OpenFlow - Setup, Learning Switch and Control a Slice of a Real Network

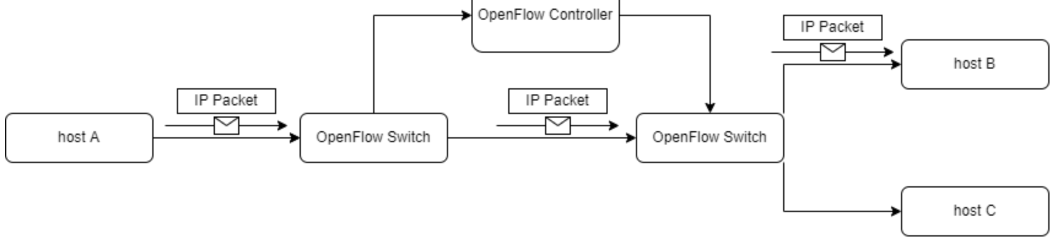
1. INTRODUCTION

As the demand of low-latency networks in latency-sensitive applications has been increasingly high today, many solutions to lowering network latency have been developed by the computer scientists and software engineers in the field of computer networks correspondingly to satisfy the various needs of use. This project is divided into three sections: explore development tools, create a learning switch, and controll a slice of a real network. After the in-depth exploration of tools and the development of a learning switch, the machine will be able to control a real network and reduce its latency using our controller.

2. BACKGROUND AND RELATED WORK

2.1 Background

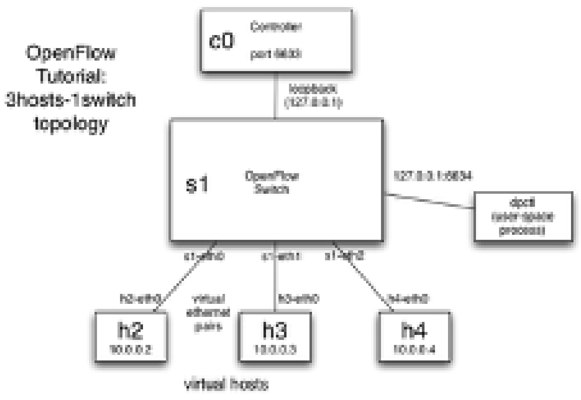
The OpenFlow that we used in this project is a communication protocol that is designed to manage and direct the traffic among routers and switches. Here is the logic of how OpenFlow forwards a packet from the source host to the destination host. Now assume host A wants to send a packet to host B and the flow table in the OpenFlow controller is empty. The first step is that the OpenFlow switch would receive the packet from host A and use the information in the packet header to find the flow from the flow table. Since the flow table is empty, the switch does not know where to direct the packet in the next step, it would ask the OpenFlow controller by sending a packet-in message. The controller would decide the routing from host A to host B and send two messages, a FlowMod message, and a packet-out message. The FlowMod message contains the information of the flow to the OpenFlow Switch. The switch would build the flow table based on the FlowMod message. The packet-out message tells the switch to send the packet based on the new flow table. After these steps, the OpenFlow switch would finally send the packet to host B.



2.2 Related Work

Our project is running on top of Mininet, which creates a virtual network and virtual VM machine to run real switches, routers and programming code.
The command of “sudo mn” creates a real virtual network to let us interact with Hosts and Switches, as well as run programming codes on this configured network. In this case we enter

“sudo mn --topo single,3 --mac --switch ovsk --controller remote” in our mininet, we start our minimal topology that comprises 3 virtual hosts with separate IP, one openflow software switch and a single software-defined-networking controller.



The OpenFlow switch determined aboved with “--switch ovsk” has a default empty flow table, so it cannot process and redirect any messages between our hosts now. To enable utilities of the switch, command “ovs-ofctl” controls over the switch and make it visible. In another terminal “ovs-ofctl dump-flows tcp:{ip address}:{port}” starts the controller to operate the switch. From the controller side, the command of “ovs-ofctl add-flow s1 in_port={},actions=output:{}” manually installs details for the flow table of the switch. Now that hosts can successfully communicate through OpenFlow Switch, following its installed flow table. If we want to see the inside to the operating principle, “ovs-ofctl dump-flows” is a very useful tool to make the flow table visible.

To run interactive commands, test connectivity and delay between hosts, we can do ping test from one host to another, and display the delay time results and success rates on our terminal. Another approach is using xterm, which brings a more clear view for our connectivity result. In our case, we do “xterm h1 h2 h3” to open three separate windows for each host. Another useful command we used is “tcpdump”, a utility to print packets seen by a host, which can let us watch debug output in separate windows.

3. Implementation

3.1 Create a Learning Switch

3.1.1 Code/Algorithm

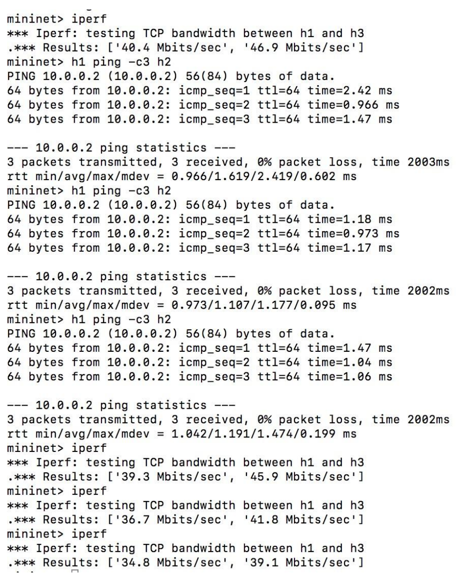
When the controller receives a packet, it would first check whether or not the port associated with the destination MAC of the packet is known. If the destination is not known, it would resend the packet to all the ports. Otherwise, it would create and send out a FlowMod message. For the FlowMod message, it uses the method of.ofp_match.from_packet(packet) to write the flow match structure and msg.actions.append(...) to add the action to tell the switch what to do for the next step. After inserting all the required information into the FlowMod message, it would send out the message by using self.connection.send(msg).

```
def act_like_switch (self, packet, packet_in):
   if (packet.src not in self.mac_to_port):
     self.mac_to_port[packet.src] = packet_in.in_port
   if (packet.dst in self.mac_to_port):
     log.debug("Installing flow...")
     msg = of.ofp_flow_mod()
     msg.match = of.ofp_match.from_packet(packet)
     msg.actions.append(of.ofp_action_output(port = self.mac_to_port[packet.dst]))
     self.connection.send(msg)
   else:
     self.resend_packet(packet_in, of.OFPP_ALL)
```

3.1.2 Instruction To Run and Test Result

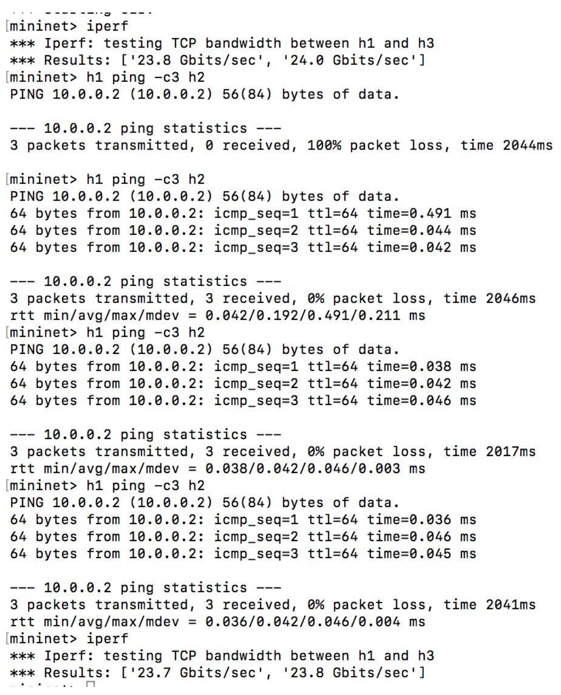
In terminal A: sudo mn --topo single,3 --mac --switch ovsk --controller remote
In terminal B: ovs-ofctl dump-flows tcp:{ip address}:{port}
ovs-ofctl add-flow s1 in_port=1,actions=output:2
ovs-ofctl add-flow s1 in_port=2,actions=output:1
./pox.py log.level --DEBUG misc.of_tutorial
In terminal A: by running iperf and h1 ping -c3 h2, should show the following result

Referenced switch controller:
Bandwidth: about 35 - 45 Mbits/s
Ping delay: 1.0 - 1.2 ms on average (except the first time)



```
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
.*** Results: ['40.4 Mbits/sec', '46.9 Mbits/sec']
mininet> h1 ping -c3 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=2.42 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.966 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=1.47 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.966/1.619/2.419/0.602 ms
mininet> h1 ping -c3 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=1.18 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.973 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=1.17 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 0.973/1.107/1.177/0.095 ms
mininet> h1 ping -c3 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=1.47 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=1.04 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=1.06 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 1.042/1.191/1.474/0.199 ms
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
.*** Results: ['39.3 Mbits/sec', '45.9 Mbits/sec']
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
.*** Results: ['36.7 Mbits/sec', '41.8 Mbits/sec']
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
.*** Results: ['34.8 Mbits/sec', '39.1 Mbits/sec']
```

Implemented learning switch:
Bandwidth: about 23 - 24 Gbits/sec
Ping delay: 0.04 ms on average (except the first time)



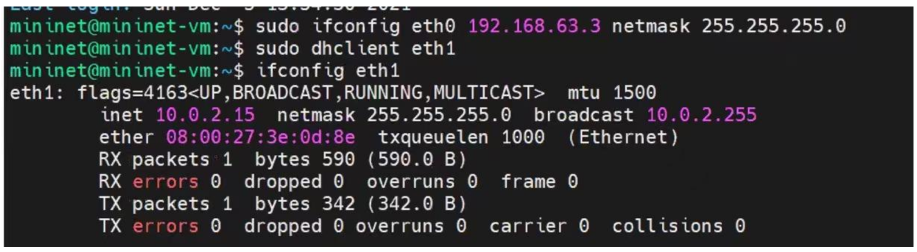
```
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['23.8 Gbits/sec', '24.0 Gbits/sec']
mininet> h1 ping -c3 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2044ms

mininet> h1 ping -c3 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.491 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.044 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.042 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2046ms
rtt min/avg/max/mdev = 0.042/0.192/0.491/0.211 ms
mininet> h1 ping -c3 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.038 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.042 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.046 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2017ms
rtt min/avg/max/mdev = 0.038/0.042/0.046/0.003 ms
mininet> h1 ping -c3 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.036 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.046 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.045 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2041ms
rtt min/avg/max/mdev = 0.036/0.042/0.046/0.004 ms
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['23.7 Gbits/sec', '23.8 Gbits/sec']
```

In terminal A: xterm h1 h2 h3
In window for h2: tcpdump -XX -n -i h2-eth0
In window for h3: tcpdump -XX -n -i h3-eth0
In window for h1: ping -c1 10.0.0.2

The ping result should be similar to that in terminal A, comparing referenced switch and our learning switch behaviour, we can observe that subsequent pings in learning switch complete with a smaller delay, and its iperf command shows a significantly higher bandwidth. The result is consistent with the description from “Testing your Controller” in https://github.com/mininet/openflow-tutorial/wiki/Create-a-Learning-Switch#Testing_Your_Controller.

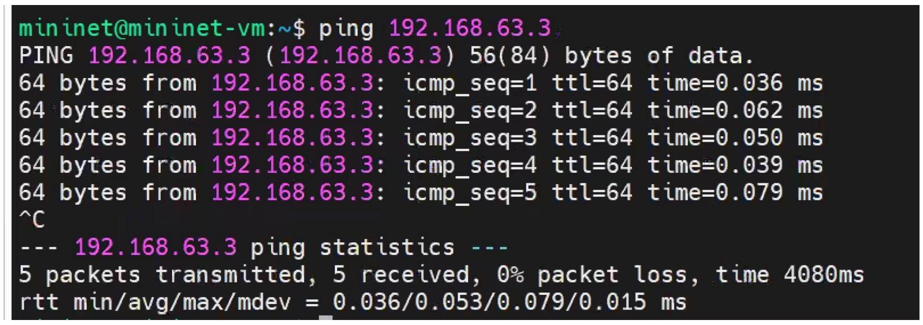
*3.2 Control a Slice of a real Network

At mininet, configure the interface



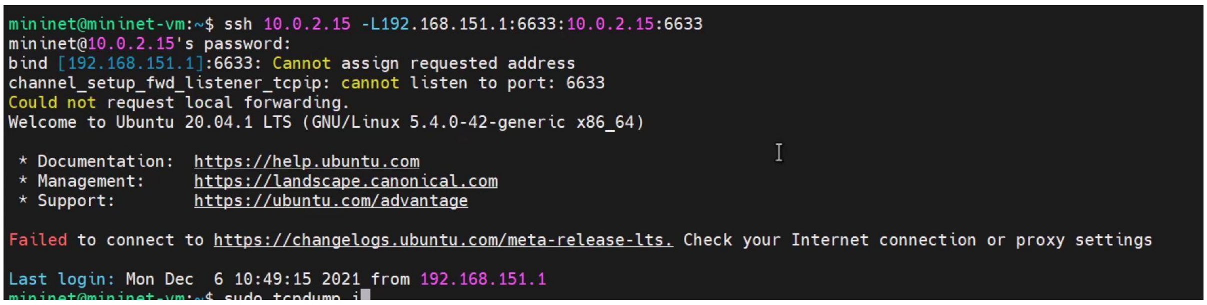
```
mininet@mininet-vm:~$ sudo ifconfig eth0 192.168.63.3 netmask 255.255.255.0
mininet@mininet-vm:~$ sudo dhclient eth1
mininet@mininet-vm:~$ ifconfig eth1
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
        ether 08:00:27:3e:0d:8e  txqueuelen 1000  (Ethernet)
        RX packets 1  bytes 590 (590.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1  bytes 342 (342.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Ping FlowVisor from mininet:



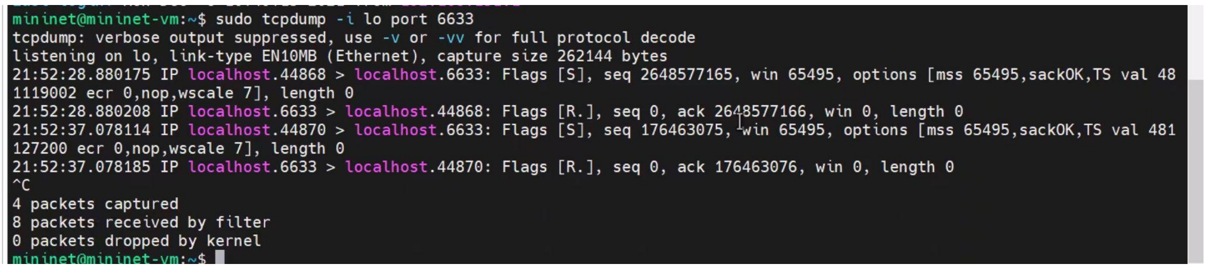
```
mininet@mininet-vm:~$ ping 192.168.63.3
PING 192.168.63.3 (192.168.63.3) 56(84) bytes of data.
64 bytes from 192.168.63.3: icmp_seq=1 ttl=64 time=0.036 ms
64 bytes from 192.168.63.3: icmp_seq=2 ttl=64 time=0.062 ms
64 bytes from 192.168.63.3: icmp_seq=3 ttl=64 time=0.050 ms
64 bytes from 192.168.63.3: icmp_seq=4 ttl=64 time=0.039 ms
64 bytes from 192.168.63.3: icmp_seq=5 ttl=64 time=0.079 ms
^C
--- 192.168.63.3 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4080ms
rtt min/avg/max/mdev = 0.036/0.053/0.079/0.015 ms
```

Use port-forwarding to ssh into the IP address of eth1, so that controller is able to connect FlowVisor.



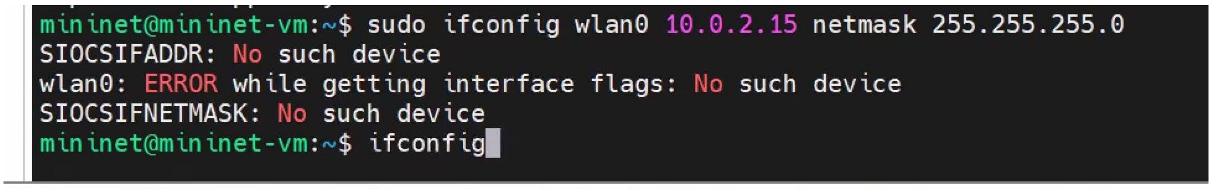
```
mininet@mininet-vm:~$ ssh 10.0.2.15 -L192.168.151.1:6633:10.0.2.15:6633
mininet@10.0.2.15's password:
bind [192.168.151.1]:6633: Cannot assign requested address
channel_setup_fwd_listener_tcpip: cannot listen to port: 6633
Could not request local forwarding.
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-42-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

Failed to connect to https://changelogs.ubuntu.com/meta-release-lts. Check your Internet connection or proxy settings

Last login: Mon Dec  6 10:49:15 2021 from 192.168.151.1
mininet@mininet-vm:~$ sudo tcpdump -i
```

In mininet, use tcpdump to verify reachability to FlowVisor, here we receive incoming packets.

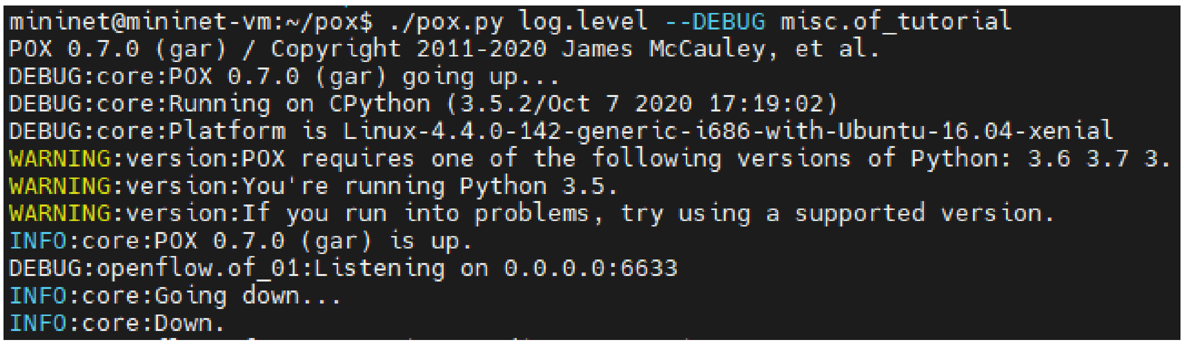


```
mininet@mininet-vm:~$ sudo tcpdump -i lo port 6633
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
21:52:28.880175 IP localhost.44868 > localhost.6633: Flags [S], seq 2648577165, win 65495, options [mss 65495,sackOK,TS val 48
1119002 ecr 0,nop,wscale 7], length 0
21:52:28.880208 IP localhost.6633 > localhost.44868: Flags [R.], seq 0, ack 2648577166, win 0, length 0
21:52:37.078114 IP localhost.44870 > localhost.6633: Flags [S], seq 176463075, win 65495, options [mss 65495,sackOK,TS val 481
127200 ecr 0,nop,wscale 7], length 0
21:52:37.078185 IP localhost.6633 > localhost.44870: Flags [R.], seq 0, ack 176463076, win 0, length 0
^C
4 packets captured
8 packets received by filter
0 packets dropped by kernel
mininet@mininet-vm:~$
```

Here we don't have the interface "wlan0", so we fail to configure our wireless network interface.



```
mininet@mininet-vm:~$ sudo ifconfig wlan0 10.0.2.15 netmask 255.255.255.0
SIOCSIFADDR: No such device
wlan0: ERROR while getting interface flags: No such device
SIOCSIFNETMASK: No such device
mininet@mininet-vm:~$ ifconfig
```



```
mininet@mininet-vm:~/pox$ ./pox.py log.level --DEBUG misc.of_tutorial
POX 0.7.0 (gar) / Copyright 2011-2020 James McCauley, et al.
DEBUG:core:POX 0.7.0 (gar) going up...
DEBUG:core:Running on CPython (3.5.2/Oct 7 2020 17:19:02)
DEBUG:core:Platform is Linux-4.4.0-142-generic-i686-with-Ubuntu-16.04-xenial
WARNING:version:POX requires one of the following versions of Python: 3.6 3.7 3.
WARNING:version:You're running Python 3.5.
WARNING:version:If you run into problems, try using a supported version.
INFO:core:POX 0.7.0 (gar) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:core:Going down...
INFO:core:Down.
```

By now, the last steps of our project seems problematic running inside of mininet instead of using a real network on our local machine. (pizza post 307 shows some of the difficulties we faced during the process)
After consulting with the professor, we decided to stop here and reflect on the problems so far we faced. Please see below <4. problems during the process> for more details.

*4. PROBLEMS DURING THE PROCESS

The third part of this project requires us to work with a real network. However, there is no prototyping "real" network that can be used to experiment and explore for this project. Due to this reason, lab machines are used instead. While using lab machines, we found that anything involving sudo privileges will not work on these machines so we tried to implement this part using Mininet. Since we are not using the network described in part three, it was quite hard for us to build the same topology as shown. When we are trying to configure the wireless client's network interface by running the command in the instruction, we need to configure a WLAN in the VM, however, this is not supported by the VM. In order to solve this issue, we were suggested to mimic this using NAT. However, we could not find the NAT network after connecting to the mininet. We also tried to use Bridged Adapter to connect to the internet on our own laptop. As a result of this, specifically, when we attempted to run the controller that we implemented with POX, which listens on the port where the guest VM tries to communicate with the FlowVisor, there were not any switches that connected to our controller. Since we failed by using POX, we changed to using NOX as described in the third part of the project instruction. First, we installed NOX from github by the link provided on piazza(https://github.com/noxrepo/nox). After this step, we found that this repo does not have the directory and the file shown in the instruction(cd ~/noxcore/build/src/) to run the code.

5. CONTRIBUTION

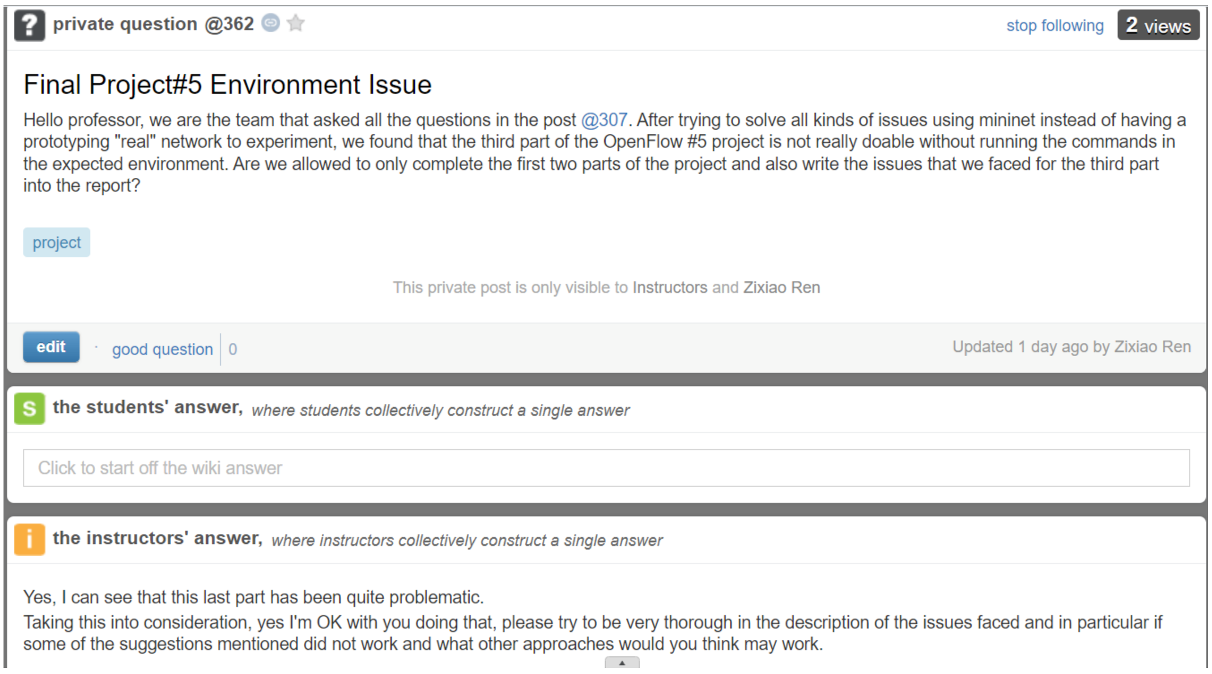
Zixiao Ren: Did some of the coding for part2 and ran all the commands to test the code for the second part. Implemented part3 and tried to solve any difficulties that we faced for this part. Wrote the section 2.1, 3.1.1, and 4 of the report.

Tongfei Li: Help implement “act_like_switch”. Wrote 2.2 related work, 3.1.2 Instruction to run and test result and 3.2 control slice of a real network in our project report.

6. REFERENCES

https://chentingz.github.io/2019/12/30/%E3%80%8COpenFlow%E3%80%8D%E5%8D%8F%E8%AE%AE%E5%85%A5%E9%97%A8/
https://github.com/mininet/openflow-tutorial/wiki/Learn-Development-Tools
https://github.com/mininet/openflow-tutorial/wiki/Create-a-Learning-Switch#Testing_Your_Controller
https://github.com/mininet/openflow-tutorial/wiki/Control-a-Slice-of-a-real-Network
https://piazza.com/class/krwhkmh7auz1x2?cid=307

*Since we did not have the expected environment for the third part of this project, we were approved to only complete the first two parts of this project and demonstrate the difficulties that we faced in part three.



private question @362

stop following 2 views

Final Project#5 Environment Issue

Hello professor, we are the team that asked all the questions in the post @307. After trying to solve all kinds of issues using mininet instead of having a prototyping "real" network to experiment, we found that the third part of the OpenFlow #5 project is not really doable without running the commands in the expected environment. Are we allowed to only complete the first two parts of the project and also write the issues that we faced for the third part into the report?

project

This private post is only visible to Instructors and Zixiao Ren

edit · good question | 0

Updated 1 day ago by Zixiao Ren

the students' answer, *where students collectively construct a single answer*

Click to start off the wiki answer

the instructors' answer, *where instructors collectively construct a single answer*

Yes, I can see that this last part has been quite problematic.
Taking this into consideration, yes I'm OK with you doing that, please try to be very thorough in the description of the issues faced and in particular if some of the suggestions mentioned did not work and what other approaches would you think may work.