Mininet and Pox

Let's look at a few examples of an OpenFlow controller interacting with some OpenFlow switches. The controller module is known as Pox, and is written in Python. The switches themselves are Linux implementations; the network is created within a single Linux host using Mininet.

Most of the material here is from the following sources:

- Mininet:
  - Walkthrough: mininet.org/walkthrough
  - FAQ: github.com/mininet/mininet/wiki/FAQ
  - Introduction: github.com/mininet/mininet/wiki/Introduction-to-Mininet
- Pox: openflow.stanford.edu/display/ONL/POX+Wiki

The Mininet system allows a single Linux host to be divided into "containers" that look like separate network **nodes**. It is possible to do this using a full virtual machine for each node (though Mininet does not); this is the "heavyweight" approach. Mininet uses what is sometimes called the "lightweight" approach. A node is represented as a "network namespace", using a feature of linux that makes it possible to have multiple "network namespaces" on one machine. Each node can have its own set of network interfaces. We choose the simplified (default) arrangement in which host nodes each get their own namespace but switch nodes all share the "root" namespace with the underlying host. This means that switch interfaces are all visible to the host system, and that switches can contact the controllers through their `localhost` interface.

The "containerized" network nodes all share the filesystem (and process space) of the host system. This sometimes complicates the creation of specialized services on different nodes. For example, if each node requires a fixed configuration file /etc/myservice.conf, there is a conflict.

Network **links** are emulated as virtual interfaces between the network namespaces. Using Linux **traffic control** (tc) mechanisms, a virtual link's delay and bandwidth can be limited so as to resemble a real link's delay and bandwidth constraints.

A range of switch implementations are available. Two basic examples are the Linux bridge and the Linux implementation of Open vSwitch, which is a "virtual" implementation of an OpenFlow switch.

We will run Mininet in a Linux virtual machine running under VirtualBox. This helps avoid conflicts between "real" services and the rather intrusive Mininet environment. (It also allows you to run the Mininet environment under Windows and iOS.) The userid and password are `mininet/mininet`; to get a root shell, use `sudo bash`.

You can download the mininet virtual machine from the mininet site, or you can download a zipfile of the disk image of my virtual machine (with X-windows pre-installed) here (it's a little over 2 GB). You will then need to do the following to create a VirtualBox virtual machine based on this disk image:

1. Unzip it to mininet-vm-x86_64.vmdk
2. Start the create-new-virtual-machine process in VirtualBox. The default settings should work.
3. When asked whether you want to create a new virtual disk, choose "use an existing virtual hard disk file". Click the VirtualBox file-manager icon, and add mininet-vm-x86_64.vmdk to the virtual-disk library. It should then be selectable as an option for your new virtual machine. (Accessing the VirtualBox file manager directly is relatively obscure.)

If you download my virtual machine, you start the windowing (X-windows) system with the shell command **startx**. You create terminal windows with CNTL-ALT-T. If the mouse is "captured" by the virtual machine, try pressing right-cntl.

In my examples, the `--switch ovsk` flag directs Mininet to use the kernel-level open vSwitch. We then use `ovs-ofctl` to examine the state of the OpenFlow controller. (It is not clear whether we need `ovsk`.)

Our basic example will be as follows (this is the actual Python code to create the topology in question). The file shown here does **not** generate a loop topology because the link between switches s1 and s4 is commented out; this will be called brectangle.py (broken rectangle). The corresponding file with the s1--s4 link active is called rectangle.py. What these files do is to create a Topo object, which we can then request at the Mininet command line.

```
"""rectangular (looping) topology example:

  h1---s1------s2---h2
       |        |
       |        |
  h4---s4------s3---h4

This will make simple hubs circulate packets endlessly.

Adding the 'topos' dict with a key/value pair to generate our newly defined
topology enables one to pass in '--topo=recttopo' from the command line.

We also need the following on the command line
        --controller remote
for this to work.

        mn --custom rectangle.py --topo recttopo

"""

from mininet.topo import Topo
from mininet.net import Mininet
from mininet.node import OVSSwitch, Controller, RemoteController

class RectTopo( Topo ):
    "Rectangular topology example."

    def __init__( self ):
        "Create custom topo."

        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switchesovs-ofctl
        h1 = self.addHost( 'h1' )
        h2 = self.addHost( 'h2' )
        h3 = self.addHost( 'h3' )
        h4 = self.addHost( 'h4' )
        s1 = self.addSwitch( 's1' )
        s2 = self.addSwitch( 's2' )
        s3 = self.addSwitch( 's3' )
        s4 = self.addSwitch( 's4' )

        # Add links
        self.addLink( h1, s1 )
        self.addLink( h2, s2 )
        self.addLink( h3, s3 )
        self.addLink( h4, s4 )

        self.addLink( s1, s2 )
        self.addLink( s2, s3 )
        self.addLink( s3, s4 )
#       self.addLink( s4, s1 )

topos = { 'recttopo': ( lambda: RectTopo() ) }
```

There is no **controller** specified above; we will run the external Pox controller (a so-called remote controller).

We now start Mininet (at a root shell prompt) with

```
mn --custom brectangle.py --topo recttopo --controller remote --switch ovsk --mac
```

The last, `--mac`, option means that network MAC addresses are chosen as small integers; for example, h1 has MAC address 00:00:00:00:00:01. The `--custom brectangle.py` flag specifies the file containing the Python topology definition, and the `--topo recttopo` flag identifies the Topo object defined in that file that we want to use.

It is possible to run everything about the Mininet network from within Mininet using Python code rather than the command line: not just topology setup, but also things like traffic generation and monitoring of the controller. We will here use Python only for the topology creation, however.

Mininet command line

The mn command above leaves us with the Mininet command-line interface prompt, mininet>. We can type commands to look at our network:

```
quit
help
nodes
links
dump            // displays most network information
h1 ifconfig     // prints interface configuration of host node h1
h1 ping h4      // causes a ping request to be sent from h1 to h4
xterm h1 h4  // starts an xterm terminal window on host nodes h1 and h4; very useful!
```

The mininet> prompt is not the same as the host-system root prompt, root#.

Starting xterms on various host nodes is the easiest way to generate traffic. For example, we can use netcat, with the listener on the h4 xterm and the client in the h1 xterm:

```
h4: netcat -l 5433
h1: netcat 10.0.0.4 5433    // We can no longer just write "h4" instead of "10.0.0.4"
```

After starting Mininet we start Pox, at a separate root prompt on our Mininet virtual machine. In this virtual machine, the mn command is on the $PATH, but pox.py is not; the pox directory is in /home/mininet:

```
pox/pox.py
```

This starts up the controller, which begins listening at the root-namespace localhost address. As all the Mininet *switches* share this namespace, they have an easy path to the controller. This means that no switch-to-controller communications ever depend on the (possibly evolving) Mininet network.

Demo Set 1: forwarding.hub

We will start with having the Pox controller tell the switches to act as traditional Ethernet hubs, broadcasting (or flooding) every packet out all interfaces except the arrival interface:

```
pox/pox.py --verbose forwarding.hub
```

We should see evidence that s1--s4 are connecting to the Pox controller:

```
INFO:openflow.of_01:[00-00-00-00-00-01 14] connected
INFO:forwarding.hub:Hubifying 00-00-00-00-00-01
INFO:openflow.of_01:[00-00-00-00-00-04 11] connected
INFO:forwarding.hub:Hubifying 00-00-00-00-00-04
INFO:openflow.of_01:[00-00-00-00-00-03 12] connected
INFO:forwarding.hub:Hubifying 00-00-00-00-00-03
```

```
INFO:openflow.of_01:[00-00-00-00-00-02 13] connected
INFO:forwarding.hub:Hubifying 00-00-00-00-00-02
```

These connections are made over the network, but sort of by "cheating": the Pox controller is at IP address `localhost` on the Mininet virtual machine. The switches s1--s4 do not use the network topology defined above to find the controller; each simply accesses it a localhost (via a TCP connection to port 6633).

The next step is to look at the controller's tables. To do this, we run ovs-ofctl from a root# prompt:

root# **ovs-ofctl dump-flows s1**
cookie=0x0, duration=256.031s, table=0, n_packets=0, n_bytes=0, idle_age=256, actions=**FLOOD**

The ovs-ofctl show s1 command is also useful. The above shows that s1's table has been set to flood packets: to transmit arriving packets on all other interfaces.

We can verify that our network is working:

mininet> h1 ping h4

We can also start tcpdump on host h4, and watch pings from h1 to h2:

h4# tcpdump -n -v -i h4-eth0

If switches rather than hubs were used, this should not happen; see below.

What happens if we try this with the full-loop rectangle.py instead of the linear brectangle.py? The switches have been told to FLOOD, and this is exactly what they will do. If we start Mininet, all is well until we try to ping h2 from h1:

mininet> h1 ping h2

We lose control of the xterm tcpdump. Sending CNTL-C to the mininet prompt concludes with:

5 packets transmitted, 5 received, +4286 duplicates

What happened?

Demo set 2: forwarding.l2_learning and friends

The pox option forwarding.l2_learning makes pox instruct the switches s1--s4 to behave *sort of* like standard Ethernet learning switches. One difference is that very individual flow entries are created; connections must match on IP source and destination and TCP port numbers. Separate flows are also created for ARP queries and ICMP (ping) messages.

We will edit pox/pox/forwarding/l2_learning to change idle_timeout and hard_timeout to 0. Otherwise, Pox deletes flows after 10 seconds, which makes them hard to monitor at the command line. Here's what I actually did:

i. In pox/pox/forwarding, copy l2_learning.py to l2nt_learning.py; make timeout changes
ii. change instances of "l2_learning" in the file to "l2nt_learning"
iii. start via pox/pox.py --verbose forwarding.l2**nt**_learning

We then start mininet with the brectangle.py option (l2_learning will fail badly with loops; the switch spanning-tree protocol is not implemented here!).

Checking the tables with ovs-ofctl dump-flows s1, we see nothing. This is because no flows have been created. We need this:

mininet> h1 ping h2

Now we see 5 flows: 3 for arp and 2 for icmp. If we try

mininet> pingall

we see something like 21 flows at s1 and s4 and 35 at s2 and s3. Again, all are for arp or icmp.

If we start tcpdump on h4, and then ping h2 from h1, we see nothing! This is what we expect with switches rather than hubs.

Now let's create some TCP flows! We start "netcat -l 5433" on h4 (using an xterm), and send from port 50001 on h1, at the mininet prompt:

```
mininet> netcat -p 50001 h4 5433
```

Now let's see what ovs-ofctl has to say:

```
ovs-ofctl dump-flows s2|egrep  ',tcp,'

cookie=0x0, duration=52.609s, table=0, n_packets=5, n_bytes=344, idle_age=44,
priority=65535,tcp,in_port=2,vlan_tci=0x0000,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:04,nw_src=10.0.0.1,
nw_dst=10.0.0.4,nw_tos=0,tp_src=50001,tp_dst=5433 actions=output:3

cookie=0x0, duration=52.599s, table=0, n_packets=3, n_bytes=206, idle_age=44,
priority=65535,tcp,in_port=3,vlan_tci=0x0000,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:01,nw_src=10.0.0.4,
nw_dst=10.0.0.1,nw_tos=0,tp_src=5433,tp_dst=50001 actions=output:2
```

One entry for each direction.

If we try again with the same source port, nothing changes. But if we change ports, new flow entries are created.

Do you see now why having a 10-second flow timeout is irritating?

If we instead use forwarding.l2_pairs, we can see that the flows are not connection-specific. We can stop and restart Pox with the new configuration without disrupting our running Mininet. We still need this, however:

```
mininet> pingall
```

Now ovs-ofctl shows a table like this (for s2):

```
cookie=0x0, ..., dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02 actions=output:1
cookie=0x0, ..., dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01 actions=output:2
cookie=0x0, ..., dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03 actions=output:3
cookie=0x0, ..., dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:01 actions=output:2
cookie=0x0, ..., dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:04 actions=output:3
cookie=0x0, ..., dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:04 actions=output:3
cookie=0x0, ..., dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:02 actions=output:1
cookie=0x0, ..., dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01 actions=output:2
cookie=0x0, ..., dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:02 actions=output:1
cookie=0x0, ..., dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:03 actions=output:3
```

That's both directions for 1↔2, 1↔3, 1↔4, 2↔4, 2↔3. What is missing? Why? (Hint: remember I took this snapshot at s2.)

Demo set 3: openflow.spanning_tree

For this demo we use the following:

```
pox/pox.py --verbose openflow.spanning_tree --no-flood --hold-down openflow.discovery forwarding.l2_multi
```

openflow.spanning_tree creates a spanning tree and enables flooding only along this tree.

openflow.discovery causes the controller to discover the switch topology.

forwarding.l2_multi allows the controller to use the discovered topology to set up forwarding tables to reach every destination.

Let's try it! For Mininet we use the rectangle.py topology, with a loop. We edit pox/pox/forwarding/l2_multi to set IDLE_TIMEOUT = 0.

We get endless DEBUG messages in the pox window. These don't matter.

The mininet pingall test works.

How does a ping from h1 get to h2? How about to h4? If we look at s1's table, we can find out.

dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02, ... actions=output:2

dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:04, ... actions=output:3

Different routes!