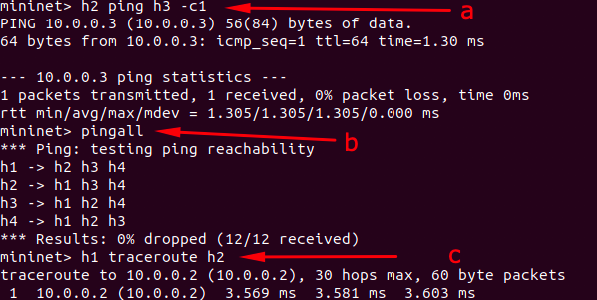
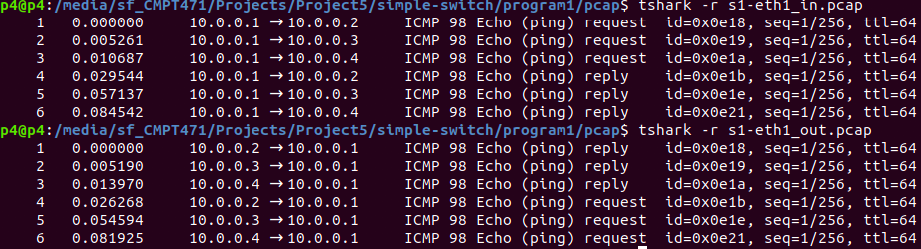
Note – Please read section 3 of this report, as it describes some problems we had with III.3 and how to test it properly given the constraints.

1.a

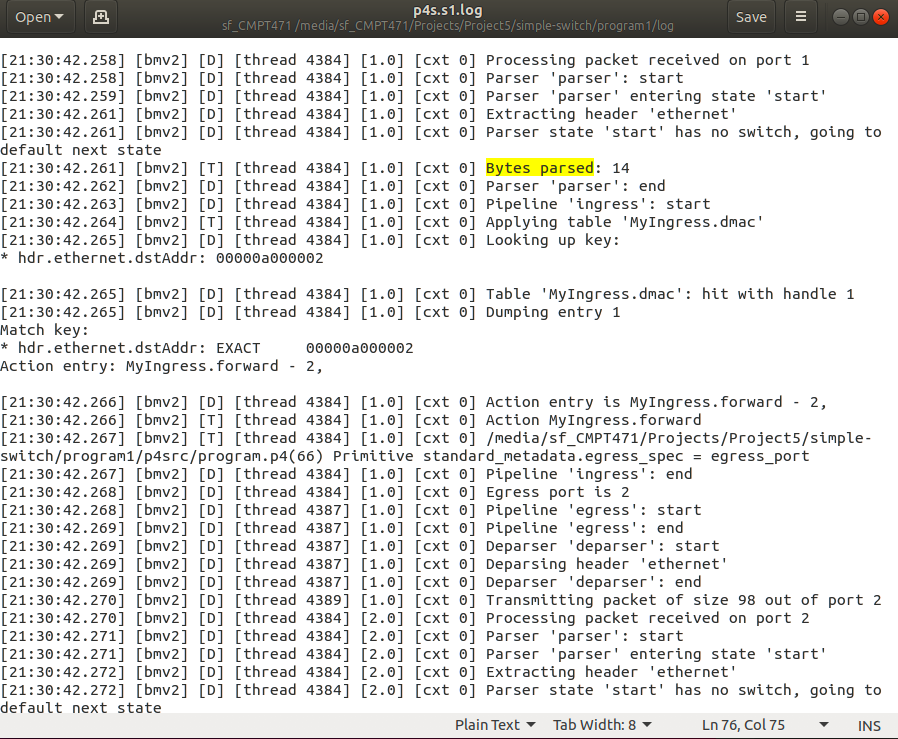


1.b

s1-eth1\_in.pcap captures all the packets that are ingress with respect to port s1-eth1. I.e., they are being sent from h1 into s1 via s1-eth1. s1-eth1\_out.pcap captures all the packets that are egress with respect to port s1-eth1. I.e., the packets are being sent from s1 outwards towards h1 via s1-eth1.



1.c



28 total bytes were parsed during the ping process – 14 bytes parsed for each request and response packet, sent to and from h2. We know this due to 2 “Bytes Parsed: 14” log entries that were populated in the log file. There was an additional Bytes Parsed entry, but this occurred before we initiated the ping process, so these bytes were not considered. The reason why only 14 bytes were parsed is because only the ethernet header is parsed in the pipeline, as is apparent by checking program.p4.

The packet size is 98. After the Bytes Parsed log entry, there is a “transmitting packet of size 98 out of port X” log entry, which reveals the packet size to us. This entry appears twice, corresponding to the request and reply packets sent throughout the ping process. The packet includes more bytes than just the ethernet header, hence why the packet size is larger than the amount of bytes processed.

2.a

Text

Description automatically generated

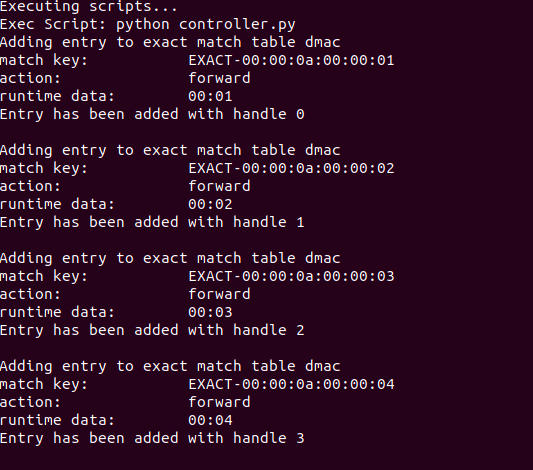
2.b

The commands s1-commands.txt are used to add entries to the table. For example,

table\_add dmac forward 00:00:0a:00:00:01 => 1

adds a match + action rule that forwards all packets with the described destination MAC address to output port 1.

2.c



The switch accepted both entries because we never explicitly prohibited the acceptance of duplicate entries by the switch. The switch itself does not create a new entry in the table for the duplicate entries as it detects that they are indeed duplicates.

We could modify the fill\_table function to query the switch and verify whether the table entry that we are attempting to add has already been added. If the entry is a duplicate, we can relay an error message.

# I.2

1.

a) This will not result in a compilation error, however, there will be error strings that are printed to notify the programmer. The switch will not add any match + action rules sent by the controller since the table is never applied in the pipeline process.

b) Text

Description automatically generated

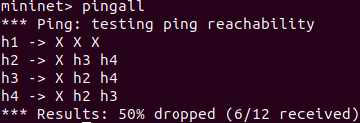
c) The ping command reveals that the packets sent out during the ping process were lost. Comparing the 2 log files (subtitled ‘ping good’ and ‘ping bad’, where ‘ping good’ does not have the dmac.apply line commented out), we can see that during the ingress phase, the dmac table is not applied. The egress port for the unsuccessful ping log file is 0, which means that the packet is simply dropped. This makes sense, as we never apply the table, so the switch by default drops the packet.

d) the dmac table is never populated. Since we uncommented out the dmac.apply() line of code, attempting to add an entry to the table will always end in failure. From the screenshot below, we see that attempting to add entries to dmac, as well as attempting to dump the dmac table, will simply write an error string to the terminal.

Text

Description automatically generated

2.

a) 

b) h1 sent 3 packets, and received 0 packets. This can be seen by inspecting s1-eth1-in-no-entry.pcap (which shows 3 packets received from source h1) and s1-eth1-out-no-entry.pcap (which shows 0 packets forwarded to h1)

c) h1 is able to send pings to h2, h3, and h4 just fine, since the match + action entries for their MAC addresses are populated in the dmac table. However, since there is no match + action entry for h1’s MAC address, the switch drops any packet that has h1’s MAC address as the destination MAC on the packet. The log reveals this fact (see p4s.s1.no-entry.log) – we see that looking up the key for 10:00:00:0a:00:00:01 misses, so the switch defaults to “noAction”. Thus, h1 never receives any packets on this network.

# III.3

We encountered an issue that significantly hindered our progress. Registers arrays have a limit of 64 bits, so the solution does not actually work for the default example of a 32 bit pattern width and flow packet threshold of 5 (as 160 bits is above the limit). Thus, concatenating the pattern strings results in an overflow in the p4 data structures. With this constraint in mind, we continued on with the natural solution but we used a bit pattern of 8 rather than 32 to test the implementation. Logically, the code is correct, it just will not work whenever flow bit width is larger than 64. I understand that this means it will likely fail a number of test cases – we are hoping that partial marks will be given, because the solution we provided is logically correct, but is bounded by the limitations of registers in p4. We genuinely gave it our best effort to come up with a workaround (even posting a detailed question outlining the issue on Piazza and discussing with other classmates), but no elegant and feasible solution came to us, so we decided to settle with the current solution. Here is the Piazza question: <https://piazza.com/class/l7o91kb2rlm6y9/post/187>