Mininet Basics - for Mac

Ben Cravens - 30th November 2021

(Note this setup assumes you are using a mac, should be OK on windows & linux as well as long as you can get a VM running.)

Mininet is a network virtualiation tool. You can use it to simlate a network. The topology and nodes in the network can be specified programmatically with the mininet API for python, or via command line args.

- Download virtual box
https://www.virtualbox.org
- Verify the SHA256 https://www.virtualbox.org/download/hashes/6.1.30/SHA256SUMS
- OR MD5 checksums https://www.virtualbox.org/download/hashes/6.1.30/MD5SUMS

```
$ shasum –a 256 download
$ md5 download
```

Download an Ubuntu iso from the ubuntu website:
http://releases.ubuntu.com/focal/
Again, verify the checksum. I used SHA256
http://releases.ubuntu.com/focal/SHA256SUMS

- Open virtual box
- Click “New”
- Set up the VM. Name it “Ubuntu”. I recommend at least 2.5 GB RAM.
- All other settings can be default.
- Now to load the iso...
- Go to Settings->Storage.
- Click on the little disc on the left, then click on the disc on the right next to “Optical Drive”. From there select your iso.

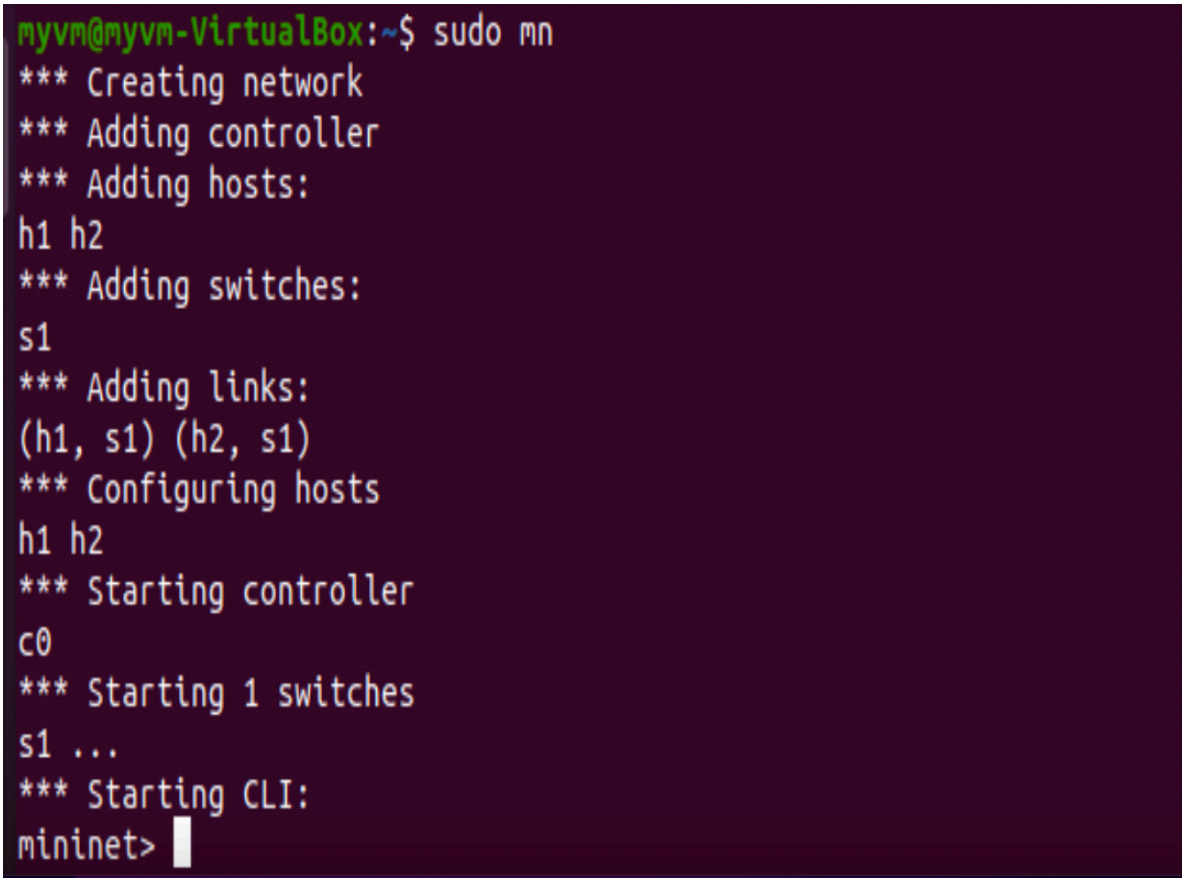
Boot up ubuntu!

Follow the Mininet tutorial to learn Mininet basics..

http://mininet.org/walkthrough/

Launch Mininet: initialises the “*minimal*” configuration: a simple network topology of 2 hosts, a switch, and a network controller.

```
$ sudo mn
```



We can prepend a command with a host name to run the command on that host.
For example:

```
$ h1 ifconfig –a
```

Runs the ifconfig command on the first host h1.

```
mininet> h1 ifconfig -a
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.1  netmask 255.0.0.0  broadcast 10.255.255.255
        inet6 fe80::1810:f8ff:fef5:900a  prefixlen 64  scopeid 0x20<link>
        ether 1a:10:f8:f5:90:0a  txqueuelen 1000  (Ethernet)
        RX packets 37  bytes 4132 (4.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 11  bytes 866 (866.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

We can see the loopback address as well as the outward link going into the switch, eth0. We don't see any other links because the host runs in its own space defined by Mininet. However only the network is virtualised, each host can see the same processes.

Pinging one host from another:

```
$ h1 ping –c 3 h2
```

```
mininet> h1 ping -c 3 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=5.31 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.409 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.080 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.080/1.933/5.311/2.392 ms
mininet>
```

You can also run other processes on hosts, like a simple web server.

I run a simple web server on h1:

```
$ h1 python3 -m http.server 80 &
```

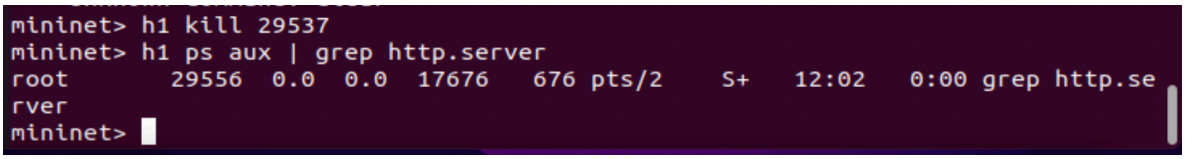
And grab it from h2:

```
$ h2 wget -O - h1
```

After this is done you can kill the process on h1

$ h1 ps aux | grep http.server

$ kill (process id)



You can also do this quickly in an automated way:

```
$ sudo mn --test pingpair
```

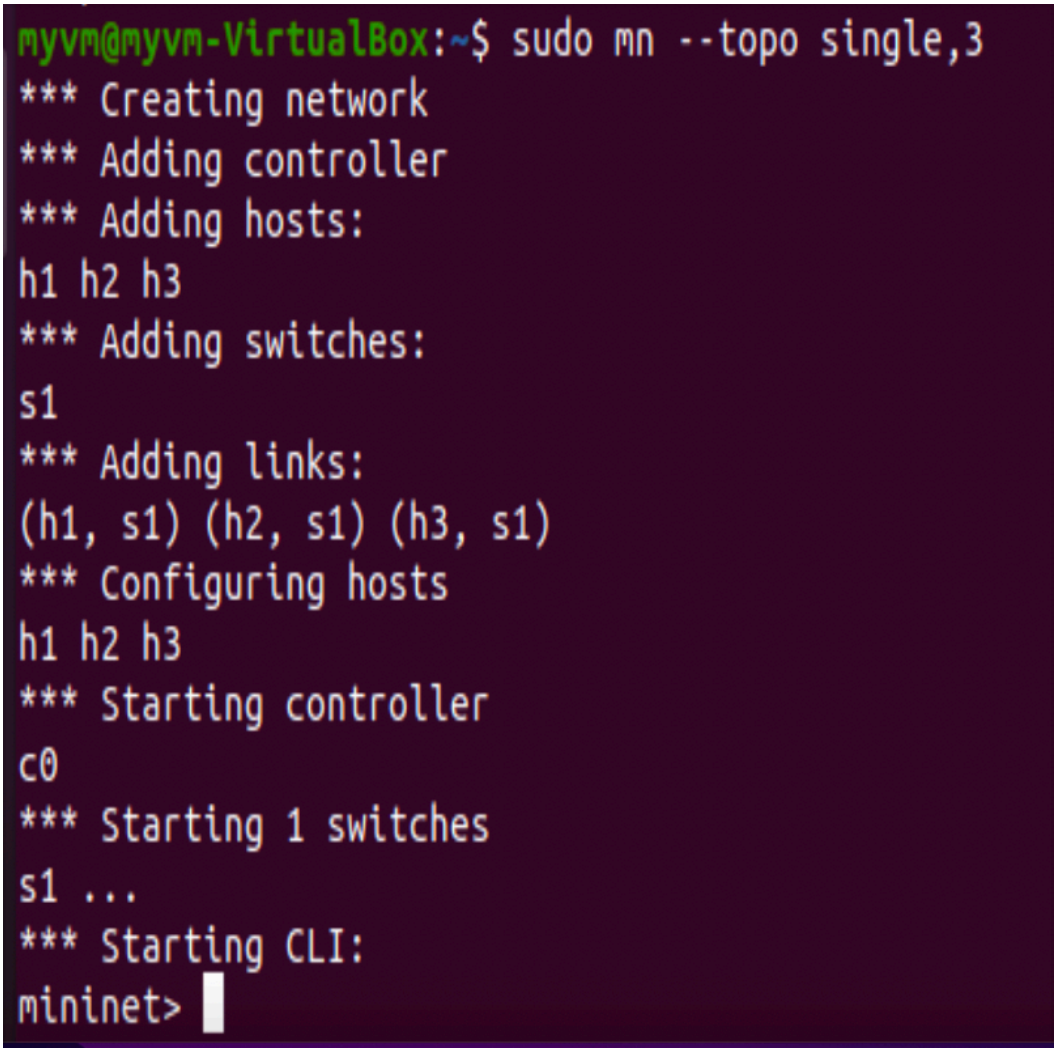
You can also parametrically test & define different network topologies.

For example, if you wanted to test a network with a switch with 3 hosts instead of two:

```
$ sudo mn --test pingall --topo single,3
```

To just initialise it, you would just use --topo

```
$ sudo mn -topo single,3
```



The mininet website says “Parametrized topologies are one of Mininet’s most useful and powerful features.”

Custom topology: You can use the mininet API to define a custom topology in a python script, and then you can load this in with mininet.

```
#topo-2sw-2host.py
#implements a custom topo with 2 switches, 2 hosts.

from mininet.topo import Topo

class MyTopo( Topo ):
    “Simple topology example”

    def build( self ):
        “Create custom network topography”

        #add hosts and switches
        leftHost = self.addHost( ‘h1’ )
        rightHost = self.addHost( ‘h2’ )
```

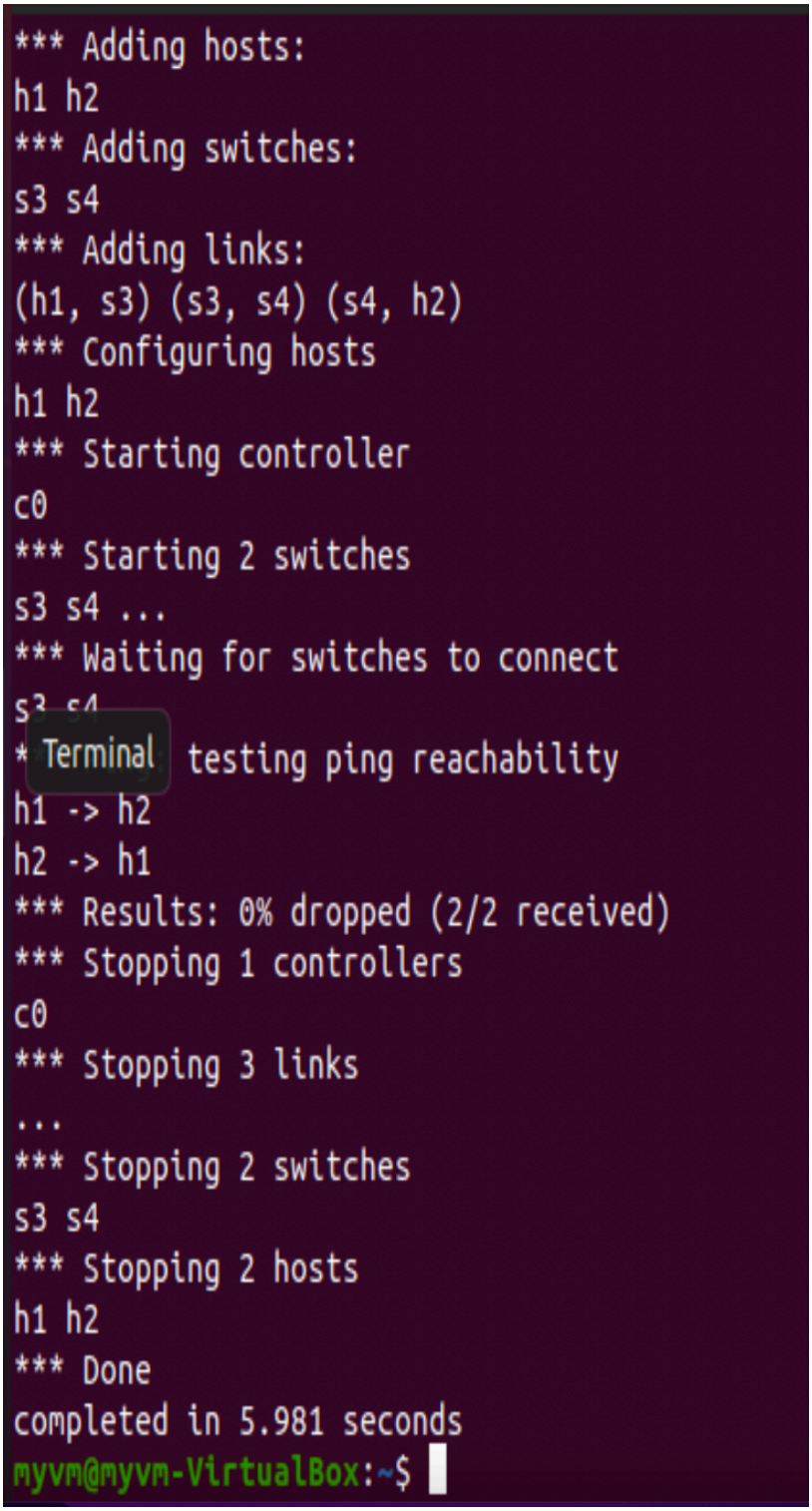
```
        leftSwitch = self.addSwitch( 's3' )
        rightSwitch = self.addSwitch( 's4' )

        #add links between nodes
        self.addLink( leftHost, leftSwitch)
        self.addLink( rightHost, rightSwitch)
        self.addLink( rightSwitch, rightHost)

topos = { 'mytopo' : (lambda: MyTopo() ) }
```

Running the following command to invoke the custom topology:

```
$ sudo mn —custom ./topo-2sw-2host.py —topo mytopo —test
pingall
```



You can also add an option to make the MAC addressing of hosts more logical, and follow a 1,2,3,... pattern instead of being random. (i.e h1's MAC is 00:00:00:00:00:01, h2's MAC is 00:00:00:00:00:02, etc..)

Makes it easier to interpret wireshark traffic, ifconfig output, etc.

Before:

```
$ sudo mn
```

```
...
mininet> h1 ifconfig
h1-eth0  Link encap:Ethernet  HWaddr f6:9d:5a:7f:41:42
          inet addr:10.0.0.1  Bcast:10.255.255.255
Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500
Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0
frame:0
          TX packets:6 errors:0 dropped:0 overruns:0
carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:392 (392.0 B)  TX bytes:392 (392.0 B)
mininet> exit
```

After (notice HWaddr is much simpler.)

```
$ sudo mn --mac
...
mininet> h1 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
          UP BROADCAST RUNNING MULTICAST  MTU:1500
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:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
mininet> exit
```

We can also use a different switch types, here's an example using the Open vSwitch (OVS), which is preinstalled on mininet VM.

```
$ sudo mn --switch ovsk --test iperf
```

Introduction to programming Mininet networks with Python

Following the tutorial specified at
https://github.com/mininet/mininet/wiki/Introduction-to-Mininet

Now we do **everything** programatically in python, no mininet command line tool.

A basic switch connected to N hosts, implemented purely in Python:

```
#!/usr/bin/python

from mininet.topo import Topo
from mininet.net import Mininet
from mininet.util import dumpNodeConnections
from mininet.log import setLogLevel

class SingleSwitchTopo(Topo):
    "Single switch connected to n hosts."
    def build(self, n=2):
        switch = self.addSwitch('s1')
        # Python's range(N) generates 0..N-1
        for h in range(n):
            host = self.addHost('h%s' % (h + 1))
            self.addLink(host, switch)

def simpleTest():
    "Create and test a simple network"
    topo = SingleSwitchTopo(n=4)
    net = Mininet(topo)
    net.start()
    print( "Dumping host connections" )
    dumpNodeConnections(net.hosts)
    print( "Testing network connectivity" )
    net.pingAll()
    net.stop()

if __name__ == '__main__':
    # Tell mininet to print useful information
    setLogLevel('info')
    simpleTest()
```

Explaining this:

“Topo” is the base class for Mininet topologies

“build():” is the method you override when you want to define a custom topology. Constructor parameters will be passed automatically by Topo.__init__().

This method creates a template for the network that consists of:

- A graph of node names
- A database of configuration information

Mininet uses this to create the network.

“addSwitch()” adds a switch to the topology and returns its name.

“addHost()” similarly adds a host.

“addLink()” creates a link between two nodes

“Mininet”: main class to create and manage a network

“start()”: starts your network

“stop()”: stops your network

“pingAll()”: Makes all nodes ping eachother. Useful for testing connectivity

“net.hosts”: all hosts in the network

“dumpNodeConnections()”: dumps connections to / from a set of nodes.

Running Programs in Hosts

One of the most important functionalities in a network simulator is the ability to run programs on hosts in the network.

Each Mininet host is a bash shell process attatched to a network interface, so you can run programs on it by passing it arguments using the cmd() method, python’s method for passing commands to the shell.

For example, to run ifconfig on h1:

```
h1 = net.get('h1')
result = h1.cmd('ifconfig')
print(result)
```

Find out the PID of a background command:

```
pid = int(h1.cmd('echo $!'))
```

You can use Mininet to start a process in the foreground using sendCmd() and wait for it to complete at some later time using waitOutput():

```
for h in hosts:
    h.sendCmd('ping -c 10 h1')

results={}
```

```
for h in hosts:
    results[h.name] = h.waitOutput()
```

Note: Shared Filesystem

One important thing to not is that the default setting is to have Mininet hosts share the same filesystem, the root filesystem of the underlying server.

This is handy because it means that Mininet hosts can share data easily. It is also potentially a problem because if you are running a program that requires configuration files (like an HTTP server), the different Mininet hosts can end up overwriting eachother's configuration files.

To get around this, you can specify private directories for each host in the following way:

h = Host('h1', privateDirs=['/some/directory'])

Host Configuration / Information Methods

- IP() - return IP address of host or interface
- MAC() - return MAC address of host or interface
- setARP() - add a static ARP entry to a host's ARP table
- setIP() - set the IP of a host or interface
- setMAC() - set the MAC address of a host or interface

print("Hostname:", h1.name, "IP ", h1.IP(), "MAC ", h1.MAC())

Invoking the CLI

You can invoke the command-line interface on a network by passing the network object into the CLI() constructor. This is useful for debugging purposes.

```
from mininet.topo import SingleSwitchTopo
from mininet.net import Mininet
from mininet.cli import CLI

net = Mininet(SingleSwitchTopo(2))
net.start()
CLI(net)
net.stop()
```

Customizing mn using --custom files

You can extend the mn command line tool using the —custom option. This

allows you to specify your own custom topologies, switches, hosts, controllers, or link classes. You can also define your own system tests.

To add new features, you need to define a dict in your —custom file based on the command line option you want to use. Here are the options

option	dict name	key: value
`--topo`	`topos`	'short name': `Topo` constructor
`--switch`	`switches`	'short name': `Switch` constructor
`--host`	`hosts`	'short name': `Host` constructor
`--controller`	`controllers`	'short name': `Controller` constructor
`--link`	`links`	'short name': `Link` constructor
`--test`	`tests`	'short name': test function to call with Mininet object

Code example:

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

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

Understanding the Mininet API:

Mininet's API has three levels of abstraction: low level, mid level and high level.

- The low-level API consists of the base node and link classes, so you can build a network from scratch defining all the links and setting IP addresses etc but it is a bit unwieldy.
- The mid-level API adds the Mininet object which serves as a container for nodes and links. The Mininet object has methods such as addHost(), addSwitch(), addLink(), start(), and stop() that make it easy to define a network and get it running.
- The high-level API adds the Topo class which allows you to programatically create topology templates, which can be passed to the "mn" command using —custom

Most of the time it is easiest to use the mid-level API.

Low-level API: nodes and links

```
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 ] )
print( h1.cmd( 'ping -c1', h2.IP() ) )
s1.stop()
c0.stop()
```

Mid-level API: Network object

```
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 )
net.start()
print( h1.cmd( 'ping -c1', h2.IP() ) )
CLI( net )
net.stop()
```

High-level API: Topology templates

```
class SingleSwitchTopo( Topo ):
    "Single Switch Topology"
    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()
```

OpenFlow and Custom Routing

One of Mininet's best features is that is uses Software Defined Networking (SDN). This uses the OpenFlow protocol, and means you can program switches to do anything you want with the packets they recieve. Therefore you can easily integrate custom switch controllers with a mininet network.

It is relatively simple to create a custom subclass of Controller() and pass it to Mininet. Here's a simple example of creating and using a custom POX Controller subclass.

```
#!/usr/bin/python

from mininet.net import Mininet
from mininet.node import Controller
from mininet.topo import SingleSwitchTopo
from mininet.log import setLogLevel

import os

class POXBridge( Controller ):
    "Custom Controller class to invoke POX
forwarding.l2_learning"
    def start( self ):
        "Start POX learning switch"
        self.pox = '%s/pox/pox.py' %
os.environ[ 'HOME' ]
        self.cmd( self.pox, 'forwarding.l2_learning &' )
    def stop( self ):
        "Stop POX"
        self.cmd( 'kill %' + self.pox )

controllers = { 'poxbridge': POXBridge }

if __name__ == '__main__':
    setLogLevel( 'info' )
    net = Mininet( topo=SingleSwitchTopo( 2 ),
controller=POXBridge )
    net.start()
    net.pingAll()
    net.stop()
```

You can then use the above script as a custom command line argument to pass to mn:

```
$ sudo mn --custom poxbridge.py --controller poxbridge
```

```
--topo tree,2,2 --test pingall -v output
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (0/12 lost)
```

Connecting Mininet to an external OpenFlow controller

The RemoteController class acts as a proxy for a controller which may be running anywhere on the control network.

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