IERG5090
Advanced Networking Protocols and Systems

2016/2017 Sem 2

HW2: SDN Lab

Due on Apr 3, 2017

Mar, 2017

Overview

In this lab, you will learn

1. how to set up a SDN emulation environment on your own laptop or PC; how to connect and access the environment from your laptop; simple examples of **Mininet** and some development tools;
2. how to write and run custom topology in Mininet; the basic knowledge and APIs of the controller platform you choose (here we take POX as an example); how to implement network functions, such as hub, switch, and firewall, on the controller platform;
3. the IP load balancer component of POX and its usage together with L2 learning switches; the problem of containing connection loop in network topology for SDN and how POX handles it via the spanning tree module; the load balancing via routing on multiple paths in SDN.

Submission

You are required to submit a lab report containing screenshots of some key steps. Besides, you need to submit some files which contains your implementation of some functions. The required screenshots and files are marked in red color through the lab task description. Please follow the guidelines below to submit your lab report.

Submit Method

You need to submit a compressed file (e.g., a .zip file), which contains the PDF version of your report and all the required files. Please include your name and student ID in the file name and the first page of your report. Submit your compressed file to eLeanring or send it to the following email address before 23:59pm of the due date. If you submit by email, the email subject should start with *IERG5090-HW2*.

```
dt016@ie.cuhk.edu.hk
```

Academic Honesty

You are required to complete the homework by yourself. Please refer to the plagiarism policy of CUHK at http://www.cuhk.edu.hk/policy/academichonesty/index.htm.

Acknowledgement

The design and write-up of this lab exercise is mostly prepared by Jacky ZHAN.

Lab Tasks

Task I: Set up Virtual Machine

1. Download and install VirtualBox on your laptop at https://www.virtualbox.org/.
2. Download the virtual machine image from HERE.
3. Install the downloaded image in VirtualBox in following steps:
   (a) start up VirtualBox, select File → Import Appliance;
   (b) select the .ova image that you downloaded, wait for the installation;
   (c) select your VM and go to the Settings Tab. Go to Network → Adapter 2. Select the "Enable adapter" box, and attach it to "host-only network". (This will allow you to easily access your VM through your host machine)

   Remarks: In case there is no valid adapter for host-only, please refer to [3] for solutions.
4. Start the VM and you should be able to login the VM using **mininet** as both username and password.

Task II: Connect and Access VM

IP for Host Access

Run the following command in your VM to figure out the network interface configuration of your VM. There should be three interfaces (2 *ethx* and 1 *loc*) if you follow the previous steps in setting up VM.

```
$ ifconfig -a
```

If the IP address of any *ethx* is missing, run the following command to assign IP address to the interface.

```
$ sudo dhclient ethx
```

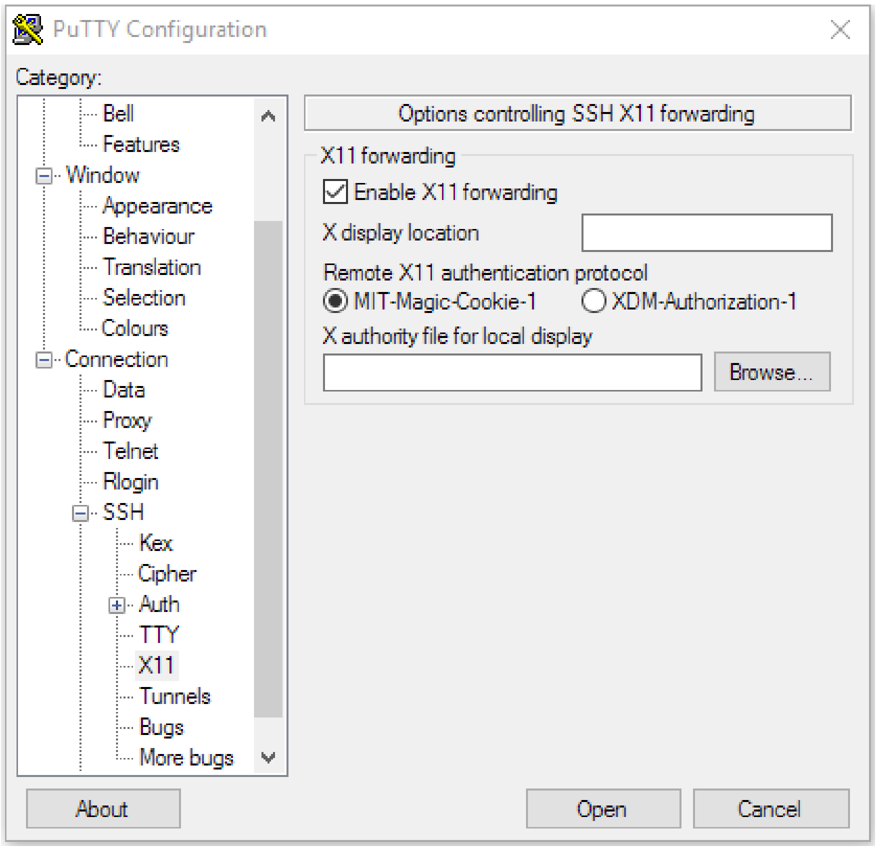
Mark down the IP address for the **host-only** network interface (probably the 192.168.x.x), it will allow you to connect your VM from your laptop via SSH.
(Include the screenshot of running $(ifconfig\ -a)$ into your lab report)

Connect VM from Host

Only the connect procedures for Windows are introduced here. For other operating systems for your laptop, please refer to the instructions at [1].
We will use PuTTY as the SSH client to access the VM, you can download and install it from http://www.putty.org/.
Before we start the connection via PuTTY, we need to setup the Xming Sever (explore it at

HERE) and enable the connection to VM with X11 forwarding. These will allow you to use X11 applications such as **xterm** and **Wireshark** in future labs.

1. Xming Sever could be downloaded from HERE and you can start it by double-clicking its icon. (No window will appear, check its running process in the task manager of Windows)
2. X11 forwarding could be enabled in PuTTY by clicking Connection → SSH → X11, and then Forwarding → Enable X11 Forwarding, see below.



Options controlling SSH X11 forwarding

X11 forwarding

[✓] Enable X11 forwarding

X display location

Remote X11 authentication protocol

(●) MIT-Magic-Cookie-1 ( ) XDM-Authorization-1

X authority file for local display

Till now, you should be able to access the VM from host (your laptop) via PuTTY, you could set up multiple connections simultaneously.

Task III: Start Mininet

Connect to the VM via PuTTY and run the following command to emulate a simple network in Mininet. It will remains in Mininet console (start with *mininet*) after initializing the network and you could play with different Mininet commands.

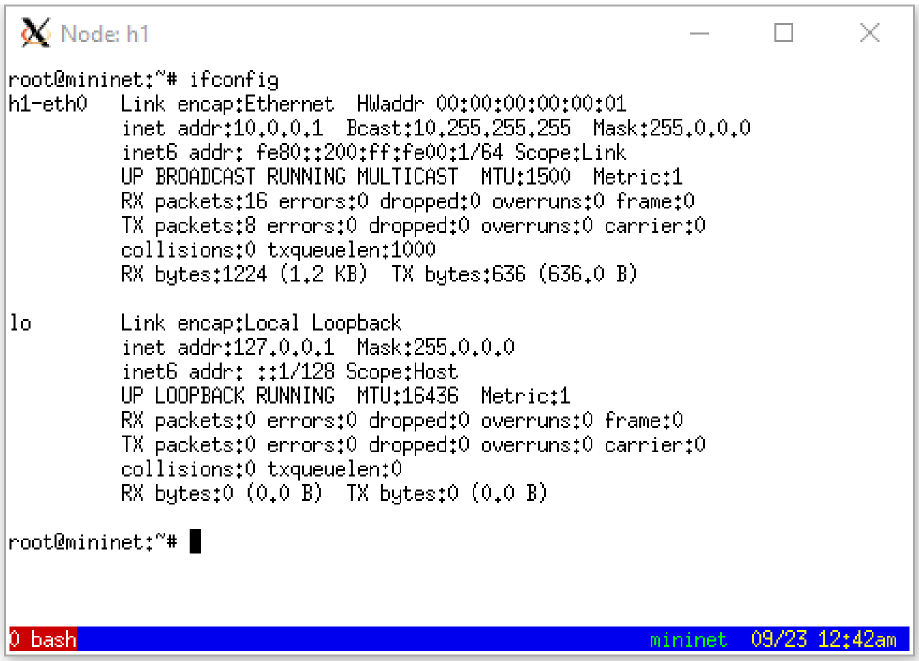
```
$ sudo mn --topo single,3 --mac
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
```

```
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Starting controller
*** Starting 1 switches
s1
*** Starting CLI:
```

Development Tool: xterm

The **xterm** is a terminal emulator and allows you to access the emulated network nodes in Mininet. Run the following command in Mininet console and it will provide the terminal for the host h1 you just created, see below.

```
mininet> xterm h1
```

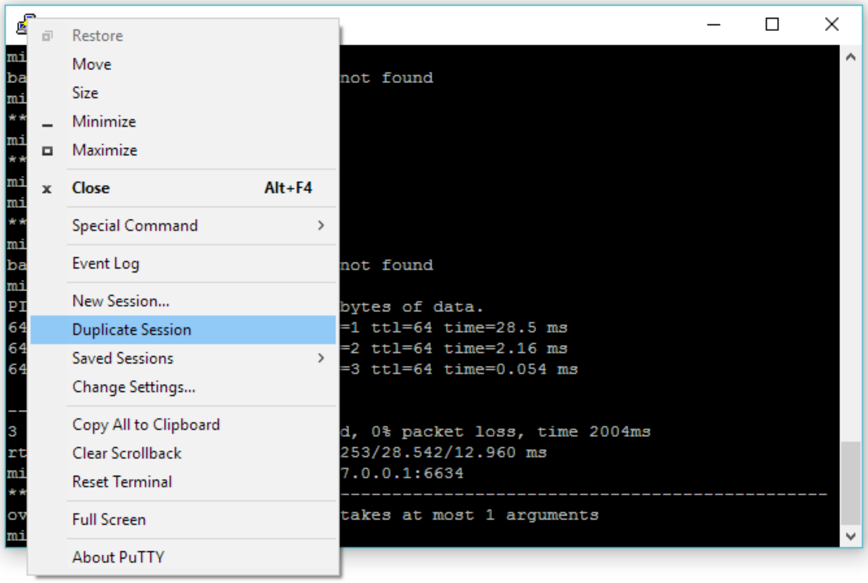


```
root@mininet:~# ifconfig
h1-eth0   Link encap:Ethernet  HWaddr 00:00:00:00:00:01
          inet addr:10.0.0.1  Bcast:10.255.255.255  Mask:255.0.0.0
          inet6 addr: fe80::200:ff:fe00:1/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:16 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1224 (1.2 KB)  TX bytes:636 (636.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@mininet:~#

0 bash                                                    mininet  09/23 12:42am
```

Development Tool: dpctl

The **dpctl** is a management utility that enables some control over the OpenFlow switch. With this tool it's possible to add flows to the flow table, query for switch features and status, and change other configurations.
To try the dpctl, you need to create another connection to the VM via PuTTY, as the

existing one is occupied by Mininet. You could duplicate the PuTTY session easily from the existing one, see below.



```
not found
not found
bytes of data.
=1 ttl=64 time=28.5 ms
=2 ttl=64 time=2.16 ms
=3 ttl=64 time=0.054 ms
d, 0% packet loss, time 2004ms
253/28.542/12.960 ms
7.0.0.1:6634
---------------------------------------------
takes at most 1 arguments
```

In the new PuTTY session, run the following command to dump the port state and capabilities of the switch s1 you just created.

```
$ dpctl show tcp:127.0.0.1:6634
```

You could configure the switch with the following commands to applying packet forward rule: forwarding packets coming at port 1 to port 2 and vice-versa.

```
$ dpctl add-flow tcp:127.0.0.1:6634 in_port=1,actions=output:2
$ dpctl add-flow tcp:127.0.0.1:6634 in_port=2,actions=output:1
```

Run the following command to verify the configuration by checking the flow-table of the switch.

```
$ dpctl dump-flows tcp:127.0.0.1:6634
```

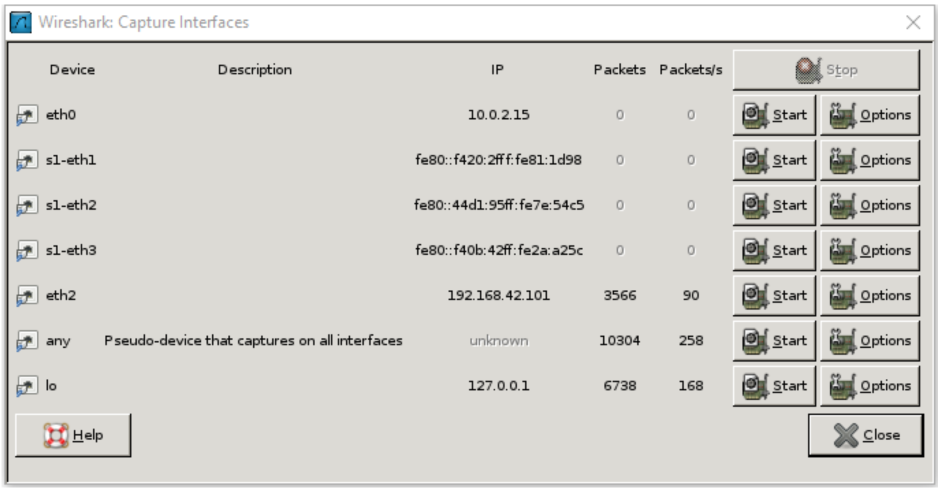
(Include the screenshot of the flow-table into your lab report)
More information about the usage of dpctl could be found at [6].

Development Tool: Wireshark

Wireshark is a network protocol analyzer and it lets you see what's happening on your network at a microscopic level. It is the de facto standard across many industries and educational institutions. We could utilize it to observe the traffics of the created network in Mininet.
Wireshark is installed by default in the VM image. Run the following command to start Wireshark for the VM on your laptop via the X11 forwarding enabled.

```
$ sudo wireshark &
```

To monitor traffics of the network, select Capture → Interfaces and pick the interface you want to monitor, see below.


Wireshark: Capture Interfaces

Device	Description	IP	Packets	Packets/s	Stop	
eth0		10.0.2.15	0	0	Start	Options
s1-eth1		fe80::f420:2fff:fe81:1d98	0	0	Start	Options
s1-eth2		fe80::44d1:95ff:fe7e:54c5	0	0	Start	Options
s1-eth3		fe80::f40b:42ff:fe2a:a25c	0	0	Start	Options
eth2		192.168.42.101	3566	90	Start	Options
any	Pseudo-device that captures on all interfaces	unknown	10304	258	Start	Options
lo		127.0.0.1	6738	168	Start	Options

Try to monitor the traffics generated by the following Ping test. **(Include the screenshot of traffic records captured in Wireshark into your lab report)**

```
mininet> h1 ping -c3 h2
```

Task IV: Mininet Walkthrough

You will be introduced some basic commands and APIs of Mininet in this section. The complete documentation of Mininet is at [7].

Basic Commands

```
# Start a minimal topology and enter the CLI:
$ sudo mn

# Display Mininet CLI commands:
mininet> help

# Exit the CLI
mininet> exit

# If Mininet crashes for some reason, clean it up
$ sudo mn -c
```

Nodes and Links

```
# Display nodes:
mininet> nodes

# Display links:
mininet> net

# Dump information about all nodes:
mininet> dump

# Run command on specific node by typing the node name first
mininet> h1 ifconfig -a
```

Connectivity

```
# Ping test from h1 to h2
mininet> h1 ping -c 1 h2

# Test all-pairs Ping
mininet> pingall
```

(Include the screenshot of verifying connectivity in your VM into your lab report)

Task V: Topology in Mininet

Built-in Topology

Mininet supports several built-in typical topologies and it allows you to run some self-contained regression tests, such as Ping Test, on the topologies. For example, the following command performs *pingall* test in a default single topology with 2 hosts connecting to a single switch.

```
$ sudo mn --test pingall
```

You could specify the topology and corresponding parameters among the built-in ones that Mininet supports, including *linear*, *minimal*, *reversed*, *single*, *tree*. For example, the following command performs the *pingall* test in a tree topology with a depth of 2 and fanout of 3. You can explore other built-in topologies at [8].

```
$ sudo mn --test pingall --topo tree,2,3
```

Custom Topology

Apart from the built-in topologies and self-contained regression tests, Mininet supports customization of both test and topology in Python, see below:

```
# filename: mytopo.py
class MyTopo( Topo ):
   def build( self, ...):
def myTest( net ):
...
topos = { 'mytopo': MyTopo }
tests = { 'mytest': myTest }
```

The above example adds the *MyTopo* class to the topos dictionary and allows you to run the *myTest*. You could specify to use *MyTopo* and run *myTest* using the $--custom$ to include the file following *sudomn*, see below:

```
$ sudo mn --custom mytopo.py --topo mytopo,3 --test mytest
```

The concrete implementation of the custom topology requires understanding of the Mininet Python APIs, which are built at three primary levels according to the Mininet documentation at [9].

1. Low-level API (nodes and links): the low-level API consists of the base node and link classes (such as Host, Switch, and Link and their subclasses) which can actually be instantiated individually and used to create a network, but it is a bit unwieldy.

```
# low-level API example
h1 = Host( 'h1' )
h2 = Host( 'h2' )
s1 = OVSSwitch( 's1', inNamespace=False )
c0 = Controller( 'c0', inNamespace=False )
Link( h1, s1 )
Link( h2, s1 )
h1.setIP( '10.1/8' )
h2.setIP( '10.2/8' )
c0.start()
s1.start( [ c0 ] )
```

2. Mid-level API (network object): The mid-level API adds the Mininet object which serves as a container for nodes and links. It provides a number of methods (such as addHost(), addSwitch(), and addLink()) for adding nodes and links to a network, as well as network configuration, startup and shutdown (notably start() and stop().)

```
# mid-level API example
net = Mininet()
```

```
h1 = net.addHost( 'h1' )
h2 = net.addHost( 'h2' )
s1 = net.addSwitch( 's1' )
c0 = net.addController( 'c0' )
net.addLink( h1, s1 )
net.addLink( h2, s1 )
```

3. High-level API (topology templates): The high-level API adds a topology template abstraction, the Topo class, which provides the ability to create reusable, parametrized topology templates. These templates can be passed to the mn command (via the –custom option) and used from the command line.

```
# high-level API example
class SingleSwitchTopo( Topo ):
    def build( self, count=1 ):
        hosts = [ self.addHost( 'h%d' % i )
                  for i in range( 1, count + 1 ) ]
        s1 = self.addSwitch( 's1' )
        for h in hosts:
            self.addLink( h, s1 )
net = Mininet( topo=SingleSwitchTopo( 3 ) )
net.start()
CLI( net )
net.stop()
```

Performance Parameters

For full reference manual of Mininet Python API, you can explore at [10]. Here we introduce the parameters for configuring network performance in Mininet APIs.
Mininet provides performance limiting and isolation features, through the *CPULimitedHost* and *TCLink* classes. A simple way to use the classes is to specify them as the default host and link classes/constructors to *Mininet()*, and then to specify the appropriate parameters in the topology, see below:

```
# network performance classes
net = Mininet(topo=topo, host=CPULimitedHost, link=TCLink)
```

The parameter *cpu* in *addHost()* allows you to specify a fraction of overall system CPU resources which will be allocated to the virtual host.

```
# CPU Limitation
self.addHost( name, cpu=f )
```

In *addLink()*, the parameter *bw* configures the link bandwidth in Mbit; *delay* is expressed as a string with units in place (e.g. '5ms', '100us', '1s'); *loss* is expressed as a percentage (between 0 and 100); and *max_queue_size* refers to the maximum queue size and is expressed in packets; and *use_htb* indicates whether to use the Hierarchical Token Bucket rate limiter.

```
# Link Performance
self.addLink( node1, node2, bw=10, delay='5ms', max_queue_size=1000,
              loss=10, use_htb=True )
```

Topology Design Practice

Suppose we are going to design the network topology for a campus department. The network will serve three types of purposes: a) Student Lab, which has 5 end hosts; b) 3 Staff Offices, among which 1 office has 3 end hosts and each of the others is equipped with one end host; and c) a Private Web Server with DB, which supports internal content sharing inside the department.
Please implement a topology class *Lab2Topo* for the above scenario, and configure the network to meet the following requirements:

1. Separate the end hosts of a), b) and c) under different switches;
2. Set the host names of a), b) and c) with prefixes as *sl*, *so* and *sw* respectively;
3. Configure the link bandwidths of all switch pairs to be 2 Gbits/sec and that of switch-host pairs to be 10 Mbits/sec, 50 Mbits/sec and 1 Gbits/sec for a), b) and c) respectively;
4. Configure the link delay of all switch pairs to be 20us and that of switch-host pairs to be 50ms, 20ms and 10ms for a), b) and c) respectively;
5. Configure the link loss of all switch pairs to be 0 and that of switch-host pairs to be 5%, 2% and 1% for a), b) and c) respectively.

(Submit your topology class file with name as lab2topo.py)
To verify your design of network topology, implement a test with name *lab2Test* to do the following tasks:

1. Dump all nodes and host connections;
2. Test the network connectivity;
3. Conduct iperf for all node pairs.

(Submit your implementation of *lab2Test* file with name as lab2test.py and include the screenshot of applying your *lab2Test* on your *Lab2Topo*)

Task VI: SDN Controller

One of Mininet's most powerful and useful features is that it uses Software Defined Networking. Using the OpenFlow protocol and related tools, you can program switches to do almost

anything you want with the packets that enter them.
In previous lab tasks, we don't specify a controller when running *sudo mn*. Therefore, Mininet uses the *ovsc* controller as the default one, which is equivalent to the following command:

```
$ sudo mn --controller ovsc ...
```

Controller Platforms

To implement your own logic of network behavior, it is more convenient to use some controller platform and connect it to Mininet as remote controller. The following lists controller platform options in different languages.

1. **Java**: Beacon, Floodlight;
2. **Python**: POX, Ryu;
3. **Ruby**: Trema.

You might choose the controller platform as you want for the following lab tasks. Introductions of different controller choices could be found at [1]. We will explore POX as an example in the following.

POX Basics

The POX controller has been pre-installed in your VM, so as some other controller options. You may find the pox directory with the *ls* command.
Before we start the POX controller, we need to restart Mininet to remove the previous default controller process, see below:

```
mininet> exit
$ sudo mn -c
```

We will use the simple single topology with 3 hosts connecting to a switch to explain the POX basics. Run the following command to start Mininet with specifying a remote controller.

```
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

You might also try to connect a remote controller in your own topology or scripts, which uses the *RemoteController* class of Mininet.

```
net = Mininet( topo=topo, controller=None)
net.addController( 'c0', controller=RemoteController, ip='127.0.0.1',
                   port=6633 )
```

After set up the network in Mininet, we could run a controller script in POX and it could automatically connect to your network. Before digging into the controller implementation, let's run a built-in example of POX, run the following commands in a separate PuTTY session from the one you run Mininet.

```
$ cd pox
$ pox.py log.level --DEBUG misc.of_tutorial
```

The above command tells POX to enable verbose logging and to start the of_tutorial component. You should see the following output of POX, which means your network is under control of the POX script *of_tutorial.py* under the *misc* directory.

```
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
DEBUG:misc.of_tutorial:Controlling [00-00-00-00-00-01 1]
```

POX APIs

It is worthy for you to download the source code of POX from [11] and study its examples (e.g. *of_tutorial.py*) under *misc* directory, which would help you with the future lab tasks. Several useful POX APIs are explained below.

```
# send an OpenFlow message to a switch
connection.send( ... )
```

When a connection to a switch starts, a *ConnectionUp* event is fired. The example code in *of_tutorial.py* creates a new Tutorial object that holds a reference to the associated Connection object. This can later be used to send commands (OpenFlow messages) to the switch.

ofp_action_output class

This is an action for use with ofp_packet_out and ofp_flow_mod. It specifies a switch port that you wish to send the packet out of. It can also take various "special" port numbers. An example of this would be OFPP_FLOOD which sends the packet out all ports except the one the packet originally arrived on.

```
# Example. Create an output action that send packets to all ports
out_action = of.ofp_action_output(port = of.OFPP_FLOOD)
```

ofp_match class

Objects of this class describe packet header fields and an input port to match on. All fields are optional – items that are not specified are "wildcards" and will match on anything. Some notable fields of ofp_match objects are:

1. dl_src - The data link layer (MAC) source address
2. dl_dst - The data link layer (MAC) destination address
3. in_port - The packet input switch port

```
# Example. Create a match that matches packets arriving on port 3:
match = of.ofp_match()
match.in_port = 3
```

ofp_packet_out OpenFlow message

The ofp_packet_out message instructs a switch to send a packet. The packet might be one constructed at the controller, or it might be one that the switch received, buffered, and forwarded to the controller (and is now referenced by a buffer_id).
Notable fields are:

1. buffer_id - The buffer_id of a buffer you wish to send. Do not set if you are sending a constructed packet.
2. data - Raw bytes you wish the switch to send. Do not set if you are sending a buffered packet.
3. actions - A list of actions to apply (for this tutorial, this is just a single ofp_action_output action).
4. in_port - The port number this packet initially arrived on if you are sending by buffer_id, otherwise OFPP_NONE.

```
# Example. Sends a packet out of the specified switch port.
def send_packet( self, buffer_id, raw_data, out_port, in_port ):
"""
Sends a packet out of the specified switch port.
If buffer_id is a valid buffer on the switch, use that.  Otherwise,
send the raw data in raw_data.
The "in_port" is the port number that packet arrived on.  Use
OFPP_NONE if you're generating this packet.
"""
msg = of.ofp_packet_out()
msg.in_port = in_port
if buffer_id != -1 and buffer_id is not None:
    # We got a buffer ID from the switch; use that
    msg.buffer_id = buffer_id
else:
    # No buffer ID from switch -- we got the raw data
    if raw_data is None:
        # No raw_data specified -- nothing to send!
        return
    msg.data = raw_data

action = of.ofp_action_output(port = out_port)
```

```python
msg.actions.append(action)

# Send message to switch
self.connection.send(msg)
```

ofp_flow_mod OpenFlow message
This instructs a switch to install a flow table entry. Flow table entries match some fields of incoming packets, and executes some list of actions on matching packets. The actions are the same as for ofp_packet_out, mentioned above (and, again, for the tutorial all you need is the simple ofp_action_output action). The match is described by an ofp_match object.
Notable fields are:

1. idle_timeout - Number of idle seconds before the flow entry is removed. Defaults to no idle timeout.
2. hard_timeout - Number of seconds before the flow entry is removed. Defaults to no timeout.
3. actions - A list of actions to perform on matching packets (e.g., ofp_action_output)
4. priority - When using non-exact (wildcarded) matches, this specifies the priority for overlapping matches. Higher values are higher priority. Not important for exact or non-overlapping entries.
5. buffer_id - The buffer_id of a buffer to apply the actions to immediately. Leave unspecified for none.
6. in_port - If using a buffer_id, this is the associated input port.
7. match - An ofp_match object. By default, this matches everything, so you should probably set some of its fields!

```python
# Example. Create a flow_mod that sends packets from port 3 out of 4.
fm = of.ofp_flow_mod()
fm.match.in_port = 3
fm.actions.append(of.ofp_action_output(port = 4))
```

Task VII: Network Functions in Controller

Hub

Hub is a common connection point for devices in a network, which is commonly used to connect segments of a LAN. A hub contains multiple ports. When a packet arrives at one port, it is copied to the other ports so that all segments of the LAN can see all packets [12]. Therefore, the behavior of a hub could be verified by conducting host ping test, in which case all hosts connected to the hub will receive the exact same traffic.

The *of_tutorial.py* of POX implements the switch as a hub by default. It evokes the function *self.act_like_hub()* when handling the in-packets, which will flood all the packets to all ports.

```
def act_like_hub (self, packet, packet_in):
    """
    Implement hub-like behavior -- send all packets to all ports
    besides the input port.
    """

    # We want to output to all ports -- we do that using the special
    # OFPP_ALL port as the output port.  (We could have also used
    # OFPP_FLOOD.)
    self.resend_packet(packet_in, of.OFPP_ALL)

    # Note that if we didn't get a valid buffer_id, a slightly better
    # implementation would check that we got the full data before
    # sending it
    # (len(packet_in.data) should be == packet_in.total_len)).
```

Let's verify the hub behavior by conducting ping test from *h1* to *h2*. We can create *xterm* for each host and view the received TCP traffic using *tcpdump*. The following command will create 3 consoles for hosts *h1*, *h2* and *h3*.

```
mininet> xterm h1 h2 h3
```

In the xterm of *h2* and *h3*, run *tcpdump* on *h2-eth0* and *h3-eth0* respectively.

```
root@mininet:~# tcpdump -XX -n -i h2-eth0
```

In the xterm of *h1*, conduct a ping test.

```
root@mininet:~# ping -c1 10.0.0.2
```

The ping packets are now going up to the controller, which then floods them out all interfaces except the sending one. You should see identical ARP and ICMP packets corresponding to the ping in both *xterms* running *tcpdump*. This is how a hub works; it sends all packets to every port on the network.

Switch

A network switch also connects computers to each other, like a hub. Where the switch differs from a hub is in the way it handles packets of data. When a switch receives a packet of data, it determines what computer or device the packet is intended for and sends it to that computer only. It does not broadcast the packet to all computers as a hub does which means bandwidth is not shared and makes the network much more efficient. For this reason alone, switches are usually preferred over a hub [12].

In the *of_tutorial.py* of POX, it has the method *self.act_like_switch()* but without concrete implementation.
(Complete the implementation of *self.act_like_switch()* in *of_tutorial.py* and submit the updated file with name *of_tutorial_switch.py*. In addition, verify the behavior of your switch and include the corresponding screenshots in your lab report.)

Firewall

A Firewall is a network security system that is used to control the flow of ingress and egress traffic usually between a more secure local-area network (LAN) and a less secure wide-area network (WAN). The system analyses data packets for parameters like L2/L3 headers (i.e., MAC and IP address) or performs deep packet inspection (DPI) for higher layer parameters (like application type and services etc) to filter network traffic. A firewall acts as a barricade between a trusted, secure internal network and another network (e.g. the Internet) which is supposed to be not very secure or trusted [13].
(Write a simple firewalling module that blocks traffic between hosts 2 and 3 in *of_tutorial.py* and submit the updated file with name *of_tutorial_firewall.py*. In addition, verify the behavior of your Firewall and include the corresponding screenshots in your lab report.)

Task VIII: IP Load Balancer in SDN

One of the main features of SDN is the capability of implementing load balancing for the target network topology or services. One type of load balancing is to forward network requests from end hosts to multiple servers smartly. Figure 1 presents a simple example of such load balancing case, in which the switch acts as a load balancer. It forwards network requests from end hosts to multiple servers with the objective of balancing loads between Server 1 and Server 2.

POX provides a built-in component *misc.ip_loadbalancer* as a simple TCP load balancer (which started in the carp branch). The POX command of running the load balancer is as following [17].

```
mininet@mininet:~/pox$ ./pox.py misc.ip_loadbalancer --ip=<Service IP>
--servers=<Server1 IP>,<Server2 IP>,.. [--dpid=<dpid>]
```

Give it a service IP and a list of server IP addresses. New TCP flows to the service IP will be randomly redirected to one of the server IPs. In the meantime, servers are periodically probed to see if they're alive by sending them ARPs.

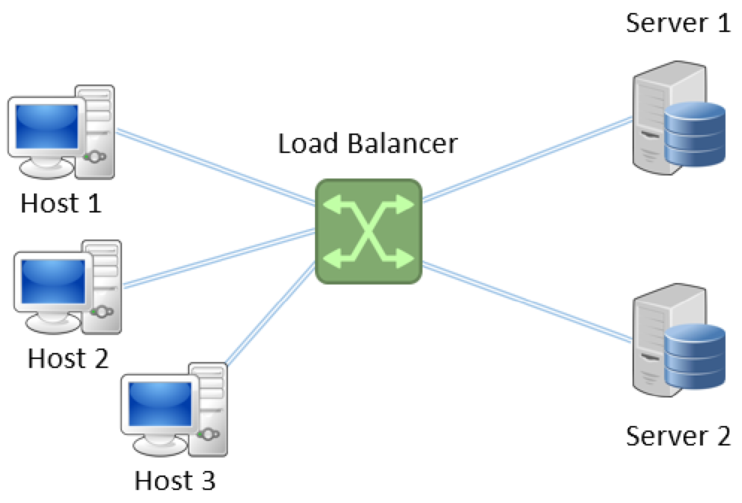


Figure 1: Network Topology with IP Load Balancer

Test Load Balancer in POX

The network in Figure 1 can be easily created from the built-in topology *single* of Mininet with 5 hosts connecting to a single switch.
Run the following POX command to start the controller, this will assume *h1* and *h2* as the servers with the service IP as *10.0.1.1*.

```
mininet@mininet:~/pox$ ./pox.py misc.ip_loadbalancer --ip=10.0.1.1
--servers=10.0.0.1,10.0.0.2
```

In another PuTTY session, run the following command to start the network in Figure 1.

```
mininet@mininet:~$ sudo mn --topo single,5 --controller remote
```

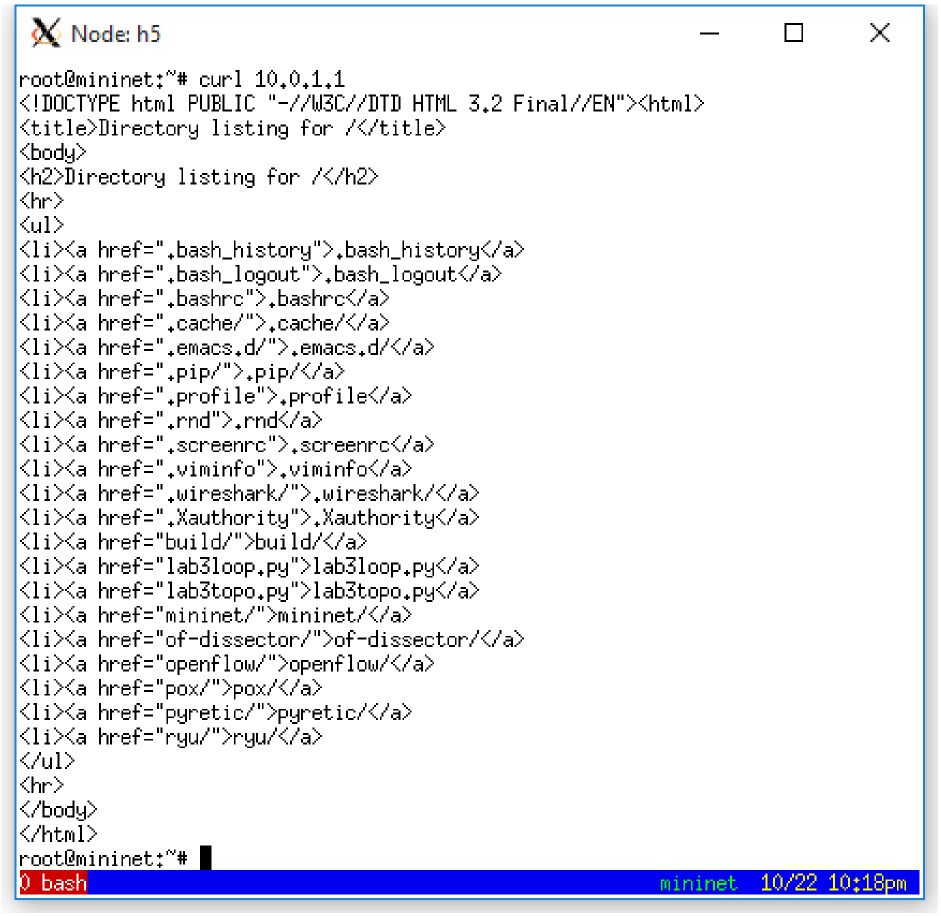
Create *xterm* console for all hosts *h1-h5*. Run the following command in *h1* and *h2* to start simple *HTTP* service at *h1* and *h2*.

```
python -m SimpleHTTPServer 80
```

At xterm console of *h3*, *h4* and *h5*, run the following command to send *HTTP* request to the service IP (*10.0.1.1*).

```
curl 10.0.1.1
```

The command above will grab the directory list in *html* from the servers (the default web page for a simple server) as shown in Figure 2.
(Include the screenshots of xterm consoles at *h1* and *h2* and describe the test result in your lab report)



```
root@mininet:~# curl 10.0.1.1
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
<li><a href=".bash_history">.bash_history</a>
<li><a href=".bash_logout">.bash_logout</a>
<li><a href=".bashrc">.bashrc</a>
<li><a href=".cache/">.cache/</a>
<li><a href=".emacs.d/">.emacs.d/</a>
<li><a href=".pip/">.pip/</a>
<li><a href=".profile">.profile</a>
<li><a href=".rnd">.rnd</a>
<li><a href=".screenrc">.screenrc</a>
<li><a href=".viminfo">.viminfo</a>
<li><a href=".wireshark/">.wireshark/</a>
<li><a href=".Xauthority">.Xauthority</a>
<li><a href="build/">build/</a>
<li><a href="lab3loop.py">lab3loop.py</a>
<li><a href="lab3topo.py">lab3topo.py</a>
<li><a href="mininet/">mininet/</a>
<li><a href="of-dissector/">of-dissector/</a>
<li><a href="openflow/">openflow/</a>
<li><a href="pox/">pox/</a>
<li><a href="pyretic/">pyretic/</a>
<li><a href="ryu/">ryu/</a>
</ul>
<hr>
</body>
</html>
root@mininet:~#
0 bash                                              mininet  10/22 10:18pm
```

Figure 2: Directory List in HTML from the Server

Load Balancer Plus Switch

By default, the load balancer component of POX will make the first switch that connects into a load balancer and ignore the other switches. If you have a topology with multiple switches, it probably makes more sense to specify which one should be the load balancer, and this can be done with the *–dpid* option in command line. In this case, you probably want the rest of the switches to do something worthwhile (like forward traffic), and you may have to create a component that does this for you. For example, you might create a simple component which does the same thing as *forwarding.l2_learning* on all the switches besides the load balancer [17].

Figure 3 shows an example of topology with multiple switches, among with *s2* acts as the load balancer to distribute network requests among multiple servers (*w1*-*w3*). *s1* and *s3* are supposed to act as normal L2 learning switch.

Complete the following tasks for the topology shown in Figure 3.

1. Create a custom topology template for the network with class name *Lab3Topo1* and file name *lab3topo1.py*;

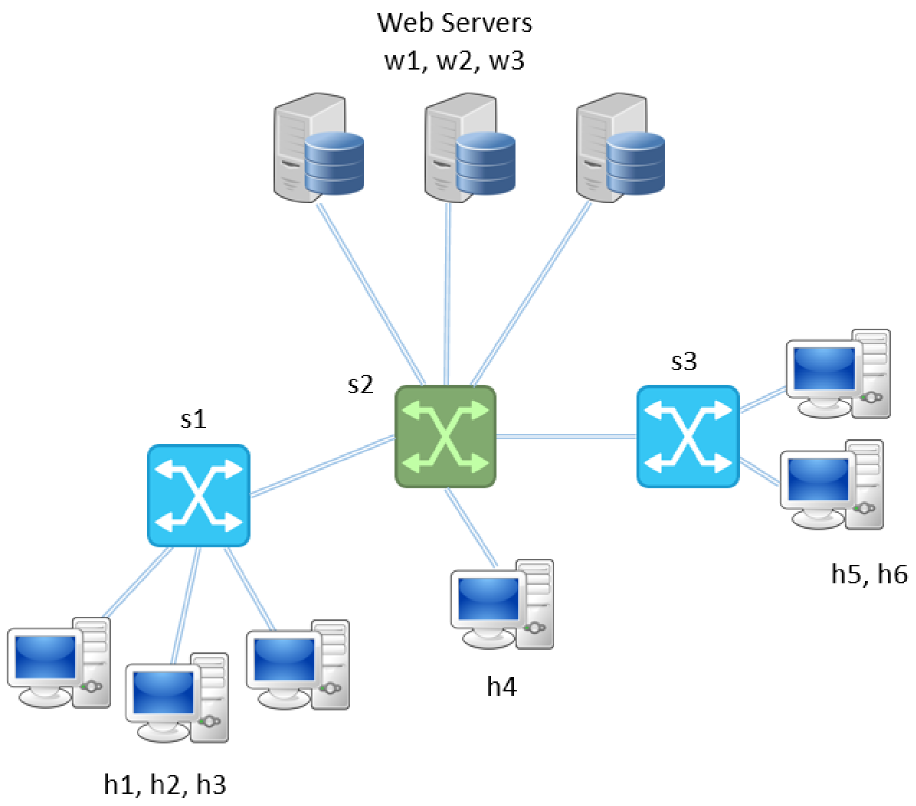


Figure 3: Load Balancer together with Learning Switches

2. Implement *s2* as a load balancer and keep *s1* and *s3* as normal L2 learning switch;
3. Create simple *HTTP* server on *w1-w3*;
4. Conduct the *curl* test (similar to the example above) on all end hosts (*h1-h6*).

***Remarks**

A quick way to run load balancer with multiple switches is to upgrade the POX in your VM from *carp* to *dart*. The *ip_loadbalancer* component has been updated in the *dart* version. Follow the *Git* commands below to complete the upgrade. Besides, the materials at [17] and [18] might be helpful for you to explore more.

```
# remeber to backup you own files implemented under ./pox directory
mininet@mininet:~/pox$ sudo git checkout -b dart
mininet@mininet:~/pox$ sudo git clean -fd
mininet@mininet:~/pox$ sudo git pull origin dart
```

(Include the screenshots of *curl* test results in your lab report and submit your implementation of topology template and controller)

Task IX: Load Balancing via Path Routing

Another type of load balancing in SDN is via routing on multiple paths. The SDN controller schedules traffic flow between hosts on the path selected from multiple path options (between the same hosts). Path routing is based on its global knowledge of the network, so as to improve the network performance and achieve the goal of load balancing.

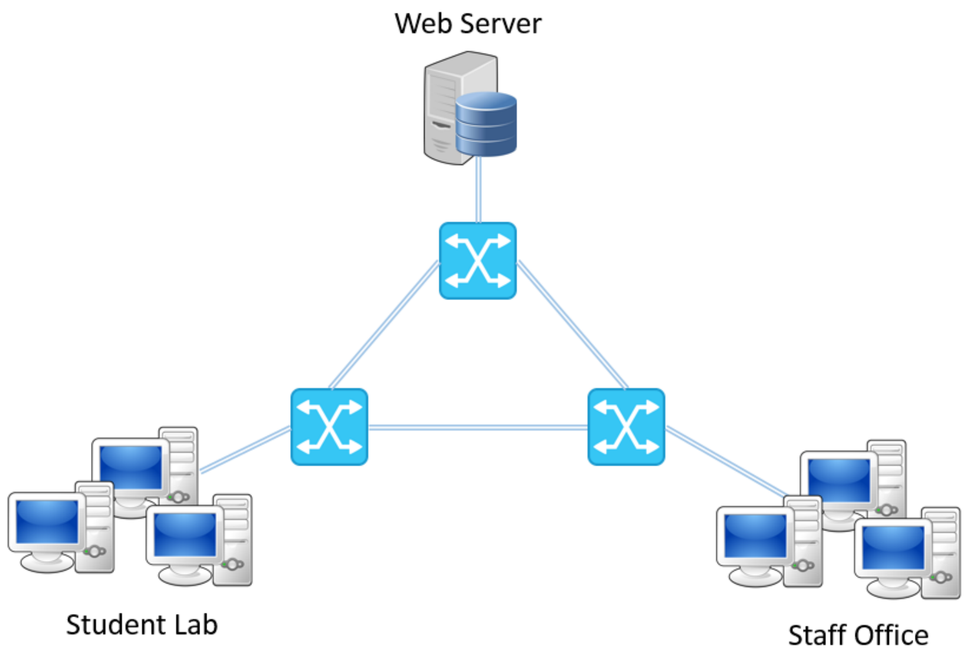


Figure 4: Network Topology with a Connection Loop

Loops in Network Topology

However, multiple paths between the same host pair means there is at least one loop in the network topology (graph). For example, in the previous lab, you are required to implement a custom topology for the given campus department. You might have tried connecting your switches in a loop, see Figure 4. In such network, there exist two paths between all host pairs. The SDN controller could select a suitable path from the two candidates according to some criteria (e.g. with less traffic).
However, Mininet will fail to work with the default ovs-controller, for example, you cannot ping anything in such a network. It's important to remember that Ethernet bridges (also known as learning switches) will flood packets that miss in their MAC tables. They will also flood broadcasts like ARP and DHCP requests. This means that if your network has loops or multiple paths in it, it will not work with the default ovs-controller and controller controllers, nor NOX's pyswitch, nor POX's l2_learning, which all act as learning switches/Ethernet bridges [14].

Therefore, to use a network with loops, you need controllers that support such a network. POX includes a spanning tree module, and other controllers (Floodlight, ONOS, ODL, etc.) may support multipath and/or spanning tree - you will want to consult the documentation for your controller, make sure it is configured correctly to support multipath or spanning tree, and test it to make sure that it actually works [15].

Test Spanning Tree Module in POX

Create a topology template with name *lab3loop.py* and implement it as following. This will create a simple network similar to the one in Figure 4, but with each switch connecting to only one host. The 3 switches form a loop in the network.

```
# file: lab3topo.py
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.cli import CLI

class Lab3LoopTopo( Topo ):
    # initiate the topology
    def __init__( self ):
        Topo.__init__( self )

        h1 = self.addHost( 'h1' )
        h2 = self.addHost( 'h2' )
        h3 = self.addHost( 'h3' )

        s1 = self.addSwitch( 's1' )
        s2 = self.addSwitch( 's2' )
        s3 = self.addSwitch( 's3' )

        self.addLink( h1, s1 )
        self.addLink( h2, s2 )
        self.addLink( h3, s3 )

        self.addLink( s1, s2 )
        self.addLink( s2, s3 )
        self.addLink( s3, s1 )

topos = { 'loop': ( lambda: Lab3LoopTopo() ) }
```

Run the following command to start your POX controller with spanning tree module enabled. You can explore more details about POX spanning tree at [16].

```
mininet@mininet:~$ cd pox
mininet@mininet:~/pox$ ./pox.py forwarding.l2_learning
```

```
openflow.discovery openflow.spanning_tree --no-flood --hold-down
```

Start Mininet with the topology defined in *lab3loop.py* using the following command, which specifies the remote POX controller (the one you just started) as the network controller.

```
mininet@mininet:~$ sudo mn --custom lab3loop.py --topo loop
--controller remote
```

The POX controller will start to discover the network topology upon connecting to the network. You could conduct *ping* tests between hosts to verify whether Mininet works normally.

(Include the screenshots of POX controller output and the test result into your lab report)

Restart your POX controller without spanning tree using the following command and repeat the same experiment.

(Describe the experiment result in your lab report)

```
mininet@mininet:~/pox$ ./pox.py forwarding.l2_learning
```

According to the above example, the spanning tree module allows loop of connection links in SDN network. However, it uses the list of all links within the network to build a spanning tree and changes the default *flood* behavior of OpenFlow switches. It only allows paths on the spanning tree, therefore, there is no multiple path between any host pairs.

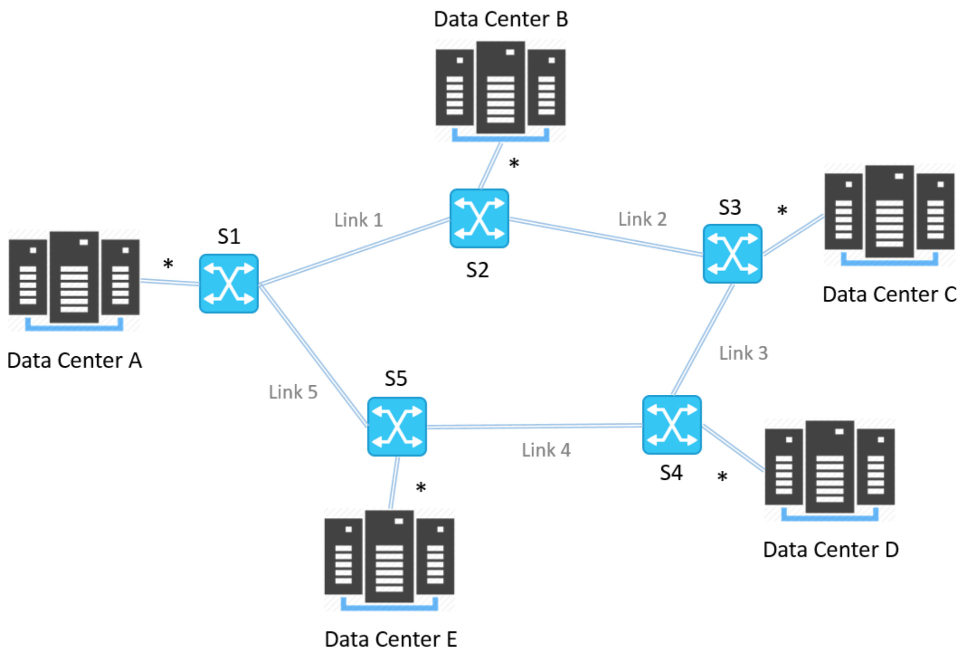


Figure 5: Network Topology of the Given Scenario

Example of Load Balancing via Routing

Suppose a company deploys 5 data centers across the world to support its applications and services. Figure 5 shows the network topology of the data centers and Table 1 gives the corresponding performance parameters of network links. To guarantee its service quality, it is necessary for the data centers to replicate and synchronize data with each other. However, the data synchronization tasks can only be scheduled at midnight and must be completed as soon as possible.
The company uses SDN to operate the network and needs smart routing at the SDN controller. The primary objective here is to minimize the time needed for completing all the data synchronization tasks. The details of data synchronization tasks between data centers are listed in Table 2. For the node pair of each task, there exist two possible paths for the data traffic. For example, for task 1, the packets from data center A to data center E could follow either *A - S1 - S5 - E* or *A - S1 - S2 - S3 - S4 - S5 - E*. The scheduling of paths for the tasks will lead to different performances in the given network. Table 3 lists a possibility of the path scheduling for the tasks, which always select path with the least number of stops. It is not for sure that the path with less number of stops is with better network performance. It depends on the link quality as well as the traffic between other nodes. Please try to schedule the routing path for each task by yourself, so as to minimize the overall time needed for all data synchronization tasks.

Link	**Bandwidth**	**Delay**	**Loss**
*	1 Gbps	20 us	0%
Link 1	50 Mbps	1 ms	1%
Link 2	40 Mbps	2 ms	1%
Link 3	80 Mbps	1 ms	1%
Link 4	50 Mbps	2 ms	1%
Link 5	50 Mbps	1 ms	1%

Table 1: Network Performance Parameters

Task	**From**	**To**	**Data Volume**
Task 1	A	E	2 GB
Task 2	B	E	1 GB
Task 3	B	C	3 GB
Task 4	A	C	2 GB

Table 2: Data Synchronization Tasks for the Example

Task	**Path**
Task 1	A - S1 - S5 - E
Task 2	B - S2 - S1 - S5 - E
Task 3	B - S2 - S3 - C
Task 4	A - S1 - S2 - S3 - C

Table 3: Path Scheduling with the Least Stop Numbers

(Include your path scheduling table for the tasks in your lab report)

Task X: Load Balancing Experiment for the Example

Connection Information

Create your custom topology in file *datacenter.py* for the data center network in Figure 3. Naming the data centers as end host *ha*, *hb*, *hc*, *hd* and *he* respectively; and the switches as switch *s1*, *s2*, *s3*, *s4* and *s5* respectively. Run the following command to start the network in Mininet.

```
mininet@mininet:~$ sudo mn --custom datacenter.py --topo datacenter
--mac --controller remote --link tc
```

A prerequisite task for our load balancing experiment is to determine the detailed network connection information, especially the port (interface) mapping at switches, so as to forward packets accordingly. In previous lab, the collection information is collected via the process of constructing the *mac_to_port* mapping dictionary. However, the switches *flood* packets to all their ports to probe the information, which cannot be used in our example (*flood* leads to disasters for network with loop).
Therefore, in our case, extra efforts are needed to collect the connection information at switches. On one hand, you can run the following command at Mininet CLI to show the interface mapping of nodes.

```
mininet> net
c0
s1 lo:  s1-eth1:s2-eth1 s1-eth2:s5-eth2 s1-eth3:ha-eth0
...
```

On the other hand, more detailed information could be found at the controller. For example, in the *misc.of_tutorial.py*, insert the following code in the function *_handle_PacketIn()*. Besides, commenting both the *act_like_hub()* and *act_like_switch* to disable the *flood* behavior.

```
def _handle_PacketIn (self, event):
    """
    Handles packet in messages from the switch.
    """
    packet = event.parsed # This is the parsed packet data.
    if not packet.parsed:
        log.warning("Ignoring incomplete packet")
        return

    # the code to display the connection ports
    print "--------------"
    print "At switch with ID %s" % event.connection.dpid
    for p in event.connection.features.ports:
        print "port %s with name %s" % (p.port_no, p.name)

    packet_in = event.ofp # The actual ofp_packet_in message.

    # Comment out the following line and uncomment the one after
    # when starting the exercise.
    # self.act_like_hub(packet, packet_in)
    # self.act_like_switch(packet, packet_in)
```

Start the POX controller via the following command and then conduct *ping* test from each host to extract the ID and port information for each switch, for example, *ha ping -c1 hb*. You may note, as the *flood* behavior is disabled at switches, the *ping* packet will stop at the first switch (stop).

```
mininet@mininet:~$ ./pox.py log.level --DEBUG misc.of_tutorial_lab3
...
-------------
At switch with ID 5
port 3 with name s5-eth3
port 2 with name s5-eth2
port 65534 with name s5
port 1 with name s5-eth1
...
```

Fill the connection mappings in Table 4 for your network using methods above.
(Include the table in your lab report)

Node	To	Interface	Port
s1	ha	s1-eth3	3
s1	s2	s1-eth1	1

Table 4: Network Topology Connection Details

Installing Flow Entry at Switch

The *ofp_flow_mod* of POX is the OpenFlow message that instructs a switch to install a flow table entry. Flow table entries match some fields of incoming packets, and executes some list of actions on matching packets. The match is described by an *ofp_match* object. If incoming packets are matched in flow table, the switch will not make extra requests to controller. You could set the corresponding behavior of flow entry via fields of matched incoming packets. Notable fields include:

1. *actions* - A list of actions to perform on matching packets (e.g., *ofp_action_output*)
2. *priority* - When using non-exact (wildcarded) matches, this specifies the priority for overlapping matches. Higher values are higher priority. Not important for exact or non-overlapping entries.

3. *match* - An *ofp_match* object. By default, this matches everything, so you should probably set some of its fields

The *match* field is used to describe and match against target incoming packets for the flow entry. It contains several fields with some widely used ones as following:

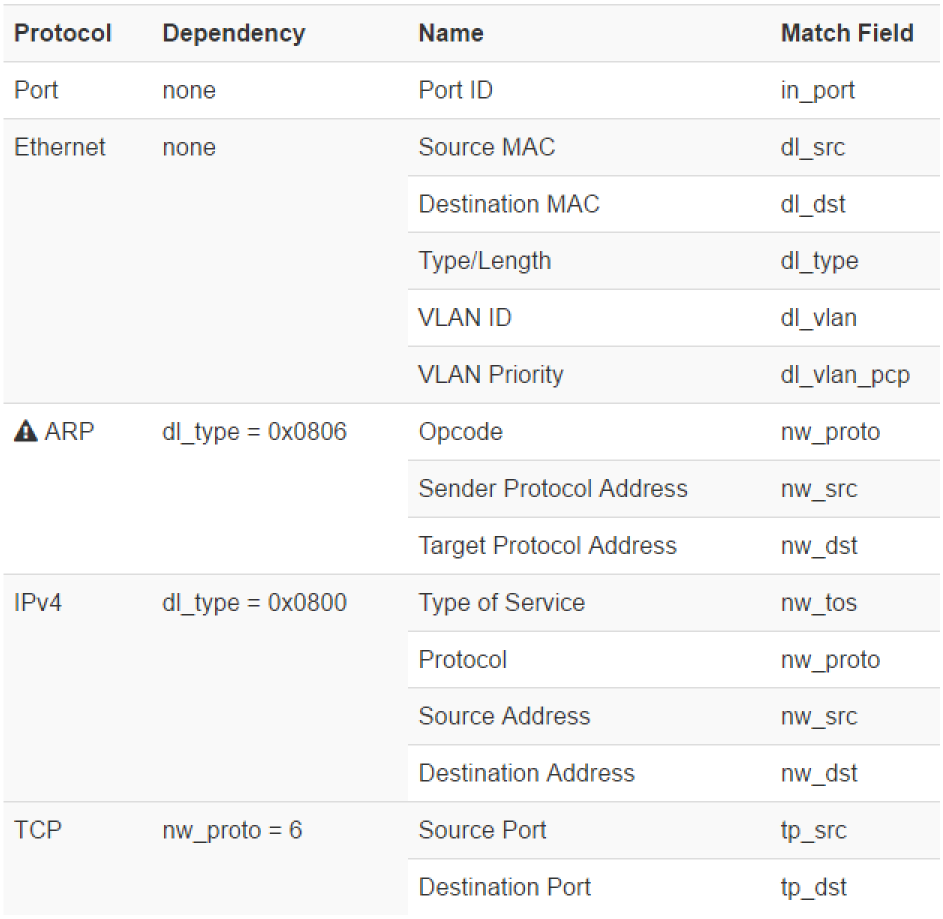
1. *dl_dst*: Ethernet destination address
2. *dl_src*: Ethernet destination address
3. *dl_type*: Ethernet frame type
4. *in_port*: Input switch port
5. *nw_dst*: IP destination address
6. *nw_src*: IP source address

Different combination of match fields should be used for matching packets under different protocols. Figure 6 lists the match fields for some protocols. More details about *match* could be found at [19] and [20].

In our case, if we want to install a flow table entry at the switch *s1*, so as to forward packets from host *ha* to host *hd* via switch *s2* and *s3*, we could implement it as following. The implementation adds flow entry for both IPv4 and ARP protocols, so that *ping* test will work in our experiment.

```
# creat a new flow_message (0-4)
msg = of.ofp_flow_mod(command=0)
msg.priority = 3

# set src and dst IP address of matching
msg.match.dl_type = 0x800
msg.match.nw_src = IP_of_ha
msg.match.nw_dst = IP_of_hd

# forward the packet to certain port X at s1 to s2
msg.actions.append(of.ofp_action_output(port=X))

# send out the message
self.connection.send(msg)


# add similar flow entry for arp
msg.match.dl_type = 0x806
self.connection.send(msg)
```

The flow entry installation steps above could be implemented as a function *install_flow_entry()* at controller. Another function *act_in_lab3()* could be created to add all entry flows to matched switch as below. The flow entries should be according to your selection of paths for the tasks in our example.



Protocol	Dependency	Name	Match Field
Port	none	Port ID	in_port
Ethernet	none	Source MAC	dl_src
		Destination MAC	dl_dst
		Type/Length	dl_type
		VLAN ID	dl_vlan
		VLAN Priority	dl_vlan_pcp
⚠ ARP	dl_type = 0x0806	Opcode	nw_proto
		Sender Protocol Address	nw_src
		Target Protocol Address	nw_dst
IPv4	dl_type = 0x0800	Type of Service	nw_tos
		Protocol	nw_proto
		Source Address	nw_src
		Destination Address	nw_dst
TCP	nw_proto = 6	Source Port	tp_src
		Destination Port	tp_dst

Figure 6: Match Fields for Different Protocols

```
# match switch s1
if switch_id == "id_of_s1":
    # install flow entry at s1
    ...
```

Therefore, we could install flow entry to switches to customize the forwarding rules of packets. The installation at a switch could be triggered when the first matching packet arrives at the switch. In Mininet, switches are identified by the *dpid* field, which could be found via *event.connection.dpid* under *_handle_PacketIn()*.
Implement your controller accordingly and conduct tests to verify the behavior of your network, that is, packets between host pairs follow your selection of paths.
(Include the screenshots of test result in your lab report and your implementation in a file with name *misc.of_tutorial_lab3.py*)

Traffic Generation

To conduct the experiment, we need to generate traffic between host pairs according to the tasks in our example. Here, we make use of simple *HTTP* downloading to emulate the data synchronization of the tasks. For example, for task 1, the data center E has to retrieve data of 2GB from A. In such case, we could start data center A as a *SimpleHTTPServer* and let data center E download a BIG file (2GB) from A. Considering the file is too large, we can change the task to download a relatively small file multiple times from A.
At the xterm console of *ha*, run the following code to generate a file (traffic) of size 100MB and start the *SimpleHTTPServer* under the same directory with the file.

```
dd if=/dev/urandom of=traffic bs=10M count=10
python -m SimpleHTTPServer 80
```

At the xterm console of *he*, run the following code to verify the behavior of the *HTTP* server at *ha*.

```
wget 10.0.0.1/traffic
```

Create a python script *download.py* with the following implementation, which will perform the downloading for multiple times. Beside, it simply measures the time to complete all the downloading.

```python
import urllib
import sys
import time

def download(count=1):
    start = time.time()
    for i in range(count):
        urllib.urlretrieve("http://10.0.0.1/traffic")
    print "Use %s seconds" % (time.time() - start)

if __name__ == '__main__':
    count = int(sys.argv[1])
    download(count)
```

At the xterm console of *he*, run *download.py* to emulate the data traffic of task 1.

```
python download.py 20
```

For the other tasks in our example, similar steps could be followed. However, please note that all tasks should be started simultaneously and the time needed to complete all tasks (should be the duration between the last end-time and the first start-time) is the experiment result for your path planning.

Load Balancing Experiment

The procedures to conduct load balancing experiment for our example are as following:

1. Implement and run the topology in Mininet for the given network;
2. Check the connection information of your network and fill in Table 4;
3. Implement the flow entry installing at the controller according to your path planning;
4. Generate the traffic for all tasks simultaneously and evaluate the completion time.

Conduct the experiment for both the path scheduling in Table 3 and your selection of paths for the tasks. Compare the completion time for the two experiments.

*** Remarks**

For the tasks in our example, both *hc* and *he* need to download data from *ha* and *hb* simultaneously, and *ha* and *hb* need to serve as HTTP server for multiple end hosts (*hc* and *he*). Therefore, *ha* and *hb* need to act as a Multi-thread HTTP Server, while *hc* and *he* need to execute download tasks in parallel.

To enable Multi-thread HTTP Server at *ha* and *hb*, create a file *MultithreadedSimpleHTTPServer.py* (at the same place with your topology) and implement it following [21]. Run the following command at xterm of *ha* and *hb* to start Multi-thread HTTP Server.

```
python MultithreadedSimpleHTTPServer.py 80
```

To execute download tasks in parallel at hosts, e.g., *hc*, run the scripts in the following manner and output the execute messages into files for reference.

```
python download_hb.py 30 > hc_hb.txt & python download_ha.py 20
> hc_ha.txt &
```

(Include the screenshots of both experiment results in your lab report and prepare a demonstration for the demo session.)

References

[1] OpenFlow Tutorial
http://archive.openflow.org/wk/index.php/OpenFlow_Tutorial

[2] Software Defined Networking Course of Princeton
https://www.cs.princeton.edu/courses/archive/fall13/cos597E/index.html

[3] No Host-only Adapter Selected
http://askubuntu.com/questions/198452/no-host-only-adapter-selected

[4] Mininet Installation Instructions
https://www.youtube.com/watch?v=yNmv7GiHIKE

[5] Dpctl Documentation
https://github.com/CPqD/ofsoftswitch13/wiki/Dpctl-Documentation

[6] Man Page of dpctl
http://ranosgrant.cocolog-nifty.com/openflow/dpctl.8.html

[7] Mininet Documentation
https://github.com/mininet/mininet/wiki/Documentation

[8] SDN 101: Using Mininet and SDN Controllers
http://pakiti.com/sdn-101-using-mininet-and-sdn-controllers/

[9] Introduction to Mininet
https://github.com/mininet/mininet/wiki/Introduction-to-Mininet

[10] Mininet Python API Reference Manual
http://mininet.org/api/annotated.html

[11] Github of POX
http://github.com/noxrepo/pox

[12] The Difference Between a Router, Switch and Hub
http://www.webopedia.com/DidYouKnow/Hardware_Software/router_switch_hub.asp

[13] Software Defined Networking Lab
http://noise.gatech.edu/classes/cs8803sdn/fall2014/

[14] Introduction to Mininet: Multipath Routing
https://github.com/mininet/mininet/wiki/Introduction-to-Mininet#multipath

[15] FAQ of Mininet
https://github.com/mininet/mininet/wiki/FAQ#ethernet-loops

[16] POX: openflow.spanning_tree
https://openflow.stanford.edu/display/ONL/POX+Wiki#POXWiki-openflow.spanning_tree

[17] POX: IP Load Balancer
https://openflow.stanford.edu/display/ONL/POX+Wiki#POXWiki-misc.ip_loadbalancer

[18] Load balancing with multiple switches
http://pox-dev.noxrepo.narkive.com/mvip3Oxn/load-bancing-with-multiple-switchs

[19] OpenFlow Protocol: Class Match
http://archive.openflow.org/doc/gui/org/openflow/protocol/Match.html

[20] OpenFlow: Classification
http://flowgrammable.org/sdn/openflow/classifiers/

[21] MultithreadedSimpleHTTPServer
https://github.com/Nakiami/MultithreadedSimpleHTTPServer