SDN and OPenFlow

Introduction

OpenFlow is one of the first standards in Software-Defined Network (SDN) and it is quite a new topic, with research still ongoing. The idea behind OpenFlow is to *only* have data-plane on routers/switches, with the control and management plane offloaded to a Controller which makes all the decisions, thereby increasing processing power on the routers/switches.

The objective of this lab is to learn about the basics of SDN, SDN Controllers and OpenFlow protocol.

part I – OPEN Vswitch

The Open vSwitch (OVS) is an open-source multilayer software switch that can be configured as a standard or software-defined switch in a virtual environment. It supports standard 802.1Q VLAN, OpenFlow protocol, NetFlow and sFlow, among many other features.

Switches in Mininet can be configured as OVS or OVSK (OVS Kernel mode) switches to support OpenFlow messages and operations. One of OVS utilities, ovs-ofctl, can be using for sending OpenFlow messages to Open vSwitch. In Mininet, this utility can be used through function call ‘dpctl’.

1. Create an instance from the snapshot titled “Student\_Image” and connect to it via the GUI (hereafter Mininet Window).
2. Start mininet using the following command:

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

This command creates a simple switch with three hosts h1, h2 and h3. The mac addresses are set to 01, 02 and 03, respectively. The switch is configured as OVS kernel based, and is made to use a remote controller. We will not set up the remote controller here; instead we will be using dpctl to manually control the flow table of the switch.

Use ‘net’ to confirm the topology. Use ‘h1 ifconfig’, ‘h2 ifconfig’ and ‘h3 ifconfig’ to find the IP and MAC addresses of the hosts.

Try ‘pingall’. Does it work? Why?

Use dpctl command to check the flow table on the switch:

|  |
| --- |
| Mininet> dpctl dump-flows |

The flow table is empty. Given that no controller is running, no flow is installed or learned, and the switch is unable to forward any packets.

Now let’s see if we can add flow entries to the table manually to facilitate the work. Let’s start by configuring IP flow entries. These would be similar to configuring the OVS as a router:

|  |
| --- |
| Mininet> dpctl add-flow ip,nw\_dst=10.0.0.1,actions=output:1  Mininet> dpctl add-flow ip,nw\_dst=10.0.0.2,actions=output:2 |

These commands add two entries to the flow table, directing the switch to forward IP Packets destined to network layer address 10.0.0.1 (h1) to port 1, and IP packets destined to network layer address 10.0.0.2 (h2) to port 2. For more information about the syntax of the add-flow and similar commands, check the “Flow Syntax” section of the ovs-ofctl man page here:

<https://www.openvswitch.org/support/dist-docs-2.5/ovs-ofctl.8.txt>

Use dpctl dump-flows to confirm that the flows have been added to the table.

Now try pingall again. Does it work? Why not? Try h1 ping h2. What message do you receive? Execute a ‘arp -a’ on h1 and note the output.

In order to troubleshoot the situation here, we can use Wireshark on h1. Open a terminal on h1 using xterm h1 and run wireshark. Start capturing traffic on h1-eth0.

Now do h1 ping h2. What messages do you see on Wireshark? What does it indicate?

Open a terminal on h2 and run wireshark. Does the ARP message reach h2?

1. Obviously the ARP broadcasts do not get through the switch because there is no flow entry to tell switch how to deal with these packets. Remember that an OpenFlow switch will not handle any packet that it does not know what to do with it.

Add a flow to allow broadcast messages to be flooded to every port:

|  |
| --- |
| Mininet> dpctl add-flow dl\_type=0x0806,dl\_dst=ff:ff:ff:ff:ff:ff,actions=flood |

In this flow entry, the datalink destination address is set to broadcast for ARP (Ethernet Protocol type 0x0806) and the switch is instructed to flood the packet. Use dump-flows to confirm.

Now try h1 ping h2 again. Does it work? What do you see in wireshark captures on h1 and h2 and how is it different from last time? Execute a ‘arp -a’ on h1 and note the output.

1. Now the ARP broadcast reaches h2 and it responds, but the response does not get through the switch to h1. The reason is that there is no flow entries to tell the switch what to do with a layer-2 packet going to h1 (our previous IP flow only works at layer 3).

Add the following entries to allow switching based on MAC address:

|  |
| --- |
| Mininet> dpctl add-flow dl\_dst=0:0:0:0:0:1,actions=output:1  Mininet> dpctl add-flow dl\_dst=0:0:0:0:0:2,actions=output:2 |

Check the flow table to confirm that the new flows have been installed.

Try h1 ping h2 again and confirm that it works. Try pingall. Which pings still don’t work?

In fact, we did not need to configure any network-layer flow for the purpose of this operation. The three DL flows (ARP broadcast flood, h1 mac and h2 mac) would have been sufficient.

part II –Custom Topologies

You can create a custom topology in Mininet by writing a Python script. For simplicity, it is always easier to edit one of the sample files, such as custom/topo-2sw-2host.py

1. Create a network via a Python Script and run it using the RYU controller. Some information is given below:
   1. Running the RYU Controller – “ryu-manager --verbose ryu.app.example\_switch\_13 ryu.app.ofctl\_rest” on the first terminal
   2. In another terminal, run the following script:

|  |
| --- |
| from mininet.net import Mininet  from mininet.node import Controller, OVSKernelSwitch, RemoteController  from mininet.cli import CLI  from mininet.link import TCLink  from mininet.log import setLogLevel, info  def netTAR():  net = Mininet(controller=RemoteController, switch=OVSKernelSwitch)  c1 = net.addController('c1', controller=RemoteController,ip="127.0.0.1", port=6653)  #Adding Hosts  h1 = net.addHost( 'h1', ip='192.168.1.10' )  h2 = net.addHost( 'h2', ip='192.168.1.11' )  #Adding switches  s1 = net.addSwitch( 's1' , protocols="OpenFlow13")  #Adding links  s1.linkTo( h1 )  s1.linkTo( h2 )  #create the network  net.build()  net.start()  net.staticArp()  CLI( net )  net.stop()  if \_\_name\_\_ == '\_\_main\_\_':  setLogLevel( 'info' )  netTAR() |

Sample Mininet output



Sample Output of the Controller

|  |
| --- |
| $ ryu-manager --verbose ryu.app.example\_switch\_13 ryu.app.ofctl\_rest  loading app ryu.app.example\_switch\_13  loading app ryu.app.ofctl\_rest  loading app ryu.controller.ofp\_handler  instantiating app None of DPSet  creating context dpset  creating context wsgi  instantiating app ryu.app.example\_switch\_13 of ExampleSwitch13  instantiating app ryu.app.ofctl\_rest of RestStatsApi  instantiating app ryu.controller.ofp\_handler of OFPHandler  BRICK dpset  CONSUMES EventOFPStateChange  CONSUMES EventOFPPortStatus  CONSUMES EventOFPSwitchFeatures  BRICK ExampleSwitch13  CONSUMES EventOFPPacketIn  CONSUMES EventOFPSwitchFeatures  BRICK RestStatsApi  CONSUMES EventOFPSwitchFeatures  CONSUMES EventOFPQueueGetConfigReply  CONSUMES EventOFPRoleReply  CONSUMES EventOFPStatsReply  CONSUMES EventOFPDescStatsReply  CONSUMES EventOFPFlowStatsReply  CONSUMES EventOFPAggregateStatsReply  CONSUMES EventOFPTableStatsReply  CONSUMES EventOFPTableFeaturesStatsReply  CONSUMES EventOFPPortStatsReply  CONSUMES EventOFPQueueStatsReply  CONSUMES EventOFPQueueDescStatsReply  CONSUMES EventOFPMeterStatsReply  CONSUMES EventOFPMeterFeaturesStatsReply  CONSUMES EventOFPMeterConfigStatsReply  CONSUMES EventOFPGroupStatsReply  CONSUMES EventOFPGroupFeaturesStatsReply  CONSUMES EventOFPGroupDescStatsReply  CONSUMES EventOFPPortDescStatsReply  BRICK ofp\_event  PROVIDES EventOFPStateChange TO {'dpset': {'main', 'dead'}}  PROVIDES EventOFPPortStatus TO {'dpset': {'main'}}  PROVIDES EventOFPSwitchFeatures TO {'dpset': {'config'}, 'ExampleSwitch13': {'config'}, 'RestStatsApi': {'main'}}  PROVIDES EventOFPPacketIn TO {'ExampleSwitch13': {'main'}}  PROVIDES EventOFPQueueGetConfigReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPRoleReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPStatsReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPDescStatsReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPFlowStatsReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPAggregateStatsReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPTableStatsReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPTableFeaturesStatsReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPPortStatsReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPQueueStatsReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPQueueDescStatsReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPMeterStatsReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPMeterFeaturesStatsReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPMeterConfigStatsReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPGroupStatsReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPGroupFeaturesStatsReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPGroupDescStatsReply TO {'RestStatsApi': {'main'}}  PROVIDES EventOFPPortDescStatsReply TO {'RestStatsApi': {'main'}}  CONSUMES EventOFPEchoReply  CONSUMES EventOFPEchoRequest  CONSUMES EventOFPErrorMsg  CONSUMES EventOFPHello  CONSUMES EventOFPPortDescStatsReply  CONSUMES EventOFPPortStatus  CONSUMES EventOFPSwitchFeatures  (2381) wsgi starting up on http://0.0.0.0:8080  connected socket:<eventlet.greenio.base.GreenSocket object at 0x7fa30edfb630> address:('127.0.0.1', 32824)  EVENT ofp\_event->dpset EventOFPStateChange  connected socket:<eventlet.greenio.base.GreenSocket object at 0x7fa30edfbe80> address:('127.0.0.1', 32826)  hello ev <ryu.controller.ofp\_event.EventOFPHello object at 0x7fa30edfbf60>  move onto config mode  EVENT ofp\_event->dpset EventOFPSwitchFeatures  EVENT ofp\_event->ExampleSwitch13 EventOFPSwitchFeatures  switch features ev version=0x4,msg\_type=0x6,msg\_len=0x20,xid=0xd80505f,OFPSwitchFeatures(auxiliary\_id=0,capabilities=79,datapath\_id=1,n\_buffers=0,n\_tables=254)  move onto main mode  EVENT ofp\_event->dpset EventOFPStateChange  DPSET: register datapath <ryu.controller.controller.Datapath object at 0x7fa30edfb9e8>  EVENT ofp\_event->ExampleSwitch13 EventOFPPacketIn  packet in 1 0e:2c:a4:28:d3:0a 33:33:00:00:00:16 1  EVENT ofp\_event->dpset EventOFPPortStatus  DPSET: A port was modified.(datapath id = 0000000000000001, port number = 2)  EVENT ofp\_event->dpset EventOFPPortStatus  DPSET: A port was modified.(datapath id = 0000000000000001, port number = 4294967294)  EVENT ofp\_event->dpset EventOFPPortStatus  DPSET: A port was modified.(datapath id = 0000000000000001, port number = 1)  EVENT ofp\_event->dpset EventOFPPortStatus  DPSET: A port was modified.(datapath id = 0000000000000001, port number = 3)  connected socket:<eventlet.greenio.base.GreenSocket object at 0x7fa30efaba20> address:('127.0.0.1', 32828)  hello ev <ryu.controller.ofp\_event.EventOFPHello object at 0x7fa30f070a58>  move onto config mode  EVENT ofp\_event->dpset EventOFPSwitchFeatures  EVENT ofp\_event->ExampleSwitch13 EventOFPSwitchFeatures  switch features ev version=0x4,msg\_type=0x6,msg\_len=0x20,xid=0x439ec2e7,OFPSwitchFeatures(auxiliary\_id=0,capabilities=79,datapath\_id=2,n\_buffers=0,n\_tables=254)  move onto main mode  EVENT ofp\_event->dpset EventOFPStateChange  DPSET: register datapath <ryu.controller.controller.Datapath object at 0x7fa30edfbcf8>  EVENT ofp\_event->dpset EventOFPPortStatus  EVENT ofp\_event->dpset EventOFPPortStatus  EVENT ofp\_event->dpset EventOFPPortStatus  EVENT ofp\_event->dpset EventOFPPortStatus  DPSET: A port was modified.(datapath id = 0000000000000002, port number = 3)  DPSET: A port was modified.(datapath id = 0000000000000002, port number = 2)  DPSET: A port was modified.(datapath id = 0000000000000002, port number = 4294967294)  DPSET: A port was modified.(datapath id = 0000000000000002, port number = 1)  EVENT ofp\_event->ExampleSwitch13 EventOFPPacketIn  packet in 2 5e:37:07:22:7e:ac 33:33:00:00:00:16 3 |

Part III - RYU API

1. Using information found at <https://inside-openflow.com/2016/06/23/interactive-ryu-with-postman/>, create a small Python Script which calls the RYU API with API base of <http://localhost:8080/> to extract the flows present on the network

|  |
| --- |
| import requests  import json  api\_url\_base = 'http://localhost:8080/'  flows = 'stats/flow/1'  response = requests.get(api\_url\_base + flows)  if response.status\_code == 200:  print json.loads(response.content.decode('utf-8'))  else:  print "Error" |

Output of running the above code (whilst the Script for the Network and the RYU Controller is running):

|  |
| --- |
| ubuntu@mininet-sept2020-baseimage:~$ sudo python api.py  {u'1': [{u'packet\_count': 3, u'hard\_timeout': 0, u'byte\_count': 238, u'duration\_sec': 57, u'actions': [u'OUTPUT:1'], u'duration\_nsec': 10000000, u'priority': 1, u'idle\_timeout': 0, u'cookie': 0, u'flags': 0, u'length': 96, u'table\_id': 0, u'match': {u'dl\_dst': u'aa:94:57:52:49:91', u'in\_port': 2}}, {u'packet\_count': 2, u'hard\_timeout': 0, u'byte\_count': 140, u'duration\_sec': 57, u'actions': [u'OUTPUT:2'], u'duration\_nsec': 6000000, u'priority': 1, u'idle\_timeout': 0, u'cookie': 0, u'flags': 0, u'length': 96, u'table\_id': 0, u'match': {u'dl\_dst': u'92:4a:16:af:09:3d', u'in\_port': 1}}, {u'packet\_count': 19, u'hard\_timeout': 0, u'byte\_count': 1454, u'duration\_sec': 59, u'actions': [u'OUTPUT:CONTROLLER'], u'duration\_nsec': 47000000, u'priority': 0, u'idle\_timeout': 0, u'cookie': 0, u'flags': 0, u'length': 80, u'table\_id': 0, u'match': {}}]} |

Part IV - FIREWALLS

1. Create the following file, which creates a network with 4 hosts and 4 switches.

|  |
| --- |
| 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' )  s4 = net.addSwitch( 's4' )    info( '\*\*\* Creating links\n' )  net.addLink( h1, s1 )  net.addLink( h2, s2 )  net.addLink( h3, s3 )  net.addLink( h4, s4 )  net.addLink( s1, s2 )  net.addLink( s1, s3 )  net.addLink( s3, s4 )    info( '\*\*\* Starting network\n')  net.start()    info( '\*\*\* Running CLI\n' )  CLI( net )    info( '\*\*\* Stopping network' )  net.stop()    if \_\_name\_\_ == '\_\_main\_\_':  setLogLevel( 'info' )  treeTopo() |

1. In this example, we will only block connectivity from H1 to H2, while allowing all others by default. To do so, navigate into ‘/home/ubuntu/pox/pox/misc’ and create a file called fw.py

|  |
| --- |
| from pox.core import core  import pox.openflow.libopenflow\_01 as of  from pox.lib.revent import \*  from pox.lib.addresses import EthAddr  rules = [['00:00:00:00:00:01','00:00:00:00:00:02']]  class SDNFirewall (EventMixin):    def \_\_init\_\_ (self):  self.listenTo(core.openflow)    def \_handle\_ConnectionUp (self, event):  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  event.connection.send(flow\_mod)    def launch ():  core.registerNew(SDNFirewall) |

1. Next, to run the network open up two SSH sessions to the VM, or two windows if you are using the GUI and run the following commands from the ‘/home/ubuntu/pox’:

|  |
| --- |
| From the /home/ubuntu/pox directory:  ./pox.py log.level --DEBUG openflow.of\_01 forwarding.l2\_learning misc.fw |

|  |
| --- |
| From the directory containing the network topology created in step 1:  sudo python *filename.py* |

1. If the setup was done correctly, the following output will confirm that pings are only blocked from H1 to H2, whilst others are not:

