ipv6

Release Latest

Jun 28, 2019

Contents

CHAPTER 1

OPNFV IPv6 Project Release Notes

1.1 OPNFV IPv6 Project Release Notes

This document provides the release notes for Hunter of IPv6 Project.

1.1.1 Version History

Date	Version	Author	Comment
2019-03-14	0.1.0	Bin Hu	Initial draft
2019-05-04	1.0.0	Bin Hu	Hunter 8.0 Release
2019-06-28	1.1.0	Bin Hu	Hunter 8.1 Release

1.1.2 Release Data

Project	IPv6
Repo/tag	opnfv-8.1.0
Release designation	Hunter 8.1
Release date	June 28, 2019
Purpose of the delivery	OPNFV Hunter 8.1 Release

1.1.3 Important Notes

Attention: Please be aware that:

- Since Danube, Apex Installer no longer supports Open Daylight L2-only environment or `odl-ovsdb-openstack`. Instead, it supports Open Daylight L3 deployment with `odl-netvirt-openstack`.
- IPv6 features are not fully supported in Open Daylight L3 with `odl-netvirt-openstack` yet. It is still a work in progress.
- Thus we cannot realize Service VM as an IPv6 vRouter using Apex Installer under OpenStack + Open Daylight L3 with `odl-netvirt-openstack environment`.

For details, please refer to our User Guide.

1.1.4 Summary

This is the Hunter release of the IPv6 feature as part of OPNFV, including:

- Installation of OPNFV on IPv6-Only Infrastructure by Apex Installer
- Configuration of setting up a Service VM as an IPv6 vRouter in OpenStack-Only environment
- User Guide, which includes:
  - gap analysis of IPv6 support in OpenStack Rocky and OpenDaylight Fluorine
  - exploration of IPv6 in container networking

Please refer to our:

- Installation Guide
- Configuration Guide
- User Guide

1.1.5 Known Limitations, Issues and Workarounds

1.1.5.1 System Limitations

None.

1.1.5.2 Known Issues

None.

1.1.5.3 Workarounds

N/A.

1.1.6 Test Result

Please refer to Testing Methodology.

1.1.7 References

For more information on the OPNFV Hunter release, please see:

http://www.opnfv.org/software

CHAPTER 2

IPv6 Installation Procedure

Abstract

This document provides the users with the Installation Procedure to install OPNFV Hunter Release on IPv6-only Infrastructure.

2.1 Install OPNFV on IPv6-Only Infrastructure

This section provides instructions to install OPNFV on IPv6-only Infrastructure. All underlay networks and API endpoints will be IPv6-only except:

1. "admin" network in underlay/undercloud still has to be IPv4.
   - It was due to lack of support of IPMI over IPv6 or PXE over IPv6.
   - iPXE does support IPv6 now. Ironic has added support for booting nodes with IPv6.
   - We are starting to work on enabling IPv6-only environment for all networks. For TripleO, this work is still ongoing.
2. Metadata server is still IPv4 only.

Except the limitations above, the use case scenario of the IPv6-only infrastructure includes:

1. Support OPNFV deployment on an IPv6 only infrastructure.
2. Horizon/ODL-DLUX access using IPv6 address from an external host.
3. OpenStack API access using IPv6 addresses from various python-clients.
4. Ability to create Neutron Routers, IPv6 subnets (e.g. SLAAC/DHCPv6-Stateful/ DHCPv6-Stateless) to support North-South traffic.
5. Inter VM communication (East-West routing) when VMs are spread across two compute nodes.
6. VNC access into a VM using IPv6 addresses.
7. IPv6 support in OVS VxLAN (and/or GRE) tunnel endpoints with OVS 2.6+.

8. IPv6 support in iPXE, and booting nodes with IPv6 (**NEW**).

2.1.1 Install OPNFV in OpenStack-Only Environment

Apex Installer:

```
# HA, Virtual deployment in OpenStack-only environment
./opnfv-deploy -v -d /etc/opnfv-apex/os-nosdn-nofeature-ha.yaml \
-n /etc/opnfv-apex/network_settings_v6.yaml

# HA, Bare Metal deployment in OpenStack-only environment
./opnfv-deploy -d /etc/opnfv-apex/os-nosdn-nofeature-ha.yaml \
-i <inventory file> -n /etc/opnfv-apex/network_settings_v6.yaml

# Non-HA, Virtual deployment in OpenStack-only environment
./opnfv-deploy -v -d /etc/opnfv-apex/os-nosdn-nofeature-noha.yaml \
-n /etc/opnfv-apex/network_settings_v6.yaml

# Non-HA, Bare Metal deployment in OpenStack-only environment
./opnfv-deploy -d /etc/opnfv-apex/os-nosdn-nofeature-noha.yaml \
-i <inventory file> -n /etc/opnfv-apex/network_settings_v6.yaml

# Note:
#
# 1. Parameter ""-v" is mandatory for Virtual deployment
# 2. Parameter "-i <inventory file>" is mandatory for Bare Metal deployment
# 2.1 Refer to https://git.opnfv.org/cgit/apex/tree/config/inventory for examples of␣
↪inventory file
# 3. You can use "-n /etc/opnfv-apex/network_settings.yaml" for deployment in IPv4␣
↪infrastructure
```

Please **NOTE** that:

- You need to refer to **installer's documentation** for other necessary parameters applicable to your deployment.
- You need to refer to **Release Notes** and **installer's documentation** if there is any issue in installation.

2.1.2 Install OPNFV in OpenStack with ODL-L3 Environment

Apex Installer:

```
# HA, Virtual deployment in OpenStack with Open Daylight L3 environment
./opnfv-deploy -v -d /etc/opnfv-apex/os-odl-nofeature-ha.yaml \
-n /etc/opnfv-apex/network_settings_v6.yaml

# HA, Bare Metal deployment in OpenStack with Open Daylight L3 environment
./opnfv-deploy -d /etc/opnfv-apex/os-odl-nofeature-ha.yaml \
-i <inventory file> -n /etc/opnfv-apex/network_settings_v6.yaml

# Non-HA, Virtual deployment in OpenStack with Open Daylight L3 environment
./opnfv-deploy -v -d /etc/opnfv-apex/os-odl-nofeature-noha.yaml \
-n /etc/opnfv-apex/network_settings_v6.yaml

# Non-HA, Bare Metal deployment in OpenStack with Open Daylight L3 environment
./opnfv-deploy -d /etc/opnfv-apex/os-odl-nofeature-noha.yaml \
-i <inventory file> -n /etc/opnfv-apex/network_settings_v6.yaml
```

(continues on next page)

(continued from previous page)

```
# Note:
#
# 1. Parameter ""-v" is mandatory for Virtual deployment
# 2. Parameter "-i <inventory file>" is mandatory for Bare Metal deployment
# 2.1 Refer to https://git.opnfv.org/cgit/apex/tree/config/inventory for examples of␣
↪inventory file
# 3. You can use "-n /etc/opnfv-apex/network_settings.yaml" for deployment in IPv4␣
↪infrastructure
```

Please **NOTE** that:

- You need to refer to **installer's documentation** for other necessary parameters applicable to your deployment.
- You need to refer to **Release Notes** and **installer's documentation** if there is any issue in installation.

2.1.3 Testing Methodology

There are 2 levels of testing to validate the deployment.

2.1.3.1 Underlay Testing for OpenStack API Endpoints

Underlay Testing is to validate that API endpoints are listening on IPv6 addresses. Currently, we are only considering the **Underlay Testing** for OpenStack API endpoints. The **Underlay Testing** for Open Daylight API endpoints is for future release.

The **Underlay Testing** for OpenStack API endpoints can be as simple as validating Keystone service, and as complete as validating each API endpoint. It is important to reuse Tempest API testing. Currently:

- Apex Installer will change `OS_AUTH_URL` in `overcloudrc` during installation process. For example: `export OS_AUTH_URL=http://[2001:db8::15]:5000/v2.0`. `OS_AUTH_URL` points to Keystone and Keystone catalog.
- When FuncTest runs Tempest for the first time, the `OS_AUTH_URL` is taken from the environment and placed automatically in `Tempest.conf`.
- Under this circumstance, `openstack catalog list` will return IPv6 URL endpoints for all the services in catalog, including Nova, Neutron, etc, and covering public URLs, private URLs and admin URLs.
- Thus, as long as the IPv6 URL is given in the `overclourc`, all the tests will use that (including Tempest).

Therefore Tempest API testing is reused to validate API endpoints are listening on IPv6 addresses as stated above. They are part of OpenStack default Smoke Tests, run in FuncTest and integrated into OPNFV's CI/CD environment.

2.1.3.2 Overlay Testing

Overlay Testing is to validate that IPv6 is supported in tenant networks, subnets and routers. Both Tempest API testing and Tempest Scenario testing are used in our Overlay Testing.

Tempest API testing validates that the Neutron API supports the creation of IPv6 networks, subnets, routers, etc:

```
tempest.api.network.test_networks.BulkNetworkOpsIpV6Test.test_bulk_create_delete_
↪network
tempest.api.network.test_networks.BulkNetworkOpsIpV6Test.test_bulk_create_delete_port
tempest.api.network.test_networks.BulkNetworkOpsIpV6Test.test_bulk_create_delete_
↪subnet
```

(continues on next page)

(continued from previous page)

```
tempest.api.network.test_networks.NetworksIpV6Test.test_create_update_delete_network_
↪subnet
tempest.api.network.test_networks.NetworksIpV6Test.test_external_network_visibility
tempest.api.network.test_networks.NetworksIpV6Test.test_list_networks
tempest.api.network.test_networks.NetworksIpV6Test.test_list_subnets
tempest.api.network.test_networks.NetworksIpV6Test.test_show_network
tempest.api.network.test_networks.NetworksIpV6Test.test_show_subnet
tempest.api.network.test_networks.NetworksIpV6TestAttrs.test_create_update_delete_
↪network_subnet
tempest.api.network.test_networks.NetworksIpV6TestAttrs.test_external_network_
↪visibility
tempest.api.network.test_networks.NetworksIpV6TestAttrs.test_list_networks
tempest.api.network.test_networks.NetworksIpV6TestAttrs.test_list_subnets
tempest.api.network.test_networks.NetworksIpV6TestAttrs.test_show_network
tempest.api.network.test_networks.NetworksIpV6TestAttrs.test_show_subnet
tempest.api.network.test_ports.PortsIpV6TestJSON.test_create_port_in_allowed_
↪allocation_pools
tempest.api.network.test_ports.PortsIpV6TestJSON.test_create_port_with_no_
↪securitygroups
tempest.api.network.test_ports.PortsIpV6TestJSON.test_create_update_delete_port
tempest.api.network.test_ports.PortsIpV6TestJSON.test_list_ports
tempest.api.network.test_ports.PortsIpV6TestJSON.test_show_port
tempest.api.network.test_routers.RoutersIpV6Test.test_add_multiple_router_interfaces
tempest.api.network.test_routers.RoutersIpV6Test.test_add_remove_router_interface_
↪with_port_id
tempest.api.network.test_routers.RoutersIpV6Test.test_add_remove_router_interface_
↪with_subnet_id
tempest.api.network.test_routers.RoutersIpV6Test.test_create_show_list_update_delete_
↪router
tempest.api.network.test_security_groups.SecGroupIPv6Test.test_create_list_update_
↪show_delete_security_group
tempest.api.network.test_security_groups.SecGroupIPv6Test.test_create_show_delete_
↪security_group_rule
tempest.api.network.test_security_groups.SecGroupIPv6Test.test_list_security_groups
```

Tempest Scenario testing validates some specific overlay IPv6 scenarios (i.e. use cases) as follows:

```
tempest.scenario.test_network_v6.TestGettingAddress.test_dhcp6_stateless_from_os
tempest.scenario.test_network_v6.TestGettingAddress.test_dualnet_dhcp6_stateless_from_
↪os
tempest.scenario.test_network_v6.TestGettingAddress.test_dualnet_multi_prefix_dhcpv6_
↪stateless
tempest.scenario.test_network_v6.TestGettingAddress.test_dualnet_multi_prefix_slaac
tempest.scenario.test_network_v6.TestGettingAddress.test_dualnet_slaac_from_os
tempest.scenario.test_network_v6.TestGettingAddress.test_multi_prefix_dhcpv6_stateless
tempest.scenario.test_network_v6.TestGettingAddress.test_multi_prefix_slaac
tempest.scenario.test_network_v6.TestGettingAddress.test_slaac_from_os
```

The above Tempest API testing and Scenario testing are quite comprehensive to validate overlay IPv6 tenant networks. They are part of OpenStack default Smoke Tests, run in FuncTest and integrated into OPNFV's CI/CD environment.

CHAPTER 3

IPv6 Configuration Guide

Abstract

This document provides the users with the Configuration Guide to set up a service VM as an IPv6 vRouter using OPNFV Hunter Release.

3.1 IPv6 Configuration - Setting Up a Service VM as an IPv6 vRouter

This section provides instructions to set up a service VM as an IPv6 vRouter using OPNFV Hunter Release installers. Because Open Daylight no longer supports L2-only option, and there is only limited support of IPv6 in L3 option of Open Daylight, setup of service VM as an IPv6 vRouter is only available under pure/native OpenStack environment. The deployment model may be HA or non-HA. The infrastructure may be bare metal or virtual environment.

3.1.1 Pre-configuration Activities

The configuration will work only in OpenStack-only environment.

Depending on which installer will be used to deploy OPNFV, each environment may be deployed on bare metal or virtualized infrastructure. Each deployment may be HA or non-HA.

Refer to the previous installer configuration chapters, installations guide and release notes.

3.1.2 Setup Manual in OpenStack-Only Environment

If you intend to set up a service VM as an IPv6 vRouter in OpenStack-only environment of OPNFV Hunter Release, please **NOTE** that:

- Because the anti-spoofing rules of Security Group feature in OpenStack prevents a VM from forwarding packets, we need to disable Security Group feature in the OpenStack-only environment.
- The hostnames, IP addresses, and username are for exemplary purpose in instructions. Please change as needed to fit your environment.

- The instructions apply to both deployment model of single controller node and HA (High Availability) deployment model where multiple controller nodes are used.

3.1.2.1 Install OPNFV and Preparation

OPNFV-NATIVE-INSTALL-1: To install OpenStack-only environment of OPNFV Hunter Release:

Apex Installer:

```
# HA, Virtual deployment in OpenStack-only environment
./opnfv-deploy -v -d /etc/opnfv-apex/os-nosdn-nofeature-ha.yaml \
-n /etc/opnfv-apex/network_setting.yaml

# HA, Bare Metal deployment in OpenStack-only environment
./opnfv-deploy -d /etc/opnfv-apex/os-nosdn-nofeature-ha.yaml \
-i <inventory file> -n /etc/opnfv-apex/network_setting.yaml

# Non-HA, Virtual deployment in OpenStack-only environment
./opnfv-deploy -v -d /etc/opnfv-apex/os-nosdn-nofeature-noha.yaml \
-n /etc/opnfv-apex/network_setting.yaml

# Non-HA, Bare Metal deployment in OpenStack-only environment
./opnfv-deploy -d /etc/opnfv-apex/os-nosdn-nofeature-noha.yaml \
-i <inventory file> -n /etc/opnfv-apex/network_setting.yaml

# Note:
#
# 1. Parameter ""-v" is mandatory for Virtual deployment
# 2. Parameter "-i <inventory file>" is mandatory for Bare Metal deployment
# 2.1 Refer to https://git.opnfv.org/cgit/apex/tree/config/inventory for examples of␣
↪inventory file
# 3. You can use "-n /etc/opnfv-apex/network_setting_v6.yaml" for deployment in IPv6-
↪only infrastructure
```

Compass Installer:

```
# HA deployment in OpenStack-only environment
export ISO_URL=file://$BUILD_DIRECTORY/compass.iso
export OS_VERSION=${COMPASS_OS_VERSION}
export OPENSTACK_VERSION=${COMPASS_OPENSTACK_VERSION}
export CONFDIR=$WORKSPACE/deploy/conf/vm_environment
./deploy.sh --dha $CONFDIR/os-nosdn-nofeature-ha.yml \
--network $CONFDIR/$NODE_NAME/network.yml

# Non-HA deployment in OpenStack-only environment
# Non-HA deployment is currently not supported by Compass installer
```

Fuel Installer:

```
# HA deployment in OpenStack-only environment
# Scenario Name: os-nosdn-nofeature-ha
# Scenario Configuration File: ha_heat_ceilometer_scenario.yaml
# You can use either Scenario Name or Scenario Configuration File Name in "-s"␣
↪parameter
sudo ./deploy.sh -b <stack-config-uri> -l <lab-name> -p <pod-name> \
-s os-nosdn-nofeature-ha -i <iso-uri>
```

(continues on next page)

(continued from previous page)

```
# Non-HA deployment in OpenStack-only environment
# Scenario Name: os-nosdn-nofeature-noha
# Scenario Configuration File: no-ha_heat_ceilometer_scenario.yaml
# You can use either Scenario Name or Scenario Configuration File Name in "-s"␣
↪parameter
sudo ./deploy.sh -b <stack-config-uri> -l <lab-name> -p <pod-name> \
-s os-nosdn-nofeature-noha -i <iso-uri>

# Note:
#
# 1. Refer to http://git.opnfv.org/cgit/fuel/tree/deploy/scenario/scenario.yaml for␣
↪scenarios
# 2. Refer to http://git.opnfv.org/cgit/fuel/tree/ci/README for description of
#    stack configuration directory structure
# 3. <stack-config-uri> is the base URI of stack configuration directory structure
# 3.1 Example: http://git.opnfv.org/cgit/fuel/tree/deploy/config
# 4. <lab-name> and <pod-name> must match the directory structure in stack␣
↪configuration
# 4.1 Example of <lab-name>: -l devel-pipeline
# 4.2 Example of <pod-name>: -p elx
# 5. <iso-uri> could be local or remote ISO image of Fuel Installer
# 5.1 Example: http://artifacts.opnfv.org/fuel/euphrates/opnfv-euphrates.1.0.iso
#
# Please refer to Fuel Installer's documentation for further information and any␣
↪update
```

Joid Installer:

```
# HA deployment in OpenStack-only environment
./deploy.sh -o mitaka -s nosdn -t ha -l default -f ipv6

# Non-HA deployment in OpenStack-only environment
./deploy.sh -o mitaka -s nosdn -t nonha -l default -f ipv6
```

Please **NOTE** that:

- You need to refer to **installer's documentation** for other necessary parameters applicable to your deployment.
- You need to refer to **Release Notes** and **installer's documentation** if there is any issue in installation.

OPNFV-NATIVE-INSTALL-2: Clone the following GitHub repository to get the configuration and metadata files

```
git clone https://github.com/sridhargaddam/opnfv_os_ipv6_poc.git \
/opt/stack/opnfv_os_ipv6_poc
```

3.1.2.2 Disable Security Groups in OpenStack ML2 Setup

Please **NOTE** that although Security Groups feature has been disabled automatically through `local.conf` configuration file by some installers such as `devstack`, it is very likely that other installers such as `Apex`, `Compass`, `Fuel` or `Joid` will enable Security Groups feature after installation.

Please make sure that Security Groups are disabled in the setup

In order to disable Security Groups globally, please make sure that the settings in **OPNFV-NATIVE-SEC-1** and **OPNFV-NATIVE-SEC-2** are applied, if they are not there by default.

OPNFV-NATIVE-SEC-1: Change the settings in `/etc/neutron/plugins/ml2/ml2_conf.ini` as follows, if they are not there by default

```
# /etc/neutron/plugins/ml2/ml2_conf.ini
[securitygroup]
enable_security_group = True
firewall_driver = neutron.agent.firewall.NoopFirewallDriver
[ml2]
extension_drivers = port_security
[agent]
prevent_arp_spoofing = False
```

OPNFV-NATIVE-SEC-2: Change the settings in `/etc/nova/nova.conf` as follows, if they are not there by default.

```
# /etc/nova/nova.conf
[DEFAULT]
security_group_api = neutron
firewall_driver = nova.virt.firewall.NoopFirewallDriver
```

OPNFV-NATIVE-SEC-3: After updating the settings, you will have to restart the `Neutron` and `Nova` services.

Please note that the commands of restarting `Neutron` and `Nova` would vary depending on the installer. Please refer to relevant documentation of specific installers

3.1.2.3 Set Up Service VM as IPv6 vRouter

OPNFV-NATIVE-SETUP-1: Now we assume that OpenStack multi-node setup is up and running. We have to source the tenant credentials in OpenStack controller node in this step. Please **NOTE** that the method of sourcing tenant credentials may vary depending on installers. For example:

Apex installer:

```
# On jump host, source the tenant credentials using /bin/opnfv-util provided by Apex␣
↪installer
opnfv-util undercloud "source overcloudrc; keystone service-list"

# Alternatively, you can copy the file /home/stack/overcloudrc from the installer VM␣
↪called "undercloud"
# to a location in controller node, for example, in the directory /opt, and do:
# source /opt/overcloudrc
```

Compass installer:

```
# source the tenant credentials using Compass installer of OPNFV
source /opt/admin-openrc.sh
```

Fuel installer:

```
# source the tenant credentials using Fuel installer of OPNFV
source /root/openrc
```

Joid installer:

```
# source the tenant credentials using Joid installer of OPNFV
source $HOME/joid_config/admin-openrc
```

devstack:

```
# source the tenant credentials in devstack
source openrc admin demo
```

Please refer to relevant documentation of installers if you encounter any issue.

OPNFV-NATIVE-SETUP-2: Download `fedora22` image which would be used for `vRouter`

```
wget https://download.fedoraproject.org/pub/fedora/linux/releases/22/Cloud/x86_64/\
Images/Fedora-Cloud-Base-22-20150521.x86_64.qcow2
```

OPNFV-NATIVE-SETUP-3: Import Fedora22 image to `glance`

```
glance image-create --name 'Fedora22' --disk-format qcow2 --container-format bare \
--file ./Fedora-Cloud-Base-22-20150521.x86_64.qcow2
```

OPNFV-NATIVE-SETUP-4: This step is Informational. OPNFV Installer has taken care of this step during deployment. You may refer to this step only if there is any issue, or if you are using other installers.

We have to move the physical interface (i.e. the public network interface) to `br-ex`, including moving the public IP address and setting up default route. Please refer to `OS-NATIVE-SETUP-4` and `OS-NATIVE-SETUP-5` in our more complete instruction.

OPNFV-NATIVE-SETUP-5: Create Neutron routers `ipv4-router` and `ipv6-router` which need to provide external connectivity.

```
neutron router-create ipv4-router
neutron router-create ipv6-router
```

OPNFV-NATIVE-SETUP-6: Create an external network/subnet `ext-net` using the appropriate values based on the data-center physical network setup.

Please **NOTE** that you may only need to create the subnet of `ext-net` because OPNFV installers should have created an external network during installation. You must use the same name of external network that installer creates when you create the subnet. For example:

- **Apex** installer: `external`
- **Compass** installer: `ext-net`
- **Fuel** installer: `admin_floating_net`
- **Joid** installer: `ext-net`

Please refer to the documentation of installers if there is any issue

```
# This is needed only if installer does not create an external work
# Otherwise, skip this command "net-create"
neutron net-create --router:external ext-net

# Note that the name "ext-net" may work for some installers such as Compass and Joid
# Change the name "ext-net" to match the name of external network that an installer␣
↪creates
neutron subnet-create --disable-dhcp --allocation-pool start=198.59.156.251,\
end=198.59.156.254 --gateway 198.59.156.1 ext-net 198.59.156.0/24
```

OPNFV-NATIVE-SETUP-7: Create Neutron networks `ipv4-int-network1` and `ipv6-int-network2` with port_security disabled

```
neutron net-create ipv4-int-network1
neutron net-create ipv6-int-network2
```

OPNFV-NATIVE-SETUP-8: Create IPv4 subnet `ipv4-int-subnet1` in the internal network `ipv4-int-network1`, and associate it to `ipv4-router`.

```
neutron subnet-create --name ipv4-int-subnet1 --dns-nameserver 8.8.8.8 \
ipv4-int-network1 20.0.0.0/24

neutron router-interface-add ipv4-router ipv4-int-subnet1
```

OPNFV-NATIVE-SETUP-9: Associate the `ext-net` to the Neutron routers `ipv4-router` and `ipv6-router`.

```
# Note that the name "ext-net" may work for some installers such as Compass and Joid
# Change the name "ext-net" to match the name of external network that an installer␣
↪creates
neutron router-gateway-set ipv4-router ext-net
neutron router-gateway-set ipv6-router ext-net
```

OPNFV-NATIVE-SETUP-10: Create two subnets, one IPv4 subnet `ipv4-int-subnet2` and one IPv6 subnet `ipv6-int-subnet2` in `ipv6-int-network2`, and associate both subnets to `ipv6-router`

```
neutron subnet-create --name ipv4-int-subnet2 --dns-nameserver 8.8.8.8 \
ipv6-int-network2 10.0.0.0/24

neutron subnet-create --name ipv6-int-subnet2 --ip-version 6 --ipv6-ra-mode slaac \
--ipv6-address-mode slaac ipv6-int-network2 2001:db8:0:1::/64

neutron router-interface-add ipv6-router ipv4-int-subnet2
neutron router-interface-add ipv6-router ipv6-int-subnet2
```

OPNFV-NATIVE-SETUP-11: Create a keypair

```
nova keypair-add vRouterKey > ~/vRouterKey
```

OPNFV-NATIVE-SETUP-12: Create ports for vRouter (with some specific MAC address - basically for automation - to know the IPv6 addresses that would be assigned to the port).

```
neutron port-create --name eth0-vRouter --mac-address fa:16:3e:11:11:11 ipv6-int-
↪network2
neutron port-create --name eth1-vRouter --mac-address fa:16:3e:22:22:22 ipv4-int-
↪network1
```

OPNFV-NATIVE-SETUP-13: Create ports for VM1 and VM2.

```
neutron port-create --name eth0-VM1 --mac-address fa:16:3e:33:33:33 ipv4-int-network1
neutron port-create --name eth0-VM2 --mac-address fa:16:3e:44:44:44 ipv4-int-network1
```

OPNFV-NATIVE-SETUP-14: Update `ipv6-router` with routing information to subnet `2001:db8:0:2::/64`

```
neutron router-update ipv6-router --routes type=dict list=true \
destination=2001:db8:0:2::/64,nexthop=2001:db8:0:1:f816:3eff:fe11:1111
```

OPNFV-NATIVE-SETUP-15: Boot Service VM (`vRouter`), VM1 and VM2

```
nova boot --image Fedora22 --flavor m1.small \
--user-data /opt/stack/opnfv_os_ipv6_poc/metadata.txt \
--availability-zone nova:opnfv-os-compute \
--nic port-id=$(neutron port-list | grep -w eth0-vRouter | awk '{print $2}') \
--nic port-id=$(neutron port-list | grep -w eth1-vRouter | awk '{print $2}') \
```

(continues on next page)

(continued from previous page)

```
--key-name vRouterKey vRouter

nova list

# Please wait for some 10 to 15 minutes so that necessary packages (like radvd)
# are installed and vRouter is up.
nova console-log vRouter

nova boot --image cirros-0.3.4-x86_64-uec --flavor m1.tiny \
--user-data /opt/stack/opnfv_os_ipv6_poc/set_mtu.sh \
--availability-zone nova:opnfv-os-controller \
--nic port-id=$(neutron port-list | grep -w eth0-VM1 | awk '{print $2}') \
--key-name vRouterKey VM1

nova boot --image cirros-0.3.4-x86_64-uec --flavor m1.tiny
--user-data /opt/stack/opnfv_os_ipv6_poc/set_mtu.sh \
--availability-zone nova:opnfv-os-compute \
--nic port-id=$(neutron port-list | grep -w eth0-VM2 | awk '{print $2}') \
--key-name vRouterKey VM2

nova list # Verify that all the VMs are in ACTIVE state.
```

OPNFV-NATIVE-SETUP-16: If all goes well, the IPv6 addresses assigned to the VMs would be as shown as follows:

```
# vRouter eth0 interface would have the following IPv6 address:
#     2001:db8:0:1:f816:3eff:fe11:1111/64
# vRouter eth1 interface would have the following IPv6 address:
#     2001:db8:0:2::1/64
# VM1 would have the following IPv6 address:
#     2001:db8:0:2:f816:3eff:fe33:3333/64
# VM2 would have the following IPv6 address:
#     2001:db8:0:2:f816:3eff:fe44:4444/64
```

OPNFV-NATIVE-SETUP-17: Now we need to disable `eth0-VM1`, `eth0-VM2`, `eth0-vRouter` and `eth1-vRouter` port-security

```
for port in eth0-VM1 eth0-VM2 eth0-vRouter eth1-vRouter
do
    neutron port-update --no-security-groups $port
    neutron port-update $port --port-security-enabled=False
    neutron port-show $port | grep port_security_enabled
done
```

OPNFV-NATIVE-SETUP-18: Now we can `SSH` to VMs. You can execute the following command.

```
# 1. Create a floatingip and associate it with VM1, VM2 and vRouter (to the port id␣
↪that is passed).
#    Note that the name "ext-net" may work for some installers such as Compass and␣
↪Joid
#    Change the name "ext-net" to match the name of external network that an␣
↪installer creates
neutron floatingip-create --port-id $(neutron port-list | grep -w eth0-VM1 | \
awk '{print $2}') ext-net
neutron floatingip-create --port-id $(neutron port-list | grep -w eth0-VM2 | \
awk '{print $2}') ext-net
neutron floatingip-create --port-id $(neutron port-list | grep -w eth1-vRouter | \
```

(continues on next page)

(continued from previous page)

```
awk '{print $2}') ext-net

# 2. To know / display the floatingip associated with VM1, VM2 and vRouter.
neutron floatingip-list -F floating_ip_address -F port_id | grep $(neutron port-list␣
↪| \
grep -w eth0-VM1 | awk '{print $2}') | awk '{print $2}'
neutron floatingip-list -F floating_ip_address -F port_id | grep $(neutron port-list␣
↪| \
grep -w eth0-VM2 | awk '{print $2}') | awk '{print $2}'
neutron floatingip-list -F floating_ip_address -F port_id | grep $(neutron port-list␣
↪| \
grep -w eth1-vRouter | awk '{print $2}') | awk '{print $2}'

# 3. To ssh to the vRouter, VM1 and VM2, user can execute the following command.
ssh -i ~/vRouterKey fedora@<floating-ip-of-vRouter>
ssh -i ~/vRouterKey cirros@<floating-ip-of-VM1>
ssh -i ~/vRouterKey cirros@<floating-ip-of-VM2>
```

If everything goes well, `ssh` will be successful and you will be logged into those VMs. Run some commands to verify that IPv6 addresses are configured on `eth0` interface.

OPNFV-NATIVE-SETUP-19: Show an IPv6 address with a prefix of `2001:db8:0:2::/64`

```
ip address show
```

OPNFV-NATIVE-SETUP-20: ping some external IPv6 address, e.g. `ipv6-router`

```
ping6 2001:db8:0:1::1
```

If the above ping6 command succeeds, it implies that `vRouter` was able to successfully forward the IPv6 traffic to reach external `ipv6-router`.

3.2 IPv6 Post Installation Procedures

Congratulations, you have completed the setup of using a service VM to act as an IPv6 vRouter. You have validated the setup based on the instruction in previous sections. If you want to further test your setup, you can `ping6` among `VM1`, `VM2`, `vRouter` and `ipv6-router`.

This setup allows further open innovation by any 3rd-party.

3.2.1 Automated post installation activities

Refer to the relevant testing guides, results, and release notes of Yardstick Project.

CHAPTER 4

Using IPv6 Feature of Hunter Release

Abstract

This section provides the users with:

- Gap Analysis regarding IPv6 feature requirements with OpenStack Rocky Official Release
- Gap Analysis regarding IPv6 feature requirements with Open Daylight Fluorine Official Release
- IPv6 Setup in Container Networking
- Use of Neighbor Discovery (ND) Proxy to connect IPv6-only container to external network
- Docker IPv6 Simple Cluster Topology
- Study and recommendation regarding Docker IPv6 NAT

The gap analysis serves as feature specific user guides and references when as a user you may leverage the IPv6 feature in the platform and need to perform some IPv6 related operations.

The IPv6 Setup in Container Networking serves as feature specific user guides and references when as a user you may want to explore IPv6 in Docker container environment. The use of NDP Proxying is explored to connect IPv6-only containers to external network. The Docker IPv6 simple cluster topology is studied with two Hosts, each with 2 Docker containers. Docker IPv6 NAT topic is also explored.

For more information, please find Neutron's IPv6 document for Rocky Release.

4.1 IPv6 Gap Analysis with OpenStack Rocky

This section provides users with IPv6 gap analysis regarding feature requirement with OpenStack Neutron in Rocky Official Release. The following table lists the use cases / feature requirements of VIM-agnostic IPv6 functionality, including infrastructure layer and VNF (VM) layer, and its gap analysis with OpenStack Neutron in Rocky Official Release.

Please **NOTE** that in terms of IPv6 support in OpenStack Neutron, there is no difference between **Rocky** release and prior, e.g. **Queens**, **Pike** and **Ocata**, releases.

Use Case / Requirement	Supported in Rocky	Notes
All topologies work in a multi-tenant environment	Yes	The IPv6 design is following the Neutron tenant networks model; dnsmasq is being used inside DHCP network namespaces, while radvd is being used inside Neutron routers namespaces to provide full isolation between tenants. Tenant isolation can be based on VLANs, GRE, or VXLAN encapsulation. In case of overlays, the transport network (and VTEPs) must be IPv4 based as of today.
IPv6 VM to VM only	Yes	It is possible to assign IPv6-only addresses to VMs. Both switching (within VMs on the same tenant network) as well as east/west routing (between different networks of the same tenant) are supported.
IPv6 external L2 VLAN directly attached to a VM	Yes	IPv6 provider network model; RA messages from upstream (external) router are forwarded into the VMs
IPv6 subnet routed via L3 agent to an external IPv6 network 1. Both VLAN and overlay (e.g. GRE, VXLAN) subnet attached to VMs; 2. Must be able to support multiple L3 agents for a given external network to support scaling (neutron scheduler to assign vRouters to the L3 agents)	1. Yes 2. Yes	Configuration is enhanced since Kilo to allow easier setup of the upstream gateway, without the user being forced to create an IPv6 subnet for the external network.
Ability for a NIC to support both IPv4 and IPv6 (dual stack) address. 1. VM with a single interface associated with a network, which is then associated with two subnets. 2. VM with two different interfaces associated with two different networks and two different subnets.	1. Yes 2. Yes	Dual-stack is supported in Neutron with the addition of `Multiple IPv6 Prefixes` Blueprint
Support IPv6 Address assignment modes. 1. SLAAC 2. DHCPv6 Stateless 3. DHCPv6 Stateful	1. Yes 2. Yes 3. Yes	

Continued on next page

Table 1 – continued from previous page

Use Case / Requirement	Supported in Rocky	Notes
Ability to create a port on an IPv6 DHCPv6 Stateful subnet and assign a specific IPv6 address to the port and have it taken out of the DHCP address pool.	Yes	
Ability to create a port with fixed_ip for a SLAAC/DHCPv6-Stateless Subnet.	**No**	The following patch disables this operation: https://review.openstack.org/#/c/129144/
Support for private IPv6 to external IPv6 floating IP; Ability to specify floating IPs via Neutron API (REST and CLI) as well as via Horizon, including combination of IPv6/IPv4 and IPv4/IPv6 floating IPs if implemented.	**Rejected**	Blueprint proposed in upstream and got rejected. General expectation is to avoid NAT with IPv6 by assigning GUA to tenant VMs. See https://review.openstack.org/#/c/139731/ for discussion.
Provide IPv6/IPv4 feature parity in support for pass-through capabilities (e.g., SR-IOV).	**To-Do**	The L3 configuration should be transparent for the SR-IOV implementation. SR-IOV networking support introduced in Juno based on the `sriovnicswitch` ML2 driver is expected to work with IPv4 and IPv6 enabled VMs. We need to verify if it works or not.
Additional IPv6 extensions, for example: IPSEC, IPv6 Anycast, Multicast	**No**	It does not appear to be considered yet (lack of clear requirements)
VM access to the meta-data server to obtain user data, SSH keys, etc. using cloud-init with IPv6 only interfaces.	**No**	This is currently not supported. Config-drive or dual-stack IPv4 / IPv6 can be used as a workaround (so that the IPv4 network is used to obtain connectivity with the meta-data service). The following blog How to Use Config-Drive for Metadata with IPv6 Network provides a neat summary on how to use config-drive for metadata with IPv6 network.
Full support for IPv6 matching (i.e., IPv6, ICMPv6, TCP, UDP) in security groups. Ability to control and manage all IPv6 security group capabilities via Neutron/Nova API (REST and CLI) as well as via Horizon.	Yes	Both IPTables firewall driver and OVS firewall driver support IPv6 Security Group API.

Continued on next page

Table 1 – continued from previous page

Use Case / Requirement	Supported in Rocky	Notes
During network/subnet/router create, there should be an option to allow user to specify the type of address management they would like. This includes all options including those low priority if implemented (e.g., toggle on/off router and address prefix advertisements); It must be supported via Neutron API (REST and CLI) as well as via Horizon	Yes	Two new Subnet attributes were introduced to control IPv6 address assignment options: • `ipv6-ra-mode`: to determine who sends Router Advertisements; • `ipv6-address-mode`: to determine how VM obtains IPv6 address, default gateway, and/or optional information.
Security groups anti-spoofing: Prevent VM from using a source IPv6/MAC address which is not assigned to the VM	Yes	
Protect tenant and provider network from rogue RAs	Yes	When using a tenant network, Neutron is going to automatically handle the filter rules to allow connectivity of RAs to the VMs only from the Neutron router port; with provider networks, users are required to specify the LLA of the upstream router during the subnet creation, or otherwise manually edit the security-groups rules to allow incoming traffic from this specific address.
Support the ability to assign multiple IPv6 addresses to an interface; both for Neutron router interfaces and VM interfaces.	Yes	
Ability for a VM to support a mix of multiple IPv4 and IPv6 networks, including multiples of the same type.	Yes	
IPv6 Support in "Allowed Address Pairs" Extension	Yes	
Support for IPv6 Prefix Delegation.	Yes	Partial support in Rocky
Distributed Virtual Routing (DVR) support for IPv6	**No**	In Rocky DVR implementation, IPv6 works. But all the IPv6 ingress/ egress traffic is routed via the centralized controller node, i.e. similar to SNAT traffic. A fully distributed IPv6 router is not yet supported in Neutron.
VPNaaS	Yes	VPNaaS supports IPv6. But this feature is not extensively tested.
FWaaS	Yes	
BGP Dynamic Routing Support for IPv6 Prefixes	Yes	BGP Dynamic Routing supports peering via IPv6 and advertising IPv6 prefixes.

Continued on next page

Table 1 – continued from previous page

Use Case / Requirement	Supported in Rocky	Notes
VxLAN Tunnels with IPv6 endpoints.	Yes	Neutron ML2/OVS supports configuring local_ip with IPv6 address so that VxLAN tunnels are established with IPv6 addresses. This feature requires OVS 2.6 or higher version.
IPv6 First-Hop Security, IPv6 ND spoofing	Yes	
IPv6 support in Neutron Layer3 High Availability (keepalived+VRRP).	Yes	

4.2 IPv6 Gap Analysis with Open Daylight Fluorine

This section provides users with IPv6 gap analysis regarding feature requirement with Open Daylight Fluorine Official Release. The following table lists the use cases / feature requirements of VIM-agnostic IPv6 functionality, including infrastructure layer and VNF (VM) layer, and its gap analysis with Open Daylight Fluorine Official Release.

Open Daylight Fluorine Status

In Open Daylight Fluorine official release, the legacy `Old Netvirt` identified by feature `odl-ovsdb-openstack` is deprecated and no longer supported. The `New Netvirt` identified by feature `odl-netvirt-openstack` is used.

Two new features are supported in Open Daylight Fluorine official release:

- Support for advertising MTU info in IPv6 RAs
- IPv6 external connectivity for FLAT/VLAN based provider networks

Use Case / Requirement	Supported in ODL Fluorine	Notes
REST API support for IPv6 subnet creation in ODL	Yes	Yes, it is possible to create IPv6 subnets in ODL using Neutron REST API. For a network which has both IPv4 and IPv6 subnets, ODL mechanism driver will send the port information which includes IPv4/v6 addresses to ODL Neutron northbound API. When port information is queried, it displays IPv4 and IPv6 addresses.
IPv6 Router support in ODL: 1. Communication between VMs on same network	Yes	
IPv6 Router support in ODL: 2. Communication between VMs on different networks connected to the same router (east-west)	Yes	

Continued on next page

Table 2 – continued from previous page

Use Case / Requirement	Supported in ODL Fluorine	Notes
IPv6 Router support in ODL: 3. External routing (north-south)	**NO**	This feature is targeted for Flourine Release. In ODL Fluorine Release, RFE "IPv6 Inter-DC L3 North-South Connectivity Using L3VPN Provider Network Types" Spec[1] is merged. But the code patch has not been merged yet. On the other hand, "IPv6 Cluster Support" is available in Fluorine Release[2]. Basically, existing IPv6 features were enhanced to work in a three node ODL Clustered Setup.
IPAM: Support for IPv6 Address assignment modes. 1. SLAAC 2. DHCPv6 Stateless 3. DHCPv6 Stateful	Yes	ODL IPv6 Router supports all the IPv6 Address assignment modes along with Neutron DHCP Agent.
When using ODL for L2 forwarding/tunneling, it is compatible with IPv6.	Yes	
Full support for IPv6 matching (i.e. IPv6, ICMPv6, TCP, UDP) in security groups. Ability to control and manage all IPv6 security group capabilities via Neutron/Nova API (REST and CLI) as well as via Horizon	Yes	
Shared Networks support	Yes	
IPv6 external L2 VLAN directly attached to a VM.	Yes	Targeted for Flourine Release
ODL on an IPv6 only Infrastructure.	Yes	Deploying OpenStack with ODL on an IPv6 only infrastructure where the API endpoints are all IPv6 addresses.
VxLAN Tunnels with IPv6 Endpoints	Yes	
IPv6 L3VPN Dual Stack with Single router	Yes	Refer to "Dual Stack VM support in OpenDaylight" Spec[3].
IPv6 Inter Data Center using L3VPNs	Yes	Refer to "IPv6 Inter-DC L3 North-South connectivity using L3VPN provider network types" Spec[1].
Support for advertising MTU info in IPv6 RAs	Yes	
IPv6 external connectivity for FLAT/VLAN based provider networks	Yes	

[1] https://docs.opendaylight.org/projects/netvirt/en/stable-fluorine/specs/oxygen/ipv6-interdc-l3vpn.html
[2] http://git.opendaylight.org/gerrit/#/c/66707/
[3] https://docs.opendaylight.org/projects/netvirt/en/stable-fluorine/specs/oxygen/l3vpn-dual-stack-vms.html

4.3 Exploring IPv6 in Container Networking

This document is the summary of how to use IPv6 with Docker.

The defualt Docker container uses 172.17.0.0/24 subnet with 172.17.0.1 as gateway. So IPv6 network needs to be enabled and configured before we can use it with IPv6 traffic.

We will describe how to use IPv6 in Docker in the following 5 sections:

1. Install Docker Community Edition (CE)
2. IPv6 with Docker
3. Design Simple IPv6 Topologies
4. Design Solutions
5. Challenges in Production Use

4.3.1 Install Docker Community Edition (CE)

Step 3.1.1: Download Docker (CE) on your system from "this link"[1].

For Ubuntu 16.04 Xenial x86_64, please refer to "Docker CE for Ubuntu"[2].

Step 3.1.2: Refer to "this link"[3] to install Docker CE on Xenial.

Step 3.1.3: Once you installed the docker, you can verify the standalone default bridge nework as follows:

```
$ docker network ls
NETWORK ID NAME DRIVER SCOPE
b9e92f9a8390 bridge bridge local
74160ae686b9 host host local
898fbb0a0c83 my_bridge bridge local
57ac095fdaab none null local
```

Note that:

- the details may be different with different network drivers.
- User-defined bridge networks are the best when you need multiple containers to communicate on the same Docker host.
- Host networks are the best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.
- Overlay networks are the best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.
- Macvlan networks are the best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.
- Third-party network plugins allow you to integrate Docker with specialized network stacks. Please refer to "Docker Networking Tutorials"[4].

[1] https://www.docker.com/community-edition#/download
[2] https://store.docker.com/editions/community/docker-ce-server-ubuntu
[3] https://docs.docker.com/install/linux/docker-ce/ubuntu/#install-docker-ce-1
[4] https://docs.docker.com/network/network-tutorial-host/#other-networking-tutorials

```
# This will have docker0 default bridge details showing
# ipv4 172.17.0.1/16 and
# ipv6 fe80::42:4dff:fe2f:baa6/64 entries

$ ip addr show
11: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group␣
↪default
link/ether 02:42:4d:2f:ba:a6 brd ff:ff:ff:ff:ff:ff
inet 172.17.0.1/16 scope global docker0
valid_lft forever preferred_lft forever
inet6 fe80::42:4dff:fe2f:baa6/64 scope link
valid_lft forever preferred_lft forever
```

Thus we see here a simple defult ipv4 networking for docker. Inspect and verify that IPv6 address is not listed here showing its enabled but not used by default docker0 bridge.

You can create user defined bridge network using command like `my_bridge` below with other than default, e.g. 172.18.0.0/24 here. **Note** that `--ipv6` is not specified yet

```
$ sudo docker network create \
              --driver=bridge \
              --subnet=172.18.0.0/24 \
              --gaeway= 172.18.0.1 \
              my_bridge

$ docker network inspect bridge
[
  {
    "Name": "bridge",
    "Id": "b9e92f9a839048aab887081876fc214f78e8ce566ef5777303c3ef2cd63ba712",
    "Created": "2017-10-30T23:32:15.676301893-07:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
        "Driver": "default",
        "Options": null,
        "Config": [
            {
                "Subnet": "172.17.0.0/16",
                "Gateway": "172.17.0.1"
            }
        ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
        "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
        "ea76bd4694a8073b195dd712dd0b070e80a90e97b6e2024b03b711839f4a3546": {
        "Name": "registry",
        "EndpointID":
↪"b04dc6c5d18e3bf4e4201aa8ad2f6ad54a9e2ea48174604029576e136b99c49d",
        "MacAddress": "02:42:ac:11:00:02",
```

(continues on next page)

(continued from previous page)

```
        "IPv4Address": "172.17.0.2/16",
        "IPv6Address": ""
        }
    },
    "Options": {
        "com.docker.network.bridge.default_bridge": "true",
        "com.docker.network.bridge.enable_icc": "true",
        "com.docker.network.bridge.enable_ip_masquerade": "true",
        "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
        "com.docker.network.bridge.name": "docker0",
        "com.docker.network.driver.mtu": "1500"
    },
    "Labels": {}
  }
]

$ sudo docker network inspect my_bridge
[
  {
    "Name": "my_bridge",
    "Id": "898fbb0a0c83acc0593897f5af23b1fe680d38b804b0d5a4818a4117ac36498a",
    "Created": "2017-07-16T17:59:55.388151772-07:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
        "Driver": "default",
        "Options": {},
        "Config": [
            {
                "Subnet": "172.18.0.0/16",
                "Gateway": "172.18.0.1"
            }
        ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
        "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {},
    "Options": {},
    "Labels": {}
  }
]
```

You can note that IPv6 is not enabled here yet as seen through network inspect. Since we have only IPv4 installed with Docker, we will move to enable IPv6 for Docker in the next step.

4.3.2 IPv6 with Docker

Verifyig IPv6 with Docker involves the following steps:

Step 3.2.1: Enable ipv6 support for Docker

In the simplest term, the first step is to enable IPv6 on Docker on Linux hosts. Please refer to "this link"[5]:

- Edit `/etc/docker/daemon.json`
- Set the `ipv6` key to true.

```
{{{ "ipv6": true }}}
```

Save the file.

Step 3.2.1.1: Set up IPv6 addressing for Docker in `daemon.json`

If you need IPv6 support for Docker containers, you need to enable the option on the Docker daemon `daemon.json` and reload its configuration, before creating any IPv6 networks or assigning containers IPv6 addresses.

When you create your network, you can specify the `--ipv6` flag to enable IPv6. You can't selectively disable IPv6 support on the default bridge network.

Step 3.2.1.2: Enable forwarding from Docker containers to the outside world

By default, traffic from containers connected to the default bridge network is not forwarded to the outside world. To enable forwarding, you need to change two settings. These are not Docker commands and they affect the Docker host's kernel.

- Setting 1: Configure the Linux kernel to allow IP forwarding:

```
$ sysctl net.ipv4.conf.all.forwarding=1
```

- Setting 2: Change the policy for the iptables FORWARD policy from DROP to ACCEPT.

```
$ sudo iptables -P FORWARD ACCEPT
```

These settings do not persist across a reboot, so you may need to add them to a start-up script.

Step 3.2.1.3: Use the default bridge network

The default bridge network is considered a legacy detail of Docker and is not recommended for production use. Configuring it is a manual operation, and it has technical shortcomings.

Step 3.2.1.4: Connect a container to the default bridge network

If you do not specify a network using the `--network` flag, and you do specify a network driver, your container is connected to the default bridge network by default. Containers connected to the default bridge network can communicate, but only by IP address, unless they are linked using the legacy `--link` flag.

Step 3.2.1.5: Configure the default bridge network

To configure the default bridge network, you specify options in `daemon.json`. Here is an example of `daemon.json` with several options specified. Only specify the settings you need to customize.

```
{
  "bip": "192.168.1.5/24",
  "fixed-cidr": "192.168.1.5/25",
  "fixed-cidr-v6": "2001:db8::/64",
  "mtu": 1500,
  "default-gateway": "10.20.1.1",
  "default-gateway-v6": "2001:db8:abcd::89",
  "dns": ["10.20.1.2","10.20.1.3"]
}
```

[5] https://docs.docker.com/config/daemon/ipv6/

Restart Docker for the changes to take effect.

Step 3.2.1.6: Use IPv6 with the default bridge network

If you configure Docker for IPv6 support (see **Step 2.1.1**), the default bridge network is also configured for IPv6 automatically. Unlike user-defined bridges, you cannot selectively disable IPv6 on the default bridge.

Step 3.2.1.7: Reload the Docker configuration file

```
$ systemctl reload docker
```

Step 3.2.1.8: You can now create networks with the `--ipv6` flag and assign containers IPv6 addresses.

Step 3.2.1.9: Verify your host and docker networks

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
↪STATUS              PORTS                    NAMES
ea76bd4694a8        registry:2          "/entrypoint.sh /e..."   x months ago
↪Up y months         0.0.0.0:4000->5000/tcp   registry

$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
b9e92f9a8390        bridge              bridge              local
74160ae686b9        host                host                local
898fbb0a0c83        my_bridge           bridge              local
57ac095fdaab        none                null                local
```

Step 3.2.1.10: Edit `/etc/docker/daemon.json` and set the ipv6 key to true.

```
{
  "ipv6": true
}
```

Save the file.

Step 3.2.1.11: Reload the Docker configuration file.

```
$ sudo systemctl reload docker
```

Step 3.2.1.12: You can now create networks with the `--ipv6` flag and assign containers IPv6 addresses using the `--ip6` flag.

```
$ sudo docker network create --ipv6 --driver bridge alpine-net--fixed-cidr-v6␣
↪2001:db8:1/64

# "docker network create" requires exactly 1 argument(s).
# See "docker network create --help"
```

Earlier, user was allowed to create a network, or start the daemon, without specifying an IPv6 `--subnet`, or `--fixed-cidr-v6` respectively, even when using the default builtin IPAM driver, which does not support auto allocation of IPv6 pools. In another word, it was an incorrect configurations, which had no effect on IPv6 stuff. It was a no-op.

A fix cleared that so that Docker will now correctly consult with the IPAM driver to acquire an IPv6 subnet for the bridge network, when user did not supply one.

If the IPAM driver in use is not able to provide one, network creation would fail (in this case the default bridge network).

So what you see now is the expected behavior. You need to remove the `--ipv6` flag when you start the daemon, unless you pass a `--fixed-cidr-v6` pool. We should probably clarify this somewhere.

The above was found on following Docker.

```
$ docker info
Containers: 27
Running: 1
Paused: 0
Stopped: 26
Images: 852
Server Version: 17.06.1-ce-rc1
Storage Driver: aufs
  Root Dir: /var/lib/docker/aufs
  Backing Filesystem: extfs
  Dirs: 637
  Dirperm1 Supported: false
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
  Volume: local
  Network: bridge host macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file logentries splunk syslog
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Init Binary: docker-init
containerd version: 6e23458c129b551d5c9871e5174f6b1b7f6d1170
runc version: 810190ceaa507aa2727d7ae6f4790c76ec150bd2
init version: 949e6fa
Security Options:
  apparmor
  seccomp
  Profile: default
Kernel Version: 3.13.0-88-generic
Operating System: Ubuntu 16.04.2 LTS
OSType: linux
Architecture: x86_64
CPUs: 4
Total Memory: 11.67GiB
Name: aatiksh
ID: HS5N:T7SK:73MD:NZGR:RJ2G:R76T:NJBR:U5EJ:KP5N:Q3VO:6M2O:62CJ
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Registry: https://index.docker.io/v1/
Experimental: false
Insecure Registries:
  127.0.0.0/8
Live Restore Enabled: false
```

Step 3.2.2: Check the network drivers

Among the 4 supported drivers, we will be using "User-Defined Bridge Network"[6].

[6] https://docs.docker.com/network/

4.3.3 Design Simple IPv6 Topologies

Step 3.3.1: Creating IPv6 user-defined subnet.

Let's create a Docker with IPv6 subnet:

```
$ sudo docker network create \
              --ipv6 \
              --driver=bridge \
              --subnet=172.18.0.0/16 \
              --subnet=fcdd:1::/48 \
              --gaeway= 172.20.0.1  \
              my_ipv6_bridge

# Error response from daemon:

cannot create network␣
↪8957e7881762bbb4b66c3e2102d72b1dc791de37f2cafbaff42bdbf891b54cc3 (br-8957e7881762):␣
↪conflicts with network
no matching subnet for range 2002:ac14:0000::/48

# try changing to ip-addess-range instead of subnet for ipv6.
# networks have overlapping IPv4

NETWORK ID          NAME                DRIVER              SCOPE
b9e92f9a8390        bridge              bridge              local
74160ae686b9        host                host                local
898fbb0a0c83        my_bridge           bridge              local
57ac095fdaab        none                null                local
no matching subnet for gateway 172.20.01

# So finally making both as subnet and gateway as 172.20.0.1 works

$ sudo docker network create \
              --ipv6 \
              --driver=bridge \
              --subnet=172.20.0.0/16 \
              --subnet=2002:ac14:0000::/48 \
              --gateway=172.20.0.1 \
              my_ipv6_bridge
898fbb0a0c83acc0593897f5af23b1fe680d38b804b0d5a4818a4117ac36498a (br-898fbb0a0c83):
```

Since lxdbridge used the ip range on the system there was a conflict. This brings us to question how do we assign IPv6 and IPv6 address for our solutions.

4.3.4 Design Solutions

For best practices, please refer to "Best Practice Document"[7].

Use IPv6 Calcualtor at "this link"[8].

- For IPv4 172.16.0.1 = 6to4 prefix 2002:ac10:0001::/48
- For IPv4 172.17.01/24 = 6to4 prefix 2002:ac11:0001::/48
- For IPv4 172.18.0.1 = 6to4 prefix 2002:ac12:0001::/48

[7] https://networkengineering.stackexchange.com/questions/119/ipv6-address-space-layout-best-practices
[8] http://www.gestioip.net/cgi-bin/subnet_calculator.cgi

- For IPv4 172.19.0.1 = 6to4 prefix 2002:ac13:0001::/48
- For IPv4 172.20.0.0 = 6to4 prefix 2002:ac14:0000::/48

To avoid overlaping IP's, let's use the .20 in our design:

```
$ sudo docker network create \
              --ipv6 \
              --driver=bridge \
              --subnet=172.20.0.0/24 \
              --subnet=2002:ac14:0000::/48
              --gateway=172.20.0.1
              my_ipv6_bridge

# created ...

052da268171ce47685fcdb68951d6d14e70b9099012bac410c663eb2532a0c87

$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
b9e92f9a8390        bridge              bridge              local
74160ae686b9        host                host                local
898fbb0a0c83        my_bridge           bridge              local
052da268171c        my_ipv6_bridge      bridge              local
57ac095fdaab        none                null                local

# Note the first 16 digits is used here as network id from what we got
# whaen we created it.

$ docker network  inspect my_ipv6_bridge
[
  {
    "Name": "my_ipv6_bridge",
    "Id": "052da268171ce47685fcdb68951d6d14e70b9099012bac410c663eb2532a0c87",
    "Created": "2018-03-16T07:20:17.714212288-07:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": true,
    "IPAM": {
        "Driver": "default",
        "Options": {},
        "Config": [
            {
                "Subnet": "172.20.0.0/16",
                "Gateway": "172.20.0.1"
            },
            {
                "Subnet": "2002:ac14:0000::/48"
            }
        ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
        "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {},
```

(continues on next page)

(continued from previous page)

```
    "Options": {},
    "Labels": {}
  }
]
```

Note that:

- IPv6 flag is ebnabled and that IPv6 range is listed besides Ipv4 gateway.
- We are mapping IPv4 and IPv6 address to simplify assignments as per "Best Pratice Document"[7].

Testing the solution and topology:

```
$ sudo docker run hello-world
Hello from Docker!
```

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash

root@62b88b030f5a:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
```

On terminal it appears that the docker is functioning normally.

Let's now push to see if we can use the `my_ipv6_bridge` network. Please refer to "User-Defined Bridge Network"[9].

4.3.4.1 Connect a container to a user-defined bridge

When you create a new container, you can specify one or more `--network` flags. This example connects a Nginx container to the `my-net` network. It also publishes port 80 in the container to port 8080 on the Docker host, so external clients can access that port. Any other container connected to the `my-net` network has access to all ports on the my-nginx container, and vice versa.

```
$ docker create --name my-nginx \
                --network my-net \
                --publish 8080:80 \
                nginx:latest
```

To connect a running container to an existing user-defined bridge, use the `docker network connect` command. The following command connects an already-running `my-nginx` container to an already-existing `my_ipv6_bridge` network:

[9] https://docs.docker.com/network/bridge/#use-ipv6-with-the-default-bridge-network

```
$ docker network connect my_ipv6_bridge my-nginx
```

Now we have connected the IPv6-enabled network to `mynginx` conatiner. Let's start and verify its IP Address:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
↪STATUS              PORTS                    NAMES
df1df6ed3efb        alpine              "ash"                    4 hours ago
↪Up 4 hours                                  alpine1
ea76bd4694a8        registry:2          "/entrypoint.sh /e..."   9 months ago
↪Up 4 months         0.0.0.0:4000->5000/tcp   registry
```

The `nginx:latest` image is not runnung, so let's start and log into it.

```
$ docker images | grep latest
REPOSITORY                                          TAG                 IMAGE ID
↪       CREATED             SIZE
nginx                                               latest              73acd1f0cfad
↪       2 days ago          109MB
alpine                                              latest              3fd9065eaf02
↪       2 months ago        4.15MB
swaggerapi/swagger-ui                               latest              e0b4f5dd40f9
↪       4 months ago        23.6MB
ubuntu                                              latest              d355ed3537e9
↪       8 months ago        119MB
hello-world                                         latest              1815c82652c0
↪       9 months ago        1.84kB
```

Now we do find the `nginx` and let's run it

```
$ docker run -i -t nginx:latest /bin/bash
root@bc13944d22e1:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
root@bc13944d22e1:/#
```

Open another terminal and check the networks and verify that IPv6 address is listed on the container:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
↪STATUS              PORTS                    NAMES
bc13944d22e1        nginx:latest        "/bin/bash"              About a minute ago
↪Up About a minute   80/tcp                   loving_hawking
df1df6ed3efb        alpine              "ash"                    4 hours ago
↪Up 4 hours                                  alpine1
ea76bd4694a8        registry:2          "/entrypoint.sh /e..."   9 months ago
↪Up 4 months         0.0.0.0:4000->5000/tcp   registry

$ ping6 bc13944d22e1

# On 2nd termoinal

$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
b9e92f9a8390        bridge              bridge              local
74160ae686b9        host                host                local
898fbb0a0c83        my_bridge           bridge              local
```

(continues on next page)

(continued from previous page)

```
052da268171c        my_ipv6_bridge      bridge              local
57ac095fdaab        none                null                local

$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group␣
↪default qlen 1000
    link/ether 8c:dc:d4:6e:d5:4b brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.80/24 brd 10.0.0.255 scope global dynamic eno1
       valid_lft 558367sec preferred_lft 558367sec
    inet6 2601:647:4001:739c:b80a:6292:1786:b26/128 scope global dynamic
       valid_lft 86398sec preferred_lft 86398sec
    inet6 fe80::8edc:d4ff:fe6e:d54b/64 scope link
       valid_lft forever preferred_lft forever
11: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group␣
↪default
    link/ether 02:42:4d:2f:ba:a6 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:4dff:fe2f:baa6/64 scope link
       valid_lft forever preferred_lft forever
20: br-052da268171c: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state␣
↪UP group default
    link/ether 02:42:5e:19:55:0d brd ff:ff:ff:ff:ff:ff
    inet 172.20.0.1/16 scope global br-052da268171c
       valid_lft forever preferred_lft forever
    inet6 2002:ac14::1/48 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::42:5eff:fe19:550d/64 scope link
       valid_lft forever preferred_lft forever
    inet6 fe80::1/64 scope link
       valid_lft forever preferred_lft forever
```

Note that on the 20th entry we have the `br-052da268171c` with IPv6 `inet6 2002:ac14::1/48` scope global, which belongs to root@bc13944d22e1.

At this time we have been able to provide a simple Docker with IPv6 solution.

4.3.4.2 Disconnect a container from a user-defined bridge

If another route needs to be added to `nginx`, you need to modify the routes:

```
# using ip route commands

$ ip r
default via 10.0.0.1 dev eno1  proto static  metric 100
default via 10.0.0.1 dev wlan0  proto static  metric 600
10.0.0.0/24 dev eno1  proto kernel  scope link  src 10.0.0.80
10.0.0.0/24 dev wlan0  proto kernel  scope link  src 10.0.0.38
10.0.0.0/24 dev eno1  proto kernel  scope link  src 10.0.0.80  metric 100
10.0.0.0/24 dev wlan0  proto kernel  scope link  src 10.0.0.38  metric 600
```

(continues on next page)

(continued from previous page)

```
10.0.8.0/24 dev lxdbr0  proto kernel  scope link  src 10.0.8.1
169.254.0.0/16 dev lxdbr0  scope link  metric 1000
172.17.0.0/16 dev docker0  proto kernel  scope link  src 172.17.0.1
172.18.0.0/16 dev br-898fbb0a0c83  proto kernel  scope link  src 172.18.0.1
172.20.0.0/16 dev br-052da268171c  proto kernel  scope link  src 172.20.0.1
192.168.99.0/24 dev vboxnet1  proto kernel  scope link  src 192.168.99.1
```

If the routes are correctly updated you should be able to see `nginx` web page on link `http://172.20.0.0.1`

We now have completed the exercise.

To disconnect a running container from a user-defined bridge, use the `docker network disconnect` command. The following command disconnects the `my-nginx` container from the `my-net` network.

```
$ docker network disconnect my_ipv6_bridge my-nginx
```

The IPv6 Docker we used is for demo purpose only. For real production we need to follow one of the IPv6 solutions we have come across.

4.3.5 Challenges in Production Use

"This link"[10] discusses the details of the use of `nftables` which is nextgen `iptables`, and tries to build production worthy Docker for IPv6 usage.

4.3.6 References

4.4 ICMPv6 and NDP

ICMP is a control protocol that is considered to be an integral part of IP, although it is architecturally layered upon IP, i.e., it uses IP to carry its data end-to-end just as a transport protocol like TCP or UDP does. ICMP provides error reporting, congestion reporting, and first-hop gateway redirection.

To communicate on its directly-connected network, a host must implement the communication protocol used to interface to that network. We call this a link layer or media-access layer protocol.

IPv4 uses ARP for link and MAC address discovery. In contrast IPv6 uses ICMPv6 though Neighbor Discovery Protocol (NDP). NDP defines five ICMPv6 packet types for the purpose of router solicitation, router advertisement, neighbor solicitation, neighbor advertisement, and network redirects. Refer RFC 122 & 3122.

Contrasting with ARP, NDP includes Neighbor Unreachability Detection (NUD), thus, improving robustness of packet delivery in the presence of failing routers or links, or mobile nodes. As long as hosts were using single network interface, the isolation between local network and remote network was simple. With requirements of multihoming for hosts with multiple interfaces and multiple destination packet transfers, the complications of maintaining all routing to remote gateways has disappeared.

To add container network to local network and IPv6 link local networks and virtual or logical routing on hosts, the complexity is now exponential. In order to maintain simplicity of end hosts (physical, virtual or containers), just maintaining sessions and remote gateways (routers), and maintaining routes independent of session state is still desirable for scaling internet connected end hosts.

For more details, please refer to[1].

[10] https://stephank.nl/p/2017-06-05-ipv6-on-production-docker.html
[1] https://en.wikipedia.org/wiki/Neighbor_Discovery_Protocol

4.4.1 IPv6-only Containers & Using NDP Proxying

IPv6-only containers will need to fully depend on NDP proxying.

If your Docker host is the only part of an IPv6 subnet but does not have an IPv6 subnet assigned, you can use NDP Proxying to connect your containers to the internet via IPv6.

If the host with IPv6 address `2001:db8::c001` is part of the subnet `2001:db8::/64`, and your IaaS provider allows you to configure the IPv6 addresses `2001:db8::c000 to 2001:db8::c00f`, your network configuration may look like the following:

```
$ ip -6 addr show

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536
   inet6 ::1/128 scope host
      valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qlen 1000
   inet6 2001:db8::c001/64 scope global
      valid_lft forever preferred_lft forever
   inet6 fe80::601:3fff:fea1:9c01/64 scope link
      valid_lft forever preferred_lft forever
```

To split up the configurable address range into two subnets `2001:db8::c000/125 and 2001:db8::c008/125`, use the following daemon.json settings.

```
{
  "ipv6": true,
  "fixed-cidr-v6": "2001:db8::c008/125"
}
```

The first subnet will be used by non-Docker processes on the host, and the second will be used by Docker.

For more details, please refer to[2].

4.4.2 References

4.5 Docker IPv6 Simple Cluster Topology

Using external switches or routers allows you to enable IPv6 communication between containers on different hosts. We have two physical hosts: Host1 & Host2, and we will study here two scenarios: one with Switch and the other one with router on the top of hierarchy, connecting those 2 hosts. Both hosts host a pair of containers in a cluster. The contents are borrowed from article[1] below, which can be used on any Linux distro (CentOS, Ubuntu, OpenSUSE etc) with latest kernel. A sample testing is pointed in the blog article[2] as a variation using ESXi & older Ubuntu 14.04.

4.5.1 Switched Network Environment

Using routable IPv6 addresses allows you to realize communication between containers on different hosts. Let's have a look at a simple Docker IPv6 cluster example:

The Docker hosts are in the `2001:db8:0::/64` subnet. Host1 is configured to provide addresses from the `2001:db8:1::/64` subnet to its containers. It has three routes configured:

[2] https://docs.docker.com/v17.09/engine/userguide/networking/default_network/ipv6/#using-ndp-proxying
[1] https://docs.docker.com/v17.09/engine/userguide/networking/default_network/ipv6/#docker-ipv6-cluster
[2] http://www.debug-all.com/?p=128

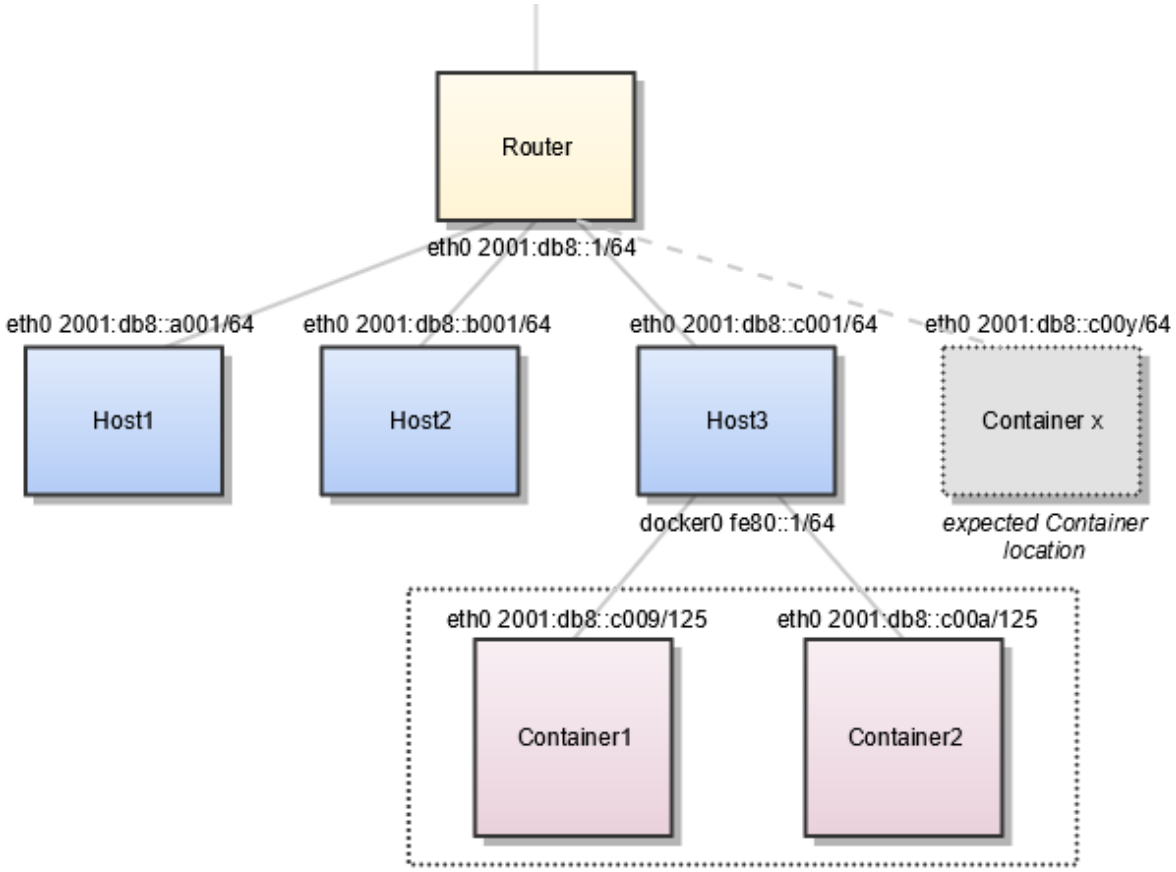


Fig. 1: Figure: Using NDP Proxying

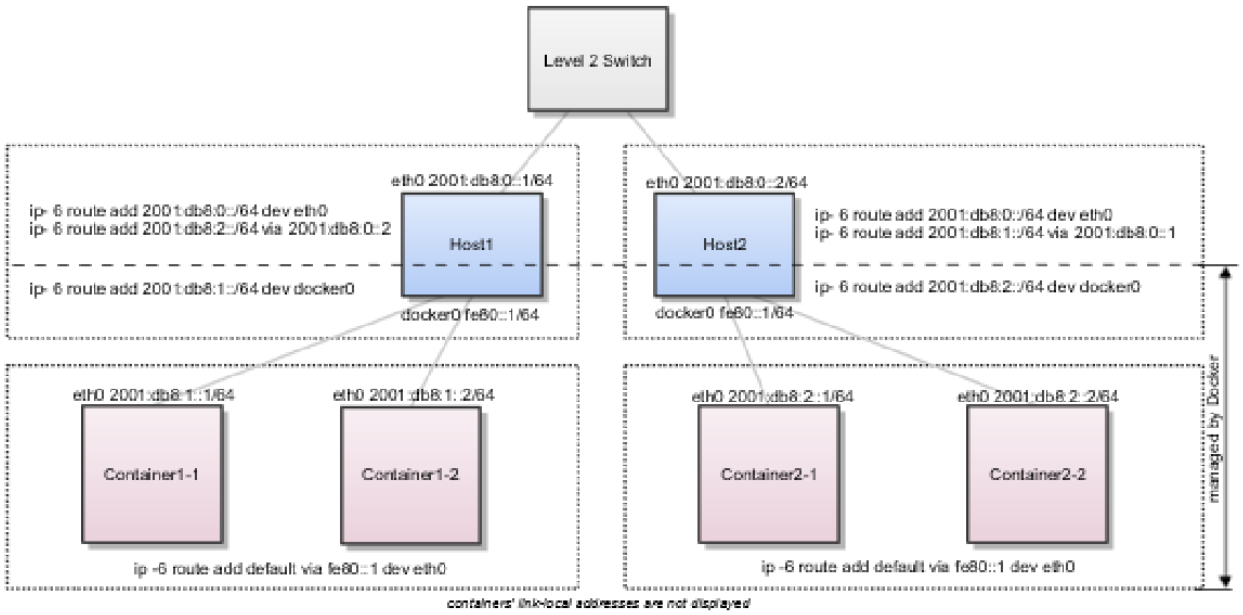


Fig. 2: Figure 1: An Docker IPv6 Cluster Example

- Route all traffic to `2001:db8:0::/64` via eth0
- Route all traffic to `2001:db8:1::/64` via docker0
- Route all traffic to `2001:db8:2::/64` via Host2 with IP `2001:db8:0::2`

Host1 also acts as a router on OSI layer 3. When one of the network clients tries to contact a target that is specified in Host1's routing table, Host1 will forward the traffic accordingly. It acts as a router for all networks it knows: `2001:db8::/64`, `2001:db8:1::/64`, and `2001:db8:2::/64`.

On Host2, we have nearly the same configuration. Host2's containers will get IPv6 addresses from `2001:db8:2::/64`. Host2 has three routes configured:

- Route all traffic to `2001:db8:0::/64` via eth0
- Route all traffic to `2001:db8:2::/64` via docker0
- Route all traffic to `2001:db8:1::/64` via Host1 with IP `2001:db8:0::1`

The difference to Host1 is that the network `2001:db8:2::/64` is directly attached to Host2 via its docker0 interface, whereas Host2 reaches `2001:db8:1::/64` via Host1's IPv6 address `2001:db8:0::1`.

This way every container can contact every other container. The containers Container1-* share the same subnet and contact each other directly. The traffic between Container1-* and Container2-* will be routed via Host1 and Host2 because those containers do not share the same subnet.

In a switched environment every host must know all routes to every subnet. You always must update the hosts' routing tables once you add or remove a host to the cluster.

Every configuration in the diagram that is shown below the dashed line across hosts is handled by Dockeri, such as the docker0 bridge IP address configuration, the route to the Docker subnet on the host, the container IP addresses and the routes on the containers. The configuration above the line across hosts is up to the user and can be adapted to the individual environment.

4.5.2 Routed Network Environment

In a routed network environment, you replace the layer 2 switch with a layer 3 router. Now the hosts just must know their default gateway (the router) and the route to their own containers (managed by Docker). The router holds all routing information about the Docker subnets. When you add or remove a host to this environment, you just must update the routing table in the router instead of on every host.

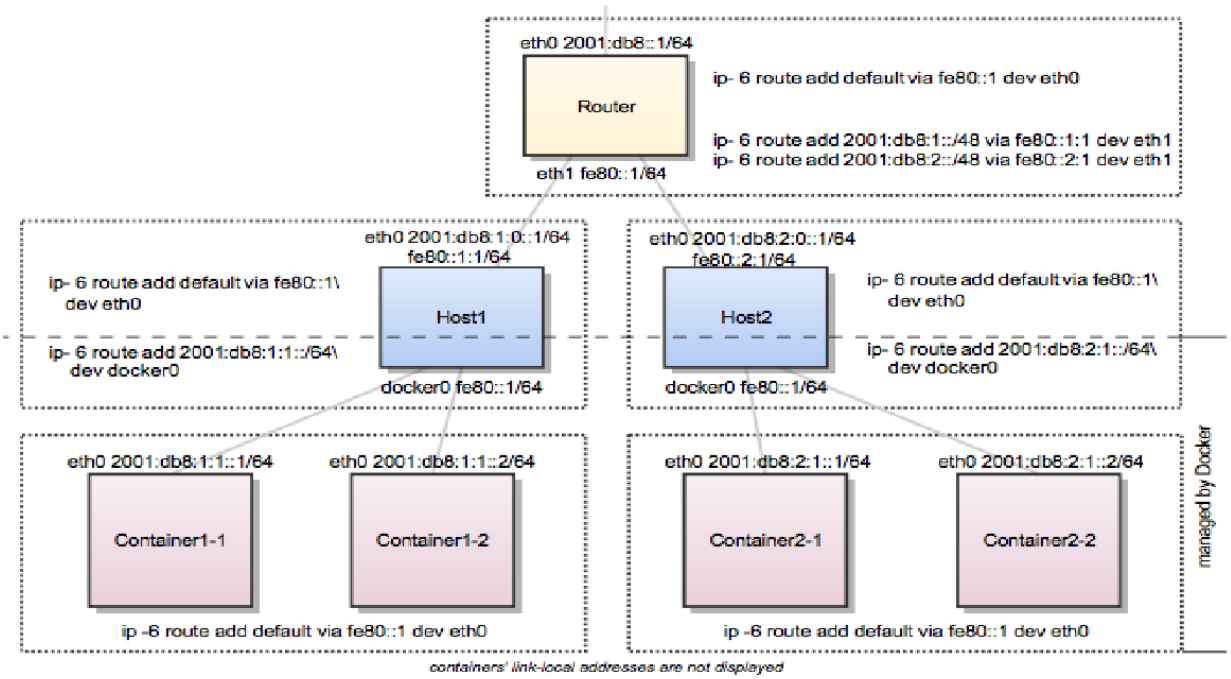


Fig. 3: Figure 2: A Routed Network Environment

In this scenario, containers of the same host can communicate directly with each other. The traffic between containers on different hosts will be routed via their hosts and the router. For example, packet from Container1-1 to Container2-1 will be routed through Host1, Router, and Host2 until it arrives at Container2-1.

To keep the IPv6 addresses short in this example a `/48` network is assigned to every host. The hosts use a `/64` subnet of this for its own services and one for Docker. When adding a third host, you would add a route for the subnet `2001:db8:3::/48` in the router and configure Docker on Host3 with `--fixed-cidr-v6=2001:db8:3:1::/64`.

Remember the subnet for Docker containers should at least have a size of `/80`. This way an IPv6 address can end with the container's MAC address and you prevent NDP neighbor cache invalidation issues in the Docker layer. So if you have a `/64` for your whole environment, use `/76` subnets for the hosts and `/80` for the containers. This way you can use 4096 hosts with 16 `/80` subnets each.

Every configuration in the diagram that is visualized below the dashed line across hosts is handled by Docker, such as the docker0 bridge IP address configuration, the route to the Docker subnet on the host, the container IP addresses and the routes on the containers. The configuration above the line across hosts is up to the user and can be adapted to the individual environment.

4.5.3 References

4.6 Docker IPv6 NAT

4.6.1 What is the Issue with Using IPv6 with Containers?

Initially Docker was not created with IPv6 in mind. It was added later. As a result, there are still several unresolved issues as to how IPv6 should be used in a containerized world.

Currently, you can let Docker give each container an IPv6 address from your (public) pool, but this has disadvantages (Refer to[1]):

- Giving each container a publicly routable address means all ports (even unexposed / unpublished ports) are suddenly reachable by everyone, if no additional filtering is done.
- By default, each container gets a random IPv6 address, making it impossible do DNS properly. An alternative is to assign a specific IPv6 address to each container, but it is still an administrative hassle.
- Published ports won't work on IPv6, unless you have the userland proxy enabled (which, for now, is enabled by default in Docker)
- The userland proxy, however, seems to be on its way out and has various issues, such as:
  - It can use a lot of RAM.
  - Source IP addresses are rewritten, making it completely unusable for many purposes, e.g. mail servers.

IPv6 for Docker can (depending on your setup) be pretty much unusable and completely inconsistent with the way how IPv4 works. Docker images are mostly designed with IPv4 NAT in mind. NAT provides a layer of security allowing only published ports through. Letting container link to user-defined networks provide inter-container communication. This does not go hand in hand with the way Docker IPv6 works, requiring image maintainers to rethink/adapt their images with IPv6 in mind.

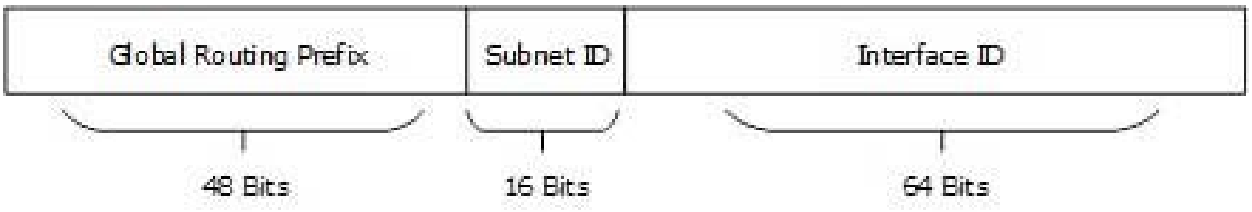
4.6.2 Why not IPv6 with NAT?

So why not try resolve above issues by managing `ip6tables` to setup IPv6 NAT for your containers, like how it is done by the Docker daemon for IPv4. This requires a locally reserved address like we do for private IP in IPv4. These are called in IPv6 as local unicast Ipv6 address. Let's first understand IPv6 addressing scheme.

We note that there are 3 types of IPv6 addresses, and all use last or least significant 64 bits as Interface ID derived by splitting 48-bit MAC address into 24 bits + 24 bits and insert an FE00 hexadecimal number in between those two and inverting the most significant bit to create an equivalent 64-bit MAC called EUI-64 bit. Refer to[2] for details.

1. Global Unicast Address

This is equivalent to IPv4's public address with always 001 as Most Significant bits of Global Routing Prefix. Subnets are 16 opposed to 8 bits in IPv4.



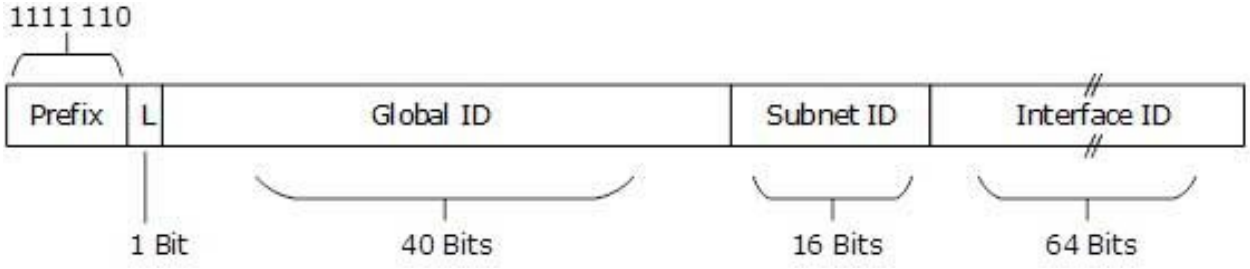
[1] https://github.com/robbertkl/docker-ipv6nat
[2] https://www.tutorialspoint.com/ipv6/ipv6_special_addresses.htm

2. Link-Local Address

Link-local addresses are used for communication among IPv6 hosts on a link (broadcast segment) only. These addresses are not routable. This address always starts with FE80. These are used for generating IPv6 addresses and 48 bits following FE80 are always set to 0. Interface ID is usual EUI-64 generated from MAC address on the NIC.

3. Unique-Local Address

This type of IPv6 address is globally unique & used only in site local communication. The second half of this address contain Interface ID and the first half is divided among Prefix, Local Bit, Global ID and Subnet ID.



Prefix is always set to 1111 110. L bit, is set to 1 if the address is locally assigned. So far, the meaning of L bit to 0 is not defined. Therefore, Unique Local IPv6 address always starts with 'FD'.

IPv6 addresses of all types are assigned to interfaces, not nodes (hosts). An IPv6 unicast address refers to a single interface. Since each interface belongs to a single node (host), any of that node's interfaces' unicast addresses may be used as an identifier for the node(host). For IPv6 NAT we prefer site scope to be within site scope using unique local address, so that they remain private within the organization.

Based on the IPv6 scope now question arises as what is needed to be mapped to what? Is it IPv6 to IPv4 or IPv6 to IPv6 with post? Thus, we land up with are we talking NAT64 with dual stack or just NAT66. Is it a standard that is agreed upon in IETF RFCs? Dwelling into questions bring us back to should we complicate life with another docker-ipv6nat?

The conclusion is simple: it is not worth it and it is highly recommended that you go through the blog listed below[3].

4.6.3 Conclusion

As IPv6 Project team in OPNFV, we recommend that IPv6 NAT is not worth the effort and should be discouraged. As part of our conclusion, we recommend that please do not use IPv6 NAT for containers for any NFV use cases.

4.6.4 References

[3] http://ipv6friday.org/blog/2011/12/ipv6-nat/

Fig. 4: Figure 1: Scope of IPv6 Unicast Addresses