IPMininet Documentation

Release v0.6

Olivier Tilmans

Nov 25, 2019

Contents

This is a python library, extending Mininet, in order to support emulation of (complex) IP networks. As such it provides new classes, such as Routers, auto-configures all properties not set by the user, such as IP addresses or router configuration files, ...

CHAPTER 1

Installation

IPMininet needs at minimum:

- Python (with pip) **2.7+** or **3.5+**
- Mininet

IPMininet needs some daemon executables to be installed and accessible through the PATH environment variable:

- FRRouting daemons: zebra, ospfd, ospf6d, bgpd, pimd
- RADVD
- SSHD

You can either download them by hand or rely on one the following methods:

1.1 Virtual Machine

We maintain a vagrant box packaged with all the daemons. To use it, first install Vagrant and Virtualbox and then, execute the following commands:

```
$ vagrant init ipmininet/ubuntu-18.04
$ vagrant up
```

This will create the VM. To access the VM with SSH, just issue the following command in the same directory as the two previous one:

```
$ vagrant ssh
```

1.2 Manual installation

You can download and install IPMininet. You can change the installed version by replacing "v0.8" in the following commands. If you have pip above **18.1**, execute:

```
$ sudo pip install --upgrade git+https://github.com/cnp3/ipmininet.git@v0.8
```

If you have an older version of pip, use:

```
$ sudo pip install --process-dependency-links --upgrade git+https://github.com/cnp3/
↪ipmininet.git@v0.8
```

Then, you can install all the daemons:

```
$ sudo python -m ipmininet.install -af
```

You can choose to install only a subset of the daemons by changing the options on the installation script. For the option documentations, use the `-h` option.

CHAPTER 2

Getting started

To start your network, you need to do two things:

1. Creating a topology
2. Running the network

2.1 Topology creation

To create a new topology, we need to declare a class that extends `IPTopo`.

```
from ipmininet.iptopo import IPTopo

class MyTopology(IPTopo):
    pass
```

Then we extend in its build method to add switches, hosts, routers and links between the nodes.

```
from ipmininet.iptopo import IPTopo

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        r1 = self.addRouter("r1")
        r2 = self.addRouter("r2")

        s1 = self.addSwitch("s1")
        s2 = self.addSwitch("s2")

        h1 = self.addHost("h1")
        h2 = self.addHost("h2")

        self.addLink(r1, r2)
```

(continues on next page)

(continued from previous page)

```
        self.addLink(s1, r1)
        self.addLink(h1, s1)
        self.addLink(s2, r2)
        self.addLink(h2, s2)

        super(MyTopology, self).build(*args, **kwargs)
```

We can add daemons to the routers and hosts as well.

```
from ipmininet.iptopo import IPTopo
from ipmininet.router.config import SSHd
from ipmininet.host.config import Named

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        r1 = self.addRouter("r1")
        r1.addDaemon(SSHd)

        h1 = self.addHost("h1")
        h1.addDaemon(Named)

        # [...]

        super(MyTopology, self).build(*args, **kwargs)
```

By default, OSPF and OSPF6 are launched on each router. This means that your network has basic routing working by default. To change that, we have to modify the router configuration class.

```
from ipmininet.iptopo import IPTopo
from ipmininet.router.config import SSHd, RouterConfig

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        r1 = self.addRouter("r1", config=RouterConfig)
        r1.addDaemon(SSHd)

        # [...]

        super(MyTopology, self).build(*args, **kwargs)
```

We can customize the daemons configuration by passing options to them. In the following code snippet, we change the hello interval of the OSPF daemon. You can find the configuration options in *Configuring daemons*

```
from ipmininet.iptopo import IPTopo
from ipmininet.router.config import OSPF, RouterConfig

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        r1 = self.addRouter("r1", config=RouterConfig)
        r1.addDaemon(OSPF, hello_int=1)
```

(continues on next page)

(continued from previous page)

```
        # [...]

        super(MyTopology, self).build(*args, **kwargs)
```

2.2 Network run

We run the topology by using the following code. The IPCLI object creates a extended Mininet CLI. More details can be found in *Command-Line interface* As for Mininet, IPMininet networks need root access to be executed.

```
from ipmininet.ipnet import IPNet
from ipmininet.cli import IPCLI

net = IPNet(topo=MyTopology())
try:
    net.start()
    IPCLI(net)
finally:
    net.stop()
```

2.3 IPMininet network cleaning

If you forget to clean your network with `net.stop()` in your script, your machine can will have ghost daemon process and uncleaned network namespaces. This can also happen if IPMininet crashes. In both cases, you have to clean it up with the following command:

```
sudo python -m ipmininet.clean
```

2.4 Mininet compatibility

IPMininet is an upper layer above Mininet. Therefore, everything that works in Mininet, also works in IPMininet. Feel free to consult the Mininet documentation as well.

CHAPTER 3

Example topologies

This directory contains example topologies, you can start them using

```
python -m ipmininet.examples --topo=[topo_name] [--args key=val,key=val]
```

Where topo_name is the name of the topology, and args are optional arguments for it.

The following sections will detail the topologies.

- *SimpleOSPFNetwork*
- *SimpleBGPNetwork*
- *BGPDecisionProcess*
- *BGPLocalPref*
- *BGPMED*
- *BGPRR*
- *BGPFullConfig*
- *BGPPolicies*
- *BGPPoliciesAdjust*
- *IPTables*
- *GRETopo*
- *SSHd*
- *RouterAdvNetwork*
- *SimpleOpenRNetwork*
- *StaticAddressNetwork*
- *PartialStaticAddressNet*
- *StaticRoutingNet*

- *StaticRoutingNetBasic*
- *StaticRoutingNetComplex*
- *StaticRoutingNetFailure*
- *SpanningTreeNet*
- *SpanningTreeHub*
- *SpanningTreeBus*
- *SpanningTreeIntermediate*
- *SpanningTreeFullMesh*
- *SpanningTreeAdjust*
- *SpanningTreeCost*
- *DNSNetwork*

3.1 SimpleOSPFNetwork

topo name : simple_ospf_network *args* : n/a

This network spawn a single AS topology, using OSPF, with multiple areas and variable link metrics. From the mininet CLI, access the routers vtysh using

```
[noecho rx] telnet localhost [ospfd/zebra]
```

Where the noecho rx is required if you don't use a separate xterm window for the node (via `xterm rx`), and ospfd/zebra is the name of the daemon you wish to connect to.

3.2 SimpleOSPFv3Network

topo name : simple_ospfv3_network *args* : n/a

This network spawn a single AS topology, using OSPFv3, with variable link metrics. From the mininet CLI, access the routers vtysh using

```
[noecho rx] telnet localhost [ospf6d/zebra]
```

Where the noecho rx is required if you don't use a separate xterm window for the node (via `xterm rx`), and ospf6d/zebra is the name of the daemon you wish to connect to.

3.3 SimpleBGPNetwork

topo name : simple_bgp_network *args* : n/a

This networks spawn ASes, exchanging reachability information.

- AS1 has one eBGP peering with AS2
- AS2 has 2 routers, using iBGP between them, and has two eBGP peering, one with AS1 and one with AS3
- AS3 has one eBGP peerin with AS2

3.4 BGPDecisionProcess

topo name : bgp_decision_process *args* : other_cost (defaults to 5)

This network is similar to SimpleBGPNetwork. However, AS2 has more routers, and not all of them run BGP. It attempts to show cases the effect of the IGP cost in the BGP decision process in Quagga.

Both AS1 and AS3 advertize a router towards 1.2.3.0/24 to AS2 eBGP routers as2r1 and as2r2. These routers participate in an OSPF topology inside their AS, which looks as follow: as2r1 -[10]- x -[1]- as2r3 -[1]- y -[other_cost]- as2r2. as2r1, as2r3 and as2r2 also participate in an iBGP fullmesh.

Depending on the value of [other_cost] (if it is greater or lower than 10), as2r3 will either choose to use as2r1 or as2r2 as nexthop for 1.2.3.0/24, as both routes are equal up to step #8 in the decision process, which is the IGP cost (in a loosely defined way, as it includes any route towards the BGP nexthop). If other_cost is 10, we then arrive at step #10 to choose the best routes, and compare the routerids of as2r1 and as2r2 to select the path (1.1.1.1 (as2r1) vs 1.1.1.2 (as2r2), so we select the route from as2r1).

You can observe this selection by issuing one of the following command sequence once BGP has converged:

- net > as2r3 ip route show 1.2.3.0/24
- [noecho as2r3] telnet localhost bgpd > password is zebra > enable > show ip bgp 1.2.3.0/24

3.5 BGPLocalPref

topo name : bgp_local_pref *args* : n/a

This topology is composed of two ASes connected in dual homing with a higher local pref on one of the BGP peerings. Thus, all the traffic coming from AS1 will go through the upper link.

3.6 BGPMED

topo name : bgp_med *args* : n/a

This topology is composed of two ASes connected in dual homing with a higher MED for routes from the upper peering than the lower one. Thus, all the traffic coming from AS1 will go through the lower link.

3.7 BGPRR

topo name : bgp_rr *args* : n/a

This topology is composed of five AS. AS1 uses two router reflectors.

3.8 BGPFullConfig

topo name : bgp_full_config *args* : n/a

This topology is composed of two AS connected in dual homing with different local pref, MED and communities. AS1 has one route reflector as well.

3.9 BGPPolicies

The following topologies are built from the exercise sessions of CNP3 syllabus.

topo name : bgp_policies_1 *args* : n/a

topo name : bgp_policies_2 *args* : n/a

topo name : bgp_policies_3 *args* : n/a

topo name : bgp_policies_4 *args* : n/a

topo name : bgp_policies_5 *args* : n/a

All of these topologies have routes exchanging BGP reachability. They use two predefined BGP policies: shared-cost and client/provider peerings.

ASes always favor routes received from clients, then routes from shared-cost peering, and finally, routes received from providers. Moreover, ASes filter out routes depending on the peering type:

- Routes learned from shared-cost are not forwarded to providers and other shared-cost peers.
- Routes learned from providers are not forwarded to shared-cost peers and other providers.

3.10 BGPPoliciesAdjust

The following topology is built from the exercise sessions of CNP3 syllabus.

topo name : bgp_policies_adjust *args* : as_start (defaults to None), as_end (defaults to None), bgp_policy (defaults to 'Share')

This network contains a topology with 5 shared-cost and 2 client-provider peerings. Some ASes cannot reach all other ASes. The user can add a peering to ensure connectivity. To do so, use the topology arguments. For instance, the following command will add a link between AS1 and AS3 and start a shared-cost BGP peering.

```
python -m ipmininet.examples --topo=bgp_policies_adjust --args as_start=as1r,as_
↪end=as3r,bgp_policy=Share
```

3.11 IPTables

topo name : iptables *args* : n/a

This network spawns two routers, which have custom ACLs set such that their inbound traffic (the INPUT chains in ip(6)tables):

- Can only be ICMP traffic over IPv4
- Can only be (properly established) TCP over IPv6

You can test this by trying to ping(6) both routers, use nc to (try to) exchange data over TCP, or tracebox to send a crafted TCP packet not part of an already established session.

3.12 GRETopo

topo name : gre *args* : n/a

This network spawns routers in a line, with two hosts attached on the ends. A GRE Tunnel for prefix 10.0.1.0/24 is established with the two hosts (h1 having 10.0.1.1 assigned and h2 10.0.1.2).

Example tests:

- Verify connectivity, normally: h1 ping h2, over the tunnel: h1 ping 10.0.1.2
- h1 traceroute h2, h1 traceroute 10.0.1.2, should show two different routes, with the second one hiding the intermediate routers.

3.13 SSHd

topo name : ssh *args* : n/a

This network spawns two routers with an ssh daemon, an a key that is renewed at each run.

You can try to connect by reusing the per-router ssh config, e.g.:

```
r1 ssh -o IdentityFile=/tmp/__ipmininet_temp_key r2
```

3.14 RIPngNetwork

topo name : ripng_network *args* : n/a

This network uses the RIPng daemon to ensure connectivity between hosts. Like all FRRouting daemons, you can access the routers vtysh using, from the mininet CLI:

```
[noecho rx] telnet localhost 2603
```

3.15 RIPngNetworkAdjust

topo name : ripng_network_adjust *args* : lr1r2_cost, lr1r3_cost, lr1r5_cost, lr2r3_cost, lr2r4_cost, lr2r5_cost, lr4r5_cost

This network also uses the RIPng daemon to ensure connectivity between hosts. Moreover, the IGP metric on each link can be customized. For instance, the following command changes IGP cost of both the link between r1 and r2 and the link between r1 and r3 to 2:

```
python -m ipmininet.examples --topo=ripng_network_adjust --args lr1r2_cost=2,lr1r3_
↪cost=2
```

3.16 RouterAdvNetwork

topo name : router_adv_network *args* : n/a

This network spawn a small topology with two hosts and a router. One of these hosts uses Router Advertisements to get its IPv6 addresses The other one's IP addresses are announced in the Router Advertisements as the DNS server's addresses.

3.17 SimpleOpenRNetwork

topo name : simple_openr_network *args* : n/a

This network represents a small OpenR network connecting three routers in a Bus topology. Each router has hosts attached. The `/tmp` folders are private to isolate the unix sockets used by OpenR. The private `/var/log` directories isolate logs.

Use breeze to investigate the routing state of OpenR.

3.18 StaticAddressNetwork

topo name : static_address_network *args* : n/a

This network has statically assigned addresses instead of using the IPMininet auto-allocator.

3.19 PartialStaticAddressNetwork

topo name : partial_static_address_network *args* : n/a

This network has some statically assigned addresses and the others are dynamically allocated.

3.20 StaticRoutingNet

topo name : static_routing_network *args* : n/a

This network uses static routes with zebra and static daemons.

3.21 StaticRoutingNetBasic

topo name : static_routing_network_basic *args* : n/a

This nework uses static routes with zebra and static daemons. This topology uses only 4 routers.

3.22 StaticRoutingNetComplex

topo name : static_routing_network_complex *args* : n/a

This network uses static routes with zebra and static daemons. This topology uses 6 routers. The routes are not the same as if they were chosen by OSPF6. The path from X to Y and its reverse path are not the same.

3.23 StaticRoutingNetFailure

topo name : static_routing_network_failure *args* : n/a

This network uses static routes with zebra and static daemons. These static routes are incorrect. They do not enable some routers to communicate with each other.

3.24 SpanningTreeNet

topo name : spanning_tree_network *args* : n/a

This network contains a single LAN with a loop. It enables the spanning tree protocol to prevent packet looping in the LAN.

3.25 SpanningTreeHub

topo name : spanning_tree_hub *args* : n/a

This network contains a more complex LAN with many loops, using hubs to simulate one-to-many links between switches. It enables the spanning tree protocol to prevent packet looping in the LAN.

3.26 SpanningTreeBus

topo name : spanning_tree_bus *args* : n/a

This network contains a single LAN without any loop, but using a hub to simulate a bus behavior. It enables the spanning tree protocol to prevent packet looping in the LAN, even if there is no loop here.

3.27 SpanningTreeIntermediate

topo name : spanning_tree_intermediate *args* : n/a

This network contains a single LAN with 2 loops inside. It shows the spanning tree protocol to avoid the packets looping in the network.

3.28 SpanningTreeFullMesh

topo name : spanning_tree_full_mesh *args* : n/a

This network contains a single LAN with many loops inside. It enables the spanning tree protocol to prevent packet looping in the LAN.

3.29 SpanningTreeAdjust

topo name : spannnig_tree_adjust

args :

- l1_start: Endpoint interface of the 1st link on which we want to change the cost
- l1_end: Endpoint interface of the 1st link on which we want to change the cost
- l1_cost: Cost to set on the first link
- l2_start: Endpoint interface of the 2nd link on which we want to change the cost
- l2_end: Endpoint interface of the 2nd link on which we want to change the cost

- l2_cost: Cost to set on the second link

This network contains a single LAN with many loops inside. It enables the spanning tree protocol to prevent packets from looping in the network. The arguments of this topology allows users to change the STP cost on two links.

For instance, the following command changes STP cost of the link between s6.1 and s3.4 to 2:

```
python -m ipmininet.examples --topo=spannnig_tree_adjust --args l1_start=s6-eth1,l1_
↪end=s3-eth4,l1_cost=2
```

3.30 SpanningTreeCost

topo name: spannnig_tree_cost *args* : n/a

This network contains a single LAN with one loop inside. It enables the spanning tree protocol to prevent packet looping in the LAN. It also changes the STP cost one link.

3.31 DNSNetwork

topo name : dns_network *args* : n/a

This network contains two DNS server, a master and a slave. The domain name is 'mydomain.org' and it contains the address mapping of all the hosts of the network. You can query the DNS servers with, for instance, one of the following commands:

```
master dig @locahost -t NS mydomain.org
master dig @locahost -t AAAA server.mydomain.org
slave dig @locahost -t NS mydomain.org
slave dig @locahost -t AAAA server.mydomain.org
```

3.32 More examples

More examples can be found in this repository.

CHAPTER 4

Command-Line interface

Most of the IPMininet CLI functionality is similar to Mininet CLI. We extended it to support IPv6 addressing and routers. For instance, the *pingall* command will test both IPv4 and IPv6 connectivity between all hosts.

You can find more documentation (valid for both CLIs) on:

- Interact with hosts and switch
- Test connectivity between hosts
- Xterm display
- Other details

However, the *mn* command won't start a IPMininet topology but a Mininet one. If you want to try the IPMininet CLI, you can launch the following command:

```
$ sudo python -m ipmininet.examples --topo simple_ospf_network
```

To get the complete list of commands, when in the CLI, run:

```
mininet> help
```

To get details about a specific command, run:

```
mininet> help <command>
```

CHAPTER 5

Configuring daemons

We can add daemons to the routers or hosts and pass options to them. In the following code snippet, we add BGP daemon to r1.

```
from ipmininet.iptopo import IPTopo
from ipmininet.router.config import OSPF, OSPF6, RouterConfig

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        r1 = self.addRouter("r1", config=RouterConfig)
        r1.addDaemon(OSPF, hello_int=1)
        r1.addDaemon(OSPF6, hello_int=1)

        # [...]

        super(MyTopology, self).build(*args, **kwargs)
```

This page presents how to configure each daemon.

5.1 BGP

When adding BGP to a router with `router.addDaemon(BGP, **kargs)`, we change the following default parameters:

BGP.**set_defaults**(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged
- **address_families** – The set of AddressFamily to use

We can declare a set of routers in the same AS by using the overlay AS:

The overlay iBGPFullMesh extends the AS class and allows us to establish iBGP sessions in full mesh between BGP routers.

There are also some helper functions:

BGPConfig.**set_local_pref**(*local_pref*, *from_peer*, *matching=()*)

Set local pref on a peering with 'from_peer' on routes matching all of the access and community lists in 'matching'

Parameters

- **local_pref** – The local pref value to set
- **from_peer** – The peer on which the local pref is applied
- **matching** – A list of AccessList and/or CommunityList

Returns self

BGPConfig.**set_med**(*med*, *to_peer*, *matching=()*)

Set MED on a peering with 'to_peer' on routes matching all of the access and community lists in 'matching'

Parameters

- **med** – The local pref value to set
- **to_peer** – The peer to which the med is applied
- **matching** – A list of AccessList and/or CommunityList

Returns self

BGPConfig.**set_community**(*community*, *from_peer=None*, *to_peer=None*, *matching=()*)

Set community on a routes received from 'from_peer' and routes sent to 'to_peer' on routes matching all of the access and community lists in 'matching'

Parameters

- **community** – The community value to set
- **from_peer** – The peer on which received routes have to have the community
- **to_peer** – The peer on which sent routes have to have the community
- **matching** – A list of AccessList and/or CommunityList

Returns self

BGPConfig.**deny**(*name=None*, *from_peer=None*, *to_peer=None*, *matching=()*, *order=10*)

Deny all routes received from 'from_peer' and routes sent to 'to_peer' matching all of the access and community lists in 'matching'

Parameters

- **name** – The name of the route-map
- **from_peer** – The peer on which received routes have to have the community
- **to_peer** – The peer on which sent routes have to have the community
- **matching** – A list of AccessList and/or CommunityList
- **order** – The order in which route-maps are applied, i.e., lower order means applied before

Returns self

BGPConfig.**permit**(*name=None*, *from_peer=None*, *to_peer=None*, *matching=()*, *order=10*)

Accept all routes received from 'from_peer' and routes sent to 'to_peer' matching all of the access and community lists in 'matching'

Parameters

- **name** – The name of the route-map
- **from_peer** – The peer on which received routes have to have the community
- **to_peer** – The peer on which sent routes have to have the community
- **matching** – A list of AccessList and/or CommunityList
- **order** – The order in which route-maps are applied, i.e., lower order means applied before

Returns self

bgp.**bgp_fullmesh**(*routers*)
Establish a full-mesh set of BGP peerings between routers

Parameters **routers** – The set of routers peering within each other

bgp.**bgp_peering**(*a*, *b*)
Register a BGP peering between two nodes

bgp.**ebgp_session**(*a*, *b*, *link_type=None*)
Register an eBGP peering between two nodes, and disable IGP adjacencies between them.

Parameters

- **topo** – The current topology
- **a** – Local router
- **b** – Peer router
- **link_type** – Can be set to SHARE or CLIENT_PROVIDER. In this case ebgp_session will create import and export filter and set local pref based on the link type

bgp.**set_rr**(*rr*, *peers=()*)
Set rr as route reflector for all router r

Parameters

- **topo** – The current topology
- **rr** – The route reflector
- **peers** – Clients of the route reflector

The following code shows how to use all these abstractions:

```
from ipmininet.iptopo import IPTopo
from ipmininet.router.config import BGP, bgp_fullmesh, bgp_peering, \
    ebgp_session, RouterConfig, AccessList, CommunityList

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        # AS1 routers
        as1r1 = self.addRouter("as1r1", config=RouterConfig)
```

(continues on next page)

(continued from previous page)

```
        as1r1.addDaemon(BGP)
        as1r2 = self.addRouter("as1r2", config=RouterConfig)
        as1r2.addDaemon(BGP)
        as1r3 = self.addRouter("as1r3", config=RouterConfig)
        as1r3.addDaemon(BGP)

        self.addLink(as1r1, as1r2)
        self.addLink(as1r1, as1r3)
        self.addLink(as1r2, as1r3)

        # AS2 routers
        as2r1 = self.addRouter("as2r1", config=RouterConfig)
        as2r1.addDaemon(BGP)
        as2r2 = self.addRouter("as2r2", config=RouterConfig)
        as2r2.addDaemon(BGP)
        as2r3 = self.addRouter("as2r3", config=RouterConfig)
        as2r3.addDaemon(BGP)

        self.addLink(as2r1, as2r2)
        self.addLink(as2r1, as2r3)
        self.addLink(as2r2, as2r3)

        # AS3 routers
        as3r1 = self.addRouter("as3r1", config=RouterConfig)
        as3r1.addDaemon(BGP)
        as3r2 = self.addRouter("as3r2", config=RouterConfig)
        as3r2.addDaemon(BGP)
        as3r3 = self.addRouter("as3r3", config=RouterConfig)
        as3r3.addDaemon(BGP)

        self.addLink(as3r1, as3r2)
        self.addLink(as3r1, as3r3)
        self.addLink(as3r2, as3r3)

        # Inter-AS links
        self.addLink(as1r1, as2r1)
        self.addLink(as2r3, as3r1)

        # Add an access list to 'any'
        # This can be an IP prefix or address instead
        all_al = AccessList('all', ('any',))

        # Add a community list to as2r1
        loc_pref = CommunityList('loc-pref', '2:80')

        # as2r1 set the local pref of all the route coming from as1r1 and matching␣
↪the community list community to 80
        as2r1.get_config(BGP).set_local_pref(80, from_peer=as1r1, matching=(loc_pref,
↪))

        # as1r1 set the community of all the route sent to as2r1 and matching the␣
↪access list all_al to 2:80
        as1r1.get_config(BGP).set_community('2:80', to_peer=as2r1, matching=(all_al,))

        #  as3r1 set the med of all the route coming from as2r3 and matching the␣
↪access list all_al to 50
        as3r1.get_config(BGP).set_med(50, to_peer=as2r3, matching=(all_al,))
```

(continues on next page)

(continued from previous page)

```
        # AS1 is composed of 3 routers that have a full-mesh set of iBGP peering␣
↪between them
        self.addiBGPFullMesh(1, routers=[as1r1, as1r2, as1r3])

        # AS2 only has one iBGP session between its routers
        self.addAS(2, routers=[as2r1, as2r2, as2r3])
        bgp_peering(self, as2r1, as2r3)

        # AS3 is also composed of 3 routers that have a full-mesh set of iBGP peering␣
↪between them
        self.addAS(3, routers=[as3r1, as3r2, as3r3])
        bgp_fullmesh(self, [as3r1, as3r2, as3r3])

        # Establish eBGP sessions between ASes
        ebgp_session(self, as1r1, as2r1)
        ebgp_session(self, as2r3, as3r1)

        super(MyTopology, self).build(*args, **kwargs)
```

5.2 IPTables

This is currently mainly a proxy class to generate a list of static rules to pass to iptables. As such, see *man iptables* and *man iptables-extensions* to see the various table names, commands, pre-existing chains, ...

It takes one parameter:

`IPTables.`**`set_defaults`**(*defaults*)

Parameters **`rules`** – The (ordered) list of iptables rules that should be executed. If a rule is an iterable of strings, these will be joined using a space.

5.3 IP6Tables

This class is the IPv6 equivalent to IPTables.

It also takes one parameter:

`IP6Tables.`**`set_defaults`**(*defaults*)

Parameters **`rules`** – The (ordered) list of iptables rules that should be executed. If a rule is an iterable of strings, these will be joined using a space.

5.4 OpenR

The OpenR daemon can be tuned by adding keyword arguments to `router.addDaemon(OpenR, **kargs)`. Here is a list of the parameters:

`OpenrDaemon.`**`_defaults`**(***kwargs*)

Default parameters of the OpenR daemon. The template file openr.mako sets the default parameters listed here. See: https://github.com/facebook/openr/blob/master/openr/docs/Runbook.md.

Parameters

- **alloc_prefix_len** – Block size of allocated prefix in terms of it's prefix length. In this case '/80' prefix will be elected for a node. e.g. 'face:b00c:0:0:1234::/80'. Default: 128.
- **assume_drained** – Default: False.
- **config_store_filepath** – Default: /tmp/aq_persistent_config_store.bin
- **decision_debounce_max_ms** – Knobs to control how often to run Decision. On receipt of first even debounce is created with MIN time which grows exponentially up to max if there are more events before debounce is executed. This helps us to react to single network failures quickly enough (with min duration) while avoid high CPU utilization under heavy network churn. Default: 250.
- **decision_debounce_min_ms** – Knobs to control how often to run Decision. On receipt of first even debounce is created with MIN time which grows exponentially up to max if there are more events before debounce is executed. This helps us to react to single network failures quickly enough (with min duration) while avoid high CPU utilization under heavy network churn. Default: 10.
- **decision_rep_port** – Default: 60004.
- **domain** – Name of domain this node is part of. OpenR will 'only' form adjacencies to OpenR instances within it's own domain. This option becomes very useful if you want to run OpenR on two nodes adjacent to each other but belonging to different domains, e.g. Data Center and Wide Area Network. Usually it should depict the Network. Default: openr.
- **dryrun** – OpenR will not try to program routes in it's default configuration. You should explicitly set this option to false to proceed with route programming. Default: False.
- **enable_subnet_validation** – OpenR supports subnet validation to avoid miscabling of v4 addresses on different subnets on each end of the link. Need to enable v4 and this flag at the same time to turn on validation. Default: True.
- **enable_fib_sync** – Default: False.
- **enable_health_checker** – OpenR can measure network health internally by pinging other nodes in the network and exports this information as counters or via breeze APIs. By default health checker is disabled. The expectation is that each node must have at least one v6 loopback addressed announced into the network for the reachability check. Default: False.
- **enable_legacy_flooding** – Default: True.
- **enable_lfa** – With this option, additional Loop-Free Alternate (LFA) routes can be computed, per RFC 5286, for fast failure recovery. Under the failure of all primary nexthops for a prefix, because of link failure, next best precomputed LFA will be used without need of an SPF run. Default: False.
- **enable_netlink_fib_handler** – Knob to enable/disable default implementation of 'FibService' that comes along with OpenR for Linux platform. If you want to run your own FIB service then disable this option. Default: True.
- **enable_netlink_system_handler** – Knob to enable/disable default implementation of 'SystemService' and 'PlatformPublisher' that comes along with OpenR for Linux platform. If you want to run your own SystemService then disable this option. Default: True.
- **enable_perf_measurement** – Experimental feature to measure convergence performance. Performance information can be viewed via breeze API 'breeze perf fib'. Default: True.

- **enable_prefix_alloc** – Enable prefix allocator to elect and assign a unique prefix for the node. You will need to specify other configuration parameters below. Default: False.
- **enable_rtt_metric** – Default mechanism for cost of a link is '1' and hence cost of path is hop count. With this option you can ask OpenR to compute and use RTT of a link as a metric value. You should only use this for networks where links have significant delay, on the order of a couple of milliseconds. Using this for point-to-point links will cause lot of churn in metric updates as measured RTT will fluctuate a lot because of packet processing overhead. RTT is measured at application level and hence the fluctuation for point-to-point links. Default: True.
- **enable_secure_thrift_server** – Flag to enable TLS for our thrift server. Disable this for plaintext thrift. Default: False.
- **enable_segment_routing** – Experinmental and partially implemented segment routing feature. As of now it only elects node/adjacency labels. In future we will extend it to compute and program FIB routes. Default: False.
- **enable_spark** – Default: True.
- **enable_v4** – OpenR supports v4 as well but it needs to be turned on explicitly. It is expected that each interface will have v4 address configured for link local transport and v4/v6 topologies are congruent. Default: False.
- **enable_watchdog** – Default: True.
- **fib_handler_port** – TCP port on which 'FibService' will be listening. Default: 60100.
- **fib_rep_port** – Default: 60009.
- **health_checker_ping_interval_s** – Configure ping interval of the health checker. The below option configures it to ping all other nodes every 3 seconds. Default: 3.
- **health_checker_rep_port** – Default: 60012.
- **ifname_prefix** – Interface prefixes to perform neighbor discovery on. All interfaces whose names start with these are used for neighbor discovery. Default: ""
- **iface_regex_exclude** – Default:"".
- **iface_regex_include** – Default: "".
- **ip_tos** – Set type of service (TOS) value with which every control plane packet from Open/R will be marked with. This marking can be used to prioritize control plane traffic (as compared to data plane) so that congestion in network doesn't affect operations of Open/R. Default: 192
- **key_prefix_filters** – This comma separated string is used to set the key prefixes when key prefix filter is enabled (See SET_LEAF_NODE). It is also set when requesting KEY_DUMP from peer to request keys that match one of these prefixes. Default: "".
- **kvstore_flood_msg_per_sec** – Default: 0.
- **kvstore_flood_msg_burst_size** – Default: 0.
- **kvstore_flood_msg_per_sec** – Default: 0.
- **kvstore_ttl_decrement_ms** – Default: 1.
- **kvstore_zmq_hwm** – Set buffering size for KvStore socket communication. Updates to neighbor node during flooding can be buffered upto this number. For larger networks where burst of updates can be high having high value makes sense. For smaller networks where burst of updates are low, having low value makes more sense. Default: 65536.

- **link_flap_initial_backoff_ms** – Default: 1000.
- **link_flap_max_backoff_ms** – Default: 60000.
- **link_monitor_cmd_port** – Default: 60006.
- **loopback_iface** – Indicates loopback address to which auto elected prefix will be assigned if enabled. Default: “lo”.
- **memory_limit_mb** – Enforce upper limit on amount of memory in mega-bytes that open/r process can use. Above this limit watchdog thread will trigger crash. Service can be auto-restarted via system or some kind of service manager. This is very useful to guarantee protocol doesn’t cause trouble to other services on device where it runs and takes care of slow memory leak kind of issues. Default: 300.
- **minloglevel** – Log messages at or above this level. Again, the numbers of severity levels INFO, WARNING, ERROR, and FATAL are 0, 1, 2, and 3, respectively. Default: 0.
- **node_name** – Name of the OpenR node. Cruical setting if you run multiple nodes. Default: “”.
- **override_loopback_addr** – Whenever new address is elected for a node, before assigning it to interface all previously allocated prefixes or other global prefixes will be overridden with the new one. Use it with care! Default: False.
- **prefix_manager_cmd_port** – Default: 60011.
- **prefixes** – Static list of comma separate prefixes to announce from the current node. Can’t be changed while running. Default: “”.
- **redistribute_ifaces** – Comma separated list of interface names whose ‘/32’ (for v4) and ‘/128’ (for v6) should be announced. OpenR will monitor address add/remove activity on this interface and announce it to rest of the network. Default: “lo1”.
- **seed_prefix** – In order to elect a prefix for the node a super prefix to elect from is required. This is only applicable when ‘ENABLE_PREFIX_ALLOC’ is set to true. Default: “”.
- **set_leaf_node** – Sometimes a node maybe a leaf node and have only one path in to network. This node does not require to keep track of the entire topology. In this case, it may be useful to optimize memory by reducing the amount of key/vals tracked by the node. Setting this flag enables key prefix filters defined by KEY_PREFIX_FILTERS. A node only tracks keys in kvstore that matches one of the prefixes in KEY_PREFIX_FILTERS. Default: False.
- **set_loopback_address** – If set to true along with ‘ENABLE_PREFIX_ALLOC’ then second valid IP address of the block will be assigned onto ‘LOOPBACK_IFACE’ interface. e.g. in this case ‘face:b00c:0:0:1234::1/80’ will be assigned on ‘lo’ interface. Default: False.
- **spark_fastinit_keepalive_time_ms** – When interface is detected UP, OpenR can perform fast initial neighbor discovery as opposed to slower keep alive packets. Default value is 100 which means neighbor will be discovered within 200ms on a link. Default: 100.
- **spark_hold_time_s** – Hold time indicating time in seconds from it’s last hello after which neighbor will be declared as down. Default: 30.
- **spark_keepalive_time_s** – How often to send spark hello messages to neighbors. Default: 3.
- **static_prefix_alloc** – Default: False.
- **tls_acceptable_peers** – A comma separated list of strings. Strings are x509 common names to accept SSL connections from. Default: “”

- **tls_ecc_curve_name** – If we are running an SSL thrift server, this option specifies the eccCurveName for the associated wangle::SSLContextConfig. Default: "prime256v1".
- **tls_ticket_seed_path** – If we are running an SSL thrift server, this option specifies the TLS ticket seed file path to use for client session resumption. Default: "".
- **x509_ca_path** – If we are running an SSL thrift server, this option specifies the certificate authority path for verifying peers. Default: "".
- **x509_cert_path** – If we are running an SSL thrift server, this option specifies the certificate path for the associated wangle::SSLContextConfig. Default: "".
- **x509_key_path** – If we are running an SSL thrift server, this option specifies the key path for the associated wangle::SSLContextConfig. Default: "".
- **logbufsecs** – Default: 0
- **log_dir** – Directory to store log files at. The folder must exist. Default: /var/log.
- **max_log_size** – Default: 1.
- **v** – Show all verbose 'VLOG(m)' messages for m less or equal the value of this flag. Use higher value for more verbose logging. Default: 1.

5.5 OSPF

You can add keyword arguments to `router.addDaemon(OSPF, **kargs)` to change the following parameters:

`OSPF.`**`set_defaults`**(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged
- **dead_int** – Dead interval timer
- **hello_int** – Hello interval timer
- **priority** – priority for the interface, used for DR election
- **redistribute** – set of OSPFRedistributedRoute sources

This daemon also uses the following interface parameters:

- igp_passive: Whether the interface is passive (default value: False)
- ospf_dead_int: Dead interval timer specific to this interface (default value: `dead_int` parameter)
- ospf_hello_int: Hello interval timer specific to this interface (default value: `hello_int` parameter)
- ospf_priority: Priority for this specific to this interface (default value: `priority` parameter)

OSPF uses two link parameters:

- igp_cost: The IGP cost of the link (default value: 1)
- igp_area: The OSPF area of the link (default value: '0.0.0.0')

We can pass parameters to links and interfaces when calling `addLink()`:

```
from ipmininet.iptopo import IPTopo

class MyTopology(IPTopo):

```

(continues on next page)

(continued from previous page)

```
    def build(self, *args, **kwargs):

        # Add routers (OSPF daemon is added by default with the default config)
        router1 = self.addRouter("router1")
        router2 = self.addRouter("router2")

        # Add link
        l = self.addLink(router1, router2,
                         igp_cost=5, igp_area="0.0.0.1")  # Link parameters
        l[router1].addParams(ospf_dead_int=1)             # Router1 interface␣
↪parameters
        l[router2].addParams(ospf_priority=1)             # Router2 interface␣
↪parameters

        super(MyTopology, self).build(*args, **kwargs)
```

OSPF can use an overlay to declare with routers or links are completely in a given OSPF area. The following code adds all the interfaces of router r1 to '0.0.0.1' while the link between r2 and r3 is in area '0.0.0.5':

```
from ipmininet.iptopo import IPTopo

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        # Add routers (OSPF daemon is added by default with the default config)
        r1 = self.addRouter("r1")
        r2 = self.addRouter("r2")
        r3 = self.addRouter("r3")

        # Add links
        self.addLink(r1, r2)
        self.addLink(r1, r3)
        self.addLink(r2, r3)

        # Define OSPF areas
        self.addOSPFArea('0.0.0.1', routers=[r1], links=[])
        self.addOSPFArea('0.0.0.5', routers=[], links=[(r2, r3)])

        super(MyTopology, self).build(*args, **kwargs)
```

5.6 OSPF6

OSPF6 supports the same parameters as OSPF. It supports the following parameter:

OSPF6.**set_defaults**(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged
- **dead_int** – Dead interval timer
- **hello_int** – Hello interval timer
- **priority** – priority for the interface, used for DR election

- **redistribute** – set of OSPFRedistributedRoute sources
- **instance_id** – the number of the attached OSPF instance

OSPF6 uses one link parameter:

- igp_cost: The IGP cost of the link (default value: 1)

It uses the following interface parameters:

- igp_passive: Whether the interface is passive (default value: False)
- instance_id: The number of the attached OSPF6 instance (default value: 0)
- ospf6_dead_int: Dead interval timer specific to this interface (default value: `ospf_dead_int` parameter)
- ospf6_hello_int: Hello interval timer specific to this interface (default value: `ospf_hello_int` parameter)
- ospf6_priority: Priority for this specific to this interface (default value: `ospf_priority` parameter)

```
from ipmininet.iptopo import IPTopo

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        # Add routers (OSPF daemon is added by default with the default config)
        router1 = self.addRouter("router1")
        router2 = self.addRouter("router2")

        # Add link
        l = self.addLink(router1, router2,
                         igp_cost=5)             # Link parameters
        l[router1].addParams(ospf6_dead_int=1)   # Router1 interface parameters
        l[router2].addParams(ospf6_priority=1)   # Router2 interface parameters

        super(MyTopology, self).build(*args, **kwargs)
```

5.7 PIMD

When adding PIMD to a router with `router.addDaemon(PIMD, **kargs)`, we can give the following parameters:

`PIMD.`**`set_defaults`**(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged
- **multicast_ssm** – Enable pim ssm mode by default or not
- **multicast_igmp** – Enable igmp by default or not

5.8 Named

When adding PIMD to a host with `host.addDaemon(Named, **kargs)`, we can give the following parameters:

`Named.`**`set_defaults`**(*defaults*)

Parameters

- **log_severity** – It controls the logging levels and may take the values defined. Logging will occur for any message equal to or higher than the level specified (=>) lower levels will not be logged. These levels are 'critical', 'error', 'warning', 'notice', 'info', 'debug' and 'dynamic'.
- **dns_server_port** – The port number of the dns server

Named uses an overlay to declare DNS zones:

DNSZone.**__init__**(*name*, *dns_master*, *dns_slaves=()*, *records=()*, *nodes=()*, *refresh_time=86400*, *retry_time=7200*, *expire_time=3600000*, *min_ttl=172800*, *ns_domain_name=None*)

Parameters

- **name** – The domain name of the zone
- **dns_master** – The name of the master DNS server
- **dns_slaves** – The list of names of DNS slaves
- **records** – The list of DNS Records to be included in the zone
- **nodes** – The list of nodes for which one A/AAAA record has to be created for each of their IPv4/IPv6 addresses
- **refresh_time** – The number of seconds before the zone should be refreshed
- **retry_time** – The number of seconds before a failed refresh should be retried
- **expire_time** – The upper limit in seconds before a zone is considered no longer authoritative
- **min_ttl** – The negative result TTL
- **ns_domain_name** – If it is defined, it is the suffix of the domain of the name servers, otherwise, parameter 'name' is used.

The following code will create a DNS server in dns_master and dns_slave with one DNS zone: 'mydomain.org'. This will also create one reverse DNS zones for both IPv4 and IPv6.

```
from ipmininet.iptopo import IPTopo
from ipmininet.host.config import Named

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        # Add router
        r = self.addRouter("r")

        # Add hosts
        h1 = self.addHost("h1")
        h2 = self.addHost("h2")
        h3 = self.addHost("h3")
        dns_master = self.addHost("dns_master")
        dns_master.addDaemon(Named)
        dns_slave = self.addHost("dns_slave")
        dns_slave.addDaemon(Named)

        # Add links
        for h in self.hosts():
```

(continues on next page)

(continued from previous page)

```
            self.addLink(r, h)

        # Define a DNS Zone
        self.addDNSZone(name="mydomain.org",
                        dns_master=dns_master,
                        dns_slaves=[dns_slave],
                        nodes=self.hosts())

        super(MyTopology, self).build(*args, **kwargs)
```

By default, the DNSZone will create all the NS, A and AAAA records. If you need to change the TTL of one of these record, you can define it explicitly.

```
from ipmininet.iptopo import IPTopo
from ipmininet.host.config import Named, NSRecord

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        # Add router
        r = self.addRouter("r")

        # Add hosts
        h1 = self.addHost("h1")
        h2 = self.addHost("h2")
        h3 = self.addHost("h3")
        dns_master = self.addHost("dns_master")
        dns_master.addDaemon(Named)
        dns_slave = self.addHost("dns_slave")
        dns_slave.addDaemon(Named)

        # Add links
        for h in self.hosts():
            self.addLink(r, h)

        # Define a DNS Zone
        records = [NSRecord("mydomain.org", dns_master, ttl=120), NSRecord("mydomain.
↪org", dns_slave, ttl=120)]
        self.addDNSZone(name="mydomain.org",
                        dns_master=dns_master,
                        dns_slaves=[dns_slave],
                        records=records,
                        nodes=self.hosts())

        super(MyTopology, self).build(*args, **kwargs)
```

By default, one reverse DNS zone are created for all A records and another for all AAAA records. However, you may want to split the PTR records more than two different zones. You may also want to change the default values of the zones or their PTR records. To change this, you can declare the reverse DNS zone yourself. No need to add the PTR records that you don't want to modify, they will be created for you and placed in the zone that you declared if they fit in its domain name. Otherwise, another zone will be created.

```
from ipmininet.iptopo import IPTopo
from ipmininet.host.config import Named, PTRRecord
from ipaddress import ip_address
```

(continues on next page)

(continued from previous page)

```
class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        # Add router
        r = self.addRouter("r")

        # Add hosts
        h1 = self.addHost("h1")
        h2 = self.addHost("h2")
        h3 = self.addHost("h3")
        dns_master = self.addHost("dns_master")
        dns_master.addDaemon(Named)
        dns_slave = self.addHost("dns_slave")
        dns_slave.addDaemon(Named)

        # Add links
        for h in [h1, h2, dns_master, dns_slave]:
            self.addLink(r, h)
        lrh3 = self.addLink(r, h3)
        self.addSubnet(links=[lrh3], subnets=["192.168.0.0/24", "fc00::/64"])

        # Define a DNS Zone
        self.addDNSZone(name="mydomain.org",
                        dns_master=dns_master,
                        dns_slaves=[dns_slave],
                        nodes=self.hosts())

        # Change the TTL of one PTR record and the retry_time of its zone
        ptr_record = PTRRecord("fc00::2", h3 + ".mydomain.org", ttl=120)
        reverse_domain_name = ip_address(u"fc00::").reverse_pointer[-10:]  # keeps "f.
↪ip6.arpa"
        self.addDNSZone(name=reverse_domain_name, dns_master=dns_master, dns_
↪slaves=[dns_slave],
                        records=[ptr_record], ns_domain_name="mydomain.org", retry_
↪time=8200)

        super(MyTopology, self).build(*args, **kwargs)
```

5.9 RADVD

When adding RADVD to a router with `router.addDaemon(RADVD, **kargs)`, we can give the following parameters:

RADVD.**set_defaults**(*defaults*)

Parameters **debuglevel** – Turn on debugging information. Takes an integer between 0 and 5, where 0 completely turns off debugging, and 5 is extremely verbose. (see radvd(8) for more details)

This daemon also uses the following interface parameters:

- ra: A list of AdvPrefix objects that describes the prefixes to advertise
- rdnss: A list of AdvRDNSS objects that describes the DNS servers to advertise

```
from ipmininet.iptopo import IPTopo
from ipmininet.router.config import RADVD, AdvPrefix, AdvRDNSS

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        r = self.addRouter('r')
        r.addDaemon(RADVD, debug=0)

        h = self.addHost('h')
        dns = self.addHost('dns')

        lrh = self.addLink(r, h)
        lrh[r].addParams(ip=("2001:1341::1/64", "2001:2141::1/64"),
                         ra=[AdvPrefix("2001:1341::/64", valid_lifetime=86400,␣
↪preferred_lifetime=14400),
                             AdvPrefix("2001:2141::/64")],
                         rdnss=[AdvRDNSS("2001:89ab::d", max_lifetime=25),
                                AdvRDNSS("2001:cdef::d", max_lifetime=25)])
        lrdns = self.addLink(r, dns)
        lrdns[r].addParams(ip=("2001:89ab::1/64", "2001:cdef::1/64"))     # Static IP␣
↪addresses
        lrdns[dns].addParams(ip=("2001:89ab::d/64", "2001:cdef::d/64"))   # Static IP␣
↪addresses

        super(MyTopology, self).build(*args, **kwargs)
```

Instead of giving all addresses explicitly, you can use AdvConnectedPrefix() to advertise all the prefixes of the interface. You can also give the name of the DNS server (instead of an IP address) in the AdvRDNSS constructor.

```
from ipmininet.iptopo import IPTopo
from ipmininet.router.config import RouterConfig, RADVD, AdvConnectedPrefix, AdvRDNSS

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        r = self.addRouter('r')
        r.addDaemon(RADVD, debug=0)

        h = self.addHost('h')
        dns = self.addHost('dns')

        lrh = self.addLink(r, h)
        lrh[r].addParams(ip=("2001:1341::1/64", "2001:2141::1/64"),
                         ra=[AdvConnectedPrefix(valid_lifetime=86400, preferred_
↪lifetime=14400)],
                         rdnss=[AdvRDNSS(dns, max_lifetime=25)])
        lrdns = self.addLink(r, dns)
        lrdns[r].addParams(ip=("2001:89ab::1/64", "2001:cdef::1/64"))     # Static IP␣
↪addresses
        lrdns[dns].addParams(ip=("2001:89ab::d/64", "2001:cdef::d/64"))   # Static IP␣
↪addresses

        super(MyTopology, self).build(*args, **kwargs)
```

5.10 RIPng

When adding RIPng to a router with `router.addDaemon(RIPng, **kargs)`, we can give the following parameters:

`RIPng.`**`set_defaults`**(*defaults*)

Parameters

- **`debug`** – the set of debug events that should be logged (default: []).
- **`redistribute`** – set of RIPngRedistributedRoute sources (default: []).
- **`split_horizon`** – the daemon uses the split-horizon method (default: False).
- **`split_horizon_with_poison`** – the daemon uses the split-horizon. with reversed poison method. If both split_horizon_with_poison and split_horizon are set to True, RIPng will use the split-horizon with reversed poison method (default: True).
- **`update_timer`** – routing table timer value in second (default value:30).
- **`timeout_timer`** – routing information timeout timer (default value:180).
- **`garbage_timer`** – garbage collection timer (default value:120).

RIPng uses one link parameter:

- igp_metric: the metric of the link (default value: 1)

We can pass parameters to links when calling addLink():

```
from ipmininet.iptopo import IPTopo
from ipmininet.router.config import RIPng, RouterConfig

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):
        r1 = self.addRouter("r1", config=RouterConfig)  # We use RouterConfig to␣
↪prevent OSPF6 to be run
        r2 = self.addRouter("r2", config=RouterConfig)
        h1 = self.addHost("h1")
        h2 = self.addHost("h2")

        self.addLink(r1, r2, igp_metric=10)  # The IGP metric is set to 10
        self.addLink(r1, h1)
        self.addLink(r2, h2)

        r1.addDaemon(RIPng)
        r2.addDaemon(RIPng)

        super(MyTopology, self).build(*args, **kwargs)
```

5.11 SSHd

The SSHd daemon does not take any parameter. The SSH private and public keys are randomly generated but you can retrieve their paths with the following line:

```
from ipmininet.router.config.sshd import KEYFILE, PUBKEY
```

5.12 Zebra

FRRouting daemons (i.e., OSPF, OSPF6, BGP and PIMD) require this daemon and automatically trigger it. So we only need to explicitly add it through `router.addDaemon(Zebra, **kargs)` if we want to change one of its parameters:

`Zebra.`**`set_defaults`**(*defaults*)

Parameters

- **`debug`** – the set of debug events that should be logged
- **`access_lists`** – The set of AccessList to create, independently from the ones already included by route_maps
- **`route_maps`** – The set of RouteMap to create

CHAPTER 6

Configuring IPv4 and IPv6 networks

In Mininet, we can only use IPv4 in the emulated network. IPMininet enables the emulation of either IPv6-only or dual-stacked networks.

6.1 Dual-stacked networks

By default, your network is dual-stacked. It has both IPv4 and IPv6 addresses dynamically assigned by the library. Moreover, both OSPF and OSPF6 daemons are running on each router to ensure basic routing.

```
from ipmininet.iptopo import IPTopo
from ipmininet.ipnet import IPNet
from ipmininet.cli import IPCLI

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        r1 = self.addRouter("r1")
        r2 = self.addRouter("r2")
        h1 = self.addHost("h1")
        h2 = self.addHost("h2")

        self.addLink(h1, r1)
        self.addLink(r1, r2)
        self.addLink(r2, h2)

        super(MyTopology, self).build(*args, **kwargs)

net = IPNet(topo=MyTopology())
try:
    net.start()
    IPCLI(net)
```

(continues on next page)

(continued from previous page)

```
finally:
    net.stop()
```

If you wait for the network to converge and execute *pingall* in the IPMininet CLI, you will see that hosts can ping each other in both IPv4 and IPv6. You can also check the routes on the nodes with *<nodename> ip [-6|-4] route*.

6.2 Single-stacked networks

You can choose to make a whole network only in IPv4 or in IPv6 by using one parameter in the IPNet constructor. The two following examples show respectively an IPv4-only and IPv6-only network. In single stacked networks, only one of the routing daemons (either OSPF or OSPF6) is launched.

```
from ipmininet.iptopo import IPTopo
from ipmininet.ipnet import IPNet
from ipmininet.cli import IPCLI

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        r1 = self.addRouter("r1")
        r2 = self.addRouter("r2")
        h1 = self.addHost("h1")
        h2 = self.addHost("h2")

        self.addLink(h1, r1)
        self.addLink(r1, r2)
        self.addLink(r2, h2)

        super(MyTopology, self).build(*args, **kwargs)

net = IPNet(topo=MyTopology(), use_v6=False)  # This disables IPv6
try:
    net.start()
    IPCLI(net)
finally:
    net.stop()
```

```
from ipmininet.iptopo import IPTopo
from ipmininet.ipnet import IPNet
from ipmininet.cli import IPCLI

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        r1 = self.addRouter("r1")
        r2 = self.addRouter("r2")
        h1 = self.addHost("h1")
        h2 = self.addHost("h2")

        self.addLink(h1, r1)
        self.addLink(r1, r2)
        self.addLink(r2, h2)
```

(continues on next page)

(continued from previous page)

```
        super(MyTopology, self).build(*args, **kwargs)

net = IPNet(topo=MyTopology(), use_v4=False)  # This disables IPv4
try:
    net.start()
    IPCLI(net)
finally:
    net.stop()
```

6.3 Hybrids networks

In some cases, it is interesting to have only some parts of the network with IPv6 and/or IPv4. The hosts will have IPv4 (resp. IPv6) routes only if its access router has IPv4 (resp. IPv6) addresses. IPv4-only (resp. IPv6-only) routers won't have an OSPF (resp. OSPF6) daemon.

```
from ipmininet.iptopo import IPTopo
from ipmininet.ipnet import IPNet
from ipmininet.cli import IPCLI

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        r1 = self.addRouter("r1")
        r2 = self.addRouter("r2", use_v4=False)  # This disables IPv4 on the router
        r3 = self.addRouter("r3", use_v6=False)  # This disables IPv6 on the router
        h1 = self.addHost("h1")
        h2 = self.addHost("h2")
        h3 = self.addHost("h3")

        self.addLink(r1, r2)
        self.addLink(r1, r3)
        self.addLink(r2, r3)

        self.addLink(r1, h1)
        self.addLink(r2, h2)
        self.addLink(r3, h3)

        super(MyTopology, self).build(*args, **kwargs)

net = IPNet(topo=MyTopology())
try:
    net.start()
    IPCLI(net)
finally:
    net.stop()
```

6.4 Static addressing

Addresses are allocated dynamically by default but you can set your own addresses if you disable auto-allocation when creating the IPNet object. You can do so for both for router loopbacks and for real interfaces of all the nodes.

```
from ipmininet.iptopo import IPTopo
from ipmininet.ipnet import IPNet
from ipmininet.cli import IPCLI

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        r1 = self.addRouter("r1", lo_addresses=["2042:1::1/64", "10.1.0.1/24"])
        r2 = self.addRouter("r2", lo_addresses=["2042:2::1/64", "10.2.0.1/24"])
        h1 = self.addHost("h1")
        h2 = self.addHost("h2")

        lr1r2 = self.addLink(r1, r2)
        lr1r2[r1].addParams(ip=("2042:12::1/64", "10.12.0.1/24"))
        lr1r2[r2].addParams(ip=("2042:12::2/64", "10.12.0.2/24"))

        lr1h1 = self.addLink(r1, h1)
        lr1h1[r1].addParams(ip=("2042:1a::1/64", "10.51.0.1/24"))
        lr1h1[h1].addParams(ip=("2042:1a::a/64", "10.51.0.5/24"))

        lr2h2 = self.addLink(r2, h2)
        lr2h2[r2].addParams(ip=("2042:2b::2/64", "10.62.0.2/24"))
        lr2h2[r2].addParams(ip=("2042:2b::b/64", "10.62.0.6/24"))

        super(MyTopology, self).build(*args, **kwargs)

net = IPNet(topo=MyTopology(), allocate_IPs=False)  # Disable IP auto-allocation
try:
    net.start()
    IPCLI(net)
finally:
    net.stop()
```

You can also declare your subnets by declaring a Subnet overlay.

```
from ipmininet.iptopo import IPTopo
from ipmininet.ipnet import IPNet
from ipmininet.cli import IPCLI

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        r1 = self.addRouter("r1", lo_addresses=["2042:1::1/64", "10.1.0.1/24"])
        r2 = self.addRouter("r2", lo_addresses=["2042:2::1/64", "10.2.0.1/24"])
        h1 = self.addHost("h1")
        h2 = self.addHost("h2")

        lr1r2 = self.addLink(r1, r2)
        self.addLink(r1, h1)
        self.addLink(r2, h2)

        # The interfaces of the nodes and links on their common LAN
        # will get an address for each subnet.
        self.addSubnet(nodes=[r1, r2], subnets=["2042:12::/64", "10.12.0.0/24"])
        self.addSubnet(nodes=[r1, h1], subnets=["2042:1a::/64", "10.51.0.0/24"])
```

(continues on next page)

(continued from previous page)

```
        self.addSubnet(links=[lr1r2],  subnets=["2042:2b::/64", "10.62.0.0/24"])

        super(MyTopology, self).build(*args, **kwargs)

net = IPNet(topo=MyTopology(), allocate_IPs=False)  # Disable IP auto-allocation
try:
    net.start()
    IPCLI(net)
finally:
    net.stop()
```

6.5 Static routing

By default, OSPF and OSPF6 are launched on each router. If you want to prevent that, you have to change the router configuration class. You can change it when adding a new router to your topology.

```
from ipmininet.iptopo import IPTopo
from ipmininet.router.config import RouterConfig, STATIC, StaticRoute
from ipmininet.ipnet import IPNet
from ipmininet.cli import IPCLI

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        # Change the config object for RouterConfig
        # because it does not add by default OSPF or OSPF6
        r1 = self.addRouter("r1", config=RouterConfig, lo_addresses=["2042:1::1/64",
↪"10.1.0.1/24"])
        r2 = self.addRouter("r2", config=RouterConfig, lo_addresses=["2042:2::1/64",
↪"10.2.0.1/24"])
        h1 = self.addHost("h1")
        h2 = self.addHost("h2")

        lr1r2 = self.addLink(r1, r2)
        lr1r2[r1].addParams(ip=("2042:12::1/64", "10.12.0.1/24"))
        lr1r2[r2].addParams(ip=("2042:12::2/64", "10.12.0.2/24"))

        lr1h1 = self.addLink(r1, h1)
        lr1h1[r1].addParams(ip=("2042:1a::1/64", "10.51.0.1/24"))
        lr1h1[h1].addParams(ip=("2042:1a::a/64", "10.51.0.5/24"))

        lr2h2 = self.addLink(r2, h2)
        lr2h2[r2].addParams(ip=("2042:2b::2/64", "10.62.0.2/24"))
        lr2h2[r2].addParams(ip=("2042:2b::b/64", "10.62.0.6/24"))

        # Add static routes
        r1.addDaemon(STATIC, static_routes=[StaticRoute("2042:2b::/64", "2042:12::2"),
                                            StaticRoute("10.62.0.0/24", "10.12.0.2")])
        r2.addDaemon(STATIC, static_routes=[StaticRoute("2042:1a::/64", "2042:12::1"),
                                            StaticRoute("10.51.0.0/24", "10.12.0.1")])

        super(MyTopology, self).build(*args, **kwargs)
```

(continues on next page)

(continued from previous page)

```
net = IPNet(topo=MyTopology(), allocate_IPs=False)  # Disable IP auto-allocation
try:
    net.start()
    IPCLI(net)
finally:
    net.stop()
```

You can also add routes manually when the network has started since you can run any command (like in Mininet).

```
net = IPNet(topo=MyTopology(), allocate_IPs=False)  # Disable IP auto-allocation
try:
    net.start()

    # Static routes
    net["r1"].cmd("ip -6 route add 2042:2b::/64 via 2042:12::2")
    net["r1"].cmd("ip -4 route add 10.62.0.0/24 via 10.12.0.2")
    net["r2"].cmd("ip -6 route add 2042:1a::/64 via 2042:12::1")
    net["r2"].cmd("ip -4 route add 10.51.0.0/24 via 10.12.0.1")

    IPCLI(net)
finally:
    net.stop()
```

CHAPTER 7

Configuring a LAN

By default, IPMininet uses *IPSwitch* to create regular switches (not Openflow switches) and hubs. The switches use the Spanning Tree Protocol by default to break the loops. The hubs are switches that do not maintain a MAC address table and always broadcast any received frame on all its interfaces.

Here is an example of a LAN with a few switches and one hub.

```
from ipmininet.iptopo import IPTopo

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        # Switches
        s1 = self.addSwitch("s1")
        s2 = self.addSwitch("s2")
        s3 = self.addSwitch("s3")
        s4 = self.addSwitch("s4")

        # Hub
        hub1 = self.addHub("hub1")

        # Links
        self.addLink(s1, s2)
        self.addLink(s1, hub1)
        self.addLink(hub1, s3)
        self.addLink(hub1, s4)

        super(MyTopology, self).build(*args, **kwargs)
```

The Spanning Tree Protocol can be configured by changing the `stp_cost` on the links (or directly on each interface). The default cost is 1.

```
from ipmininet.iptopo import IPTopo
```

(continues on next page)

(continued from previous page)

```
class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        # Switches
        s1 = self.addSwitch("s1")
        s2 = self.addSwitch("s2")
        s3 = self.addSwitch("s3")
        s4 = self.addSwitch("s4")

        # Hub
        hub1 = self.addHub("hub1")

        # Links
        self.addLink(s1, s2, stp_cost=2)  # Cost changed for both interfaces
        self.addLink(s1, hub1)
        ls3 = self.addLink(hub1, s3)
        ls3[s3].addParams(stp_cost=2)  # Cost changed on a single interface
        self.addLink(hub1, s4)

        super(MyTopology, self).build(*args, **kwargs)
```

In the Spanning Tree Protocol, each switch has a priority. The lowest priority switch becomes the root of the spanning tree. By default, switches declared first have a lower priority number. You can manually set this value when you create the switch.

```
from ipmininet.iptopo import IPTopo

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        # Switches with manually set STP priority
        s1 = self.addSwitch("s1", prio=1)
        s2 = self.addSwitch("s2", prio=2)
        s3 = self.addSwitch("s3", prio=3)
        s4 = self.addSwitch("s4", prio=4)

        # Hub
        hub1 = self.addHub("hub1")

        # Links
        self.addLink(s1, s2, stp_cost=2)  # Cost changed for both interfaces
        self.addLink(s1, hub1)
        ls3 = self.addLink(hub1, s3)
        ls3[s3].addParams(stp_cost=2)  # Cost changed on a single interface
        self.addLink(hub1, s4)

        super(MyTopology, self).build(*args, **kwargs)
```

CHAPTER 8

Developer Guide

This section details some essential points to contribute to the code base. Don't hesitate to ask for advice by opening an issue on Github.

8.1 Setting up the development environment

To develop a new feature, you have to install IPMininet from source in development mode.

First get the source code of your fork:

```
$ git clone <your-fork-url>
$ cd ipmininet
```

Then, install your version of IPMininet in development mode. If you have pip above **18.1**, execute:

```
$ sudo pip -e install .
```

If you have an older version of pip, use:

```
$ sudo pip -e install --process-dependency-links .
```

Finally, you can install all the daemons:

```
$ sudo python -m ipmininet.install -af
```

8.2 Running the tests

The pytest framework is used for the test suite and are integrated within setuptools. Currently the suite has end-to-end tests that check if the daemons work as expected. Therefore, the tests require an operating environment, i.e. daemons have to be installed and must be in PATH.

To run the whole test suite go the top level directory and run:

```
sudo pytest
```

You can also run a single test by passing options to pytest:

```
sudo pytest ipmininet/tests/test_sshd.py --fulltrace
```

8.3 Building the documentation

First, you have to install the requirements to build the project. When at the root of the documentation, run:

```
pip install -r requirements.txt
```

Then you can generate the html documentation in the folder `docs/_build/html/` with:

```
make html
```

The examples in the documentation can also be tested when changing the code base with the following command:

```
sudo make doctest
```

8.4 Adding a new example

When adding a new example of topology to IPMininet, you have to perform the following tasks:

- Create a new `IPTopo` subclass in the folder `ipmininet/examples/`.
- Add the new class to the dictionary `TOPOS` of `ipmininet/examples/__main__.py`.
- Document its layout in the `build()` method docstring.
- Document the example in `ipmininet/examples/README.md`.
- Add a test to check the correctness of the example.

8.5 Adding a new daemon

When adding a new daemon to IPMininet, you have to perform the following tasks:

- Create a new mako template in the folder `ipmininet/router/config/templates/` or `ipmininet/host/config/templates/` for the daemon configuration.
- Create a new `RouterDaemon` or `HostDaemon` subclass in the folder `ipmininet/router/config/` or `ipmininet/host/config/`. The following things are required in this new subclass:
  - Set the class variable `NAME` with a unique name.
  - Set the class variable `KILL_PATTERNS` that lists all the process names that have to be cleaned if a user uses the cleaning command in *IPMininet network cleaning*.
  - Extend the property `startup_line` that gives the command line to launch the daemon.
  - Extend the property `dry_run` that gives the command line to check the generated configuration.
  - Extend the method `set_defaults()` to set default configuration values and document them all in the method docstring.

  - Extend the method `build()` to set the ConfigDict object that will be fed to the template.
  - Declare the daemon and its helper classes in `ipmininet/router/config/__init__.py` or `ipmininet/host/config/__init__.py`.
- Add at least one example for the users (see *Adding a new example*).
- Implement the tests to prove the correct configuration of the daemon.
- Update the setup of IPMininet to install the new daemon by updating `ipmininet/install/__main__.py` and `ipmininet/install/install.py`.
- Document the daemon and its configuration options in the sphinx documentation in `docs/daemons.rst`.

CHAPTER 9

IPMininet API

9.1 ipmininet package

This is a python library, extending [Mininet](http://mininet.org), in order to support emulation of (complex) IP networks. As such it provides new classes, such as Routers, auto-configures all properties not set by the user, such as IP addresses or router configuration files, ...

9.1.1 Subpackages

ipmininet.host package

This module defines a modular host that is able to support multiple daemons

class ipmininet.host.**IPHost**(*name*, *config=<class 'ipmininet.host.config.base.HostConfig'>*, **args*, ***kwargs*)

Bases: ipmininet.router.__router.IPNode

A Host which manages a set of daemons

class ipmininet.host.**CPULimitedHost**(*name*, *sched='cfs'*, ***kwargs*)

Bases: ipmininet.host.__host.IPHost

CPU limited host

cfsInfo(*f*)

Internal method: return parameters for CFS bandwidth

cgroupDel()

Clean up our cgroup

cgroupGet(*param*, *resource='cpu'*)

Return value of cgroup parameter

cgroupSet(*param*, *value*, *resource='cpu'*)

Set a cgroup parameter and return its value

classmethod checkRtGroupSched()
Check (Ubuntu,Debian) kernel config for CONFIG_RT_GROUP_SCHED for RT

chrt()
Set RT scheduling priority

cleanup()
Clean up Node, then clean up our cgroup

config(*cpu=-1*, *cores=None*, ***params*)
cpu: desired overall system CPU fraction cores: (real) core(s) this host can run on params: parameters for Node.config()

classmethod init()
Initialization for CPULimitedHost class

inited = False

popen(**args*, ***kwargs*)
Return a Popen() object in node's namespace args: Popen() args, single list, or string kwargs: Popen() keyword args

rtInfo(*f*)
Internal method: return parameters for RT bandwidth

setCPUFrac(*f*, *sched=None*)
Set overall CPU fraction for this host f: CPU bandwidth limit (positive fraction, or -1 for cfs unlimited) sched: 'rt' or 'cfs' Note 'cfs' requires CONFIG_CFS_BANDWIDTH, and 'rt' requires CONFIG_RT_GROUP_SCHED

setCPUs(*cores*, *mems=0*)
Specify (real) cores that our cgroup can run on

Subpackages

ipmininet.host.config package

This module holds the configuration generators for daemons that can be used in a host.

class ipmininet.host.config.**HostConfig**(*node*, *daemons=()*, *sysctl=None*, **args*, ***kwargs*)
Bases: *ipmininet.router.config.base.NodeConfig*

Initialize our config builder

Parameters

- **node** – The node for which this object will build configurations
- **daemons** – an iterable of active routing daemons for this node
- **sysctl** – A dictionnary of sysctl to set for this node. By default, it enables IPv4/IPv6 forwarding on all interfaces.

class ipmininet.host.config.**HostDaemon**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)
Bases: *ipmininet.router.config.base.Daemon*

class ipmininet.host.config.**Named**(*node*, ***kwargs*)
Bases: *ipmininet.host.config.base.HostDaemon*

KILL_PATTERNS = ('named',)

NAME = 'named'

build()
Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

build_largest_reverse_zone(*cfg_zones*, *records*)
Create the ConfigDict object representing a new reverse zone whose prefix is the largest one that includes all the PTR records. Then it adds it to the cfg_zones dict.

Parameters

- **cfg_zones** – The dict of ConfigDict representing existing zones
- **records** – The list of PTR records to place a new reverse zone

build_reverse_zone(*cfg_zones*)
Build non-existing PTR records. Then, adds them to an existing reverse zone if any. The remaining ones are inserted in a new reverse zone that is added to cfg_zones dictionary.

build_zone(*zone*)

cfg_filenames
Return the list of filenames in which this daemon config should be stored

dry_run
The startup line to use to check that the daemon is well-configured

set_defaults(*defaults*)

Parameters

- **log_severity** – It controls the logging levels and may take the values defined. Logging will occur for any message equal to or higher than the level specified (=>) lower levels will not be logged. These levels are 'critical', 'error', 'warning', 'notice', 'info', 'debug' and 'dynamic'.
- **dns_server_port** – The port number of the dns server

startup_line
Return the corresponding startup_line for this daemon

template_filenames

zone_filename(*domain_name*)

class ipmininet.host.config.**DNSZone**(*name*, *dns_master*, *dns_slaves=()*, *records=()*, *nodes=()*, *refresh_time=86400*, *retry_time=7200*, *expire_time=3600000*, *min_ttl=172800*, *ns_domain_name=None*)
Bases: *ipmininet.overlay.Overlay*

Parameters

- **name** – The domain name of the zone
- **dns_master** – The name of the master DNS server
- **dns_slaves** – The list of names of DNS slaves
- **records** – The list of DNS Records to be included in the zone
- **nodes** – The list of nodes for which one A/AAAA record has to be created for each of their IPv4/IPv6 addresses
- **refresh_time** – The number of seconds before the zone should be refreshed

- **retry_time** – The number of seconds before a failed refresh should be retried
- **expire_time** – The upper limit in seconds before a zone is considered no longer authoritative
- **min_ttl** – The negative result TTL
- **ns_domain_name** – If it is defined, it is the suffix of the domain of the name servers, otherwise, parameter 'name' is used.

apply(*topo*)
Apply the Overlay properties to the given topology

check_consistency(*topo*)
Check that this overlay is consistent

class ipmininet.host.config.**ARecord**(*domain_name*, *address*, *ttl=60*)
Bases: *ipmininet.host.config.named.DNSRecord*

rdata

class ipmininet.host.config.**NSRecord**(*domain_name*, *name_server*, *ttl=60*)
Bases: *ipmininet.host.config.named.DNSRecord*

rdata

class ipmininet.host.config.**AAAARecord**(*domain_name*, *address*, *ttl=60*)
Bases: *ipmininet.host.config.named.ARecord*

class ipmininet.host.config.**SOARecord**(*domain_name*, *refresh_time=86400*, *retry_time=7200*, *expire_time=3600000*, *min_ttl=172800*, *records=()*)
Bases: *ipmininet.host.config.named.DNSRecord*

add_record(*record*)

rdata

records

class ipmininet.host.config.**PTRRecord**(*address*, *domain_name*, *ttl=60*)
Bases: *ipmininet.host.config.named.DNSRecord*

rdata

v6

Submodules

ipmininet.host.config.base module

This modules provides a config object for a host, that is able to provide configurations for a set of daemons. It also defines the base class for a host daemon, as well as a minimalistic configuration for a host.

class ipmininet.host.config.base.**HostConfig**(*node*, *daemons=()*, *sysctl=None*, **args*, ***kwargs*)
Bases: *ipmininet.router.config.base.NodeConfig*

Initialize our config builder

Parameters

- **node** – The node for which this object will build configurations

- **daemons** – an iterable of active routing daemons for this node
- **sysctl** – A dictionnary of sysctl to set for this node. By default, it enables IPv4/IPv6 forwarding on all interfaces.

class ipmininet.host.config.base.**HostDaemon**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)

Bases: *ipmininet.router.config.base.Daemon*

ipmininet.host.config.named module

Base classes to configure a Named daemon

class ipmininet.host.config.named.**AAAARecord**(*domain_name*, *address*, *ttl=60*)

Bases: *ipmininet.host.config.named.ARecord*

class ipmininet.host.config.named.**ARecord**(*domain_name*, *address*, *ttl=60*)

Bases: *ipmininet.host.config.named.DNSRecord*

rdata

class ipmininet.host.config.named.**DNSRecord**(*rtype*, *domain_name*, *ttl=60*)

Bases: object

full_domain_name

rdata

class ipmininet.host.config.named.**DNSZone**(*name*, *dns_master*, *dns_slaves=()*, *records=()*, *nodes=()*, *refresh_time=86400*, *retry_time=7200*, *expire_time=3600000*, *min_ttl=172800*, *ns_domain_name=None*)

Bases: *ipmininet.overlay.Overlay*

Parameters

- **name** – The domain name of the zone
- **dns_master** – The name of the master DNS server
- **dns_slaves** – The list of names of DNS slaves
- **records** – The list of DNS Records to be included in the zone
- **nodes** – The list of nodes for which one A/AAAA record has to be created for each of their IPv4/IPv6 addresses
- **refresh_time** – The number of seconds before the zone should be refreshed
- **retry_time** – The number of seconds before a failed refresh should be retried
- **expire_time** – The upper limit in seconds before a zone is considered no longer authoritative
- **min_ttl** – The negative result TTL
- **ns_domain_name** – If it is defined, it is the suffix of the domain of the name servers, otherwise, parameter 'name' is used.

apply(*topo*)

Apply the Overlay properties to the given topology

check_consistency(*topo*)

Check that this overlay is consistent

class ipmininet.host.config.named.**NSRecord**(*domain_name*, *name_server*, *ttl=60*)
Bases: *ipmininet.host.config.named.DNSRecord*

rdata

class ipmininet.host.config.named.**Named**(*node*, ***kwargs*)
Bases: *ipmininet.host.config.base.HostDaemon*

KILL_PATTERNS = ('named',)

NAME = 'named'

build()
Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

build_largest_reverse_zone(*cfg_zones*, *records*)
Create the ConfigDict object representing a new reverse zone whose prefix is the largest one that includes all the PTR records. Then it adds it to the cfg_zones dict.

Parameters

- **cfg_zones** – The dict of ConfigDict representing existing zones
- **records** – The list of PTR records to place a new reverse zone

build_reverse_zone(*cfg_zones*)
Build non-existing PTR records. Then, adds them to an existing reverse zone if any. The remaining ones are inserted in a new reverse zone that is added to cfg_zones dictionary.

build_zone(*zone*)

cfg_filenames
Return the list of filenames in which this daemon config should be stored

dry_run
The startup line to use to check that the daemon is well-configured

set_defaults(*defaults*)

Parameters

- **log_severity** – It controls the logging levels and may take the values defined. Logging will occur for any message equal to or higher than the level specified (=>) lower levels will not be logged. These levels are 'critical', 'error', 'warning', 'notice', 'info', 'debug' and 'dynamic'.
- **dns_server_port** – The port number of the dns server

startup_line
Return the corresponding startup_line for this daemon

template_filenames

zone_filename(*domain_name*)

class ipmininet.host.config.named.**PTRRecord**(*address*, *domain_name*, *ttl=60*)
Bases: *ipmininet.host.config.named.DNSRecord*

rdata

v6

class ipmininet.host.config.named.**SOARecord**(*domain_name*, *refresh_time=86400*, *retry_time=7200*, *expire_time=3600000*, *min_ttl=172800*, *records=()*)

Bases: *ipmininet.host.config.named.DNSRecord*

add_record(*record*)

rdata

records

ipmininet.router package

This module defines a modular router that is able to support multiple daemons

class ipmininet.router.**IPNode**(*name*, *config=<class 'ipmininet.router.config.base.NodeConfig'>*, *cwd='/tmp'*, *process_manager=<class 'ipmininet.router.__router.ProcessHelper'>*, *use_v4=True*, *use_v6=True*, **args*, ***kwargs*)

Bases: mininet.node.Node

A Node which manages a set of daemons

Most of the heavy lifting for this node should happen in the associated config object.

Parameters

- **config** – The configuration generator for this node. Either a class or a tuple (class, kwargs)
- **cwd** – The base directory for temporary files such as configs
- **process_manager** – The class that will manage all the associated processes for this node
- **use_v4** – Whether this node has IPv4
- **use_v6** – Whether this node has IPv6

get(*key*, *val=None*)
Check for a given key in the node parameters

start()
Start the node: Configure the daemons, set the relevant sysctls, and fire up all needed processes

terminate()
Stops this node and sets back all sysctls to their old values

class ipmininet.router.**Router**(*name*, *config=<class 'ipmininet.router.config.base.BasicRouterConfig'>*, *password='zebra'*, *lo_addresses=()*, **args*, ***kwargs*)

Bases: ipmininet.router.__router.IPNode, *ipmininet.utils.L3Router*

The actual router, which manages a set of daemons

Parameters

- **password** – The password for the routing daemons vtysh access
- **lo_addresses** – The list of addresses to set on the loopback interface

asn

class ipmininet.router.**ProcessHelper**(*node*, **args*, ***kwargs*)

Bases: object

This class holds processes that are part of a given family, e.g. routing daemons. This also provides the abstraction to execute a new process, currently in a mininet namespace, but could be extended to execute in a different environment.

Parameters **node** – The object to use to create subprocesses.

call(**args*, ***kwargs*)
Call a command, wait for it to end and return its ouput.

Parameters

- **args** – the command + arguments
- **kwargs** – key-val arguments, as used in subprocess.Popen

get_process(*pid*)
Return a given process handle in this family

Parameters **pid** – a process index, as return by popen

pexec(**args*, ***kw*)
Call a command, wait for it to terminate and save stdout, stderr and its return code

popen(**args*, ***kwargs*)
Call a command and return a Popen handle to it.

Parameters

- **args** – the command + arguments
- **kwargs** – key-val arguments, as used in subprocess.Popen

Returns a process index in this family

terminate()
Terminate all processes in this family

Subpackages

ipmininet.router.config package

This module holds the configuration generators for daemons that can be used in a router.

class ipmininet.router.config.**BasicRouterConfig**(*node*, *daemons=()*, *additional_daemons=()*, **args*, ***kwargs*)
Bases: *ipmininet.router.config.base.RouterConfig*

A basic router that will run an OSPF daemon

A simple router made of at least an OSPF daemon

Parameters **additional_daemons** – Other daemons that should be used

class ipmininet.router.config.**NodeConfig**(*node*, *daemons=()*, *sysctl=None*, **args*, ***kwargs*)
Bases: object

This class manages a set of daemons, and generates the global configuration for a node

Initialize our config builder

Parameters

- **node** – The node for which this object will build configurations

- **daemons** – an iterable of active routing daemons for this node
- **sysctl** – A dictionnary of sysctl to set for this node. By default, it enables IPv4/IPv6 forwarding on all interfaces.

build()
Build the configuration for each daemon, then write the configuration files

cleanup()
Cleanup all temporary files for the daemons

daemon(*key*)
Return the Daemon object in this config for the given key

Parameters **key** – the daemon name or a daemon class or instance

Returns the Daemon object

Raises **KeyError** – if not found

daemons

post_register_daemons()
Method called after all daemon classes were instantiated

register_daemon(*cls*, ***daemon_opts*)
Add a new daemon to this configuration

Parameters

- **cls** – Daemon class or object, or a 2-tuple (Daemon, dict)
- **daemon_opts** – Options to set on the daemons

sysctl
Return an list of all sysctl to set on this node

class ipmininet.router.config.**Zebra**(**args*, ***kwargs*)
Bases: *ipmininet.router.config.zebra.QuaggaDaemon*

KILL_PATTERNS = ('zebra',)

NAME = 'zebra'

PRIO = 0

STARTUP_LINE_EXTRA = '-k'

build()
Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

has_started()
Return whether this daemon has started or not

listening()

set_defaults(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged
- **access_lists** – The set of AccessList to create, independently from the ones already included by route_maps
- **route_maps** – The set of RouteMap to create

class ipmininet.router.config.**OSPF**(*node*, **args*, ***kwargs*)
Bases: *ipmininet.router.config.zebra.QuaggaDaemon*

This class provides a simple configuration for an OSPF daemon. It advertizes one network per interface (the primary one), and set interfaces not facing another L3Router to passive

DEPENDS = (<class 'ipmininet.router.config.zebra.Zebra'>,)

KILL_PATTERNS = ('ospfd',)

NAME = 'ospfd'

build()
Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

static is_active_interface(*itf*)
Return whether an interface is active or not for the OSPF daemon

set_defaults(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged
- **dead_int** – Dead interval timer
- **hello_int** – Hello interval timer
- **priority** – priority for the interface, used for DR election
- **redistribute** – set of OSPFRedistributedRoute sources

class ipmininet.router.config.**OSPF6**(*node*, **args*, ***kwargs*)
Bases: *ipmininet.router.config.ospf.OSPF*

This class provides a simple configuration for an OSPF6 daemon. It advertizes one network per interface (the primary one), and set interfaces not facing another L3Router to passive

DEAD_INT = 3

KILL_PATTERNS = ('ospf6d',)

NAME = 'ospf6d'

set_defaults(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged
- **dead_int** – Dead interval timer
- **hello_int** – Hello interval timer
- **priority** – priority for the interface, used for DR election
- **redistribute** – set of OSPFRedistributedRoute sources
- **instance_id** – the number of the attached OSPF instance

class ipmininet.router.config.**OSPFArea**(*area*, *routers=()*, *links=()*, ***props*)
Bases: *ipmininet.overlay.Overlay*

An overlay to group OSPF links and routers by area

Parameters

- **area** – the area for this overlay
- **routers** – the set of routers for which all their interfaces belong to that area
- **links** – individual links belonging to this area

apply(*topo*)
Apply the Overlay properties to the given topology

area

class ipmininet.router.config.**BGP**(*node*, *port=179*, **args*, ***kwargs*)
Bases: *ipmininet.router.config.zebra.QuaggaDaemon*

This class provides the configuration skeletons for BGP routers.

DEPENDS = (<class 'ipmininet.router.config.zebra.Zebra'>,)

KILL_PATTERNS = ('bgpd',)

NAME = 'bgpd'

STARTUP_LINE_EXTRA
We add the port to the standard startup line

build()
Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

build_access_list()
Build and return a list of acess_filter :return:

build_community_list()
Build and return a list of community_filter

build_route_map(*neigbors*)
Build and return a list of route map for the current node

classmethod get_config(*topo*, *router*, ***kwargs*)
Returns a config object for the daemon if any

set_defaults(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged
- **address_families** – The set of AddressFamily to use

class ipmininet.router.config.**AS**(*asn*, *routers=()*, ***props*)
Bases: *ipmininet.overlay.Overlay*

An overlay class that groups routers by AS number

Parameters

- **asn** – The number for this AS
- **routers** – an initial set of routers to add to this AS
- **props** – key-vals to set on all routers of this AS

asn

class ipmininet.router.config.**iBGPFullMesh**(*asn*, *routers=()*, ***props*)
Bases: *ipmininet.router.config.bgp.AS*

An overlay class to establish iBGP sessions in full mesh between BGP routers.

Parameters

- **asn** – The number for this AS
- **routers** – an initial set of routers to add to this AS
- **props** – key-vals to set on all routers of this AS

apply(*topo*)
Apply the Overlay properties to the given topology

ipmininet.router.config.**bgp_peering**(*topo*, *a*, *b*)
Register a BGP peering between two nodes

class ipmininet.router.config.**RouterConfig**(*node*, *sysctl=None*, **args*, ***kwargs*)
Bases: *ipmininet.router.config.base.NodeConfig*

compute_routerid()
Computes the default router id for all daemons. If a router ids were explicitly set for some of its daemons, the router id set to the daemon with the highest priority is chosen as the global router id. Otherwise if it has IPv4 addresses, it returns the most-visible one among its router interfaces. If both conditions are wrong, it generates a unique router id.

static incr_last_routerid()

post_register_daemons()
Method called after all daemon classes were instantiated

ipmininet.router.config.**bgp_fullmesh**(*topo*, *routers*)
Establish a full-mesh set of BGP peerings between routers

Parameters **routers** – The set of routers peering within each other

ipmininet.router.config.**ebgp_session**(*topo*, *a*, *b*, *link_type=None*)
Register an eBGP peering between two nodes, and disable IGP adjacencies between them.

Parameters

- **topo** – The current topology
- **a** – Local router
- **b** – Peer router
- **link_type** – Can be set to SHARE or CLIENT_PROVIDER. In this case ebgp_session will create import and export filter and set local pref based on the link type

class ipmininet.router.config.**CommunityList**(*name=None*, *action='permit'*, *community=0*)
Bases: object

A zebra community-list entry

Parameters

- **name** –
- **action** –
- **community** –

count = 0

ipmininet.router.config.**set_rr**(*topo*, *rr*, *peers=()*)
Set rr as route reflector for all router r

Parameters

- **topo** – The current topology
- **rr** – The route reflector
- **peers** – Clients of the route reflector

class ipmininet.router.config.**AccessList**(*name=None*, *entries=()*)

Bases: object

A zebra access-list class. It contains a set of AccessListEntry, which describes all prefix belonging or not to this ACL

Setup a new access-list

Parameters

- **name** – The name of the acl, which will default to acl## where ## is the instance number
- **entries** – A sequence of AccessListEntry instance, or of ip_interface which describes which prefixes are composing the ACL

count = 0

class ipmininet.router.config.**IPTables**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)

Bases: *ipmininet.router.config.base.Daemon*

iptables: the default Linux firewall/ACL engine for IPv4. This is currently mainly a proxy class to generate a list of static rules to pass to iptables.

As such, see *man iptables* and *man iptables-extensions* to see the various table names, commands, pre-existing chains, ...

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

NAME = 'iptables'

build()

Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

dry_run

The startup line to use to check that the daemon is well-configured

set_defaults(*defaults*)

Parameters **rules** – The (ordered) list of iptables rules that should be executed. If a rule is an iterable of strings, these will be joined using a space.

startup_line

Return the corresponding startup_line for this daemon

class ipmininet.router.config.**IP6Tables**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)

Bases: *ipmininet.router.config.iptables.IPTables*

The IPv6 counterpart to iptables ...

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

NAME = 'ip6tables'

class ipmininet.router.config.**SSHd**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)

Bases: *ipmininet.router.config.base.Daemon*

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

KILL_PATTERNS = ('None -D -u0',)

NAME = 'sshd'

STARTUP_LINE_BASE = 'None -D -u0'

build()

Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

dry_run

The startup line to use to check that the daemon is well-configured

set_defaults(*defaults*)

Update defaults to contain the defaults specific to this daemon

startup_line

Return the corresponding startup_line for this daemon

class ipmininet.router.config.**RADVD**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)

Bases: *ipmininet.router.config.base.RouterDaemon*

The class representing the radvd daemon, used for router advertisements

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

KILL_PATTERNS = ('radvd',)

NAME = 'radvd'

build()

Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

cleanup()

Cleanup the files belonging to this daemon

dry_run

The startup line to use to check that the daemon is well-configured

set_defaults(*defaults*)

Parameters **debuglevel** – Turn on debugging information. Takes an integer between 0 and 5, where 0 completely turns off debugging, and 5 is extremely verbose. (see radvd(8) for more details)

startup_line

Return the corresponding startup_line for this daemon

class ipmininet.router.config.**AdvPrefix**(*prefix=()*, *valid_lifetime=86400*, *preferred_lifetime=14400*)

Bases: *ipmininet.router.config.utils.ConfigDict*

The class representing advertised prefixes in a Router Advertisement

Parameters

- **prefix** – the list of IPv6 prefixes to advertise
- **valid_lifetime** – corresponds to the AdvValidLifetime in radvd.conf(5) for this prefix
- **preferred_lifetime** – corresponds to the AdvPreferredLifetime in radvd.conf(5) for this prefix

class ipmininet.router.config.**AdvConnectedPrefix**(*valid_lifetime=86400*, *preferred_lifetime=14400*)

Bases: *ipmininet.router.config.radvd.AdvPrefix*

This class forces the advertisement of all prefixes on the interface

Parameters

- **valid_lifetime** – corresponds to the AdvValidLifetime in radvd.conf(5) for this prefix
- **preferred_lifetime** – corresponds to the AdvPreferredLifetime in radvd.conf(5) for this prefix

class ipmininet.router.config.**AdvRDNSS**(*node*, *max_lifetime=25*)

Bases: *ipmininet.router.config.utils.ConfigDict*

The class representing an advertised DNS server in a Router Advertisement

Parameters

- **node** – Either the IPv6 address of the DNS server or the node name
- **max_lifetime** – corresponds to the AdvValidLifetime in radvd.conf(5) for this dns server address

class ipmininet.router.config.**PIMD**(*node*, **args*, ***kwargs*)

Bases: *ipmininet.router.config.zebra.QuaggaDaemon*

This class configures a PIM daemon to responds to IGMP queries in order to setup multicast routing in the network.

DEPENDS = (<class 'ipmininet.router.config.zebra.Zebra'>,)

KILL_PATTERNS = ('pimd',)

NAME = 'pimd'

build()

Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

set_defaults(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged
- **multicast_ssm** – Enable pim ssm mode by default or not
- **multicast_igmp** – Enable igmp by default or not

class ipmininet.router.config.**RIPng**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)

Bases: *ipmininet.router.config.zebra.QuaggaDaemon*

This class provides a simple configuration for an RIP daemon. It advertizes one network per interface (the primary one), and set interfaces not facing another L3Router to passive

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

DEPENDS = (<class 'ipmininet.router.config.zebra.Zebra'>,)

KILL_PATTERNS = ('ripngd',)

NAME = 'ripngd'

build()

Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

static is_active_interface(*itf*)

Return whether an interface is active or not for the OSPF daemon

set_defaults(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged (default: []).
- **redistribute** – set of RIPngRedistributedRoute sources (default: []).
- **split_horizon** – the daemon uses the split-horizon method (default: False).
- **split_horizon_with_poison** – the daemon uses the split-horizon. with reversed poison method. If both split_horizon_with_poison and split_horizon are set to True, RIPng will use the split-horizon with reversed poison method (default: True).
- **update_timer** – routing table timer value in second (default value:30).
- **timeout_timer** – routing information timeout timer (default value:180).
- **garbage_timer** – garbage collection timer (default value:120).

class ipmininet.router.config.**STATIC**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)

Bases: *ipmininet.router.config.zebra.QuaggaDaemon*

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

DEPENDS = (<class 'ipmininet.router.config.zebra.Zebra'>,)

KILL_PATTERNS = ('staticd',)

NAME = 'staticd'

build()
Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

set_defaults(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged
- **static_routes** – The set of StaticRoute to create

class ipmininet.router.config.**StaticRoute**(*prefix*, *nexthop*, *distance=1*)
Bases: object

A class representing a static route

Parameters

- **prefix** – The prefix for this static route
- **nexthop** – The nexthop for this prefix, one of: <IP address, interface name, null0, blackhole, reject>
- **distance** – The distance metric of the route

class ipmininet.router.config.**OpenrDaemon**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)
Bases: *ipmininet.router.config.base.RouterDaemon*

The base class for the OpenR daemon

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

NAME = 'openr'

STARTUP_LINE_EXTRA

build()
Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

dry_run
The OpenR dryrun runs the daemon and does not shutdown the daemon. As a workaround we only show the version of the openr daemon

set_defaults(*defaults*)

Parameters

- **logfile** – the path to the logfile for the daemon
- **routerid** – the router id for this daemon

startup_line
Return the corresponding startup_line for this daemon

class ipmininet.router.config.**Openr**(*node*, **args*, ***kwargs*)
Bases: *ipmininet.router.config.openrd.OpenrDaemon*

This class provides a simple configuration for an OpenR daemon.

DEPENDS = (<class 'ipmininet.router.config.openrd.OpenrDaemon'>,)

KILL_PATTERNS = ('openr',)

NAME = 'openr'

build()
Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

static is_active_interface(*itf*)
Return whether an interface is active or not for the OpenR daemon

set_defaults(*defaults*)
Updates some options of the OpenR daemon to run a network of routers in mininet. For a full list of parameters see OpenrDaemon:_defaults in openrd.py

class ipmininet.router.config.**OpenrDomain**(*domain*, *routers=()*, *links=()*, ***props*)
Bases: *ipmininet.overlay.Overlay*

An overlay to group OpenR links and routers by domain

Parameters

- **domain** – the domain for this overlay
- **routers** – the set of routers for which all their interfaces belong to that area
- **links** – individual links belonging to this area

apply(*topo*)
Apply the Overlay properties to the given topology

domain

ipmininet.router.config.**AF_INET**(**args*, ***kwargs*)
The ipv4 (unicast) address family

ipmininet.router.config.**AF_INET6**(**args*, ***kwargs*)
The ipv6 (unicast) address family

Submodules

ipmininet.router.config.base module

This modules provides a config object for a router, that is able to provide configurations for a set of routing daemons. It also defines the base class for a daemon, as well as a minimalistic configuration for a router.

class ipmininet.router.config.base.**BasicRouterConfig**(*node*, *daemons=()*, *additional_daemons=()*, **args*, ***kwargs*)
Bases: *ipmininet.router.config.base.RouterConfig*

A basic router that will run an OSPF daemon

A simple router made of at least an OSPF daemon

Parameters **additional_daemons** – Other daemons that should be used

class ipmininet.router.config.base.**Daemon**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)

Bases: object

This class serves as base for routing daemons

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

DEPENDS = ()

KILL_PATTERNS = ()

NAME = None

PRIO = 10

build()
Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

cfg_filename
Return the main filename in which this daemon config should be stored

cfg_filenames
Return the list of filenames in which this daemon config should be stored

cleanup()
Cleanup the files belonging to this daemon

dry_run
The startup line to use to check that the daemon is well-configured

classmethod get_config(*topo*, *router*, ***kwargs*)
Returns a config object for the daemon if any

has_started()
Return whether this daemon has started or not

options
Get the options ConfigDict for this daemon

render(*cfg*, ***kwargs*)
Render the configuration content for each config file of this daemon

Parameters

- **cfg** – The global config for the node
- **kwargs** – Additional keywords args. will be passed directly to the template

set_defaults(*defaults*)
Update defaults to contain the defaults specific to this daemon

startup_line
Return the corresponding startup_line for this daemon

template_filenames

write(*cfg*)
Write down the configuration files for this daemon

Parameters **cfg** – The configuration string

class ipmininet.router.config.base.**NodeConfig**(*node*, *daemons=()*, *sysctl=None*, **args*, ***kwargs*)
Bases: object

This class manages a set of daemons, and generates the global configuration for a node

Initialize our config builder

Parameters

- **node** – The node for which this object will build configurations
- **daemons** – an iterable of active routing daemons for this node
- **sysctl** – A dictionnary of sysctl to set for this node. By default, it enables IPv4/IPv6 forwarding on all interfaces.

build()
Build the configuration for each daemon, then write the configuration files

cleanup()
Cleanup all temporary files for the daemons

daemon(*key*)
Return the Daemon object in this config for the given key

Parameters **key** – the daemon name or a daemon class or instance

Returns the Daemon object

Raises **KeyError** – if not found

daemons

post_register_daemons()
Method called after all daemon classes were instantiated

register_daemon(*cls*, ***daemon_opts*)
Add a new daemon to this configuration

Parameters

- **cls** – Daemon class or object, or a 2-tuple (Daemon, dict)
- **daemon_opts** – Options to set on the daemons

sysctl
Return an list of all sysctl to set on this node

class ipmininet.router.config.base.**RouterConfig**(*node*, *sysctl=None*, **args*, ***kwargs*)
Bases: *ipmininet.router.config.base.NodeConfig*

compute_routerid()
Computes the default router id for all daemons. If a router ids were explicitly set for some of its daemons, the router id set to the daemon with the highest priority is chosen as the global router id. Otherwise if it has IPv4 addresses, it returns the most-visible one among its router interfaces. If both conditions are wrong, it generates a unique router id.

static incr_last_routerid()

post_register_daemons()
Method called after all daemon classes were instantiated

class ipmininet.router.config.base.**RouterDaemon**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)

Bases: *ipmininet.router.config.base.Daemon*

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

build()

Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

set_defaults(*defaults*)

Parameters

- **logfile** – the path to the logfile for the daemon
- **routerid** – the router id for this daemon

ipmininet.router.config.bgp module

Base classes to configure a BGP daemon

ipmininet.router.config.bgp.**AF_INET**(**args*, ***kwargs*)

The ipv4 (unicast) address family

ipmininet.router.config.bgp.**AF_INET6**(**args*, ***kwargs*)

The ipv6 (unicast) address family

class ipmininet.router.config.bgp.**AS**(*asn*, *routers=()*, ***props*)

Bases: *ipmininet.overlay.Overlay*

An overlay class that groups routers by AS number

Parameters

- **asn** – The number for this AS
- **routers** – an initial set of routers to add to this AS
- **props** – key-vals to set on all routers of this AS

asn

class ipmininet.router.config.bgp.**AddressFamily**(*af_name*, *redistribute=()*, *networks=()*)

Bases: object

An address family that is exchanged through BGP

class ipmininet.router.config.bgp.**BGP**(*node*, *port=179*, **args*, ***kwargs*)

Bases: *ipmininet.router.config.zebra.QuaggaDaemon*

This class provides the configuration skeletons for BGP routers.

DEPENDS = (<class 'ipmininet.router.config.zebra.Zebra'>,)

KILL_PATTERNS = ('bgpd',)

NAME = 'bgpd'

STARTUP_LINE_EXTRA
We add the port to the standard startup line

build()
Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

build_access_list()
Build and return a list of acess_filter :return:

build_community_list()
Build and return a list of community_filter

build_route_map(*neigbors*)
Build and return a list of route map for the current node

classmethod get_config(*topo*, *router*, ***kwargs*)
Returns a config object for the daemon if any

set_defaults(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged
- **address_families** – The set of AddressFamily to use

class ipmininet.router.config.bgp.**BGPConfig**(*topo*, *router*)
Bases: object

add_set_action(*peer*, *set_action*, *matching*, *direction*)
Add a 'RouteMapSetAction' to a BGP peering between two nodes

Parameters

- **peer** – The peer to which the routemap is applied
- **set_action** – The RouteMapSetAction to set
- **matching** – A list of filter, can be empty
- **direction** – direction of the route map: 'in', 'out' or 'both'

Returns self

deny(*name=None*, *from_peer=None*, *to_peer=None*, *matching=()*, *order=10*)

Deny all routes received from 'from_peer' and routes sent to 'to_peer' matching all of the access and community lists in 'matching'

Parameters

- **name** – The name of the route-map
- **from_peer** – The peer on which received routes have to have the community
- **to_peer** – The peer on which sent routes have to have the community
- **matching** – A list of AccessList and/or CommunityList
- **order** – The order in which route-maps are applied, i.e., lower order means applied before

Returns self

filter(*name=None*, *policy='deny'*, *from_peer=None*, *to_peer=None*, *matching=()*, *order=10*)

Either accept or deny all routes received from 'from_peer' and routes sent to 'to_peer' matching all of the access and community lists in 'matching'

Parameters

- **name** – The name of the route-map
- **policy** – Either 'deny' or 'permit'
- **from_peer** – The peer on which received routes have to have the community
- **to_peer** – The peer on which sent routes have to have the community
- **matching** – A list of AccessList and/or CommunityList
- **order** – The order in which route-maps are applied, i.e., lower order means applied before

Returns self

filters_to_match_cond(*filter_list*)

permit(*name=None*, *from_peer=None*, *to_peer=None*, *matching=()*, *order=10*)

Accept all routes received from 'from_peer' and routes sent to 'to_peer' matching all of the access and community lists in 'matching'

Parameters

- **name** – The name of the route-map
- **from_peer** – The peer on which received routes have to have the community
- **to_peer** – The peer on which sent routes have to have the community
- **matching** – A list of AccessList and/or CommunityList
- **order** – The order in which route-maps are applied, i.e., lower order means applied before

Returns self

set_community(*community*, *from_peer=None*, *to_peer=None*, *matching=()*)

Set community on a routes received from 'from_peer' and routes sent to 'to_peer' on routes matching all of the access and community lists in 'matching'

Parameters

- **community** – The community value to set
- **from_peer** – The peer on which received routes have to have the community
- **to_peer** – The peer on which sent routes have to have the community
- **matching** – A list of AccessList and/or CommunityList

Returns self

set_local_pref(*local_pref*, *from_peer*, *matching=()*)

Set local pref on a peering with 'from_peer' on routes matching all of the access and community lists in 'matching'

Parameters

- **local_pref** – The local pref value to set
- **from_peer** – The peer on which the local pref is applied
- **matching** – A list of AccessList and/or CommunityList

Returns self

set_med(*med*, *to_peer*, *matching=()*)

Set MED on a peering with 'to_peer' on routes matching all of the access and community lists in 'matching'

Parameters

- **med** – The local pref value to set
- **to_peer** – The peer to which the med is applied
- **matching** – A list of AccessList and/or CommunityList

Returns self

class ipmininet.router.config.bgp.**Peer**(*base*, *node*, *v6=False*)

Bases: object

A BGP peer

Parameters

- **base** – The base router that has this peer
- **node** – The actual peer

ipmininet.router.config.bgp.**bgp_fullmesh**(*topo*, *routers*)

Establish a full-mesh set of BGP peerings between routers

Parameters **routers** – The set of routers peering within each other

ipmininet.router.config.bgp.**bgp_peering**(*topo*, *a*, *b*)

Register a BGP peering between two nodes

ipmininet.router.config.bgp.**ebgp_session**(*topo*, *a*, *b*, *link_type=None*)

Register an eBGP peering between two nodes, and disable IGP adjacencies between them.

Parameters

- **topo** – The current topology
- **a** – Local router
- **b** – Peer router
- **link_type** – Can be set to SHARE or CLIENT_PROVIDER. In this case ebgp_session will create import and export filter and set local pref based on the link type

class ipmininet.router.config.bgp.**iBGPFullMesh**(*asn*, *routers=()*, ***props*)

Bases: *ipmininet.router.config.bgp.AS*

An overlay class to establish iBGP sessions in full mesh between BGP routers.

Parameters

- **asn** – The number for this AS

- **routers** – an initial set of routers to add to this AS
- **props** – key-vals to set on all routers of this AS

apply(*topo*)
Apply the Overlay properties to the given topology

ipmininet.router.config.bgp.**set_rr**(*topo*, *rr*, *peers=()*)
Set rr as route reflector for all router r

Parameters

- **topo** – The current topology
- **rr** – The route reflector
- **peers** – Clients of the route reflector

ipmininet.router.config.iptables module

This module defines IP(6)Table configuration. Due to the current (sad) state of affairs of IPv6, one is required to explicitely make two different daemon instances, one to manage iptables, one to manage ip6tables ...

class ipmininet.router.config.iptables.**IP6Tables**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)
Bases: *ipmininet.router.config.iptables.IPTables*

The IPv6 counterpart to iptables ...

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

NAME = 'ip6tables'

class ipmininet.router.config.iptables.**IPTables**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)
Bases: *ipmininet.router.config.base.Daemon*

iptables: the default Linux firewall/ACL engine for IPv4. This is currently mainly a proxy class to generate a list of static rules to pass to iptables.

As such, see *man iptables* and *man iptables-extensions* to see the various table names, commands, pre-existing chains, ...

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

NAME = 'iptables'

build()
Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

dry_run
The startup line to use to check that the daemon is well-configured

set_defaults(*defaults*)

Parameters **rules** – The (ordered) list of iptables rules that should be executed. If a rule is an iterable of strings, these will be joined using a space.

startup_line
Return the corresponding startup_line for this daemon

class ipmininet.router.config.iptables.**Rule**(**args*, ***kw*)
Bases: object

A wrapper to represent an IPTable rule

Parameters

- **args** – the rule members, which will joined by a whitespace
- **table** – Specify the table in which the rule should be installed. Defaults to filter.

ipmininet.router.config.openr module

Base classes to configure an OpenR daemon

class ipmininet.router.config.openr.**Openr**(*node*, **args*, ***kwargs*)
Bases: *ipmininet.router.config.openrd.OpenrDaemon*

This class provides a simple configuration for an OpenR daemon.

DEPENDS = (<class 'ipmininet.router.config.openrd.OpenrDaemon'>,)

KILL_PATTERNS = ('openr',)

NAME = 'openr'

build()
Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

static is_active_interface(*itf*)
Return whether an interface is active or not for the OpenR daemon

set_defaults(*defaults*)
Updates some options of the OpenR daemon to run a network of routers in mininet. For a full list of parameters see OpenrDaemon:_defaults in openrd.py

class ipmininet.router.config.openr.**OpenrDomain**(*domain*, *routers=()*, *links=()*, ***props*)
Bases: *ipmininet.overlay.Overlay*

An overlay to group OpenR links and routers by domain

Parameters

- **domain** – the domain for this overlay
- **routers** – the set of routers for which all their interfaces belong to that area
- **links** – individual links belonging to this area

apply(*topo*)
Apply the Overlay properties to the given topology

domain

class ipmininet.router.config.openr.**OpenrNetwork**(*domain*)
Bases: object

A class holding an OpenR network properties

class ipmininet.router.config.openr.**OpenrPrefixes**(*prefixes*)
Bases: object

A class representing a prefix type in OpenR

ipmininet.router.config.openrd module

class ipmininet.router.config.openrd.**OpenrDaemon**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)
Bases: *ipmininet.router.config.base.RouterDaemon*

The base class for the OpenR daemon

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

NAME = 'openr'

STARTUP_LINE_EXTRA

build()
Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

dry_run
The OpenR dryrun runs the daemon and does not shutdown the daemon. As a workaround we only show the version of the openr daemon

set_defaults(*defaults*)

Parameters

- **logfile** – the path to the logfile for the daemon
- **routerid** – the router id for this daemon

startup_line
Return the corresponding startup_line for this daemon

ipmininet.router.config.ospf module

Base classes to configure an OSPF daemon

class ipmininet.router.config.ospf.**OSPF**(*node*, **args*, ***kwargs*)
Bases: *ipmininet.router.config.zebra.QuaggaDaemon*

This class provides a simple configuration for an OSPF daemon. It advertizes one network per interface (the primary one), and set interfaces not facing another L3Router to passive

DEPENDS = (<class 'ipmininet.router.config.zebra.Zebra'>,)

KILL_PATTERNS = ('ospfd',)

NAME = 'ospfd'

build()
Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

static is_active_interface(*itf*)
Return whether an interface is active or not for the OSPF daemon

set_defaults(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged
- **dead_int** – Dead interval timer
- **hello_int** – Hello interval timer
- **priority** – priority for the interface, used for DR election
- **redistribute** – set of OSPFRedistributedRoute sources

class ipmininet.router.config.ospf.**OSPFArea**(*area*, *routers=()*, *links=()*, ***props*)
Bases: *ipmininet.overlay.Overlay*

An overlay to group OSPF links and routers by area

Parameters

- **area** – the area for this overlay
- **routers** – the set of routers for which all their interfaces belong to that area
- **links** – individual links belonging to this area

apply(*topo*)
Apply the Overlay properties to the given topology

area

class ipmininet.router.config.ospf.**OSPFNetwork**(*domain*, *area*)
Bases: object

A class holding an OSPF network properties

class ipmininet.router.config.ospf.**OSPFRedistributedRoute**(*subtype*, *metric_type=1*, *metric=1000*)
Bases: object

A class representing a redistributed route type in OSPF

ipmininet.router.config.ospf6 module

Base classes to configure an OSPF6 daemon

class ipmininet.router.config.ospf6.**OSPF6**(*node*, **args*, ***kwargs*)
Bases: *ipmininet.router.config.ospf.OSPF*

This class provides a simple configuration for an OSPF6 daemon. It advertizes one network per interface (the primary one), and set interfaces not facing another L3Router to passive

DEAD_INT = 3

KILL_PATTERNS = ('ospf6d',)

NAME = 'ospf6d'

set_defaults(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged
- **dead_int** – Dead interval timer
- **hello_int** – Hello interval timer
- **priority** – priority for the interface, used for DR election
- **redistribute** – set of OSPFRedistributedRoute sources
- **instance_id** – the number of the attached OSPF instance

class ipmininet.router.config.ospf6.**OSPF6RedistributedRoute**(*subtype*, *metric_type=1*, *metric=1000*)

Bases: *ipmininet.router.config.ospf.OSPFRedistributedRoute*

A class representing a redistributed route type in OSPF6

ipmininet.router.config.pimd module

class ipmininet.router.config.pimd.**PIMD**(*node*, **args*, ***kwargs*)

Bases: *ipmininet.router.config.zebra.QuaggaDaemon*

This class configures a PIM daemon to responds to IGMP queries in order to setup multicast routing in the network.

DEPENDS = (<class 'ipmininet.router.config.zebra.Zebra'>,)

KILL_PATTERNS = ('pimd',)

NAME = 'pimd'

build()

Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

set_defaults(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged
- **multicast_ssm** – Enable pim ssm mode by default or not
- **multicast_igmp** – Enable igmp by default or not

ipmininet.router.config.radvd module

class ipmininet.router.config.radvd.**AdvConnectedPrefix**(*valid_lifetime=86400*, *preferred_lifetime=14400*)

Bases: *ipmininet.router.config.radvd.AdvPrefix*

This class forces the advertisement of all prefixes on the interface

Parameters

- **valid_lifetime** – corresponds to the AdvValidLifetime in radvd.conf(5) for this prefix
- **preferred_lifetime** – corresponds to the AdvPreferredLifetime in radvd.conf(5) for this prefix

class ipmininet.router.config.radvd.**AdvPrefix**(*prefix=()*, *valid_lifetime=86400*, *preferred_lifetime=14400*)

Bases: *ipmininet.router.config.utils.ConfigDict*

The class representing advertised prefixes in a Router Advertisement

Parameters

- **prefix** – the list of IPv6 prefixes to advertise
- **valid_lifetime** – corresponds to the AdvValidLifetime in radvd.conf(5) for this prefix
- **preferred_lifetime** – corresponds to the AdvPreferredLifetime in radvd.conf(5) for this prefix

class ipmininet.router.config.radvd.**AdvRDNSS**(*node*, *max_lifetime=25*)

Bases: *ipmininet.router.config.utils.ConfigDict*

The class representing an advertised DNS server in a Router Advertisement

Parameters

- **node** – Either the IPv6 address of the DNS server or the node name
- **max_lifetime** – corresponds to the AdvValidLifetime in radvd.conf(5) for this dns server address

class ipmininet.router.config.radvd.**RADVD**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)

Bases: *ipmininet.router.config.base.RouterDaemon*

The class representing the radvd daemon, used for router advertisements

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

KILL_PATTERNS = ('radvd',)

NAME = 'radvd'

build()

Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

cleanup()

Cleanup the files belonging to this daemon

dry_run

The startup line to use to check that the daemon is well-configured

set_defaults(*defaults*)

Parameters **debuglevel** – Turn on debugging information. Takes an integer between 0 and 5, where 0 completely turns off debugging, and 5 is extremely verbose. (see radvd(8) for more details)

startup_line
Return the corresponding startup_line for this daemon

ipmininet.router.config.ripng module

Base classes to configure a RIP daemon

class ipmininet.router.config.ripng.**RIPNetwork**(*domain*)
Bases: object

A class holding an RIP network properties

class ipmininet.router.config.ripng.**RIPRedistributedRoute**(*subtype*, *metric=1000*)
Bases: object

A class representing a redistributed route type in RIP

class ipmininet.router.config.ripng.**RIPng**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)
Bases: *ipmininet.router.config.zebra.QuaggaDaemon*

This class provides a simple configuration for an RIP daemon. It advertizes one network per interface (the primary one), and set interfaces not facing another L3Router to passive

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

DEPENDS = (<class 'ipmininet.router.config.zebra.Zebra'>,)

KILL_PATTERNS = ('ripngd',)

NAME = 'ripngd'

build()
Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

static is_active_interface(*itf*)
Return whether an interface is active or not for the OSPF daemon

set_defaults(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged (default: []).
- **redistribute** – set of RIPngRedistributedRoute sources (default: []).
- **split_horizon** – the daemon uses the split-horizon method (default: False).
- **split_horizon_with_poison** – the daemon uses the split-horizon. with reversed poison method. If both split_horizon_with_poison and split_horizon are set to True, RIPng will use the split-horizon with reversed poison method (default: True).
- **update_timer** – routing table timer value in second (default value:30).

- **timeout_timer** – routing information timeout timer (default value:180).
- **garbage_timer** – garbage collection timer (default value:120).

ipmininet.router.config.sshd module

This module defines an sshd configuration.

class ipmininet.router.config.sshd.**SSHd**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)

Bases: *ipmininet.router.config.base.Daemon*

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

KILL_PATTERNS = ('None -D -u0',)

NAME = 'sshd'

STARTUP_LINE_BASE = 'None -D -u0'

build()

Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

dry_run

The startup line to use to check that the daemon is well-configured

set_defaults(*defaults*)

Update defaults to contain the defaults specific to this daemon

startup_line

Return the corresponding startup_line for this daemon

ipmininet.router.config.staticd module

class ipmininet.router.config.staticd.**STATIC**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)

Bases: *ipmininet.router.config.zebra.QuaggaDaemon*

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

DEPENDS = (<class 'ipmininet.router.config.zebra.Zebra'>,)

KILL_PATTERNS = ('staticd',)

NAME = 'staticd'

build()

Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

set_defaults(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged
- **static_routes** – The set of StaticRoute to create

class ipmininet.router.config.staticd.**StaticRoute**(*prefix*, *nexthop*, *distance=1*)
Bases: object

A class representing a static route

Parameters

- **prefix** – The prefix for this static route
- **nexthop** – The nexthop for this prefix, one of: <IP address, interface name, null0, blackhole, reject>
- **distance** – The distance metric of the route

ipmininet.router.config.utils module

This modules contains various utilies to streamline config generation

class ipmininet.router.config.utils.**ConfigDict**(***kwargs*)
Bases: dict

A dictionary whose attributes are its keys. Be careful if subclassing, as attributes defined by doing assignments such as self.xx = yy in __init__ will be shadowed!

ipmininet.router.config.utils.**ip_statement**(*ip*)
Return the zebra ip statement for a given ip prefix

ipmininet.router.config.zebra module

class ipmininet.router.config.zebra.**AccessList**(*name=None*, *entries=()*)
Bases: object

A zebra access-list class. It contains a set of AccessListEntry, which describes all prefix belonging or not to this ACL

Setup a new access-list

Parameters

- **name** – The name of the acl, which will default to acl## where ## is the instance number
- **entries** – A sequence of AccessListEntry instance, or of ip_interface which describes which prefixes are composing the ACL

count = 0

class ipmininet.router.config.zebra.**AccessListEntry**(*prefix*, *action='permit'*)
Bases: object

A zebra access-list entry

Parameters

- **prefix** – The ip_interface prefix for that ACL entry
- **action** – Wether that prefix belongs to the ACL (PERMIT) or not (DENY)

class ipmininet.router.config.zebra.**CommunityList**(*name=None*, *action='permit'*, *community=0*)

Bases: object

A zebra community-list entry

Parameters

- **name** –
- **action** –
- **community** –

count = 0

class ipmininet.router.config.zebra.**QuaggaDaemon**(*node*, *template_lookup=<mako.lookup.TemplateLookup object>*, ***kwargs*)

Bases: *ipmininet.router.config.base.RouterDaemon*

The base class for all Quagga-derived daemons

Parameters

- **node** – The node for which we build the config
- **template_lookup** – The TemplateLookup object of the template directory
- **kwargs** – Pre-set options for the daemon, see defaults()

STARTUP_LINE_EXTRA = ''

build()

Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

dry_run

The startup line to use to check that the daemon is well-configured

set_defaults(*defaults*)

Parameters **debug** – the set of debug events that should be logged

startup_line

Return the corresponding startup_line for this daemon

zebra_socket

Return the path towards the zebra API socket for the given node

class ipmininet.router.config.zebra.**RouteMap**(*name=None*, *match_policy='permit'*, *match_cond=()*, *set_actions=()*, *call_action=None*, *exit_policy=None*, *order=10*, *proto=()*, *neighbor=()*, *direction='in'*)

Bases: object

A class representing a set of route maps applied to a given protocol

Parameters

- **name** – The name of the route-map, defaulting to rm##
- **match_policy** – Deny or permit the actions if the route match the condition
- **match_cond** – Specify one or more conditions which must be matched if the entry is to be considered further
- **set_actions** – Specify one or more actions to do if there is a match
- **call_action** – call to an other route map
- **exit_policy** – An entry may, optionally specify an alternative exit policy if the entry matched or of (action, [acl, acl, ...]) tuples that will compose the route map
- **order** – Priority of the route map compare to others
- **proto** – The set of protocols to which this route-map applies
- **neighbor** – List of peers this route map is applied to
- **direction** – Direction of the routemap(in, out, both)

append_match_cond(*match_conditions*)

Returns

append_set_action(*set_actions*)

Parameters **set_actions** –

Returns

count = 0

describe

Return the zebra description of this route map and apply it to the relevant protocols

class ipmininet.router.config.zebra.**RouteMapMatchCond**(*cond_type*, *condition*)

Bases: object

A class representing a RouteMap matching condition

Parameters

- **condition** – Can be an ip address, the id of an access or prefix list
- **cond_type** – The type of condition access list, prefix list, peer ...

class ipmininet.router.config.zebra.**RouteMapSetAction**(*action_type*, *value*)

Bases: object

A class representing a RouteMap set action

Parameters

- **action_type** – Type of value to me modified
- **value** – Value to be modified

class ipmininet.router.config.zebra.**Zebra**(**args*, ***kwargs*)

Bases: *ipmininet.router.config.zebra.QuaggaDaemon*

KILL_PATTERNS = ('zebra',)

NAME = 'zebra'

PRIO = 0

STARTUP_LINE_EXTRA = '-k'

build()
Build the configuration tree for this daemon

Returns ConfigDict-like object describing this configuration

has_started()
Return whether this daemon has started or not

listening()

set_defaults(*defaults*)

Parameters

- **debug** – the set of debug events that should be logged
- **access_lists** – The set of AccessList to create, independently from the ones already included by route_maps
- **route_maps** – The set of RouteMap to create

9.1.2 Submodules

ipmininet.clean module

ipmininet.clean.**cleanup**(*level='info'*)
Cleanup all possible junk that we may have started.

ipmininet.clean.**killprocs**(*patterns*, *timeout=10*)
Reliably terminate processes matching a pattern (including args)

ipmininet.cli module

An enhanced CLI providing IP-related commands

class ipmininet.cli.**IPCLI**(*mininet*, *stdin=<_io.TextIOWrapper name='<stdin>' mode='r' encoding='UTF-8'>*, *script=None*)
Bases: mininet.cli.CLI

Start and run interactive or batch mode CLI mininet: Mininet network object stdin: standard input for CLI script: script to run in batch mode

default(*line*)
Called on an input line when the command prefix is not recognized. Overridden to run shell commands when a node is the first CLI argument. Past the first CLI argument, node names are automatically replaced with corresponding addresses if possible. We select only one IP version for these automatic replacements. The chosen IP version chosen is first restricted by the addresses available on the first node. Then, we choose the IP version that enables every replacement. We use IPv4 as a tie-break.

do_ip(*line*)
ip IP1 IP2 . . . : return the node associated to the given IP

do_ips(*line*)
ips n1 n2 . . . : return the ips associated to the given node name

do_ping4all(*line*)
Ping (IPv4-only) between all hosts.

do_ping4pair(*_line*)
Ping (IPv4-only) between first two hosts, useful for testing.

do_ping6all(*line*)
Ping (IPv4-only) between all hosts.

do_ping6pair(*_line*)
Ping (IPv6-only) between first two hosts, useful for testing.

do_route(*line=''*)
route destination: Print all the routes towards that destination for every router in the network

ipmininet.ipnet module

IPNet: The Mininet that plays nice with IP networks. This modules will auto-generate all needed configuration properties if unspecified by the user

class ipmininet.ipnet.**BroadcastDomain**(*interfaces=None*, **args*, ***kwargs*)
Bases: object

An IP broadcast domain in the network. This class stores the set of interfaces belonging to the same broadcast domain, as well as the associated IP prefix if any

Initialize the broadcast domain and optionally explore a set of interfaces

Parameters **interfaces** – one Intf or a list of Intf

BOUNDARIES = (<class 'mininet.node.Host'>, <class 'ipmininet.host.__host.IPHost'>, <cl

explore(*itfs*)
Explore a new list of interfaces and add them and their neightbors to this broadcast domain

Parameters **itf** – a list of Intf

static is_domain_boundary(*node*)
Check whether the node is a L3 broadcast domain boundary

Parameters **node** – a Node instance

len_v4()
The number of IPv4 addresses in this broadcast domain

len_v6()
The number of IPv6 addresses in this broadcast domain

max_v4prefixlen
Return the maximal IPv4 prefix suitable for this domain

max_v6prefixlen
Return the maximal IPv6 prefix suitable for this domain

next_ipv4()
Allocate and return the next available IPv4 address in this domain

Return ip_interface

next_ipv6()
Allocate and return the next available IPv6 address in this domain

Return ip_interface

routers
List all interfaces in this domain belonging to a L3 router

use_ip_version(*ip_version*)
Checks whether it makes sense to allocate a subnet of an IP version to this domain. If there is no other node allowing it, there is no point in allocating an address to a single host.

Parameters **ip_version** – either 4 or 6

Returns True iif there is more than one interface on the domain enabling this IP version

class ipmininet.ipnet.**IPNet**(*router=<class 'ipmininet.router.__router.Router'>*, *config=<class 'ipmininet.router.config.base.BasicRouterConfig'>*, *use_v4=True*, *ipBase='192.168.0.0/16'*, *max_v4_prefixlen=24*, *use_v6=True*, *ip6Base='fc00::/7'*, *allocate_IPs=True*, *max_v6_prefixlen=48*, *igp_metric=1*, *igp_area='0.0.0.0'*, *host=<class 'ipmininet.host.__host.IPHost'>*, *link=<class 'ipmininet.link.IPLink'>*, *intf=<class 'ipmininet.link.IPIntf'>*, *switch=<class 'ipmininet.ipswitch.IPSwitch'>*, *controller=None*, **args*, ***kwargs*)

Bases: mininet.net.Mininet

IPNet: An IP-aware Mininet

Extends Mininet by adding IP-related ivars/functions and configuration knobs.

Parameters

- **router** – The class to use to build routers
- **config** – The default configuration for the routers
- **use_v4** – Enable IPv4
- **max_v4_prefixlen** – The maximal IPv4 prefix for the auto-allocated broadcast domains
- **use_v6** – Enable IPv6
- **ip6Base** – Base prefix to use for IPv6 allocations
- **max_v6_prefixlen** – Maximal IPv6 prefixlen to auto-allocate
- **allocate_IPs** – wether to auto-allocate subnets in the network
- **igp_metric** – The default IGP metric for the links
- **igp_area** – The default IGP area for the links

addHost(*name*, ***params*)

Prevent Mininet from forcing the allocation of IPv4 addresses on hosts. We delegate it to the address auto-allocation of IPNet.

addLink(*node1*, *node2*, *igp_metric=None*, *igp_area=None*, *igp_passive=False*, *v4_width=1*, *v6_width=1*, **args*, ***params*)

Register a link with additional properties

Parameters

- **igp_metric** – the associated igp metric for this link
- **igp_area** – the associated igp area for this link
- **igp_passive** – whether IGP should create adjacencies over this link or not
- **v4_width** – the number of IPv4 addresses to allocate on the interfaces
- **v6_width** – the number of IPv6 addresses to allocate on the interfaces
- **ra** – list of AdvPrefix objects, each one representing an advertised prefix
- **rdnss** – list of AdvRDNSS objects, each one representing an advertised DNS server

addRouter(*name*, *cls=None*, ***params*)
Add a router to the network

Parameters

- **name** – the node name
- **cls** – the class to use to instantiate it

build()
Build mininet.

buildFromTopo(*topo*)
Build mininet from a topology object At the end of this function, everything should be connected and up.

node_for_ip(*ip*)
Return the node owning a given IP address

Parameters **ip** – an IP address

Returns a node name

ping(*hosts=None*, *timeout=None*, *use_v4=True*, *use_v6=True*)
Ping between all specified hosts. If use_v4 is true, pings over IPv4 are used between any pair of hosts having at least one IPv4 address on one of their interfaces (loopback excluded). If use_v6 is true, pings over IPv6 are used between any pair of hosts having at least one non-link-local IPv6 address on one of their interfaces (loopback excluded).

Parameters

- **hosts** – list of hosts or None if all must be pinged
- **timeout** – time to wait for a response, as string
- **use_v4** – whether IPv4 addresses can be used
- **use_v6** – whether IPv6 addresses can be used

Returns the packet loss percentage of IPv4 connectivity if self.use_v4 is set the loss percentage of IPv6 connectivity otherwise

ping4All(*timeout=None*)
Ping (IPv4-only) between all hosts. return: ploss packet loss percentage

ping4Pair()
Ping (IPv4-only) between first two hosts, useful for testing. return: ploss packet loss percentage

ping6All(*timeout=None*)
Ping (IPv6-only) between all hosts. return: ploss packet loss percentage

ping6Pair()
Ping (IPv6-only) between first two hosts, useful for testing. return: ploss packet loss percentage

pingAll(*timeout=None*, *use_v4=True*, *use_v6=True*)
Ping between all hosts. return: ploss packet loss percentage

pingPair(*use_v4=True*, *use_v6=True*)
Ping between first two hosts, useful for testing. return: ploss packet loss percentage

start()
Start controller and switches.

stop()
Stop the controller(s), switches and hosts

ipmininet.ipswitch module

This modules defines the IPSwitch class allowing to better support STP and to create hubs

class ipmininet.ipswitch.**IPSwitch**(*name*, *stp=True*, *hub=False*, *prio=None*, ***kwargs*)
Bases: mininet.nodelib.LinuxBridge

Linux Bridge (with optional spanning tree) extended to include the hubs

Parameters

- **name** – the name of the node
- **stp** – whether to use spanning tree protocol
- **hub** – whether this switch behaves as a hub (this disable stp)
- **prio** – optional explicit bridge priority for STP

start(*_controllers*)
Start Linux bridge

ipmininet.iptopo module

This module defines topology class that supports adding L3 routers

class ipmininet.iptopo.**HostDescription**(*o*, *topo=None*)
Bases: *ipmininet.iptopo.NodeDescription*

addDaemon(*daemon*, *default_cfg_class=<class 'ipmininet.host.config.base.HostConfig'>*, ***kwargs*)
Add the daemon to the list of daemons to start on the node.

Parameters

- **daemon** – daemon class
- **default_cfg_class** – config class to use if there is no configuration class defined for the router yet.
- **cfg_daemon_list** – name of the parameter containing the list of daemons in your config class constructor. For instance, RouterConfig uses 'daemons' but BasicRouterConfig uses 'additional_daemons'.
- **daemon_params** – all the parameters to give when instantiating the daemon class.

class ipmininet.iptopo.**IPTopo**(**args*, ***kwargs*)
Bases: mininet.topo.Topo

A topology that supports L3 routers

OVERLAYS = {'AS': <class 'ipmininet.router.config.bgp.AS'>, 'DNSZone': <class 'ipmini

addDaemon(*node*, *daemon*, *default_cfg_class=<class 'ipmininet.router.config.base.BasicRouterConfig'>*, *cfg_daemon_list='daemons'*, ***daemon_params*)
Add the daemon to the list of daemons to start on the router.

Parameters

- **node** – node name
- **daemon** – daemon class
- **default_cfg_class** – config class to use if there is no configuration class defined for the router yet.

- **cfg_daemon_list** – name of the parameter containing the list of daemons in your config class constructor. For instance, RouterConfig uses 'daemons' but BasicRouterConfig uses 'additional_daemons'.
- **daemon_params** – all the parameters to give when instantiating the daemon class.

addHost(*name*, ***kwargs*)
Add a host to the topology

Parameters **name** – the name of the node

addHub(*name*, ***opts*)
Convenience method: Add hub to graph. name: hub name opts: hub options returns: hub name

addLink(*node1*, *node2*, *port1=None*, *port2=None*, *key=None*, ***opts*)

Parameters

- **node1** – first node to link
- **node2** – second node to link
- **port1** – port of the first node (optional)
- **port2** – port of the second node (optional)
- **key** – a key to identify the link (optional)
- **opts** – link options (optional)

Returns link info key

addOverlay(*overlay*)
Add a new overlay on this topology

addRouter(*name*, ***kwargs*)
Add a router to the topology

Parameters **name** – the name of the node

build(**args*, ***kwargs*)
Override this method to build your topology.

capture_physical_interface(*intfname*, *node*)
Adds a pre-existing physical interface to the given node.

getLinkInfo(*l*, *key*, *default*)
Attempt to retrieve the information for the given link/key combination. If not found, set to an instance of default and return it

getNodeInfo(*n*, *key*, *default*)
Attempt to retrieve the information for the given node/key combination. If not found, set to an instance of default and return it

hosts(*sort=True*)
Return hosts. sort: sort hosts alphabetically returns: list of hosts

hubs(*sort=True*)
Return a list of hub node names

isHub(*n*)
Check whether the given node is a router

Parameters **n** – node name

isNodeType(*n*, *x*)
Return wether node n has a key x set to True

Parameters

- **n** – node name
- **x** – the key to check

isRouter(*n*)
Check whether the given node is a router

Parameters **n** – node name

post_build(*net*)
A method that will be invoced once the topology has been fully built and before it is started.

Parameters **net** – The freshly built (Mininet) network

routers(*sort=True*)
Return a list of router node names

class ipmininet.iptopo.**IntfDescription**(*o*, *topo*, *link*, *intf_attrs*)
Bases: *ipmininet.iptopo.NodeDescription*

addParams(***kwargs*)

class ipmininet.iptopo.**LinkDescription**(*topo*, *src*, *dst*, *key*, *link_attrs*)
Bases: object

class ipmininet.iptopo.**NodeDescription**(*o*, *topo=None*)
Bases: str

addDaemon(*daemon*, *default_cfg_class=<class 'ipmininet.router.config.base.BasicRouterConfig'>*, *cfg_daemon_list='daemons'*, ***daemon_params*)
Add the daemon to the list of daemons to start on the node.

Parameters

- **daemon** – daemon class
- **default_cfg_class** – config class to use if there is no configuration class defined for the router yet.
- **cfg_daemon_list** – name of the parameter containing the list of daemons in your config class constructor. For instance, RouterConfig uses 'daemons' but BasicRouterConfig uses 'additional_daemons'.
- **daemon_params** – all the parameters to give when instantiating the daemon class.

get_config(*daemon*, ***kwargs*)

class ipmininet.iptopo.**OverlayWrapper**(*topo*, *overlay*)
Bases: object

class ipmininet.iptopo.**RouterDescription**(*o*, *topo=None*)
Bases: *ipmininet.iptopo.NodeDescription*

addDaemon(*daemon*, *default_cfg_class=<class 'ipmininet.router.config.base.BasicRouterConfig'>*, ***kwargs*)
Add the daemon to the list of daemons to start on the node.

Parameters

- **daemon** – daemon class

- **default_cfg_class** – config class to use if there is no configuration class defined for the router yet.
- **cfg_daemon_list** – name of the parameter containing the list of daemons in your config class constructor. For instance, RouterConfig uses 'daemons' but BasicRouterConfig uses 'additional_daemons'.
- **daemon_params** – all the parameters to give when instantiating the daemon class.

ipmininet.link module

Classes for interfaces and links that are IP-agnostic. This basically enhance the Intf class from Mininet, and then define sane defaults for the link classes and a new TCIntf base.

class ipmininet.link.**GRETunnel**(*if1*, *if2*, *if1address*, *if2address=None*, *bidirectional=True*)
Bases: object

The GRETunnel class, which enables to create a GRE Tunnel in a network linking two existing interfaces.

Currently, these tunnels only define stretched IP subnets.

The instantiation of these tunnels should happen *after* the network has been built and *before* the network has been started. You can leverage the IPTopo.post_build method to do it.

Parameters

- **if1** – The first interface of the tunnel
- **if2** – The second interface of the tunnel
- **if1address** – The ip_interface address for if1
- **if2address** – The ip_interface address for if2
- **bidirectional** – Whether both end of the tunnel should be established or not. GRE is stateless so there is no handshake per-say, however if one end of the tunnel is not established, the kernel will drop by defualt the encapsualted packets.

cleanup()

setup_tunnel()

class ipmininet.link.**IPIntf**(**args*, ***kwargs*)
Bases: mininet.link.Intf

This class represents a node interface. It is IP-agnostic, as in its *addresses* attribute is a dictionnary keyed by IP version, containing the list of all addresses for a given version

describe
Return a string describing the interface facing this one

get(*key*, *val*)
Check for a given key in the interface parameters

igp_area
Return the igp area associated to this interface

igp_metric
Return the igp metric associated to this interface

interface_width
Return the number of addresses that should be allocated to this interface, per address family

ip

ip6
Return the default IPv6 for this interface

ip6s(*exclude_lls=False*, *exclude_lbs=True*)
Return a generator over all IPv6 assigned to this interface

Parameters

- **exclude_lls** – Whether Link-locals should be included or not
- **exclude_lbs** – Whether Loopback addresses should be included or not

ips(*exclude_lbs=True*)
Return a generator over all IPv4 assigned to this interface

Parameters **exclude_lbs** – Whether Loopback addresses should be included or not

prefixLen

prefixLen6
Return the prefix length for the default IPv6 for this interface

setIP(*ip*, *prefixLen=None*)
Set one or more IP addresses, possibly from different families. This will remove previously set addresses of the affected families.

Parameters

- **ip** – either an IP string (mininet-like behavior), or an ip_interface like, or a sequence of both
- **prefixLen** – the prefix length to use for all cases where the addresses is given as a string without a given prefix.

setIP6(*ip*, *prefixLen=None*)
Set one or more IP addresses, possibly from different families. This will remove previously set addresses of the affected families.

Parameters

- **ip** – either an IP string (mininet-like behavior), or an ip_interface like, or a sequence of both
- **prefixLen** – the prefix length to use for all cases where the addresses is given as a string without a given prefix.

updateAddr()
Return IP address and MAC address based on ifconfig.

updateIP()
Return updated IP address based on ifconfig

updateIP6()

updateMAC()
Return updated MAC address based on ifconfig

class ipmininet.link.**IPLink**(*node1*, *node2*, *intf=<class 'ipmininet.link.IPIntf'>*, **args*, ***kwargs*)
Bases: mininet.link.Link

A Link class that defaults to IPIntf

We override Link intf default to use IPIntf

class ipmininet.link.**OrderedAddress**(*addr*)
Bases: object

class ipmininet.link.**PhysicalInterface**(*name*, **args*, ***kw*)
Bases: *ipmininet.link.IPIntf*

An interface that will wrap around an existing (physical) interface, and try to preserve its addresses. The interface must be present in the root namespace.

ipmininet.link.**address_comparator**(*a*, *b*)
Return -1, 0, 1 if a is less, equally, more visible than b. We define visibility according to IP version, address scope, address class, and address value

ipmininet.overlay module

class ipmininet.overlay.**Overlay**(*nodes=()*, *links=()*, *nprops=None*, *lprops=None*)
Bases: object

This overlay simply defines groups of nodes and links, and properties that are common to all of them. It then registers these properties to the element when apply() is called.

Elements are referenced in the same way than for the IPTopo: node -> node name link -> (node1 name, node2 name).

Parameters

- **nodes** – The nodes in this overlay
- **links** – the links in this overlay
- **nprops** – the properties shared by all nodes in this overlay
- **lprops** – the properties shared by all links in this overlay

add_link(**link*)
Add one or more link to this overlay

add_node(**node*)
Add one or more nodes to this overlay

apply(*topo*)
Apply the Overlay properties to the given topology

check_consistency(*topo*)
Check that this overlay is consistent

link_property(*l*)
Return the properties for the given link

node_property(*n*)
Return the properties for the given node

set_link_property(*n*, *key*, *val*)
Set the property of a given link

set_node_property(*n*, *key*, *val*)
Set the property of a given node

class ipmininet.overlay.**Subnet**(*nodes=()*, *links=()*, *subnets=()*)
Bases: *ipmininet.overlay.Overlay*

This overlay simply defines groups of routers and hosts that share a common set of subnets. These routers and hosts have to be on the same LAN.

Parameters

- **nodes** – The routers and hosts that needs an address on their common LAN.
- **links** – The links that has to be in the LAN. This parameter is useful to identify LANs if there is more than one common LAN between the nodes. Routers and Hosts of the links will have an address assigned.
- **subnets** – For each subnet, an address will be added to the interface of the nodes in their common LAN.

apply(*topo*)
Apply the Overlay properties to the given topology

check_consistency(*topo*)
Check that this overlay is consistent

ipmininet.topologydb module

This module defines a data-store to help dealing with all (possibly) auto-allocated properties of a topology: ip addresses, router ids, ...

class ipmininet.topologydb.**TopologyDB**(*db=None*, *net=None*, **args*, ***kwargs*)
Bases: object

A convenience store for auto-allocated mininet properties. This is *NOT* to be used as IGP graph as it does not reflect the actual availability of a node in the network (as-in it is a static view of the network).

Either extract properties from a network or load a save file

Parameters

- **db** – a path towards a saved version of this class which will be loaded
- **net** – an IPNet instance which will be parsed in order to extract useful properties

add_host(*n*)
Register an host

Parameters **n** – Host instance

add_router(*n*)
Register an router

Parameters **n** – Router instance

add_switch(*n*)
Register an switch

Parameters **n** – Switch instance

interface(*x*, *y*)
Return the ip address of the interface of x facing y

Parameters

- **x** – the node from which we want an IP address
- **y** – the node on the other side of the link

Returns ip_interface-like object

interface_bandwidth(*x*, *y*)
Return the bandwidth capacity of the interface on node x facing node y.

Parameters

- **x** – node name
- **y** – node name

Returns The bandwidth of link x-y, -1 if unlimited

interfaces(*x*)
Return the list of interface names of node x

load(*fpath*)
Load a topology database

Parameters **fpath** – path towards the file to load

node(*x*)

parse_net(*net*)
Stores the content of the given network

Parameters **net** – IPNet instance

routerid(*x*)
Return the router id of a node

Parameters **x** – router name

Returns the routerid

save(*fpath*)
Save the topology database

Parameters **fpath** – the save file name

subnet(*x*, *y*)
Return the subnet linking node x and y

Parameters

- **x** – node name
- **y** – node name

Returns ip_network-like object

ipmininet.utils module

utils: utility functions to manipulate host, interfaces, ...

class ipmininet.utils.**L3Router**
Bases: object

Placeholder class to identify L3 routing devices (primarely routers, but this could also be used for a device needing to participate to some routing protocol e.g. for TE purposes)

static is_l3router_intf(*itf*)
Returns whether an interface belongs to an L3Router (in the Mininet meaning: an intf with an associated node)

ipmininet.utils.**address_pair**(*n*, *use_v4=True*, *use_v6=True*)
Returns a tuple (ip, ip6) with ip/ip6 being one of the IPv4/IPv6 addresses of the node n

ipmininet.utils.**find_node**(*start*, *node_name*)

Parameters

- **start** – The starting node of the search
- **node_name** – The name of the node to find

Returns The interface of the node connected to start with node_name as name

ipmininet.utils.**get_set**(*d*, *key*, *default*)
Attempt to return the value for the given key, otherwise intialize it

Parameters

- **d** – dict
- **default** – constructor

ipmininet.utils.**has_cmd**(*cmd*)
Return whether the given executable is available on the system or not

ipmininet.utils.**is_container**(*x*)
Return whether x is a container (=iterable but not a string)

ipmininet.utils.**otherIntf**(*intf*)
"Get the interface on the other side of a link

ipmininet.utils.**prefix_for_netmask**(*mask*)
Return the prefix length associated to a given netmask. Will return garbage if the netmask is unproperly formatted!

ipmininet.utils.**realIntfList**(*n*)
Return the list of interfaces of node n excluding loopback

ipmininet.utils.**require_cmd**(*cmd*, *help_str=None*)
Ensures that a command is available in $PATH

Parameters

- **cmd** – the command to test
- **help_str** – an optional help string to display if cmd is not found

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

i

Index

A

B

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z