南京大学本科生实验报告

课程名称：**计算机网络**　　任课教师：田臣/李文中　　助教：

学院	**计算机科学与技术系**	专业（方向）	**计算机科学与技术**
学号	**191220154**	姓名	**张涵之**
Email	**1683762615@qq.com**	开始/完成日期	**2021/4/3 – 2021/4/4**

1. 实验名称：Lab 2: Learning Switch

2. 实验目的：

Implement the core functionalities of an Ethernet learning switch.

3. 实验内容

a) Task 2: Basic Switch
implement the logic in the flowchart using the Switchyard framework.

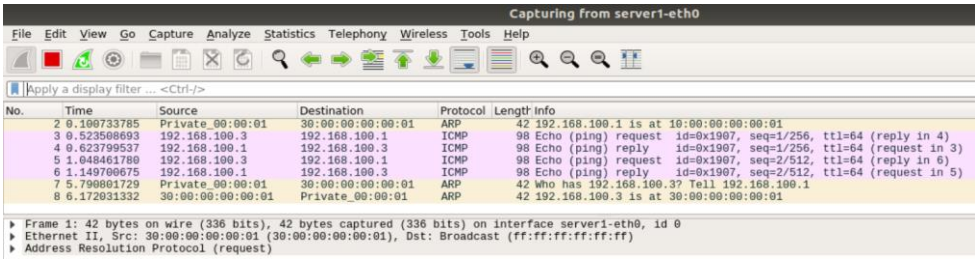
b) Task 3: Timeouts
Remove an entry from the forwarding table after 10 seconds have elapsed.

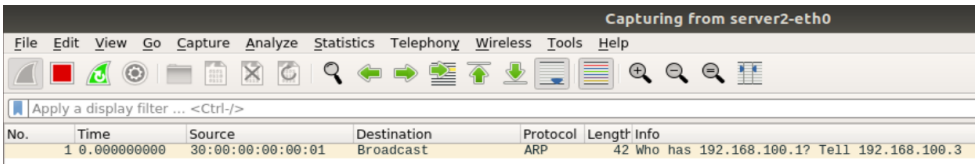
c) Task 4: Least Recently Used
Remove the least recently used (LRU) entry from the forwarding table.
(For this functionality assume that the table can only hold 5 entries. If a new entry comes and the table is full, remove the entry that has not been matched with an Ethernet frame destination address for the longest time.)

d) Task 5: Least Traffic Volume
Remove the entry that has the least traffic volume.
(For this functionality assume the table can only hold 5 entries. Traffic volume for an entry is the number of frames that the switch received where Destination MAC address == MAC address of the entry.)

4. 实验结果

a) Task 2: Basic Switch
Two echo requests and echo replies are seen in Wireshark on server1, as well as a couple other packets (ARP, or Address Resolution Protocol, packets).
There is no echo request or reply packets in Wireshark on server2, but there are ARP packets, since they are sent with broadcast destination addresses).





b) Task 3: Timeouts

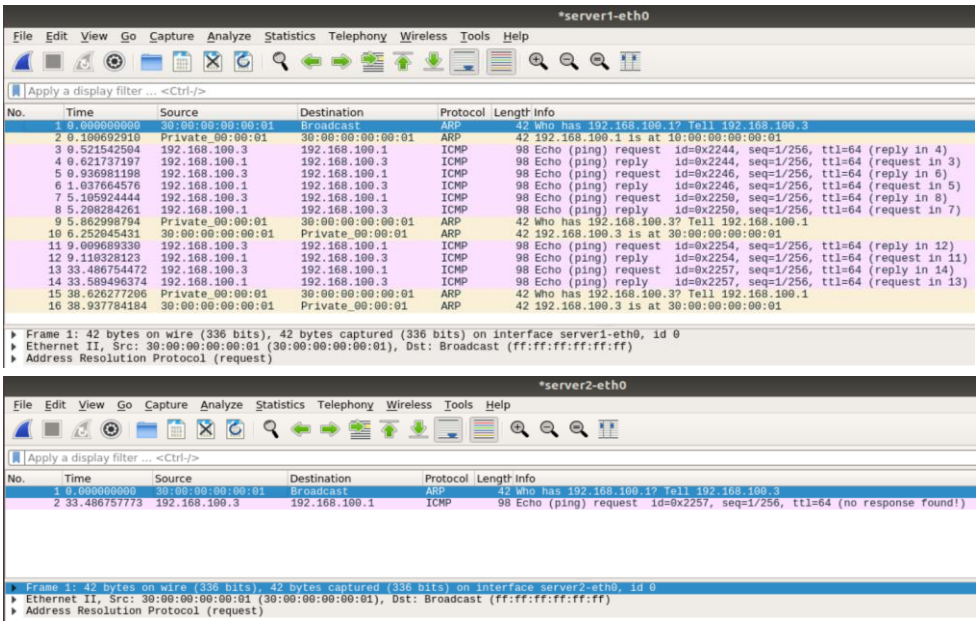
I chose not to write the test files myself. The result of the provided test file for the timeout mechanism are presented as below.

```
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!
```

Testing my timeout mechanism in Mininet:

```
mininet> client ping -c 1 server1
PING 192.168.100.1 (192.168.100.1) 56(84) bytes of data.
^[[A
64 bytes from 192.168.100.1: icmp_seq=1 ttl=64 time=982 ms

--- 192.168.100.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 982.581/982.581/982.581/0.000 ms
mininet> client ping -c 1 server1
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=416 ms

--- 192.168.100.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 416.874/416.874/416.874/0.000 ms
PING 192.168.100.1 (192.168.100.1) 56(84) bytes of data.
mininet> client ping -c 1 server1
64 bytes from 192.168.100.1: icmp_seq=1 ttl=64 time=506 ms

--- 192.168.100.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 506.163/506.163/506.163/0.000 ms
PING 192.168.100.1 (192.168.100.1) 56(84) bytes of data.
mininet> client ping -c 1 server1
64 bytes from 192.168.100.1: icmp_seq=1 ttl=64 time=455 ms

--- 192.168.100.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 455.635/455.635/455.635/0.000 ms
PING 192.168.100.1 (192.168.100.1) 56(84) bytes of data.
```

I tried to ping server1 with client for four times. The first three times are pinged less than 10 seconds apart in pairs, and a longer time was waited between the third and the fourth time, resulting in captured files below:



It can be inferred from the second screenshot that at the first time client pinged server1, it was broadcasted to both servers. The second and third time server2 received no packets, for the switch still “remembers” where server1 was. The fourth time the switch had already “forgotten” which port can be used to reach server1, so server2 received another ICMP packet and gave no response.

c) Task 4: Least Recently Used

Testing my switch with the provided test file.

```
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!
```

* I changed the TABLE_SIZE from 2 to 5 using the same test file, with 2 and 3 it failed and with 4 and 5 it passed. I tried adding some log info to see which entries was deleted in each case that led to the difference:

```
22:32:37 2021/04/03      INFO Sending packet
oRequest 0 0 (0 data bytes) to eth1
Delete 20:00:00:00:00:01 eth0 from table

Add 20:00:00:00:00:03 eth2 to table.
```

And… After testcase No. 8…

```
8    Ethernet frame destined to 20:00:00:00:00:01 should arrive
     on eth0 after self-learning
```

```
Failed:
    An Ethernet frame from 20:00:00:00:00:01 to
    30:00:00:00:00:04 should arrive on eth0
        Expected event: recv_packet Ethernet
        20:00:00:00:00:01->30:00:00:00:00:04 IP | IPv4... | ICMP...
        on eth0
```

Perhaps the table size is too small that 20:…:01 eth0 was deleted from the table before 30:…04 sends to 20:…01, therefore, flooding is used instead of sending directly, thus resulting in the error.

Log info for testcase No. 8-9 when TABLE_SIZE = 5:

```
22:31:44 2021/04/03      INFO Sending packet Ethernet 30:00:00:00:00:04->20:00:00:00:00:01 IP
oRequest 0 0 (0 data bytes) to eth0
22:31:44 2021/04/03      INFO Sending packet Ethernet 20:00:00:00:00:01->30:00:00:00:00:04 IP
oReply 0 0 (0 data bytes) to eth3
Add 40:00:00:00:00:05 eth4 to table.
```

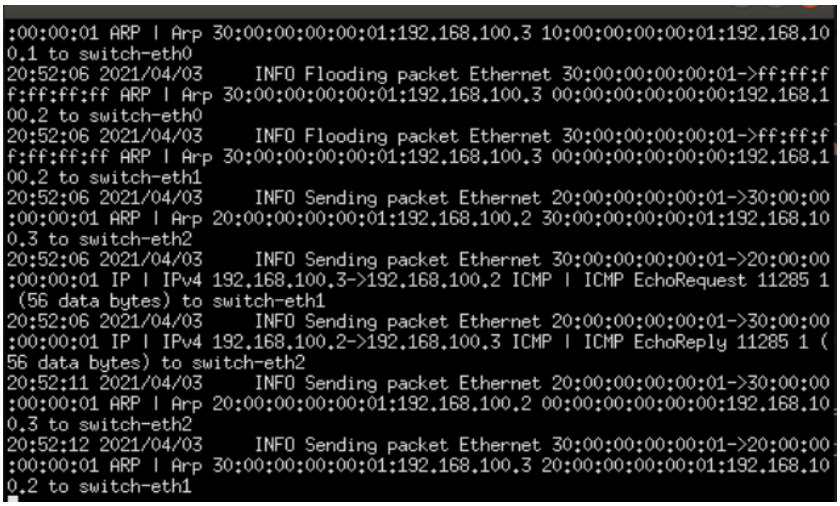
Log info for testcase No. 8 when TABLE_SIZE = 2:

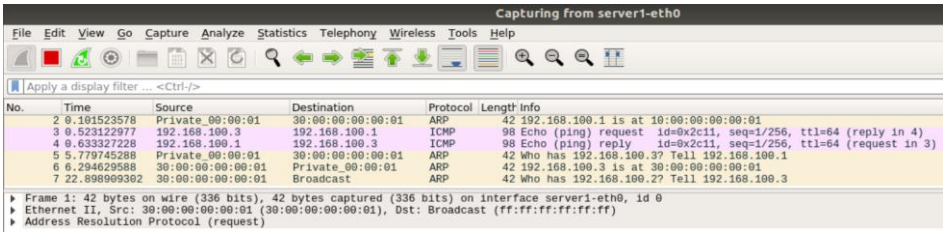
```
22:32:37 2021/04/03      INFO Flooding packet Ethernet 30:00:00:00:00:04->20:00:00:00:00:01 IP
hoRequest 0 0 (0 data bytes) to eth0
22:32:37 2021/04/03      INFO Flooding packet Ethernet 30:00:00:00:00:04->20:00:00:00:00:01 IP
hoRequest 0 0 (0 data bytes) to eth1
```

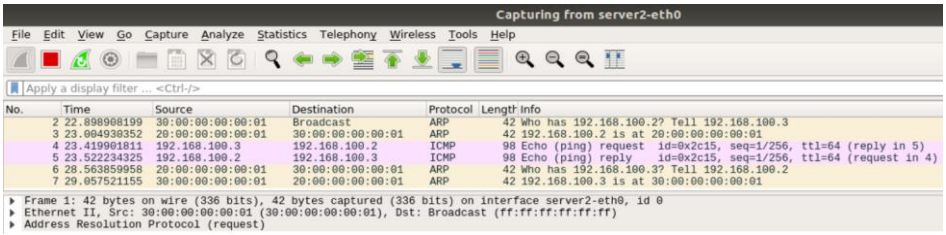
Hopefully it is somewhere near the correct explanation.

Running my switch in Mininet: client ping -c 1 server1
client ping -c 1 server2
(Because there are only three nodes (not including switch) in the network, I set the variable “TIMEOUT” from 5 to 2 to test the LRU function.)

```
"Node: switch"
20:46:46 2021/04/03      INFO Using network devices: switch-eth0 switch-eth2 swit
ch-eth1
20:47:13 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-eth0
20:47:13 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
20:47:13 2021/04/03      INFO Sending 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
20:47:14 2021/04/03      INFO Sending 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 EchoRequest 10937 1
 (56 data bytes) to switch-eth0
20:47:14 2021/04/03      INFO Sending packet Ethernet 10:00:00:00:00:01->30:00:00
:00:00:01 IP | IPv4 192.168.100.1->192.168.100.3 ICMP | ICMP EchoReply 10937 1 (
56 data bytes) to switch-eth2
20:47:19 2021/04/03      INFO Sending 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 00:00:00:00:00:00:192.168.10
0.3 to switch-eth2
20:47:19 2021/04/03      INFO Sending 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 10:00:00:00:00:01:192.168.10
0.1 to switch-eth0
```



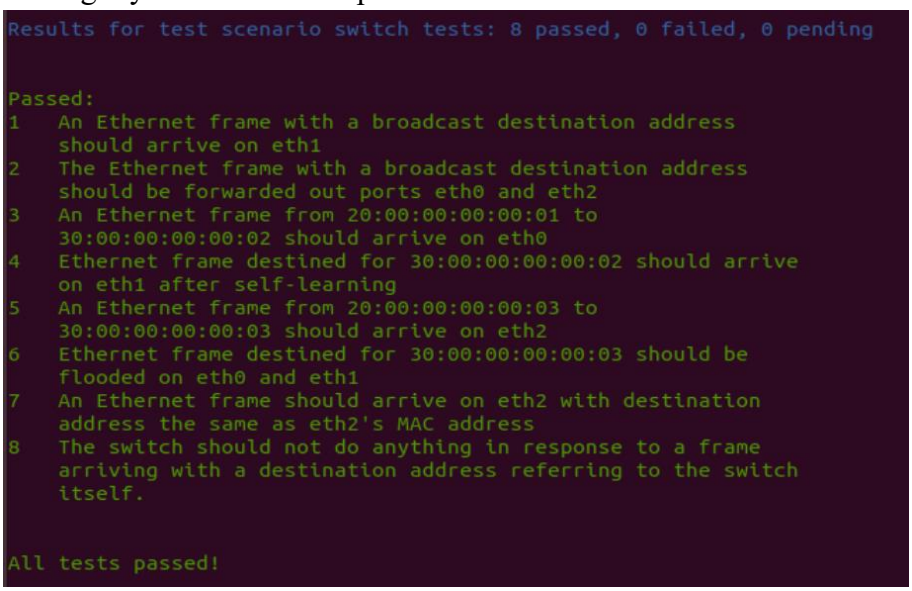




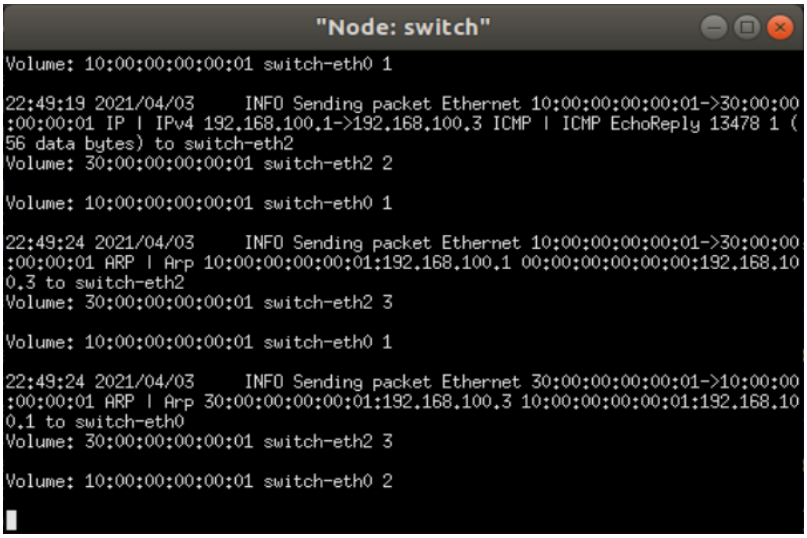
It can be inferred from the first two screenshots that after pinging server1 there are already two entries in the table (10 and 30), when client pings server2 one of them (30) is deleted to make room for a new entry (20).

d) Task 5: Least Traffic Volume

Testing my switch with the provided test file.

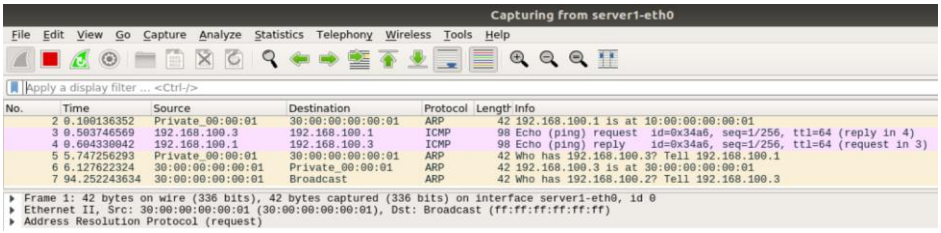


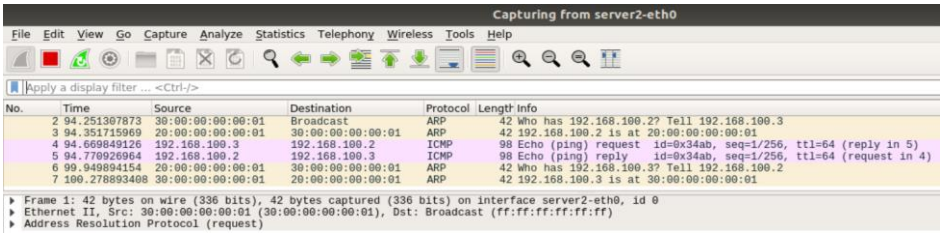
Running my switch in Mininet: client ping -c 1 server1



client ping -c 1 server2

In the two screenshots above I printed out log info to see the traffic volume of the entries currently stored in the info table.





These are the captured files on sever1 and server2.

5. 核心代码

a) Task 2: Basic Switch

```
class info:
    def __init__(self,interface):
        self.interface = interface

def main(net: switchyard.llnetbase.LLNetBase):
    my_interfaces = net.interfaces()
    mymacs = [intf.ethaddr for intf in my_interfaces]

    info_table = {}
```

* The class “info” currently contains only the interface information, but more information will be added to it in later tasks.
* A dictionary using MAC Address as the key and the class “info” as its value is used as the table to store learning information.

```
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.src not in info_table.keys():
    info_table[eth.src] = info(fromIface)
if eth.dst in mymacs:
    log_info("Received a packet intended for me")
else:
    if eth.dst not in info_table.keys():
        for intf in my_interfaces:
            if fromIface!= intf.name:
                log_info (f"Flooding packet {packet} to {intf.name}")
                net.send_packet(intf, packet)
    else:
        port = info_table[eth.dst].interface
        log_info(f"Sending packet {packet} to {port}")
        net.send_packet(port, packet)
```

* When a switch receives a new packet, it looks for the packet’s source address in the info table, if not found, the address and interface is added to the table.
* The additional cases (frame intended for the switch or broadcast) are already covered in the code for the “hub”, so we just deal with the following case:
* When an Ethernet frame arrive on any port, the switch processes its header to find the destination host. If it does not know where the host is (the address is not found in the table), it floods the frame out of all ports except the input port. Otherwise, it sends out the frame from the port according to the table.

b) Task 3: Timeouts

```
TIMEOUT = 10

class info:
    def __init__(self,interface,input_time):
        self.interface = interface
        self.input_time = input_time

def update_info(table, timeout):
    for mac in list(table):
        elapsed_time = time.time() - table[mac].input_time
        if (elapsed_time >= TIMEOUT):
            table.pop(mac)
```

* Added input time information to the class.
* The update_info function helps calculate the elapsed time of each entry in the table and delete those with elapsed time over 10 seconds.

```
#if eth.src not in info_table.keys():
info_table[eth.src] = info(fromIface, time.time())
update_info(info_table, TIMEOUT)
```

* If a packet's source address is NOT in the info table, it is added, and if it IS in the table, its timestamp is updated, so the "if" statement is no longer needed to judge whether the address is already in the table or not.
* The info table is updated every time the switch receives a new packet.

c) Task 4: Least Recently Used

* The code simply follows the logic in the flowchart.

```
TABLE_SIZE = 5

class info:
    def __init__(self,interface,index):
        self.interface = interface
        self.index = index

def main(net: switchyard.llnetbase.LLNetBase):
    my_interfaces = net.interfaces()
    mymacs = [intf.ethaddr for intf in my_interfaces]

    info_table = {}
    info_counter = 0
```

* The class now has two variables, interface and index, which marks the entry's priority level. An integer info_counter determines when the table is full.

```
if eth.src in info_table.keys():
    if info_table[eth.src].interface != fromIface:
        info_table[eth.src].interface = fromIface
else:
    if info_counter == TABLE_SIZE:
        max_index = 0
        for mac in list(info_table):
            temp_index = info_table[mac].index
            if (temp_index > max_index):
                max_index = temp_index
                max_mac = mac
        info_table.pop(max_mac)
    else:
        info_counter += 1
    for mac in list(info_table):
        info_table[mac].index += 1
    info_table[eth.src] = info(fromIface, 0)
```

* This part of the code finds out whether table contains entry for source address to updated port info, removes LRU entry if table is full, and adds the new port to the table, setting its "index" to zero. It also makes sure the priority level of other ports is correct through increasing the index to each of them.

* I really deeply regret using dictionary at this point. To delete or to modify an item through the iterating process is a true nightmare. I will definitely hand off this kind of obscure and dangerous data structure and use a list instead.

```
else:
    if eth.dst not in info_table.keys():
        for intf in my_interfaces:
            if fromIface!= intf.name:
                log_info (f"Flooding packet {packet} to {intf.name}")
                net.send_packet(intf, packet)
    else:
        dst_index = info_table[eth.dst].index
        for mac in list(info_table):
            if (info_table[mac].index <= dst_index):
                info_table[mac].index += 1
        info_table[eth.dst].index = 0
        port = info_table[eth.dst].interface
        log_info(f"Sending packet {packet} to {port}")
        net.send_packet(port, packet)
```

* If the destination entry exists, update it to be the MRU entry.
* Which involves modifying dictionary items again, another nightmare.

d) Task 5: Least Traffic Volume

This phase is pretty much the same to Task 4.

```
TABLE_SIZE = 5

class info:
    def __init__(self,interface,volume):
        self.interface = interface
        self.volume = volume
```

* The class, now with interface and traffic volume.

```
if eth.src in info_table.keys():
    if info_table[eth.src].interface != fromIface:
        info_table[eth.src].interface = fromIface
else:
    if info_counter == TABLE_SIZE:
        min_volume = 10000000000
        for mac in list(info_table):
            temp_volume = info_table[mac].volume
            if (temp_volume < min_volume):
                min_volume = temp_volume
                min_mac = mac
        info_table.pop(min_mac)
    else:
        info_counter += 1
    info_table[eth.src] = info(fromIface, 0)
```

* Finding the entry with the smallest volume and deleting it before adding new entry if the table is full. Update interface if mac already exists.

```
else:
    if eth.dst not in info_table.keys():
        for intf in my_interfaces:
            if fromIface!= intf.name:
                log_info (f"Flooding packet {packet} to {intf.name}")
                net.send_packet(intf, packet)
    else:
        info_table[eth.dst].volume += 1
        port = info_table[eth.dst].interface
        log_info(f"Sending packet {packet} to {port}")
        net.send_packet(port, packet)
```

* Updating the volume if new packet sent from the stored port.
* Finally, almost over!!!

6. 总结与感想

a) I remember using the function dict.has_key() before, but it seems not available here and would cause errors, perhaps it is due to the version of Python? Then I switched to using if _ in __ and if _ not in __ instead.

b) My knowledge of Python is so limited and outdated that I had no idea how to delete an item from a dictionary while iterating through it. Seeing the Runtime Error "dictionary changed size during iteration I looked up on the Internet and learned to change the dictionary into a list before iterating it. Then why didn't I use list in the very first place? Perhaps because finding an item using its key seems so simple and appealing. I don't quite understand the internal structure of a dictionary, but iterating the keys and using a key to find the item, it seems to me a double loop. Is it why the "update_info" function in Task 3 (Timeouts) so time consuming? Hash tables are supposed to save more time than ordinary lists, but is it worthwhile? Are there better options of data structure or solutions I failed to think of? Such mistakes make me feel so silly… ☹

c) The position became even more awkward in Task 4 and 5, when I had to delete and modify items from the dictionary. If I understood the rules correctly, the data structure provides functions like dict.values(), dict.items(), but all of them are lists, like copies of the information in the dictionary, which I can only use for purposes like calculation, output, etc. But I cannot modify the original dict using its list copy, can I? The only way I know to change the value of an item that surely can't go wrong is through "dict[key] = new_value", so I had to use it no matter how stupid it looks. Next time I will definitely read through the entire project and carefully choose a method or data structure that can be safely and efficiently used throughout the whole project. When I modified my own code from the first two steps, I had the feeling of trying to modify a pile of dog shite (sorry) and make it look like a piece of chocolate. Horrible.

d) Writing my own testcases seems somehow confusing. Observing the log info running the given testcases, I assume there are five ports (eth0 ~ eth5) on the switch. Then I printed message every time an entry is added to or deleted from the table, changed the table size and ran the test many times; its behaviors seem to match my prediction. The port name and source address in the log messages match the test cases, and the number of "add" and "delete" seems okay.

** In the last two tasks, printed messages are used to debug.

```
        #print(f"Delete {max_mac} {info_table[max_mac].interface} from table\n")
        info_table.pop(max_mac)
    else:
        info_counter += 1
    for mac in list(info_table):
        info_table[mac].index += 1
    info_table[eth.src] = info(fromIface, 0)
    #print(f"Add {eth.src} {fromIface} to table.\n")
```

```
    #for m, inf in info_table.items():
        #print(f"Volume: {m} {inf.interface} {inf.volume}\n")
```

Here are some of the results:

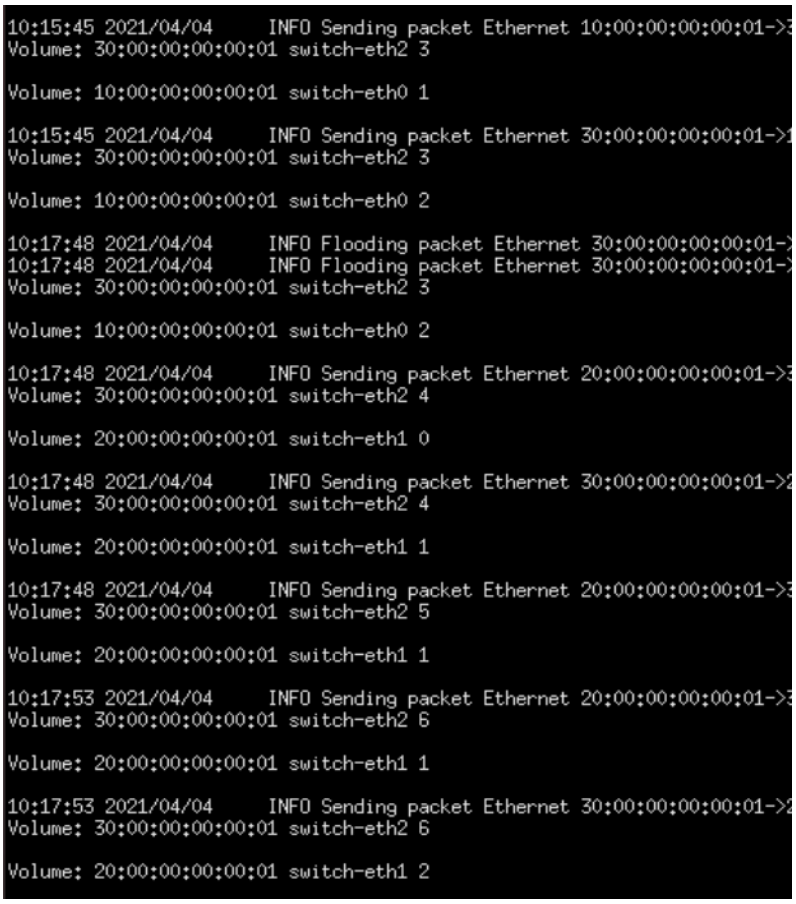
```
22:31:44 2021/04/03     INFO Starting test scenario lab-2-RainTreeCrow/t
Add 30:00:00:00:00:02 eth1 to table.

22:31:44 2021/04/03     INFO Flooding packet Ethernet 30:00:00:00:00:02-
EchoRequest 0 0 (0 data bytes) to eth0
22:31:44 2021/04/03     INFO Flooding packet Ethernet 30:00:00:00:00:02-
EchoRequest 0 0 (0 data bytes) to eth2
22:31:44 2021/04/03     INFO Flooding packet Ethernet 30:00:00:00:00:02-
EchoRequest 0 0 (0 data bytes) to eth3
22:31:44 2021/04/03     INFO Flooding packet Ethernet 30:00:00:00:00:02-
EchoRequest 0 0 (0 data bytes) to eth4
Add 20:00:00:00:00:01 eth0 to table.

22:31:44 2021/04/03     INFO Sending packet Ethernet 20:00:00:00:00:01->
oRequest 0 0 (0 data bytes) to eth1
Add 20:00:00:00:00:03 eth2 to table.

22:31:44 2021/04/03     INFO Sending packet Ethernet 20:00:00:00:00:03->
oRequest 0 0 (0 data bytes) to eth1
Add 30:00:00:00:00:04 eth3 to table.

22:31:44 2021/04/03     INFO Sending packet Ethernet 30:00:00:00:00:04->
oRequest 0 0 (0 data bytes) to eth0
22:31:44 2021/04/03     INFO Sending packet Ethernet 20:00:00:00:00:01->
oReply 0 0 (0 data bytes) to eth3
Add 40:00:00:00:00:05 eth4 to table.

22:31:44 2021/04/03     INFO Sending packet Ethernet 40:00:00:00:00:05->
oRequest 0 0 (0 data bytes) to eth0
Delete 20:00:00:00:00:03 eth2 from table

Add 30:00:00:00:00:05 eth4 to table.

22:31:44 2021/04/03     INFO Sending packet Ethernet 30:00:00:00:00:05->
oRequest 0 0 (0 data bytes) to eth0
Delete 30:00:00:00:00:02 eth1 from table
```

For TABLE_SIZE = 5, there are 5 “add”s before the first “delete”.

```
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
```

When “An Ethernet frame from _src_address_ to _dst_address_ should arrive on _port_name_”, a message of “Add _src_address_ _port_name_ to table” is printed. I assume those “should arrive on __ after self-learning” all passed so the contents of the table should be correct (which means I probably added what should be there and did not delete what shouldn’t be removed).

The messages on traffic volume are tested and observed similarly. Where there seem to be only three ports (eth0 to eth2). And there is only one test concerning self-learning, further observation shows no matter what table size I use all the testcases can always pass, for there simply aren't any tests on the replacement strategy. So, I tried some manual testing in the Mininet. There are screenshots of this in the Result section, the volumes match the packets received and the port with the smaller volume was replaced when a new port is added.



If client pings server1 and sever2 in order, the switch first learns how to reach client, then when sever1 replies it learns about server1, and the port to clients adds a volume. Other packets are treated similarly. When client pings server2 and server2 replies, the table (with size 2) is full and the link to server1 with a smaller volume is dropped. The new entry is treated similarly.

Apart from writing new testcases, there are other ways to test the logic of the switches. Trying different methods (like changing the table size) may result in failure and confusion, and my code may still be redundant or even wrong. But it helps me understand the principle of the tasks better (hopefully).

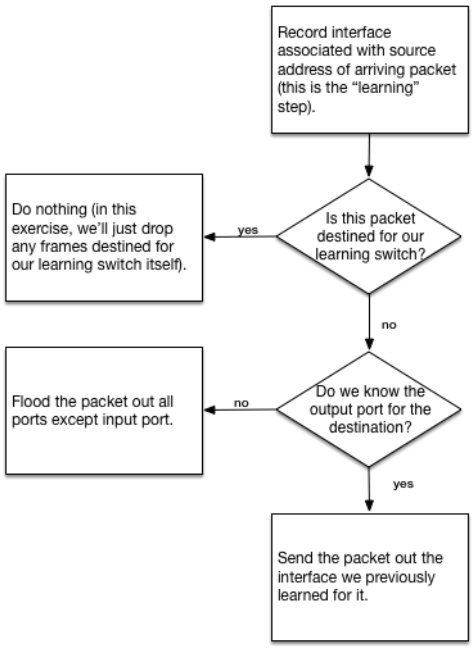
*** Please go on reading the next page. ***

** Some more thinking on the function of switch:

While reading the textbook after finishing the lab and the report, a thought occurred to me. In Section 5.4.3 Link-Layer Switches, there are three possible cases when a frame with destination address DD…DD arrives at the switch on interface x:

- There is no entry in the table for DD-DD-DD-DD-DD-DD. In this case, the switch forwards copies of the frame to the output buffers preceding *all* interfaces except for interface *x*. In other words, if there is no entry for the destination address, the switch broadcasts the frame.
- There is an entry in the table, associating DD-DD-DD-DD-DD-DD with interface *x*. In this case, the frame is coming from a LAN segment that contains adapter DD-DD-DD-DD-DD-DD. There being no need to forward the frame to any of the other interfaces, the switch performs the filtering function by discarding the frame.
- There is an entry in the table, associating DD-DD-DD-DD-DD-DD with interface $y \neq x$. In this case, the frame needs to be forwarded to the LAN segment attached to interface *y*. The switch performs its forwarding function by putting the frame in an output buffer that precedes interface *y*.

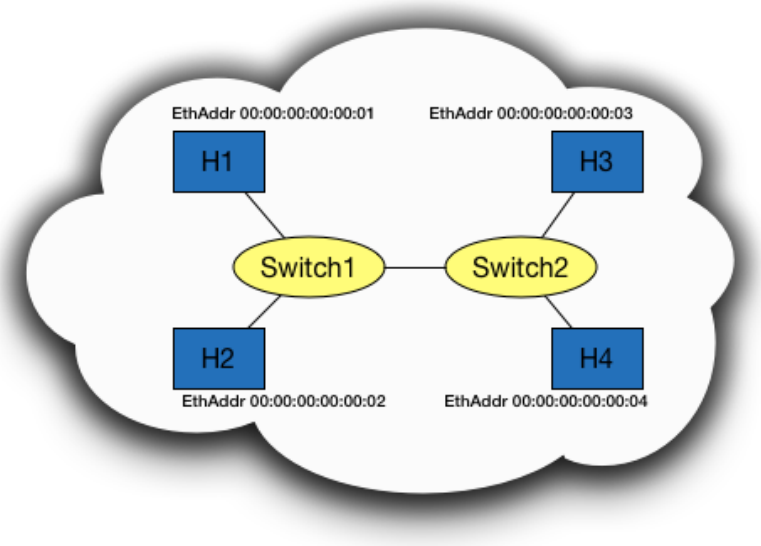
I read the lab manual over and over again, and I think it only deals with the case where the packet is destined for our learning switch, but not the second case above where the interface found in the table is the very interface the packet comes from.



I mean, "flood packet out all ports except input port", and "send packet out the interface previously learned", but what if the learned interface is also the input port?

In the following case, suppose H1 attempts to send a packet to H2, and for some reason (a certain replacement strategy, for example), switch2 knows how to reach H2 while switch1 does not (forgets), then switch1 will flood the packet to H2 and switch2, and

switch2 will send the packet back to switch1, which is unnecessary, and well, hopefully not very harmful. Let us pray that by the second time the packet reaches switch1, it has already received reply from H2, so H2 only receives one redundant packet. Otherwise I suppose the packet will go on flooding and sending between switch1 and switch2 until switch1 finally learns about H2. At least it still gets the job done, H2 receives at least one packet it needs. Or does it count as “getting the job done”?



Of course, the situation only happens when there are two switches, one of them is lucky enough to remember H2, while the other one is lucky enough to have forgotten. And in the worst case H2 just receive several extra copies of the packet it should have got. Not a big deal? Is this issue mentioned in the manual or elsewhere I failed to notice, or is it something I should find out on my own and pay attention to in my code? Please pardon me if there are stupid mistakes or misunderstandings in this “further thinking”.