

Mininet Lab 1: Introduction to Mininet

tags: RSE Labs

This lab demonstrates most Mininet commands, as well as its typical usage in concert with the Wireshark dissector.

First, a (perhaps obvious) note on command syntax for this walkthrough:

- `$` precedes Linux commands that should be typed at the shell prompt
- `mininet>` precedes Mininet commands that should be typed at Mininet's CLI, (is OK if you see `mininet-wifi`)
- `#` precedes Linux commands that are typed at a root shell prompt

Part 1: Basic Mininet Usage

Display Startup Options

Let's get started with Mininet's startup options.

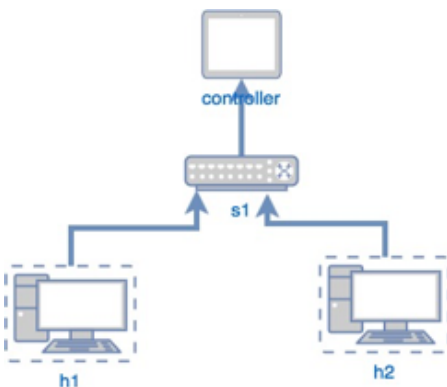
Type the following command to display a help message describing Mininet's startup options:

```
$ sudo mn -h
```

Interact with Hosts and Switches

Now, 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. This topology could also be specified on the command line with `--topo=minimal`. Other topologies are also available out of the box; see the `--topo` section in the output of `mn -h`.

All four entities (2 host processes, 1 switch process, 1 basic controller) are now running in the VM. If no specific test is passed as a parameter, the Mininet CLI comes up.

Some basic commands:

- Display Mininet CLI commands:

```
mininet> help
```

- Display nodes:

```
mininet> nodes
```

- Display links:

```
mininet> net
```

- Dump information about all nodes:

```
mininet> dump
```

You should see the switch and two hosts listed.

More on commands:

If the **first string** typed into the Mininet CLI is a host, switch or controller name, the command is executed on that node. Run a command on a host process:

```
mininet> h1 ifconfig -a
```

You should see the host's `h1-eth0` and loopback (`lo`) interfaces. Note that this interface (`h1-eth0`) is not seen by the primary Linux system when `ifconfig` is run, because it is specific to the network namespace of the host process.

In contrast, the switch by default runs in the root network namespace, so running a command on the "switch" is the same as running it from a regular terminal:

```
mininet> s1 ifconfig -a
```

This will show the switch interfaces, plus the VM's connection out (`ens33` previously called `eth0`).

1.- Prueba a ejecutar el comand `route` en los diferentes dispositivos

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
```

It would be possible to use separate process spaces with Linux containers, but currently Mininet doesn't do that. Having everything run in the "root" process namespace is convenient for debugging, because it allows you to see all of the processes from the console using `ps`, `kill`, etc.

Test connectivity between hosts

To view control traffic using the OpenFlow Wireshark dissector, first open Wireshark in the background... or in a different `xterm`:

```
$ sudo wireshark
```

In the Wireshark filter box enter the value `openflow_v1`, then click `Apply`.

In Wireshark, click `Capture`, then `Interfaces`, then select `Start` on the loopback interface (`lo`).

Wireshark will allow you to view all the traffic, in this case OpenFlow related, that is interchanged during the examples.

Now, verify that you can ping from host 0 to host 1:

```
mininet> h1 ping -c 1 h2
```

If a string appears later in the command with a node name, that node name is replaced by its IP address; this happened for `h2`.

You should see OpenFlow control traffic. The first host ARPs for the MAC address of the second, which causes a `packet_in` message to go to the controller. The controller then sends a `packet_out` message to flood the broadcast packet to other ports on the switch (in this example, the only other data port). The second host sees the ARP request and sends a reply. This reply goes to the controller, which sends it to the first host and pushes down a flow entry.

Now the first host knows the MAC address of the second, and can send its ping via an ICMP Echo Request. This request, along with its corresponding reply from the second host, both go the controller and result in a flow entry pushed down (along with the actual packets getting sent out).

Repeat the last `ping`:

```
mininet> h1 ping -c 1 h2
```

You should see a much lower `ping` time for the second try ($< 100\mu s$). A flow entry covering ICMP `ping` traffic was previously installed in the switch, so no control traffic was generated, and the packets immediately pass through the switch.

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
```

Exiting

To leave Mininet, from the CLI:

```
mininet> exit
```

Cleanup

If Mininet crashes for some reason, clean it up:

```
$ sudo mn -c
```

Part 2: Advanced Startup Options

XTerm Display

To start an `xterm` for every host and switch, pass the `-x` option:

```
$ sudo mn -x
```

After a second, the `xterms` will pop up, with automatically set window names.

Alternately, you can bring up additional `xterms` as shown below.

```
mininet> xterm h1 h2
```

You can tell whether an `xterm` is in the root namespace by checking `ifconfig`; if all interfaces are shown (including `ens33` previously called `eth0`), it's in the root namespace. Additionally, its title should contain `“(root)”`.

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:

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

Another example, with a linear topology:

```
$ sudo mn --test pingall --topo linear,4
```

2.- Dibuja la estructura de red que obtienes con este ultimo comando

Link variations

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

```
$ sudo mn --link tc,bw=10,delay=10ms
```

3.- Utilizando `ping` calcula la diferencia con el caso sin definir los "link parameters"

4.- Ahora repite lo mismo pero utilizando el comando `iperf`. ¿Como?

Link Up/Down

For fault tolerance testing, it can be helpful to bring links up and down.

To disable both halves of a virtual ethernet pair:

```
mininet> link s1 h1 down
```

You should see an OpenFlow Port Status Change notification get generated. To bring the link back up:

```
mininet> link s1 h1 up
```

Namespaces and xterms

By default, only the hosts are put in a separate namespace; the window for each switch is unnecessary (that is, equivalent to a regular terminal), but can be a convenient place to run and leave up switch debug commands, such as flow counter dumps.

Xterms are also useful for running interactive commands that you may need to cancel, for which you'd like to see the output.

For example, start mininet with:

```
$ sudo mn -x
```

In the xterm labeled "switch: s1 (root)", run:

```
# dpctl dump-flows tcp:127.0.0.1:6634
```

`dpctl` is a management utility that enable some control over the OpenFlow switch

Nothing will print out; the switch has no flows added. To use `dpctl` with other switches, start up mininet in verbose mode and look at the passive listening ports for the switches when they're created.

Now, in the xterm labeled "host: h1", run:

```
# ping 10.0.0.2
```

Go back to `s1` and dump the flows: `# dpctl dump-flows tcp:127.0.0.1:6634`

You should see multiple flow entries now.

Run a simple web server and client

Remember that `ping` isn't the only command you can run on a host! 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..)

5.- Sabiendo que:

```
$ python -m SimpleHTTPServer 80
```

ejecuta un simple HTTP server, y que:

```
$ wget -O - _HTTP_server_address_
```

ejecuta una get como cliente al server *HTTP_server_address*

indica la secuencias de ordenes para ejecutar el server HTTP en h1 y el cliente get en h2 en la topologia minima de Mininet

Part 3: Python API Examples

Python Interpreter

If the first phrase on the Mininet 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'
```

Print the accessible local variables:

```
mininet> py locals()
```

Next, see the methods and properties available for a node, using the `dir()` function:

```
mininet> py dir(s1)
```

You can read the on-line documentation for methods available on a node by using the `help()` function:

```
mininet> py help(h1)
```

(Press "q" to quit reading the documentation.)

You can also evaluate methods of variables:

```
mininet> py h1.IP()
```

Python can be used to create network to be evaluated. With a few lines of Python code, you can create a flexible topology which can be configured based on the parameters you pass into it, and reused for multiple experiments.

Naming in Mininet

In order to use Mininet effectively, it is important to understand its naming scheme for hosts, switches and interfaces. Usually, hosts are called `h1..hN` and switches are called `s1..sN`. We recommend that you follow this convention or a similar one. For clarity, interfaces belonging to a node are named beginning with the node's name, for example `h1-eth0` is host `h1`'s default interface, and `s1-eth1` is switch `s1`'s first data port. Host interfaces are only visible from within the host itself, but switch data ports are visible in the "root" namespace (you can see them by typing `ip link show` in another window while Mininet is running.) As a result, it's easy to examine switch interfaces but slightly trickier to examine host interfaces, since you must tell the host to do so (typically using `host.cmd()`.)

For example, here is a simple network topology (File: emptynet.py (<http://emptynet.py>)) which consists of 4 hosts connected to a single switch (`s1`):


```
1  #!/usr/bin/python
2  # File: emptynet.py
3  from mininet.net import Mininet
4  from mininet.node import Controller
5  from mininet.cli import CLI
6  from mininet.log import setLogLevel, info
7
8  def emptyNet():
9
10     "Create an empty network and add nodes to it."
11
12     net = Mininet( controller=Controller )
13
14     info( '*** Adding controller\n' )
15     net.addController( 'c0' )
16
17     info( '*** Adding hosts\n' )
18     h1 = net.addHost( 'h1', ip='10.0.0.1' )
19     h2 = net.addHost( 'h2', ip='10.0.0.2' )
20     h3 = net.addHost( 'h3', ip='10.0.0.3' )
21     h4 = net.addHost( 'h4', ip='10.0.0.4' )
22
23     info( '*** Adding switch\n' )
24     s1 = net.addSwitch( 's1' )
25
26     info( '*** Creating links\n' )
27     net.addLink( h1, s1 )
28     net.addLink( h2, s1 )
29     net.addLink( h3, s1 )
30     net.addLink( h4, s1 )
31
32     info( '*** Starting network\n' )
33     net.start()
34
35     info( '*** Running CLI\n' )
36     CLI( net )
37
38     info( '*** Stopping network' )
39     net.stop()
40
41 if __name__ == '__main__':
42     setLogLevel( 'info' )
43     emptyNet()
44
```

Important classes, methods, functions and variables in the above code include:

- `addSwitch()` : adds a switch to a topology and returns the switch name
- `addHost()` : adds a host to a topology and returns the host name
- `addLink()` : adds a bidirectional link to a topology (and returns a link key, but this is not important). Links in Mininet are bidirectional unless noted otherwise.

- Mininet : main class to create and manage a network
- `start()` : starts your network
- `pingAll()` : tests connectivity by trying to have all nodes ping each other
- `stop()` : stops your network
- `net.hosts` : all the hosts in a network
- `dumpNodeConnections()` : dumps connections to/from a set of nodes.
- `setLogLevel('info' | 'debug' | 'output')` : set Mininet's default output level; 'info' is recommended as it provides useful information.

Running this script:

```
$ cd ~/RSEmininet/mininet
$ sudo python emptynet.py
```

should result in something like the following:

```
*** Adding controller
*** Adding hosts
*** Adding switch
*** Creating links
*** Starting network
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Running CLI
*** Starting CLI:
mininet>
```

use `ping` or `pingall` to evaluate connectivity.

Host Configuration Methods

Mininet hosts provide a number of convenience methods for network configuration:

- `IP()`: Return IP address of a host or specific interface.
- `MAC()`: Return MAC address of a host or specific interface.

- `setARP()`: Add a static ARP entry to a host's ARP cache.
- `setIP()`: Set the IP address for a host or specific interface.
- `setMAC()`: Set the MAC address for a host or specific interface

For example:

```
print "Host", h1.name, "has IP address", h1.IP(), "and MAC address", h1.MAC()
```

6.- Modifica el código de la red de arriba para que imprima la dirección IP y el MAC de cada host

Setting Performance Parameters

In addition to basic behavioral networking, Mininet provides performance limiting and isolation features, through the `CPUimitedHost` and `TCLink` classes.

```
1 from mininet.node import CPUimitedHost
2 from mininet.link import TCLink
3 ...
4 net = Mininet( controller=Controller, host = CPUimitedHost, link = TCLink )
5
```

- `self.addHost(name, cpu=f)`: This allows you to specify a fraction of overall system CPU resources which will be allocated to the virtual host.
- `self.addLink(node1, node2, bw=10, delay='5ms', max_queue_size=1000, loss=10, use_htb=True)`: adds a bidirectional link with bandwidth, delay and loss characteristics, with a maximum queue size of 1000 packets using the Hierarchical Token Bucket rate limiter and netem delay/loss emulator. The parameter `bw` is expressed as a number in Mbit; `delay` is expressed as a string with units in place (e.g. '5ms', '100us', '1s'); `loss` is expressed as a percentage (between 0 and 100); and `max_queue_size` is expressed in packets.

7.- Modifica el código de la red de arriba, para que los enlaces sean de 10 Mbps y con un retardo de 10ms. Calcula las prestaciones del enlace entre h1 y h3

Appendix (just for reference)

Important: Shared Filesystem!

One thing to keep in mind is that by default Mininet hosts share the root filesystem of the underlying server. Usually this is a very good thing, since it is a huge pain (and slow) to create a separate filesystem for each Mininet host (which you can do if you like and then chroot into

it!)

Sharing the root filesystem also means that you almost never need to copy data between Mininet hosts because it's already there.

One side-effect of this however is that hosts share the Mininet server's /etc directory. This means that if you require specific configuration for a program (e.g. httpd) then you may need to create different configuration files for each Mininet host and specify them as startup options to the program that you are running.

Another side-effect is that you can have file collisions if you try to create the same file in the same directory on multiple hosts.

If you need per-host private directories, you can specify them as options to Host, for example:

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

Mininet API Documentation

Mininet includes Python documentation strings for each module and API call. These may be accessed from Python's regular help() mechanism. For example,

```
python
>>> from mininet.node import Host
>>> help(Host.IP)
Help on method IP in module mininet.node:

IP(self, intf=None) unbound mininet.node.Host method
    Return IP address of a node or specific interface.
```

This same documentation is also available on the Mininet web site at <http://api.mininet.org> (<http://api.mininet.org>).


Understanding the Mininet API

Over the course of this lab, you have been exposed to a number of Python classes which comprise Mininet's API, including classes such as Topo, Mininet, Host, Switch, Link and their subclasses. It is convenient to divide these classes into levels (or layers), since in general the high-level APIs are built using the lower-level APIs.

Mininet's API is built at three primary levels:

- **Low-level API:** The low-level API consists of the base node and link classes (such as Host, Switch, and Link and their subclasses) which can actually be instantiated individually and used to create a network, but it is a bit unwieldy.
- **Mid-level API:** The mid-level API adds the Mininet object which serves as a container for nodes and links. It provides a number of methods (such as addHost(), addSwitch(), and addLink()) for adding nodes and links to a network, as well as network configuration, startup and shutdown (notably start() and stop().)

Mininet Lab 1: Introduction to ...

Try  HackMD (https://hackmd.io?utm_source=view-page&utm_medium=logo-nav)

- **High-level API:** The high-level API adds a topology template abstraction, *the Topo class*, which provides the ability to create reusable, parametrized topology templates. These templates can be passed to the mn command (via the `-custom` option) and used from the command line.

It is valuable to understand each of the API levels. In general when you want to control nodes and switches directly, you use the low-level API. When you want to start or stop a network, you usually use the mid-level API (notably the Mininet class.)

Things become interesting when you start thinking about creating full networks. Full networks can be created using any of the API levels (as seen in the examples), but usually you will want to pick either the mid-level API (e.g. `Mininet.add*()`) or the high-level API (`Topo.add*()`) to create your networks.

Here are examples of creating networks using each API level:

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()
```

As you can see, the mid-level API is actually the simplest and most concise for this example, because it doesn't require creation of a topology class. The low-level and mid-level APIs are flexible and powerful, but may be less convenient to reuse compared to the high-level Topo API and its topology templates.