OpenStack Grizzly on IPv6

Version 1.9

Shixiong Shang (shshang@nephos6.com)
Randy Tuttle (randy.m.tuttle@gmail.com)
Ciprian Popoviciu (chip@nephos6.com)
Table of Content

1. INTRODUCTION ..................................................................................................................6
   1.1 MOTIVATION ....................................................................................................................6
   1.2 OBJECTIVES ...................................................................................................................7
   1.3 SCOPE .............................................................................................................................7
   1.4 ONE MORE THING ............................................................................................................7
   1.5 DISCLAIMER ....................................................................................................................8

2. POC OVERVIEW ...................................................................................................................9
   2.1 ARCHITECTURE ...............................................................................................................9
   2.2 HARDWARE AND SOFTWARE MATRIX .........................................................................10
   2.3 IP ADDRESS AND DNS NAME .....................................................................................11

3. PART I: ENABLE IPV6 ON OPENSTACK INFRASTRUCTURE ............................................12
   3.1 PREPARING UBUNTU .......................................................................................................12
      3.1.1 Enable IPv6 .............................................................................................................12
      3.1.2 Jumbo Frames ..........................................................................................................12
   3.2 COMMON NODE ............................................................................................................13
      3.2.1 MySQL ....................................................................................................................13
      3.2.2 Horizon ....................................................................................................................14
      3.2.3 RabbitMQ ................................................................................................................14
   3.3 CONTROLLER NODE ......................................................................................................15
      3.3.1 Nova .........................................................................................................................15
      3.3.2 Cinder .......................................................................................................................16
      3.3.3 Glance ......................................................................................................................17
      3.3.4 Quantum ..................................................................................................................17
   3.4 NETWORK NODE ..........................................................................................................18
   3.5 COMPUTE NODE ..........................................................................................................18
   3.6 CONVERTING TO IPV6 IN PRODUCTION ECOSYSTEMS ............................................18

4. PART II: CREATE YOUR FIRST DUAL-STACK VIRTUAL MACHINE ..................................20
   4.1 TENANT NETWORK AND SUBNETS .............................................................................20
   4.2 NAT AND GARP ............................................................................................................22
   4.3 YOUR FIRST (FAILED) DUAL STACK VIRTUAL MACHINE .............................................23

5. PART III: DRINK AN IPV6 REFRESHMENT ......................................................................24
   5.1 IPV6 ADDRESSING ........................................................................................................24
   5.2 NEIGHBOR DISCOVERY ...............................................................................................25
      5.2.1 Router and Prefix Discovery .....................................................................................26
         5.2.1.1 Router Advertisement .......................................................................................26
         5.2.1.2 Address Auto-configuration .................................................................................26
         5.2.1.3 Default Router .................................................................................................27
List of Figures

FIGURE 1: OVERALL ARCHITECTURE ................................................................. 9
FIGURE 2: ENABLE IPv6 AND JUMBO FRAME ON Ubuntu 13.04 (64-bit) SERVER ................................................................. 13
FIGURE 3: NETSTAT COMMAND OUTPUT ON COMMON NODE ............................................................................................................ 15
FIGURE 4: NETSTAT COMMAND OUTPUT ON CONTROLLER NODE ............................................................................................................ 18
FIGURE 5: QUANTUM NETWORK AND SUBNET CONFIGURATION ............................................................................................................ 21
FIGURE 6: TWO TENANT GATEWAY INTERFACES ................................................................................................................................. 21
FIGURE 7: TENANT NETWORK INTERFACES ................................................................................................................................. 22
FIGURE 8: VIRTUAL MACHINE FAILED TO OBTAIN IPv6 ADDRESS ............................................................................................................ 23
FIGURE 9: TENANT NETWORK INTERFACES AND NAMESPACES ............................................................................................................ 30
FIGURE 10: METHOD “CALL_DRIVER” IN QUANTUM_AGENT.DHCP_AGENT.PY ............................................................................................................ 33
FIGURE 11: METHOD “SPAWN_PROCESS” IN QUANTUM_AGENT.LINUX.DHCP.PY ............................................................................................................ 34
FIGURE 12: MODIFIED METHOD “SPAWN_PROCESS” TO ENABLE RA ............................................................................................................ 35
FIGURE 13: ROUTER NAMESPACE NAME AND QR- INTERFACE NAME ............................................................................................................ 36
FIGURE 14: MODIFIED METHOD “SPAWN_PROCESS” TO MOVE DNSMASQ TO ROUTER’S NAMESPACE ............................................................................................................ 38
FIGURE 15: DNSMASQ PROCESS FOR BOTH IPv4 AND IPv6 SUBNETS ............................................................................................................ 39
FIGURE 16: TRANSLATE MAC TO EUI ................................................................................................................................. 40
FIGURE 17: METHOD “_generate_ipv4v6” IN QUANTUM_DB.DB_BASE_PLUGIN_V2.PY ............................................................................................................ 41
FIGURE 18: OpenStack Calculated Virtual Machine IPv6 Address ............................................................................................................ 42
FIGURE 19: IPv6tables Filter Table and Chains ................................................................................................................................. 43
FIGURE 20: Default FORWARD Chain in Filter Table On Compute Node ............................................................................................................ 44
FIGURE 21: METHOD “_ADD_RULE_BY_SECURITY_GROUP” IN QUANTUM_AGENT.LINUX.IPTABLES_FILEWALL.PY ............................................................................................................ 45
FIGURE 22: Virtual Machine (Centos1) IPv6 Address ................................................................................................................................. 47
FIGURE 23: Virtual Machine (Centos1) IPv6 Reachability To Another Virtual Machine (Centos2) ................................................................................................................................. 47
FIGURE 24: Virtual Machine (Centos1) IPv6 Route ................................................................................................................................. 48
FIGURE 25: Virtual Machine (Centos1) IPv6 Reachability To Default Gateway ................................................................................................................................. 48
FIGURE 26: Modified “_populate_subnet_for_ports” In QUANTUM_DB.L3_DB.PY (Part I) ................................................................................................................................. 50
FIGURE 28: Modified “fixed_ips” Field For DEVICE_OWNER_ROUTER_GW PORT ................................................................................................................................. 52
FIGURE 29: Regular “fixed_ips” Field For ROUTER-INTERFACE PORT ................................................................................................................................. 53
FIGURE 30: Modified “EXTERNAL_GATEWAY_ADDED” Method To Allow Multiple Subnets ................................................................................................................................. 54
FIGURE 31: Dual-Stack External Network Port ................................................................................................................................. 55
FIGURE 32: Modified “EXTERNAL_GATEWAY_ADDED” Method To Avoid NAT And GARP ................................................................................................................................. 56
FIGURE 33: Modified “INTERNAL_NETWORK_ADDED” Method To Avoid SNAT ................................................................................................................................. 56
FIGURE 34: Modified “EXTERNAL_GATEWAY_ADDED” Method To Accept Default Routers ................................................................................................................................. 57
FIGURE 35: IPv6 Default Route In Router’s Namespace ................................................................................................................................. 58
FIGURE 36: Next-hop Device’s IPv6 Interface ................................................................................................................................. 58
FIGURE 37: Disable Learning Of Default Routers On Internal-Network Port ................................................................................................................................. 59
FIGURE 38: Network View ......................................................................................................................................................... 60
FIGURE 39: Virtual Machine (Centos1) IPv6 Reachability To Tenant External Network ................................................................................................................................. 60
FIGURE 40: Virtual Machine (Centos1) IPv6 Reachability To External Network ................................................................................................................................. 61
List of Tables

TABLE 1: HARDWARE AND SOFTWARE MATRIX ......................................................................................................................... 10
TABLE 2: IP ADDRESS AND DNS NAME .................................................................................................................................... 11
TABLE 3: KEYSTONE, MYSQL AND HORIZON IPv6 CONFIGURATION .......................................................................................... 14
TABLE 4: NOVA IPv6 CONFIGURATION ..................................................................................................................................... 16
TABLE 5: CINDER IPv6 CONFIGURATION .................................................................................................................................. 16
TABLE 6: GLANCE IPv6 CONFIGURATION .................................................................................................................................. 17
TABLE 7: QUANTUM IPv6 CONFIGURATION ................................................................................................................................. 18
TABLE 8: TWO APPROACHES ...................................................................................................................................................... 20
1. Introduction

1.1 Motivation

Over time, we have seen that Cloud adoption is gaining significant momentum, driven largely by the need to improve service agility and reduce operational cost; however, this expansion does not come without some pain. Part of that pain comes from the additional strain being placed on an already exhausting IPv4 address space. Clearly, IPv6 is the answer to this pressing need.

While IPv6 adoption is quietly picking up steam, in the Cloud world, we can clearly see OpenStack gaining more and more traction. The opportunity offered by IPv6 to design and implement a new, more scalable and manageable infrastructure is more and more leveraged towards the realization of the next generation IT environment. The promise of open-cloud infrastructure and strong support from developers in the community are leading early adopters to reconsider the way they design public, private, or even hybrid clouds. We see a marriage of these two developments as a significant inflection point offering the following benefits:

- The larger IP address space makes it easier to manage large cloud infrastructures
- The larger IP address space makes it easier to manage the creation, expansion and removal of virtual private cloud environments without changes in policy and easy integration with on-premise enterprise infrastructures
- Neighbor Discovery protocol inherent functionality simplifies implementation and management of large scale broadcast domains
- IPv6 specific provisioning mechanisms can be used to simplify aspects of orchestration
- IPv6 protocol capabilities can make the implementation of VXLANs significantly easier than in IPv4.

**The promise of Cloud cannot be met without IPv6!**

By comparison with the increasing IPv6 adoption across multiple verticals in the industry, very little has been done in OpenStack to form a strong position on IPv6. The first thing we noticed was we had hard time finding many documents that explain how to deploy OpenStack in IPv6 environment with the exception of a single chapter in OpenStack Compute Administrator Guide (Folsom) available on OpenStack (www.openstack.org) web site. Moreover, the blueprint published on the OpenStack web site gives no clear direction regarding how OpenStack will eventually evolve to an IPv6-friendly cloud operating system. Finally, after spending a tremendous time in reading source code, we realized that some code was developed with IPv6 in mind, while other code not.
With both trends evolving rapidly in the industry, the friction between “what is today” and "what will be tomorrow" bothered us. With that frustration, we decided to roll up our sleeves and immerse ourselves in creating a relatively small proof-of-concept (PoC) environment. We have one simple, but ambitious goal: enable OpenStack with IPv6.

1.2 Objectives

By the end of our proof of concept, we hoped that we would achieve the following goals to verify the basic functionalities of IPv6 + OpenStack system:

- All OpenStack infrastructure nodes should be able to communicate with each other by IPv6
- OpenStack should be able to spin up IPv6-enabled virtual machines in multi-tenant environment
- Virtual machine should be able to gain connectivity to existing IPv6 network beyond OpenStack boundary.
- And finally, the latter two should also coexist with IPv4 (i.e. dual-stack).

1.3 Scope

In this first phase of our investigation, we evaluated the following key building blocks (projects) of OpenStack:

- Nova (Compute)
- Glance (Image)
- Cinder (Block Storage)
- Quantum (Network)
- Horizon (Dashboard)
- Keystone (Identity)

We will give more thoughts on other sibling projects and share what we learn in the future phases of our project.

1.4 One More Thing

From initial idea to a functioning system, it was often a bumpy road. Throughout the journey, we encountered various problems, as you will see, in the remainder of this paper. Team members decided to document what we have achieved and share our experience as an attempt to contribute back to the Hint

*These inconsistencies led to the realization that the Grizzly release cannot be deployed in an IPv6 environment and cannot support IPv6 virtual machines out of box.*
community. More importantly, we hope this paper can provoke more thought about the IPv6-OpenStack combination so that eventually, our customers can enjoy the benefits of the two.

1.5 Disclaimer

Please note that, this document is not intended to showcase OpenStack installation or configuration. We assume that the audience already has basic understanding of the key functions and capabilities of OpenStack. Furthermore, we certainly are not implying that the architecture we adopted as part of our proof of concept is the only one that can support IPv6. Lastly, the list of issues we present here should not be considered as comprehensive. It is also worth mentioning that, neither code changes nor workarounds discussed in this paper were targeted at production deployment.
2. POC Overview

2.1 Architecture

The overall architecture of our lab setup in the context of OpenStack is shown in Figure 1.

Figure 1: Overall Architecture

The Controller node serves as the front end for API calls to the compute service (Nova), image service (Glance), volume service (Cinder) and network service (Quantum).

The Network node, hosting multiple quantum plugins and agents, acts as an aggregation point to route traffic within tenant network, and between tenant network and outside world.

The Compute node does the heavy lifting to create virtual machines on-demand using KVM as the hypervisor.
We also added one more node labeled as “Common Node” back in Folsom release as an addition to three-node architecture posted on the OpenStack web site. The intention at that time was to collapse the shared components onto a standalone machine with dedicated resources, so we can analyze the performance of Keystone, MySQL and RabbitMQ more closely.

For this paper, we adopt the same architecture to simply maintain the continuity of the architecture. By doing so, we are neither implying that this is the only architecture that works with IPv6, nor that the issues and fixes are only applicable to the diagram shown above.

It is also worth mentioning that the proof of concept took multi-tenancy into consideration. We have a total of two tenants. Based on design, tenant1 and tenant2 were configured to verify the basic functionality of IPv4-only and dual-stack virtual machines respectively. Tenant2 will be our focus in the following chapters.

### 2.2 Hardware and Software Matrix

All machines are running Ubuntu 13.04 (64-bit) Server edition. Three of them are running as virtual machines and only Compute Node is bare-metal based. The Hardware-Software matrix used is detailed in Table 1.

**Table 1: Hardware and Software Matrix**

<table>
<thead>
<tr>
<th>Node</th>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Node</td>
<td>Virtual machine</td>
<td>Ubuntu 13.04 Server (64-bit)</td>
</tr>
<tr>
<td></td>
<td>2 vCPU, 4GB Memory, 100 GB HD,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 vNIC</td>
<td></td>
</tr>
<tr>
<td>Controller Node</td>
<td>Virtual machine</td>
<td>Ubuntu 13.04 Server (64-bit)</td>
</tr>
<tr>
<td></td>
<td>2 vCPU, 4 GB Memory, 100 GB HD,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 vNIC</td>
<td></td>
</tr>
<tr>
<td>Network Node</td>
<td>Virtual machine</td>
<td>Ubuntu 13.04 Server (64-bit)</td>
</tr>
<tr>
<td></td>
<td>2 vCPU, 4 GB Memory, 100 GB HD,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 vNICs</td>
<td></td>
</tr>
<tr>
<td>Compute Node</td>
<td>Bare Metal server</td>
<td>Ubuntu 13.04 Server (64-bit)</td>
</tr>
<tr>
<td></td>
<td>24 CPUs, 64 GB Memory, 100 GB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HD, 4 NICs</td>
<td></td>
</tr>
</tbody>
</table>
2.3 IP Address and DNS Name

Table 2 captures the DNS name and IP addresses assigned to each node. It is worth mentioning that, the only purpose to preserve IPv4 address here is to continue supporting metadata service as part of virtual machine boot process.

Hint: The metadata service used to enable VM instances can currently be accessed only by sending a request to http://169.254.169.254. No IPv6 equivalent is currently implemented.

Table 2: IP Address and DNS Name

<table>
<thead>
<tr>
<th>Node</th>
<th>DNS*</th>
<th>IPv4 Address</th>
<th>IPv6 Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common</td>
<td>net-common</td>
<td>7.10.180.1/24</td>
<td>2001:7:10:180::1/64</td>
</tr>
<tr>
<td>Controller</td>
<td>net-controller</td>
<td>7.10.180.2/24</td>
<td>2001:7:10:180::2/64</td>
</tr>
<tr>
<td>Compute</td>
<td>net-compute1</td>
<td>7.10.180.4/24</td>
<td>2001:7:10:180::4/64</td>
</tr>
</tbody>
</table>

* Domain name is: sandbox.com
3. Part I: Enable IPv6 on OpenStack Infrastructure

As a first step in our journey to enable IPv6 on OpenStack, we try to document the effort made to enable IPv6 across all OpenStack nodes as illustrated in Figure 1. The overall architecture of our lab setup in the context of OpenStack is shown in Figure 1.

Figure 1. As you can see in the following sections, tweaking relevant parameters in configuration files can largely achieve the enablement of IPv6. By the end of the section, all nodes should be able to communicate with each other by IPv6.

3.1 Preparing Ubuntu

3.1.1 Enable IPv6

Before we started working on any OpenStack related tasks, we had to configure static IPv6 networking on the underlying operating system. On Ubuntu 13.04 (64-bit) server, we simply edited “/etc/network/interfaces” file, and then restart networking (see Figure 2).

3.1.2 Jumbo Frames

The MTU size of the NIC on Ubuntu defaults to 1500 bytes. Large MTU size (i.e., jumbo frames) not only gives you better network performance, but also provides you with workaround for software issues. During our efforts, we encountered a bug on RabbitMQ. Most of OpenStack components, such as Nova, Cinder, Quantum, etc., implements RPC over AMQP. RabbitMQ as AMQP broker sits between two components and allows them to communicate in a loosely coupled fashion. In one of the corner cases, we observed that RabbitMQ received a message close to 1500-bytes without any problem. However, after adding header, the outbound message destined to the L3 Agent exceeded 1500-bytes and became corrupted.

Hint: To enable IPv6 (or dual-stack) compute resources in the tenant networks, it is not mandatory to have the OpenStack infrastructure on IPv6. The infrastructure can be either IPv6 or IPv4 independent of what is implemented for the tenants.

In the future, we hope the OpenStack community will consider ways to reduce the payload size so RPC call between components can work more efficiently. For example, instead of sending information for every router and port, we can consider only sending changed content for the relevant route and port in the event of port addition or deletion.

Hint: It is very important to enable jumbo frames on every node involved in the OpenStack setup, including routers and switches supporting the environment.
3.2 Common Node

3.2.1 MySQL

IPv6 addresses are not supported before MySQL 5.5.3. Fortunately, Ubuntu 13.04 comes with MySQL 5.5.31, which can bind to either IPv4, or IPv6 or both types of addresses depending on the “bind-address” options. Here, we chose the third option.

- If the address is *, the server accepts TCP/IP connections on all server host IPv6 and IPv4 interfaces if the server host supports IPv6, or accepts TCP/IP connections on all IPv4 addresses otherwise. Use this address to permit both IPv4 and IPv6 connections on all server interfaces. This value is the default.
- If the address is 0.0.0.0, the server accepts TCP/IP connections on all server hosts IPv4 interfaces.
- If the address is::, the server accepts TCP/IP connections on all server host IPv4 and IPv6 interfaces. Use this address to permit both IPv4 and IPv6 connections on all server interfaces.
- If the address is an IPv4-mapped address, the server accepts TCP/IP connections for that address, in either IPv4 or IPv6 format. For example, if the server is bound to ::ffff:127.0.0.1, clients can connect using --host=127.0.0.1 or --host=::ffff:127.0.0.1.
- If the address is a “regular” IPv4 or IPv6 address (such as 127.0.0.1 or ::1), the server accepts TCP/IP connections only for that IPv4 or IPv6 address.
3.2.2 Horizon

For Horizon, the trick is allowing Apache to allocate IPv6 sockets and to handle requests sent over IPv6. By default, Apache binds to all addresses on the machine, including IPv6 address. If we want Apache to run exclusively on IPv6 address, then we need to update “Listen” field in “ports.conf” file as you can see in our case.

Hint: Remember that the IPv6 address must be enclosed in square brackets!

3.2.3 RabbitMQ

By default, RabbitMQ will bind to all interfaces, on IPv4 and on IPv6 if available. No additional configuration is required. Table 3 illustrates the changes made.

Table 3: Keystone, MySQL and Horizon IPv6 Configuration

<table>
<thead>
<tr>
<th>Process</th>
<th>Configuration File</th>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keystone</td>
<td>/etc/keystone/keystone.conf</td>
<td>[DEFAULT]</td>
<td>bind_host</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2001:7:10:180::1</td>
</tr>
<tr>
<td>MySQL DB</td>
<td>/etc/mysql/my.cnf</td>
<td>[mysqld]</td>
<td>bind-address</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>::</td>
</tr>
<tr>
<td>Apache</td>
<td>/etc/apache2/ports.conf</td>
<td>Listen</td>
<td>[2001:7:10:180::1]:80</td>
</tr>
</tbody>
</table>

After reboot, we can use netstat tool to verify whether these processes are listening to the right ports as illustrated in Figure 3:

Hint: As a next step, don’t forget to log into MySQL and grant other OpenStack processes the privilege to access the database with corresponding credentials!
3.3 Controller Node

3.3.1 Nova

On the Controller node, a large number of processes cooperate to handle end user API request.

- nova-api accepts and responds to end user compute API calls.
- nova-api-metadata accepts metadata requests from instances.
- nova-consoleauth daemon authorizes user’s tokens that console proxies provide. This service must be running in order for console proxies to work.
- nova-novncproxy (daemon) provides a proxy for accessing running instances through browser-based novnc clients.
- nova-cert daemon manages x509 certificates.

The behavior of the above processes is dictated by a list of available configuration options in /etc/nova/nova.conf file.

Table 4 captures the fields, which should be updated to force these processes to bind to local IPv6 address, with the only exception of metadata service. Metadata service enables virtual machine instances to retrieve instance-specific data such as SSH keys or user data by accessing http://169.254.169.254. It doesn’t make sense to NAT this request to IPv6 destination address, which adds unnecessary overhead for operation. So for now, we chose to stay with IPv4 address and assess the workaround in the future.
Table 4: Nova IPv6 Configuration

<table>
<thead>
<tr>
<th>Process</th>
<th>Functions</th>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>use_ipv6</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>MetaData</td>
<td>[DEFAULT] metadata_listen</td>
<td>7.10.180.2</td>
</tr>
<tr>
<td></td>
<td>VNC</td>
<td>[DEFAULT] novncproxy_host</td>
<td>2001:7:10:180::2</td>
</tr>
</tbody>
</table>

3.3.2 Cinder

We didn’t use Cinder for block storage access, and as such is not evaluated. Since Cinder gained a lot of traction in Grizzly release, we still listed the configuration below in case you plan to replace nova-volume with Cinder in your setup. Table 5 illustrates the Cinder setup.

Table 5: Cinder IPv6 Configuration

<table>
<thead>
<tr>
<th>Process</th>
<th>Configuration File</th>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cinder</td>
<td>/etc/cinder/cinder.conf</td>
<td>[DEFAULT] my_ip</td>
<td>2001:7:10:180::2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[DEFAULT] glance_host</td>
<td>2001:7:10:180::2</td>
</tr>
</tbody>
</table>
3.3.3 Glance

Glance provides services for registering, discovering, and retrieving virtual machine images. It has RESTFUL API that allows querying of virtual machine image metadata as well as retrieval of the actual image. In order to bind RESTFUL API to right IPv6 address, we need to look at both Glance API and Glance Registry servers.

Note that, although various storage backend options are available, we simply store virtual machine images on local disk in our lab setup.

In order to point Glance API server to Glance Registry server running on the same server, we can use either local IPv6 address or DNS name. The former approach led to error message in the log due to malformed request URL. As a workaround, we replaced the address with DNS name resolved to the same IPv6 address to eliminate the error messages.

Table 6 summarizes the Glance configuration.

**Table 6: Glance IPv6 Configuration**

<table>
<thead>
<tr>
<th>Process</th>
<th>Configuration File</th>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glance API</td>
<td>/etc/glance/glance-api.conf</td>
<td>[DEFAULT]</td>
<td>bind_host</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2001:7:10:180::2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>registry_host</td>
<td>net-glance.sandbox.com</td>
</tr>
<tr>
<td>Glance Registry</td>
<td>/etc/glance/glance-registry.conf</td>
<td>[DEFAULT]</td>
<td>bind_host</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2001:7:10:180::2</td>
</tr>
</tbody>
</table>

3.3.4 Quantum

For quantum, we only need to worry about one parameter in quantum.conf file. As you can imagine, it is “bind_host” under “[Default]” section. Table 7 summarizes the Quantum changes.
Table 7: Quantum IPv6 Configuration

<table>
<thead>
<tr>
<th>Process</th>
<th>Configuration File</th>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
</table>

After reboot, we can use netstat tool to verify whether Nova, Glance, Quantum and Cinder processes are listening to the right ports (see Figure 4).

**Figure 4: Netstat Command Output On Controller Node**

```
sudo netstat -tulpn | grep python
```

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Local Address</th>
<th>Port</th>
<th>Flags</th>
<th>Type</th>
<th>State</th>
<th>Remote Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp</td>
<td>0 7.10.180.2:8775</td>
<td>LISTEN 3067/python</td>
<td>LISTEN</td>
<td>LISTEN</td>
<td>0.0.0.0:*</td>
<td>python</td>
</tr>
<tr>
<td>tcp5</td>
<td>0 2001:7:10:180:2:9191</td>
<td>LISTEN 3139/python</td>
<td>LISTEN</td>
<td>LISTEN</td>
<td>:::*</td>
<td>python</td>
</tr>
<tr>
<td>tcp6</td>
<td>0 2001:7:10:180:2:8776</td>
<td>LISTEN 3165/python</td>
<td>LISTEN</td>
<td>LISTEN</td>
<td>:::*</td>
<td>python</td>
</tr>
<tr>
<td>tcp6</td>
<td>0 2001:7:10:180:2:9292</td>
<td>LISTEN 3133/python</td>
<td>LISTEN</td>
<td>LISTEN</td>
<td>:::*</td>
<td>python</td>
</tr>
<tr>
<td>tcp6</td>
<td>0 2001:7:10:180:2:9696</td>
<td>LISTEN 3152/python</td>
<td>LISTEN</td>
<td>LISTEN</td>
<td>:::*</td>
<td>python</td>
</tr>
<tr>
<td>tcp6</td>
<td>0 2001:7:10:180:2:6080</td>
<td>LISTEN 3115/python</td>
<td>LISTEN</td>
<td>LISTEN</td>
<td>:::*</td>
<td>python</td>
</tr>
<tr>
<td>tcp6</td>
<td>0 2001:7:10:180:2:8773</td>
<td>LISTEN 3067/python</td>
<td>LISTEN</td>
<td>LISTEN</td>
<td>:::*</td>
<td>python</td>
</tr>
<tr>
<td>tcp6</td>
<td>0 2001:7:10:180:2:8774</td>
<td>LISTEN 3067/python</td>
<td>LISTEN</td>
<td>LISTEN</td>
<td>:::*</td>
<td>python</td>
</tr>
</tbody>
</table>

### 3.4 Network Node

Since network node is configured with multiple Quantum plugins and agents to handle tenant traffic, the only thing we need to pay attention to here is ensure the consistency of “quantum.conf” file with Controller node as described in the previous section.

### 3.5 Compute Node

Similar to the Network node, the Quantum configuration file must be fully in sync with the one on Controller node. In addition, Nova configuration file must also be consistent with the one on Controller node in order to guarantee nova-compute process to run smoothly.

### 3.6 Converting to IPv6 in production ecosystems

While Part I of this document generally has described making your OpenStack infrastructure work with IPv6, it has not addressed the possibility that some parts of a production OpenStack ecosystem may be using IPv4 with a desire to move to IPv6. And while it’s possible to implement a dual-stack on the underlying Linux platform, we have not proven that OpenStack services can take advantage of the dual-
stack. In fact, we simply migrated ours completely over to IPv6 without regard to the possibility that in a production environment, both might certainly be needed. We believe that in order to perform a migration to IPv6, OpenStack components must take advantage of the dual-stack.

While we certainly believe this is a valid concern, and do not wish to diminish its importance, our intent was to prove that we can make IPv6 work. We will leave this discussion for a later phase of the project and/or further thought within the OpenStack community.
4. Part II: Create Your First Dual-Stack Virtual Machine

4.1 Tenant Network and Subnets

In this section, we will explain how to create networks, subnets, routers and a gateway for tenant network in preparation of instantiating our first dual-stack virtual machine. The configuration for the external network will be elaborated on more in Part VI.

There are two potential ways to mimic dual-stack environment (Table 8). What we did differently in Method 1 was to create two subnets and associate both to the same network. Another approach (i.e. Method 2) adopted one-to-one mapping between network and subnet as we usually did in IPv4 setup. However, we bonded both IPv4 and IPv6 address ranges to the same subnet in hope that, the resulting router port will have two “fixed-ip” fields. However, this attempt didn’t really go very far. We quickly realized that, OpenStack actually created two subnets with identical name, but with unique subnet id. So essentially, it is still the same as Method 1.

Table 8: Two Approaches

<table>
<thead>
<tr>
<th>Method</th>
<th>Tenant</th>
<th>Router</th>
<th>Network Name</th>
<th>Subnet Name</th>
<th>IP Subnet</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>tenant2</td>
<td>router2</td>
<td>net2_tenant</td>
<td>sub2_192_168_2, sub2_2001_192_168_2</td>
<td>192.168.2.0/24, 2001:192:168:2::0/64</td>
</tr>
<tr>
<td>2</td>
<td>tenant2</td>
<td>router2</td>
<td>net2_tenant</td>
<td>sub2_ipv4_ipv6</td>
<td>192.168.2.0/24, 2001:192:168:2::0/64</td>
</tr>
</tbody>
</table>

The quantum commands involved in this configuration (Figure 5) have nothing special beyond what you may already be familiar with. By using “--ip-version”, we clearly called out one subnet is IPv4 based, and another is IPv6 based.
Surprisingly, OpenStack took the commands very smoothly. Both subnets showed up in the namespace as separate internal gateway interfaces as expected as shown in Figure 6 and Figure 7.
4.2  NAT and GARP

When tenant network is created, the L3 agents hold the responsibility of sending GARP (i.e. Gratuitous ARP) to detect IP conflicts and assist in refreshing other nodes’ ARP cache. If the external gateway already exists, the L3 agent also will trigger an update on iptables to insert NAT rules on Network Node.

These default behaviors are not applicable for IPv6. ARP is replaced by Neighbor Discovery in the IPv6 world and IPv6 NAT (i.e. NAT6) is not largely adopted given massive ranges of available addresses.

**Hint:** With IPv6, the Neighbor Discovery protocol inherent to the stack and its intrinsic Duplicate Address Detection mechanism cover the functionality of GARP in IPv4.
We will explain the reason in further detail in the next chapter. To avoid L3 agent from taking actions on IPv6 based tenant subnets, we modified “internal_network_added” method in quantum.agent.l3_agent.py. This method will be discussed in further details in Part VI.

4.3 Your First (Failed) Dual Stack Virtual Machine

At this moment, it appears that we have everything we need to launch our first dual stack virtual machine on IPv6-based OpenStack infrastructure! Isn’t it exciting?

After the virtual machine (Centos 6.4 64-bit) is instantiated, it was only able to obtain an IPv4 address and had IPv4 connectivity. But where is IPv6 address (see Figure 8)?

Figure 8: Virtual Machine Failed to Obtain IPv6 Address

```
root@centos ~]# ifconfig
eth0   Link encap:Ethernet  HWaddr Fa:16:3E:73:83:D9
       inet addr:192.168.2.3 Bcast:192.168.2.255 Mask:255.255.255.0
       inet6 addr: fe00::f016:3eff:fe73:0349/64 Scope:Link
       UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
       RX packets:10 errors:0 dropped:0 overruns:0 frame:0
       TX packets:28 errors:0 dropped:0 overruns:0 carrier:0
       collisions:0 txqueuelen:1000
       RX bytes:2468 (2.4 KiB) TX bytes:2676 (2.6 KiB)
```
5. Part III: Drink an IPv6 Refreshment

An IPv6-based virtual machine can acquire an address either by stateless address auto-configuration, or by DHCPv6. In this section, we will only focus on stateless address auto-configuration. DHCPv6 based address allocation will be evaluated in the future.

5.1 IPv6 Addressing

The driver for the development of IPv6 and the main reason organizations are currently adopting the new protocol is the larger IP address space. While the transition from a 32 bit address to a 128 bit address provided significantly more resources to the new protocol, the changes in the IP address architecture were not only quantitative, they were also qualitative. Considering the importance of the IP address resources it is important to understand and become familiar with the specificities of the new protocol.

IPv6 specific terminology:

- **Type** – IPv6 addresses are Unicast, Multicast and Anycast addresses but not Broadcast addresses. Multicast takes over functionality previously implemented using broadcast. In fact, unlike IPv4, in IPv6 multicast is intrinsic to the operation of the protocol and so, it is important to understand its role in the operation of a cloud environment. For example, for each unicast address there is a multicast address built based on it that is used for Layer2-Layer3 address resolution (solicited-node multicast address)

- **Scope** - IPv6 addresses observe three scopes: Link-Local (relevant only on the link), Unique-Local (relevant within an organization’s administrative domain) and Global. The scope of the address is clearly shown in its formatting

- **Prefix** – is the equivalent of the network ID in IPv4

- **Interface Identifier (IID)** – is somewhat equivalent to the Host ID in IPv4. In IPv6 however, addresses are associated with links not hosts.

**Hint:** OpenStack terminology might not always align consistently with the IPv6 terminology. This is why it is important to become familiar with both.

Prefixes are managed very much as in IPv4 but in general, it is recommended that for a link, the length of the prefix should not be longer than 64 bits (or a /64). This is not a requirement but rather a best practices recommendation.

Interface identifiers can be allocated in many ways:

- Fixed, manual assignment
- Dynamic assignment (DHCP for example)
- Host generated
- EUI-64 based on the interface MAC address (and example is provided in Section 7.1.4)
- Randomly generated (used to minimize the probability a host can be identified by an attacker)
- Cryptographically generated (CGAs are used to protect against spoofing)

These IID options will come up in IPv6 enabled cloud deployments, as many operating systems prefer various mechanisms to generate their addresses.

**Hint:** Since hosts can generate their own IID, all IPv6 addresses assigned to an Interface will go through a Duplicate Address Detection (DAD) process. The address will not become active until the DAD process is successfully completed.

In IPv6 an interface can have multiple prefixes and IP addresses, and unlike IPv4, all of them are primary. All interfaces will have a Link-Local address since this address is used to implement many control plane functions. The IPv6 protocol stack will not become operational on an interface until a link-local address was assigned/generated and it passed DAD verification. At the same time, when you look inside of many data centers, IPv6 is running on self-generated link-local addresses even though no explicit enablement took place.

**Hint:** Link-local addresses are very important in IPv6. For example, the next hope for an IP route, even default gateway, is identified by the link-local address, not the Global address.

Moreover, every IPv6 enabled interface will subscribe to a set of multicast groups that are essential to the control plane operation (solicited-nod for each unicast address, all hosts, all routers, etc).

### 5.2 Neighbor Discovery

Neighbor Discovery is a set of messages and processes defined in RFC 4861 (Neighbor Discovery for IP Version 6) and RFC 4862 (IPv6 Stateless Address Auto-configuration) that solves a set of problems related to the interaction between nodes attached to the same link. It defines mechanisms for solving each of the following problems:

- Router discovery
- Prefix Discovery and Parameter Discovery
- Address Auto-configuration
- Address Resolution
- Next-hop Determination
- Neighbor Unreachability Detection (NUD)
• Duplicate Address Detection (DAD)
• Redirection.

Neighbor Discovery utilizes ICMPv6 and there are a total of five different types of messages:

• Router Solicitation (ICMPv6 type 133)
• Router Advertisement (ICMPv6 type 134)
• Neighbor Solicitation (ICMPv6 type 135)
• Neighbor Advertisement (ICMPv6 type 136)
• Redirect (ICMPv6 type 137)

5.2.1 Router and Prefix Discovery

Router Discovery is used to locate neighboring routers as well as learn prefixes and configuration parameters related to stateless address auto-configuration. Prefix Discovery is the process through which hosts learn the ranges of IP addresses that reside on-link and can be reached directly without going through a router.

5.2.1.1 Router Advertisement

Among all five different NDP messages, two of them are directly involved in Router and Prefix Discovery: Router Advertisement and Router Solicitation.

IPv6-capable routers send unsolicited Router Advertisement periodically out of each interface to “all-nodes link-local scope multicast address” (FF02::1). In addition to providing router address, the Router Advertisement message contains the information required by hosts to determine the link prefixes, the link MTU, and a list of candidate routers for default router selection, etc.

A router can also send solicited Router Advertisement message in response to the receipt of Router Solicitation message. This option proves particularly useful in mobility to accelerate auto-configuration.

5.2.1.2 Address Auto-configuration

Neighbor Discovery also introduces the mechanisms necessary to allow nodes to configure an address for an interface in a stateless manner. It is called Stateless Address Auto-configuration (i.e. SLAAC) as defined in RFC 4862.

There are several flags in Router Advertisement messages determining whether client can use address auto-configuration or not:

   M - Managed Address Configuration. DHCPv6 is available for IPv6 address allocation.
O - Other Configuration. Other configuration information (ex: DNS server IPv6 address) is available through DHCPv6.

For each IPv6 prefix conveyed in Router Advertisement, it has L and A flags:

L - On-Link Flag. The prefix can be used for on-link determination (other IPv6 addresses with the same prefix are on the same L2 subnet).

A - Address Configuration Flag. The prefix can be used for stateless address auto-configuration.

For SLAAC, a router must include at least one prefix with the A-flag set in the Router Advertisement message. When a host receives the Router Advertisement, it will use the prefix and interface MAC address in IEEE’s 64-bit Extended Unique Identifier (EUI-64) format to construct IPv6 address based on SLAAC Privacy Extensions described in RFC 4941.

On the other hand, Dynamic Host Configuration Protocol for IPv6 (DHCPv6) [RFC3315] is used when a site requires tighter control over exact address assignments. Both stateless address auto-configuration and DHCPv6 may be used simultaneously. Due to time constraints, we didn’t have opportunity to explore the approach using DHCPv6. The relative information will be released in a future publication.

Please keep in mind that, one limitation of SLAAC is that, it only occurs for prefixes with a length of 64. Setting the A-flag for any other prefix length is meaningless.

5.2.1.3 Default Router

It is worth mentioning that, in both stateless auto-configuration and DHCPv6 cases, receipt of Router Advertisement will cause virtual machine to install default route pointing to the link-local address of the interface by which router uses to send out Router Advertisement.

**Hint:** This is the ONLY way a virtual machine can auto-configure the default route in IPv6 since DHCPv6 does not (at this time) carry default route information.

5.2.1.4 Duplicate Address Detection

Duplicate Address Detection (i.e. DAD) MUST be performed on all unicast addresses prior to assigning them to an interface, regardless of whether they are obtained through stateless auto-configuration, DHCPv6, or manual configuration, unless DAD is turned off at interface level.

In our environment, we choose Centos 6.4 (64-bit) as virtual machines operating system. There is a potential kernel bug which constantly generated “IPv6 duplicate address detected!” message in syslog
and subsequently not permitting IPv6 address to be assigned to an interface. We had to disable DAD in order to ignore this false-positives alarm. Please see Appendix A for details.

**Hint:** *The IPv6 stack implementations are still maturing. Unlike IPv4, bugs are still going undetected for long periods of time and they affect systems such as OpenStack infrastructure.*

### 5.2.2 Address Resolution and Neighbor Unreachability Detection

Address resolution is the process through which a node determines the link-layer address of a neighbor given only its IP address. Address resolution is performed only on addresses that are determined to be on-link and for which the sender does not know the corresponding link-layer address. It heavily relies on Neighbor Solicitation and Advertisement messages, which is also used for Duplicate Address Detection. The Neighbor Solicitation is the query and contains the target IPv6 address. It is a multicast to the solicited-node multicast group embedding the rightmost 24 bits of the queried IPv6 address. The Neighbor Advertisement is the response and contains the link-layer address of the matching interface.

After the sender has received the Neighbor Advertisement, the neighbor from whom it has been received is considered “reachable”. Monitoring the reachability to the neighbor is the purpose of Neighbor Unreachability Detection (NUD).

As you can imagine, Address Resolution is the counterpart of ARP in IPv4 world.

### 5.2.3 Redirect

IPv6 Redirect works in a very similar way as redirect in IPv4. Router sends a single redirect message to inform hosts of a better next hop, whether another router or the final destination itself, should it be on-link. The redirect message simply contains the IP address that serves as better choice of next hop and the IP address of the destination.
6. Part IV: What Went Wrong?

After the brief introduction of how neighbor discovery works in IPv6, let us take a look at what is missing in OpenStack Grizzly Release.

6.1 IP Address Assignment In Tenant Network

There are several caveats in the implementation preventing virtual machine from obtaining IPv6 address.

- In the Grizzly release, the Quantum dhcp agent (quantum-dhcp-agent) provides DHCP services to tenant networks. The default, also the only driver, launches dnsmasq, which is a lightweight DHCP server in the backend. The default setting of dnsmasq is tailored to support IPv4 DHCP only.
- In the multi-tenancy environment, OpenStack uses Linux namespace on network node to separate traffic from tenants and to avoid overlapping subnets. By doing so, for each tenant network, OpenStack will create a namespace beginning with “qdhcp-“and dnsmasq process will run inside that namespace; In the meantime, the tenant router will reside in the different namespace beginning with “qrouter-“, which contains default gateways for virtual machines attached to the tenant network (i.e. “qr-“ interfaces) and an exit point to external networks (i.e. “qg-“ interfaces).

The compound impact of these caveats leads to the following limitations in Grizzly release:

- The dnsmasq process will not send Router Advertisement even when IPv6-based tenant network is supported.
- Even if the dnsmasq process could send Router Advertisement, the default gateway would bind to its own link-local address in the qdhcp- namespace. As a result, traffic leaving tenant network will be drawn to DHCP interface, instead of gateway port on router. That is not desirable!

In summary, the dnsmasq process should bind to default gateway (i.e. “qr-“) interface in router’s namespace and send Router Advertisement from there (see Figure 9). There is one side effect of this, which is that the Network Node sees the Router Advertisement on the tenant’s router interface too. As a result, the tenant router automatically installs a default route to its own local interface. This is not a desired behavior as it can cause routing issues. It is discussed later in Part VI.
6.2 IPtables vs. IPv6Tables

IPtables is a user space application that allows system administrator to configure Linux kernel firewall by tables, chains and rules. IPtables has four predefined tables: Filter, NAT, Mangle, and RAW. Each table contains a number of built-in chains together with user-defined chains. Each chain is composed of a list of rules and each rule specifies what to match within a packet and what action should take on such a packet.
When OpenStack starts, it creates several OpenStack specific chains and attaches them to the built-in chains in Filter table. When virtual machine instances are created, additional instance-specific chains are inserted to accept or drop virtual machine traffic in both directions. By default, source-based NAT (SNAT) rules also show up in NAT table so outbound traffic from virtual machine will be NATed to router’s external network IP; If floating IP address is assigned, then rules for destination-based NAT (DNAT) are also populated in NAT table.

IPtables only works on IPv4. For IPv6, the counterpart is IP6tables. Currently Grizzly release code handles IP6tables in the same way as IPv4. Considering how Neighbor Discovery works, we need to make sure Router Advertisement, which largely relies on ICMPv6 and multicast, can hit virtual machine without being silently dropped by IP6table rules on Compute Node.

In addition, with IPv6, you will get a vast address space which gives you room to divide your network into many subnets. For example, with a /48 subnet mask, you can create 65536 subnets, each with 64 bit addresses. That makes NAT in IPv6 unnecessary and undesirable.

In summary, Router Advertisement should be allowed by Linux firewall rules all the way from Network Node to Compute Node, and eventually be delivered to virtual machine.

**Hint:** NAT should be turned OFF so traffic from IPv6 tenant network to IPv6 external network is purely routed. No network address translation should occur.

### 6.3 Routing to External Network

Up to this point, we have only examined the pitfalls within the tenant network. What about the external network? Can we simulate dual stack on qg- interface facing the outside world? Can Linux namespace handle IPv6 routing natively?

In the OpenStack Grizzly release, Quantum allows setting of the gateway on the router by assignment of the network (VLAN) defined as “external”\(^1\). This gateway, by its very name, is the gateway for all traffic from and/or to the tenant.

Furthermore, this external network should also contain a subnet that is also defined on the next-hop device, which has a presence in this subnet. When the gateway is set, Quantum creates and associates an interface to the “qrouter-” namespace where this network can be “attached”.

Now generally, OpenStack/Quantum does not support multiple subnets, and therefore multiple IP addresses, on the same interface. So allowing IPv4 and IPv6 to coexist on the same network is

\(^1\) Achieved by adding the option “--router:external=True” with the “quantum net-create” command.
somewhat problematic for supporting a dual-stack OpenStack ecosystem. This is particularly true on the router external gateway (qg- interface). We explain in the following paragraphs.

Let’s juxtapose, for a moment, what goes on for internal networks versus external networks. When an internal network is created, it is merely a shell, or a container, for holding subnets. It’s not attached to any routers. It’s essentially useless until we add a subnet to it and subsequently attach it to a router. The same is true for the external network. The key differences lie in the way these are associated with the router, and, more importantly, how their interfaces are created.

When an internal network is associated with the router, the subnet is specified. This essentially creates an interface for the subnet and associates it with that router’s namespace. So, each subnet, and therefore IP address, has it’s own interface. Moreover, even if multiple subnets exist in the same internal network, the association is made by subnet, and as a result, creates a single interface for each subnet.

On the other hand, and as mentioned previously, an external network is specified when setting the gateway. This association creates an interface in the router’s namespace regardless of the number of subnets contained within the external network. The issue here is that Quantum ignores any more than one subnet in the external network. Herein lies the issue with supporting a dual-stack ecosystem.

One might argue then, why not two external networks? Create one with an IPv4 subnet and the other with an IPv6 subnet. Then each of these can be used to set independent gateways on the router.

The issue with this approach is that Quantum does not handle this well either. Each L3 agent can be associated with at most one external network. If multiple external networks are defined, Quantum does allow you to specify which of the multiple external networks provide external access by explicitly setting it in the l3_agent.ini file (gateway_external_network_id). But for and IPv4 and IPv6 OpenStack ecosystem, they both must be considered gateways to their respective networks.

**Hint:** To work around this, we focused our solution on allowing multiple subnets to be assigned to a single interface when the network is external.
7. Part V: Make Your First Dual-Stack Virtual Machine Work

7.1 IPv6 Address Assignment

7.1.1 Start Dnsmasq for IPv6

On Ubuntu 13.04, OpenStack source code is stored in “/usr/share/pyshared” directory. Among those python code, there are two files directly related to quantum dhcp agent on Network Node: `quantum.agent.dhcp_agent.py` and `quantum.agent.linux.dhcp.py`

The first file defines DhcpAgent class and its corresponding methods to start, stop a quantum dhcp agent and maintain its state for each tenant network. The hook between dhcp agent and dnsmasq process is a method named “call_driver” under DhcpAgent class, which invoke an action on DHCP driver instance (see Figure 10).

```
def call_driver(self, action, network):
    """Invoke an action on a DHCP driver instance."""
    try:
        # the Driver expects something that is duck typed similar to
        # the base models.
        driver = self.dhcp_driver_cls(self.conf,
                                      network,
                                      self.root_helper,
                                      self.device_manager,
                                      self._ns_name(network),
                                      self.dhcp_version)
    
    getattr(driver, action)()
    return True
```

Figure 10: Method “call_driver” in quantum.agent.dhcp_agent.py

When we move onto “dhc.py” file, the critical method is “spawn_process” under class Dnsmasq. The method is called directly and indirectly every time when dnsmasq process is started or reloaded.

In respect of the efforts made by other developers, we decided not to change the main body of this method, which works perfectly fine for IPv4 based virtual machine. Instead, we added “if” and “elif” statements inside “for” loop to intentionally address IPv4 and IPv6 scenarios separately. For the rest of this chapter, other code changes are primarily under “elif” line (see Figure 11).
First, let us enable Dnsmasq to send Router Advertisement to each of the subnets where IPv6 addresses are attached. To achieve this goal, dnsmasq must be spawned up with both of the following options, either as part of command line or in the configuration file:

- **enable-ra**: This option tells dnsmasq to advertise the IPv6 prefix included in each “--dhcp-range”. By default, the M bit (Managed Address Configuration) is set and A bit (Address Configuration Flag) is reset. This can be overridden by mode keywords in each individual DHCP address range.

- **dhcp-range**: This option should include available IPv6 address pool range associated with network. The optional “mode” parameter may be a combination of ra-only, slaac, ra-names, ra-stateless:
  - ra-only: dnsmasq will send only Router Advertisement on this subnet, but it doesn't provide DHCPv6
  - slaac: dnsmasq will offer Router Advertisement on this subnet with A-bit set, so client can have a calculated IPv6 address using SLAAC algorithm.
  - ra-stateless: dnsmasq will send Router Advertisement with both O-bit and A-bit set, and provide DHCPv6 stateless service. Client can use SLAAC to auto-generate IPv6 address and use DHCP to retrieve other configuration.

The final dnsmasq command triggered by OpenStack is composed in “spawn_process” method. In our configuration, we added “--enable-ra” option and set “mode” to “slaac”. In addition, we also commented out the line of “--dhcp-hostsfile” since it is only relevant to DHCP approach (see Figure 12).
Figure 12: Modified method “spawn_process” to enable RA

```python
cmd = [
    'dnsmasq',
    '--no-hosts',
    '--no-resolv',
    '--strict-order',
    '--bind-interfaces',
    '--interface=%s % gateway_interface_name,
    '--except-interface=lo',
    '--pid-file=%s % self.get_conf_file_name('ipv6pid', ensure_conf_dir=True),
    # '--dhcp-hostsfile=%s % self._output_ipv6hosts_file(),
    '--dhcp-optfile=%s % self._output_opts_file(),
    '--dhcp-script=%s % self._lease_relay_script_path(),
    '--leasefile-ro',
    '--enable-ra',
]
mode = 'slaac'
```

7.1.3 Move To Router Namespace

The next challenge we encountered is how to move dnsmasq process to the namespace where router and gateway interfaces are hosted. As explained previously in Chapter 6, when dnsmasq is called, it only has visibility to network and its subnets. Namespace, as part of router information, is not passed to the method.

**Hint:** From an end-user perspective, neither “quantum net-show” nor “quantum subnet-show” command returns namespace value explicitly.

After taking a closer look, we realized a couple of things that will help us derive the router’s namespace:

- A router port contains a field called “device_id”. A special kind of route port, which serves as the gateway for the tenant network, has “device_id” value identical to the id of the router to which it is attached.
- A tenant router’s namespace is constructed by adding “qr-“ prefix to the router’s id.
- The tenant gateway router port appears in router namespace with the interface name built by prepending “qr-“ prefix to the first 5-byte of its port-id.

Figure 13 illustrates our observations.
With these patterns in mind, we modified “spawn_process” method in `quantum.agent.linux.dhcp.py` as below. The logic is very simple:
1. Search all ports available on the network
2. If a port belongs to the interested subnet, AND its IP is equal to the subnet’s gateway IP, then:
   - Use this port’s device id to build router’s namespace, and save the value to a string called “router_namespace”
   - Use this port’s id to build interface name inside router’s namespace, and save the value to a string called “gateway_interface_name”

Later on, when dnsmasq command is invoked for IPv4 subnet, continue to use DHCP namespace as usual. When dnsmasq command is invoked for IPv6 subnet, not only we use router’s namespace, but also, we only bind dnsmasq to the gateway interface for that specific subnet.

The code segment is show in Figure 14.
Figure 14: Modified method “spawn_process” to move dnsmasq to router’s namespace

def spawn_process(self, subnet, port)
    if subnet.ip_version == 6:
        for port in self.network.ports:
            for fixed_ips in port.fixed_ips:
                if fixed_ips.subnet_id == subnet.id and fixed_ips.ip_address == subnet.gateway_ip:
                    router_namespace = 'qrouter-' + str(port.device_id)
                    gateway_interface_name = 'qr-' + str(port.id)[11]

                    cmd = ['dnsmasq', '--no-hosts', '--strict-order', '--bind-intfaces',
                          '--interfaces=%s %gateway_interface_name, --except-interface=lo',
                          '--pid-file=%s %self.get_conf_file_name(
                           'ipv6pid', ensure_conf_dir=True),
                          #'--dhcp-hostsfile=%s %self.output_ipv6hosts_file(),
                          '--dhcp-optfile=%s %self.output_opts_file(),
                          '--dhcp-script=%s %self.lease Relay_script_path(),
                          '--leasefile-ro',
                          '--enable-ra',
                      ]
                    mode = 'slaac'

                    cmd.append('--dhcp-range=%s,%s,%s,%s' %
                        (set_tag, self._TAG_PREFIX % i,
                        netaddr.IPNetwork(subnet.cidr).network,
                        mode,
                        self.conf.dhcp_lease_time))

                if self.namespace:
                    ip_wrapper = ip_lib.IPWrapper(self.root_helper, router_namespace)
                    ip_wrapper.netns.execute(cmd, add_env=env)
                else:
                    # For normal sudo prepend the env vars before command
                    env = ['%s=%s' % pair for pair in env.items()]
                    cmd = ['%s=%s' % pair for pair in env.items()] + cmd
                    self.execute(cmd, self.root_helper)

With this code change (see Figure 15), you will see two dnsmasq processes for the same network, one for IPv4 subnet in qdhcp- namespace, and one for IPv6 subnet in qrouter- namespace; Dnsmasq also starts sending Router Advertisement with IPv6 prefix to the subnet.
7.1.4 Generate SLAAC IPv6 Address

At this moment, if there are no IPTABLES rules blocking Router Advertisement, virtual machine should be able to automatically formulate its own IPv6 address by marrying 64-bit IPv6 prefix learned from Router Advertisement and 64-bit interface identifier in EUI-64 format without the need of manual configuration and DHCPv6. This is a default behavior for many operating systems however, as discussed in section 5.1, there are multiple ways in which the Interface Identifier can be generated.

RFC2327 defines a two-steps process to translate interface 48-bit interface MAC address to 64-bit extended unique identifier (EUI). Let’s review the process here.
The first step is to convert 48-bit MAC address to 64-bit value. As you may already know, a MAC address is composed of Organizationally Unique Identifier (OUI) and the NIC specific part. Each is 24-bit long. A 16-bit hex value 0xFFFE is inserted between these two halves to form a 64-bit value.

The second step is to invert the universal/local (U/L) flag, the 7th bit, in the OUI portion of the address. Globally unique addresses should have this bit set to zero and local addresses should have this bit set to one. This can be seen in Figure 16.

Figure 16: Translate MAC to EUI

Assuming IPv6 address is associated to the local interface, does that mean virtual machine can reach its default gateway now? The answer is NOT YET!

Remember that iptables manages the forwarding of packets to and from instances, forwarding floating IP traffic and enforcing security group rules. A virtual machine is completely isolated until iptables opens up the ports on the Compute Node to allow bi-directional traffic pass through. This is true for IPv4, and it is also true for IPv6.

The problem is deeply rooted in OpenStack’s implementation of fixed IP address allocation. After current available IP address in fixed IP address pool is assigned to a virtual machine, the next available address is simply incremented by one. The python method that dictates this behavior is called “_generate_ip” in quantum.db.db_base_plugin_v2.py file on Controller Node.

The approach described above works beautifully for IPv4, but as you can tell, it is worlds apart from the way a virtual machine constructs its own IPv6 SLAAC address. As a result, OpenStack updates IP6Tables to allow traffic based on value X, but virtual machine is actually using Y. As you’d expect, it does not work, does it?
The idea here is, if the IPv6 address picked by OpenStack is identical to the IPv6 SLAAC address calculated by virtual machine independently, then we can let OpenStack naturally handle the IP6Table filter rules with minimum code change. That is what we are going to achieve in the rest of the section.

Before we started, we had noticed that one thing is missing from the input to default “/_generate_ip” method: the MAC address of the interface. Before invoking “/_generate_ip”, method “/_allocate_ips_for_port” actually creates a unique MAC address for the virtual machine already. To allow system to calculate IPv6 address based on interface MAC, we modified original “/_generate_ip” method and created a new one called “/_generate_ip_v4v6”. Figure 17 shows these changes.

Figure 17: Method “/_generate_ipv4v6” in quantum.db.db_base_plugin_v2.py

```python
if subnet['ip_version'] == 6:
    cidr = subnet['cidr']
    tmp = cidr.split(':')
    subnet_prefix = ':'.join(tmp[:4])

    tmp = mac_address.split(':')
    tmp0_int = int("0x" + tmp[0], 16) & 253
    tmp0_hex = hex(tmp0_int)[2:]
    eui64 = ':'.join(new_tmp)

    ip_address = subnet_prefix.lower() + ':' + eui64.lower()

    return {'ip_address': ip_address, 'subnet_id': subnet['id']}  
```

For the port being created, if the target subnet is IPv4 based, then execute the original code to return the IP address and subnet id; if the target subnet is IPv6 based, then:

1. Use CIDR to create /64 subnet prefix
2. Divide interface MAC to two halves
3. Reset the 7th bit in the first OUI byte to zero
4. Insert 0xfffe between two halves, and create two-byte groups separated by “:" to formulate interface identifier
5. Prepend /64 subnet prefix to the interface identifier

For the virtual machine with MAC equal to “fa:16:3e:73:83:d9”, the calculated IPv6 address with prefix “2001:192:168:2::/64” is shown in the screen shot below (see Figure 18):
### 7.2 IP6Table Rules

Since Network Node is where routing takes place, the IPTABLES\(^2\) rules created by OpenStack primarily focus on address manipulation to support SNAT and DNAT between tenant network address space and external network address space. Very little is done on Network Node to filter packets at either direction.

**Hint:** As a matter of fact, OpenStack only inserts empty chains in Filter Table and allows everything to go through by default.

Ip6table rules on Compute Node, however, serve different purpose. Since Compute Node is where virtual machines are hosted, the ip6table rules are added to secure virtual machines by filtering both inbound and outbound traffic based on built-in rules and rules inherited from security group.

As we mentioned in the previous chapter, our primary goals here is making sure Router Advertisement announced by dnsmasq on Network Node to be delivered to virtual machines on Compute Node. So as you may tell already, the code changes discussed in the next section was on Compute Node.

#### 7.2.1 Filter Table

Similarly to IPtables, IP6Tables also has four built-in tables, each with several built-in chains. Please note that, in Grizzly release, OpenStack only uses Filter table and NAT table.

The Filter table is responsible for packet filtering. It has three built-in chains:

- **INPUT**: Used to filter packets destined to local server.
- **OUTPUT**: Used to filter packets generated locally and going out of the local server.
- **FORWARD**: Used to filter packets routed through the local server via another NIC.

\(^2\) IPTABLES (uppercase) refers generically to the policies implemented in either iptables or ip6tables in Linux.
As you can see in Figure 19, OpenStack also creates several chains beginning with “quantum-openvswi-” and hooks them up to the built-in chains at the time when OpenStack starts; When a virtual machine instance is launched, the instance specific chains are also attached to “quantum-openvswi-sg-chain”. One of them works on inbound direction to virtual machine with name beginning with “quantum-openvswi-i”. Another works on outbound direction from virtual machine with name beginning with “quantum-openvswi-o”.

At the Compute Node level, ip6tables doesn’t really drop anything in both INPUT and OUTPUT chains given this structure. Although “quantum-openvswi-o” chain is also dotted lined to INPUT chain, it only works on the packets leaving virtual machine’s tap interface into Linux userspace, not on the packets entering compute host from outside world (Figure 20).
The rules to allow Neighbor Discovery related packets should be available to all virtual machines and should only be inserted to ip6tables on Compute Node. The same rules become unnecessary on Network Node given the fact that all outbound packets are already allowed.

**Hint:** We believe the best place for us to add more rules is at level of virtual machine specific chains.

With that being said, we modified the method called “\_add\_rule\_by\_security\_group” in `quantum.agent.linux.iptables__firewall.py` on Compute Node (see Figure 21).
After restarting quantum-plugin-openvswitch-agent service, the `ip6tables` showed a couple of new entries, which should allow Router Solicitation, Router Advertisement, Neighbor Solicitation, Neighbor Advertisement, and Redirect messages, all running above ICMPv6. Furthermore, you may also notice that virtual machine’s MAC address and IPv6 address are being used to match traffic:
7.2.2 Your First (Successful) Dual Stack Virtual Machine

Switching back to the virtual machine instance we launched previously, it now shows IPv6 address identical to the value calculated by OpenStack and has gained reachability to its default gateway with IPv6 address “2001:192:168:2::1” and link local address “fe80::f816:3eff:fe7f:6ba6”! The following four (4) figures (Figure 22 through Figure 25) illustrate our connectivity to our default gateway.
Figure 22: Virtual Machine (Centos1) IPv6 Address

```
[root@centos1 ~]# ifconfig
eth0   Link encap:Ethernet  HWaddr FA:16:3E:73:83:D9
       inet addr:192.168.2.3  Bcast:192.168.2.255  Mask:255.255.255.0
       inet6 addr: fe80::f816:3eff:fe73:83d9/64 Scope:Link
       UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
       RX packets:303 errors:0 dropped:0 overruns:0 frame:0
       TX packets:90 errors:0 dropped:0 overruns:0 carrier:0
       collisions:0 txqueuelen:1000
       RX bytes:46067 (44.9 KiB) TX bytes:8924 (8.7 KiB)
```

Figure 23: Virtual Machine (Centos1) IPv6 Reachability to Another Virtual Machine (Centos2)

```
[root@centos1 ~]# ping6 2001:192:168:2:f816:3eff:fe11:1741
64 bytes from 2001:192:168:2:f816:3eff:fe11:1741: icmp_seq=1 ttl=64 time=1.59 ms
64 bytes from 2001:192:168:2:f816:3eff:fe11:1741: icmp_seq=2 ttl=64 time=0.797 ms
64 bytes from 2001:192:168:2:f816:3eff:fe11:1741: icmp_seq=3 ttl=64 time=0.770 ms
64 bytes from 2001:192:168:2:f816:3eff:fe11:1741: icmp_seq=4 ttl=64 time=0.694 ms
64 bytes from 2001:192:168:2:f816:3eff:fe11:1741: icmp_seq=5 ttl=64 time=0.560 ms
64 bytes from 2001:192:168:2:f816:3eff:fe11:1741: icmp_seq=6 ttl=64 time=0.667 ms
64 bytes from 2001:192:168:2:f816:3eff:fe11:1741: icmp_seq=7 ttl=64 time=0.765 ms
64 bytes from 2001:192:168:2:f816:3eff:fe11:1741: icmp_seq=8 ttl=64 time=0.753 ms
```
Figure 24: Virtual Machine (Centos1) IPv6 Route

```
[root@centos1 ~]# route -n -A inet6
Kernel IPv6 routing table
Destination            Flags Metric Ref    Use Iface          Next Hop
2001:192:168:2::/64    U     256    105    0 eth0
fe80::/64              U     256    0      0 eth0
::/0                   U     1024   29     0 eth0
::1/128                U     0      9      1 lo
fe80::f816:3eff:fe73:83d9/128        U     0      3      1 lo
ff02::1/128            UC     0      105   0 eth0
ff00::/8               U     256    0      0 eth0
```

Figure 25: Virtual Machine (Centos1) IPv6 Reachability to Default Gateway

```
[root@centos1 ~]# ping6 2001:192:168:2::1
64 bytes from 2001:192:168:2::1: icmp_seq=1 ttl=64 time=4.48 ms
64 bytes from 2001:192:168:2::1: icmp_seq=2 ttl=64 time=0.738 ms
64 bytes from 2001:192:168:2::1: icmp_seq=3 ttl=64 time=0.640 ms
64 bytes from 2001:192:168:2::1: icmp_seq=4 ttl=64 time=0.629 ms
64 bytes from 2001:192:168:2::1: icmp_seq=5 ttl=64 time=1.15 ms
64 bytes from 2001:192:168:2::1: icmp_seq=6 ttl=64 time=0.590 ms
64 bytes from 2001:192:168:2::1: icmp_seq=7 ttl=64 time=0.727 ms
64 bytes from 2001:192:168:2::1: icmp_seq=8 ttl=64 time=0.617 ms
```
8. Part VI: Get me off this [IPv6] Island

Throughout this document, we’ve been discussing how to implement an IPv4 and IPv6 OpenStack environment where both network versions can coexist. But without access to the external world over both network protocols, the tenant’s VMs can become stranded, much like Gilligan from the famous TV show “Gilligan’s Isle”. So what do we do to get off the IPv6 Island? Answer: Build an IPv6 bridge (no, not a networking bridge).

Unlike a tenant network, adding a router gateway to an external network is done by attaching the network to the router. Since OpenStack allows multiple subnets per network, we'd expect that the interface should also support multiple IP address, and in particular, IPv6 and IPv4. However, this is not the case. So one of our focuses in this chapter is to address the issue of multiple subnets per interface, particularly for external networks.

We also felt it was necessary to defeat NAT as we believe that with IPv6 tenant NATing was unnecessary given the large address space that IPv6 offers. Moreover, as NAT was not needed for IPv6 addresses, we do not create an allocation pool for VM instances to utilize for purposes of floating IP addresses.

While we contend that the above two goals should be possible, several code changes are required on both the quantum-server (Controller Node, l3_db.py) and the quantum-l3-agent (Network Node, l3_agent.py).

8.1 First things first

For the explanation that follows, we have placed an IPv4 and an IPv6 subnet in the same external network. Based on current code, quantum-server skipped subnet information in the packet addressed to L3 agent simply because it doesn’t know how to construct the “subnet” field for the port with more than one subnet in it. However, subnet information, including ip_version, cidr, ip_address and gateway_ip, is essential for L3 agent to build dual-stack interface inside router’s namespace.

To make this work, we first needed to change how the quantum-server sends subnet details to the quantum-l3-agent instances. In the method called “_populate_subnet_for_ports” in quantum.db.l3_db.py on Controller Node, we altered the code to:

- Allow multiple IP addresses per port when device_owner is network:router_gateway.
- Convey additional information such as cidr, ip_version, and gateway_ip to be sent along with subnet_id and ip_address in the fixed_ips section of the payload.
We considered the possibility of preventing multiple (>1) IPv4 or multiple (>1) IPv6 address, and only allowing one IPv4 and one IPv6. We think this may be the correct way, but we did not code it this way in the l3_agent as we contend this should be done by the quantum-server, thereby, preventing the request from even getting to the l3_agent(s).

Figure 26 shows this portion of our changes to this method.

**Figure 26: Modified “_populate_subnet_for_ports” in quantum.db.l3_db.py (Part I)**

```python
def _populate_subnet_for_ports(self, context, ports):
    """Populate ports with subnet.
    """
    if not ports:
        return
    subnet_id_ports_dict = {}
    for port in ports:
        dev_owner = port.get('device_owner', None)
        fixed_ips = port.get('fixed_ips', [])
        if len(fixed_ips) > 1:
            # Skip multiple IPs on router ports except gateway ports
            if (dev_owner != DEVICE_OWNER_ROUTER_GW):
                LOG.info(_("Ignoring multiple IPs on router port %s"),
                        port['id'])
                continue
        if not fixed_ips:
            # Skip ports without IPs, which can occur if a subnet
            # attached to a router is deleted
            LOG.info(_("Skipping port %s as no IP is configure on it"),
                     port['id'])
            continue
        for fixed_ip in fixed_ips:
            my_ports = subnet_id_ports_dict.get(fixed_ip['subnet_id'], [])
            my_ports.append(port)
            subnet_id_ports_dict[fixed_ip['subnet_id']] = my_ports
```

For the second goal, you may notice in the second part of “_populate_subnet_for_ports” method that we only do this for network:router_gateway. Other subnet info is left alone (although we do send ip_version now too) and sent as it typically is sent; that is, in the "subnet" section of the port info. Ideally, all of it is packaged together to minimize payload. This is the extent of the changes here.

**Hint:** It is our opinion that it would be preferable to perform semantic checking earlier rather than sending the data to the agent where checking is performed later.
Figure 27 shows the changes made, especially when the device_owner is a router gateway.

**Figure 27: Modified “*_populate_subnet_for_ports*” in quantum.db.l3_db.py (Part II)**

```python
for subnet_dict in subnet_dicts:
    ports = subnet_id_ports_dict.get(subnet_dict['id'], [])
    for port in ports:
        dev_owner = port.get('device_owner', None)
        if (dev_owner != DEVICE_OWNER_ROUTER_GW):
            port['subnet'] = {'id': subnet_dict['id'],
                              'cidr': subnet_dict['cidr'],
                              'ip_version': subnet_dict['ip_version'],
                              'gateway_ip': subnet_dict['gateway_ip']}
        else:
            fixed_ips = port.get('fixed_ips', [])
            x = 0
            while (x < len(fixed_ips)):
                filters = {'id': [fixed_ips[x]['subnet_id']]}
                fields = ['id', 'cidr', 'gateway_ip', 'ip_version']
                subnet = self.get_subnets(context, filters, fields)
                port['fixed_ips'][x] = {'subnet_id': fixed_ips[x]['subnet_id'],
                                        'ip_address': fixed_ips[x]['ip_address'],
                                        'cidr': subnet[0]['cidr'],
                                        'ip_version': subnet[0]['ip_version'],
                                        'gateway_ip': subnet[0]['gateway_ip']}
                x += 1
```

In Figure 28, you can see that for a port with router_gateway as device_owner, we included the additional information in the fixed_ips section of the payload. Specifically, we’ve allowed multiple IPs on router ports that are DEVICE_OWNER_ROUTER_GW type. Then, we make sure the information that will be used by the L3 Agent is populated.
When it is not a router gateway (i.e., a router interface), we basically leave it alone. We did, however, add cidr, and IP version as Figure 29 illustrates.
Once that has been done, we are ready to have the payload to be processed by the appropriate L3 agent(s).

### 8.2 The L3 Agent takes over

In the L3 agent of the Network Nodes, we needed to make several changes, many of which were due to the way the subnet information is presented from quantum-server as mentioned previously. Here, we are principally concerned with allowing multiple IP addresses on the same interface, and preventing NAT for IPv6 networks, but still allowing it to continue for IPv4. So, many of the code changes were checks for IP version, which we are now providing from the quantum-server.

The main methods that were impacted are invoked when the routers_updated message is sent from the quantum-server via the RPC. This invoked the routers_updated method, which in turn invokes self._process_routers, which then invokes self.process_routers. It is this final method with which we are concerned since all of the methods requiring changes are invoked from here. The process_routers method is also invoked when routers are deleted/synch, so making these changes where we did is an appropriate place.

The key methods changed in `quantum.agent.l3_agent.py` are as follows. We will discuss each in subsequent sections.

- `external_gateway_added`
external_gateway_removed
internal_network_added
internal_network_removed

8.2.1 Allowing multiple subnets

The first change we made was to accommodate the multiple subnets appearing in the "fixed_ips" section of the payload for external gateways that we are now sending from the quantum-server. This was in method external_gateway_added in class L3NATAgent in quantum.agent.l3_agent.py.

We build up all IP addresses, both IPv4 and IPv6, which will be added to the interface when layer 3 is initialized for the external gateway interface. The L3 driver (OVSInterfaceDriver) is already equipped to support multiple IP addresses on the interface.

Our revisions continue to add the single external network interface when the gateway is set for the tenant’s router as was implemented previously. We diverge by building a list of all possible IP addresses that we want to add to the interfaces. These, of course, include the IPv6 subnet and the IPv4 subnet. The following code segment from this method illustrates constructing the provided cidr addresses and pass them into the driver that will initialize the interface, and more importantly, add our IP addresses to the interface.

Figure 30 illustrates these changes need to accommodate multiple IP addresses on the same router gateway interface.

Figure 30: Modified “external_gateway_added” method to allow multiple subnets

```
# Build up the IP addresses that will be added to the interface.
ip_cidsrs = []
ips = ex_gw_port['fixed_ips']
for ip in ips:
    prefixlen = netaddr.IPNetwork(ip['cidr']).prefixlen
    ip_cidr = '%s/%s' % (ip['ip_address'], prefixlen)
    ip_cidsrs.append(ip_cidr)

# Initialize the device driver with IP addresses
self.driver.init_l3(interface_name, ip_cidsrs,
                     namespace=ri.ns_name())
```

The resultant interfaces appear as follows in the tenant router’s namespace as shown in Figure 31:
For our qg- interface, you can see that we have our IPv4 address (172.26.185.70) and our IPv6 address (2001:172:26:185::f816:3eff:feaf:768b). The IPv6 address is automatically calculated based on interface MAC address as the result of _generate_ipv4v6 method in quantum.db.db_base_plugin_v2.py.

### 8.2.2 Disallow NAT and GARP for IPv6

As we’ve mentioned before, we don’t think NAT is necessarily needed for IPv6, so we prevent destination NAT (DNAT) from being added to the NAT (ip6)tables when the gateway IP is IPv6-based for external network gateways. As you may have guessed, we simply checked the IP version sent to us from quantum-server on the subnet, thereby allowing only IPv4 external networks to perform GARP and DNAT configuration.

The following code fragment (Figure 32) from method `external_gateway_added` in class L3NATAgent in quantum.agent.l3_agent.py illustrates preventing implementation of NAT and GARP for anything but IPv4-based subnets.
Figure 32: Modified “external_gateway_added” method to avoid NAT and GARP

```python
for ip in ips:
    if (ip['ip_version'] == 4):
        ip_addr = ip['ip_address']
        self._send_gratuitous_arp_packet(ri, interface_name, ip_addr)
    gw_ip = ip['gateway_ip']
    if gw_ip:
        cmd = ['route', 'add', 'default', 'gw', gw_ip]
        if self.conf.use_namespaces:
            ip_wrapper = ip_lib.IPWrapper(self.root_helper,
                namespace=ri/ns_name())
            ip_wrapper.netns.execute(cmd, check_exit_code=False)
        else:
            utils.execute(cmd, check_exit_code=False,
                root_helper=self.root_helper)
        for (c, r) in self.external_gateway_nat_rules(ip_addr,
            internal_cidrs,
            interface_name):
            ri.iptables_manager.ipv4['nat'].add_rule(c, r)
        ri.iptables_manager.apply()
```

We also need to disallow source NAT (SNAT) from being added to the NAT (ip6)tables when the gateway IP is IPv6-based for internal networks. We see the iptables_manager of the RouterInfo class is equipped to support this, but we don’t find it necessary. Again, as you might imagine, it’s simply a check of the IP version in method `internal_network_added` in class `L3NATAgent` in `quantum.agent.l3_agent.py` to prevent actions based on IPv6 networks.

Figure 33 shows the changes needed to check IP version to prevent NAT rules from being applied.

Figure 33: Modified “internal_network_added” method to avoid SNAT

```python
net = netaddr.IPNetwork(internal_cidr)
if (net.version == 4):
    ip_address = internal_cidr.split('/')[0]
    self._send_gratuitous_arp_packet(ri, interface_name, ip_address)
    if ex_gw_port:
        fixed_ips = ex_gw_port['fixed_ips']
        for fixed_ip in fixed_ips:
            # Assume that if it's IPv4, then it's the gateway we want
            # from this port/interface.
            if (fixed_ip['ip_version'] == 4):
                for c, r in self.internal_network_nat_rules(fixed_ip['ip_address'],
                    net, internal_cidr):
                    ri.iptables_manager.ipv4['nat'].add_rule(c, r)
                ri.iptables_manager.apply()
```
8.2.3 Establishing Default Routes

Lastly, we do not set the default route for IPv6; rather we allow it to be discovered through router advertisement from the upstream router. For our configuration, our next-hop device generated a router advertisement (RA), which allowed a default route to be installed for our tenant (namespace) without having to explicitly code the route installation when the external gateway was set. As noted later, we do not wish to accept learning of the default route from internal network router advertisements. Of course, we continue to set the default route for IPv4 external networks.

The following code segment (Figure 34) from `external_gateway_added` from class `L3NATAgent` in `quantum.agent.l3_agent.py` illustrates setting up IPv6 to accept the default router learned from router advertisements on our specific interface. It is worth mentioning that “accept_ra” value must be set to ‘2’ in order to accept Router Advertisement and enable IPv6 routing at the same time.

**Figure 34: Modified “external_gateway_added” method to accept default routers**

```python
parm = "net.ipv6.conf.%s.accept_ra=2" % str(interface_name)
cmd = ["sysctl", parm]
if self.conf.use_namespaces:
ip_wrapper = ip_lib.IPWrapper(self.conf.root_helper,
namespace=ri.ns_name())
ip_wrapper.netns.execute(cmd, check_exit_code=False)
else:
    utils.execute(cmd, check_exit_code=False,
        root_helper=self.conf.root_helper)

parm = "net.ipv6.conf.%s.forwarding=1" % str(interface_name)
cmd = ["sysctl", parm]
if self.conf.use_namespaces:
ip_wrapper = ip_lib.IPWrapper(self.conf.root_helper,
namespace=ri.ns_name())
ip_wrapper.netns.execute(cmd, check_exit_code=False)
else:
    utils.execute(cmd, check_exit_code=False,
        root_helper=self.conf.root_helper)

parm = "net.ipv6.conf.%s.accept_ra_defrtr=1" % str(interface_name)
cmd = ["sysctl", parm]
if self.conf.use_namespaces:
ip_wrapper = ip_lib.IPWrapper(self.conf.root_helper,
namespace=ri.ns_name())
ip_wrapper.netns.execute(cmd, check_exit_code=False)
else:
    utils.execute(cmd, check_exit_code=False,
        root_helper=self.conf.root_helper)
```

As can be seen in Figure 35, our external gateway interface has learned default route, which now points to the link-local address of our next-hop device (Figure 36).
One drawback with router advertisement (RA) has to do with dnsmasq setup described previously, specifically for internal networks. This configuration can also generate a RA that is intended for the tenant VMs for purposes of establishing their default routes, but has the unfortunate side effect of being learned by the tenant router, i.e., the same router namespace. As a result, we have multiple (in our case, two) default routes to different networks. This would most certainly create routing issues.
To remedy the situation where internal networks are attached to the router with IPv6 subnets, we disable the ability to accept the default route with router advertisements. We only want a default route that is advertised from our external network gateway. We feel the L3 Agent is the most appropriate place for this since the driver (OVSInterfaceDriver) is also used on Compute nodes, which in that case, we may want the default route learned through router advertisement.

We modified method internal_network_added (Figure 37) in class L3NATAgent to prevent learning a default route on internal-network router interfaces, but to allow it on external network router gateways.

Figure 37: Disable learning of default routers on internal-network port

```python
# Disallow any router advertisement of gateway (default route)
# for IPv6 on this interface for all internal networks.
parm = "net.ipv6.conf.%s.accept_ra_defrtr=0" % str(interface_name)
cmd = ['sysctl', parm]
if self.conf.use_namespaces:
    ip_wrapper = ip_lib.IPWrapper(self.conf.root_helper,
                               namespace=ri.ns_name())
    ip_wrapper.netns.execute(cmd, check_exit_code=False)
else:
    utils.execute(cmd, check_exit_code=False,
                   root_helper=self.conf.root_helper)
```

8.3  Hello, World!

As the final test, we tried to verify the IPv6 reachability from virtual machine to our next-hop device’s port, and to a subnet beyond that point. The ping6 commands on virtual machine showed that the virtual machine can now go off the IPv6 Island (Figure 39) and say “Hello, World!” (Figure 40) by pinging a network beyond our attached next-hop device.

Figure 38 illustrates our final network view that we were attempting to achieve.
Figure 38: Network View

Figure 39: Virtual Machine (Centos1) IPv6 reachability to tenant external network

```
[root@centos1 ~]# ping6 2001:172:26:185::1
64 bytes from 2001:172:26:185::1: icmp_seq=1 ttl=63 time=3.50 ms
64 bytes from 2001:172:26:185::1: icmp_seq=2 ttl=63 time=1.63 ms
64 bytes from 2001:172:26:185::1: icmp_seq=3 ttl=63 time=2.39 ms
64 bytes from 2001:172:26:185::1: icmp_seq=4 ttl=63 time=1.58 ms
64 bytes from 2001:172:26:185::1: icmp_seq=5 ttl=63 time=1.38 ms
64 bytes from 2001:172:26:185::1: icmp_seq=6 ttl=63 time=4.27 ms
64 bytes from 2001:172:26:185::1: icmp_seq=7 ttl=63 time=2.46 ms
64 bytes from 2001:172:26:185::1: icmp_seq=8 ttl=63 time=1.38 ms
```
Figure 40: Virtual Machine (Centos1) IPv6 reachability to external network

```
[root@centos1 ~]# ping6 2001:7:10:177::254
64 bytes from 2001:7:10:177::254: icmp_seq=1 ttl=63 time=3.20 ms
64 bytes from 2001:7:10:177::254: icmp_seq=2 ttl=63 time=1.30 ms
64 bytes from 2001:7:10:177::254: icmp_seq=3 ttl=63 time=1.43 ms
64 bytes from 2001:7:10:177::254: icmp_seq=4 ttl=63 time=1.08 ms
64 bytes from 2001:7:10:177::254: icmp_seq=5 ttl=63 time=1.15 ms
64 bytes from 2001:7:10:177::254: icmp_seq=6 ttl=63 time=0.924 ms
64 bytes from 2001:7:10:177::254: icmp_seq=7 ttl=63 time=1.65 ms
64 bytes from 2001:7:10:177::254: icmp_seq=8 ttl=63 time=1.25 ms
```
9. Conclusion
IPv6 and Cloud are two of several major inflection points in the ongoing evolution of IT industry. With the firm belief that Cloud will not be able to unleash its potential without IPv6, we invested significant time and effort to evaluate the IPv6 readiness of OpenStack.

This paper provides you with a glimpse of the gaps in latest Grizzly release and potential fixes including both Python code changes and workarounds. Despite the difficulties encountered on our journey, we have a working system at this moment to demonstrate we reached our original objectives to enable OpenStack with IPv6 in the terms of:

- All OpenStack infrastructure nodes can now communicate with each other by IPv6
- OpenStack can now spin up IPv6-enabled virtual machines in multi-tendency environment
- Virtual machine can now reach existing IPv6 network beyond the control of OpenStack system
- And finally, the latter two also coexist with IPv4 (dual-stack).

10. Next Steps
We certainly recognize that there is still a long way to go before we can claim “mission complete”. To fully understand the advantage of IPv6 + OpenStack, we are now further motivated to take one step forward and evaluate the performance improvement, if any, brought by IPv6 on OpenStack. More importantly, discover what are the impacts of IPv6 on various services hosted in cloud ecosystems running OpenStack.

With all of these pending questions in our minds, we would like to invite you to join us to continue enhancing OpenStack to be production-ready for real IPv6 deployment!
11. Appendix

11.1 Appendix A: Disable IPv6 Duplicate Address Detection on Centos

[root@centos1 ~] sysctl net.ipv6.conf.eth0.accept_dad=0

net.ipv6.conf.eth0.accept_dad = 0