SEE
TEXT
ONLY

A Quick Tour OF SDN USING MININET

I've been running both FreeBSD and Linux uninterrupted since the mid-1990s. I've run many different Linux distributions over the years, most recently focusing on CentOS. I also have significant experience with Mac OS X and NetBSD and have experimented with many other BSD platforms. As a staunch agnostic with a firm belief in the value of open standards, I like to remain familiar with all the options in the POSIX world so I'm always prepared to choose the best tool for the job.

By Ayaka **Koshibe**

In the most basic sense, software-defined networking (SDN) can be thought of as an approach to building a network that can be managed as if it were one logical entity. An SDN-based network is typically built from programmable whitebox and software switches, and is managed from control applications that use their global view of the network to coordinate the switches to act as one. The result can look fairly unfamiliar to those used to "classic" networks that are configured on a per-device basis and the behavior of which is determined by distributed network protocols. Network emulators can be useful tools for gaining better insight into how these networks behave and are put together.

Mininet

Mininet is a fairly well-known emulator for SDN-based networks that was popularized by its ties to OpenFlow, a network control protocol from the dawn of SDN. It was also recently added to the ports collection. Using that as an occasion, here is a quick tour of Mininet in the form of an SDN primer.

Before we begin: Mininet depends on VIMAGE for emulating network hosts, so readers wishing to follow along will need a host with VIMAGE support. The ported version of Mininet also doesn't support the full set of features of the original and is very much a work-in-progress. It is also heavy-handed with cleanup, so it is best not run on machines used for hosting other jails or Open vSwitch instances.

Installation

Mininet can be installed like any other application: with `pkg(8)` as 'py27-mininet', or from the ports tree as 'net/mininet'.

The *mn* Command

The Mininet version of a "Hello world" is a tiny network launched with the *mn* command:

```
# mn --controller=ryu
*** Creating network
...
*** Starting CLI:
mininet>
```

This creates a network with two hosts connected through a switch controlled to act as a learning switch. It also launches a CLI for interacting with the network. For example, ***links*** shows all of the links in the network:

```
mininet> links
h1-eth0<->s1-eth1 (OK OK)
h2-eth0<->s1-eth2 (OK OK)
```

And ***dump*** shows information about the nodes in the network:

```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=8410>
<Host h2: h2-eth0:10.0.0.2 pid=8414>
<OVSSwitch s1: lo0:127.0.0.1,s1-eth1:None,s1-eth2:None pid=8420>
<Ryu c0: 127.0.0.1:6653 pid=8401>
```

Either ? or ***help*** will show all available commands.

Inspecting Control Traffic

The output of ***dump*** shows that each node has a name, one or more ports, and a PID of the bash process that represents it. It also shows that the hosts—actually vnet jails—are at 10.0.0.1 and 10.0.0.2 in this network, and the controller, Ryu, is listening for switches on port 6653. Ryu uses OpenFlow to program the switches that connect on this port. A typical way of troubleshooting OpenFlow switches and controllers is to inspect the control messages on this channel. We can try this by running tcpdump (or another packet analyzer) in another terminal:

```
# tcpdump -i lo0 port 6653
```

We should be able to see the keepalive `ECHO_REQUEST` and `ECHO_REPLY` messages sent between `s1` and `c0`. Next, ping one host from the other. The CLI interprets any commands after a host's name as bash commands to be run from that host:

```
mininet> h1 ping -c1 h2
```

A host's name is translated by the CLI into its corresponding IP address.
Alternatively, the ***pingall*** CLI command can be used to ping between all pairs of hosts:

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
```

In either case, we should see the switch (`localhost.<high port>`) sending `PACKET_IN` messages to the controller (`localhost.6653`), and `PACKET_OUT` and `FLOW_MOD` messages sent back by the controller in response. Switches use `PACKET_IN`s to send packets that it doesn't understand how to process to the controller, in this case, the ARP and ICMP messages. Controllers use `PACKET_OUT`s to instruct a switch to output a particular packet (the one sent in the `PACKET_IN`, so that it is not "lost"), and `FLOW_MOD`s to modify how a switch handles different types of traffic. No new `PACKET_IN`s should be generated by `s1` in response to another ping until the modifications expire from disuse.
A ctrl-D or the ***exit*** command will exit the CLI and tear the network down.

Experimenting with Controllers

The controller's role in the network can be directly demonstrated by running a network without one. This can be done by passing 'none' instead of 'ryu' to ***mn***'s ***—controller*** option. The hosts on this "headless" network should not be able to ping each other.
Another useful option is 'remote', which allows a network to use a controller running outside of Mininet's

control. Developers might test their controllers by pointing a Mininet network at them with this option. Assuming that a controller is running at 192.168.0.100 and listening on port 6633, the following will launch a network and connect the switch to it:

```
# mn --controller=remote,ip=192.168.0.100,port=6633
```

Creating Various Topologies

The --*topo* option is used to create various topologies with ***mn***. The *linear* and *tree* topologies are useful for creating larger loop-free networks, whereas the *torus* topology is useful for testing a controller's loop-handling abilities. Topologies are parameterized so that their sizes can be specified. For example, to create a tree three levels high and fanout of two:

```
# mn --controller=ryu, topo=tree,3,2
```

The *torus* also takes two values, and *linear* takes one.

Scripting with Mininet

Mininet can also be used as a collection of Python libraries for scripting experiments. With the caveat that they are in their original forms (and will most likely not work on FreeBSD), the package includes several example scripts that demonstrate how to create custom topologies, network components, and experiments. As with other applications that come with examples, they should be found under `/usr/local/share/examples/mininet/`. But, as a small example, the following script defines a custom topology resembling mn's default topology, uses host `h1` to ping `h2`'s address, and exits:

```
from mininet.topo import Topo
from mininet.net import Mininet

class MinimalTopo (Topo):
  def build(self):
    h1 = self.addHost('h1', ip='192.168.0.1')
    h2 = self.addHost('h2', ip='192.168.0.2')
    s1 = self.addSwitch('s1')

    self.addLink(h1, s1)
    self.addLink(s1, h2)

net = Mininet(topo=MinimalTopo())
net.start()
h1 = net.getNodeByName('h1')
print(h1.cmd('ping -c1 192.168.0.2'))
net.stop()
```

Once saved, it can be run like any Python script:

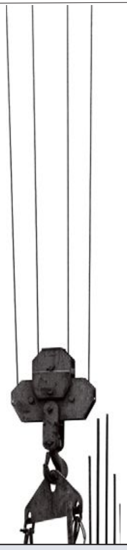
```
# python example.py
```

Finding Out More

While only a subset of the upstream features are supported by this port, the main project maintains resources that should provide a better idea of how Mininet can be used. These can be found at:
https://github.com/mininet/mininet/wiki/Documentation

And the port itself is maintained at:
https://github.com/akoshibe/mininet

So this concludes our whirlwind tour of Mininet. Hopefully, it serves as a decent starting point for those interested in exploring the area of SDN. •



Ayaka Koshibe became involved in the area of SDN as a college student assisting in the deployment of infrastructure for the GENI OpenFlow campus trials. She currently works at Big Switch Networks as a member of the SDN controller platform team, and is also both maintainer and upstream for the Mininet port for FreeBSD and OpenBSD.