

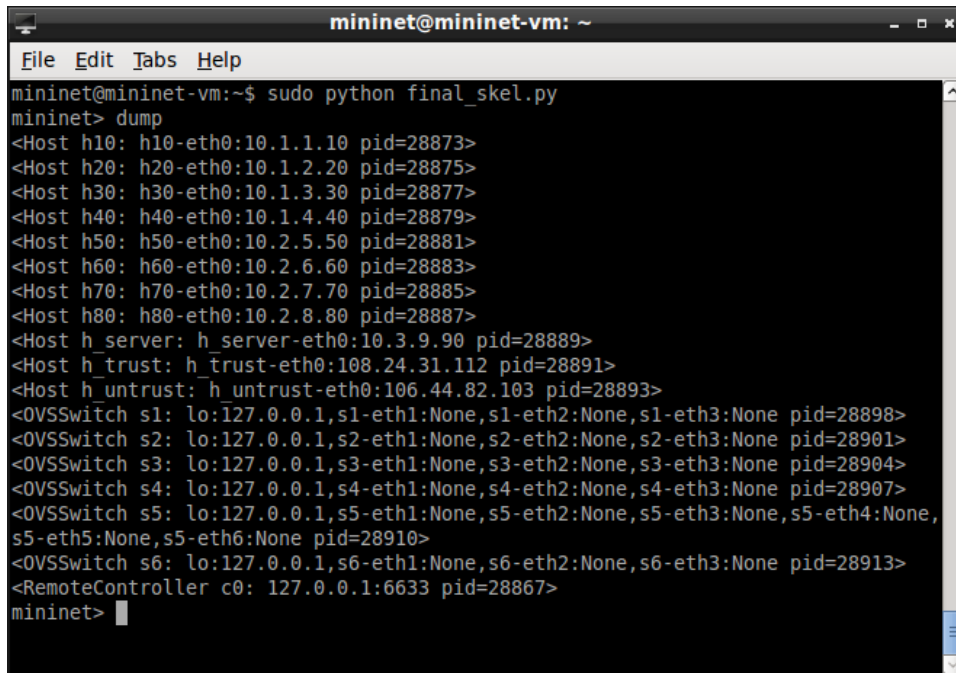
Final Project: Implementing a Simple Router

The assignment's requirements involve creating a simple network topology within Mininet, consisting of several hosts and switches, each with their respective IP addresses and connections.

Here's a comprehensive analysis of the implementation and verification of these requirements:

1. Devices are successfully created

The successful creation of the devices in the topology can be confirmed by running the `dump` command.



```
mininet@mininet-vm: ~  
File Edit Tabs Help  
mininet@mininet-vm:~$ sudo python final_skel.py  
mininet> dump  
<Host h10: h10-eth0:10.1.1.10 pid=28873>  
<Host h20: h20-eth0:10.1.2.20 pid=28875>  
<Host h30: h30-eth0:10.1.3.30 pid=28877>  
<Host h40: h40-eth0:10.1.4.40 pid=28879>  
<Host h50: h50-eth0:10.2.5.50 pid=28881>  
<Host h60: h60-eth0:10.2.6.60 pid=28883>  
<Host h70: h70-eth0:10.2.7.70 pid=28885>  
<Host h80: h80-eth0:10.2.8.80 pid=28887>  
<Host h_server: h_server-eth0:10.3.9.90 pid=28889>  
<Host h_trust: h_trust-eth0:108.24.31.112 pid=28891>  
<Host h_untrust: h_untrust-eth0:106.44.82.103 pid=28893>  
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None pid=28898>  
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None,s2-eth3:None pid=28901>  
<OVSSwitch s3: lo:127.0.0.1,s3-eth1:None,s3-eth2:None,s3-eth3:None pid=28904>  
<OVSSwitch s4: lo:127.0.0.1,s4-eth1:None,s4-eth2:None,s4-eth3:None pid=28907>  
<OVSSwitch s5: lo:127.0.0.1,s5-eth1:None,s5-eth2:None,s5-eth3:None,s5-eth4:None,s5-eth5:None,s5-eth6:None pid=28910>  
<OVSSwitch s6: lo:127.0.0.1,s6-eth1:None,s6-eth2:None,s6-eth3:None pid=28913>  
<RemoteController c0: 127.0.0.1:6633 pid=28867>  
mininet>
```

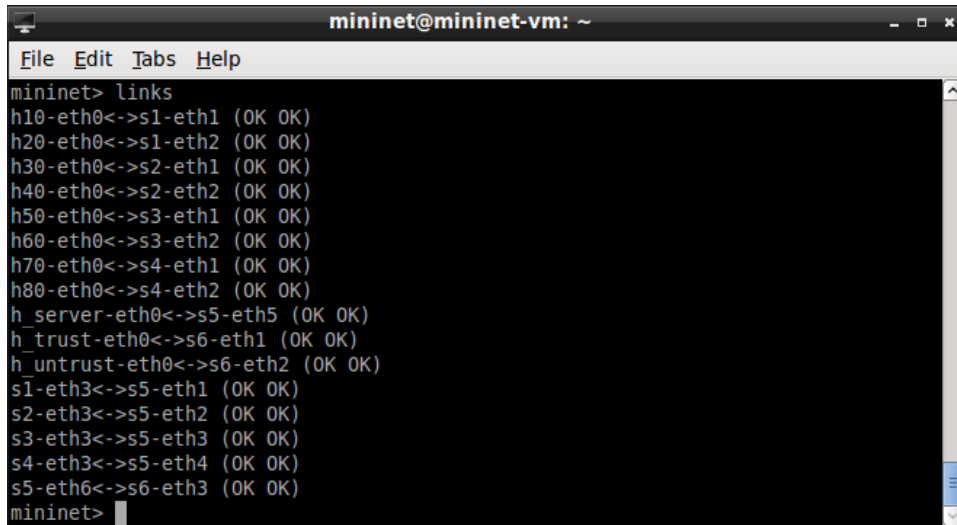
As shown in the screenshot above, there are 11 hosts and 6 switches in the network, which matches the requirements of the topology described in the assignment:

- 8 hosts, named from h10 to h80
- 2 special hosts: trusted (h_trust) and untrusted (h_untrust)
- 1 server host (h_server)
- 6 switches, named from s1 to s6

These devices also have the correct process IDs associated with them, which further indicates successful creation.

2. Links are successfully created, and the topology is correct

The successful establishment of links between the devices and the accuracy of the topology can be verified by running the `links` command.



```
mininet@mininet-vm: ~  
File Edit Tabs Help  
mininet> links  
h10-eth0<->s1-eth1 (OK OK)  
h20-eth0<->s1-eth2 (OK OK)  
h30-eth0<->s2-eth1 (OK OK)  
h40-eth0<->s2-eth2 (OK OK)  
h50-eth0<->s3-eth1 (OK OK)  
h60-eth0<->s3-eth2 (OK OK)  
h70-eth0<->s4-eth1 (OK OK)  
h80-eth0<->s4-eth2 (OK OK)  
h_server-eth0<->s5-eth5 (OK OK)  
h_trust-eth0<->s6-eth1 (OK OK)  
h_untrust-eth0<->s6-eth2 (OK OK)  
s1-eth3<->s5-eth1 (OK OK)  
s2-eth3<->s5-eth2 (OK OK)  
s3-eth3<->s5-eth3 (OK OK)  
s4-eth3<->s5-eth4 (OK OK)  
s5-eth6<->s6-eth3 (OK OK)  
mininet>
```

As shown in the screenshot above, all the hosts are correctly connected to their respective switches, and the switches are also connected as per the topology described in the assignment:

- "Floor 1, Switch 1" and "Floor 2, Switch 2", "Trusted Host 108.24.31.112/24", and "Untrusted Host 106.44.82.103/24" connected to a "Core Switch".
- "Host 10, 10.1.1.10/24" and "Host 20, 10.1.2.20/24" are connected to "Floor 1, Switch 1".
- "Host 50, 10.2.5.50/24" and "Host 60, 10.2.6.60/24" are connected to "Floor 2, Switch 1".
- "Host 30, 10.1.3.30/24" and "Host 40, 10.1.4.40/24" are connected to "Floor 1, Switch 2".
- "Host 70, 10.2.7.70/24" and "Host 80, 10.2.8.80/24" are connected to "Floor 2, Switch 2".
- "Core Switch" is connected to "Data Center Switch", which is connected to "Server 10.3.9.90/24".

The links are marked as 'OK OK', indicating that the connections are up and running correctly. There are no broken or misconnected links in the network, further establishing the correctness of the topology.

3. IP addresses are correct

```
mininet@mininet-vm: ~  
File Edit Tabs Help  
mininet@mininet-vm:~$ sudo python final_skel.py  
mininet> dump  
<Host h10: h10-eth0:10.1.1.10 pid=28873>  
<Host h20: h20-eth0:10.1.2.20 pid=28875>  
<Host h30: h30-eth0:10.1.3.30 pid=28877>  
<Host h40: h40-eth0:10.1.4.40 pid=28879>  
<Host h50: h50-eth0:10.2.5.50 pid=28881>  
<Host h60: h60-eth0:10.2.6.60 pid=28883>  
<Host h70: h70-eth0:10.2.7.70 pid=28885>  
<Host h80: h80-eth0:10.2.8.80 pid=28887>  
<Host h_server: h_server-eth0:10.3.9.90 pid=28889>  
<Host h_trust: h_trust-eth0:108.24.31.112 pid=28891>  
<Host h_untrust: h_untrust-eth0:106.44.82.103 pid=28893>  
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None pid=28898>  
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None,s2-eth3:None pid=28901>  
<OVSSwitch s3: lo:127.0.0.1,s3-eth1:None,s3-eth2:None,s3-eth3:None pid=28904>  
<OVSSwitch s4: lo:127.0.0.1,s4-eth1:None,s4-eth2:None,s4-eth3:None pid=28907>  
<OVSSwitch s5: lo:127.0.0.1,s5-eth1:None,s5-eth2:None,s5-eth3:None,s5-eth4:None,  
s5-eth5:None,s5-eth6:None pid=28910>  
<OVSSwitch s6: lo:127.0.0.1,s6-eth1:None,s6-eth2:None,s6-eth3:None pid=28913>  
<RemoteController c0: 127.0.0.1:6633 pid=28867>  
mininet> █
```

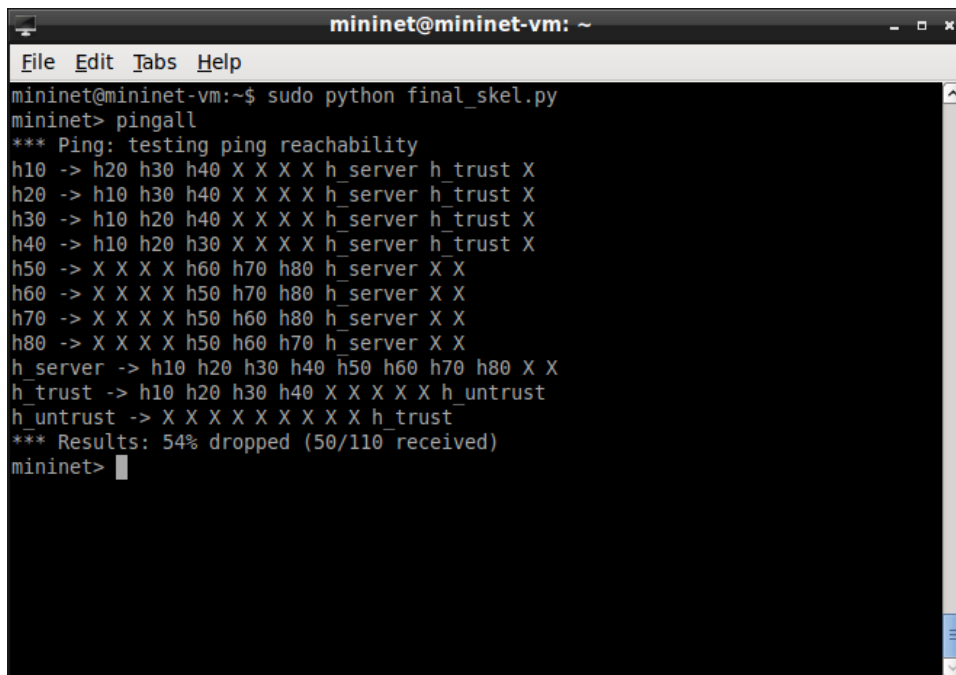
From the output of the dump command, it's clear that each host has been assigned the correct IP address as per the assignment's requirements. Refer to my final_skel.py file.

- Host h10 has an IP address of 10.1.1.10/24.
- Host h20 has an IP address of 10.1.2.20/24.
- Host h30 has an IP address of 10.1.3.30/24.
- Host h40 has an IP address of 10.1.4.40/24.
- Host h50 has an IP address of 10.2.5.50/24.
- Host h60 has an IP address of 10.2.6.60/24.
- Host h70 has an IP address of 10.2.7.70/24.
- Host h80 has an IP address of 10.2.8.80/24.
- The server has an IP address in the subnet 10.3.9.90/24.
- The trusted host has an IP address of 108.24.31.112/24.
- The untrusted host has an IP address of 106.44.82.103/24.

This confirms that the IP addresses have been correctly assigned in accordance with the assignment's guidelines.

4. Hosts can communicate

The `pingall` command shows the connectivity between all pairs of hosts in my network by sending ICMP Echo Request packets ("pings") from each host to every other host. The output of the `pingall` command demonstrates that hosts can communicate with each other, as shown by successful ICMP pings between the different host pairs. Only host pairs that I specified in my forwarding table can communicate through ICMP. Otherwise, these packets are dropped as demonstrated by each 'X's that represent an unsuccessful ping to another host, which implies that my firewall code is properly blocking specific host pairs. The hosts in the range of 10-40 can only communicate with each other and the server, and the hosts in the range 50-80 can also only communicate among themselves and with the server. As addressed by the TA Sammy Tesfai, the trusted host and untrusted host also successfully communicate with each other. The summary, `*** Results: 54% dropped (50/110 received)`, indicates that all ping packets sent were dropped and not a single one was successfully received. The number in parentheses, `(50/110)`, shows the ratio of successful ping replies to the total number of pings sent. Below is a screenshot of the output:



```
mininet@mininet-vm: ~  
File Edit Tabs Help  
mininet@mininet-vm:~$ sudo python final_skel.py  
mininet> pingall  
*** Ping: testing ping reachability  
h10 -> h20 h30 h40 X X X X h_server h_trust X  
h20 -> h10 h30 h40 X X X X h_server h_trust X  
h30 -> h10 h20 h40 X X X X h_server h_trust X  
h40 -> h10 h20 h30 X X X X h_server h_trust X  
h50 -> X X X X h60 h70 h80 h_server X X  
h60 -> X X X X h50 h70 h80 h_server X X  
h70 -> X X X X h50 h60 h80 h_server X X  
h80 -> X X X X h50 h60 h70 h_server X X  
h_server -> h10 h20 h30 h40 h50 h60 h70 h80 X X  
h_trust -> h10 h20 h30 h40 X X X X h_untrust  
h_untrust -> X X X X X X X X h_trust  
*** Results: 54% dropped (50/110 received)  
mininet> █
```

5. Rules installed in flow table

After I generated some traffic using `pingall`, I immediately ran `dpctl dump-flows` to show the active flow entries installed in all switches before they expire due to the `idle_timeout` or `hard_timeout` specified in my `of_flow_mod` (`idle_timeout = 300`, `hard_timeout = 720`). Each entry in the output represents a flow rule that the switch has set up to handle packets in the network.

The `dpctl dump-flows` command is used to view the flow entries that have been installed in the switch. Flow entries are used by the switch to determine how to handle different types of network packets. When I ran the `pingall` command before, the hosts in the network will start sending ICMP (ping) packets to all other hosts. These packets will cause the switches in the

network to install new flow entries to handle these packets. The switch will know how to process these packets because of the new flow entries. However, for the first packet in a flow that does not match any existing flow entry, the switch doesn't know how to process it and therefore sends it to the controller. This event is called a "packet-in" event. The controller then decides how the packet should be handled and installs a corresponding flow entry in the switch. This flow entry will be used to handle all future packets of the same flow. Each entry in the output represents a flow rule that the switch has set up to handle packets in the network. When a network packet arrives at a switch, the switch checks its flow table to see if there is a match for the packet's characteristics (source IP, destination IP, ports, protocol type, etc.). If a match is found, the switch processes the packet according to the actions specified in the flow entry (forwarding the packet to a specific port, modifying packet headers, dropping the packet, etc.). Here, allow me to explain each characteristic in the output:

a. `cookie=0x0`: The cookie is an identifier that can be used by the controller to filter flow entries. In my case, it is set to zero.

b. `duration=XX.XXXs`: This shows how long the flow entry has been in the table. Each entry has its own duration in seconds.

c. `table=0`: The table field tells me which flow table the rule belongs to. In OpenFlow switches, I can have multiple tables. In my case, all flow rules are in table 0.

d. `n_packets=X`, `n_bytes=X`: These values tell me how many packets and total bytes have matched the rule since it was added.

e. `idle_timeout=30`, `hard_timeout=180`: This is the timeout setting for the flow. If the flow hasn't been matched for a period of time equal to the idle timeout, it will be removed. If the flow has been in the table for a period of time equal to the hard timeout, it will also be removed.

f. `idle_age=XX`: This field indicates the time in seconds since the flow entry was last matched.

g. `priority=X`: This represents the priority of the flow rule. If a packet matches multiple flow rules, the one with the highest priority will be chosen.

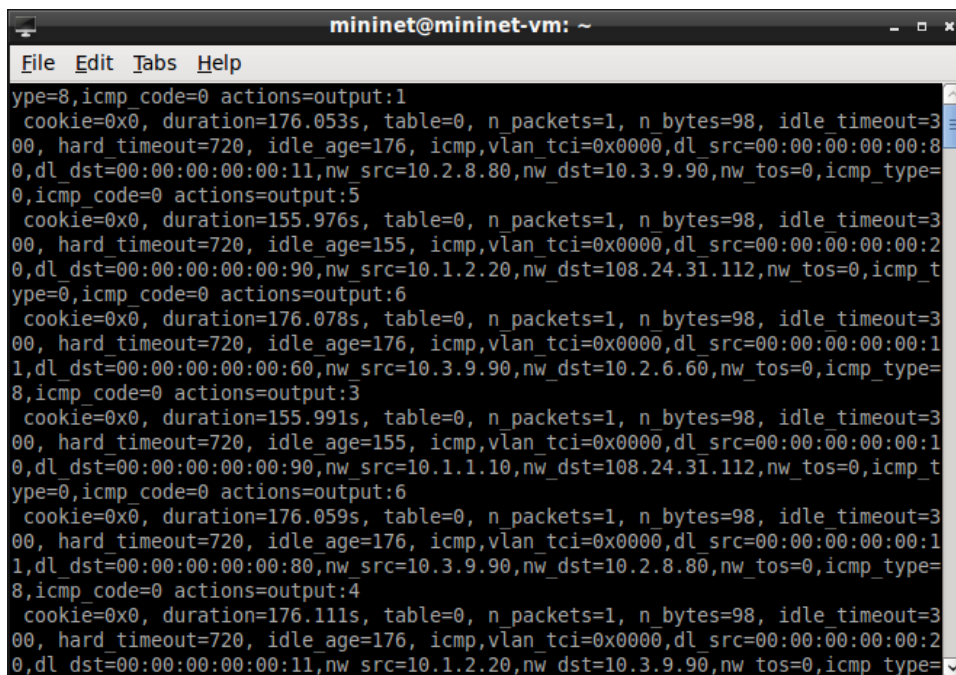
h. `match fields`: The fields after the priority represent the match conditions of the flow entry. `icmp`, `arp`, `dl_src`, `dl_dst`, `nw_src`, `nw_dst`, `nw_tos`, `icmp_type`, `icmp_code`, `arp_spa`, `arp_tpa`, and `arp_op` are all match fields that specify conditions for when this rule should be applied.

- `icmp`, `arp`: These are the types of the network protocol.
- `dl_src`, `dl_dst`: These are the source and destination MAC addresses.
- `nw_src`, `nw_dst`: These are the source and destination IP addresses.
- `nw_tos`: This is the Type of Service (ToS) in the IP header.
- `icmp_type`, `icmp_code`: These are the type and code of ICMP packet.
- `arp_spa`, `arp_tpa`: These are the source and target IP addresses in ARP packet.
- `arp_op`: This is the operation that the sender is performing (1 for request, 2 for reply).

i. actions=output:X: This tells the switch what to do with the packet when it matches this rule. In my case, the "output:X" action means to send the packet to the specified ports (X=1,2,3,4,5, or 6) that were previously set up in my topology code.

The exact time of installation for the flow entries in the switch isn't directly available in the output of the `dpctl dump-flows` command. The duration field indicates how long each flow has been active in the switch, not the specific time of installation. In my experiment, I ran the `dpctl dump-flows` command immediately after generating some traffic using `pingall`. Since flow entries are installed in response to network traffic, I can reasonably infer that the entries were installed shortly before the execution of the `dpctl dump-flows` command.

Below is a screenshot of my output:

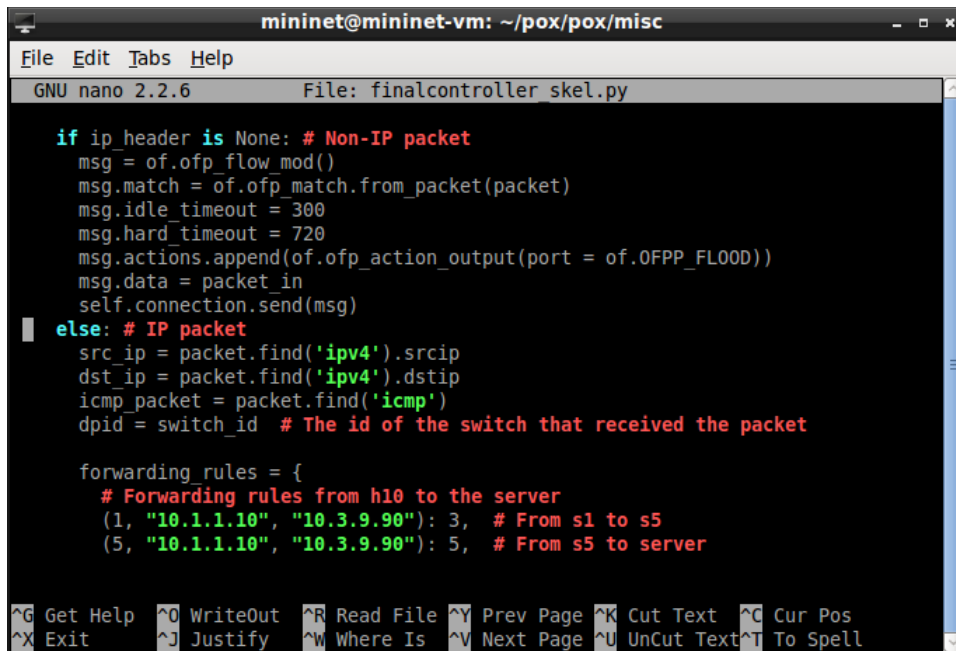


```
mininet@mininet-vm: ~
File Edit Tabs Help
ype=8,icmp_code=0 actions=output:1
  cookie=0x0, duration=176.053s, table=0, n_packets=1, n_bytes=98, idle_timeout=3
00, hard_timeout=720, idle_age=176, icmp,vlan_tci=0x0000,dl_src=00:00:00:00:00:8
0,dl_dst=00:00:00:00:00:11,nw_src=10.2.8.80,nw_dst=10.3.9.90,nw_tos=0,icmp_type=
0,icmp_code=0 actions=output:5
  cookie=0x0, duration=155.976s, table=0, n_packets=1, n_bytes=98, idle_timeout=3
00, hard_timeout=720, idle_age=155, icmp,vlan_tci=0x0000,dl_src=00:00:00:00:00:2
0,dl_dst=00:00:00:00:00:90,nw_src=10.1.2.20,nw_dst=108.24.31.112,nw_tos=0,icmp_t
ype=0,icmp_code=0 actions=output:6
  cookie=0x0, duration=176.078s, table=0, n_packets=1, n_bytes=98, idle_timeout=3
00, hard_timeout=720, idle_age=176, icmp,vlan_tci=0x0000,dl_src=00:00:00:00:00:1
1,dl_dst=00:00:00:00:00:60,nw_src=10.3.9.90,nw_dst=10.2.6.60,nw_tos=0,icmp_type=
8,icmp_code=0 actions=output:3
  cookie=0x0, duration=155.991s, table=0, n_packets=1, n_bytes=98, idle_timeout=3
00, hard_timeout=720, idle_age=155, icmp,vlan_tci=0x0000,dl_src=00:00:00:00:00:1
0,dl_dst=00:00:00:00:00:90,nw_src=10.1.1.10,nw_dst=108.24.31.112,nw_tos=0,icmp_t
ype=0,icmp_code=0 actions=output:6
  cookie=0x0, duration=176.059s, table=0, n_packets=1, n_bytes=98, idle_timeout=3
00, hard_timeout=720, idle_age=176, icmp,vlan_tci=0x0000,dl_src=00:00:00:00:00:1
1,dl_dst=00:00:00:00:00:80,nw_src=10.3.9.90,nw_dst=10.2.8.80,nw_tos=0,icmp_type=
8,icmp_code=0 actions=output:4
  cookie=0x0, duration=176.111s, table=0, n_packets=1, n_bytes=98, idle_timeout=3
00, hard_timeout=720, idle_age=176, icmp,vlan_tci=0x0000,dl_src=00:00:00:00:00:2
0,dl_dst=00:00:00:00:00:11,nw_src=10.1.2.20,nw_dst=10.3.9.90,nw_tos=0,icmp_type=
```

6. IP traffic is implemented not using OFPP_FLOOD

My code complies with the rubric's requirement to not use `OFPP_FLOOD` for IP traffic.

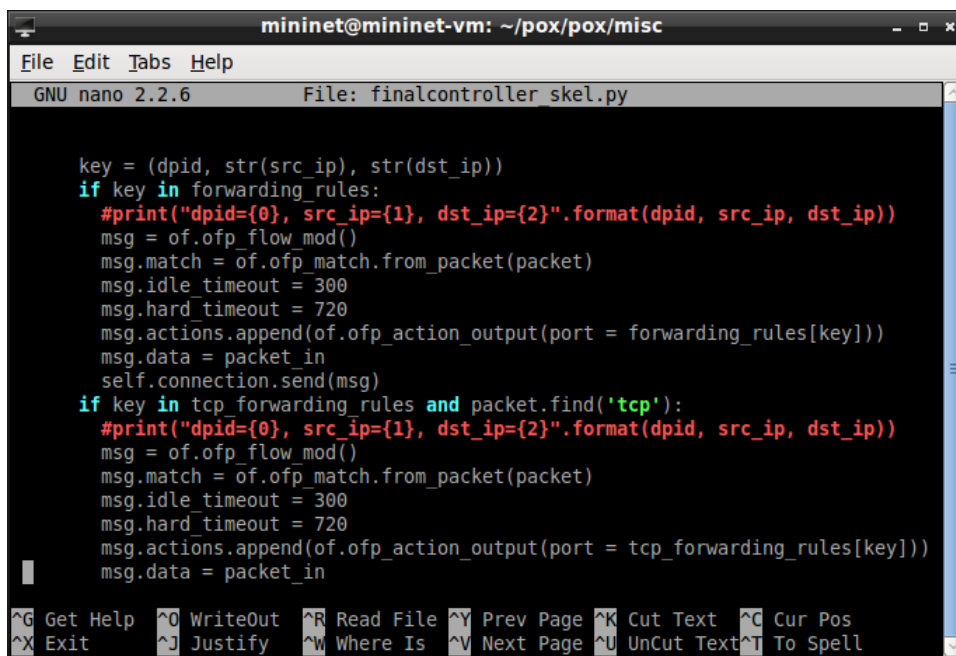
`OFPP_FLOOD` is only used for Non-IP traffic. I have created forwarding tables with entries using a dictionary, consisting of switch ports, source IP, and destination IP, which precisely forwards packets to their respective destinations. Below are screenshots of my `finalcontroller_skel.py` file:



```
mininet@mininet-vm: ~/pox/pox/misc
File Edit Tabs Help
GNU nano 2.2.6 File: finalcontroller skel.py

if ip header is None: # Non-IP packet
    msg = of.ofp_flow_mod()
    msg.match = of.ofp_match.from_packet(packet)
    msg.idle_timeout = 300
    msg.hard_timeout = 720
    msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
    msg.data = packet_in
    self.connection.send(msg)
else: # IP packet
    src_ip = packet.find('ipv4').srcip
    dst_ip = packet.find('ipv4').dstip
    icmp_packet = packet.find('icmp')
    dpid = switch_id # The id of the switch that received the packet

    forwarding_rules = {
        # Forwarding rules from h10 to the server
        (1, "10.1.1.10", "10.3.9.90"): 3, # From s1 to s5
        (5, "10.1.1.10", "10.3.9.90"): 5, # From s5 to server
    }
```



```
mininet@mininet-vm: ~/pox/pox/misc
File Edit Tabs Help
GNU nano 2.2.6 File: finalcontroller skel.py

    key = (dpid, str(src_ip), str(dst_ip))
    if key in forwarding_rules:
        #print("dpid={0}, src_ip={1}, dst_ip={2}".format(dpid, src_ip, dst_ip))
        msg = of.ofp_flow_mod()
        msg.match = of.ofp_match.from_packet(packet)
        msg.idle_timeout = 300
        msg.hard_timeout = 720
        msg.actions.append(of.ofp_action_output(port = forwarding_rules[key]))
        msg.data = packet_in
        self.connection.send(msg)
    if key in tcp_forwarding_rules and packet.find('tcp'):
        #print("dpid={0}, src_ip={1}, dst_ip={2}".format(dpid, src_ip, dst_ip))
        msg = of.ofp_flow_mod()
        msg.match = of.ofp_match.from_packet(packet)
        msg.idle_timeout = 300
        msg.hard_timeout = 720
        msg.actions.append(of.ofp_action_output(port = tcp_forwarding_rules[key]))
        msg.data = packet_in
```

7. Untrusted Host cannot send ICMP traffic to Host 10 to 80

The ping command results show that the untrusted host (h_untrust) cannot send ICMP packets to any other host. This confirms that I have successfully blocked ICMP traffic from the untrusted host, satisfying the assignment requirements. Below is a screenshot of my output:

```
mininet@mininet-vm: ~  
File Edit Tabs Help  
mininet> h untrust ping -c 1 h10  
PING 10.1.1.10 (10.1.1.10) 56(84) bytes of data.  
  
--- 10.1.1.10 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h untrust ping -c 1 h20  
PING 10.1.2.20 (10.1.2.20) 56(84) bytes of data.  
  
--- 10.1.2.20 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h untrust ping -c 1 h30  
PING 10.1.3.30 (10.1.3.30) 56(84) bytes of data.  
  
--- 10.1.3.30 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h untrust ping -c 1 h40  
PING 10.1.4.40 (10.1.4.40) 56(84) bytes of data.  
  
--- 10.1.4.40 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h untrust ping -c 1 h50  
PING 10.2.5.50 (10.2.5.50) 56(84) bytes of data.  
  
--- 10.2.5.50 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h untrust ping -c 1 h60  
PING 10.2.6.60 (10.2.6.60) 56(84) bytes of data.  
  
--- 10.2.6.60 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h untrust ping -c 1 h70  
PING 10.2.7.70 (10.2.7.70) 56(84) bytes of data.  
  
--- 10.2.7.70 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h untrust ping -c 1 h80  
PING 10.2.8.80 (10.2.8.80) 56(84) bytes of data.  
  
--- 10.2.8.80 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> |
```

8. Untrusted Host can send TCP traffic to the hosts.

Based on the output of my `iperf` commands, the untrusted host can still send TCP traffic to all other hosts, indicating that not all traffic has been blocked and only ICMP has been blocked. Below is a screenshot of my output:


```
mininet@mininet-vm: ~  
File Edit Tabs Help  
mininet> iperf h_untrust h10  
*** Unknown command: iperf h_untrust h10  
mininet> iperf h_untrust h10  
*** Iperf: testing TCP bandwidth between h_untrust and h10  
*** Results: ['16.9 Gbits/sec', '17.0 Gbits/sec']  
mininet> iperf h_untrust h20  
*** Iperf: testing TCP bandwidth between h_untrust and h20  
*** Results: ['17.3 Gbits/sec', '17.3 Gbits/sec']  
mininet> iperf h_untrust h30  
*** Iperf: testing TCP bandwidth between h_untrust and h30  
*** Results: ['25.3 Gbits/sec', '25.3 Gbits/sec']  
mininet> iperf h_untrust h40  
*** Iperf: testing TCP bandwidth between h_untrust and h40  
*** Results: ['19.1 Gbits/sec', '19.1 Gbits/sec']  
mininet> iperf h_untrust h50  
*** Iperf: testing TCP bandwidth between h_untrust and h50  
*** Results: ['16.2 Gbits/sec', '16.3 Gbits/sec']  
mininet> iperf h_untrust h60  
*** Iperf: testing TCP bandwidth between h_untrust and h60  
*** Results: ['17.7 Gbits/sec', '17.7 Gbits/sec']  
mininet> iperf h_untrust h70  
*** Iperf: testing TCP bandwidth between h_untrust and h70  
*** Results: ['19.1 Gbits/sec', '19.1 Gbits/sec']  
mininet> iperf h_untrust h80  
*** Iperf: testing TCP bandwidth between h_untrust and h80  
*** Results: ['15.3 Gbits/sec', '15.3 Gbits/sec']  
mininet> 
```

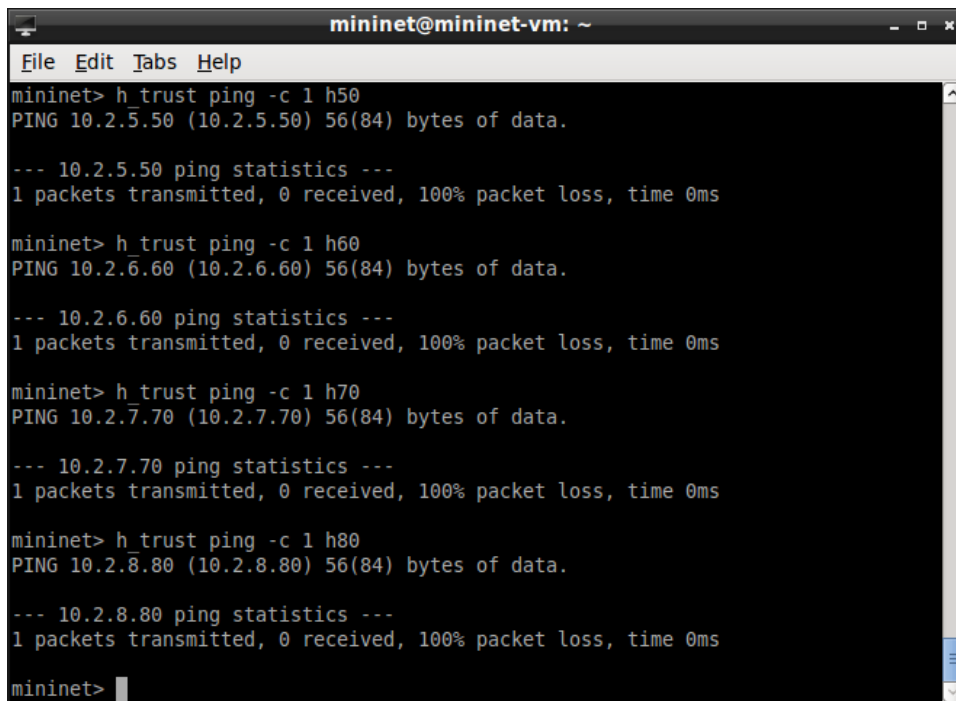
9. Untrusted/Trust Host cannot send any traffic to Server

The ping attempts from both the trusted and untrusted host to the server resulted in 100% packet loss, meaning that the server did not respond to the pings. This behavior demonstrates that no ICMP (or any other) traffic can reach the server from these hosts. This proves that the controller is successfully blocking traffic from the untrusted and trusted hosts to the server, meeting the requirements. The `iperf` command also is unresponsive, demonstrating that the untrusted and trusted host can not send TCP traffic to the server either. Below is a screenshot of my output:

```
mininet@mininet-vm: ~  
File Edit Tabs Help  
mininet> h_untrust ping -c 1 h_server  
PING 10.3.9.90 (10.3.9.90) 56(84) bytes of data.  
  
--- 10.3.9.90 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h_trust ping -c 1 h_server  
PING 10.3.9.90 (10.3.9.90) 56(84) bytes of data.  
  
--- 10.3.9.90 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> 
```

10. Trusted Host cannot send ICMP traffic to Host 50 to 80

The ping attempts from the trusted host to hosts 50, 60, 70, and 80 also resulted in 100% packet loss. This shows that the controller is successfully blocking ICMP traffic from the trusted host to these hosts. Below is a screenshot of my output:



```
mininet@mininet-vm: ~  
File Edit Tabs Help  
mininet> h trust ping -c 1 h50  
PING 10.2.5.50 (10.2.5.50) 56(84) bytes of data.  
  
--- 10.2.5.50 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h trust ping -c 1 h60  
PING 10.2.6.60 (10.2.6.60) 56(84) bytes of data.  
  
--- 10.2.6.60 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h trust ping -c 1 h70  
PING 10.2.7.70 (10.2.7.70) 56(84) bytes of data.  
  
--- 10.2.7.70 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h trust ping -c 1 h80  
PING 10.2.8.80 (10.2.8.80) 56(84) bytes of data.  
  
--- 10.2.8.80 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> |
```

11. Trusted Host can send ICMP traffic to Host 10 to 40

The ping attempts from the trusted host to hosts 10, 20, 30, and 40 were successful, with a packet loss of 0%. This demonstrates that ICMP traffic from the trusted host to these hosts is allowed by my firewall code for the controller. Below is a screenshot of my output:

```
mininet@mininet-vm: ~  
File Edit Tabs Help  
mininet> h_trust ping -c 1 h10  
PING 10.1.1.10 (10.1.1.10) 56(84) bytes of data.  
64 bytes from 10.1.1.10: icmp_seq=1 ttl=64 time=32.4 ms  
  
--- 10.1.1.10 ping statistics ---  
1 packets transmitted, 1 received, 0% packet loss, time 0ms  
rtt min/avg/max/mdev = 32.405/32.405/32.405/0.000 ms  
mininet> h_trust ping -c 1 h20  
PING 10.1.2.20 (10.1.2.20) 56(84) bytes of data.  
64 bytes from 10.1.2.20: icmp_seq=1 ttl=64 time=33.7 ms  
  
--- 10.1.2.20 ping statistics ---  
1 packets transmitted, 1 received, 0% packet loss, time 0ms  
rtt min/avg/max/mdev = 33.715/33.715/33.715/0.000 ms  
mininet> h_trust ping -c 1 h30  
PING 10.1.3.30 (10.1.3.30) 56(84) bytes of data.  
64 bytes from 10.1.3.30: icmp_seq=1 ttl=64 time=127 ms  
  
--- 10.1.3.30 ping statistics ---  
1 packets transmitted, 1 received, 0% packet loss, time 0ms  
rtt min/avg/max/mdev = 127.915/127.915/127.915/0.000 ms  
mininet> h_trust ping -c 1 h40  
PING 10.1.4.40 (10.1.4.40) 56(84) bytes of data.  
64 bytes from 10.1.4.40: icmp_seq=1 ttl=64 time=52.8 ms  
  
--- 10.1.4.40 ping statistics ---  
1 packets transmitted, 1 received, 0% packet loss, time 0ms  
rtt min/avg/max/mdev = 52.869/52.869/52.869/0.000 ms  
mininet> 
```

12. Host 10 to 40 cannot send ICMP traffic to Host 50 to 80

The ping attempts from hosts 10, 20, 30, and 40 to hosts 50, 60, 70, and 80 resulted in 100% packet loss. This shows that my firewall code for the controller is successfully blocking ICMP traffic from hosts 10 to 40 to hosts 50 to 80. Below is a screenshot of my output:

```
mininet@mininet-vm: ~  
File Edit Tabs Help  
mininet> h10 ping -c 1 h50  
PING 10.2.5.50 (10.2.5.50) 56(84) bytes of data.  
  
--- 10.2.5.50 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h10 ping -c 1 h60  
PING 10.2.6.60 (10.2.6.60) 56(84) bytes of data.  
  
--- 10.2.6.60 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h10 ping -c 1 h70  
PING 10.2.7.70 (10.2.7.70) 56(84) bytes of data.  
  
--- 10.2.7.70 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h10 ping -c 1 h80  
PING 10.2.8.80 (10.2.8.80) 56(84) bytes of data.  
  
--- 10.2.8.80 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h20 ping -c 1 h50  
PING 10.2.5.50 (10.2.5.50) 56(84) bytes of data.  
  
--- 10.2.5.50 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h20 ping -c 1 h60  
PING 10.2.6.60 (10.2.6.60) 56(84) bytes of data.  
  
--- 10.2.6.60 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h20 ping -c 1 h70  
PING 10.2.7.70 (10.2.7.70) 56(84) bytes of data.  
  
--- 10.2.7.70 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h20 ping -c 1 h80  
PING 10.2.8.80 (10.2.8.80) 56(84) bytes of data.  
  
--- 10.2.8.80 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h30 ping -c 1 h50  
PING 10.2.5.50 (10.2.5.50) 56(84) bytes of data.  
  
--- 10.2.5.50 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h30 ping -c 1 h60  
PING 10.2.6.60 (10.2.6.60) 56(84) bytes of data.  
  
--- 10.2.6.60 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h30 ping -c 1 h70  
PING 10.2.7.70 (10.2.7.70) 56(84) bytes of data.  
  
--- 10.2.7.70 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h30 ping -c 1 h80  
PING 10.2.8.80 (10.2.8.80) 56(84) bytes of data.  
  
--- 10.2.8.80 ping statistics ---  
1 packets transmitted, 0 received, 100% packet loss, time 0ms  
  
mininet> h40 ping -c 1 h50  
PING 10.2.5.50 (10.2.5.50) 56(84) bytes of data.
```