Lab 2 - Part 1
OpenFlow python controller (POX)

Objective

Link an external controller to mininet optical and write a simple controller in python.

What you will learn

- Create a simple openflow controller using the python-based POX controller framework.
- Construct a tree topology in Mininet using python.
- Use the controller to forward traffic in the tree topology network. (Layer 2 forwarding App)
- Modify the Layer 2 forwarding App with functionality to block specific traffic (Firewall App)

Prerequisites

Defining Topologies in Mininet.
Knowledge of Arp

Openflow controller

In this lab, we will create a rudimentary network tree topology in Mininet. Instead of inserting rules manually into the Openflow switch, we will use a controller which is external to Mininet. The Openflow controller we will use is called POX and is written in Python (https://noxrepo.github.io/pox-doc/html/#). It thus uses Python object oriented methodology to obfuscate the complexity of the controller. This makes it easy to write complex controllers, in a short few lines.
We create a shell of the controller using the POX *_handle_ConnectionUp()* and *_handle_PacketIn() functions.* These functions, respectively, listen out for connection requests from switches, and for packets which are transmitted from the switch to the controller.

Create the follow file *simple_controller.py* in directory ~/pox/pox/misc/

```
from pox.core import core
import pox.lib.packet as pkt
```

```
import pox.openflow.libopenflow_01 as of
from pox.lib.util import dpidToStr
from pox.lib.addresses import EthAddr

log = core.getLogger()

table={}

def launch ():
   core.openflow.addListenerByName("ConnectionUp", _handle_ConnectionUp)
   core.openflow.addListenerByName("PacketIn",  _handle_PacketIn)
   log.info("Switch running.")

def _handle_ConnectionUp ( event):
   log.info("Starting Switch %s", dpidToStr(event.dpid))
   msg = of.ofp_flow_mod(command = of.OFPFC_DELETE)
   event.connection.send(msg)


def _handle_PacketIn ( event):
   dpid = event.connection.dpid
   sw=dpidToStr(event.dpid)
   inport = event.port
   packet = event.parsed
   print("Event: switch %s port %s packet %s" % (sw, inport, packet))
```

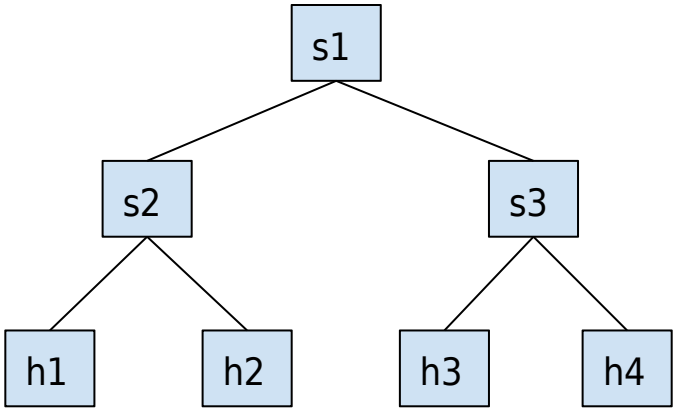
To run this controller, enter the following

```
cd ~/pox
./pox.py log.level --DEBUG misc.simple_controller
```

All this controller does is reports back connection requests from switches and any packets forwarded to it from the switch. We will use this with a simple topology created in Mininet.

Create Tree topology

The Mininet tree topology uses 2 layers of switches, to which 4 hosts h1, h2, h3 and h4 are connected, as per the following diagram.



You will see in the following program, that we define the variable controller=RemoteController when we instantiate Mininet. Mininet will attempt to connect to a controller listening on TCP port 6633 located on the same machine (localhost, or ip address 127.0.0.1). All openflow switches will connect using their own dedicated Openflow channels to the controller, and accept rules from the controller. We define each of the four hosts with ip addresses and simple MAC addresses. We connect the four hosts to the 2 lower layer switches s2 and s3, and create links from the s2 and s3 to the root switch s1.

This is the script to prepare a mininet topology

```
from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.cli import CLI
from mininet.log import setLogLevel, info

def treeTopo():
    net = Mininet( controller=RemoteController )

    info( '*** Adding controller\n' )
    net.addController('c0')

    info( '*** Adding hosts\n' )
    h1 = net.addHost( 'h1', ip='10.0.0.1', mac='00:00:00:00:00:01' )
    h2 = net.addHost( 'h2', ip='10.0.0.2', mac='00:00:00:00:00:02' )
    h3 = net.addHost( 'h3', ip='10.0.0.3', mac='00:00:00:00:00:03' )
    h4 = net.addHost( 'h4', ip='10.0.0.4', mac='00:00:00:00:00:04' )

    info( '*** Adding switches\n' )
    s1 = net.addSwitch( 's1' )
    s2 = net.addSwitch( 's2' )
    s3 = net.addSwitch( 's3' )
```

```
    info( '*** Creating links\n' )
    net.addLink( h1, s1 )     # bandwidth (in Mb/s) and delay can be added as follows
                              # net.addLink( h1, s1 ,cls=TCLink, bw=50,delay='10ms')

    net.addLink( h2, s1 )
    net.addLink( h3, s2 )
    net.addLink( h4, s2 )

    root = s1
    layer1 = [s2,s3]

    for idx,l1 in enumerate(layer1):
        net.addLink( root,l1 )

    info( '*** Starting network\n')

################################  this part up to here creates the mininet topology
    net.start()    # this commands starts the topology in mininet

    info( '*** Running CLI\n' )
    CLI( net )      # this command creates the mininet prompt, so that now we can interact with      # mininet through the same terminal (or link from an external controller).

    info( '*** Stopping network' )
    net.stop()     # this command is invoked after we enter the exit command in the mininet prompt

if __name__ == '__main__':
    setLogLevel( 'info' )
    treeTopo()
```

Unlike standard ethernet switches, which broadcast out all ports to learning the location of all hosts and devices, the topology we have just described is dumb and has no forwarding rules. It must rely on rules to be inserted manually or from the controller to forward traffic.

Run simple_controller and tree0.py topology

Run the controller, followed by the tree0.py topology maker, in separate screens.

```
mininet@mininet-vm:~/pox$ ./pox.py log.level --DEBUG misc.simple_controller
POX 0.7.0 (gar) / Copyright 2011-2020 James McCauley, et al.
INFO:misc.simple_controller:Switch running.
DEBUG:core:POX 0.7.0 (gar) going up...
DEBUG:core:Running on CPython (3.8.5/Jul 28 2020 12:59:40)
DEBUG:core:Platform is Linux-5.4.0-42-generic-x86_64-with-glibc2.29
WARNING:version:Support for Python 3 is experimental.
INFO:core:POX 0.7.0 (gar) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
INFO:misc.simple_controller:Starting Switch 00-00-00-00-00-01
INFO:openflow.of_01:[00-00-00-00-00-02 3] connected
INFO:misc.simple_controller:Starting Switch 00-00-00-00-00-02
INFO:openflow.of_01:[00-00-00-00-00-03 4] connected
INFO:misc.simple_controller:Starting Switch 00-00-00-00-00-03
```

```
mininet@mininet-vm:~$ sudo python tree0.py
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Connecting to remote controller at 127.0.0.1:6633
*** Adding hosts
*** Adding switches
*** Creating links
*** Starting network
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Running CLI
*** Starting CLI:
```

Left is controller. Right is Mininet topology

You will see the switches s1, s1 and s2 register with the controller. No traffic is generated yet on the network. We can generate simple traffic by pinging host h2 from h1.

```
Event: switch 00-00-00-00-00-01 port 1 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
Event: switch 00-00-00-00-00-01 port 1 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
Event: switch 00-00-00-00-00-01 port 1 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
Event: switch 00-00-00-00-00-01 port 1 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
Event: switch 00-00-00-00-00-01 port 1 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
Event: switch 00-00-00-00-00-01 port 1 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
```

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
```

Left is controller. Right is Mininet topology

The ping fails from h1 to h2. Why ? Because, the switches have empty flow tables, and do not know how to transmit traffic between h2 and h1. Refer to Lab 2 to answer this question. Ping uses ip addresses to refer to hosts, but switches uses ethernet addresses to refer hosts. Arp is used to associate the host ip addresses and ethernet addresses at the hosts.

Consider: On a large, open network, where users are trying to send confidential traffic, how secure is this approach (Arp) in how hosts learn the ethernet / mac addresses of remote hosts.

Forwarding App

Enhance the shell controller to learn the layer 2 forwarding rules.

We can use the *table{}* data structure in our simple_controller to record where hosts are on network. When we generate a ping, the network stack on the source machine sends an arp request that asks "*which host (i.e. ethernet address) has this ip address?*". We modify our controller to flood this request out all its ports (except the port it came in on). The host with that ip address responds with a response "*I am the host (i.e. ethernet address) with that ip address*", which the controller floods to all the other ports. This way, even though one host asks the Arp request question, all hosts learn the answer.

The controller can then generate flow table rules in each switch that match source and destination ethernet addresses, and forwards the traffic out the correct port.

```
def _handle_PacketIn ( event):
    dpid = event.connection.dpid
    sw=dpidToStr(event.dpid)
    inport = event.port
    packet = event.parsed
    print("Event: switch %s port %s packet %s" % (sw, inport, packet))

    # Learn the source
    table[(event.connection,packet.src)] = event.port

    dst_port = table.get((event.connection,packet.dst))

    if dst_port is None:
        # The switch does not know the destination, so sends the message out all ports.
        # We could use either of the special ports OFPP_FLOOD or OFP_ALL.
        #  But not all switches support OFPP_FLOOD.
        msg = of.ofp_packet_out(data = event.ofp)
        msg.actions.append(of.ofp_action_output(port = of.OFPP_ALL))
        event.connection.send(msg)
    else:
        # The switch knows the destination, so can route the packet. We also install the forward rule into the switch
        msg = of.ofp_flow_mod()
        msg.priority=100
        msg.match.dl_dst = packet.src
        msg.match.dl_src = packet.dst
        msg.actions.append(of.ofp_action_output(port = event.port))
        event.connection.send(msg)

        # We must forward the incoming packet...
        msg = of.ofp_packet_out()
        msg.data = event.ofp
        msg.actions.append(of.ofp_action_output(port = dst_port))
        event.connection.send(msg)

        log.debug("Installing %s <-> %s" % (packet.src, packet.dst))
```

This time when the ping is generated from h1 to h2, it succeeds. The controller transmits the arp request and response packets correctly between 2 hosts, and the ping traffic is transmitted also correctly.

```
Event: switch 00-00-00-00-00-01 port 1 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
Event: switch 00-00-00-00-00-02 port 3 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
Event: switch 00-00-00-00-00-03 port 1 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
Event: switch 00-00-00-00-00-01 port 2 packet [00:00:00:00:00:02>00:00:00:00:00:01 ARP]
DEBUG:misc.simple_controller:Installing 00:00:00:00:00:02 <-> 00:00:00:00:00:01
```

```
mininet> h1 ping 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=23.1 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.116 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.118 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.122 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.122 ms
```

Firewall App

Firewall Rules and POX Implementation

Using the simple_controller, we can now construct slightly more complex functionality, such as a firewall. A firewall blocks traffic under certain conditions. In the following exercise, the conditions are

1. By default, all hosts may communicate with every other host except ...
2. h1 and h2 may not communicate with each other
3. h3 and h4 may not communicate with each other

We modify the simple_controller with 2 tuples for the rules 2 and 3.

rules=[['00:00:00:00:00:01','00:00:00:00:00:02'],['00:00:00:00:00:03', '00:00:00:00:00:04']]

We insert these rules into the switches at initialisation stage of the simple_controller, when _handle_ConnectionUp() is triggered.

```
def _handle_ConnectionUp ( event):
   log.info("Starting Switch %s", dpidToStr(event.dpid))
   msg = of.ofp_flow_mod(command = of.OFPFC_DELETE)
   event.connection.send(msg)

   for rule in rules:
      block = of.ofp_match()
      block.dl_src = EthAddr(rule[0])
      block.dl_dst = EthAddr(rule[1])
      flow_mod = of.ofp_flow_mod()
      flow_mod.match = block
      flow_mod.priority = 32000
      event.connection.send(flow_mod)
```

You will notice that the priority of these block rules is high (32000) and supercedes any subsequently learned rules with a lower priority, which we had already defined as 100.

Test the firewall application against the topology

To test the topology and the firewall application, we use the Mininet pingall application to ping from every host to every other host.

mininet> pingall
**** Ping: testing ping reachability*
h1 -> X h3 h4
h2 -> X h3 h4
h3 -> h1 h2 X
h4 -> h1 h2 X
**** Results: 33% dropped (8/12 received)*

Here we see that h1 can reach h3 and h4, but is blocked to h2, as expected. Likewise, the firewall blocks h2 to h1, blocks h3 to h4, and blocks h4 to h3.

You should inspect the flow table rules in the switches. You will notice the higher priority rules for implementing blocking of traffic, as well as lower priority rules for passing traffic.

mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1
cookie=0x0, duration=171.743s, table=0, n_packets=27, n_bytes=2142, priority=32000,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02 actions=drop
cookie=0x0, duration=171.742s, table=0, n_packets=0, n_bytes=0, priority=32000,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:04 actions=drop
cookie=0x0, duration=113.474s, table=0, n_packets=0, n_bytes=0, priority=100,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02 actions=output:"s1-eth2"
cookie=0x0, duration=113.474s, table=0, n_packets=12, n_bytes=728, priority=100,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
cookie=0x0, duration=72.563s, table=0, n_packets=6, n_bytes=532, priority=100,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03 actions=output:"s1-eth3"
cookie=0x0, duration=72.563s, table=0, n_packets=7, n_bytes=574, priority=100,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
cookie=0x0, duration=54.864s, table=0, n_packets=3, n_bytes=238, priority=100,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:04 actions=output:"s1-eth3"

cookie=0x0, duration=54.864s, table=0, n_packets=4, n_bytes=280, priority=100,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
cookie=0x0, duration=44.847s, table=0, n_packets=3, n_bytes=238, priority=100,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:03 actions=output:"s1-eth3"
cookie=0x0, duration=44.847s, table=0, n_packets=4, n_bytes=280, priority=100,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:02 actions=output:"s1-eth2"
cookie=0x0, duration=44.832s, table=0, n_packets=3, n_bytes=238, priority=100,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:04 actions=output:"s1-eth3"
cookie=0x0, duration=44.832s, table=0, n_packets=4, n_bytes=280, priority=100,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:02 actions=output:"s1-eth2"