Name: Shubhan Singh

Class: SE Comps B

Roll no.: 2022300118

CCN Experiment 10

Mininet

Objective: The objective of this lab exercise is to create a realistic virtual network using Mininet, a tool for emulating network environments. By the end of this exercise, students should be able to set up a virtual network, run real kernel, switch, and application code, and understand the basic workflow of Mininet.

Outcomes:

1. Understand the basics of Software Defined Networking (SDN).
2. Learn how to install and configure Mininet.
3. Create a virtual network with hosts, switches, and controllers.
4. Run real kernel, switch, and application code within the virtual network.

System Requirements:

1. A computer running a Unix-based operating system (e.g., Linux, macOS).
2. Python installed (recommended version 2.7.x or 3.x).
3. Superuser (root) privileges or sudo access.

Procedure:

Step 1: Installation and Setup

1. Visit the Mininet website (mininet.org) and follow the instructions to download and install Mininet on your system. [1][2]
2. Once installed, open a terminal and start Mininet by typing `sudo mn`.

```
mininet@mininet-vm:~$ sudo mn
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

Step 2: Sample Workflow

1. Create a simple network topology using the following command:

   mininet> h1 = net.addHost('h1')

   mininet> h2 = net.addHost('h2')

   mininet> s1 = net.addSwitch('s1')

   mininet> net.addLink(h1, s1)

   mininet> net.addLink(h2, s1)

2. Start the network:

   mininet> net.start()

3. Test connectivity between hosts:

   mininet> h1 ping h2

```
mininet> net.addLink(h1,s1)
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0
c0
mininet> net.addLink(h2,s1)
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0
c0
mininet> net.start()
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0
c0
```

```
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=1.46 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.270 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.038 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.082 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.075 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.072 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.057 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.059 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.127 ms
^C
--- 10.0.0.2 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8167ms
rtt min/avg/max/mdev = 0.038/0.248/1.455/0.431 ms
```

Step 3: Walkthrough

Part 1: Everyday Mininet Usage

1. Display Startup Options

Type the following command to display a help message describing Mininet's startup options:
Sudo mn -h

```
mininet@mininet-vm:~$ sudo mn -h
Usage: mn [options]
(type mn -h for details)

The mn utility creates Mininet network from the command line. It can create
parametrized topologies, invoke the Mininet CLI, and run tests.

Options:
  -h, --help            show this help message and exit
  --switch=SWITCH       default|ivs|lxbr|ovs|ovsbr|ovsk|user[,param=value...]
                        user=UserSwitch ovs=OVSSwitch ovsbr=OVSBridge
                        ovsk=OVSSwitch ivs=IVSSwitch lxbr=LinuxBridge
                        default=OVSSwitch
  --host=HOST           cfs|proc|rt[,param=value...] proc=Host
                        rt=CPULimitedHost{'sched': 'rt'}
                        cfs=CPULimitedHost{'sched': 'cfs'}
  --controller=CONTROLLER
                        default|none|nox|ovsc|ref|remote|ryu[,param=value...]
                        ref=Controller ovsc=OVSController nox=NOX
                        remote=RemoteController ryu=Ryu
                        default=DefaultController none=NullController
  --link=LINK           default|ovs|tc|tcu[,param=value...] default=Link
                        tc=TCLink tcu=TCULink ovs=OVSLink
```

2. Installing wireshark

```
mininet@mininet-vm:~$ sudo apt autoremove
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages will be REMOVED:
  libpcre2-16-0 libxcb-xinerama0 libxcb-xinput0 qttranslations5-l10n
0 upgraded, 0 newly installed, 4 to remove and 358 not upgraded.
After this operation, 13.2 MB disk space will be freed.
Do you want to continue? [Y/n] y
(Reading database ... 114408 files and directories currently installed.)
Removing libpcre2-16-0:amd64 (10.34-7) ...
Removing libxcb-xinerama0:amd64 (1.14-2) ...
Removing libxcb-xinput0:amd64 (1.14-2) ...
Removing qttranslations5-l10n (5.12.8-0ubuntu1) ...
Processing triggers for libc-bin (2.31-0ubuntu9) ...
mininet@mininet-vm:~$ sudo apt install wireshark
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

Interact with Hosts and Switches

Start a minimal topology and enter the CLI:

$ sudo mn

The default topology is the **minimal** topology, which includes one OpenFlow kernel switch connected to two hosts, plus the OpenFlow reference controller.

```
mininet@mininet-vm:~$ sudo mn
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

check the network configuration details (including IP addresses, interfaces, and other network-related information) of host h1 in the Mininet network.

```
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
        ether 8a:16:d6:1c:c2:dd  txqueuelen 1000  (Ethernet)
        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

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        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
```

mininet> s1 ifconfig -a

This will show the switch interfaces, plus the VM's connection out (eth0).

```
mininet> s1 ifconfig -a
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
        ether 08:00:27:84:51:72  txqueuelen 1000  (Ethernet)
        RX packets 11765  bytes 15564327 (15.5 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4054  bytes 336764 (336.7 KB)
        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
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 367  bytes 22854 (22.8 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 367  bytes 22854 (22.8 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Note that *only* the network is virtualized; each host process sees the same set of processes and directories. For example, print the process list from a host process:

mininet> h1 ps -a

This should be the exact same as that seen by the root network namespace:

mininet> s1 ps -a

```
mininet>  s1 ps -a
    PID TTY          TIME CMD
    682 tty1     00:00:00 dbus-run-sessio
    683 tty1     00:00:00 dbus-daemon
    684 tty1     00:00:00 gnome-session-b
    723 tty1     00:00:12 gnome-shell
    762 tty1     00:00:00 at-spi-bus-laun
    767 tty1     00:00:00 dbus-daemon
    770 tty1     00:00:00 Xwayland
    816 tty1     00:00:00 gjs
    817 tty1     00:00:00 at-spi2-registr
    823 tty1     00:00:00 gsd-color
    824 tty1     00:00:00 gsd-print-notif
    826 tty1     00:00:00 gsd-a11y-settin
    827 tty1     00:00:00 gsd-power
    828 tty1     00:00:00 gsd-media-keys
    829 tty1     00:00:00 gsd-rfkill
    844 tty1     00:00:00 gsd-keyboard
    846 tty1     00:00:00 gsd-wacom
    849 tty1     00:00:00 gsd-housekeepin
    853 tty1     00:00:00 gsd-sound
    855 tty1     00:00:00 gsd-datetime
    856 tty1     00:00:00 gsd-smartcard
    857 tty1     00:00:00 gsd-screensaver
    859 tty1     00:00:00 gsd-sharing
    876 tty1     00:00:00 gsd-printer
    945 tty1     00:00:00 ibus-daemon
    952 tty1     00:00:00 ibus-memconf
    954 tty1     00:00:00 ibus-x11
    959 tty1     00:00:00 ibus-portal
    970 tty1     00:00:00 ibus-engine-sim
   2271 pts/0    00:00:00 sudo
   2272 pts/0    00:00:00 mn
   2313 pts/1    00:00:00 controller
   2365 pts/4    00:00:00 ps
```

Test connectivity between hosts

Verifying that you can ping from host 0 to host 1:

mininet> h1 ping -c 1 h2

```
mininet> h1 ping -c 1 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=2.50 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.495/2.495/2.495/0.000 ms
```

An easier way to run this test is to use the Mininet CLI built-in **pingall** command, which does an all-pairs **ping**:

mininet> pingall

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

Run a simple web server and client

Mininet hosts can run any command or application that is available to the underlying Linux system (or VM) and its file system. You can also enter any **bash** command, including job control (**&**, **jobs**, **kill**, etc..) Here we are starting a simple HTTP server on **h1**, making a request from **h2**, then shutting down the web server:

mininet> h1 python -m http.server 80 &

mininet> h2 wget -O - h1

mininet> h1 kill %python

```
mininet>  h1 python -m http.server 80 &
mininet> h2 wget -O - h1
--2024-04-24 10:19:03--  http://10.0.0.1/
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1617 (1.6K) [text/html]
Saving to: 'STDOUT'

-                   0%[                    ]       0  --.-KB/s
   <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso8859-1">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
<li><a href=".bash_history">.bash_history</a></li>
<li><a href=".bash_logout">.bash_logout</a></li>
<li><a href=".bashrc">.bashrc</a></li>
<li><a href=".cache/">.cache/</a></li>
<li><a href=".config/">.config/</a></li>
<li><a href=".gitconfig">.gitconfig</a></li>
```

Part 2: Advanced Startup Options

Run a Regression Test

Mininet can also be used to run self-contained regression tests.

Run a regression test:

$ sudo mn --test pingpair

```
mininet@mininet-vm:~$  sudo mn --test pingpair
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Waiting for switches to connect
s1
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
*** Stopping 1 controllers
c0
*** Stopping 2 links
..
*** Stopping 1 switches
s1
*** Stopping 2 hosts
h1 h2
*** Done
completed in 5.344 seconds
```

This command created a minimal topology, started up the OpenFlow reference controller, ran an all-pairs-**ping** test, and tore down both the topology and the controller.

Changing Topology Size and Type

The default topology is a single switch connected to two hosts. You could change this to a different topo with **--topo**, and pass parameters for that topology's creation. For example, to verify all-pairs ping connectivity with one switch and three hosts:

Run a regression test :

```
mininet@mininet-vm:~$ sudo mn --test pingall --topo single,3
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Waiting for switches to connect
s1
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
*** Stopping 1 controllers
c0
*** Stopping 3 links
...
*** Stopping 1 switches
s1
*** Stopping 3 hosts
h1 h2 h3
*** Done
completed in 5.681 seconds
```

Link variations

Mininet 2.0 allows you to set link parameters, and these can even be set automatially from the command line:

$ sudo mn --link tc,bw=10,delay=10ms mininet> iperf

...

mininet> h1 ping -c10 h2

```
mininet@mininet-vm:~$ sudo mn --link tc,bw=10,delay=10ms
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(10.00Mbit 10ms delay) (10.00Mbit 10ms delay) (h1, s1) (10.00Mbit 10ms delay
) (10.00Mbit 10ms delay) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...(10.00Mbit 10ms delay) (10.00Mbit 10ms delay)
*** Starting CLI:
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['9.46 Mbits/sec', '11.8 Mbits/sec']
mininet> h1 ping -c10 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=42.1 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=41.6 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=42.6 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=41.6 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=42.8 ms
^C
--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 41.594/42.141/42.797/0.499 ms
mininet>
```

Adjustable Verbosity

The default verbosity level is **info**, which prints what Mininet is doing during startup and teardown. Compare this with the full **debug** output with the **-v** param:

$ sudo mn -v debug

...

mininet> exit

```
mininet@mininet-vm:~$ sudo mn -v debug
*** errRun: ['which', 'controller']
/usr/local/bin/controller
  0*** errRun: ['grep', '-c', 'processor', '/proc/cpuinfo']
1
  0*** Setting resource limits
*** Creating network
*** Adding controller
*** errRun: ['which', 'mnexec']
/usr/bin/mnexec
  0*** errRun: ['which', 'ifconfig']
/usr/sbin/ifconfig
  0_popen ['mnexec', '-cd', 'env', 'PS1=\x7f', 'bash', '--norc', '--noeditin
g', '-is', 'mininet:c0'] 3411*** c0 : ('unset HISTFILE; stty -echo; set +m',
)
unset HISTFILE; stty -echo; set +m
*** errRun: ['which', 'telnet']
/usr/bin/telnet
  0*** c0 : ('echo A | telnet -e A 127.0.0.1 6653',)
Telnet escape character is 'A'.
Trying 127.0.0.1...
telnet: Unable to connect to remote host: Connection refused
*** Adding hosts:
*** errRun: ['which', 'mnexec']
/usr/bin/mnexec
```

Custom Topologies

Custom topologies can be easily defined as well, using a simple Python API, and an example is provided in **custom/topo-2sw-2host.py**. This example connects two switches directly, with a single host off each switch:

simple topology example (topo-2sw-2host.py)

```
"""Custom topology example

Two directly connected switches plus a host for each switch:

host --- switch --- switch --- host

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

"""

from mininet.topo import Topo

class MyTopo( Topo ):
    "Simple topology example."

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

        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's3' )
        rightSwitch = self.addSwitch( 's4' )
```

24

25 # Add links

26 self**.**addLink(leftHost, leftSwitch)

27 self**.**addLink(leftSwitch, rightSwitch)

28 self**.**addLink(rightSwitch, rightHost)

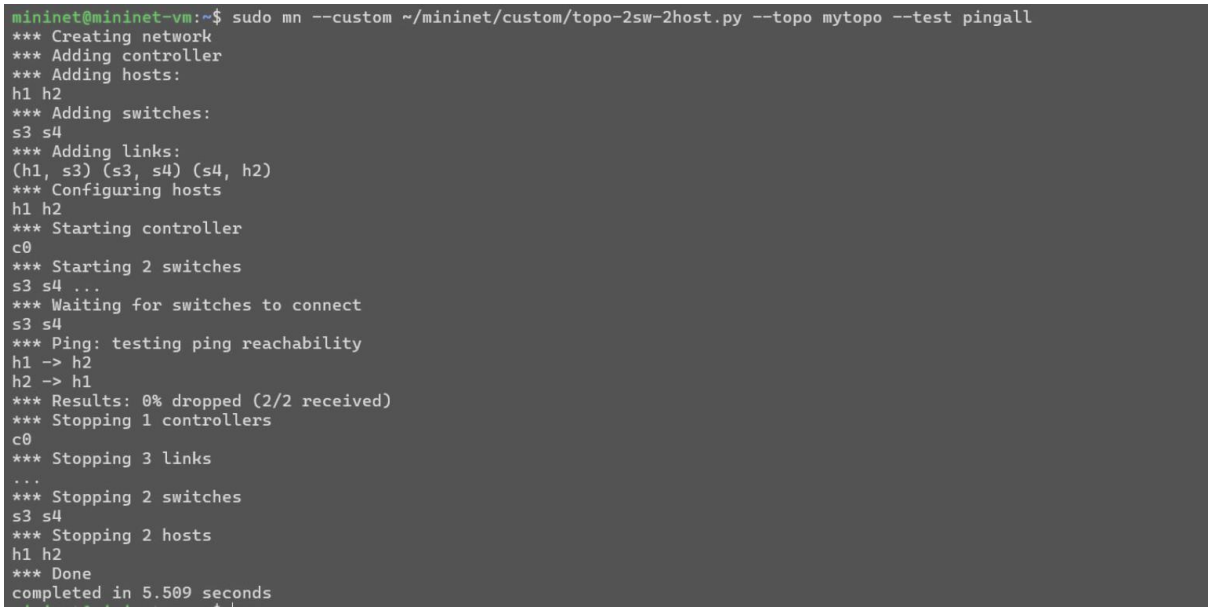
29

30

31topos **=** { 'mytopo': (lambda: MyTopo()) }

When a custom mininet file is provided, it can add new topologies, switch types, and tests to the command-line. For example:

$ sudo mn --custom ~/mininet/custom/topo-2sw-2host.py --topo mytopo --test pingall



ID = MAC

By default, hosts start with randomly assigned MAC addresses. This can make debugging tough, because every time the Mininet is created, the MACs change, so correlating control traffic with specific hosts is tough.

The --mac option is super-useful, and sets the host MAC and IP addrs to small, unique, easy-to-read IDs.

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:

$ 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

XTerm Display

For more complex debugging, you can start Mininet so that it spawns one or more xterms. To start an **xterm** for every host and switch, pass the **-x** option:

$ sudo mn -x

Other Switch Types

Other switch types can be used. For example, to run the user-space switch:

$ sudo mn --switch user --test iperf

Note the much lower TCP iperf-reported bandwidth compared to that seen earlier with the kernel switch.

If you do the ping test shown earlier, you should notice a much higher delay, since now packets must endure additional kernel-to-user-space transitions. The ping time will be more

variable, as the user-space process representing the host may be scheduled in and out by the OS.

On the other hand, the user-space switch can be a great starting point for implementing new functionality, especially where software performance is not critical.

Another example switch type is Open vSwitch (OVS), which comes preinstalled on the Mininet VM. The iperf-reported TCP bandwidth should be similar to the OpenFlow kernel module, and possibly faster:

$ sudo mn --switch ovsk --test iperf

```
mininet@mininet-vm:~$ sudo mn --switch ovsk --test iperf
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Waiting for switches to connect
s1
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['55.5 Gbits/sec', '55.7 Gbits/sec']
*** Stopping 1 controllers
c0
*** Stopping 2 links
..
*** Stopping 1 switches
s1
*** Stopping 2 hosts
h1 h2
*** Done
completed in 10.718 seconds
```

Part 3: Mininet Command-Line Interface (CLI) Commands Display Options

Python Interpreter

If the first phrase on the Mininiet command line is **py**, then that command is executed with Python. This might be useful for extending Mininet, as well as probing its inner workings. Each host, switch, and controller has an associated Node object.

At the Mininet CLI, run:

mininet> py 'hello ' + 'world'

```
mininet>  py 'hello ' + 'world'
hello world
mininet> py locals()
{'net': <mininet.net.Mininet object at 0x7f3f91cdbb50>, 'h1': <Host h1: h1-e
th0:10.0.0.1 pid=4482> , 'h2': <Host h2: h2-eth0:10.0.0.2 pid=4484> , 's1':
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=4489> , 'c0': <Con
troller c0: 127.0.0.1:6653 pid=4475> }
mininet> py dir(s1)
['IP', 'MAC', 'OVSVersion', 'TCReapply', '__class__', '__delattr__', '__dict
__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute
```

You can also evaluate methods of variables:

mininet> py h1.IP()

```
mininet> py h1.IP()
10.0.0.1
mininet> link s1 h1 down
mininet> link s1 h1 up
mininet> xterm h1 h2
```

Part 4: Python API Examples

The examples directory in the Mininet source tree includes examples of how to use Mininet's Python API, as well as potentially useful code that has not been integrated into the main code base.

SSH daemon per host

One example that may be particularly useful runs an SSH daemon on every host:

$ sudo ~/mininet/examples/sshd.py

```
mininet@mininet-vm:~$ sudo ~/mininet/examples/sshd.py
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
*** Adding links:
(s1, h1) (s1, h2) (s1, h3) (s1, h4)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Waiting for switches to connect
s1
*** Waiting for ssh daemons to start
.
*** Hosts are running sshd at the following addresses:
h1 10.0.0.1
h2 10.0.0.2
h3 10.0.0.3
h4 10.0.0.4

*** Type 'exit' or control-D to shut down network
*** Starting CLI:
```

From another terminal, you can ssh into any host and run interactive commands:

$ ssh 10.0.0.1

$ ping 10.0.0.2

…

$ exit

```
mininet@mininet-vm:~$ ssh 10.0.0.1
The authenticity of host '10.0.0.1 (10.0.0.1)' can't be established.
ECDSA key fingerprint is SHA256:Q2kQlJrVHnKEv+W5x4m1Ba8gVOdIcqZcbe3AABzIYpw.
Are you sure you want to continue connecting (yes/no/[fingerprint])? y
Please type 'yes', 'no' or the fingerprint: yes
Warning: Permanently added '10.0.0.1' (ECDSA) to the list of known hosts.
mininet@10.0.0.1's password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-42-generic x86_64)

 * Documentation:  https://help.ubuntu.com
```

```
mininet@mininet-vm:~$ ping 10.0.0.2
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=1.08 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.173 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.082 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.078 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.046 ms
^C
--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4051ms
rtt min/avg/max/mdev = 0.046/0.292/1.084/0.397 ms
```

Mininet Walkthrough complete.

Conclusion:

In this experiment, we learnt how to set up a virtual network using mininet to implement Software Defined Networking (SDN), through the walkthrough tutorial on mininet's website.

We installed mininet as a VM on our machine, which was preconfigured to contain all packages and permission required to use it. Using mininet,we made virtual networks with different topologies, ran python code inside of mininet and used the mininet python API, and learnt about many mininet features and commands.