NetReady: Network Readiness

Release draft (90f7b0b)

OPNFV

August 11, 2016

Project NetReady, https://wiki.opnfv.org/display/netready/NetReady

Editors TBD

Authors Bin Hu (AT&T), Gergely Csatari (Nokia), Georg Kunz (Ericsson) and others

Abstract OPNFV provides an infrastructure with different SDN controller options to realize NFV functionality on the platform it builds. As OPNFV uses OpenStack as a VIM, we need to analyze the capabilities this component offers us. The networking functionality is provided by a single component called Neutron, which hides the controller under it, let it be Neutron itself or any supported SDN controller. As NFV wasn't taken into consideration at the time when Neutron was designed we are already facing several bottlenecks and architectural shortcomings while implementing our use cases.

The NetReady project aims at evolving OpenStack networking step-by-step to find the most efficient way to fulfill the requirements of the identified NFV use cases, taking into account the NFV mindset and the capabilities of SDN controllers.

History

Date	Description
22.03.2016	Project creation
19.04.2016	Initial version of the deliverable uploaded to Gerrit
22.07.2016	First version ready for sharing with the community

Definition of terms

Different standards developing organizations and communities use different terminology related to Network Function Virtualization, Cloud Computing, and Software Defined Networking. This list defines the terminology in the contexts of this document.

API Application Programming Interface.

Cloud Computing A model that enables access to a shared pool of configurable computing resources, such as networks, servers, storage, applications, and services, that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Edge Computing Edge computing pushes applications, data and computing power (services) away from centralized points to the logical extremes of a network.

Instance Refers in OpenStack terminology to a running VM, or a VM in a known state such as suspended, that can be used like a hardware server.

NFV Network Function Virtualization.

NFVI Network Function Virtualization Infrastructure. Totality of all hardware and software components which build up the environment in which VNFs are deployed.

SDN Software-Defined Networking. Emerging architecture that decouples the network control and forwarding functions, enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services.

Server Computer that provides explicit services to the client software running on that system, often managing a variety of computer operations. In OpenStack terminology, a server is a VM instance.

vForwarder vForwarder is used as generic and vendor neutral term for a software packet forwarder. Concrete examples includes OpenContrail vRouter, OpenvSwitch, Cisco VTF.

VIM Virtualized Infrastructure Manager. Functional block that is responsible for controlling and managing the NFVI compute, storage and network resources, usually within one operator's Infrastructure Domain, e.g. NFVI Point of Presence (NFVI-PoP).

Virtual network Virtual network routes information among the network interfaces of VM instances and physical network interfaces, providing the necessary connectivity.

VM Virtual Machine. Virtualized computation environment that behaves like a physical computer/server by modeling the computing architecture of a real or hypothetical computer.

VNF Virtualized Network Function. Implementation of an Network Function that can be deployed on a Network Function Virtualization Infrastructure (NFVI).

WAN Wide Area Network.

CHAPTER

ONE

INTRODUCTION

This document represents and describes the results of the OPNFV NetReady (Network Readiness) project. Specifically, the document comprises a selection of NFV-related networking use cases and their networking requirements. For every use case, it furthermore presents a gap analysis of the aforementioned requirements with respect to the current OpenStack networking architecture. Finally it provides a description of potential solutions and improvements.

1.1 Scope

NetReady is a project within the OPNFV initiative. Its focus is on NFV (Network Function Virtualization) related networking use cases and their requirements on the underlying NFVI (Network Function Virtualization Infrastructure).

The NetReady project addresses the OpenStack networking architecture, specifically OpenStack Neutron, from a NFV perspective. Its goal is to identify gaps in the current OpenStack networking architecture with respect to NFV requirements and to propose and evaluate improvements and potential complementary solutions.

1.2 Problem Description

Telco ecosystem's movement towards the cloud domain results in Network Function Virtualization that is discussed and specified in ETSI NFV. This movement opens up many green field areas which are full of potential growth in both business and technology. This new NFV domain brings new business opportunities and new market segments as well as emerging technologies that are exploratory and experimental in nature, especially in NFV networking.

It is often stated that NFV imposes additional requirements on the networking architecture and feature set of the underlying NFVI beyond those of data center networking. For instance, the NFVI needs to establish and manage connectivity beyond the data center to the WAN (Wide Area Network). Moreover, NFV networking use cases often abstract from L2 connectivity and instead focus on L3-only connectivity. Hence, the NFVI networking architecture needs to be flexible enough to be able to meet the requirements of NFV-related use cases in addition to traditional data center networking.

Traditionally, OpenStack networking, represented typically by the OpenStack Neutron project, targets virtualized data center networking. This comprises originally establishing and managing layer 2 network connectivity among VMs (Virtual Machines). Over the past releases of OpenStack, Neutron has grown to provide an extensive feature set, covering both L2 as well as L3 networking services such as virtual routers, NATing, VPNaaS and BGP VPNs.

It is an ongoing debate how well the current OpenStack networking architecture can meet the additional requirements of NFV networking. Hence, a thorough analysis of NFV networking requirements and their relation to the OpenStack networking architecture is needed.

Besides current additional use cases and requirements of NFV networking, more importantly, because of the **green field** nature of NFV, it is foreseen that there will be more and more new NFV networking use cases and services, which will bring new business, in near future. The challenges for telco ecosystem are to:

- Quickly catch the new business opportunity;
- Execute it in agile way so that we can accelerate the time-to-market and improve the business agility in offering our customers with innovative NFV services.

Therefore, it is critically important for telco ecosystem to quickly develop and deploy new NFV networking APIs on-demand based on market need.

1.3 Goals

The goals of the NetReady project and correspondingly this document are the following:

- This document comprises a collection of relevant NFV networking use cases and clearly describes their requirements on the NFVI. These requirements are stated independently of a particular implementation, for instance OpenStack Neutron. Instead, requirements are formulated in terms of APIs (Application Programming Interfaces) and data models needed to realize a given NFV use case.
- The list of use cases is not considered to be all-encompassing but it represents a carefully selected set of use cases that are considered to be relevant at the time of writing. More use cases may be added over time. The authors are very open to suggestions, reviews, clarifications, corrections and feedback in general.
- This document contains a thorough analysis of the gaps in the current OpenStack networking architecture with respect to the requirements imposed by the selected NFV use cases. To this end, we analyze existing functionality in OpenStack networking.
- Beyond current list of use cases and gap analysis in the document, more importantly, it is the future of NFV networking that needs to be made easy to innovate, quick to develop, and agile to deploy and operate. A model-driven, extensible framework is expected to achieve agility for innovations in NFV networking.
- This document will in future revisions describe the proposed improvements and complementary solutions needed to enable OpenStack to fulfill the identified NFV requirements.

CHAPTER

TWO

USE CASES

The following sections address networking use cases that have been identified to be relevant in the scope of NFV and NetReady.

2.1 Multiple Networking Backends

2.1.1 Description

Network Function Virtualization (NFV) brings the need of supporting multiple networking back-ends in virtualized infrastructure environments.

First of all, a Service Providers' virtualized network infrastructure will consist of multiple SDN Controllers from different vendors for obvious business reasons. Those SDN Controllers may be managed within one cloud or multiple clouds. Jointly, those VIMs (e.g. OpenStack instances) and SDN Controllers need to work together in an interoperable framework to create NFV services in the Service Providers' virtualized network infrastructure. It is needed that one VIM (e.g. OpenStack instance) shall be able to support multiple SDN Controllers as back-end.

Secondly, a Service Providers' virtualized network infrastructure will serve multiple, heterogeneous administrative domains, such as mobility domain, access networks, edge domain, core networks, WAN, enterprise domain, etc. The architecture of virtualized network infrastructure needs different types of SDN Controllers that are specialized and targeted for specific features and requirements of those different domains. The architectural design may also include global and local SDN Controllers. Importantly, multiple local SDN Controllers may be managed by one VIM (e.g. OpenStack instance).

Furthermore, even within one administrative domain, NFV services could also be quite diversified. Specialized NFV services require specialized and dedicated SDN Controllers. Thus a Service Provider needs to use multiple APIs and back-ends simultaneously in order to provide users with diversified services at the same time. At the same time, for a particular NFV service, the new networking APIs need to be agnostic of the back-ends.

2.1.2 Requirements

Based on the use cases described above, we derive the following requirements.

It is expected that in NFV networking service domain:

- One OpenStack instance shall support multiple APIs and SDN Controllers simultaneously
- New NFV Networking APIs shall be agnostic of back-ends
- Interoperability is needed among multi-vendor SDN Controllers at back-end

2.1.3 Current Implementation

In the current implementation of OpenStack networking, SDN controllers are hooked up to Neutron by means of dedicated plugins. A plugin translates requests coming in through the Neutron northbound API, e.g. the creation of a new network, into the appropriate northbound API calls of the corresponding SDN controller.

There are multiple different plugin mechanisms currently available in Neutron, each targeting a different purpose. In general, there are *core plugins*, covering basic networking functionality and *service plugins*, providing layer 3 connectivity and advanced networking services such as FWaaS or LBaaS.

Core and ML2 Plugins

The Neutron core plugins cover basic Neutron functionality, such as creating networks and ports. Every core plugin implements the functionality needed to cover the full range of the Neutron core API. A special instance of a core plugin is the ML2 core plugin, which in turn allows for using sub-drivers - separated again into type drivers (VLAN, VxLAN, GRE) or mechanism drivers (OVS, OpenDaylight, etc.). This allows to using dedicated sub-drivers for dedicated functionality.

In practice, different SDN controllers use both plugin mechanisms to integrate with Neutron. For instance OpenDaylight uses a ML2 mechanism plugin driver whereas OpenContrail integrated by means of a full core plugin.

In its current implementation, only one Neutron core plugin can be active at any given time. This means that if a SDN controller utilizes a dedicated core plugin, no other SDN controller can be used at the same time for the same type of service.

In contrast, the ML2 plugin allows for using multiple mechanism drivers simultaneously. In principle, this enables a parallel deployment of multiple SDN controllers if and only if all SDN controllers integrate through a ML2 mechanism driver.

Neutron Service Plugins

Neutron service plugins target L3 services and advanced networking services, such as BGPVPN or LBaaS. Typically, a service itself provides a driver plugin mechanism which needs to be implemented for every SDN controller. As the architecture of the driver mechanism is up to the community developing the service plugin, it needs to be analyzed for every driver plugin mechanism individually if and how multiple back-ends are supported.

2.1.4 Gaps in the current solution

Given the use case description and the current implementation of OpenStack Neutron, we identify the following gaps:

[MB-GAP1] Limited support for multiple back-ends

As pointed out above, the Neutron core plugin mechanism only allows for one active plugin at a time. The ML2 plugin allows for running multiple mechanism drivers in parallel, however, successful inter-working strongly depends on the individual driver.

2.1.5 Conclusion

We conclude that a clean method of integrating multiple SDN controllers into a single OpenStack deployment is needed to fulfill the needs of operators.

2.2 L3VPN Use Cases

L3VPNs are virtual layer 3 networks described in multiple standards and RFCs, such as *[RFC4364]* and *[RFC7432]*. Connectivity as well as traffic separation is achieved by exchanging routes between VRFs (Virtual Routing and Forwarding).

Moreover, a Service Providers' virtualized network infrastructure may consist of one or more SDN Controllers from different vendors. Those SDN Controllers may be managed within one cloud or multiple clouds. Jointly, those VIMs (e.g. OpenStack instances) and SDN Controllers work together in an interoperable framework to create L3 services in the Service Providers' virtualized network infrastructure.

While interoperability between SDN controllers and the corresponding data planes is ensured based on standardized protocols (e.g., *[RFC4364]* and *[RFC7432]*), the integration and management of different SDN domains from the VIM is not clearly defined. Hence, this section analyses three L3VPN use cases involving multiple SDN Controllers.

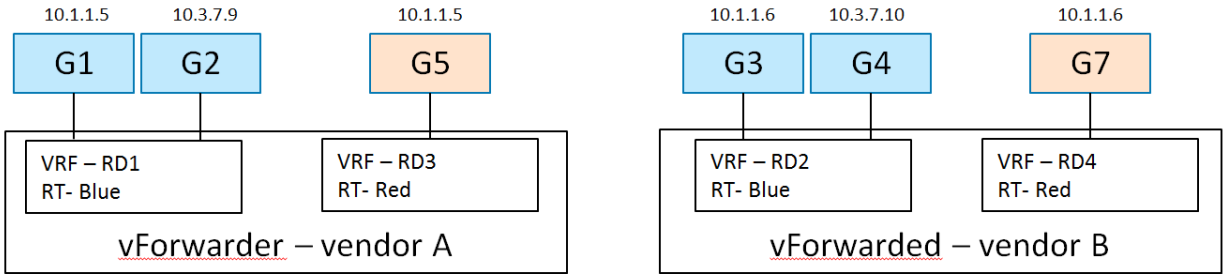
2.2.1 Any-to-Any Base Case

Description

This any-to-any use case is the base scenario, providing layer 3 connectivity between VNFs in the same L3VPN while separating the traffic and IP address spaces of different L3VPNs belonging to different tenants.

There are 2 hosts (compute nodes). SDN Controller A and vForwarder A are provided by Vendor A and run on host A. SDN Controller B and vForwarder B are provided by Vendor B, and run on host B.

There are 2 tenants. Tenant 1 creates L3VPN Blue with 2 subnets: 10.1.1.0/24 and 10.3.7.0/24. Tenant 2 creates L3VPN Red with 1 subnet and an overlapping address space: 10.1.1.0/24. The network topology is shown in Fig. 2.2.1.



In L3VPN Blue, VMs G1 (10.1.1.5) and G2 (10.3.7.9) are spawned on host A, and attached to 2 subnets (10.1.1.0/24 and 10.3.7.0/24) and assigned IP addresses respectively. VMs G3 (10.1.1.6) and G4 (10.3.7.10) are spawned on host B, and attached to 2 subnets (10.1.1.0/24 and 10.3.7.0/24) and assigned IP addresses respectively.

In L3VPN Red, VM G5 (10.1.1.5) is spawned on host A, and attached to subnet 10.1.1.0/24. VM G6 (10.1.1.6) is spawned on host B, and attached to the same subnet 10.1.1.0/24.

Derived Requirements

Northbound API / Workflow

An example of the desired workflow is as follows:

1. Create Network
2. Create Network VRF Policy Resource `Any-to-Any`

   2.1. This policy causes the following configuration when a VM of this tenant is spawned on a host:

      2.1.1. There will be a RD assigned per VRF

      2.1.2. There will be a RT used for the common any-to-any communication
3. Create Subnet
4. Create Port (subnet, network VRF policy resource). This causes the controller to:

   4.1. Create a VRF in vForwarder's FIB, or update VRF if it already exists

   4.2. Install an entry for the guest's host route in FIBs of the vForwarder serving this tenant's virtual network

   4.3. Announce guest host route to WAN-GW via MP-BGP

Current implementation

Support for creating and managing L3VPNs is available in OpenStack Neutron by means of the *[BGPVPN]* project. In order to create the L3VPN network configuration described above using the API *[BGPVPN]* API, the following workflow is needed:

1. Create Neutron networks for tenant "Blue"

   ```
   neutron net-create --tenant-id Blue net1
   ```

   ```
   neutron net-create --tenant-id Blue net2
   ```
2. Create subnets for the Neutron networks for tenant "Blue"

   ```
   neutron subnet-create --tenant-id Blue --name subnet1 net1
   10.1.1.0/24
   ```

   ```
   neutron subnet-create --tenant-id Blue --name subnet2 net2
   10.3.7.0/24
   ```
3. Create Neutron ports in the corresponding networks for tenant "Blue"

   ```
   neutron port-create --tenant-id Blue --name G1 --fixed-ip
   subnet_id=subnet1,ip_address=10.1.1.5 net1
   ```

   ```
   neutron port-create --tenant-id Blue --name G2 --fixed-ip
   subnet_id=subnet1,ip_address=10.1.1.6 net1
   ```

   ```
   neutron port-create --tenant-id Blue --name G3 --fixed-ip
   subnet_id=subnet2,ip_address=10.3.7.9 net2
   ```

   ```
   neutron port-create --tenant-id Blue --name G4 --fixed-ip
   subnet_id=subnet2,ip_address=10.3.7.10 net2
   ```
4. Create Neutron network for tenant "Red"

   ```
   neutron net-create --tenant-id Red net3
   ```
5. Create subnet for the Neutron network of tenant "Red"

   ```
   neutron subnet-create --tenant-id Red --name subnet3 net3 10.1.1.0/24
   ```
6. Create Neutron ports in the networks of tenant "Red"

```
neutron port-create --tenant-id Red --name G5 --fixed-ip
subnet_id=subnet3,ip_address=10.1.1.5 net3

neutron port-create --tenant-id Red --name G7 --fixed-ip
subnet_id=subnet3,ip_address=10.1.1.6 net3
```

7. Create a L3VPN by means of the BGPVPN API for tenant "Blue"

```
neutron bgpvpn-create --tenant-id Blue --route-targets AS:100 --name
vpn1
```

8. Associate the L3VPN of tenant "Blue" with the previously created networks

```
neutron bgpvpn-net-assoc-create --tenant-id Blue --network net1
--name vpn1

neutron bgpvpn-net-assoc-create --tenant-id Blue --network net2
--name vpn1
```

9. Create a L3VPN by means of the BGPVPN API for tenant "Red"

```
neutron bgpvpn-create --tenant-id Red --route-targets AS:200 --name
vpn2
```

10. Associate the L3VPN of tenant "Red" with the previously created networks

```
neutron bgpvpn-net-assoc-create --tenant-id Red --network net3 --name
vpn2
```

Comments:

- In this configuration only one BGPVPN for each tenant is created.
- The ports are associated indirectly to the VPN through their networks.
- The BGPVPN backend takes care of distributing the /32 routes to the vForwarder instances and assigning appropriate RD values.

Gaps in the current solution

In terms of the functionality provided by the BGPVPN project, there are no gaps preventing this particular use case from a L3VPN perspective.

However, in order to support the multi-vendor aspects of this use case, a better support for integrating multiple backends is needed (see previous use case).

2.2.2 L3VPN: ECMP Load Splitting Case (Anycast)

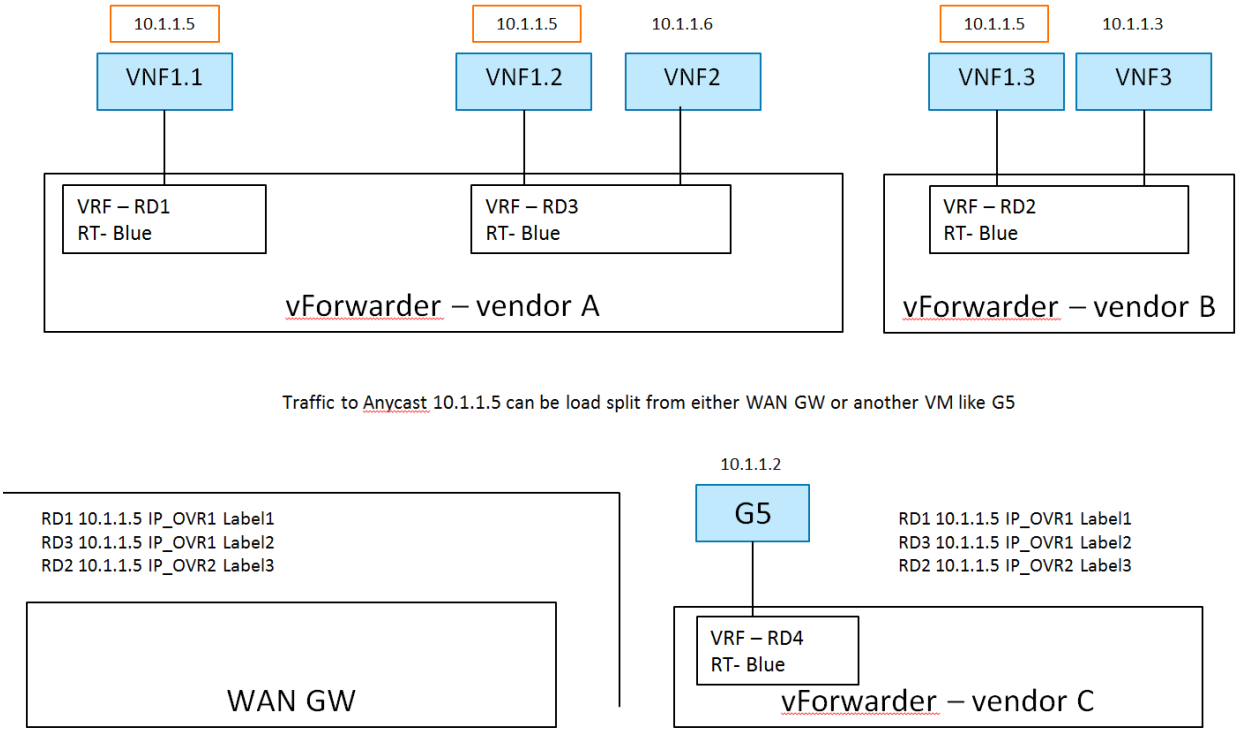
Description

In this use case, multiple instances of a VNF are reachable through the same IP. The networking infrastructure is then responsible for spreading the network load across the VNF instances using Equal-Cost Multi-Path (ECMP) or perform a fail-over in case of a VNF failure.

There are 2 hosts (compute nodes). SDN Controller A and vForwarder A are provided by Vendor A, and run on host A. SDN Controller B and vForwarder B are provided by Vendor B, and run on host B.

There is one tenant. Tenant 1 creates L3VPN Blue with subnet 10.1.1.0/24.

The network topology is shown in Fig. 2.2.2:



In L3VPN Blue, VNF1.1 and VNF1.2 are spawned on host A, attached to subnet 10.1.1.0/24 and assigned the same IP address 10.1.1.5. VNF1.3 is spawned on host B, attached to subnet 10.1.1.0/24 and assigned the same IP addresses 10.1.1.5. VNF 2 and VNF 3 are spawned on host A and B respectively, attached to subnet 10.1.1.0/24, and assigned different IP addresses 10.1.1.6 and 10.1.1.3 respectively.

Here, the Network VRF Policy Resource is `ECMP/AnyCast`. Traffic to the anycast IP **10.1.1.5** can be load split from either WAN GW or another VM like G5.

Current implementation

Support for creating and managing L3VPNs is, in general, available in OpenStack Neutron by means of the BGPVPN project *[BGPVPN]*. However, the BGPVPN project does not yet fully support ECMP as described in the following.

There are (at least) two different approached to configuring ECMP:

1. Using Neutron ports with identical IP addresses, or
2. Using Neutron ports with unique IPs addresses and creating static routes to a common IP prefix with next hops pointing to the unique IP addresses.

Ports with identical IP addresses

In this approach, multiple Neutron ports using the same IP address are created. In the current Neutron architecture, a port has to reside in a specific Neutron network. However, re-using the same IP address multiple times in a given Neutron network is not possible as this would create an IP collision. As a consequence, creating one Neutron network for each port is required.

Given multiple Neutron networks, the BGPVPN API allows for associating those networks with the same VPN. It is then up to the networking backend to implement ECMP load balancing. This behavior and the corresponding API for configuring the behavior is currently not available. It is nevertheless on the road map of the BGPVPN project.

Static Routes to ports with unique IP addresses

In this approach, Neutron ports are assigned unique IPs and static routes pointing to the same ECMP load-balanced prefix are created. The static routes define the unique Neutron port IPs as next-hop addresses.

Currently, the API for static routes is not yet available in the BGPVPN project, but it is on the road map. The following work flow shows how to realize this particular use case under the assumption that support for static routes is available in the BGPVPN API.

1. Create Neutron network for tenant "Blue"

   ```
   neutron net-create --tenant-id Blue net1
   ```

2. Create subnet for the network of tenant "Blue"

   ```
   neutron subnet-create --tenant-id Blue --name subnet1 net1 5.1.1.0/24
   ```

3. Create Neutron ports in the network of tenant "Blue"

   ```
   neutron port-create --tenant-id Blue --name G1 --fixed-ip
   subnet_id=subnet1,ip_address=5.1.1.1 net1

   neutron port-create --tenant-id Blue --name G2 --fixed-ip
   subnet_id=subnet1,ip_address=5.1.1.2 net1

   neutron port-create --tenant-id Blue --name G3 --fixed-ip
   subnet_id=subnet1,ip_address=5.1.1.3 net1

   neutron port-create --tenant-id Blue --name G4 --fixed-ip
   subnet_id=subnet1,ip_address=5.1.1.4 net1

   neutron port-create --tenant-id Blue --name G5 --fixed-ip
   subnet_id=subnet1,ip_address=5.1.1.5 net1

   neutron port-create --tenant-id Blue --name G6 --fixed-ip
   subnet_id=subnet1,ip_address=5.1.1.6 net1
   ```

4. Create a L3VPN for tenant "Blue"

   ```
   neutron bgpvpn-create --tenant-id Blue --route-target AS:100 vpn1
   ```

5. Associate the BGPVPN with the network of tenant "Blue"

   ```
   neutron bgpvpn-network-associate --tenant-id Blue --network-id net1
   vpn1
   ```

6. Create static routes which point to the same target

   ```
   neutron bgpvpn-static-route-add --tenant-id Blue --cidr 10.1.1.5/32
   --nexthop-ip 5.1.1.1 vpn1

   neutron bgpvpn-static-route-add --tenant-id Blue --cidr 10.1.1.5/32
   --nexthop-ip 5.1.1.2 vpn1

   neutron bgpvpn-static-route-add --tenant-id Blue --cidr 10.1.1.5/32
   --nexthop-ip 5.1.1.3 vpn1
   ```

Gaps in the current solution

Given the use case description and the currently available implementation in OpenStack provided by BGPVPN project, we identify the following gaps:

- [L3VPN-ECMP-GAP1] Static routes are not yet supported by the BGPVPN project.

  Currently, no API for configuring static routes is available in the BGPVPN project. This feature is on the road map, however.

- [L3VPN-ECMP-GAP2] Behavior not defined for multiple Neutron ports of the same IP

  The Neutron and BGPVPN API allow for creating multiple ports with the same IP in different networks and associating the networks with the same VPN. The exact behavior of this configuration is however not defined and an API for configuring the behavior (load-balancing or fail-over) is missing. Development of this feature is on the road map of the project, however.

- [L3VPN-ECMP-GAP3] It is not possible to assign the same IP to multiple Neutron ports within the same Neutron subnet.

  This is due to the fundamental requirement of avoiding IP collisions within the L2 domain which is a Neutron network.

Conclusions

In the context of the ECMP use case, three gaps have been identified. Gap [L3VPN-ECMP-GAP1] and [L3VPN-ECMP-GAP2] are missing or undefined functionality in the BGPVPN project. There is no architectural hindrance preventing the implementation of the missing features in the BGPVPN project as well as in Neutron.

The third gap [L3VPN-ECMP-GAP3] is based on the fact that Neutron ports always have to exist in a Neutron network. As a consequence, in order to create ports with the same IP, multiple networks must be used. This port-network binding will most likely not be relaxed in future releases of Neutron to retain backwards compatibility. A clean alternative to Neutron can instead provide more modeling flexibility.

2.2.3 Hub and Spoke Case

Description

In a traditional Hub-and-spoke topology there are two types of network entities: a central hub and multiple spokes. The corresponding VRFs of the hub and the spokes are configured to import and export routes such that all traffic is directed through the hub. As a result, spokes cannot communicate with each other directly, but only indirectly via the central hub. Hence, the hub typically hosts central network functions such firewalls.

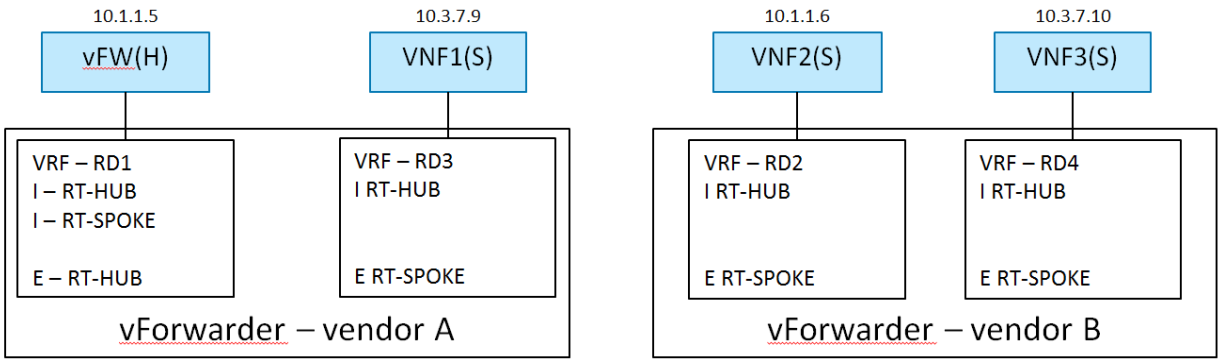
Furthermore, there is no layer 2 connectivity between the VNFs.

In addition, in this use case, the deployed network infrastructure comprises equipment from two different vendors, Vendor A and Vendor B. There are 2 hosts (compute nodes). SDN Controller A and vForwarder A are provided by Vendor A, and run on host A. SDN Controller B and vForwarder B are provided by Vendor B, and run on host B.

There is 1 tenant. Tenant 1 creates L3VPN Blue with 2 subnets: 10.1.1.0/24 and 10.3.7.0/24.

The network topology is shown in Fig. 2.2.3:

In L3VPN Blue, vFW(H) is acting the role of `hub` (a virtual firewall). The other 3 VNF VMs are `spoke`. vFW(H) and VNF1(S) are spawned on host A, and VNF2(S) and VNF3(S) are spawned on host B. vFW(H) (10.1.1.5) and VNF2(S) (10.1.1.6) are attached to subnet 10.1.1.0/24. VNF1(S) (10.3.7.9) and VNF3(S) (10.3.7.10) are attached to subnet 10.3.7.0/24.



Derived Requirements

Northbound API / Workflow

Exemplary vFW(H) Hub VRF is as follows:

- RD1 10.1.1.5 IP_vForwarder1 Label1
- RD1 0/0 IP_vForwarder1 Label1
- Label 1 Local IF (10.1.1.5)
- RD3 10.3.7.9 IP_vForwarder1 Label2
- RD2 10.1.1.6 IP_vForwarder2 Label3
- RD4 10.3.7.10 IP_vForwarder2 Label3

Exemplary VNF1(S) Spoke VRF is as follows:

- RD1 0/0 IP_vForwarder1 Label1
- RD3 10.3.7.9 IP_vForwarder1 Label2

Exemplary workflow is described as follows:

1. Create Network
2. Create VRF Policy Resource

   2.1. Hub and Spoke
3. Create Subnet
4. Create Port

   4.1. Subnet

   4.2. VRF Policy Resource, [H | S]

Current implementation

Different APIs have been developed to support creating a L3 network topology and directing network traffic through specific network elements in specific order, for example, *[BGPVPN]* and *[NETWORKING-SFC]*. We analyzed those APIs regarding the Hub-and-Spoke use case.

BGPVPN Support for creating and managing L3VPNs is in general available in OpenStack Neutron by means of the BGPVPN API *[BGPVPN]*. The *[BGPVPN]* API currently supports the concepts of network- and router-associations. An association maps Neutron network objects (networks and routers) to a VRF with the following semantics:

- A *network association* interconnects all subnets and ports of a Neutron network by binding them to a given VRF
- a *router association* interconnects all networks, and hence indirectly all ports, connected to a Neutron router by binding them to a given VRF

It is important to notice that these associations apply to entire Neutron networks including all ports connected to a network. This is due to the fact that in the Neutron, ports can only exist within a network but not individually. Furthermore, Neutron networks were originally designed to represent layer 2 domains. As a result, ports within the same Neutron network typically have layer connectivity among each other. There are efforts to relax this original design assumption, e.g. routed networks, which however do not solve the problem at hand here (see the gap analysis further down below).

In order to realize the hub-and-spoke topology outlined above, VRFs need to be created on a per port basis. Specifically, ports belonging to the same network should not be interconnected except through a corresponding configuration of a per-port-VRF. This configuration includes setting up next-hop routing table, labels, I-RT and E-RT etc. in order to enable traffic direction from hub to spokes.

It may be argued that given the current network- and router-association mechanisms, the following workflow establishes a network topology which aims to achieve the desired traffic flow from Hub to Spokes. The basic idea is to model separate VRFs per VM by creating a dedicated Neutron network with two subnets for each VRF in the Hub-and-Spoke topology.

1. Create Neutron network "hub"

   ```
   neutron net-create --tenant-id Blue hub
   ```

2. Create a separate Neutron network for every "spoke"

   ```
   neutron net-create --tenant-id Blue spoke-i
   ```

3. For every network (hub and spokes), create two subnets

   ```
   neutron subnet-create <hub/spoke-i UUID> --tenant-id Blue 10.1.1.0/24
   ```

   ```
   neutron subnet-create <hub/spoke-i UUID> --tenant-id Blue 10.3.7.0/24
   ```

4. Create the Neutron ports in the corresponding networks

   ```
   neutron port-create --tenant-id Blue --name vFW(H) --fixed-ip
   subnet_id=<hub UUID>,ip_address=10.1.1.5
   ```

   ```
   neutron port-create --tenant-id Blue --name VNF1(S) --fixed-ip
   subnet_id=<spoke-i UUID>,ip_address=10.3.7.9
   ```

   ```
   neutron port-create --tenant-id Blue --name VNF2(S) --fixed-ip
   subnet_id=<spoke-i UUID>,ip_address=10.1.1.6
   ```

   ```
   neutron port-create --tenant-id Blue --name VNF3(S) --fixed-ip
   subnet_id=<spoke-i UUID>,ip_address=10.3.7.10
   ```

5. Create a BGPVPN object (VRF) for the hub network with the corresponding import and export targets

   ```
   neutron bgpvpn-create --name hub-vrf --import-targets <RT-hub
   RT-spoke> --export-targets <RT-hub>
   ```

6. Create a BGPVPN object (VRF) for every spoke network with the corresponding import and export targets

   ```
   neutron bgpvpn-create --name spoke-i-vrf --import-targets <RT-hub>
   --export-targets <RT-spoke>
   ```

7. Associate the hub network with the hub VRF

```
bgpvpn-net-assoc-create hub --network <hub network-UUID>
```

8. Associate each spoke network with the corresponding spoke VRF

   ```
   bgpvpn-net-assoc-create spoke-i --network <spoke-i network-UUID>
   ```

9. Add static route to direct all traffic to vFW VNF running at the hub.

   Note: Support for static routes not yet available.

   ```
   neutron bgpvpn-static-route-add --tenant-id Blue --cidr 0/0
   --nexthop-ip 10.1.1.5 hub
   ```

After step 9, VMs can be booted with the corresponding ports.

The resulting network topology intents to resemble the target topology as shown in Fig. 2.2.3, and achieve the desired traffic direction from Hub to Spoke. However, it deviates significantly from the essence of the Hub-and-Spoke use case as described above in terms of desired network topology, i.e. one L3VPN with multiple VRFs associated with vFW(H) and other VNFs(S) separately. And this method of using the current network- and router-association mechanism is not scalable when there are large number of Spokes, and in case of scale-in and scale-out of Hub and Spokes.

The gap analysis in the next section describes the technical reasons for this.

Network SFC Support of Service Function Chaining is in general available in OpenStack Neutron through the Neutron API for Service Insertion and Chaining project *[NETWORKING-SFC]*. However, the *[NETWORKING-SFC]* API is focused on creating service chaining through NSH at L2, although it intends to be agnostic of backend implementation. It is unclear whether or not the service chain from vFW(H) to VNFs(S) can be created in the way of L3VPN-based VRF policy approach using *[NETWORKING-SFC]* API.

Hence, it is currently not possible to configure the networking use case as described above.

Gaps in the Current Solution

Given the use case description and the currently available implementation in OpenStack provided by *[BGPVPN]* project and *[NETWORKING-SFC]* project, we identify the following gaps:

[L3VPN-HS-GAP1] No means to disable layer 2 semantic of Neutron networks Neutron networks were originally designed to represent layer 2 broadcast domains. As such, all ports connected to a network are in principle inter-connected on layer 2 (not considering security rules here). In contrast, in order to realize L3VPN use cases such as the hub-and-spoke topology, connectivity among ports must be controllable on a per port basis on layer 3.

There are ongoing efforts to relax this design assumption, for instance by means of routed networks (*[NEUTRON-ROUTED-NETWORKS]*). In a routed network, a Neutron network is a layer 3 domain which is composed of multiple layer 2 segments. A routed network only provides layer 3 connectivity across segments, but layer 2 connectivity across segments is **optional**. This means, depending on the particular networking backend and segmentation technique used, there might be layer 2 connectivity across segments or not. A new flag `l2_adjacency` indicates whether or not a user can expect layer 2 connectivity or not across segments.

This flag, however, is ready-only and cannot be used to overwrite or disable the layer 2 semantics of a Neutron network.

[L3VPN-HS-GAP2] No port-association available in the BGPVPN project yet Due to gap [L3VPN-HS-GAP1], the *[BGPVPN]* project was not yet able to implement the concept of a port association. A port association would allow to associate individual ports with VRFs and thereby control layer 3 connectivity on a per port basis.

The workflow described above intents to mimic port associations by means of separate Neutron networks. Hence, the resulting workflow is overly complicated and not intuitive by requiring to create additional Neutron entities (networks) which are not present in the target topology. Moreover, creating large numbers of Neutron networks limits scalability.

Port associations are on the road map of the *[BGPVPN]* project, however, no design that overcomes the problems outlined above has been specified yet. Consequently, the time-line for this feature is unknown.

As a result, creating a clean Hub-and-Spoke topology is current not yet supported by the *[BGPVPN]* API.

[L3VPN-HS-GAP3] No support for static routes in the BGPVPN project yet In order to realize the hub-and-spoke use case, a static route is needed to attract the traffic at the hub to the corresponding VNF (direct traffic to the firewall). Support for static routes in the BGPVPN project is available for the router association by means of the Neutron router extra routes feature. However, there is no support for static routes for network and port associations yet.

Design work for supporting static routes for network associations has started, but no final design has been proposed yet.

2.2.4 Conclusion

Based on the gap analyses of the three specific L3VPN use cases we conclude that there are gaps in both the functionality provided by the BGPVPN project as well as the support for multiple backends in Neutron.

Some of the identified gaps [L3VPN-ECMP-GAP1, L3VPN-ECMP-GAP2, L3VPN-HS-GAP3] in the BGPVPN project are merely missing functionality which can be integrated in the existing OpenStack networking architecture.

Other gaps, such as the inability to explicitly disable the layer 2 semantics of Neutron networks [L3VPN-HS-GAP1] or the tight integration of ports and networks [L3VPN-HS-GAP2] hinder a clean integration of the needed functionality. In order to close these gaps, fundamental changes in Neutron or alternative approaches need to be investigated.

2.3 Service Binding Design Pattern

2.3.1 Description

This use case aims at binding multiple networks or network services to a single vNIC (port) of a given VM. There are several specific application scenarios for this use case:

- Shared Service Functions: A service function connects to multiple networks of a tenant by means of a single vNIC.

  Typically, a vNIC is bound to a single network. Hence, in order to directly connect a service function to multiple networks at the same time, multiple vNICs are needed - each vNIC binds the service function to a separate network. For service functions requiring connectivity to a large number of networks, this approach does not scale as the number of vNICs per VM is limited and additional vNICs occupy additional resources on the hypervisor.

  A more scalable approach is to bind multiple networks to a single vNIC and let the service function, which is now shared among multiple networks, handle the separation of traffic itself.

- Multiple network services: A service function connects to multiple different network types such as a L2 network, a L3(-VPN) network, a SFC domain or services such as DHCP, IPAM, firewall/security, etc.

In order to achieve a flexible binding of multiple services to vNICs, a logical separation between a vNIC (instance port) - that is, the entity that is used by the compute service as hand-off point between the network and the VM - and a service interface - that is, the interface a service binds to - is needed.

Furthermore, binding network services to service interfaces instead of to the vNIC directly enables a more dynamic management of the network connectivity of network functions as there is no need to add or remove vNICs.

2.3.2 Requirements

Data model

This section describes a general concept for a data model and a corresponding API. It is not intended that these entities are to be implemented exactly as described. Instead, they are meant to show a design pattern for future network service models and their corresponding APIs. For example, the "service" entity should hold all required attributes for a specific service, for instance a given L3VPN service. Hence, there would be no entity "service" but rather "L3VPN".

- `instance-port`

  An instance port object represents a vNIC which is bindable to an OpenStack instance by the compute service (Nova).

  Attributes: Since an instance-port is a layer 2 device, its attributes include the MAC address, MTU and others.

- `interface`

  An interface object is a logical abstraction of an instance-port. It allows to build hierarchies of interfaces by means of a reference to a parent interface. Each interface represents a subset of the packets traversing a given port or parent interface after applying a layer 2 segmentation mechanism specific to the interface type.

  Attributes: The attributes are specific to the type of interface.

  Examples: trunk interface, VLAN interface, VxLAN interface, MPLS interface

- `service`

  A service object represents a specific networking service.

  Attributes: The attributes of the service objects are service specific and valid for given service instance.

  Examples: L2, L3VPN, SFC

- `service-port`

  A service port object binds an interface to a service.

  Attributes: The attributes of a service-port are specific for the bound service.

  Examples: port services (IPAM, DHCP, security), L2 interfaces, L3VPN interfaces, SFC interfaces.

Northbound API

An exemplary API for manipulating the data model is described below. As for the data model, this API is not intended to be a concrete API, but rather an example for a design pattern that clearly separates ports from services and service bindings.

- `instance-port-{create,delete} <name>`

  Creates or deletes an instance port object that represents a vNIC in a VM.

- `interface-{create,delete} <name> [interface type specific parameters]`

  Creates or deletes an interface object.

- `service-{create,delete} <name> [service specific parameters]`

  Create a specific service object, for instance a L3VPN, a SFC domain, or a L2 network.

- `service-port-{create,delete} <service-id> <interface-id> [service specific parameters]`

  Creates a service port object, thereby binding an interface to a given service.

Orchestration

None.

Dependencies on other resources

The compute service needs to be enabled to consume instance ports instead of classic Neutron ports.

2.3.3 Current Implementation

The core Neutron API does not follow the service binding design pattern. For example, a port has to exist in a Neutron network - specifically it has to be created for a particular Neutron network. It is not possible to create just a port and assign it to a network later on as needed. As a result, a port cannot be moved from one network to another, for instance.

Regarding the shared service function use case outlined above, there is an ongoing activity in Neutron *[VLAN-AWARE-VMs]*. The solution proposed by this activity allows for creating a trunk-port and multiple sub-ports per Neutron port which can be bound to multiple networks (one network per sub-port). This allows for binding a single VNIC to multiple networks and allow the corresponding VMs to handle the network segmentation (VLAN tagged traffic) itself. While this is a step in the direction of binding multiple services (networks) to a port, it is limited by the fundamental assumption of Neutron that a port has to exist on a given network.

There are extensions of Neutron that follow the service binding design pattern more closely. An example is the BGPVPN project. A rough mapping of the service binding design pattern to the data model of the BGPVPN project is as follows:

- instance-port -> Neutron port
- service -> VPN
- service-port -> network association

This example shows that extensions of Neutron can in fact follow the described design pattern in their respective data model and APIs.

2.3.4 Conclusions

In conclusion, the design decisions taken for the core Neutron API and data model do not follow the service binding model. As a result, it is hard to implement certain use cases which rely on a flexible binding of services to ports. Due to the backwards compatibility to the large amount of existing Neutron code, it is unlikely that the core Neutron API will adapt to this design pattern.

New extension to Neutron however are relatively free to choose their data model and API - within the architectural boundaries of Neutron of course. In order to provide the flexibility needed, extensions shall aim for following the service binding design pattern if possible.

For the same reason, new networking frameworks complementing Neutron, such as Gluon, shall follow this design pattern and create the foundation for implementing networking services accordingly.

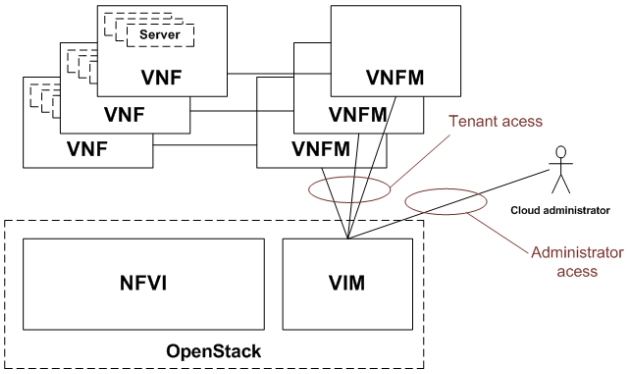
2.4 Programmable Provisioning of Provider Networks

2.4.1 Description

In a NFV environment the VNFMs (Virtual Network Function Manager) are consumers of the OpenStack IaaS API. They are often deployed without administrative rights on top of the NFVI platform. Furthermore, in the telco domain

provider networks are often used. However, when a provider network is created administrative rights are needed what in the case of a VNFM without administrative rights requires additional manual configuration work. It shall be possible to configure provider networks without administrative rights. It should be possible to assign the capability to create provider networks to any roles.

The following figure (Fig. 2.4.1) shows the possible users of an OpenStack API and the relation of OpenStack and ETSI NFV components. Boxes with solid line are the ETSI NFV components while the boxes with broken line are the OpenStack components.



2.4.2 Derived Requirements

- Authorize the possibility of provider network creation based on policy
- There should be a new entry in `policy.json` which controls the provider network creation
- Default policy of this new entry should be `rule:admin_or_owner`.
- This policy should be respected by the Neutron API

Northbound API / Workflow

- No changes in the API

Data model objects

- No changes in the data model

2.4.3 Current implementation

Only admin users can manage provider networks *[OS-NETWORKING-GUIDE-ML2]*.

2.4.4 Potential implementation

- Policy engine shall be able to handle a new provider network creation and modification related policy.
- When a provider network is created or modified neutron should check the authority with the policy engine instead of requesting administrative rights.

2.5 Georedundancy

Georedundancy refers to a configuration which ensures the service continuity of the VNF-s even if a whole datacenter fails.

It is possible that the VNF application layer provides additional redundancy with VNF pooling on top of the georedundancy functionality described here.

It is possible that either the VNFC-s of a single VNF are spread across several datacenters (this case is covered by the OPNFV multi-site project *[MULTISITE]* or different, redundant VNF-s are started in different datacenters.

When the different VNF-s are started in different datacenters the redundancy can be achieved by redundant VNF-s in a hot (spare VNF is running its configuration and internal state is synchronized to the active VNF), warm (spare VNF is running, its configuration is synchronized to the active VNF) or cold (spare VNF is not running, active VNF-s configuration is stored in a persistent, central store and configured to the spare VNF during its activation) standby state in a different datacenter from where the active VNF-s are running. The synchronization and data transfer can be handled by the application or by the infrastructure.

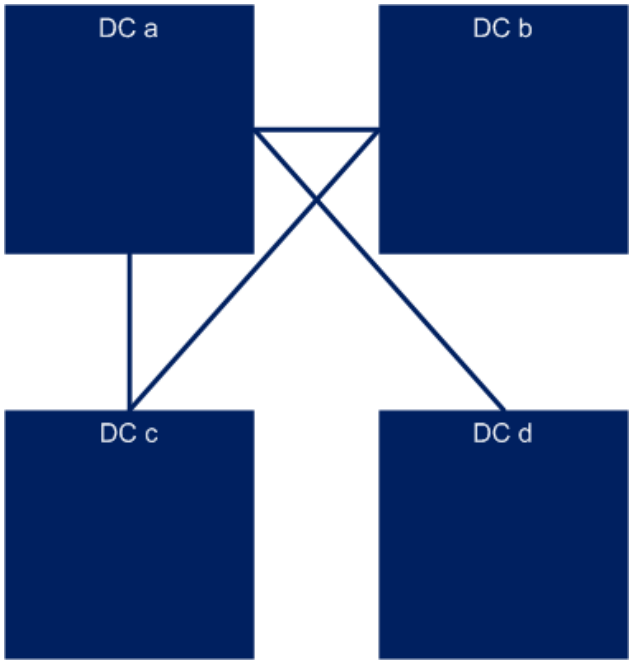
In all of these georedundancy setups there is a need for a network connection between the datacenter running the active VNF and the datacenter running the spare VNF.

In case of a distributed cloud it is possible that the georedundant cloud of an application is not predefined or changed and the change requires configuration in the underlay networks when the network operator uses network isolation. Isolation of the traffic between the datacenters might be needed due to the multi-tenant usage of NFVI/VIM or due to the IP pool management of the network operator.

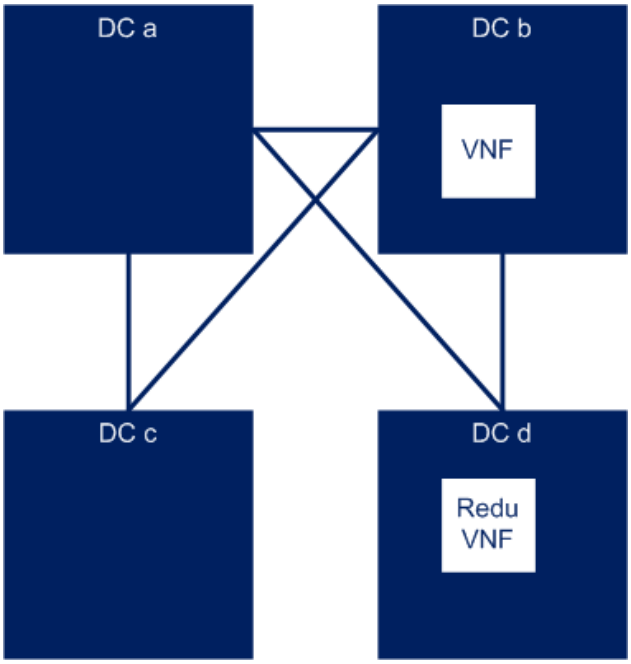
This set of georedundancy use cases is about enabling the possibility to select a datacenter as backup datacenter and build the connectivity between the NFVI-s in the different datacenters in a programmable way.

The focus of these uses cases is on the functionality of OpenStack it is not considered how the provisioning of physical resources is handled by the SDN controllers to interconnect the two datacenters.

As an example the following picture (Fig. 2.5) shows a multi-cell cloud setup where the underlay network is not fully meshed.



Each datacenter (DC) is a separate OpenStack cell, region or instance. Let's assume that a new VNF is started in DC b with a Redundant VNF in DC d. In this case a direct underlay network connection is needed between DC b and DC d. The configuration of this connection should be programmable in both DC b and DC d. The result of the deployment is shown in the following figure (Fig. 2.5):

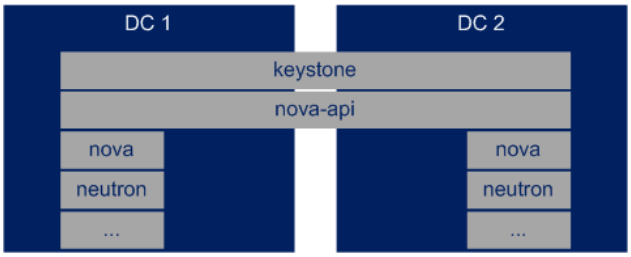


2.5.1 Connection between different OpenStack cells

Description

There should be an API to manage the infrastructure-s networks between two OpenStack cells. (Note: In the Mitaka release of OpenStack cells v1 are considered as experimental, while cells v2 functionality is under implementation). Cells are considered to be problematic from maintainability perspective as the sub-cells are using only the internal message bus and there is no API (and CLI) to do maintenance actions in case of a network connectivity problem between the main cell and the sub cells.

The following figure (Fig. 2.5.1) shows the architecture of the most relevant OpenStack components in multi cell OpenStack environment.



The functionality behind the API depends on the underlying network providers (SDN controllers) and the networking setup. (For example OpenDaylight has an API to add new BGP neighbor.)

OpenStack Neutron should provide an abstracted API for this functionality what calls the underlying SDN controllers API.

Derived Requirements

- Possibility to define a remote and a local endpoint
- As in case of cells the nova-api service is shared it should be possible to identify the cell in the API calls

Northbound API / Workflow

- An infrastructure network management API is needed
- API call to define the remote and local infrastructure endpoints
- When the endpoints are created neutron is configured to use the new network.

Dependencies on compute services

None.

Data model objects

- local and remote endpoint objects (Most probably IP addresses with some

additional properties).

Current implementation

Current OpenStack implementation provides no way to set up the underlay network connection. OpenStack Tricircle project *[TRICIRCLE]* has plans to build up inter datacenter L2 and L3 networks.

Gaps in the current solution

An infrastructure management API is missing from Neutron where the local and remote endpoints of the underlay network could be configured.

2.5.2 Connection between different OpenStack regions or cloud instances

Description

There should be an API to manage the infrastructure-s networks between two OpenStack regions or instances.

The following figure (Fig. 2.5.2) shows the architecture of the most relevant OpenStack components in multi instance OpenStack environment.

The functionality behind the API depends on the underlying network providers (SDN controllers) and the networking setup. (For example OpenDaylight has an API to add new BGP neighbor.)

OpenStack Neutron should provide an abstracted API for this functionality what calls the underlying SDN controllers API.

Derived Requirements

- Possibility to define a remote and a local endpoint
- As in case of cells the nova-api service is shared it should be possible to identify the cell in the API calls

Northbound API / Workflow

- An infrastructure network management API is needed
- API call to define the remote and local infrastructure endpoints
- When the endpoints are created neutron is configured to use the new network.

Data model objects

- local and remote endpoint objects (Most probably IP addresses with some

additional properties).

Current implementation

Current OpenStack implementation provides no way to set up the underlay network connection. OpenStack Tricircle project *[TRICIRCLE]* has plans to build up inter datacenter L2 and L3 networks.

Gaps in the current solution

An infrastructure management API is missing from Neutron where the local and remote endpoints of the underlay network could be configured.

2.5.3 Conclusion

An API is needed what provides possibility to set up the local and remote endpoints for the underlay network. This API present in the SDN solutions, but OpenStack does not provides and abstracted API for this functionality to hide the differences of the SDN solutions.

CHAPTER

THREE

SUMMARY AND CONCLUSION

This document presented the results of the OPNFV NetReady (Network Readiness) project (*[NETREADY]*). It described a selection of NFV-related networking use cases and their corresponding networking requirements. Moreover, for every use case, it describes an associated gap analysis which analyses the aforementioned networking requirements with respect to the current OpenStack networking architecture.

The contents of the current document are the selected use cases and their derived requirements and identified gaps for OPNFV C release.

OPNFV NetReady is open to take any further use cases under analysis in later OPNFV releases. The project backlog (*[NETREADY-JIRA]*) lists the use cases and topics planned to be developed in future releases of OPNFV.

Based on the gap analyses, we draw the following conclusions:

- Besides current requirements and gaps identified in support of NFV networking, more and more new NFV networking services are to be innovated in the near future. Those innovations will bring additional requirements, and more significant gaps will be expected. On the other hand, NFV networking business requires it to be made easy to innovate, quick to develop, and agile to deploy and operate. Therefore, a model-driven, extensible framework is expected to support NFV networking on-demand in order to accelerate time-to-market and achieve business agility for innovations in NFV networking business.
- Neutron networks are implicitly, because of their reliance on subnets, L2 domains. L2 network overlays are the only way to implement Neutron networks because of their semantics. However, L2 networks are inefficient ways to implement cloud networking, and while this is not necessarily a problem for enterprise use cases with moderate traffic it can add expense to the infrastructure of NFV cases where networking is heavily used and efficient use of capacity is key.
- In NFV environment it should be possible to execute network administrator tasks without OpenStack administrator rights.
- In a multi-site setup it should be possible to manage the connection between the sites in a programmable way.

The latest version of this document can be found at *[SELF]*.

[L3VPN-HS-GAP4] Creating a clean hub-and-spoke topology is current not yet supported by the NETWORKING-SFC API.

[BGPVPN] http://docs.openstack.org/developer/networking-bgpvpn/

[MULTISITE] https://wiki.opnfv.org/display/multisite/Multisite

[NETREADY] https://wiki.opnfv.org/display/netready/NetReady

[NETREADY-JIRA] https://jira.opnfv.org/projects/NETREADY/issues/NETREADY-19?filter=allopenissues

[NETWORKING-SFC] https://wiki.openstack.org/wiki/Neutron/ServiceInsertionAndChaining

[NEUTRON-ROUTED-NETWORKS] https://specs.openstack.org/openstack/neutron-specs/specs/newton/routed-networks.html

[OS-NETWORKING-GUIDE-ML2] http://docs.openstack.org/mitaka/networking-guide/config-ml2-plug-in.html

[RFC4364] http://tools.ietf.org/html/rfc4364

[RFC7432] https://tools.ietf.org/html/rfc7432

[SELF] http://artifacts.opnfv.org/netready/docs/requirements/index.html

[TRICIRCLE] https://wiki.openstack.org/wiki/Tricircle#Requirements

[VLAN-AWARE-VMs] https://blueprints.launchpad.net/neutron/+spec/vlan-aware-vms