**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.

1. **实验内容**
   1. Task 2: Basic Switch

implement the logic in the flowchart using the Switchyard framework.

* 1. Task 3: Timeouts

Remove an entry from the forwarding table after 10 seconds have elapsed.

* 1. 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.)

* 1. Task 5: Least Traffic Volume

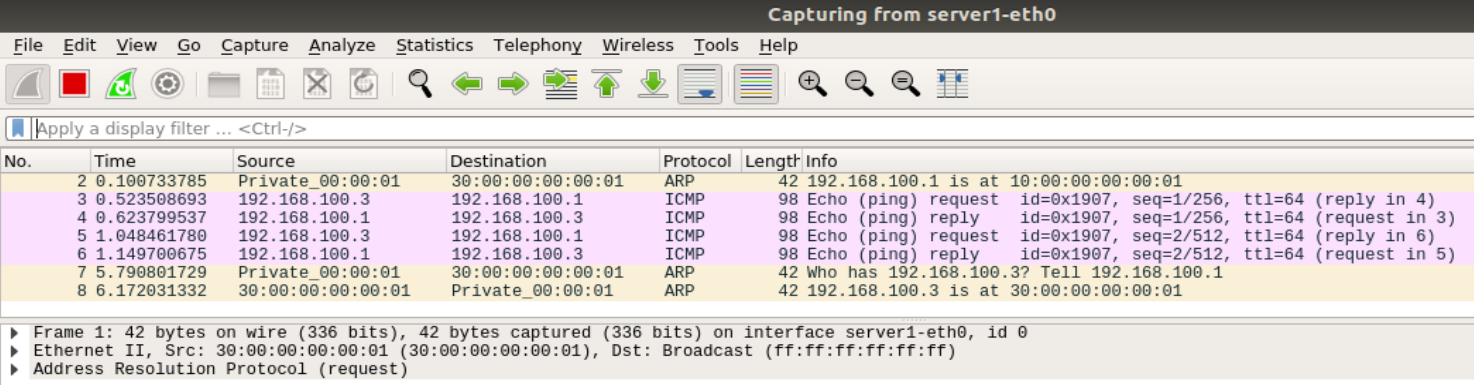
Remove the entry that has the least traffic volume.

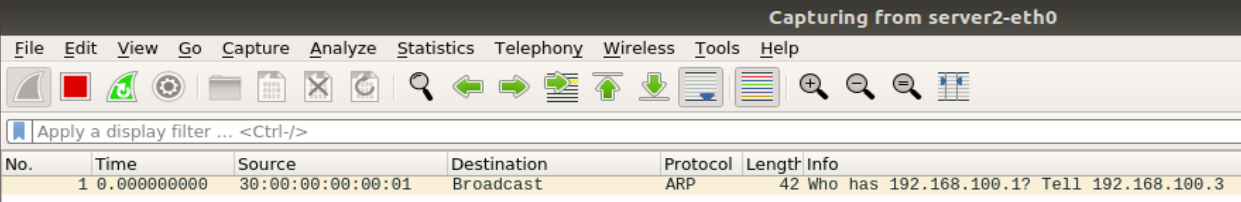
(For this functionality assume the table can only hold 5 entries. Traffic volume for an entry is the number of frames that the switch received where Destination MAC address == MAC address of the entry.)

1. **实验结果**
   1. 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).

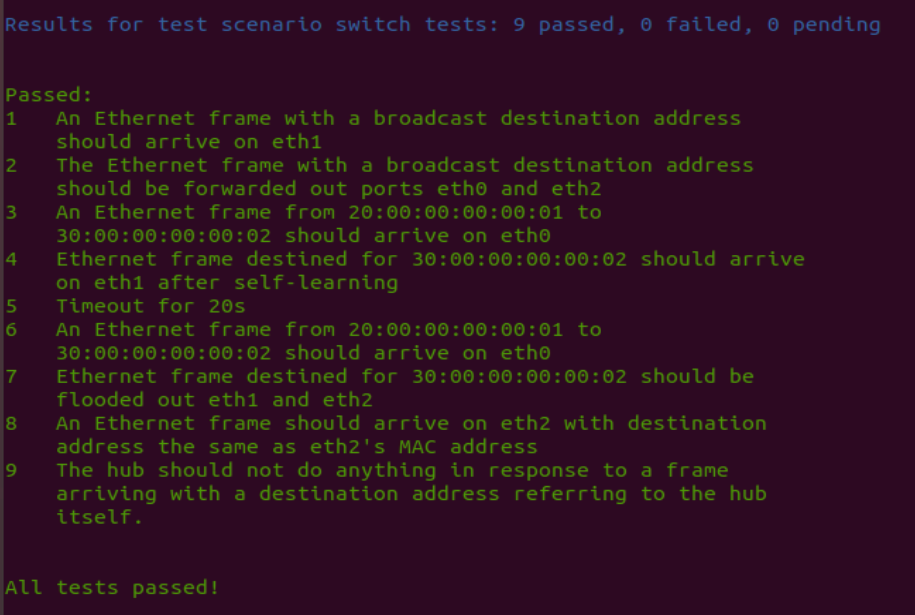
There is no echo request or reply packets in Wireshark on server2, but there are ARP packets, since they are sent with broadcast destination addresses).



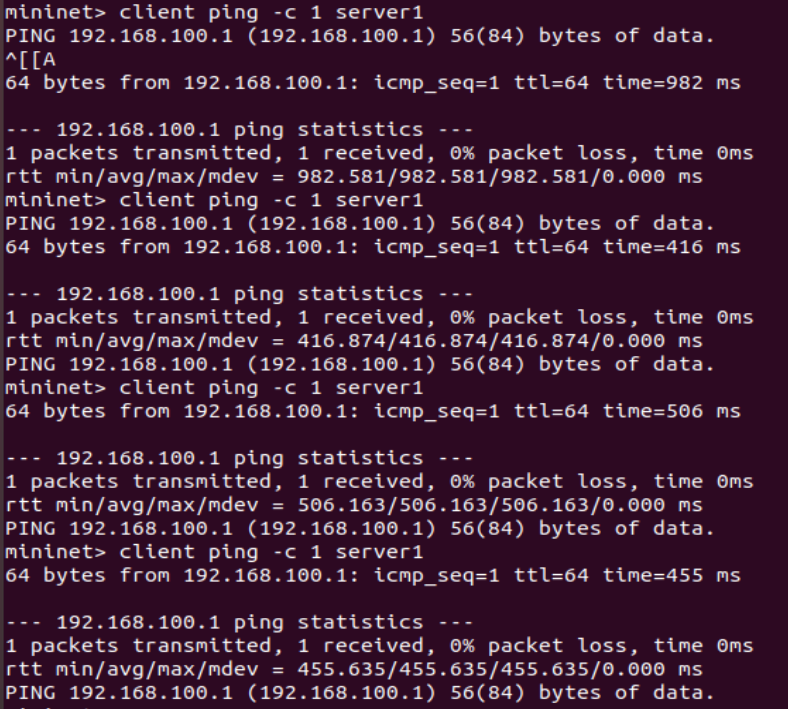


* 1. Task 3: Timeouts

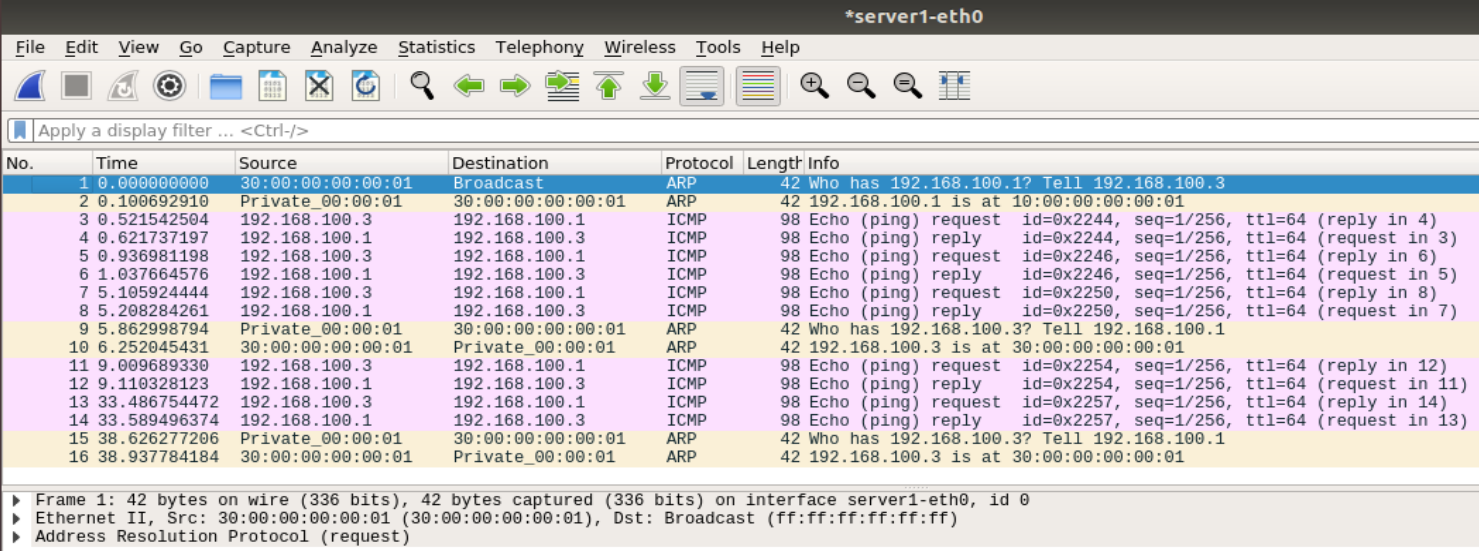
I chose not to write the test files myself. The result of the provided test file for the timeout mechanism are presented as below.

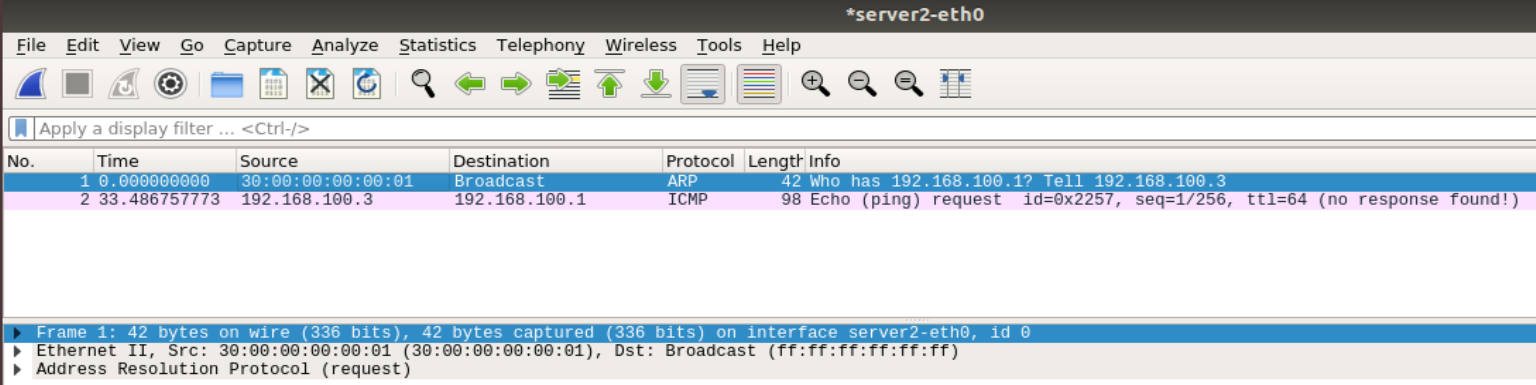


Testing my timeout mechanism in Mininet:



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:





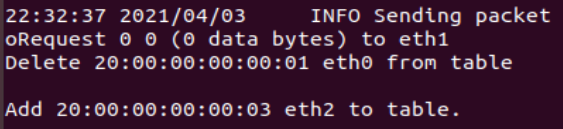
It can be inferred from the second screenshot that at the first time client pinged server1, it was broadcasted to both servers. The second and third time server2 received no packets, for the switch still “remembers” where server1 was. The fourth time the switch had already “forgotten” which port can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

* 1. Task 4: Least Recently Used

Testing my switch with the provided test file.

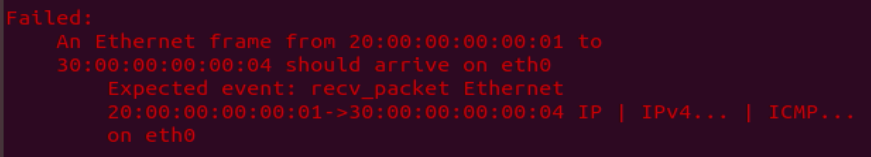


\* I changed the TABLE\_SIZE from 2 to 5 using the same test file, with 2 and 3 it failed and with 4 and 5 it passed. I tried adding some log info to see which entries was deleted in each case that led to the difference:



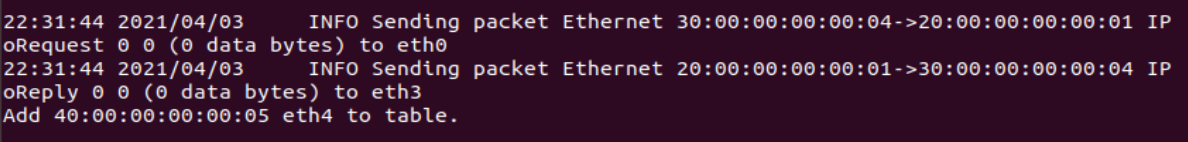
And… After testcase No. 8…



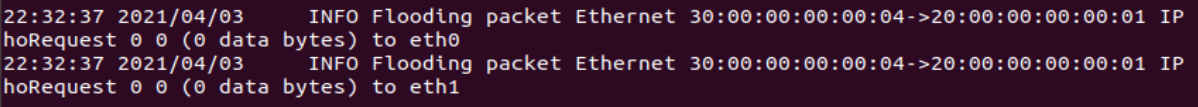


Perhaps the table size is too small that 20:…:01 eth0 was deleted from the table before 30:…04 sends to 20:…01, therefore, flooding is used instead of sending directly, thus resulting in the error.

Log info for testcase No. 8-9 when TABLE\_SIZE = 5:



Log info for testcase No. 8 when TABLE\_SIZE = 2:



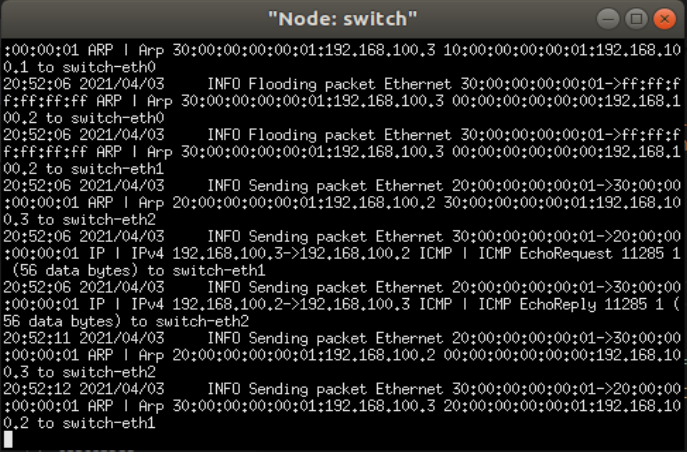
Hopefully it is somewhere near the correct explanation.

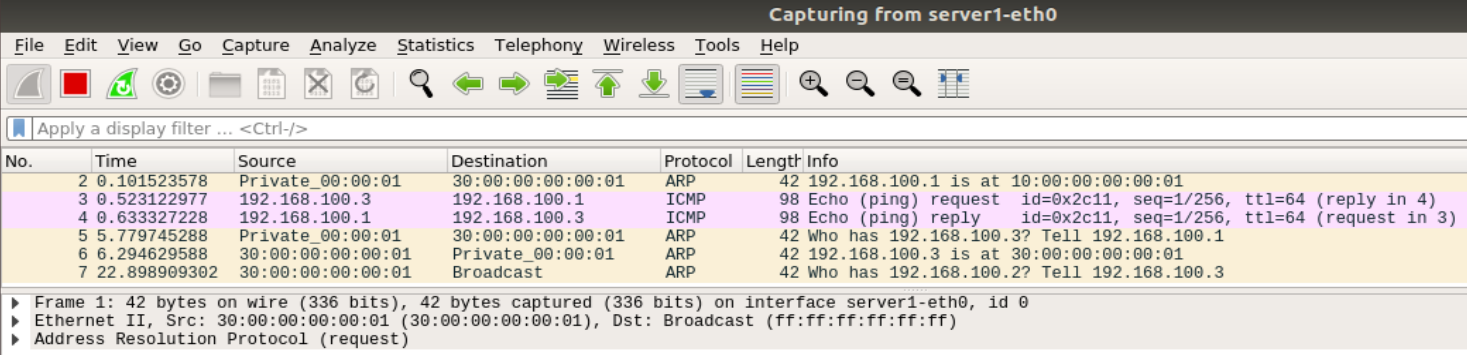
Running my switch in Mininet: client ping -c 1 server1

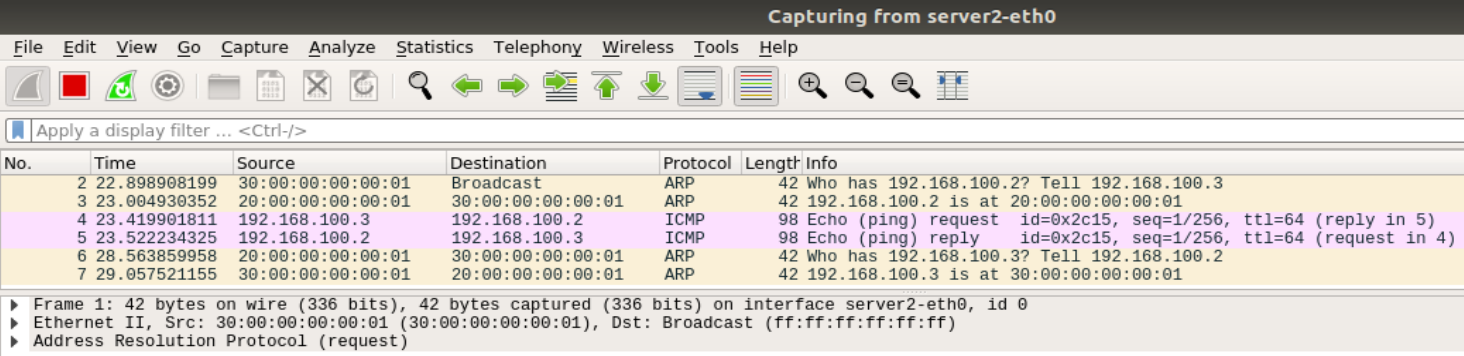
client ping -c 1 server2

(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.)









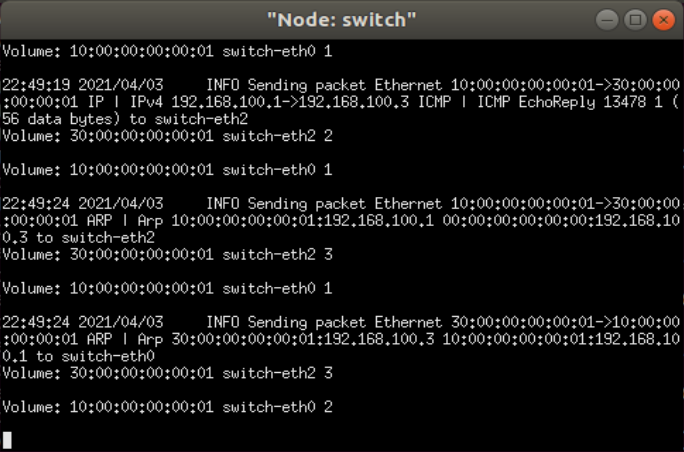
It can be inferred from the first two screenshots that after pinging server1 there are already two entries in the table (10 and 30), when client pings server2 one of them (30) is deleted to make room for a new entry (20).

* 1. Task 5: Least Traffic Volume

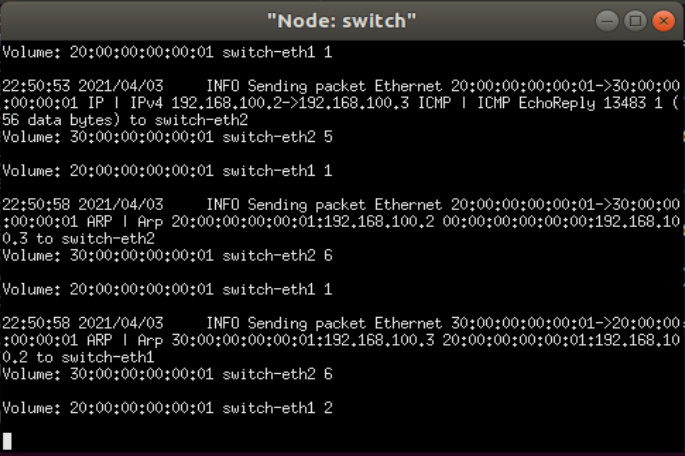
Testing my switch with the provided test file.



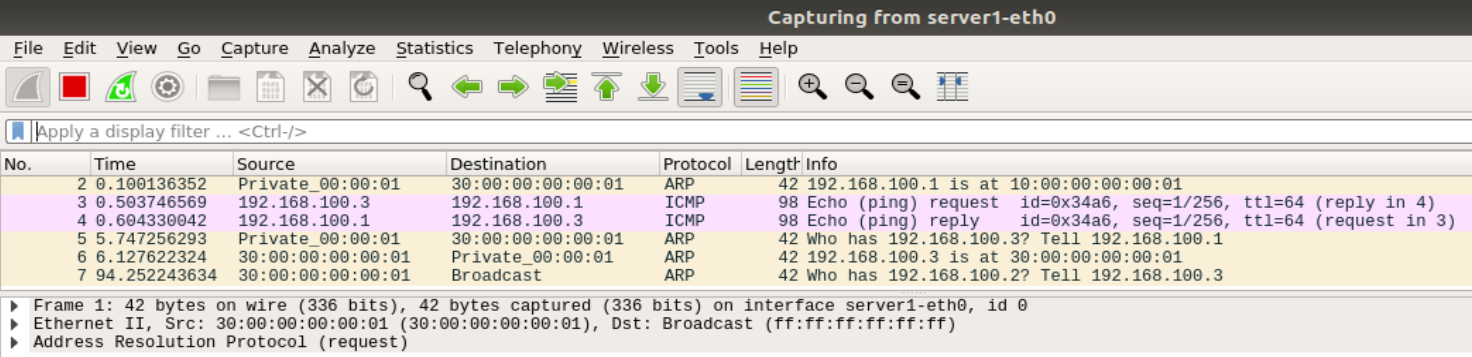
Running my switch in Mininet: client ping -c 1 server1

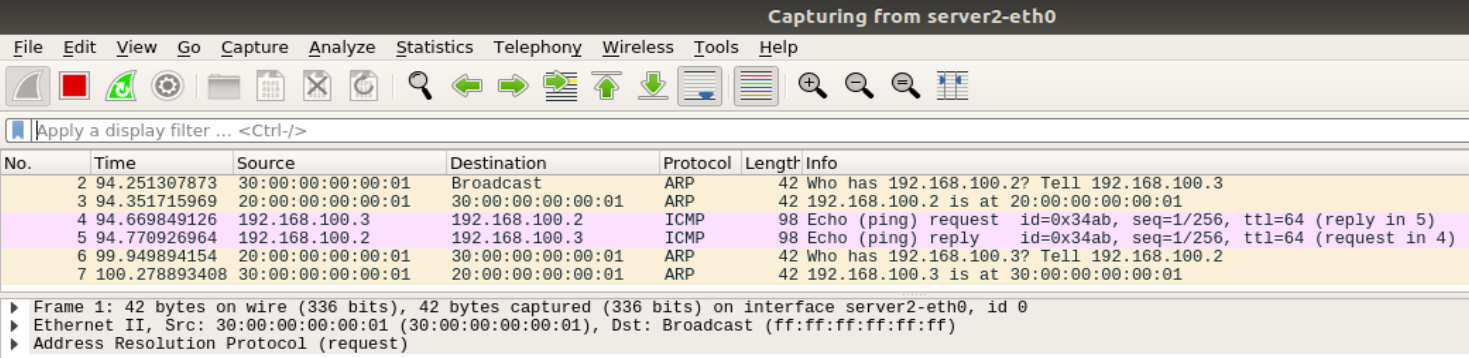


client ping -c 1 server2



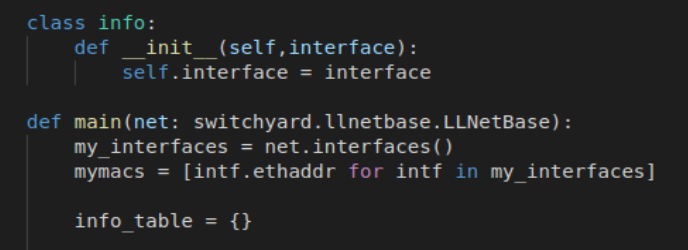
In the two screenshots above I printed out log info to see the traffic volume of the entries currently stored in the info table.





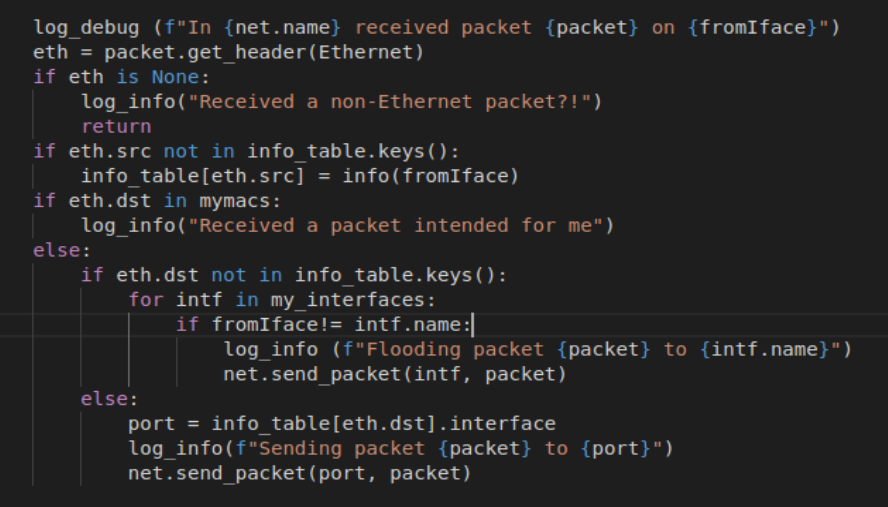
These are the captured files on sever1 and server2.

1. **核心代码**
   1. Task 2: Basic Switch



\* 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.

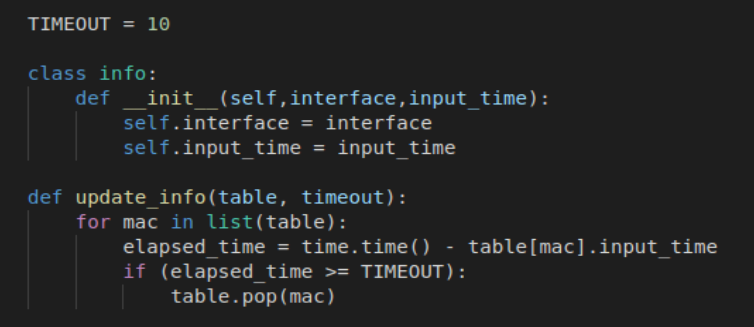


\* When a switch receives a new packet, it looks for the packet’s source address in the info table, if not found, the address and interface is added to the table.

\* 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:

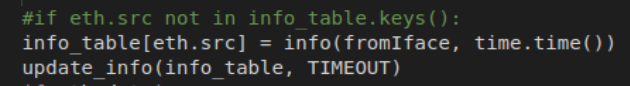
\* When an Ethernet frame arrive on any port, the switch processes its header to find the destination host. If it does not know where the host is (the address is not found in the table), it floods the frame out of all ports except the input port. Otherwise, it sends out the frame from the port according to the table.

* 1. Task 3: Timeouts



\* 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.

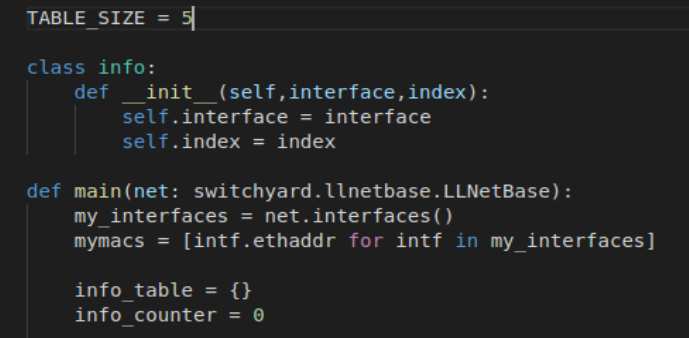


\* If a packet’s source address is NOT in the info table, it is added, and if it IS in the table, its timestamp is updated, so the “if” statement is no longer needed to judge whether the address is already in the table or not.

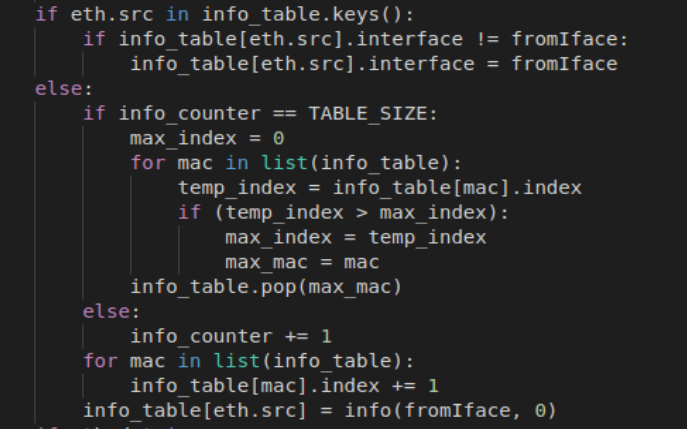
\* The info table is updated every time the switch receives a new packet.

* 1. Task 4: Least Recently Used

\* The code simply follows the logic in the flowchart.



\* 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.



\* 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.

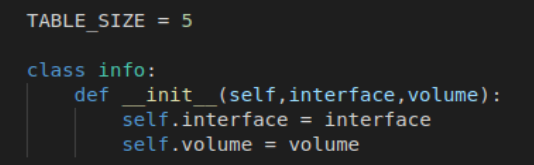


\* If the destination entry exists, update it to be the MRU entry.

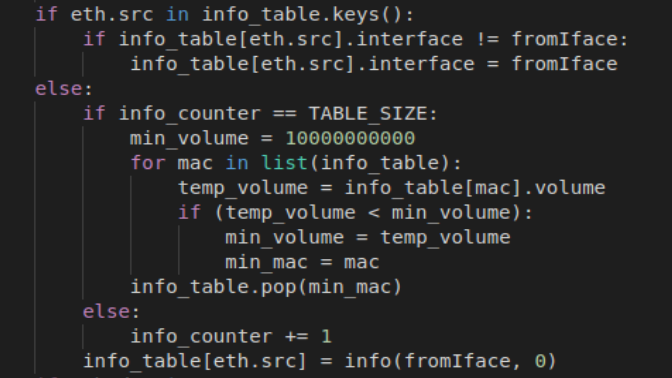
\* Which involves modifying dictionary items again, another nightmare.

* 1. Task 5: Least Traffic Volume

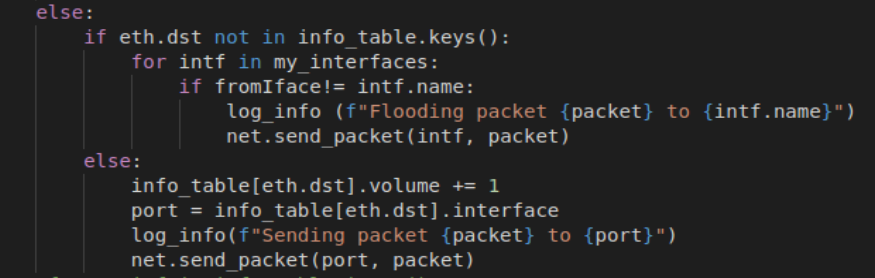
This phase is pretty much the same to Task 4.



\* The class, now with interface and traffic volume.



\* 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.

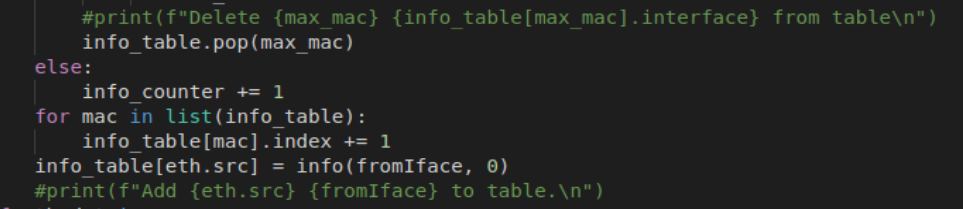


\* Updating the volume if new packet sent from the stored port.

\* Finally, almost over!!!

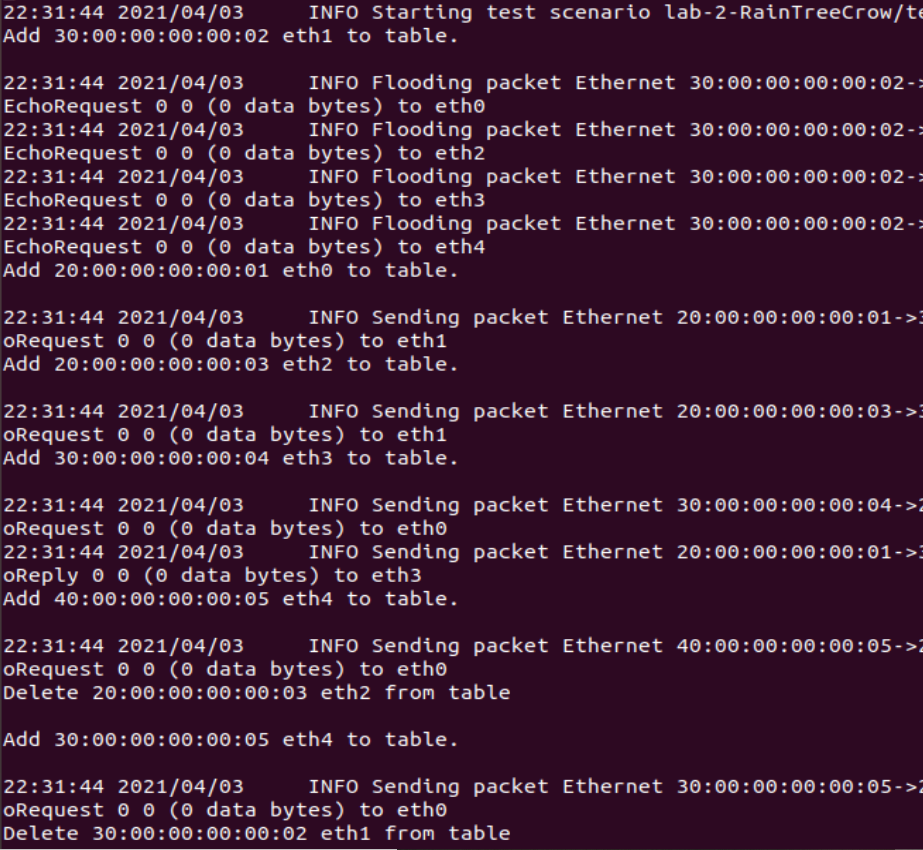
1. **总结与感想**
   1. I remember using the function dict.has\_key() before, but it seems not available here and would cause errors, perhaps it is due to the version of Python? Then I switched to using if \_ in \_\_ and if \_ not in \_\_ instead.
   2. My knowledge of Python is so limited and outdated that 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… ☹
   3. 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.
   4. 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.

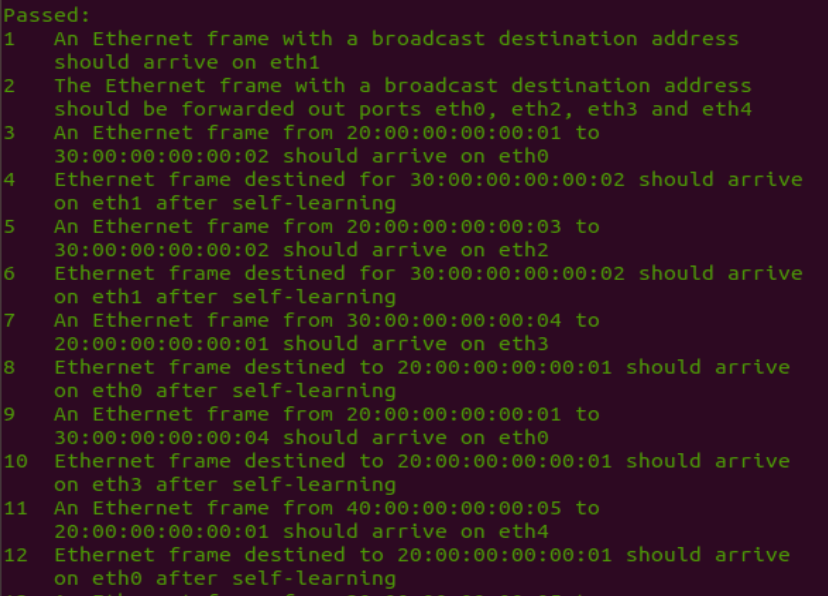




Here are some of the results:

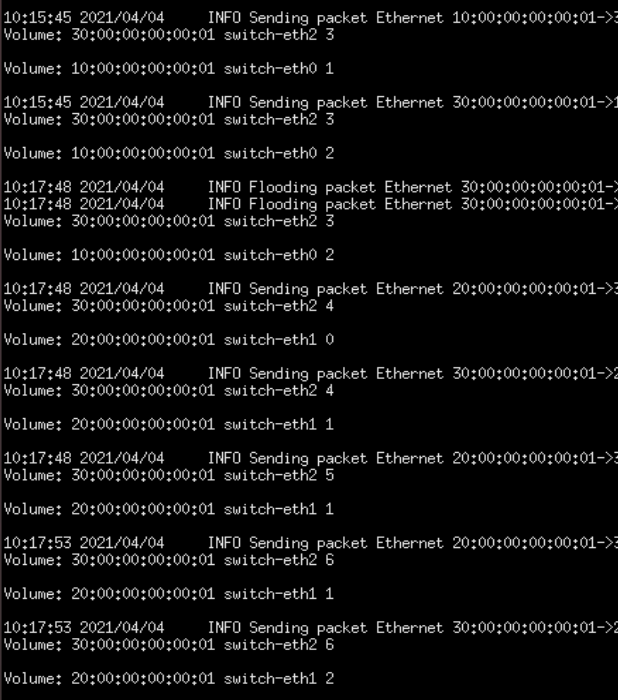


For TABLE\_SIZE = 5, there are 5 “add”s before the first “delete”.



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.



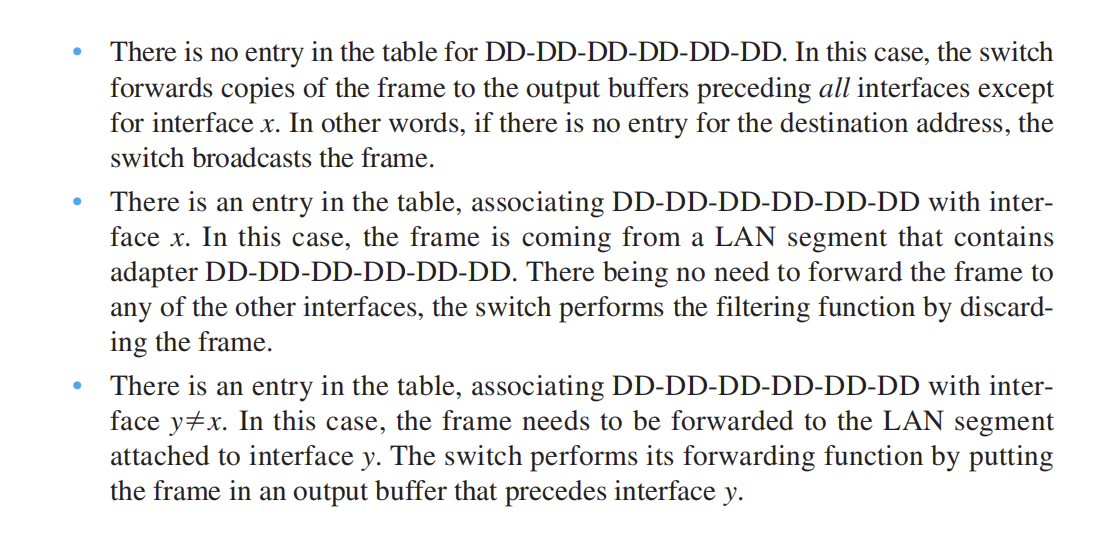
If client pings server1 and sever2 in order, the switch first learns how to reach client, then when sever1 replies it learns about server1, and the port to clients adds a volume. Other packets are treated similarly. When client pings server2 and server2 replies, the table (with size 2) is full and the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

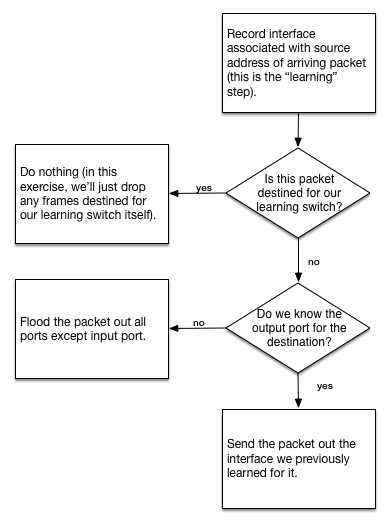
\*\*\* Please go on reading the next page. \*\*\*

\*\* 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:

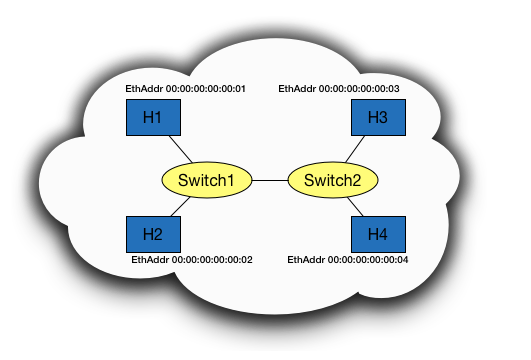


I read the lab manual over and over again, and I think it only deals with the case where the packet is destined for our learning switch, but not the second case above where the interface found in the table is the very interface the packet comes from.



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”.