} received packet {packet} on {fromIface}")

        eth = packet.get_header(Ethernet)
        if eth is None:
            log_info("Received a non-Ethernet packet?!")
            return
        if eth.dst in mymacs:
            log_info("Received a packet intended for me")
        else:
            #1. add entry according to the source MAC
            table.add_entry(eth.src.toStr(), fromIface)
            dstEth = eth.dst.toStr()
            found = False    # set True if current MAC exists in the table
            if dstEth != "ff:ff:ff:ff:ff:ff":
                IFace = table.get_interface(dstEth)
                # interface token from the table
                if IFace != None: # hit!
                    found = True
                    log_info(f"Unicast packet {packet} to {IFace}")
                    # traverse my interfaces and forward the packet to the matched interface
                    for intf in my_interfaces:
                        if intf.name == IFace and intf.name != fromIface:
                            net.send_packet(intf,packet)
                            break

            #2. flood non-existing entries and broadcast entries
            if dstEth == "ff:ff:ff:ff:ff:ff" or not found:
                for intf in my_interfaces:
                    if fromIface != intf.name:
                        log_info(f"Flooding packet {packet} to {intf.name}")
                        net.send_packet(intf, packet)

    net.shutdown()
```

基础交换机的原理已经在代码中尽数体现，简单来讲就是：源地址添加/修改表项 $\to$ 目的地址查找表项 $\to$ 找到转发/缺失泛洪。

值得注意的是，之后几种交换机的本质工作原理与上无异（对于Time out交换机仅需更新时间）另外根据替换算法的特性会对两个核心函数的接口做参数改动。

为了检验交换机部分能够正常工作，我们选用最基础的转发表实现方法，即遍历寻找，动态更新。他们的核心函数实现逻辑非常简单：





3.2. Time Out交换机

`Time Out` 交换机需要对转发表中的每一个表项设置一个时间戳，在一个表项过时时需要将该项剔除，因此我们在交换机的主循环中每次都检查一遍，对于超时的节点就过滤出去。

该表依托在如下的数据结构：

```
self._dic = { mac : [interface, time_stamp]}
```

其中 `list[0], list[1]` 分别为映射的端口号与建立表项的时间

表项更新检查函数 `update` 如下：

```
def update(self):
        curTime = time.time()
        for k in list(self._dic.keys()):
            if curTime - self._dic[k][1] >= self._stayTime:
                #pay attention to the stay time, must be equal to stayTime
                self._dic.pop(k)
```

函数每一次都会获取当前时间并检查是否有表项的时间差大于等于10s

我们需要在基础交换机上稍作修改：每一轮函数都做更新

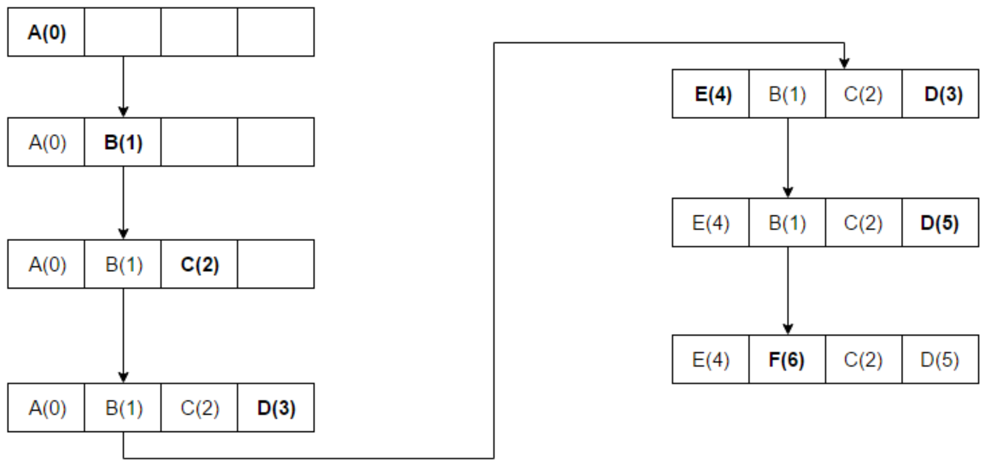
```
...
    while True:
        try:
            table.update()
            _, fromIface, packet = net.recv_packet()
...
```

`add_entry` 也需要做相应更改，需要注意的是无论什么情况都会改变时间戳（MAC不存在/存在且端口相等/存在但端口不相等）

```
def add_entry(self, mac, interface):
    self._dic[mac] = [interface, time.time()]
```

3.3. LRU交换机

3.3.1. LRU算法



- https://en.wikipedia.org/wiki/Cache_replacement_policies

如图当表满时，此时再传入一个编号为E的表项，会替换A（最不经常用的）。

结合FAQ中的例子我们能够更好的理解：

> `(h1, h4)` , `(h2, h1)` , `(h3, h1)` , `(h4, h1)` , `(h5, h1)` , `(h6, h7)` , `(h4, h5)`
>
> **A:** Assuming that the leftmost entry is the most recently used and the rightmost is the least recently used: `[h1] → [h1, h2] → [h1, h3, h2] → [h1, h4, h3, h2] → [h1, h5, h4, h3, h2] → [h6, h1, h5, h4, h3] → [h5, h6, h1, h4, h3]`

每一个 `tuple` 代表一个 `src_mac` 和 `dst_mac` 的的有序对，前者负责装入表项，后者负责更新访问状态，例如 `(h4, h5)` 到来时 `h5` 需要被更改为最经常用的表项，于是需要将其放在序列开头。

3.3.2. 数据结构

那么问题来了，哪一种数据结构能高效的找到LRU项呢？上面的例子可以给我们一点启发，我们只需要把 `LRU` 放在队列末尾，将 `MRU` 放在队列开头即可，即 `add_entry` 函数中每一次删除表项时将最后一项删除，并把新表项加入到开头。我们会想到队列的 `FIFO` 特性，但是对于 `get_interface` 需要将访问到的表项摘到开头，对于队列来说这是不可行的。双向链表可行吗？答案是肯定的，但是查找需要 `O(n)` 的复杂度。 `dictionary` 可行吗？仍然可行，但是从中删除也需要 `O(n)` 的复杂度。于是我们希望采用的结构就呼之欲出了，需要结合哈希表快速查找和双向链表快速增删的优点，就有了**哈希双向链表**。这样我们就可以在 `O(1)` 的复杂度下完成查询和删除插入操作。

一个哈希双向链表可以这样维护，用一个 `dictionary` 建立key值和节点的键值对，并用一个双向链表将每一个节点串联起来。

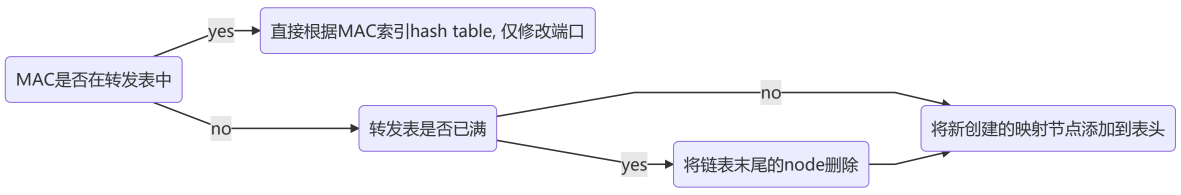
table的定义如下：

```python
class Table_LRU(object):
    '''
    head is the recently used entry while tail is the  LRU entry
    '''
    def __init__(self, capacity=5):
        self._capacity = capacity
        self._hashTable = {}    # {MAC : DulNode}, DulNodes are in self._entries
        self._entries = DulList()
```

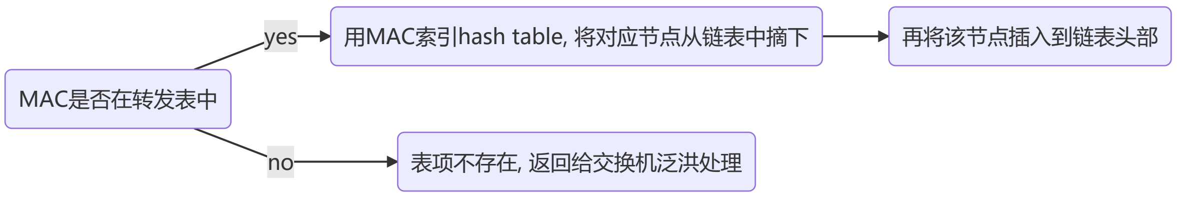
链表的详细操作我们不做重点解读，可以直接参考源码。对于两个核心函数我们简单介绍：

```python
 def add_entry(self, key, value):
        if key not in self._hashTable.keys():
            if len(self._hashTable) == self._capacity: #table is full
                self.del_LRU()  # delete the tail of the doubly linked list
            node = DulNode(key, value)  #create a new node
            self._entries.push_head(node)   #add the node to the head
            self._hashTable[key] = node
        else:
            '''
                when an entry need updating , do not change the LRU state
            '''
            self._hashTable[key].value = value

    def get_interface(self, mac):
        if mac not in self._hashTable.keys():
            return None
        else:
            self.set_recent(mac)
            return self._hashTable[mac].value
```

`add_entry` 执行（包括操作链表）的过程如下：


MAC是否在转发表中
yes
直接根据MAC索引hash table, 仅修改端口
no
转发表是否已满
no
yes
将链表末尾的node删除
将新创建的映射节点添加到表头

`get_interface` 更新的过程如下：


MAC是否在转发表中
yes
用MAC索引hash table, 将对应节点从链表中摘下
再将该节点插入到链表头部
no
表项不存在, 返回给交换机泛洪处理

4. Traffic Volume 交换机

4.1. 设计思路

Traffic volume交换机的实现思路和TO交换机类似，需要记录一下每一个目标端口的转发量。将转发量最低的表项直接剔除

4.2. 数据结构

与LRU能够采用哈希双向链表有所区别的是，当有表项更新且需要剔除表项时，如果更新的不是最小表项，那么最小表项的情况不变，可以达到 `O(1)` 的查询速度，但是如果更新了最小表项，那么需要经过 `O(n)` 的时间复杂度重新找到最小表项的所在位置。于是，通过记录最小表项的MAC并维护一个字典，可以实现上面的想法，平均情况时间复杂度为

$$A = p_{least} \times n + (1 - p_{least}) \times 1 = 1 + \frac{n-1}{n} = O(1)$$
$$p_1 = p_2 = \ldots = p_{least} = \ldots = p_n = \frac{1}{n}$$

最坏情况时间复杂度$W = O(n)$，对应每一次发送的帧的目的地址都是最小表项

4.3. 两个核心函数的实现

`add_entry` 函数：由于一个新鲜的表项总是最小的，因此将最小的MAC设为该帧的源地址并将该表项的 `volume` 置为0记录下来；如果该表项的源地址已在转发表中（拓扑改变的情况），我们仅仅更新他的 `interface` 而不更新他的 `volume` ，此时最小 `MAC` 保持不变。

`get_interface` 函数：表项不存在的处理与上相同，而对于一个存在的表项，如果源地址不是最小 `volume` 的MAC地址，直接将该表项的 `volume` 增加当前帧的 `size` ，否则对于最小表项，不经需要增加 `volume` ， 还需要遍历查找最小表项。

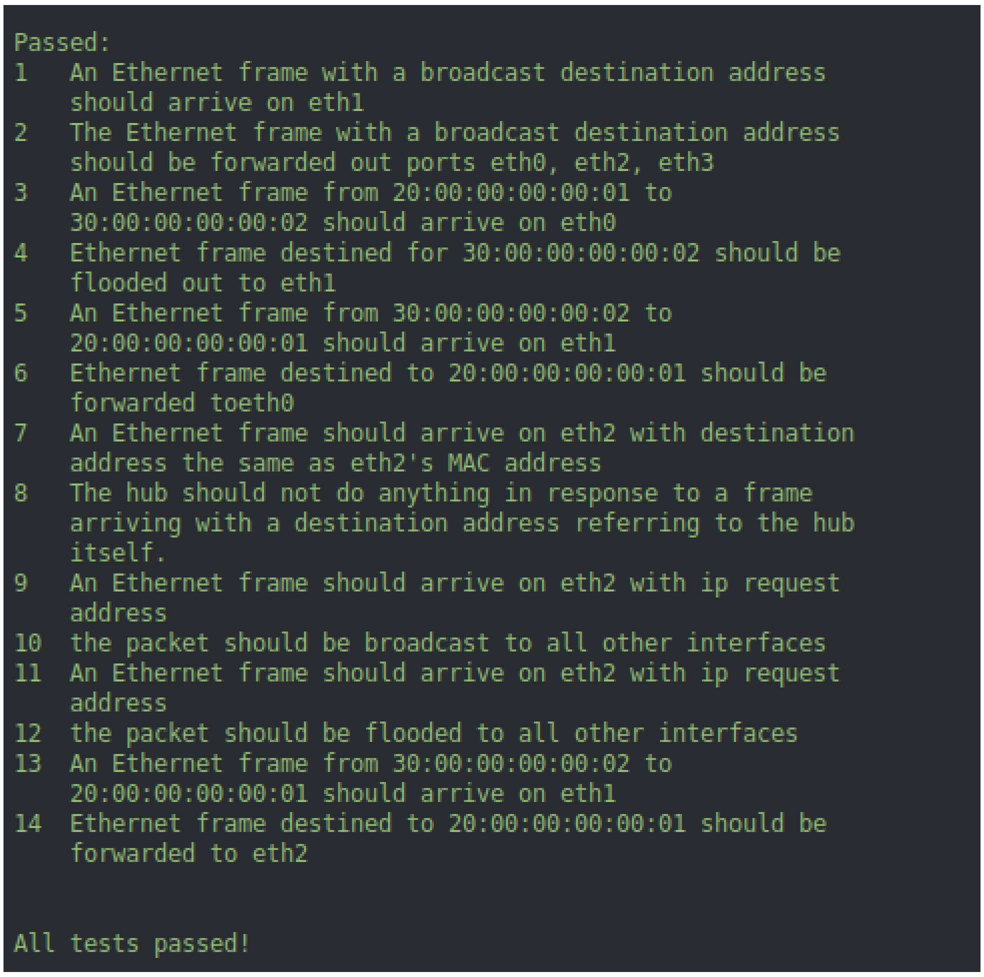
4. 测试与结果

4.1. Basic switch

自定义测试

由于基础交换机是拥有各种复杂替换算法的交换机得以正确运行的基础。因此对于基础交换机，我们的测试用例覆盖了一下几种情况：

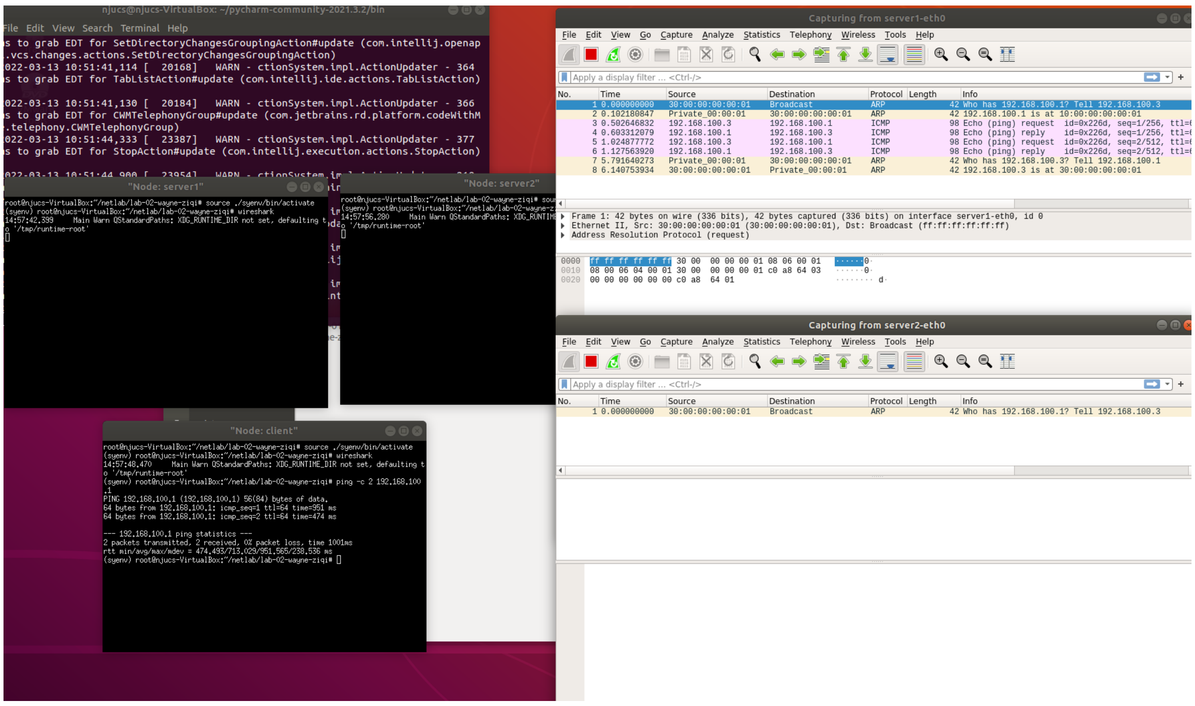
- test1：广播信号，需要所有接口都收到该帧
- test2：单播信号（表项存在），按照表中记录好的表项信息将帧转发
- test3：目标信号，目标地址是路由器的一个端口，路由器收下该帧
- test4：广播信号，源IP地址为初生地址，目的地址为广播地址，并改变拓扑结构
- test5：缺失地址，在表项中不存在该地址，应泛洪
- test6：重发信号，拓扑改变后重新发送test2的数据帧



```
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
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 be
    flooded out to eth1
5   An Ethernet frame from 30:00:00:00:00:02 to
    20:00:00:00:00:01 should arrive on eth1
6   Ethernet frame destined to 20:00:00:00:00:01 should be
    forwarded toeth0
7   An Ethernet frame should arrive on eth2 with destination
    address the same as eth2's MAC address
8   The hub should not do anything in response to a frame
    arriving with a destination address referring to the hub
    itself.
9   An Ethernet frame should arrive on eth2 with ip request
    address
10  the packet should be broadcast to all other interfaces
11  An Ethernet frame should arrive on eth2 with ip request
    address
12  the packet should be flooded to all other interfaces
13  An Ethernet frame from 30:00:00:00:00:02 to
    20:00:00:00:00:01 should arrive on eth1
14  Ethernet frame destined to 20:00:00:00:00:01 should be
    forwarded to eth2


All tests passed!
```

测试结果均正确

wireshark测试



```
s to grab EDT for SetDirectoryChangesGroupingAction#update (com.intellij.openap
.vcs.changes.actions.SetDirectoryChangesGroupingAction)
022-03-13 10:51:41,114 [  20168]   WARN - ctionSystem.impl.ActionUpdater - 364
s to grab EDT for TabListAction#update (com.intellij.ide.actions.TabListAction)

022-03-13 10:51:41,130 [  20184]   WARN - ctionSystem.impl.ActionUpdater - 366
s to grab EDT for CWMTelephonyGroup#update (com.jetbrains.rd.platform.codeWithM
.telephony.CWMTelephonyGroup)
022-03-13 10:51:44,333 [  23387]   WARN - ctionSystem.impl.ActionUpdater - 377
s to grab EDT for StopAction#update (com.intellij.execution.actions.StopAction)
```

"Node: server1"

```
root@njucs-VirtualBox:~/netlab/lab-02-wayne-ziqi# source ./syenv/bin/activate
(syenv) root@njucs-VirtualBox:~/netlab/lab-02-wayne-ziqi# wireshark
14:57:42.399     Main Warn QStandardPaths: XDG_RUNTIME_DIR not set, defaulting t
o '/tmp/runtime-root'
```

"Node: server2"

```
root@njucs-VirtualBox:~/netlab/lab-02-wayne-ziqi# sou
(syenv) root@njucs-VirtualBox:~/netlab/lab-02-wayne-z
14:57:56.280     Main Warn QStandardPaths: XDG_RUNTIME
o '/tmp/runtime-root'
```

"Node: client"

```
root@njucs-VirtualBox:~/netlab/lab-02-wayne-ziqi# source ./syenv/bin/activate
(syenv) root@njucs-VirtualBox:~/netlab/lab-02-wayne-ziqi# wireshark
14:57:48.470     Main Warn QStandardPaths: XDG_RUNTIME_DIR not set, defaulting t
o '/tmp/runtime-root'
(syenv) root@njucs-VirtualBox:~/netlab/lab-02-wayne-ziqi# ping -c 2 192.168.100
.1
PING 192.168.100.1 (192.168.100.1) 56(84) bytes of data.
64 bytes from 192.168.100.1: icmp_seq=1 ttl=64 time=951 ms
64 bytes from 192.168.100.1: icmp_seq=2 ttl=64 time=474 ms

--- 192.168.100.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 474.493/713.029/951.565/238.536 ms
(syenv) root@njucs-VirtualBox:~/netlab/lab-02-wayne-ziqi# 
```

Capturing from server1-eth0

```
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 0.102180847  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 0.502646832  192.168.100.3      192.168.100.1      ICMP         98 Echo (ping) request  id=0x226d, seq=1/256, ttl=
   4 0.603312079  192.168.100.1      192.168.100.3      ICMP         98 Echo (ping) reply    id=0x226d, seq=1/256, ttl=
   5 1.024877772  192.168.100.3      192.168.100.1      ICMP         98 Echo (ping) request  id=0x226d, seq=2/512, ttl=
   6 1.127563920  192.168.100.1      192.168.100.3      ICMP         98 Echo (ping) reply    id=0x226d, seq=2/512, ttl=
   7 5.791640273  Private_00:00:01   30:00:00:00:00:01  ARP          42 Who has 192.168.100.3? Tell 192.168.100.1
   8 6.140753934  30:00:00:00:00:01  Private_00:00:01   ARP          42 192.168.100.3 is at 30:00:00:00:00:01

▸ Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface server1-eth0, id 0
▸ Ethernet II, Src: 30:00:00:00:00:01 (30:00:00:00:00:01), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▸ Address Resolution Protocol (request)

0000  ff ff ff ff ff ff 30 00  00 00 00 01 08 06 00 01   ······0·
0010  08 00 06 04 00 01 30 00  00 00 00 01 c0 a8 64 03   ······0·
0020  00 00 00 00 00 00 c0 a8  64 01                     ········ d·
```

Capturing from server2-eth0

```
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
```

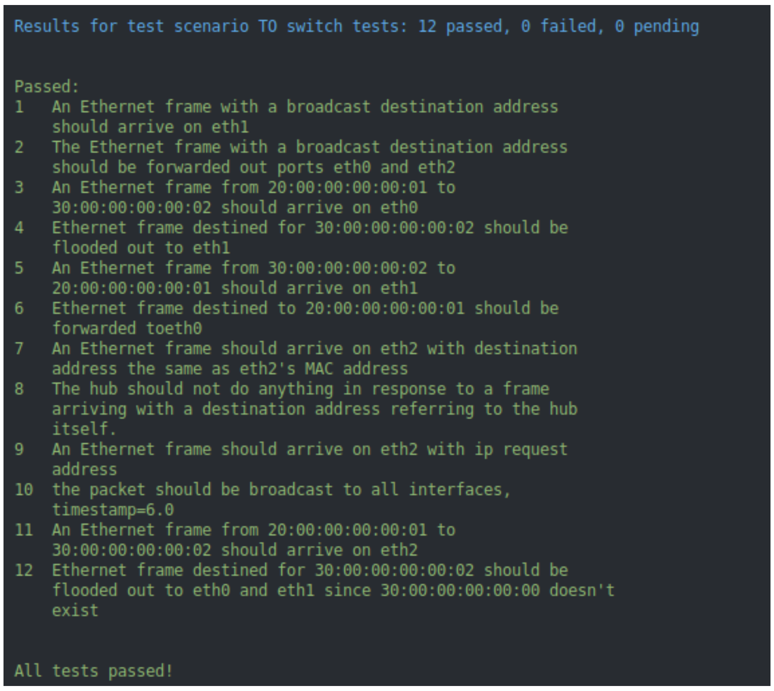
njucs@njucs-VirtualBox: ~/pycharm-community-2021.3.2/bin
File Edit View Search Terminal Help
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

在 `mininet` 中测试，首先向在client中ping两个echo到192.168.100.1（server1），预期的结果是收到两个echo request和两个echo reply，但在server2中只能看到第一次发送的广播地址（因为第二次在switch中已经有了转发的目的表项）

4.2. Time out switch

自定义测试

在basic switch的基础上编写，去掉了test5和test6，新增test5重发数据帧针对测试表项是否存在。即在 `sleep(10)` 后新的test5仍然会发出广播地址

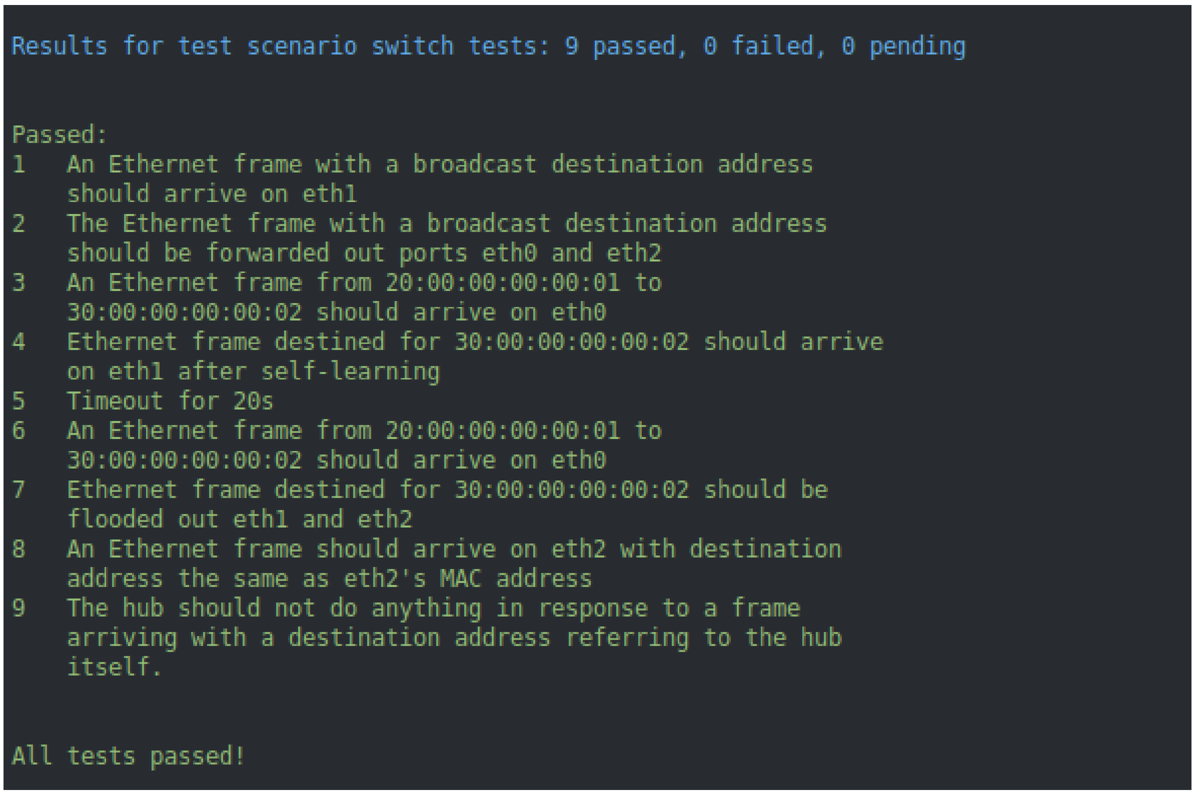


```
Results for test scenario TO switch tests: 12 passed, 0 failed, 0 pending


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 be
    flooded out to eth1
5   An Ethernet frame from 30:00:00:00:00:02 to
    20:00:00:00:00:01 should arrive on eth1
6   Ethernet frame destined to 20:00:00:00:00:01 should be
    forwarded toeth0
7   An Ethernet frame should arrive on eth2 with destination
    address the same as eth2's MAC address
8   The hub should not do anything in response to a frame
    arriving with a destination address referring to the hub
    itself.
9   An Ethernet frame should arrive on eth2 with ip request
    address
10  the packet should be broadcast to all interfaces,
    timestamp=6.0
11  An Ethernet frame from 20:00:00:00:00:01 to
    30:00:00:00:00:02 should arrive on eth2
12  Ethernet frame destined for 30:00:00:00:00:02 should be
    flooded out to eth0 and eth1 since 30:00:00:00:00:00 doesn't
    exist


All tests passed!
```

如图，11，12项为等待后的输入输出，在等待后由于不存在mac和端口的映射关系于是泛洪给所有端口

附带测试

运行.srpy文件也能正确通过

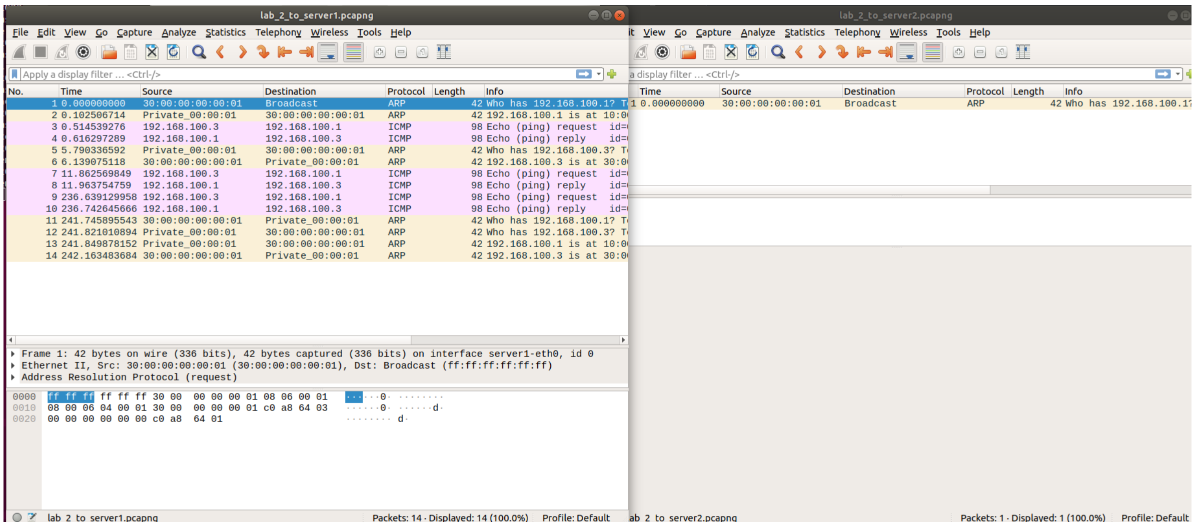


```
Results for test scenario switch tests: 9 passed, 0 failed, 0 pending


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!
```

wireshark测试

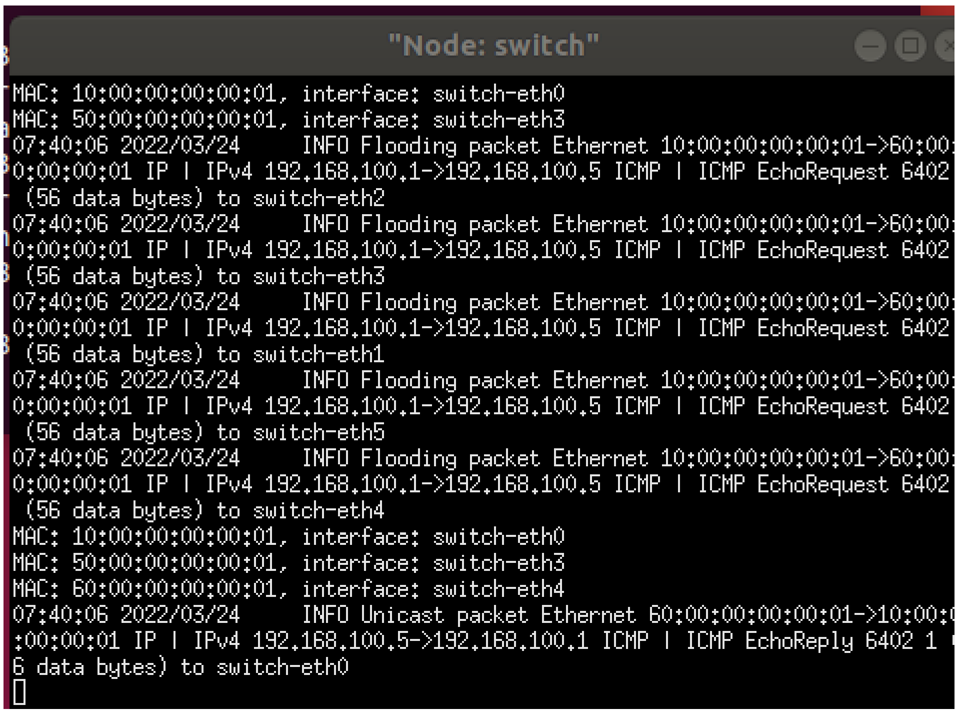
首先 `client` 连续 `ping` 连个数据包给 `server1` ，在 `server2` 中会收到广播数据包，在 `server1` 中第一次会收到 `ARP` 请求和确认以及 `ICMP` 协议请求和应答数据包，第二次只会收到 `ICMP` 请求和应答数据包。

10s后再次发送，server1中的收发包情况与第一次相同，说明表项已不再转发表中



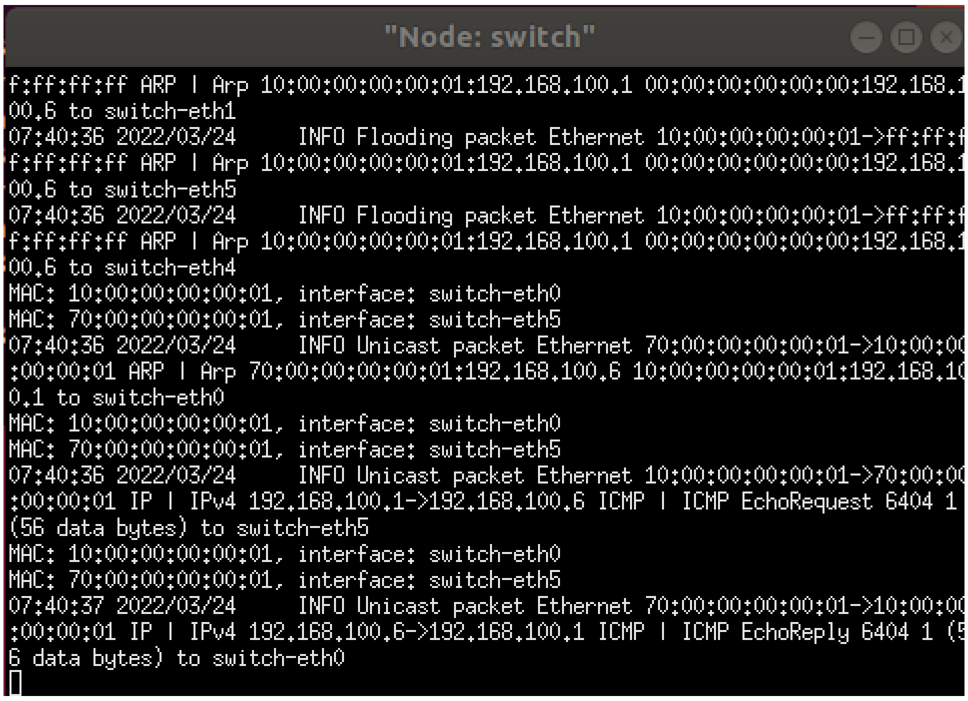
```
lab_2_to_server1.pcapng
No.  Time           Source              Destination         Protocol Length Info
   1 0.000000000    30:00:00:00:00:01   Broadcast           ARP        42 Who has 192.168.100.1? T
   2 0.102506714    Private_00:00:01    30:00:00:00:00:01   ARP        42 192.168.100.1 is at 10:0
   3 0.514539276    192.168.100.3       192.168.100.1       ICMP       98 Echo (ping) request  id=
   4 0.616297289    192.168.100.1       192.168.100.3       ICMP       98 Echo (ping) reply    id=
   5 5.790336592    Private_00:00:01    30:00:00:00:00:01   ARP        42 Who has 192.168.100.3? T
   6 6.139075118    30:00:00:00:00:01   Private_00:00:01    ARP        42 192.168.100.3 is at 30:0
   7 11.862569849   192.168.100.3       192.168.100.1       ICMP       98 Echo (ping) request  id=
   8 11.963754759   192.168.100.1       192.168.100.3       ICMP       98 Echo (ping) reply    id=
   9 236.639129958  192.168.100.3       192.168.100.1       ICMP       98 Echo (ping) request  id=
  10 236.742645666  192.168.100.1       192.168.100.3       ICMP       98 Echo (ping) reply    id=
  11 241.745895543  30:00:00:00:00:01   Private_00:00:01    ARP        42 Who has 192.168.100.1? T
  12 241.821010894  Private_00:00:01    30:00:00:00:00:01   ARP        42 Who has 192.168.100.3? T
  13 241.849878152  Private_00:00:01    30:00:00:00:00:01   ARP        42 192.168.100.1 is at 10:0
  14 242.163483684  30:00:00:00:00:01   Private_00:00:01    ARP        42 192.168.100.3 is at 30:0

Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface server1-eth0, id 0
Ethernet II, Src: 30:00:00:00:00:01 (30:00:00:00:00:01), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)

0000  ff ff ff ff ff ff 30 00  00 00 00 01 08 06 00 01   ······0· ········
0010  08 00 06 04 00 01 30 00  00 00 00 01 c0 a8 64 03   ······0· ······d·
0020  00 00 00 00 00 00 c0 a8  64 01                     ········ d·

lab_2_to_server1.pcapng          Packets: 14 · Displayed: 14 (100.0%)   Profile: Default

lab_2_to_server2.pcapng
Time           Source              Destination         Protocol Length Info
1 0.000000000  30:00:00:00:00:01   Broadcast           ARP        42 Who has 192.168.100.1?

lab_2_to_server2.pcapng          Packets: 1 · Displayed: 1 (100.0%)   Profile: Default
```

mininet测试



```
"Node: switch"
MAC: 10:00:00:00:00:01, interface: switch-eth0
MAC: 50:00:00:00:00:01, interface: switch-eth3
07:40:06 2022/03/24      INFO Flooding packet Ethernet 10:00:00:00:00:01->60:00:
0:00:00:01 IP | IPv4 192.168.100.1->192.168.100.5 ICMP | ICMP EchoRequest 6402
 (56 data bytes) to switch-eth2
07:40:06 2022/03/24      INFO Flooding packet Ethernet 10:00:00:00:00:01->60:00:
0:00:00:01 IP | IPv4 192.168.100.1->192.168.100.5 ICMP | ICMP EchoRequest 6402
 (56 data bytes) to switch-eth3
07:40:06 2022/03/24      INFO Flooding packet Ethernet 10:00:00:00:00:01->60:00:
0:00:00:01 IP | IPv4 192.168.100.1->192.168.100.5 ICMP | ICMP EchoRequest 6402
 (56 data bytes) to switch-eth1
07:40:06 2022/03/24      INFO Flooding packet Ethernet 10:00:00:00:00:01->60:00:
0:00:00:01 IP | IPv4 192.168.100.1->192.168.100.5 ICMP | ICMP EchoRequest 6402
 (56 data bytes) to switch-eth5
07:40:06 2022/03/24      INFO Flooding packet Ethernet 10:00:00:00:00:01->60:00:
0:00:00:01 IP | IPv4 192.168.100.1->192.168.100.5 ICMP | ICMP EchoRequest 6402
 (56 data bytes) to switch-eth4
MAC: 10:00:00:00:00:01, interface: switch-eth0
MAC: 50:00:00:00:00:01, interface: switch-eth3
MAC: 60:00:00:00:00:01, interface: switch-eth4
07:40:06 2022/03/24      INFO Unicast packet Ethernet 60:00:00:00:00:01->10:00:0
:00:00:01 IP | IPv4 192.168.100.5->192.168.100.1 ICMP | ICMP EchoReply 6402 1
6 data bytes) to switch-eth0
```

由 `server1` `ping` 出去连个包后等待10s左右再ping给地址 `192.168.100.6` 未在表项出现过



```
f:ff:ff:ff ARP | Arp 10:00:00:00:00:01:192.168.100.1 00:00:00:00:00:00:192.168.1
00.6 to switch-eth1
07:40:36 2022/03/24     INFO Flooding packet Ethernet 10:00:00:00:00:01->ff:ff:f
f:ff:ff:ff ARP | Arp 10:00:00:00:00:01:192.168.100.1 00:00:00:00:00:00:192.168.1
00.6 to switch-eth5
07:40:36 2022/03/24     INFO Flooding packet Ethernet 10:00:00:00:00:01->ff:ff:f
f:ff:ff:ff ARP | Arp 10:00:00:00:00:01:192.168.100.1 00:00:00:00:00:00:192.168.1
00.6 to switch-eth4
MAC: 10:00:00:00:00:01, interface: switch-eth0
MAC: 70:00:00:00:00:01, interface: switch-eth5
07:40:36 2022/03/24     INFO Unicast packet Ethernet 70:00:00:00:00:01->10:00:00
:00:00:01 ARP | Arp 70:00:00:00:00:01:192.168.100.6 10:00:00:00:00:01:192.168.10
0.1 to switch-eth0
MAC: 10:00:00:00:00:01, interface: switch-eth0
MAC: 70:00:00:00:00:01, interface: switch-eth5
07:40:36 2022/03/24     INFO Unicast packet Ethernet 10:00:00:00:00:01->70:00:00
:00:00:01 IP | IPv4 192.168.100.1->192.168.100.6 ICMP | ICMP EchoRequest 6404 1
(56 data bytes) to switch-eth5
MAC: 10:00:00:00:00:01, interface: switch-eth0
MAC: 70:00:00:00:00:01, interface: switch-eth5
07:40:37 2022/03/24     INFO Unicast packet Ethernet 70:00:00:00:00:01->10:00:00
:00:00:01 IP | IPv4 192.168.100.6->192.168.100.1 ICMP | ICMP EchoReply 6404 1 (5
6 data bytes) to switch-eth0
```

较早的eth3和eth4均被替换

4.3. LRU测试

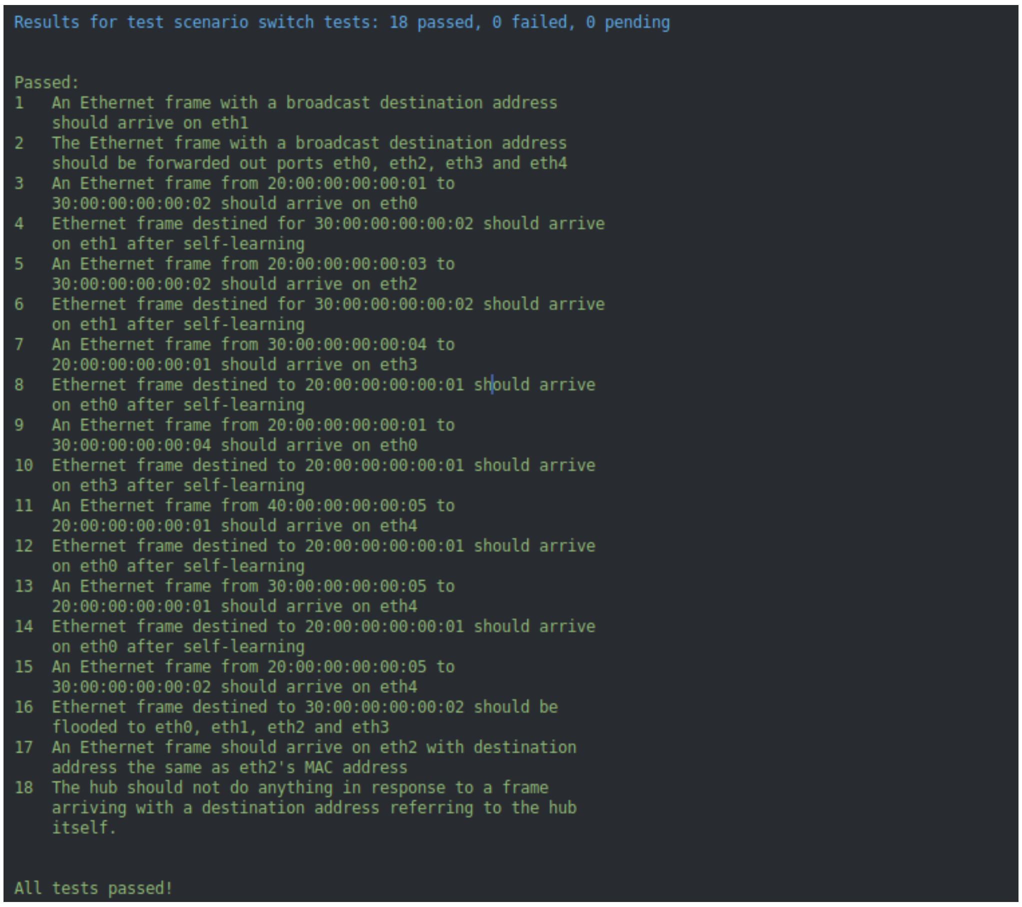
自定义检测

在检查了交换机逻辑的正确性后，我们简化一下测试方法，即通过大量模拟数据检测首发过程中表项的变化情况。在检查无误后，会用 `basic switch` 的测试文件再测试一次

```python
'''
    test_LRU.py
'''
from Table_LRU import Table_LRU
def check(res, expect):
    if res != expect:
        print("ERROR: the expected result is {0}, but got {1} instead".format(expect, res))
    else:
        print("Pass!")

if __name__ == '__main__':
    cache = Table_LRU()
    cache.add_entry('h1', 1)
    Iface = cache.get_interface('h4')
    check(Iface, None)
    cache.add_entry('h2', 2)
    Iface = cache.get_interface('h1')
    check(Iface,1)
    cache.add_entry('h3', 3)
    Iface = cache.get_interface('h1')
    check(Iface, 1)
    cache.add_entry('h4', 4)
    Iface = cache.get_interface('h1')
    check(Iface,1)
    cache.add_entry('h5', 5)
    Iface = cache.get_interface('h1')
    check(Iface, 1)
    cache.add_entry('h6', 6)
    Iface = cache.get_interface('h7')
    check(Iface, None)
    cache.add_entry('h4', 4)
    Iface = cache.get_interface('h5')
    check(Iface, 5)


```

结果均通过

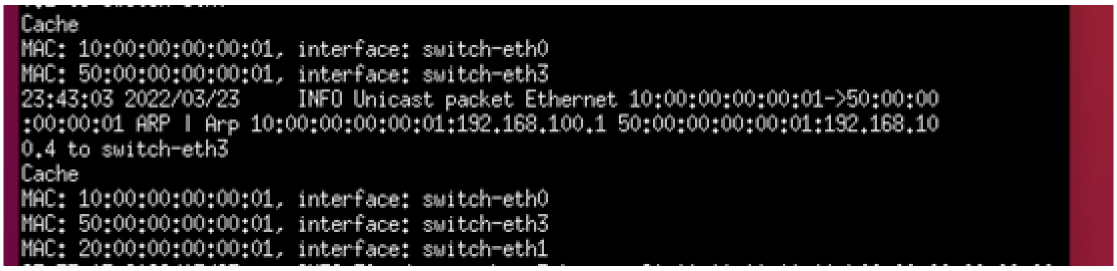
附带测试



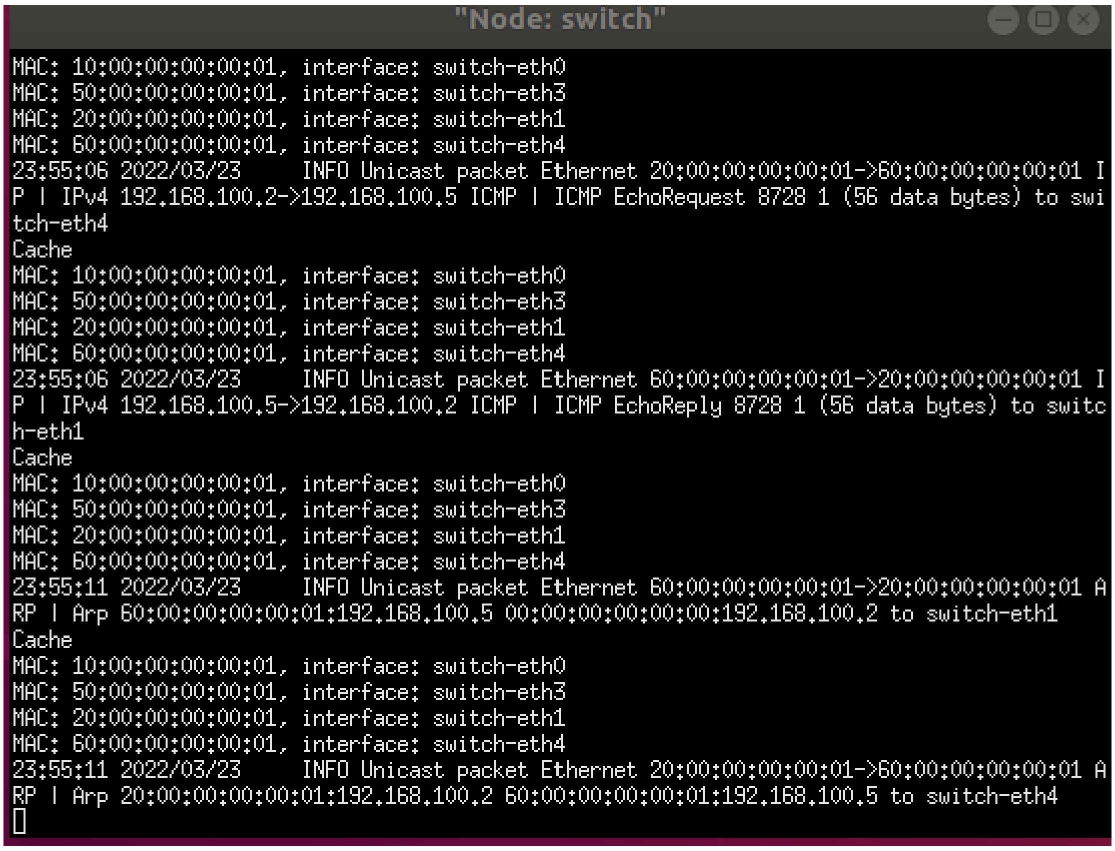
```
Results for test scenario switch tests: 18 passed, 0 failed, 0 pending


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, 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!
```

`wireshark` 测试：4.3和4.4的wireshark测试均采用连续3次从client `ping` 到server1，捕获文件见同级目录

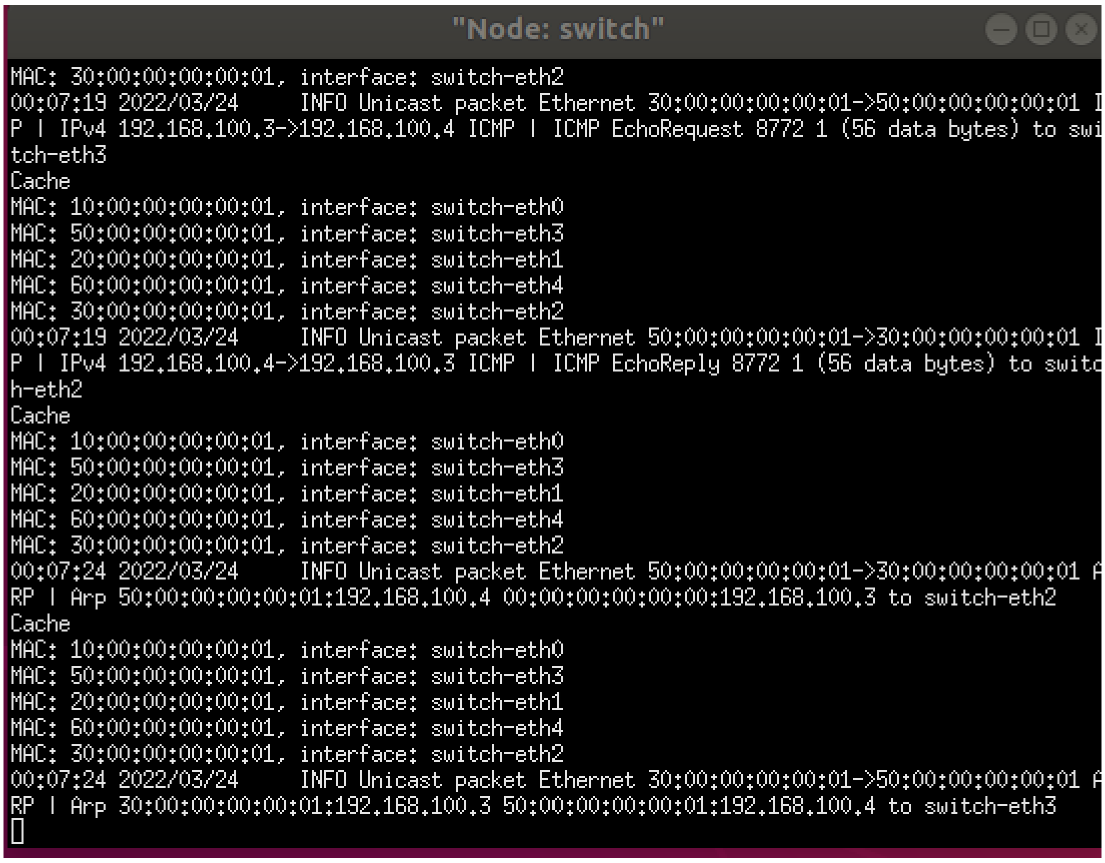
mininet 测试



```
Cache
MAC: 10:00:00:00:00:01, interface: switch-eth0
MAC: 50:00:00:00:00:01, interface: switch-eth3
23:43:03 2022/03/23     INFO Unicast packet Ethernet 10:00:00:00:00:01->50:00:00
:00:00:01 ARP | Arp 10:00:00:00:00:01:192.168.100.1 50:00:00:00:00:01:192.168.10
0.4 to switch-eth3
Cache
MAC: 10:00:00:00:00:01, interface: switch-eth0
MAC: 50:00:00:00:00:01, interface: switch-eth3
MAC: 20:00:00:00:00:01, interface: switch-eth1
```

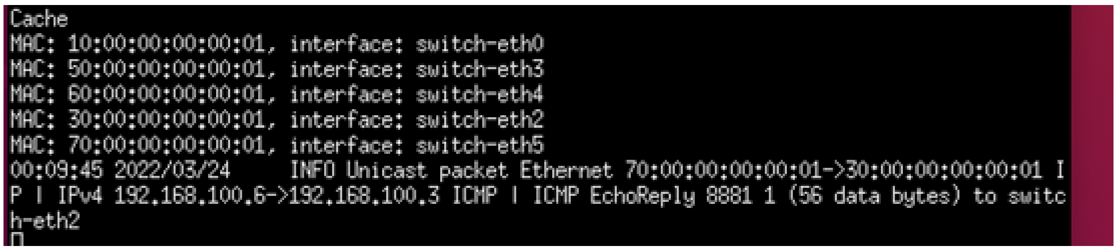


```
MAC: 10:00:00:00:00:01, interface: switch-eth0
MAC: 50:00:00:00:00:01, interface: switch-eth3
MAC: 20:00:00:00:00:01, interface: switch-eth1
MAC: 60:00:00:00:00:01, interface: switch-eth4
23:55:06 2022/03/23     INFO Unicast packet Ethernet 20:00:00:00:00:01->60:00:00:00:00:01 I
P | IPv4 192.168.100.2->192.168.100.5 ICMP | ICMP EchoRequest 8728 1 (56 data bytes) to swi
tch-eth4
Cache
MAC: 10:00:00:00:00:01, interface: switch-eth0
MAC: 50:00:00:00:00:01, interface: switch-eth3
MAC: 20:00:00:00:00:01, interface: switch-eth1
MAC: 60:00:00:00:00:01, interface: switch-eth4
23:55:06 2022/03/23     INFO Unicast packet Ethernet 60:00:00:00:00:01->20:00:00:00:00:01 I
P | IPv4 192.168.100.5->192.168.100.2 ICMP | ICMP EchoReply 8728 1 (56 data bytes) to switc
h-eth1
Cache
MAC: 10:00:00:00:00:01, interface: switch-eth0
MAC: 50:00:00:00:00:01, interface: switch-eth3
MAC: 20:00:00:00:00:01, interface: switch-eth1
MAC: 60:00:00:00:00:01, interface: switch-eth4
23:55:11 2022/03/23     INFO Unicast packet Ethernet 60:00:00:00:00:01->20:00:00:00:00:01 A
RP | Arp 60:00:00:00:00:01:192.168.100.5 00:00:00:00:00:00:192.168.100.2 to switch-eth1
Cache
MAC: 10:00:00:00:00:01, interface: switch-eth0
MAC: 50:00:00:00:00:01, interface: switch-eth3
MAC: 20:00:00:00:00:01, interface: switch-eth1
MAC: 60:00:00:00:00:01, interface: switch-eth4
23:55:11 2022/03/23     INFO Unicast packet Ethernet 20:00:00:00:00:01->60:00:00:00:00:01 A
RP | Arp 20:00:00:00:00:01:192.168.100.2 60:00:00:00:00:01:192.168.100.5 to switch-eth4
```



```
MAC: 30:00:00:00:00:01, interface: switch-eth2
00:07:19 2022/03/24     INFO Unicast packet Ethernet 30:00:00:00:00:01->50:00:00:00:00:01 I
P | IPv4 192.168.100.3->192.168.100.4 ICMP | ICMP EchoRequest 8772 1 (56 data bytes) to swi
tch-eth3
Cache
MAC: 10:00:00:00:00:01, interface: switch-eth0
MAC: 50:00:00:00:00:01, interface: switch-eth3
MAC: 20:00:00:00:00:01, interface: switch-eth1
MAC: 60:00:00:00:00:01, interface: switch-eth4
MAC: 30:00:00:00:00:01, interface: switch-eth2
00:07:19 2022/03/24     INFO Unicast packet Ethernet 50:00:00:00:00:01->30:00:00:00:00:01 I
P | IPv4 192.168.100.4->192.168.100.3 ICMP | ICMP EchoReply 8772 1 (56 data bytes) to switc
h-eth2
Cache
MAC: 10:00:00:00:00:01, interface: switch-eth0
MAC: 50:00:00:00:00:01, interface: switch-eth3
MAC: 20:00:00:00:00:01, interface: switch-eth1
MAC: 60:00:00:00:00:01, interface: switch-eth4
MAC: 30:00:00:00:00:01, interface: switch-eth2
00:07:24 2022/03/24     INFO Unicast packet Ethernet 50:00:00:00:00:01->30:00:00:00:00:01 A
RP | Arp 50:00:00:00:00:01:192.168.100.4 00:00:00:00:00:00:192.168.100.3 to switch-eth2
Cache
MAC: 10:00:00:00:00:01, interface: switch-eth0
MAC: 50:00:00:00:00:01, interface: switch-eth3
MAC: 20:00:00:00:00:01, interface: switch-eth1
MAC: 60:00:00:00:00:01, interface: switch-eth4
MAC: 30:00:00:00:00:01, interface: switch-eth2
00:07:24 2022/03/24     INFO Unicast packet Ethernet 30:00:00:00:00:01->50:00:00:00:00:01 A
RP | Arp 30:00:00:00:00:01:192.168.100.3 50:00:00:00:00:01:192.168.100.4 to switch-eth3
```

在下图中我们发现， 在最后一个包到来时，将最不经常用的 `eth1` 端口的表项去掉了，并兑换成了 `eth5` 的表项



```
Cache
MAC: 10:00:00:00:00:01, interface: switch-eth0
MAC: 50:00:00:00:00:01, interface: switch-eth3
MAC: 60:00:00:00:00:01, interface: switch-eth4
MAC: 30:00:00:00:00:01, interface: switch-eth2
MAC: 70:00:00:00:00:01, interface: switch-eth5
00:09:45 2022/03/24     INFO Unicast packet Ethernet 70:00:00:00:00:01->30:00:00:00:00:01 I
P | IPv4 192.168.100.6->192.168.100.3 ICMP | ICMP EchoReply 8881 1 (56 data bytes) to switc
h-eth2
```

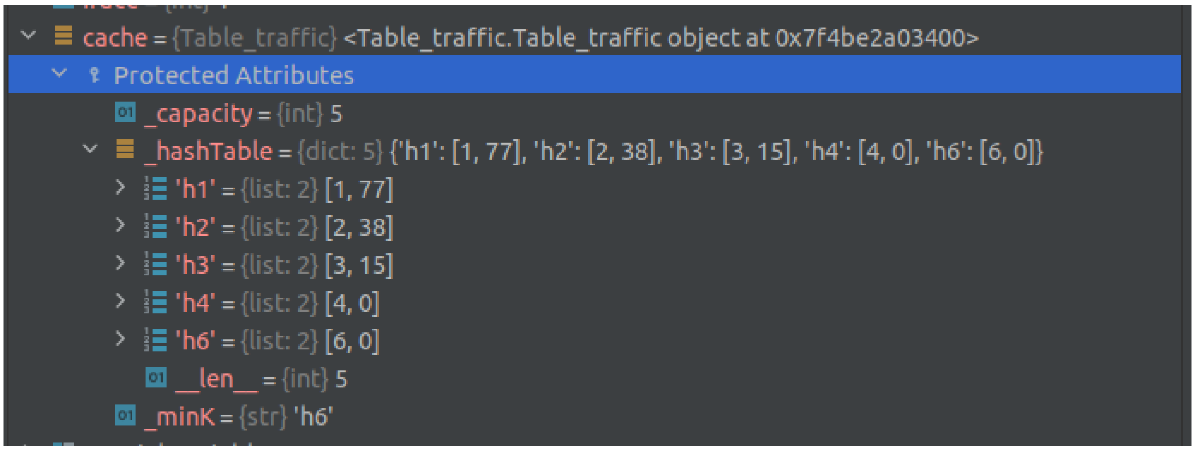
`Time Out` 测试和下面的测试都是基于此种方法，将不再详细展示（原理均一致）

4.4. Traffic Volume switch

同样，再次采用4.3中用到的测试方法，这次直接通过debug查看表中的数据。

```python
'''
    test_Traffic.py
'''
from Table_traffic import Table_traffic

if __name__ == '__main__':
    cache = Table_traffic()   # cache: {MAC: [interface, volume]}
    cache.add_entry('h1', 1)
    Iface = cache.get_interface('h4', 20)# 其中第二个参数表示数据包大小
    cache.add_entry('h2', 2)
    Iface = cache.get_interface('h2', 38)
    cache.add_entry('h3', 3)
    Iface = cache.get_interface('h3', 15)
    cache.add_entry('h4', 4)
    Iface = cache.get_interface('h5', 50)
    cache.add_entry('h5', 5)
    Iface = cache.get_interface('h1', 47)
    cache.add_entry('h6', 6)
    Iface = cache.get_interface('h1', 30)
    cache.add_entry('h4', 4)
    Iface = cache.get_interface('h5', 25)
    cache.add_entry('h2', 7)
    Iface = cache.get_interface('h4', 30)
    cache.add_entry('h4', 10)
    Iface = cache.get_interface('h1', 30)

```

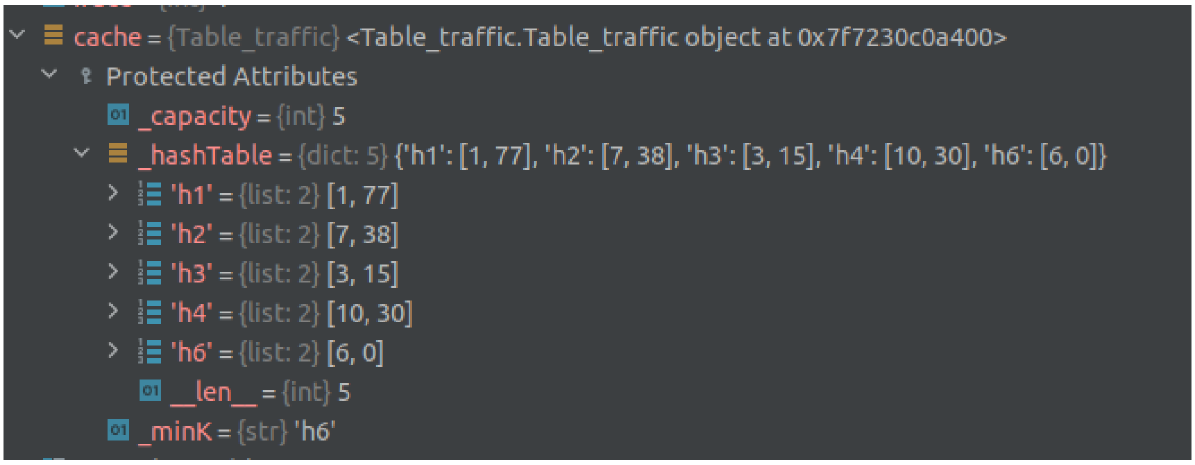
下图展示了第19行执行结束的转发表的结果



```
cache = {Table_traffic} <Table_traffic.Table_traffic object at 0x7f4be2a03400>
  Protected Attributes
    _capacity = {int} 5
    _hashTable = {dict: 5} {'h1': [1, 77], 'h2': [2, 38], 'h3': [3, 15], 'h4': [4, 0], 'h6': [6, 0]}
      'h1' = {list: 2} [1, 77]
      'h2' = {list: 2} [2, 38]
      'h3' = {list: 2} [3, 15]
      'h4' = {list: 2} [4, 0]
      'h6' = {list: 2} [6, 0]
      __len__ = {int} 5
    _minK = {str} 'h6'
```

此时显示最小MAC self._minK为h6，正是将上一次的最小表项h5替换的结果

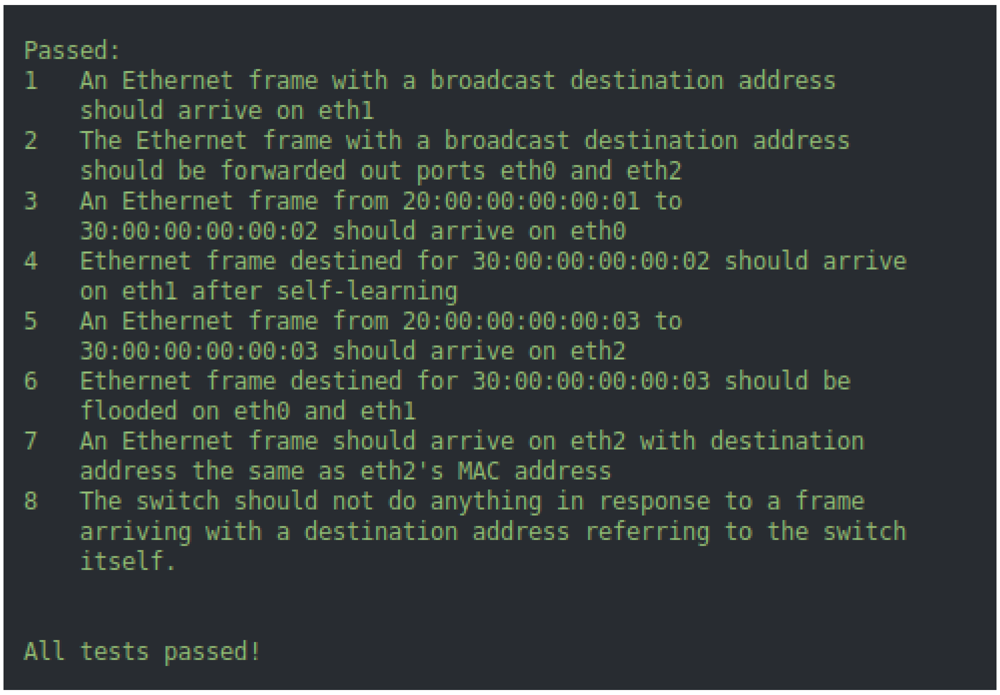
下图展示了执行完24行后的转发表的结果



```
cache = {Table_traffic} <Table_traffic.Table_traffic object at 0x7f7230c0a400>
  Protected Attributes
    _capacity = {int} 5
    _hashTable = {dict: 5} {'h1': [1, 77], 'h2': [7, 38], 'h3': [3, 15], 'h4': [10, 30], 'h6': [6, 0]}
      'h1' = {list: 2} [1, 77]
      'h2' = {list: 2} [7, 38]
      'h3' = {list: 2} [3, 15]
      'h4' = {list: 2} [10, 30]
      'h6' = {list: 2} [6, 0]
      __len__ = {int} 5
    _minK = {str} 'h6'
```

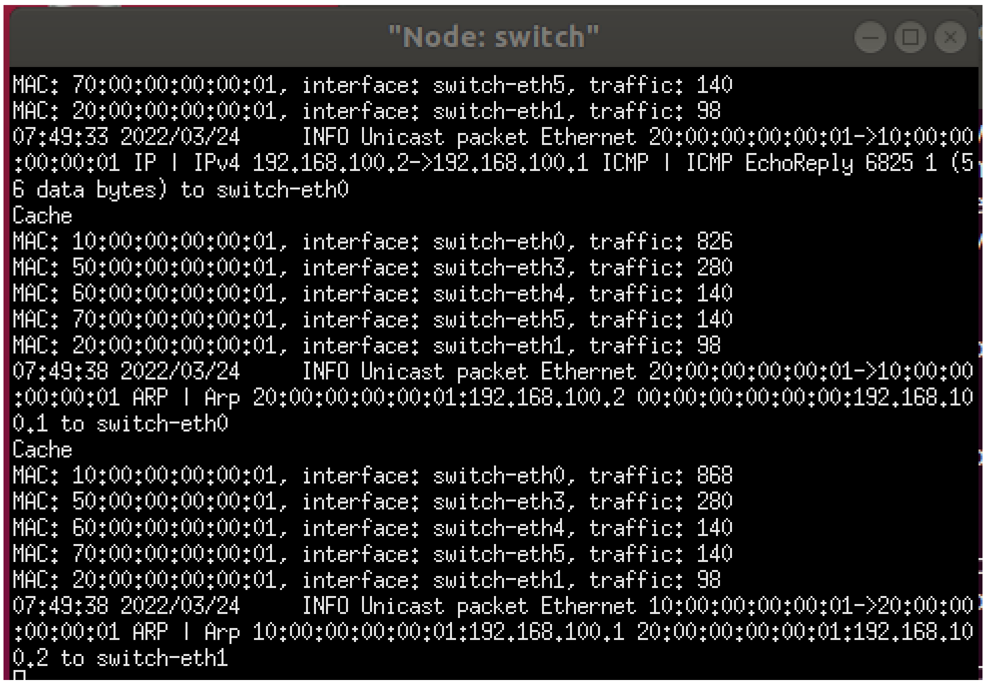
此时h4的端口变为10，minK保持h6不变

附带测试



```
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!
```

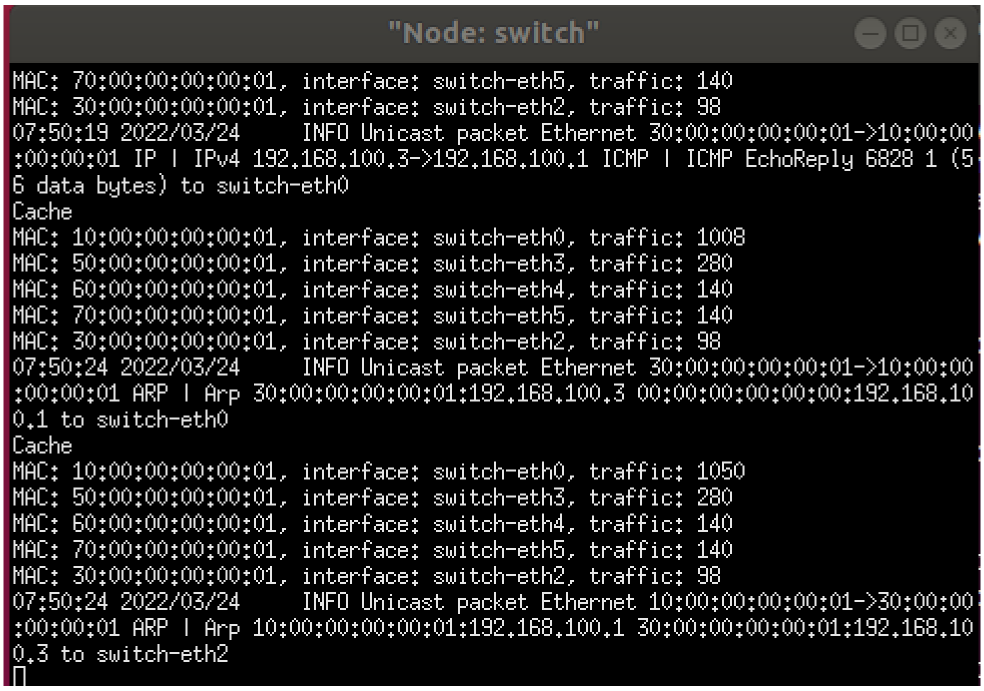
mininet测试



"Node: switch"

```
MAC: 70:00:00:00:00:01, interface: switch-eth5, traffic: 140
MAC: 20:00:00:00:00:01, interface: switch-eth1, traffic: 98
07:49:33 2022/03/24     INFO Unicast packet Ethernet 20:00:00:00:00:01->10:00:00
:00:00:01 IP | IPv4 192.168.100.2->192.168.100.1 ICMP | ICMP EchoReply 6825 1 (5
6 data bytes) to switch-eth0
Cache
MAC: 10:00:00:00:00:01, interface: switch-eth0, traffic: 826
MAC: 50:00:00:00:00:01, interface: switch-eth3, traffic: 280
MAC: 60:00:00:00:00:01, interface: switch-eth4, traffic: 140
MAC: 70:00:00:00:00:01, interface: switch-eth5, traffic: 140
MAC: 20:00:00:00:00:01, interface: switch-eth1, traffic: 98
07:49:38 2022/03/24     INFO Unicast packet Ethernet 20:00:00:00:00:01->10:00:00
:00:00:01 ARP | Arp 20:00:00:00:00:01:192.168.100.2 00:00:00:00:00:00:192.168.10
0.1 to switch-eth0
Cache
MAC: 10:00:00:00:00:01, interface: switch-eth0, traffic: 868
MAC: 50:00:00:00:00:01, interface: switch-eth3, traffic: 280
MAC: 60:00:00:00:00:01, interface: switch-eth4, traffic: 140
MAC: 70:00:00:00:00:01, interface: switch-eth5, traffic: 140
MAC: 20:00:00:00:00:01, interface: switch-eth1, traffic: 98
07:49:38 2022/03/24     INFO Unicast packet Ethernet 10:00:00:00:00:01->20:00:00
:00:00:01 ARP | Arp 10:00:00:00:00:01:192.168.100.1 20:00:00:00:00:01:192.168.10
0.2 to switch-eth1
```

在 `server1` 向三个client和server2发包后表中有四项，最小traffic为 `server2` 对应的端口



```
MAC: 70:00:00:00:00:01, interface: switch-eth5, traffic: 140
MAC: 30:00:00:00:00:01, interface: switch-eth2, traffic: 98
07:50:19 2022/03/24     INFO Unicast packet Ethernet 30:00:00:00:00:01->10:00:00
:00:00:01 IP | IPv4 192.168.100.3->192.168.100.1 ICMP | ICMP EchoReply 6828 1 (5
6 data bytes) to switch-eth0
Cache
MAC: 10:00:00:00:00:01, interface: switch-eth0, traffic: 1008
MAC: 50:00:00:00:00:01, interface: switch-eth3, traffic: 280
MAC: 60:00:00:00:00:01, interface: switch-eth4, traffic: 140
MAC: 70:00:00:00:00:01, interface: switch-eth5, traffic: 140
MAC: 30:00:00:00:00:01, interface: switch-eth2, traffic: 98
07:50:24 2022/03/24     INFO Unicast packet Ethernet 30:00:00:00:00:01->10:00:00
:00:00:01 ARP | Arp 30:00:00:00:00:01:192.168.100.3 00:00:00:00:00:00:192.168.10
0.1 to switch-eth0
Cache
MAC: 10:00:00:00:00:01, interface: switch-eth0, traffic: 1050
MAC: 50:00:00:00:00:01, interface: switch-eth3, traffic: 280
MAC: 60:00:00:00:00:01, interface: switch-eth4, traffic: 140
MAC: 70:00:00:00:00:01, interface: switch-eth5, traffic: 140
MAC: 30:00:00:00:00:01, interface: switch-eth2, traffic: 98
07:50:24 2022/03/24     INFO Unicast packet Ethernet 10:00:00:00:00:01->30:00:00
:00:00:01 ARP | Arp 10:00:00:00:00:01:192.168.100.1 30:00:00:00:00:01:192.168.10
0.3 to switch-eth2
```

当再向 `server3` 发包后， `server2` 的表项被剔除，并更换成 `server2` 所连端口对应的表项

5. 实验收获

1. 了解了交换机的工作原理
2. 掌握了在 `pycharm` 中实现了定点调试的方法，提高了debug效率
3. 理解了几种替换算法的实现机制与优化方法