UNIVERSITY OF THESSALY

Department of Electrical and Computer Engineering

Thesis

Title: "Implementation of Networking Protocols with use of OpenDaylight in SDN"

Panagiotis Karamichailidis

supervised by
Athanasios Korakis
Antonios Argiriou

Volos, 2018

Ευχαριστίες

Με την ολοκλήρωση της διπλωματικής μου εργασίας, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Αθανάσιο Κοράκη για την ευκαιρία που μου έδωσε αναθέτοντας μου το συγκεκριμένο θέμα, καθώς, επίσης, και για την εμπιστοσύνη που μου έδειξε.

Είμαι ευγνώμων στον κ. Κωνσταντίνο Χούμα, η πολύτιμη βοήθειά του και οι γνώσεις που απέκτησα χάρη σε αυτόν, καθ' όλη τη διάρκεια μιας άψογης συνεργασίας, σε πολυάριθμες ώρες διδακτικών συζητήσεων, είναι πολύτιμες. Η επιστημονική και ηθική ενθάρρυνση, αλλά και η συνεισφορά του στην εκπόνηση της παρούσας μελέτης είναι ανεκτίμητη.

Ακόμα, θα ήθελα να ευχαριστήσω όλους τους καθηγητές και συμφοιτητές ανεξαιρέτως για όλα αυτά που μου πρόσφεραν, ο καθένας με το δικό του τρόπο, στα χρόνια φοίτησής μου.

Θα ήθελα να ευχαριστήσω την οικογένεια και τους φίλους μου για την κατανόηση και την ενθάρρυνση που έδειξαν όλο αυτό το διάστημα, αλλά και την οικονομική υποστήριξη που μου παρείχαν όλα αυτά τα χρόνια.

Στην οικογένεια και τους φίλους μου.

Περίληψη Οι προκλήσεις μπροστά από τους διαχειριστές δικτύων είναι τεράστιες καθώς αναμένεται να προκύψουν σε συνάρτηση με τη ζήτηση των πελατών. Οι κύριες προκλήσεις είναι η κάλυψη της αυξανόμενης ζήτησης για εύρος ζώνης (bandwidth) και η ταχεία ανάπτυξη νέων υπηρεσιών για τους πελάτες. Αυτό σημαίνει ότι οι διαχειριστές δικτύων δεν χρειάζονται απλώς ένα κλιμακωτό δίκτυο, αλλά επίσης ένα ¨έξυπνο¨. Εδώ μπαίνει το SDN.

Η απαίτηση για προγραμματιζόμενα δίκτυα, τα οποία μπορούν να καθοριστούν με την ώθηση ενός κλειδιού που εξελίχθηκε μετά από τον πολλαπλασιασμό των προσωπικών συσκευών και των εφαρμογών cloud - δύο από τις μεγαλύτερες τάσεις που μαζί οδηγούν μια βασική αλλαγή στη σχέση μεταξύ της επιχειρηματικής στρατηγικής και της πληροφορικής. Το SDN δίνει την ευκαιρία να επιταχυνθεί η παροχή πληροφοριών καθώς και να μειωθεί το κόστος.

Βασικά, ένα SDN είναι σε συμβατική δικτύωση οτι είναι το σύννεφο (cloud) σε μια συμβατική πλατφόρμα υπολογιστών. Οι μέθοδοι με τις οποίες ελέγχεται το SDN, είναι τελείως διαφορετικές από το υλικό ελέγχου - αυτό επιτρέπει πιο ολοκληρωμένη και πλήρη βελτιστοποίηση του λογισμικού καθώς και του υλικού. Παρέχει επίσης ακριβώς το επίπεδο ευελιξίας και κλιμάκωσης που απαιτείται για την περαιτέρω εξέλιξη του cloud computing.

Εκτός από το επαρκές εύρος ζώνης για συνεχή λειτουργία και τη σωστή τεχνολογία αυτοματισμού, το SDN υποδεικνύει ένα ακόμη βήμα για μια πλήρως ψηφιακή υποδομή τόσο για τους πωλητές όσο και για τους πελάτες. Όσον αφορά τις λειτουργίες δικτύου, τα SDN παρουσιάζουν πολλά από τα παρόμοια πλεονεκτήματα όπως το cloud computing που προσδίδει στην επιχείρηση. Η αυξημένη ευελιξία και η ευελιξία θα επιτρέψουν την αποτελεσματικότερη χρήση των πόρων δικτύωσης, ενώ η μείωση του λειτουργικού κόστους θα μπορούσε ενδεχομένως να οδηγήσει σε ακόμα μεγαλύτερη καινοτομία και σημαντική εξοικονόμηση πόρων για τον πελάτη.

Εξετάστε οποιοδήποτε σύστημα - το σύνολο είναι εξίσου έξυπνο με τα συστατικά του στοιχεία - το σύννεφο δεν αποτελεί εξαίρεση στον κανόνα αυτό.

Ενώ είναι αλήθεια ότι το cloud computing είναι ένα από τα πιο ισχυρά και αποτελεσματικά εργαλεία για οποιαδήποτε επιχείρηση, ταυτόχρονα, το πλήρες δυναμικό του δεν μπορεί να υλοποιηθεί εάν φορτωθεί με συμβατικό υλικό δικτύωσης. Αυτός ακριβώς είναι ο λόγος για τον οποίο η ΣΔΝ έχει τόσο ζωτική και στενή σχέση με το σύννεφο.

Χωρίς το SDN, το cloud computing απλά δεν μπορεί να συνεχίσει την εξέλιξή του και η σύνδεση μεταξύ του cloud computing και του λογισμικού που καθορίζεται από το δίκτυο είναι πολύ ισχυρή.

Οι cloud υποδομές αποτελούνται από πόρους υπολογιστών, αποθήκευσης και δικτύωσης. Όσον αφορά το δίκτυο, η δικτύωση που έχει καθοριστεί από το λογισμικό (SDN) έχει καταστεί μία από τις σημαντικότερες αρχιτεκτονικές για τη διαχείριση δικτύων που απαιτούν συχνή επανεξέταση ή επανεγκατάσταση. Σε αυτή τη διπλωματική εργασία, υπογραμμίζουμε πως το OpenDaylight μπορεί να ενσωματωθεί με το OpenStack για την παροχή μιας λύσης δικτύωσης βασισμένης σε SDN για OpenStack Clouds.

Σκοπός της εργασιάς είναι να αποκτήσουμε γνώσεις σε δύο από τις πιο δημοφιλείς αναδυόμενες τεχνολογίες, συγκεκριμένα το OpenStack και το Software-Defined Networking, και να δούμε πόσο καλά θα μπορούσαν να συνεργαστούν.

Abstract The challenges in front of network operators are vast as they are expected to emerge in pace with the demand by customers. The main challenges are to meet the increasing demand for bandwidth and the quick deployment of new services for customers. This implies that network operators don't just need a scalable network, but also a brilliant one. This is where SDN steps in.

The requirement for programmable networks, which can be stipulated at the push of a key evolved after the proliferation of personal devices and cloud apps { two of the biggest trends that together are driving a basic changeover in the relationship between business strategy and IT. SDN gives a chance to speed up the delivery of information as well as cut costs.

Basically, a SDN is to conventional networking what the cloud is to a conventional computing platform. The methods using which, SDN is controlled, are fully different from the controlling hardware { this permit more comprehensive and complete optimization of software as well as hardware. It also gives precisely the flexibility and scalability level needed for further cloud computing evolution.

In addition to adequate bandwidth for non-stop functioning and the right automation technology, SDN indicates yet another step to a fully digital infrastructure for vendors as well as clients. With respect to network operations, SDNs present many of the similar advantages like cloud computing confers to the enterprise. Enhanced flexibility and agility will permit more efficient use of networking resources, while the decrease in operating costs could possibly result in even greater innovation and significant savings on the client's part.

Consider any system { the whole is just as resourceful as its component elements { the cloud is no exception to this rule.

While it's true that cloud computing is one of the most powerful and efficient tools for any business, at the same time, its complete potential cannot be realized if loaded with conventional networking hardware. This is exactly why SDN has such a vital and close connection with the cloud.

Without SDN, cloud computing just cannot continue its evolution, and the link between cloud computing and software defined networking is very strong.

Cloud infrastructures are composed fundamentally of computing, storage, and networking resources. In regards to network, Software-Defined Networking has become one of the most important architectures for the management of networks that require frequent re-policing or re-configurations. In this thesis, we highlight how OpenDaylight can be integrated with OpenStack to provide a SDN-based networking solution for OpenStack Clouds.

The aim of this work was to gain insights into two of the most popular emerging technologies, namely OpenStack and Software-Defined Networking, and to see how well these two open source solutions could collaborate.

Contents

1 Introduction

1.1 Motivation

Manual hardware configuration is the scourge of the modern data center. Server virtualization and pooled storage have gone a long way toward making infrastructure configurable on the fly via software, but the third leg of the stool, networking, has lagged behind with fragmented technology and standards.

Server virtualization gives cloud infrastructure a part of that flexibility. In order to fully enable the power of cloud computing, networking is required to be dynamic and scalable. Software-Defined Networking (SDN) and Network Function Virtualization (NFV) are two emerging technologies that get a lot of hype in the network industry nowadays, especially SDN. And they are said to deliver the flexibility and agility demanded by cloud computing. On the cloud platform side, OpenStack is also prominent. Since its launch in 2010, OpenStack has quietly become one of the fastest growing open source platforms for enterprise cloud infrastructure.

When both OpenStack and SDN are advancing swiftly and becoming platforms for innovation, it is important to understand the key technologies at their intersection. The goal of this work is to gain technology and implementation insights into OpenStack and SDN and to see how well these open source software applications can play together.

1.2 Thesis Outline

In this thesis, we present two new technologies in the network industry, cloud computing and software define networking. Furthermore, we show how these technologies can collaborate with each other to create network solutions. Below, we enlist the included chapters below, as well as a summary of their content.

In **Section 2** we present the OpenDaylight, which is a SDN controller, what is the architecture and how it works. Furthermore, we take a closer look to SDN and the OpenFlow protocol. Finally, we analyze a feature of the controller, Virtual Tenant Network (VTN).

In **Section 3** we present the OpenStack, a software platform for cloud computing, the components and the architecture. In addition, we determine Network Functions Virtualization (NFV) and Virtual Infrastructure Manager (VIM). At last, we see some deployment methods of OpenStack.

In **Section 4** we present the integration of OpenDaylight into OpenStack. How it can be done and how it works.

In **Section 5** we present the SDN assist of OpenStack. What is it and how can be used with OpenStack.

In **Section 6** we present the software that we developed and a demo that demonstrates it.

2 OpenDaylight

2.1 Introduction

The OpenDaylight project is an open source project hosted by the Linux Foundation with the mutual goal of furthering the adoption and innovation of Software Defined Networking (SDN) through the creation of a common industry supported framework. Rather than hammer out new standards, the project aims to produce an extensible, open source, virtual networking platform atop such existing standards as OpenFlow.

OpenDaylight provides a model-driven service abstraction platform that allows users to write applications that easily work across a wide variety of hardware and south-bound protocols.

The approach of OpenDaylight is similar to that of OpenStack, where industry players come together to develop core open source bits collaboratively, around which participants can add unique value. That roughly describes the Linux model as well, which may help explain why the Linux Foundation is hosting OpenDaylight.

2.2 Software Defined Networking (SDN)

Software-Defined Networking (SDN) is a network architecture approach that enables the network to be intelligently and centrally controlled, or "programmed", using software applications. This helps operators manage the entire network consistently and holistically, regardless of the underlying network technology.

SDN is meant to address the fact that the static architecture of traditional networks is decentralized and complex while current networks require more flexibility and easy troubleshooting. SDN suggests to centralize network intelligence in one network component by disassociating the forwarding process of network packets (Data Plane) from the routing process (Control plane). The control plane consists of one or more controllers which are considered as the brain of SDN network where the whole intelligence is incorporated.

An SDN can be considered a series of network objects – such as switches, routers and firewalls – that are deployed in a highly automated manner. The automation may be achieved by using commercial or open source tools – like SDN controllers and OpenFlow – based on the administrator's requirements. A full software-defined network may cover only relatively straightforward networking requirements, such as VLAN and interface provisioning.

All software-defined network solutions have some version of an SDN Controller, as well as southbound APIs and northbound APIs:

- **Controllers:** The "brains" of the network, SDN Controllers offer a centralized view of the overall network, and enable network administrators to dictate to the underlying systems (like switches and routers) how the forwarding plane should handle network traffic.
- **Southbound APIs:** Software-defined networking uses southbound APIs to relay information to the switches and routers "below". OpenFlow, considered the first standard in SDN, was the original southbound API and remains as one of the most common protocols. Despite some considering OpenFlow and SDN to be one in the same, OpenFlow is merely one piece of the bigger landscape.
- **Northbound APIs:** Software-Defined Networking uses northbound APIs to communicates with the applications and business logic "above" These help network administrators to programmatically shape traffic and deploy services.

2.3 OpenFlow

OpenFlow is an open standard for a communications protocol that enables the control plane to interact with the forwarding plane.

The OpenFlow switch protocol provides an open interface for controlling connectivity and flows within that connectivity in a SDN. OpenFlow is an extensible protocol, providing mechanisms for SDN programmers to define additional protocol elements (e.g., new match fields, actions, port properties, etc.) to address new network technologies and behaviors. OpenFlow Table Type Patterns are a vehicle for describing an OpenFlow controllable datapath, allowing switch and controller vendors to work independently to create interoperable SDN products.

To work in an OF environment, any device that wants to communicate to an SDN Controller must support the OpenFlow protocol. Through this interface, the SDN Controller pushes down changes to the switch/router flow-table allowing network administrators to partition traffic, control flows for optimal performance, and start testing new configurations and applications.

2.4 Mininet

Mininet creates a realistic virtual network, running real kernel, switch and application code, on a single machine (VM, cloud or native).

Mininet provides a virtual test bed and development environment for software-defined networks (SDN). Mininet enables SDN development on any laptop or PC, and SDN designs can move seamlessly between Mininet (allowing inexpensive and streamlined development), and the real hardware running at line rate in live deployments. Mininet enables

- Rapid prototyping of software-defined networks
- Complex topology testing without the need to wire up a physical network
- Multiple concurrent developers to work independently on the same topology

In addition, provides an extensible Python API for network creation and experimentations. It is released under a permissive BSD Open Source license and is actively developed and supported by community of networking and SDN enthusiasts.

Mininet networks run real code including standard Unix/Linux network applications as well as the real Linux kernel and network stack.

2.5 OpenDaylight

2.5.1 Concepts and Tools

To date, OpenDaylight developers have formed more than 50 projects to address ways to extend network functionality. The projects are a formal structure for developers from the community to meet, document release plans, code, and release the functionality they create in an OpenDaylight release.

Apache Karaf provides a lightweight runtime to install the Karaf features you want to implement and is included in the OpenDaylight platform software. By default, OpenDaylight has no pre-installed features.

After installing OpenDaylight, you install your selected features using the Karaf console to expand networking capabilities.

Model-Driven Service Abstraction Layer (MD-SAL) is the OpenDaylight framework that allows developers to create new Karaf features in the form of services and protocol drivers and connects them to one another. You can think of the MD-SAL as having the following two components:

1. A shared datastore that maintains the following tree-based structures:
   (a) The Config Datastore, which maintains a representation of the desired network state.
   (b) The Operational Datastore, which is a representation of the actual network state based on data from the managed network elements.
2. A message bus that provides a way for the various services and protocol drivers to notify and communicate with one another.

If you're interacting with OpenDaylight through DLUX or the REST APIs while using the the OpenDaylight interfaces, the microservices architecture allows you to select available services, protocols, and REST APIs.

2.5.2 Controller Overview

The OpenDaylight controller is Java Virtual Machine (JVM) software and can be run from any operating system and hardware as long as it supports Java. The controller is an implementation of the SDN concept and makes use of the following tools:

- **Maven:** OpenDaylight uses Maven for easier build automation. Maven uses pom.xml (Project Object Model) to script the dependencies between bundle and also to describe what bundles to load and start.
- **OSGi:** This framework is the back-end of OpenDaylight as it allows dynamically loading bundles and packages JAR files, and binding bundles together for exchanging information.
- **JAVA interfaces:** Java interfaces are used for event listening, specifications, and forming patterns. This is the main way in which specific bundles implement call-back functions for events and also to indicate awareness of specific state.
- **REST APIs:** These are northbound APIs such as topology manager, host tracker, flow programmer, static routing, and so on.

The controller exposes open northbound APIs which are used by applications. The OSGi framework and bidirectional REST are supported for the northbound APIs. The OSGi framework is used for applications that run in the same address space as the controller while the REST (web-based) API is used for applications that do not run in the same address space (or even the same system) as the controller. The business logic and algorithms reside in the applications. These applications use the controller to gather network intelligence, run its algorithm to do analytics, and then orchestrate the new rules throughout the network. On the southbound, multiple protocols are supported as plugins, e.g. OpenFlow 1.0, OpenFlow 1.3, BGP-LS, and so on. The OpenDaylight controller starts with an OpenFlow 1.0 southbound plugin. Other OpenDaylight contributors begin adding to the controller code. These modules are linked dynamically into a Service Abstraction Layer (SAL).

The SAL exposes services to which the modules north of it are written. The SAL figures out how to fulfill the requested service irrespective of the underlying protocol used between the controller and the network devices. This provides investment protection to the applications as OpenFlow and other protocols evolve over time. For the controller to control devices in its domain, it needs to know about the devices, their capabilities, reachability, and so on. This information is stored and managed by the Topology Manager. The other components like ARP handler, Host Tracker, Device Manager, and Switch Manager help in generating the topology database for the Topology Manager.

2.5.3 Architecture

OpenDaylight components include a fully pluggable controller, interfaces, protocol plug-ins, and applications. The controller consists of three key blocks:

- The controller platform
- Northbound applications and services
- Southbound plugins and protocols

Here you can see the overall architecture setup and components:

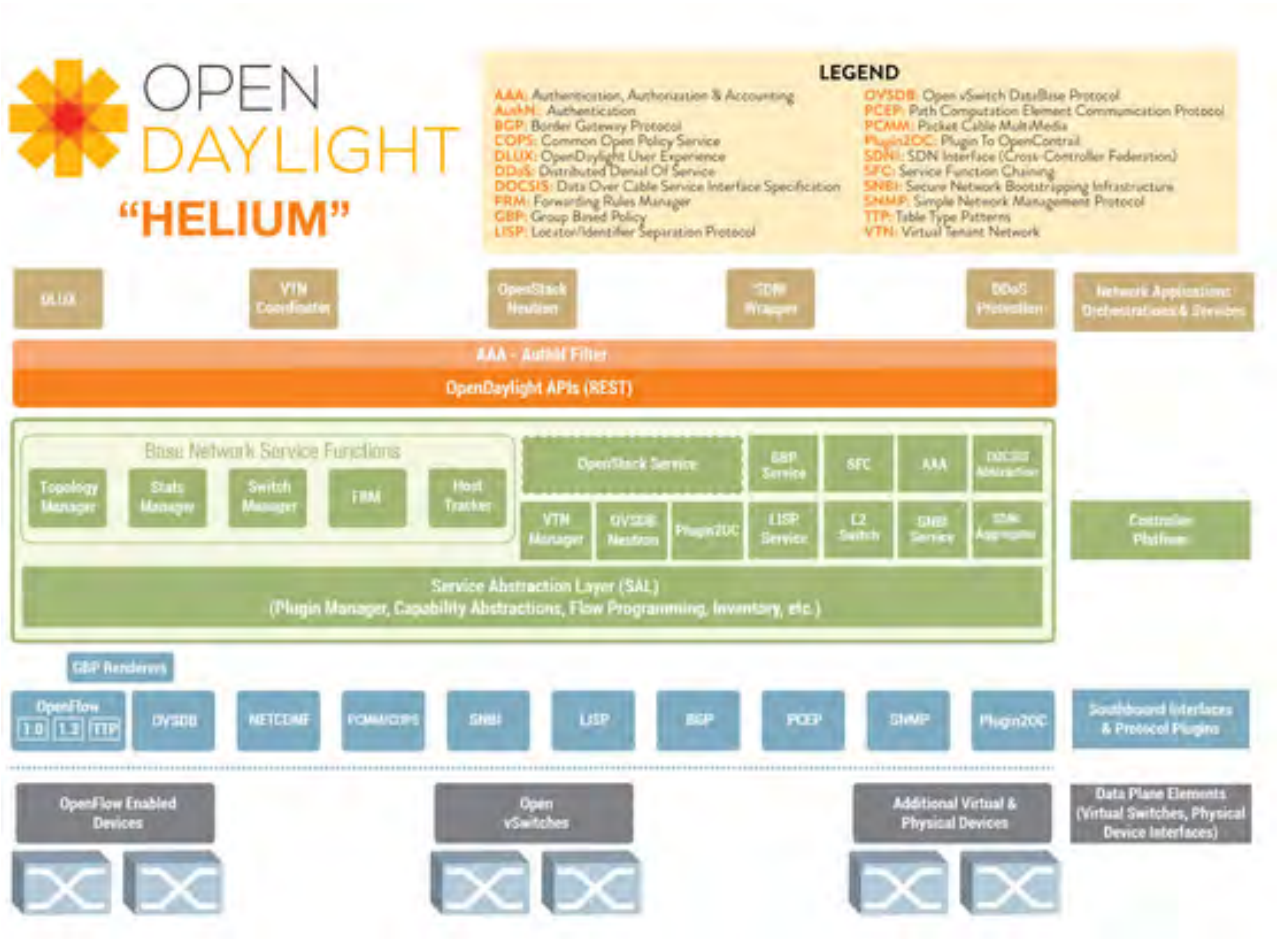

Figure 1: OpenDaylight architecture and components

2.5.4 Controller Platform

The controller platform is a modular layer (as shown in Figure 2) and has a "northbound" and "southbound" interface. The northbound interface provides controller services and a set of common REST APIs that applications can leverage to manage networking infrastructure configuration. You can access the northbound interface using the authentication and authorization models in the AAA project shown as the top layer of the OpenDaylight architecture in Figure 1.

The southbound interface implements protocols to manage and control the underlying networking infrastructure. The southbound level has multiple plugins that either implement various networking protocols or directly communicate with hardware. OpenFlow, NetConf, and SNMP are the best known and most widely used configuration and management protocols.

The controller platform communicates with the underlying network infrastructure using southbound plugins and provides basic networking services via a set of managers displayed in the Base Network Service

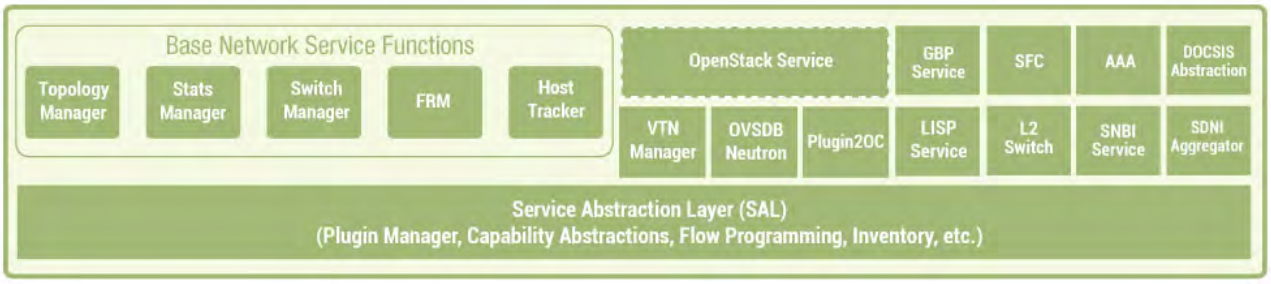

Figure 2: Controller components

Functions section of Figure 2, including Topology Manager, Switch Manager, and so on. Any custom application can use these network services; for example, OpenStack Networking (Neutron) can use northbound APIs and related components.

2.5.5 Base Network Service Functions

The Base Network Service Functions are provided by the following platform managers and components:

- Topology Manager - stores and handles information about the managed networking devices. When the controller starts, the Topology Manager creates the root node in the topology operational subtree. Then it listens for notifications and updates this subtree with topology details, including all discovered switches and their interconnections. Notifications from other components, such as the Switch Manager or Device Manager, may also provide relevant topology information.
- Statistics Manager - implements statistics collection, sending statistics requests to all enabled nodes (managed switches) and storing responses in the statistics operational subtree. The Statistics Manager also exposes northbound APIs to return information on the following:
  - { node-connector (switch port)
  - { flow
  - { meter
  - { table
  - { group statistics
- Switch Manager - provides network nodes (switches) and node connectors (switch ports) details. As soon as the controller discovers network components, their parameters are saved to the Switch Manager data tree . You can use northbound APIs to get information on the discovered nodes and port devices.
- Forwarding Rules Manager (FRM) - manages basic forwarding rules (such as OpenFlow rules), resolves their conflicts, and validates them. The Forwarding Rules Manager communicates with southbound plugins and loads OpenFlow rules into the managed switches.
- Inventory Manager - queries and updates information about switches and ports managed by OpenDaylight, guaranteeing that the inventory database is accurate and up-to-date.
- Host Tracker - stores information about the end hosts (data layer address, switch type, port type, network address), and provides APIs that retrieve end node information. Host Tracker may work in a static or dynamic way. In case of dynamic operation, the Host Tracker uses ARP to track the status of the database. In static mode, the Host Tracker database is populated manually via northbound APIs.

2.5.6 Other services

Apart from the core network services above, a number of other services are on the same architectural layer, interact with core controller modules, and provide specific features for appropriate services and modules.

2.5.7 Networking Services Abstractions

The central concept of the OpenDaylight controller is the Service Abstraction Layer (SAL), which connects the protocol plugins and Service Network Function Modules. Because the original API-Driven SAL (AD-SAL) approach proved ineffective, OpenDaylight and all dependent projects are migrating to Model-Driven SAL (MD-SAL), which is now our focus.

2.5.8 How the Model-Driven SAL works

The Model-Driven SAL provides a common approach to plugin development, enabling unification between both northbound and southbound APIs and the data structures used in various components of the controller. In MD-SAL, all status-related data is stored in the form of a document object model (DOM), known as a "data tree". The following two types of data trees are used in the OpenDaylight Controller:

- The operational DOM tree, which controller modules use to store certain temporary runtime information
- The configuration DOM tree, used to store the current status of the controller.

MD-SAL uses YANG, a single schema language, as a modeling language for describing all network services. YANG makes it easy to develop any controller-related core module, northbound application, or platform component providing built-in structures and types, such as containers, lists, and leaves, while a developer may specify any additional structures for certain tasks' data types. After you define the YANG models, a compiler outputs appropriate Java interfaces, and the next step is to implement those auto-generated Java interfaces.

Now let's look at the example of the southbound plugin and controller operations. In this case, we'll look at the OpenFlow plugin and its interaction with MD-SAL platform services, seen in Figure 3.

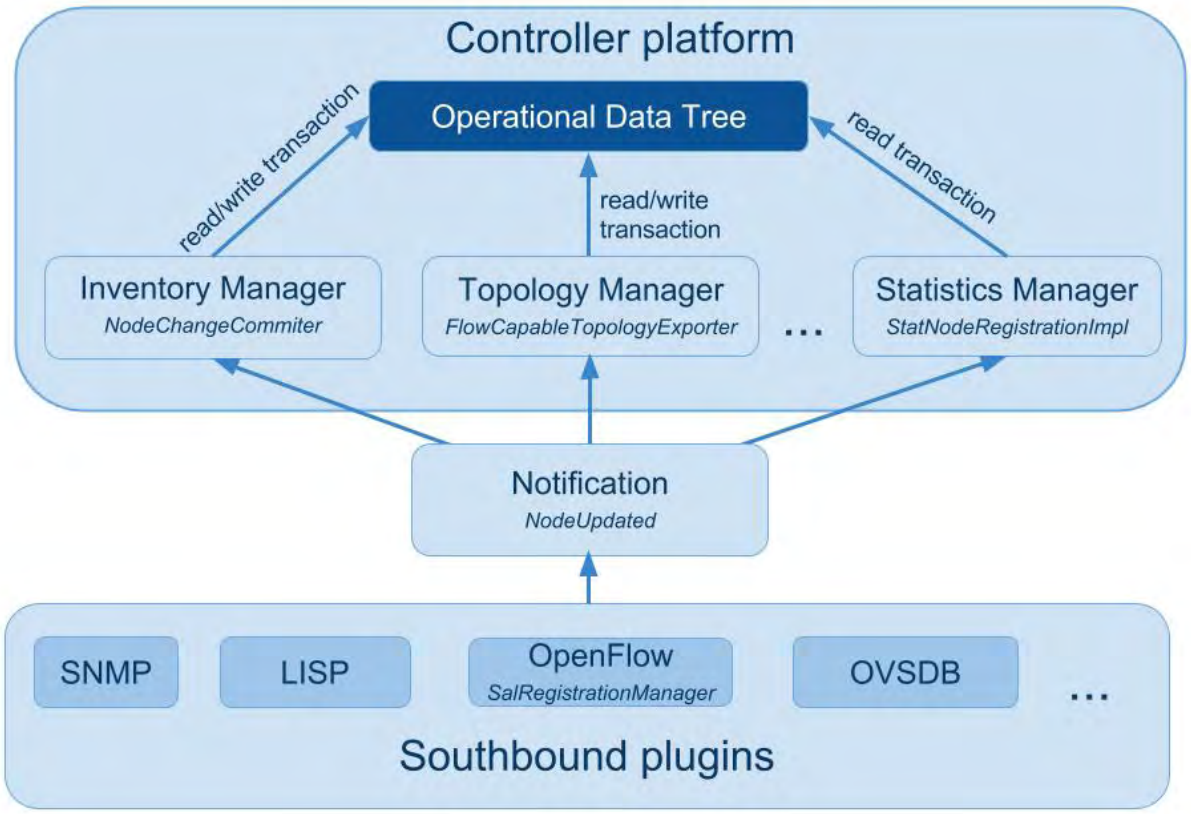

Figure 3: Interaction between southbound plugin and controller

In this example, when the switch attempts to register with the controller, it first registers with the southbound plugin, which produces a NodeUpdated notification. All controller modules - Topology Manager, Inventory Manager, Statistics Manager - listen for the NodeUpdated notification. Actions after receiving notifications include:

- Inventory Manager adds the new node (switch) to the operational data tree
- Statistics Manager gets the updated information from the inventory database via read transactions to the operational data tree
- Topology Manager updates the corresponding topology DOM subtree

2.6 Virtual Tenant Network (VTN)

2.6.1 VTN Overview

OpenDaylight Virtual Tenant Network (VTN) is an application that provides multi-tenant virtual network on an SDN controller.

Conventionally, huge investment in the network systems and operating expenses are needed because the network is configured as a silo for each department and system. So, various network appliances must be installed for each tenant and those boxes cannot be shared with others. It is a heavy work to design, implement and operate the entire complex network.

The uniqueness of VTN is a logical abstraction plane. This enables the complete separation of logical plane from physical plane. Users can design and deploy any desired network without knowing the physical network topology or bandwidth restrictions.

VTN allows the users to define the network with a look and feel of conventional L2/L3 network. Once the network is designed on VTN, it will automatically be mapped into underlying physical network, and then configured on the individual switch leveraging SDN control protocol. The definition of logical plane makes it possible not only to hide the complexity of the underlying network but also to better manage network resources. It achieves reducing reconfiguration time of network services and minimizing network configuration errors.

It is implemented as two major components

- VTN Manager
- VTN Coordinator

2.6.2 VTN Manager

An OpenDaylight Plugin that interacts with other modules to implement the components of the VTN model. It also provides a REST interface to configure VTN components in OpenDaylight. VTN Manager is implemented as one plugin to the OpenDaylight. This provides a REST interface to create/update/delete VTN components. The user command in VTN Coordinator is translated as REST API to VTN Manager by the OpenDaylight Driver component. In addition to the above mentioned role, it also provides an implementation to the OpenStack L2 Network Functions API.

2.6.3 VTN Coordinator

VTN Coordinator is an external application that provides a REST interface for an user to use OpenDaylight VTN Virtualization. It interacts with VTN Manager plugin to implement the user configuration. It is also capable of multiple OpenDaylight orchestration. It realizes Virtual Tenant Network (VTN) provisioning in OpenDaylight instances. In the OpenDaylight architecture VTN Coordinator is part of the network application, orchestration and services layer. VTN Coordinator will use the REST interface exposed by the VTN Manger to realize the virtual network using OpenDaylight. It uses OpenDaylight APIs (REST) to construct the virtual network in OpenDaylight instances. It provides REST APIs for northbound VTN applications and supports virtual networks spanning across multiple OpenDaylight by coordinating across OpenDaylight.

2.6.4 Network Virtualization Function

The user first defines a VTN. Then, the user maps the VTN to a physical network, which enables communication to take place according to the VTN definition. With the VTN definition, L2 and L3 transfer functions and flow-based traffic control functions (filtering and redirect) are possible.

2.6.5 Virtual Network Construction

The following table shows the elements which make up the VTN. In the VTN, a virtual network is constructed using virtual nodes (vBridge, vRouter) and virtual interfaces and links. It is possible to configure a network which has L2 and L3 transfer function, by connecting the virtual interfaces made on virtual nodes via virtual links.

vBridge	The logical representation of L2 switch function.
vRouter	The logical representation of router function.
vTep	The logical representation of Tunnel End Point - TEP.
vTunnel	The logical representation of Tunnel.
vBypass	The logical representation of connectivity between controlled networks.
Virtual interface	The representation of end point on the virtual node.
Virtual Linkv(vLink)	The logical representation of L1 connectivity between virtual interfaces.

The following figure(Figure 4) shows an example of a constructed virtual network. VRT is defined as the vRouter, BR1 and BR2 are defined as vBridges. interfaces of the vRouter and vBridges are connected using vLinks.

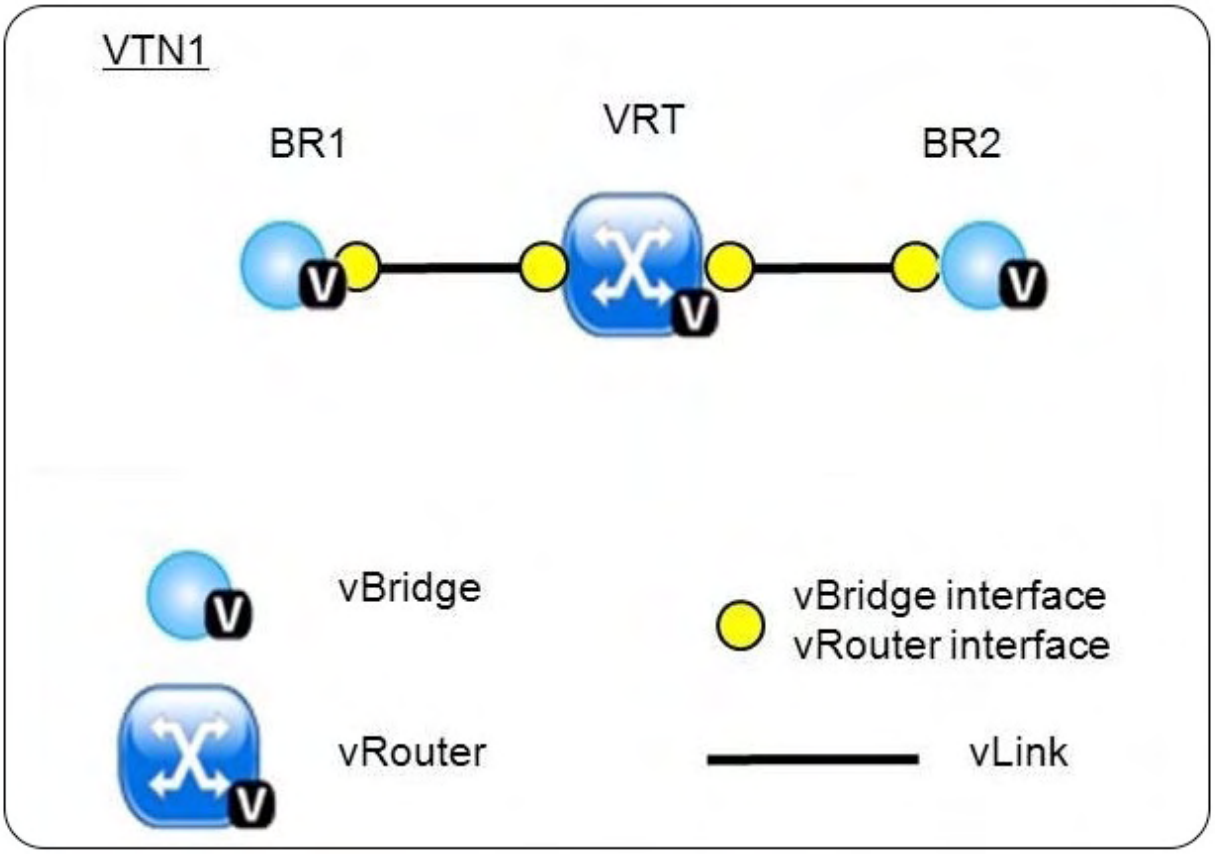

Figure 4: VTN Construction

2.6.6 Mapping of Physical Network Resources

Map physical network resources to the constructed virtual network. Mapping identifies which virtual network each packet transmitted or received by an OpenFlow switch belongs to, as well as which interface in the OpenFlow switch transmits or receives that packet. There are two mapping methods. When a packet is received from the OFS, port mapping is first searched for the corresponding mapping definition, then VLAN mapping is searched, and the packet is mapped to the relevant vBridge according to the first matching mapping.

Port mapping	Maps physical network resources to an interface of vBridge using Switch ID, Port ID and VLAN ID of the incoming L2 frame. Untagged frame mapping is also supported.
VLAN mapping	Maps physical network resources to a vBridge using VLAN ID of the incoming L2 frame.Maps physical resources of a particular switch to a vBridge using switch ID and VLAN ID of the incoming L2 frame.
MAC mapping	Maps physical resources to an interface of vBridge using MAC address of the incoming L2 frame(The initial contribution does not include this method).

VTN can learn the terminal information from a terminal that is connected to a switch which is mapped to VTN. Further, it is possible to refer that terminal information on the VTN.

- Learning terminal information VTN learns the information of a terminal that belongs to VTN. It will store the MAC address and VLAN ID of the terminal in relation to the port of the switch.
- Aging of terminal information Terminal information, learned by the VTN, will be maintained until the packets from terminal keep flowing in VTN. If the terminal gets disconnected from the VTN, then the aging timer will start clicking and the terminal information will be maintained till timeout.

The following figure(Figure 5)shows an example of mapping. An interface of BR1 is mapped to port GBE0/1 of OFS1 using port mapping. Packets received from GBE0/1 of OFS1 are regarded as those from the corresponding interface of BR1. BR2 is mapped to VLAN 200 using VLAN mapping. Packets with VLAN tag 200 received from any ports of any OFSs are regarded as those from an interface of BR2.

2.6.7 vBridge Functions

The vBridge provides the bridge function that transfers a packet to the intended virtual port according to the destination MAC address. The vBridge looks up the MAC address table and transmits the packet to the corresponding virtual interface when the destination MAC address has been learned. When the destination MAC address has not been learned, it transmits the packet to all virtual interfaces other than the receiving port (flooding). MAC addresses are learned as follows.

- MAC address learning The vBridge learns the MAC address of the connected host. The source MAC address of each received frame is mapped to the receiving virtual interface, and this MAC address is stored in the MAC address table created on a per-vBridge basis.
- MAC address aging The MAC address stored in the MAC address table is retained as long as the host returns the ARP reply. After the host is disconnected, the address is retained until the aging timer times out. To have the vBridge learn MAC addresses statically, you can register MAC addresses manually.

2.6.8 vRouter Functions

The vRouter transfers IPv4 packets between vBridges. The vRouter supports routing, ARP learning, and ARP aging functions. The following outlines the functions.

- Routing function When an IP address is registered with a virtual interface of the vRouter, the default routing information for that interface is registered. It is also possible to statically register routing information for a virtual interface.
- ARP learning function The vRouter associates a destination IP address, MAC address and a virtual interface, based on an ARP request to its host or a reply packet for an ARP request, and maintains this information in an ARP table prepared for each routing domain. The registered ARP entry is retained until the aging timer, described later, times out. The vRouter transmits an ARP request on an individual aging timer basis and deletes the associated entry from the ARP table if no reply is returned. For static ARP learning, you can register ARP entry information manually.
- DHCP relay agent function The vRouter also provides the DHCP relay agent function.

2.6.9 Flow Filter Functions

Flow Filter function is similar to Access Control Lists (ACL). It is possible to allow or prohibit communication with only certain kind of packets that meet a particular condition. Also, it can perform a processing called Redirection - WayPoint routing, which is different from the existing ACL. Flow Filter can be applied to any

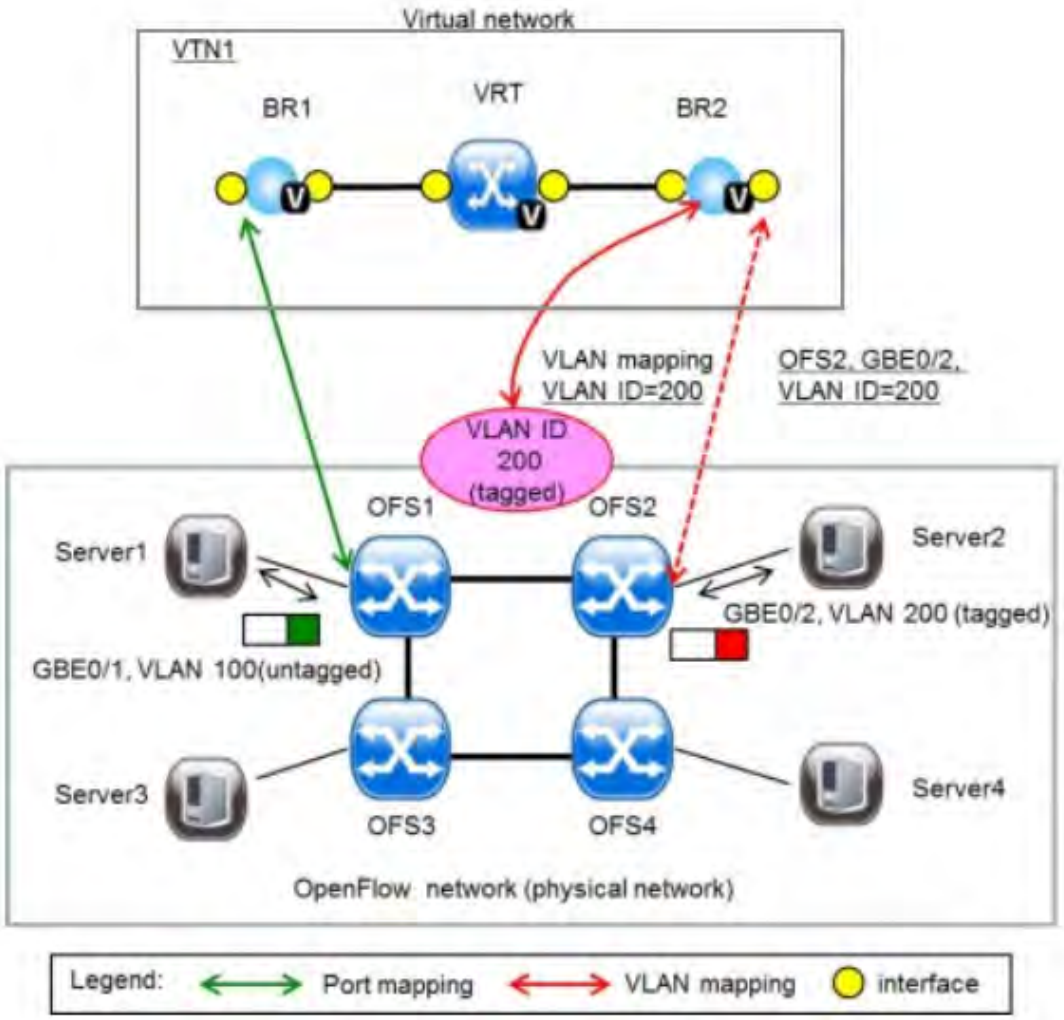

Figure 5: VTN Mapping

interface of a vNode within VTN, and it is possible to the control the packets that pass interface. The match conditions that could be specified in Flow Filter are as follows. It is also possible to specify a combination of multiple conditions.

- Source MAC address
- Destination MAC address
- MAC ether type
- VLAN Priority
- Source IP address
- Destination IP address
- DSCP
- IP Protocol
- TCP/UDP source port
- TCP/UDP destination port

- ICMP type
- ICMP code

The types of Action that can be applied on packets that match the Flow Filter conditions are given in the following table. It is possible to make only those packets, which match a particular condition, to pass through a particular server by specifying Redirection in Action. E.g., path of flow can be changed for each packet sent from a particular terminal, depending upon the destination IP address. VLAN priority control and DSCP marking are also supported.

Action	Function
Pass	Pass particular packets matching the specified conditions.
Drop	Discards particular packets matching the specified conditions.
Redirection	Redirects the packet to a desired virtual interface. Both Transparent Redirection (not changing MAC address) and Router Redirection (changing MAC address) are supported.

The following figure(Figure 6) shows an example of how the flow filter function works.

If there is any matching condition specified by flow filter when a packet being transferred within a virtual network goes through a virtual interface, the function evaluates the matching condition to see whether the packet matches it. If the packet matches the condition, the function applies the matching action specified by flow filter. In the example shown in the Figure 6, the function evaluates the matching condition at BR1 and discards the packet if it matches the condition.

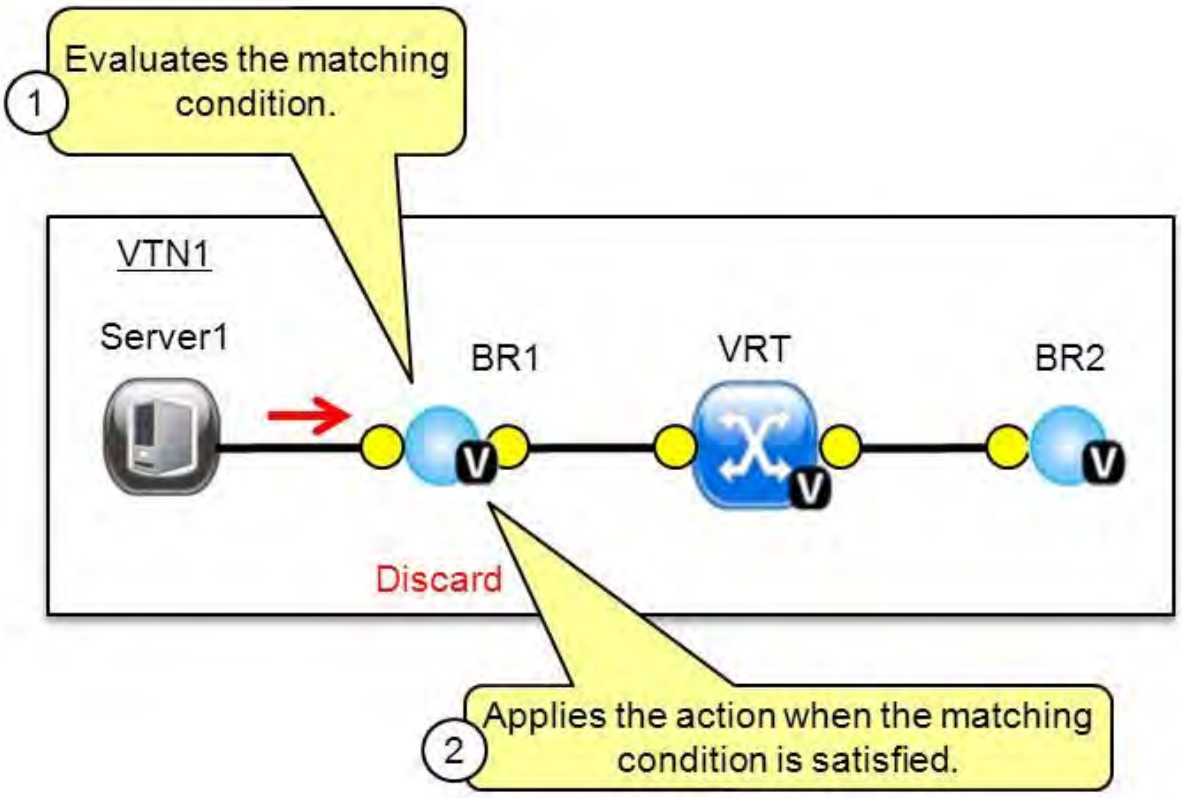

Figure 6: VTN Flow Filter

3 OpenStack

3.1 Introduction

OpenStack is a set of software tools for building and managing cloud computing platforms for public and private clouds. Backed by some of the biggest companies in software development and hosting, as

well as thousands of individual community members, many think that OpenStack is the future of cloud computing. OpenStack is managed by the OpenStack Foundation, a non-profit that oversees both development and community-building around the project.

OpenStack lets users deploy virtual machines and other instances that handle different tasks for managing a cloud environment on the fly. It makes horizontal scaling easy, which means that tasks that benefit from running concurrently can easily serve more or fewer users on the fly by just spinning up more instances. For example, a mobile application that needs to communicate with a remote server might be able to divide the work of communicating with each user across many different instances, all communicating with one another but scaling quickly and easily as the application gains more users.

And most importantly, OpenStack is open source software, which means that anyone who chooses to can access the source code, make any changes or modifications they need, and freely share these changes back out to the community at large. It also means that OpenStack has the benefit of thousands of developers all over the world working in tandem to develop the strongest, most robust, and most secure product that they can.

The cloud is all about providing computing for end users in a remote environment, where the actual software runs as a service on reliable and scalable servers rather than on each end-user's computer. Cloud computing can refer to a lot of different things, but typically the industry talks about running different items "as a service" | software, platforms, and infrastructure. OpenStack falls into the latter category and is considered Infrastructure as a Service (IaaS). Providing infrastructure means that OpenStack makes it easy for users to quickly add new instance, upon which other cloud components can run. Typically, the infrastructure then runs a "platform" upon which a developer can create software applications that are delivered to the end users.

3.2 Network Functions Virtualization (NFV)

Network functions virtualization (NFV) (also known as virtual network function (VNF)) offers a new way to design, deploy and manage networking services. NFV decouples the network functions, such as network address translation (NAT), firewalling, intrusion detection, domain name service (DNS), and caching, to name a few, from proprietary hardware appliances so they can run in software.

It's designed to consolidate and deliver the networking components needed to support a fully virtualized infrastructure { including virtual servers, storage, and even other networks. It utilizes standard IT virtualization technologies that run on high-volume service, switch and storage hardware to virtualize network functions. It is applicable to any data plane processing or control plane function in both wired and wireless network infrastructures

NFV virtualizes network services via software to enable operators to:

- **Reduce CapEx:** reducing the need to purchase purpose-built hardware and supporting pay-as-you-grow models to eliminate wasteful over-provisioning.
- **Reduce OpEx:** reducing space, power and cooling requirements of equipment and simplifying the roll out and management of network services.
- **Accelerate Time-to-Market:** reducing the time to deploy new networking services to support changing business requirements, seize new market opportunities and improve return on investment of new services. Also lowers the risks associated with rolling out new services, allowing providers to easily trial and evolve services to determine what best meets the needs of customers.
- **Deliver Agility and Flexibility:** quickly scale up or down services to address changing demands; support innovation by enabling services to be delivered via software on any industry-standard server hardware.

3.3 Virtual Infrastructure Manager (VIM)

The Management and Organization Working Group of the European Telecommunications Standards Institute (ETSI) has defined the network functions virtualization management and orchestration (NFV-MANO) architecture, comprising three major functional blocks: VIM manager, VNF manager, and NFV orchestrator. The virtualized infrastructure manager (VIM) is a key component of the NFV-MANO architectural framework. It is responsible for controlling and managing the NFV infrastructure (NFVI) compute, storage, and network resources, usually within one operator's infrastructure domain.

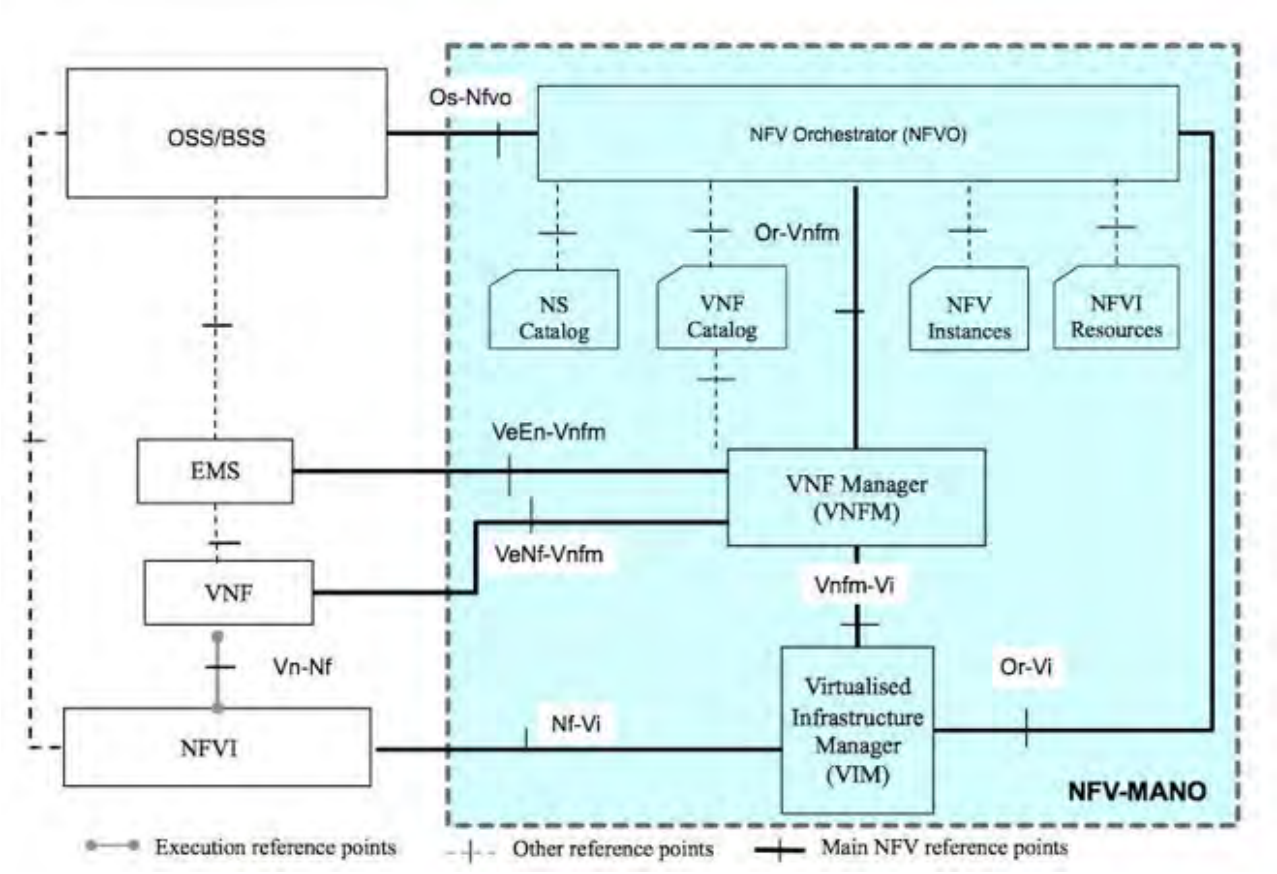

Figure 7: NFV Architecture

These functional blocks help standardize the functions of virtual networking to increase interoperability of software-defined networking elements. VIMs can also handle hardware in a multidomain environment or may be optimized for a specific NFVI environment.

The VIM is responsible for managing the virtualized infrastructure of an NFV-based solution. VIM operations include:

- It keeps an inventory of the allocation of virtual resources to physical resources. This allows the VIM to orchestrate the allocation, upgrade, release, and reclamation of NFVI resources and optimize their use.
- It supports the management of VNF forwarding graphs by organizing virtual links, networks, subnets, and ports. The VIM also manages security group policies to ensure access control.
- It manages a repository of NFVI hardware resources (compute, storage, networking) and software resources (hypervisors), along with the discovery of the capabilities and features to optimize the use of such resources.

The VIM performs other functions as well { such as collecting performance and fault information via notifications; managing software images (add, delete, update, query, copy) as requested by other NFV-MANO functional blocks; and managing catalogues of virtualized resources that can be consumed from the NFVI. In summary, the VIM is the management glue between hardware and software in the NFV world.

Virtual infrastructure managers are critical to realizing the business benefits enabled by the NFV architecture. They coordinate the physical resources necessary to deliver network services. This is particularly visible for infrastructure-as-a-service (IaaS) providers. The IaaS providers have to ensure that their servers, networks, and storage work smoothly with those onsite. They must ensure that resources can be dynamically allocated based on requirements, which is a key feature of cloud computing.

VIMs address this need. Some consider it a morphing of a traditional OS, but it is not. It doesn't work with a single node, but collects information from many machines simultaneously and uses that information for management functions. So, even if multiple machines are working in concert at the NFVI layer, applications and users are ensured of a good, uniform experience.

3.4 OpenStack Components

OpenStack is made up of many different moving parts. Because of its open nature, anyone can add additional components to OpenStack to help it to meet their needs. But the OpenStack community has collaboratively identified nine key components that are a part of the "core" of OpenStack, which are distributed as a part of any OpenStack system and officially maintained by the OpenStack community.

- **Nova** is the primary computing engine behind OpenStack. It is used for deploying and managing large numbers of virtual machines and other instances to handle computing tasks.
- **Switft** is a storage system for objects and files. Rather than the traditional idea of a referring to files by their location on a disk drive, developers can instead refer to a unique identifier referring to the file or piece of information and let OpenStack decide where to store this information. This makes scaling easy, as developers don't have the worry about the capacity on a single system behind the software. It also allows the system, rather than the developer, to worry about how best to make sure that data is backed up in case of the failure of a machine or network connection.
- **Cinder** is a block storage component, which is more analogous to the traditional notion of a computer being able to access specific locations on a disk drive. This more traditional way of accessing files might be important in scenarios in which data access speed is the most important consideration.
- **Neutron** provides the networking capability for OpenStack. It helps to ensure that each of the components of an OpenStack deployment can communicate with one another quickly and efficiently.
- **Horizon** is the dashboard behind OpenStack. It is the only graphical interface to OpenStack, so for users wanting to give OpenStack a try, this may be the first component they actually \see." Developers can access all of the components of OpenStack individually through an application programming interface (API), but the dashboard provides system administrators a look at what is going on in the cloud, and to manage it as needed.
- **Keystone** provides identity services for OpenStack. It is essentially a central list of all of the users of the OpenStack cloud, mapped against all of the services provided by the cloud, which they have permission to use. It provides multiple means of access, meaning developers can easily map their existing user access methods against Keystone.
- **Glance** provides image services to OpenStack. In this case, "images" refers to images (or virtual copies) of hard disks. Glance allows these images to be used as templates when deploying new virtual machine instances.
- **Ceilometer** provides telemetry services, which allow the cloud to provide billing services to individual users of the cloud. It also keeps a verifiable count of each user's system usage of each of the various components of an OpenStack cloud. Think metering and usage reporting.
- **Heat** is the orchestration component of OpenStack, which allows developers to store the requirements of a cloud application in a file that defines what resources are necessary for that application. In this way, it helps to manage the infrastructure needed for a cloud service to run.

Figure 8 shows the main services and components of the OpenStack.

3.5 Architecture

Openstack can't be directly installed on hardware. It requires operating systems which supports virtualization in the back-end. At present , Ubuntu(kvm), Redhat enterprise Linux(kvm) , oracle Linux(xen) , Oracle Solaris(zones), Microfsoft Hyper-v, VMware ESXi supports openstack cloud platform. That's why openstack is the strategic choice of many types of organizations from service providers looking to offer cloud computing services on standard hardware, to companies looking to deploy private cloud, to large enterprises deploying a global cloud solution across multiple continents.

Figure 10 shows how the OpenStack components are interconnected.

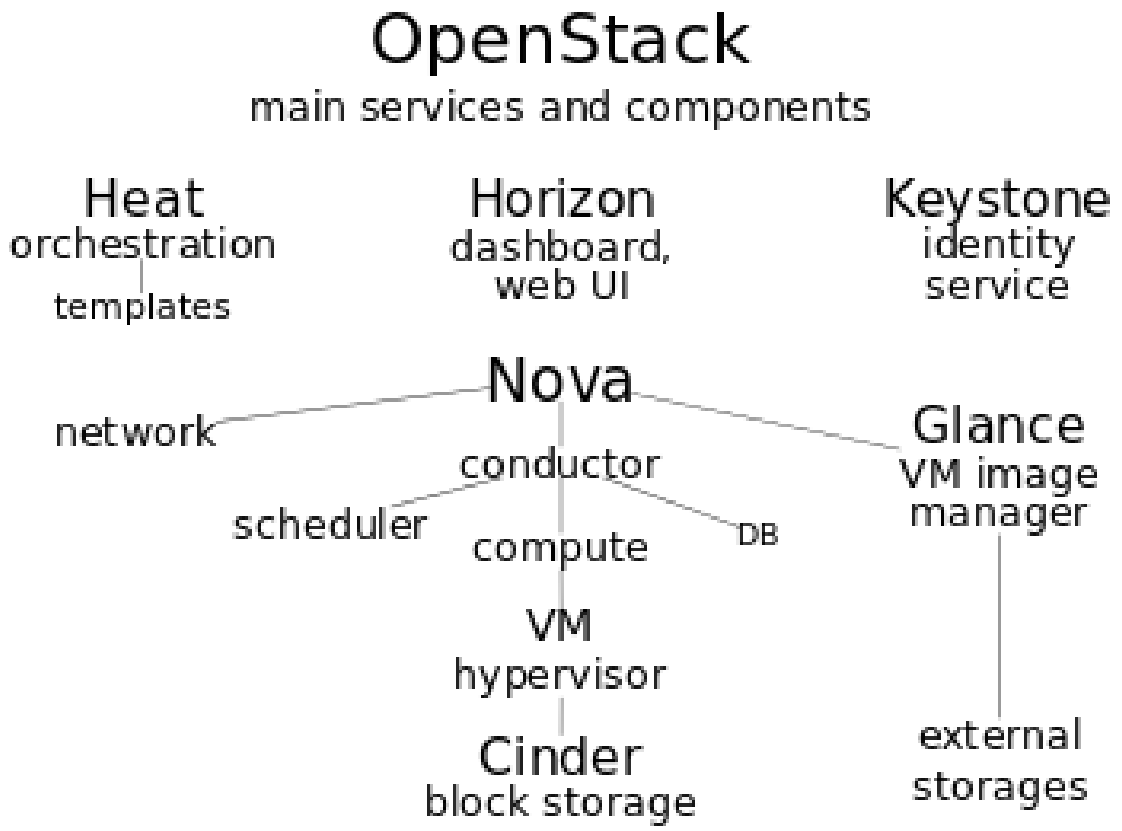

Figure 8: OpenStack main services

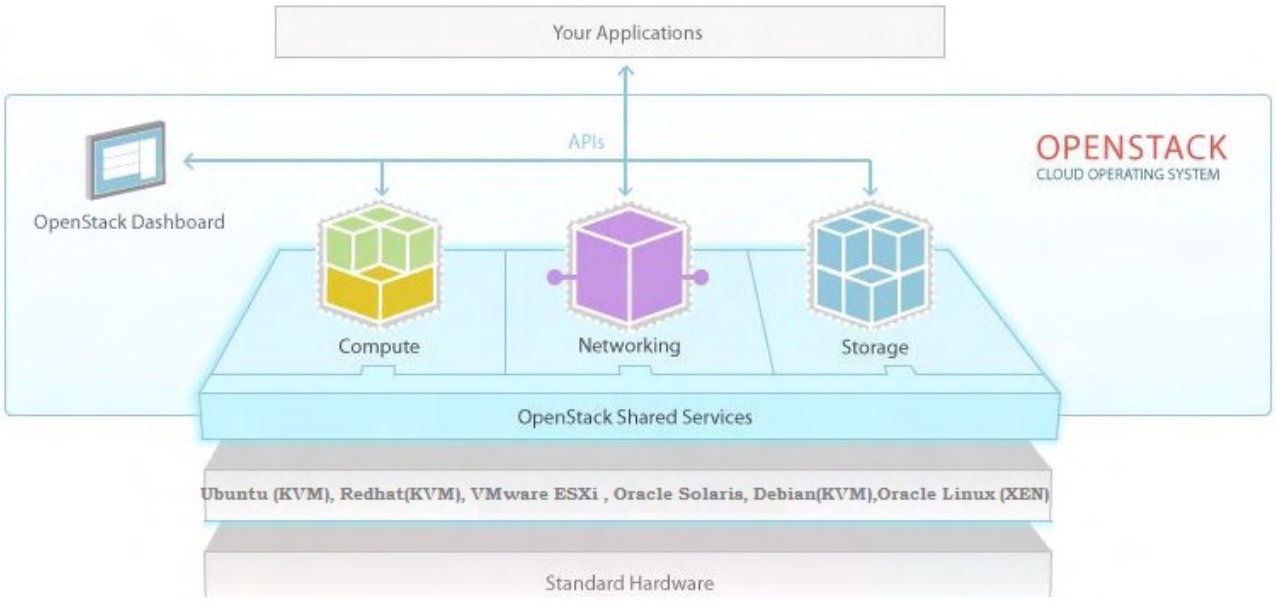

Figure 9: OpenStack Basic

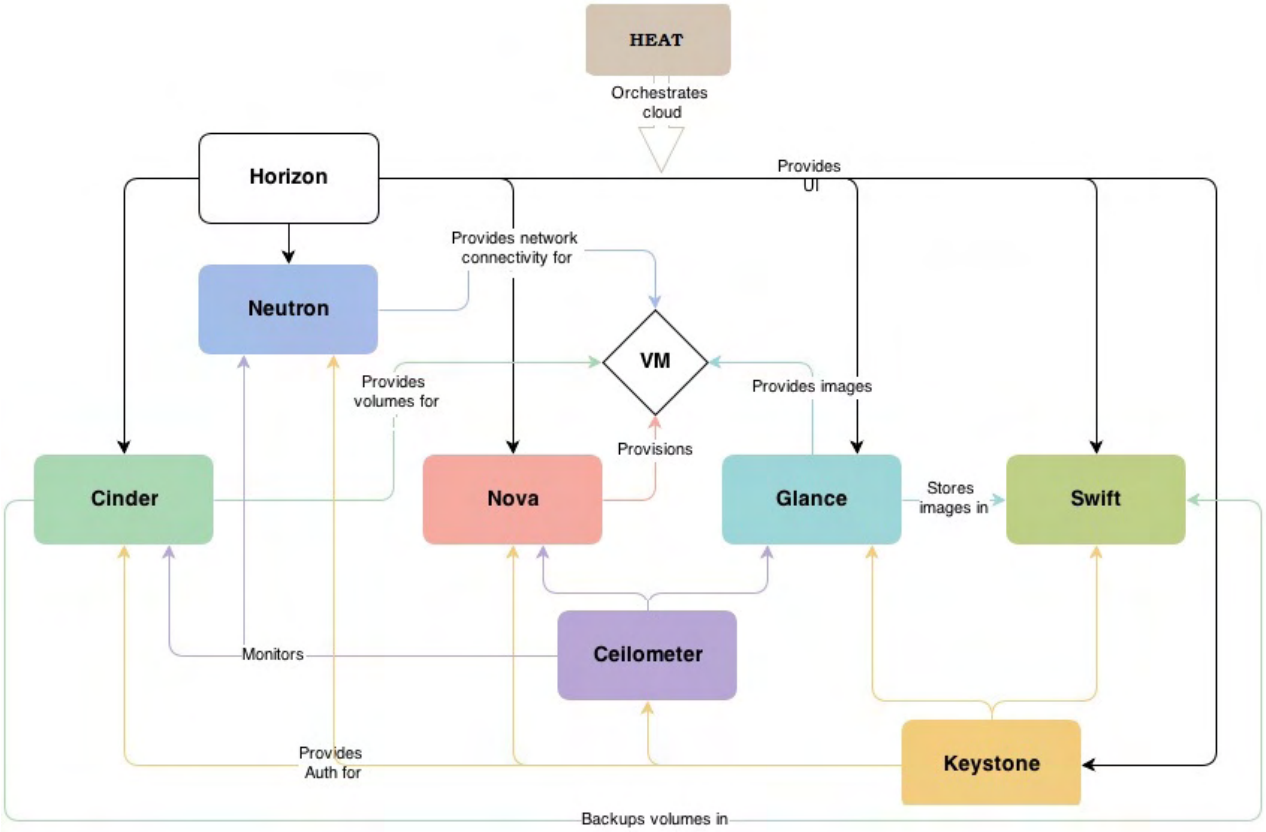

Figure 10: OpenStack Conceptual Architecture

3.6 Networking architecture

3.6.1 Networking components

OpenStack Networking is a standalone service that often deploys several processes across a number of nodes. These processes interact with each other and other OpenStack services. The main process of the OpenStack Networking service is neutron-server, a Python daemon that exposes the OpenStack Networking API and passes tenant requests to a suite of plug-ins for additional processing.

The OpenStack Networking components are:

neutron server (neutron-server and neutron-*-plugin) This service runs on the network node to service the Networking API and its extensions. It also enforces the network model and IP addressing of each port. The neutron-server requires indirect access to a persistent database. This is accomplished through plugins, which communicate with the database using AMQP (Advanced Message Queuing Protocol).

plugin agent (neutron-*-agent) Runs on each compute node to manage local virtual switch (vswitch) configuration. The plug-in that you use determine which agents run. This service requires message queue access and depends on the plugin used. Some plugins like OpenDaylight(ODL) and Open Virtual Network (OVN) do not require any python agents on compute nodes.

DHCP agent (neutron-dhcp-agent) Provides DHCP services to tenant networks. This agent is the same across all plug-ins and is responsible for maintaining DHCP configuration. The neutron-dhcp-agent requires message queue access. Optional depending on plug-in.

L3 agent (neutron-l3-agent) Provides L3/NAT forwarding for external network access of VMs on tenant networks. Requires message queue access. Optional depending on plug-in.

network provider services (SDN server/services) Provides additional networking services to tenant networks. These SDN services may interact with neutron-server, neutron-plugin, and plugin-agents through communication channels such as REST APIs.
The following figure(Figure 11) show an architectural and networking flow diagram of the OpenStack Networking components

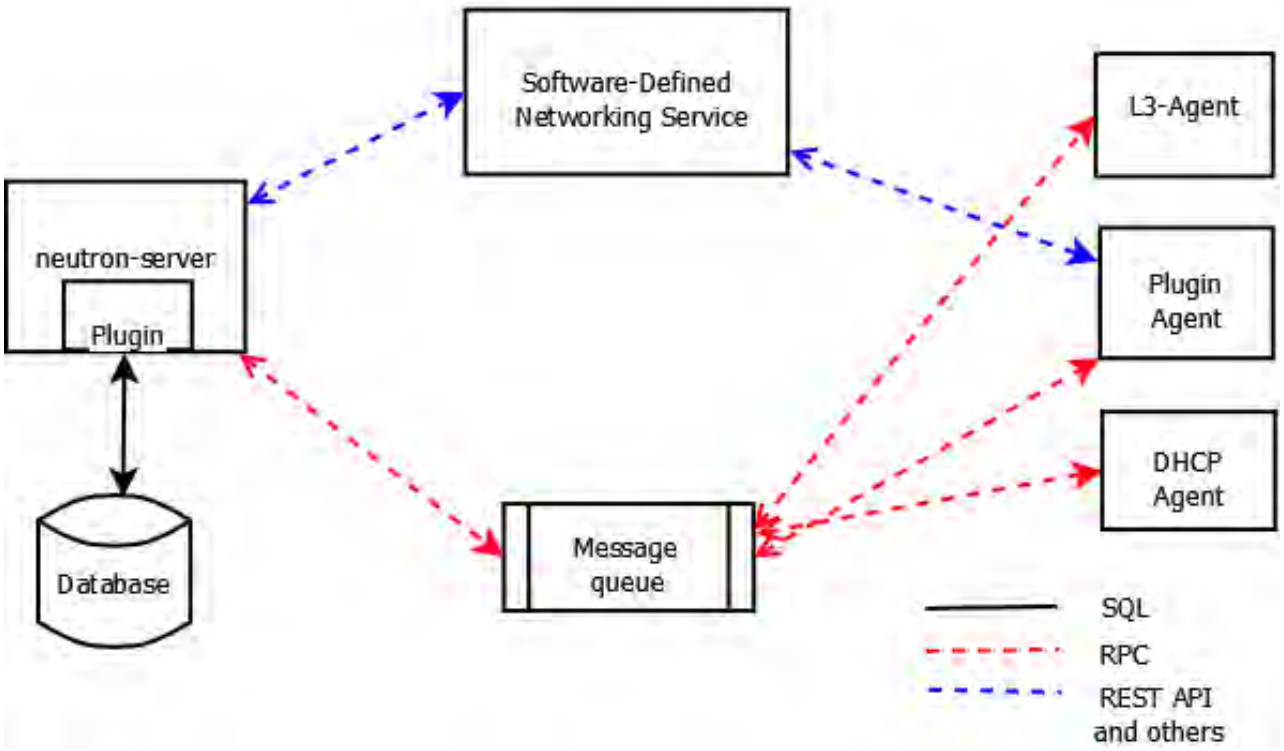

Figure 11: Networking flow diagram

3.6.2 Modular Layer 2 Core Plugin

The Modular Layer 2 (ML2) plugin is a framework allowing OpenStack Neutron to simultaneously utilize the variety of layer 2 networking technologies found in complex real-world data centers. ML2 achieves modularity through its driver model. As seen in the Figure 12, it includes two categories of drivers: type and mechanism. Type drivers (such as flat, VLAN, GRE and VXLAN) define a particular L2 type, where each available network type is managed by a corresponding type driver. The driver maintains type-specific state information and realizes the isolation among the tenant networks along with validation of provider networks.

On the other hand, the mechanism drivers | which are vendor specific (such as OVS, and drivers from ODL, Cisco, NEC, etc.), based on the enabled type driver | support creating, updating, and deletion of network, subnet and port resources. We should note that vendors may implement a completely new plug-in similar to ML2, or just implement a mechanism driver.

3.6.3 Network connectivity of physical servers

A standard OpenStack Networking setup has up to four distinct physical data center networks:

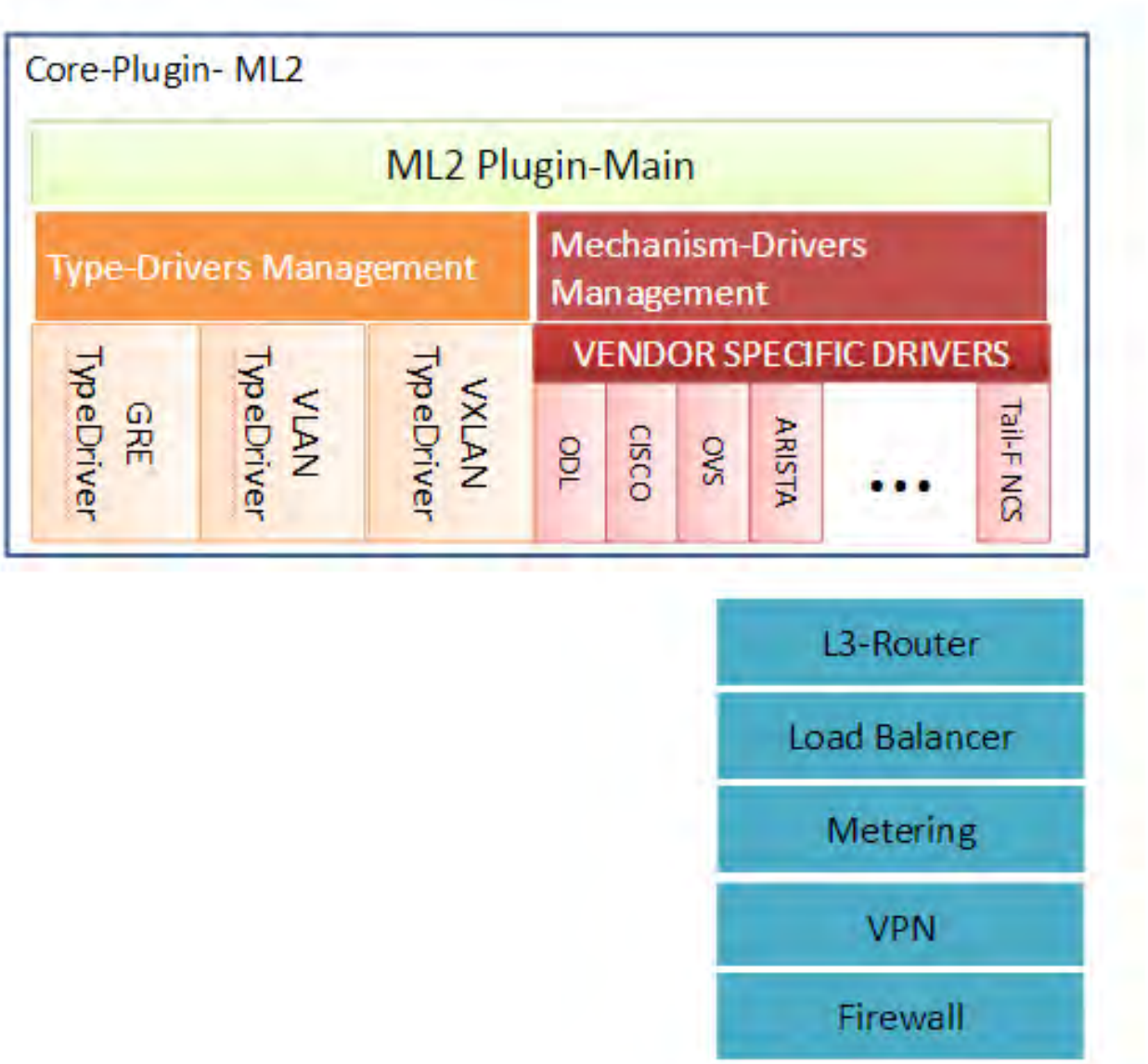

Figure 12: ML2 Plugin Architecture

Management network Used for internal communication between OpenStack Components. The IP addresses on this network should be reachable only within the data center and is considered the Management Security Domain.

Guest network Used for VM data communication within the cloud deployment. The IP addressing requirements of this network depend on the OpenStack Networking plug-in in use and the network configuration choices of the virtual networks made by the tenant. This network is considered the Guest Security Domain.

External network Used to provide VMs with Internet access in some deployment scenarios. The IP addresses on this network should be reachable by anyone on the Internet. This network is considered to be in the Public Security Domain.

API network Exposes all OpenStack APIs, including the OpenStack Networking API, to tenants. The IP addresses on this network should be reachable by anyone on the Internet. This may be the same network as the external network, as it is possible to create a subnet for the external network that uses IP allocation ranges to use only less than the full range of IP addresses in an IP block. This network is considered the

Public Security Domain.

3.7 Compute server architecture

When designing compute resource pools, consider the number of processors, amount of memory, network requirements, the quantity of storage required for each hypervisor, and any requirements for bare metal hosts provisioned through ironic.

When architecting an OpenStack cloud, as part of the planning process, you must not only determine what hardware to utilize but whether compute resources will be provided in a single pool or in multiple pools or availability zones. You should consider if the cloud will provide distinctly different profiles for compute.

For example, CPU, memory or local storage based compute nodes. For NFV or HPC based clouds, there may even be specific network configurations that should be reserved for those specific workloads on specific compute nodes. This method of designing specific resources into groups or zones of compute can be referred to as bin packing.

Increasing the size of the supporting compute environment increases the network traffic and messages, adding load to the controllers and administrative services used to support the OpenStack cloud or networking nodes. When considering hardware for controller nodes, whether using the monolithic controller design, where all of the controller services live on one or more physical hardware nodes, or in any of the newer shared nothing control plane models, adequate resources must be allocated and scaled to meet scale requirements. Effective monitoring of the environment will help with capacity decisions on scaling. Proper planning will help avoid bottlenecks and network oversubscription as the cloud scales.

Compute nodes automatically attach to OpenStack clouds, resulting in a horizontally scaling process when adding extra compute capacity to an OpenStack cloud. To further group compute nodes and place nodes into appropriate availability zones and host aggregates, additional work is required. It is necessary to plan rack capacity and network switches as scaling out compute hosts directly affects data center infrastructure resources as would any other infrastructure expansion.

While not as common in large enterprises, compute host components can also be upgraded to account for increases in demand, known as vertical scaling. Upgrading CPUs with more cores, or increasing the overall server memory, can add extra needed capacity depending on whether the running applications are more CPU intensive or memory intensive. We recommend a rolling upgrade of compute nodes for redundancy and availability. After the upgrade, when compute nodes return to the OpenStack cluster, they will be re-scanned and the new resources will be discovered adjusted in the OpenStack database.

When selecting a processor, compare features and performance characteristics. Some processors include features specific to virtualized compute hosts, such as hardware-assisted virtualization, and technology related to memory paging (also known as EPT shadowing). These types of features can have a significant impact on the performance of your virtual machine.

The number of processor cores and threads impacts the number of worker threads which can be run on a resource node. Design decisions must relate directly to the service being run on it, as well as provide a balanced infrastructure for all services.

Another option is to assess the average workloads and increase the number of instances that can run within the compute environment by adjusting the overcommit ratio. This ratio is configurable for CPU and memory. The default CPU overcommit ratio is 16:1, and the default memory overcommit ratio is 1.5:1. Determining the tuning of the overcommit ratios during the design phase is important as it has a direct impact on the hardware layout of your compute nodes.

Insufficient disk capacity could also have a negative effect on overall performance including CPU and memory usage. Depending on the back end architecture of the OpenStack Block Storage layer, capacity includes adding disk shelves to enterprise storage systems or installing additional Block Storage nodes. Upgrading directly attached storage installed in Compute hosts, and adding capacity to the shared storage for additional ephemeral storage to instances, may be necessary.

Consider the Compute requirements of non-hypervisor nodes (also referred to as resource nodes). This includes controller, Object Storage nodes, Block Storage nodes, and networking services.

The ability to create pools or availability zones for unpredictable workloads should be considered. In some cases, the demand for certain instance types or flavors may not justify individual hardware design. Allocate hardware designs that are capable of servicing the most common instance requests. Adding hardware to the overall architecture can be done later.

3.8 Deployment

3.8.1 Example Architecture

OpenStack provides an Infrastructure-as-a-Service (IaaS) solution through a variety of complementary services. Each service offers an Application Programming Interface (API) that facilitates this integration.

The example architecture requires at least two nodes (hosts) to launch a basic virtual machine or instance. Optional services such as Block Storage and Object Storage require additional nodes.

This example architecture differs from a minimal production architecture as follows:

- Networking agents reside on the controller node instead of one or more dedicated network nodes.
- Overlay (tunnel) traffic for self-service networks traverses the management network instead of a dedicated network.

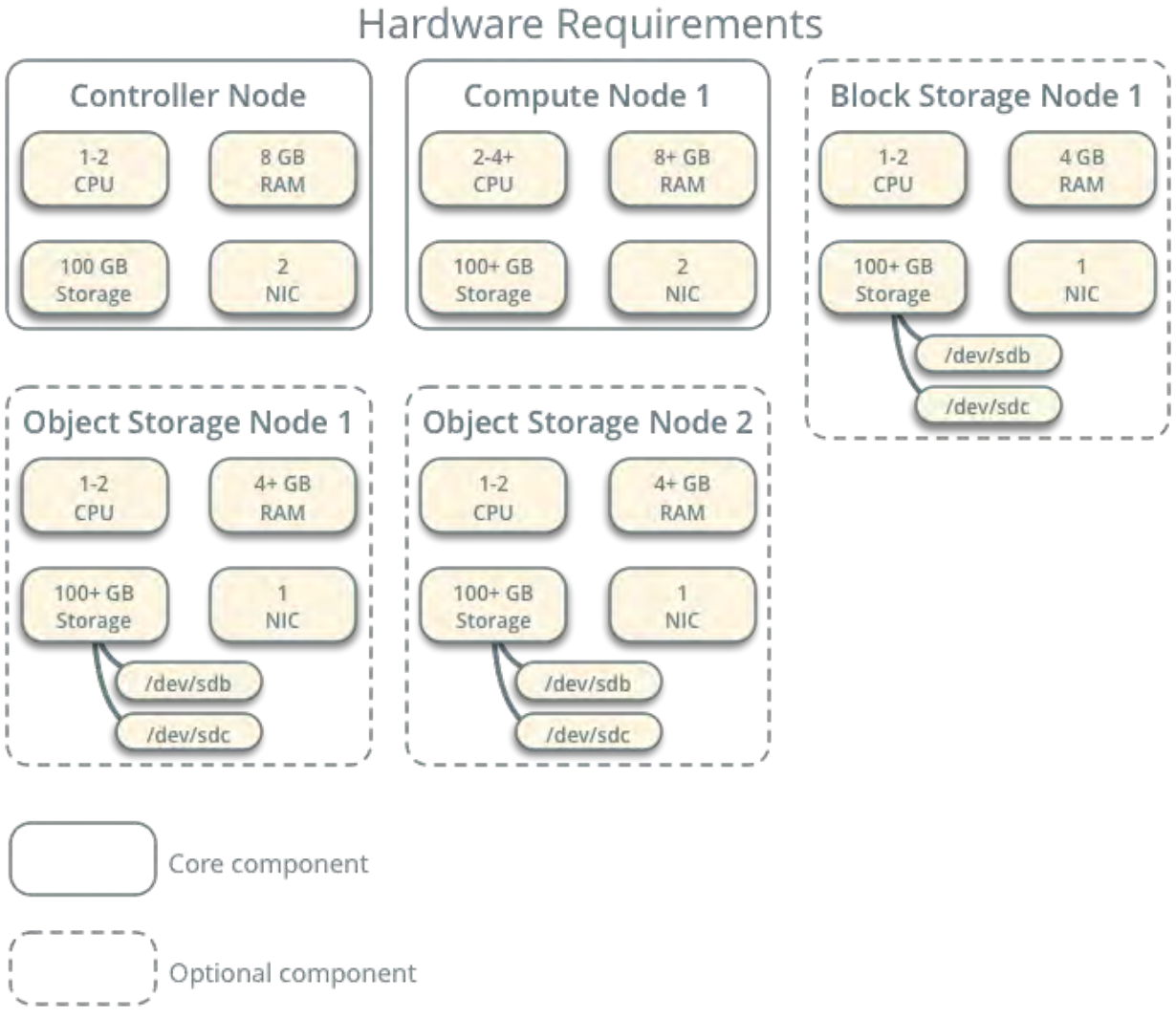

Figure 13: Hardware requirements

Controller The controller node runs the Identity service, Image service, management portions of Compute, management portion of Networking, various Networking agents, and the dashboard. It also includes supporting services such as an SQL database, message queue, and NTP.

Optionally, the controller node runs portions of the Block Storage, Object Storage, Orchestration, and Telemetry services.
The controller node requires a minimum of two network interfaces.

Compute The compute node runs the hypervisor portion of Compute that operates instances. By default, Compute uses the KVM hypervisor. The compute node also runs a Networking service agent that connects instances to virtual networks and provides firewalling services to instances via security groups.
You can deploy more than one compute node. Each node requires a minimum of two network interfaces.

Block Storage The optional Block Storage node contains the disks that the Block Storage and Shared File System services provision for instances.
For simplicity, service traffic between compute nodes and this node uses the management network. Production environments should implement a separate storage network to increase performance and security.
You can deploy more than one block storage node. Each node requires a minimum of one network interface.

Object Storage The optional Object Storage node contain the disks that the Object Storage service uses for storing accounts, containers, and objects.
For simplicity, service traffic between compute nodes and this node uses the management network. Production environments should implement a separate storage network to increase performance and security.
This service requires two nodes. Each node requires a minimum of one network interface. You can deploy more than two object storage nodes.

Networking Option 1: Provider networks The provider networks option deploys the OpenStack Networking service in the simplest way possible with primarily layer-2 (bridging/switching) services and VLAN segmentation of networks. Essentially, it bridges virtual networks to physical networks and relies on physical network infrastructure for layer-3 (routing) services. Additionally, a DHCP service provides IP address information to instances.

Networking Option 2: Self-service networks The self-service networks option augments the provider networks option with layer-3 (routing) services that enable self-service networks using overlay segmentation methods such as VXLAN. Essentially, it routes virtual networks to physical networks using NAT. Additionally, this option provides the foundation for advanced services such as LBaaS and FWaaS.

3.8.2 Three-Node Architecture

The architecture described here is deployed on the following three systems:

- Controller node. The Controller node is where most of the shared OpenStack services and other tools run. The Controller node supplies API, scheduling, and other shared services for the cloud. The Controller node has the dashboard, the image store, and the identity service. Additionally, Nova compute management service as well as the Neutron server are also configured in this node.
- Network node. The Network node provides virtual networking and networking services to Nova instances using the Neutron Layer 3 and DHCP network services.
- Compute node. The Compute node is where the VM instances (Nova compute instances) are installed. The VM instances use iSCSI targets provisioned by the Cinder volume service.

In this architecture, the three nodes share a common subnet called the management subnet. The Controller node and each compute node share a separate common subnet called the data subnet. Each system is attached to the management network through its net0 physical interface. The Network node and Compute node are attached to the data network through their net1 physical interfaces.
The following figure(Figure 16) shows a high-level view of the three node architecture.

4 OpenDaylight as OpenStack SDN controller

4.1 Integration of OpenStack and OpenDaylight

The overall process of OpenStack and OpenDaylight integration is summarized in Figure 17. On the OpenStack front, Neutron consists of the ML2 mechanism driver, which acts as a REST proxy and passes

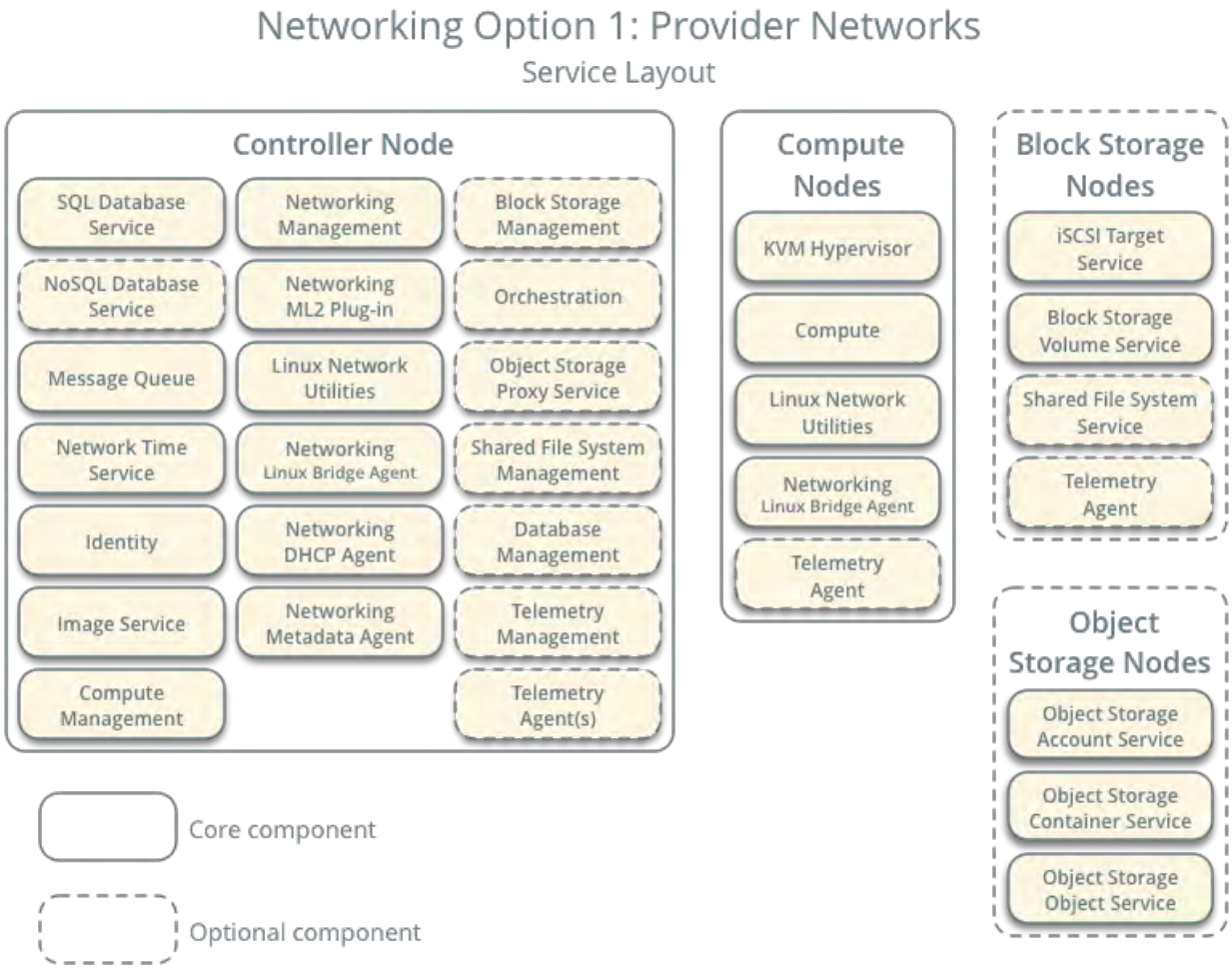

Figure 14: Provider networks

all Neutron API calls into OpenDaylight. OpenDaylight contains a northbound REST service (called Neutron API service) which caches data from these proxied API calls and makes it available to other services inside of OpenDaylight. Shown below, when we describe the two components in detail, these RESTful APIs achieve the binding of OpenStack and OpenDaylight.

4.2 OpenStack

The modular layer 2 (ML2) plugin for OpenStack Neutron is a framework designed to utilize the variety of layer 2 networking technologies simultaneously. The main idea behind the ML2 plugin is to separate the network type from the mechanism that realizes the network type. Drivers within the ML2 plugin implement extensible sets of network types (local, flat, VLAN, GRE and VXLAN), and mechanisms to access these networks.

In ML2, the registered mechanism drivers, which are typically vendor-specific, are called twice when the three core resources | networks, subnets and ports | are created, updated or deleted. The first call, typically referred to as a pre-commit call, is part of the DB transaction, where driver-specific states are maintained. In the case of the OpenDaylight mechanism driver, this pre-commit operation is not necessary. Once the transaction has been committed, the drivers are called again, typically referred as a post-commit call, at which point they can interact with external devices and controllers.

Mechanism drivers are also called as part of the port binding process, to determine whether the associated mechanism can provide connectivity for the network, and if so, the network segment and VIF driver to be used.

Figure 17 above summarizes OpenStack Neutron's ML2 OpenDaylight mechanism driver architecture. The OpenDaylight mechanism driver is made up of a single file \mechanism_odl.py" and a separate networking

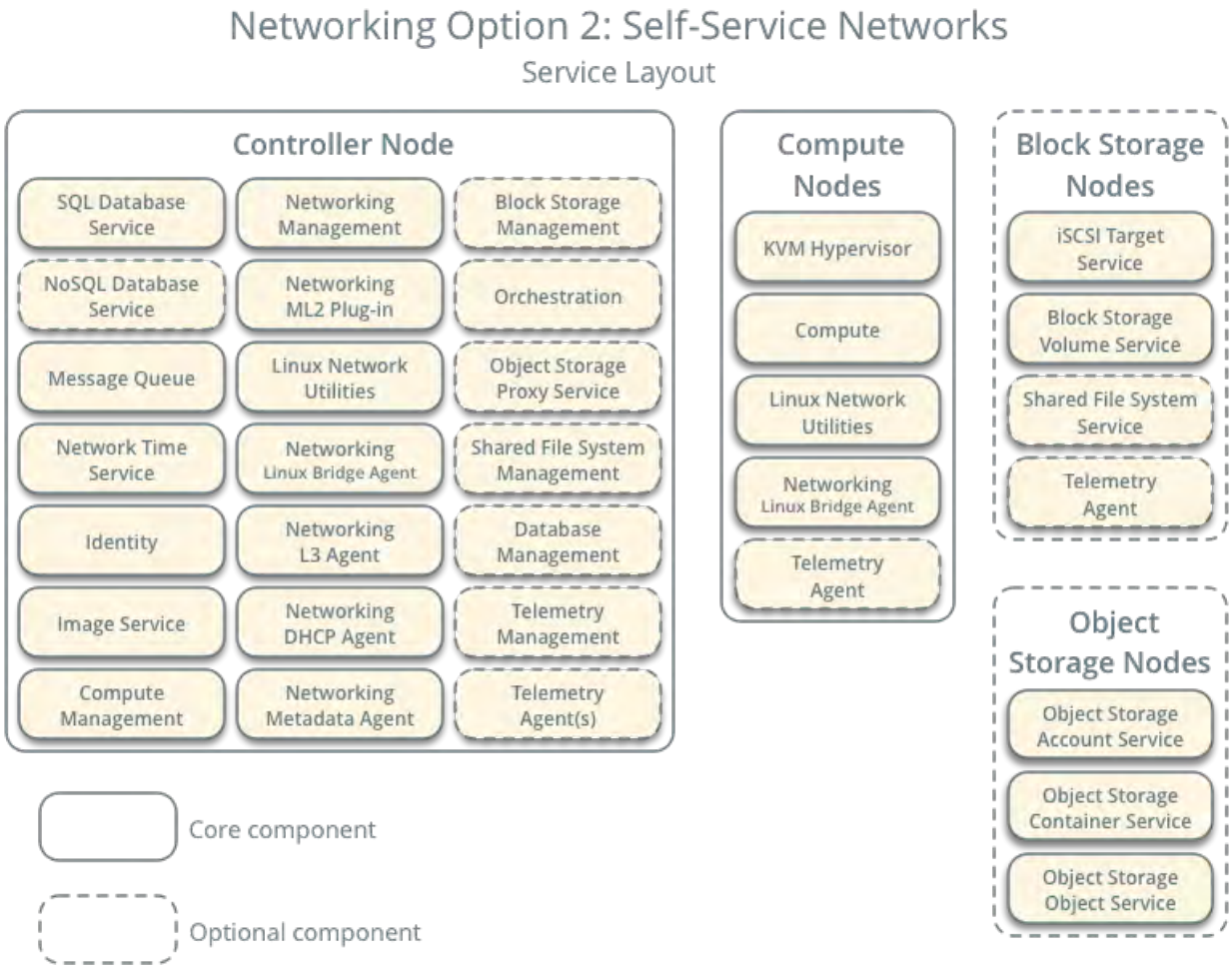

Figure 15: Self-service networks

OpenDaylight driver. The mechanism driver is divided into two different parts (core and extension), based on API handling. The OpenDaylight mechanism driver and OpenDaylight drive classes implement the core APIs. OpenDaylight's L3 router plugin class realizes the extension APIs only. Firewall as a service (FWaaS) and load balancing as a service (LBaaS) are currently not supported by the ODL driver.

The OpenDaylight mechanism driver receives the calls to create/update/delete the core resources (network, subnet and port). It forwards these calls to the OpenDaylight driver class by invoking the synchronize function. This function, in turn, invokes the 'sendjson' API.

Similarly, the OpenDaylight L3 router plugin class handles the L3 APIs to create/update/delete the router and floating IPs. Hence, the final call for both the core and L3 extension APIs is \sendjson" { which sends a REST request to the OpenDaylight controller and waits for the response.

4.3 OpenDaylight

OpenDaylight exposes the OpenStack Neutron API service { which provides Neutron API handling for multiple implementations. Figure 19 summarizes the architecture of Neutron API implementation in OpenDaylight. There are mainly three different bundles that constitute the Neutron API service { termed Northbound API, Neutron Southbound provider interface (SPI) and transcriber { and a collection of implementations. In this section, we will take a detailed look at these components.

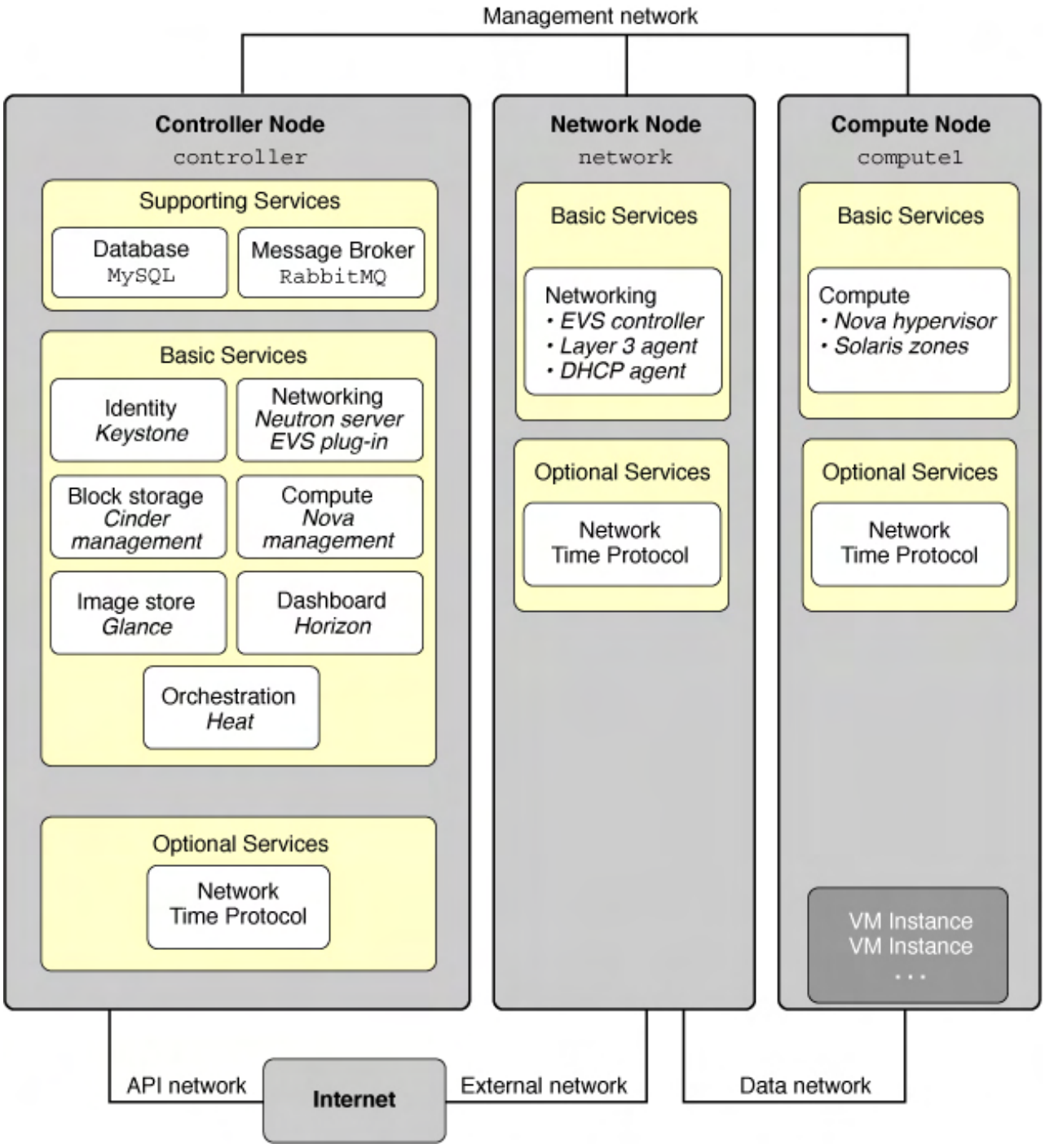

Figure 16: Three-Node Configuration Reference Architecture

4.4 Northbound API Bundle

This bundle handles the REST requests from the Openstack plugin and returns the appropriate responses. The contents of the Northbound API bundle can be described as follows:

1. A single parent class for requests: IneutronRequest.

2. A collection of JAXB (Java architecture for XML Binding) annotated request classes for each of the resources: network, subnet, port, firewall, load balancer, etc. These classes are used to represent a specific request, which implements the IneutronRequest interface. For example, the network request contains the following attributes: class NeutronNetworkRequest implements INeutronRequest<NeutronNetwork>

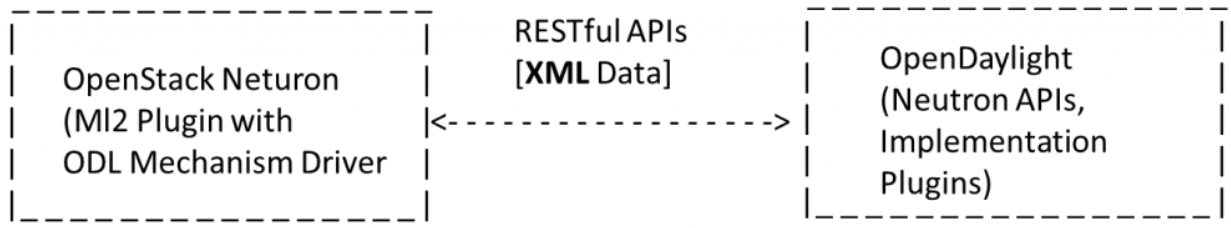

Figure 17: OpenStack and OpenDaylight integration

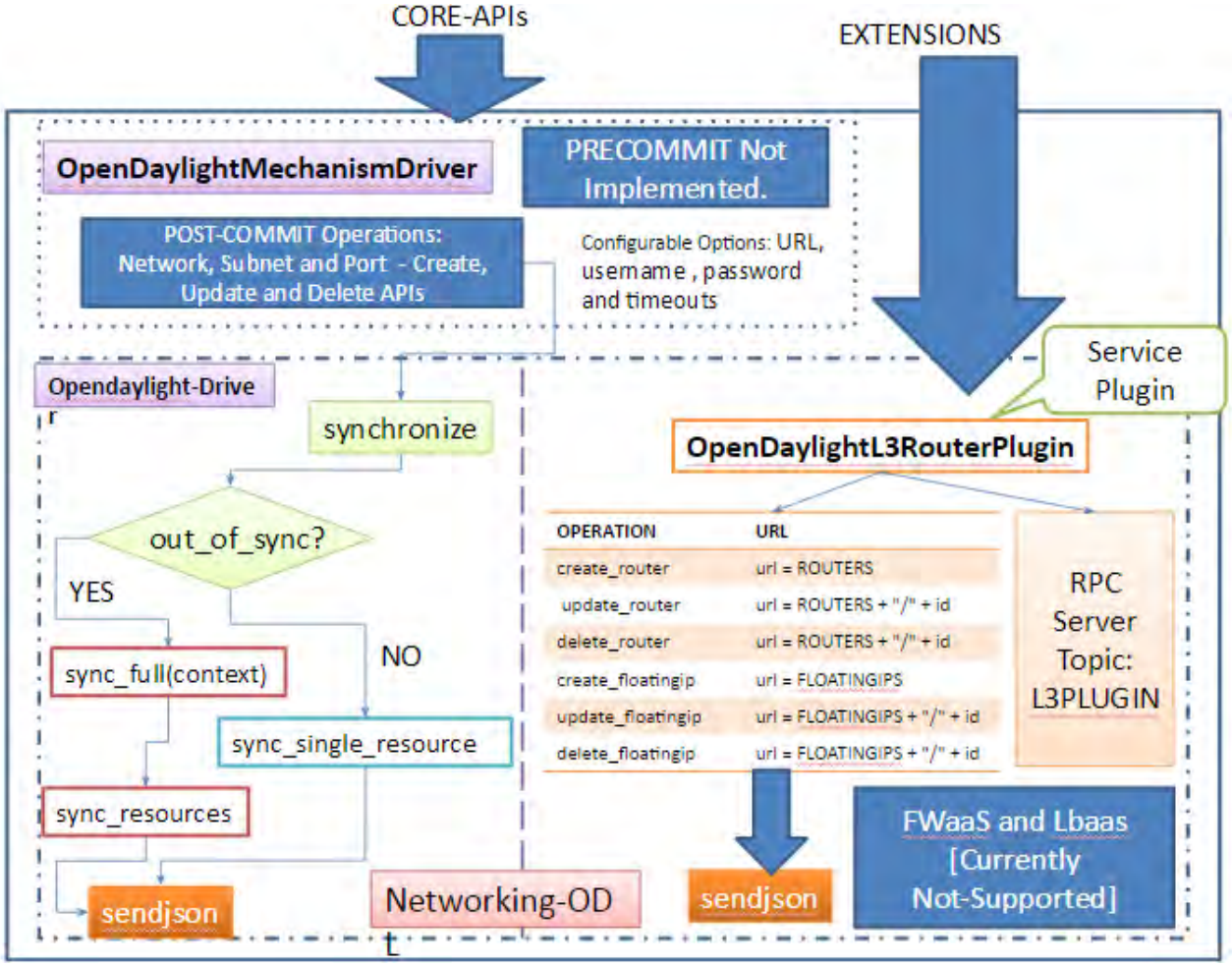

Figure 18: ML2 Mechanism Driver Architecture

3. A collection of Neutron northbound classes* which provide REST APIs for managing corresponding resources. For example, NeutronNetworksNorthbound class includes the following APIs: listNetworks(), showNetwork(), createNetworks(), updateNetwork() and deleteNetwork().

The symbol *, unless mentioned otherwise, represents any of the following: network, subnet, port, router, floating IP, security group, security group rules, load balancer, load balancer health, load balancer listener, load balancer pool, etc.

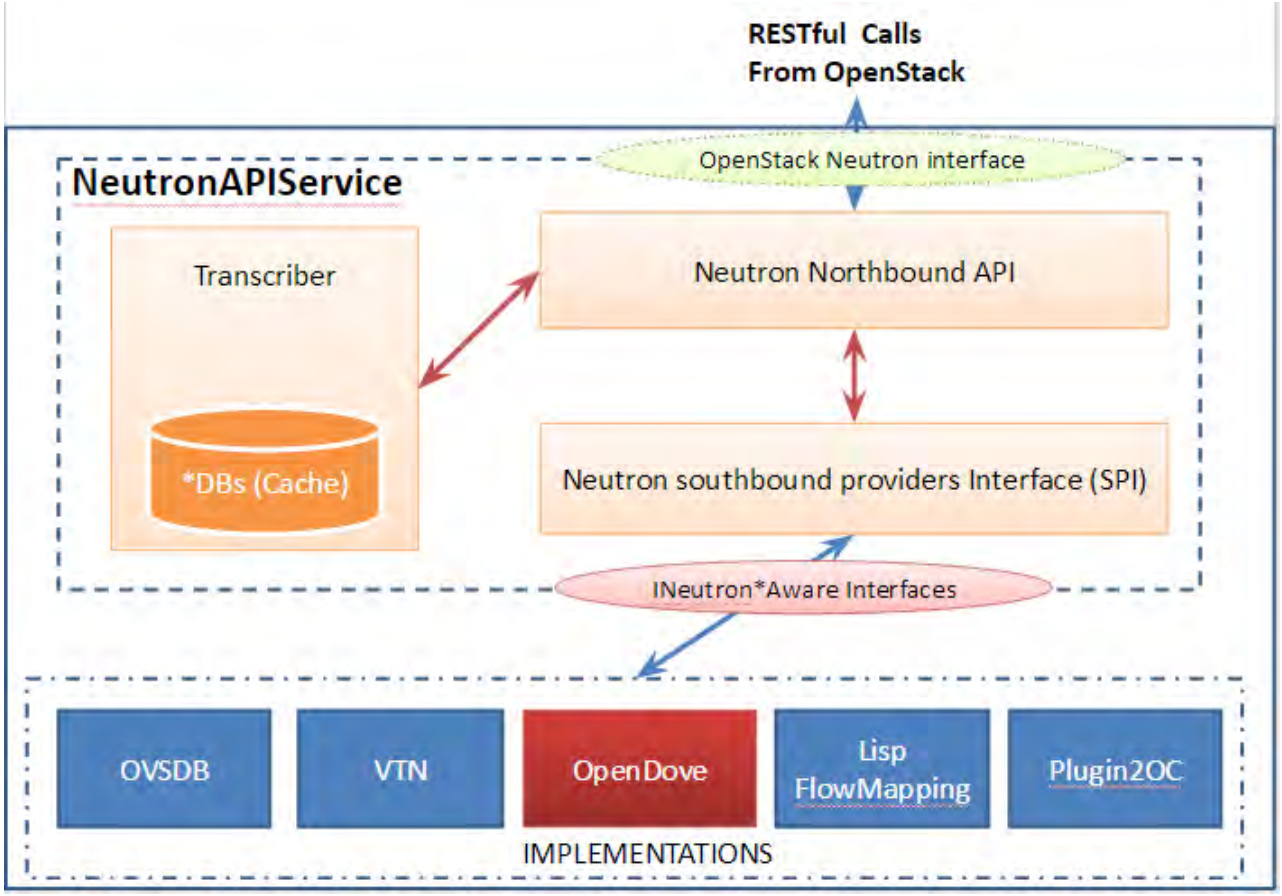

Figure 19: OpenDaylight Neutron API Implementation Architecture

4.5 Neutron SPI Bundle

This is the most important bundle that links the northbound APIs to the appropriate implementations. The Neutron southbound protocol interface (SPI) bundle includes the following:

1. JAXB (Java architecture for XML binding) annotated base class and subclasses, named Neutron* for supporting the API documented in networking API v2.0.
2. INeutron*CRUD interfaces, which are implemented by the transcriber bundle.
3. INeutron*Aware interfaces, which are implemented by the specific plugins (OpenDove, OVSDB, VTN, etc.).

The symbol *, unless mentioned otherwise, represents any of the following: Network, subnet, port, router, floating-IP, security-group, security-group rules, load-balancer, load-balancer health, load-balancer listener and load-balancer-pool etc.

4.6 Transcriber Bundle

The transcriber module consists of a collection of Neutron*Interface classes, which implement the INeutron*CRUD interfaces for storing Neutron objects in caches. Most of these classes include a concurrent HashMap. For example, private ConcurrentMap<String, NeutronPort> portDB = new ConcurrentHashMap<String, NeutronPort>() { and all the add, remove, and get operations work on this HashMap.

4.7 Implementation Bundle

The advantage of OpenDaylight is it includes multiple implementations of Neutron networks, providing several ways to integrate with OpenStack. The majority of the northbound services that aim to provide

network virtualization can be used as an implementation of the Neutron networks. Hence, OpenDaylight includes the following options for Neutron API implementations:

1. OVSDB: OpenDaylight has northbound APIs to interact with Neutron, and uses OVSDB for southbound configuration of vSwitches on compute nodes. Thus OpenDaylight can manage network connectivity and initiate GRE or VXLAN tunnels for compute nodes. OVSDB Integration is a bundle for OpenDaylight that will implement the Open vSwitch Database management protocol, allowing southbound configuration of vSwitches. It is a critical protocol for Network Virtualization with Open vSwitch forwarding elements. OVSDB neutron bundle in the virtualization edition supports network virtualization using VXLAN and GRE tunnel for OpenStack and CloudStack deployments

2. VTN Manager (Virtual Tenant Network): VTN manager, one of the network virtualization solutions in OpenDaylight, is implemented as an OSGi (Open Services Gateway initiative) bundle of controllers using AD-SAL, and manages OpenFlow switches. VTN Manager can also include a separate component that works as a network service provider for OpenStack. VTN Manager's Neutron component enables OpenStack to work in pure OpenFlow environments, in which all switches in the data plane support OpenFlow. VTN Manager can also make use of OVSDB-enhanced VTN. Neutron bundles can make use of OVSDB plugins for operations such as port creation.

3. Open DOVE: Open DOVE is a \network virtualization" platform with a full control plane implementation for OpenDaylight and data plane based on \Open vSwitch." It aims to provide logically isolated multitenant networks with layer-2 or layer-3 connectivity, and runs on any IP network in a virtualized data center. Open DOVE is based on IBM SDN virtual environments and DOVE technology from IBM Research. Open DOVE has not been updated after the Hydrogen release, and its existence in the Lithium release of OpenDaylight is doubtful.

4. OpenContrail (plugin2oc): provides the integration/interworking between the OpenDaylight controller and the OpenContrail platform. This combined open source solution will seamlessly enable OpenContrail platform capabilities such as cloud networking and NFV within the OpenDaylight project.

5. LISP Flow Mapping: Locator/ID Separation Protocol (LISP) aims to provide a \flexible map-and-encap framework that can be used for overlay network applications, and decouples network control plane from the forwarding plane." LISP includes two namespaces: endpoint identifiers (EIDs | IP address of the host), and routing locators (RLOCs |IP address of the LISP router to the host). LISP flow mapping provides LISP mapping system services, which store and serve the mapping data (including a variety of routing policies such as traffic engineering and load balancing) to data plane nodes, as well as to OpenDaylight applications.

These implementations typically realize some or all of the following handlers: network, subnet, port, router, floating-IP, firewall, firewall policy, firewall rule, security group, security group rules, load balancer, load balancer health, load balancer listener, load balancer pool and load balancer pool member. These handlers support create, delete and update operations for the corresponding resource.

The exact mechanism involved in these handlers depends on the southbound plugin they use: OpenFlow (1.0 or 1.3), OVSDB, LISP, REST (OpenContrail), etc. Let us use the example of a NeutronNetworkCreated handler in VTN Manager. The steps involved in this handler can be summarized as:

1. Check if the network can be created (again) by calling canCreateNetwork.

2. Convert Neutron network's tenant ID and network ID to tenant ID and bridge ID, respectively.

3. Check if a tenant already exists, and if not, create a tenant.

4. Create a bridge and perform VLAN mapping.

For the actual operations, the Neutron component of VTN manager invokes VTN manager's core function, which in turn uses the OpenFlow (1.0) plugin to make necessary configurations on the OpenFlow switches.

4.8 Using All Bundles for Network Creation

Figure 20 above briefly summarizes the process involved in network creation, and the corresponding calls in all of the above described bundles of the Opendaylight Neutron implementation. This figure should help the reader understand the control flow across all the bundles.

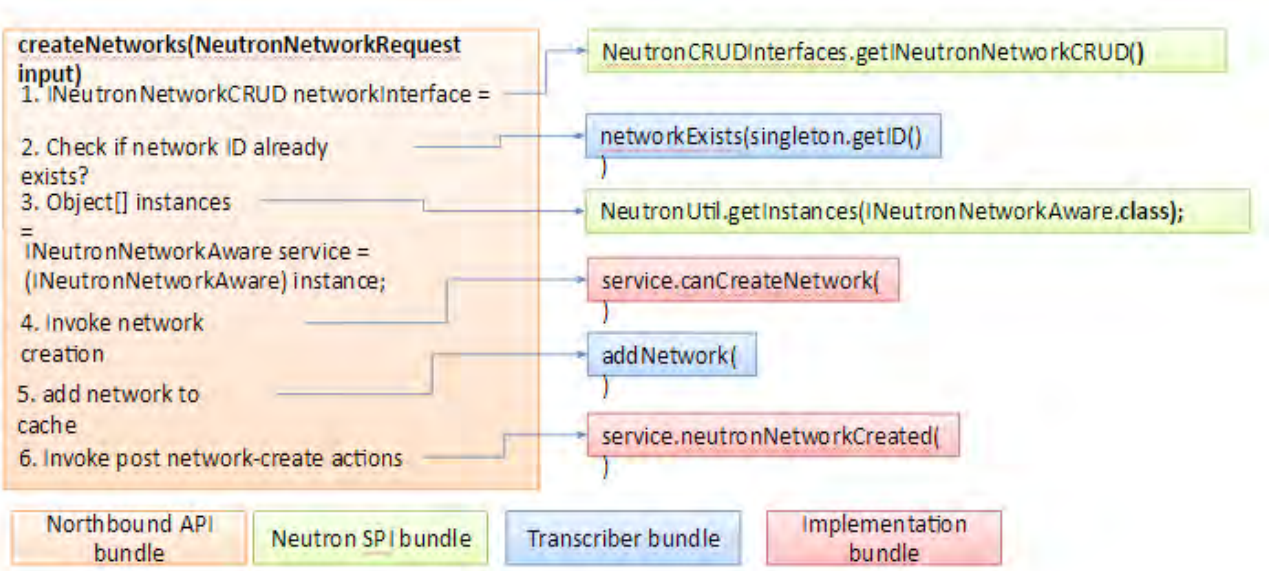

Figure 20: Process for Network Creation in OpenDaylight

In summary, OpenDaylight is one of the best open source controllers for providing OpenStack integration. Though the support for load balancer and firewall services is still missing, the freedom of multiple implementations and support of complete core APIs itself provides immense advantage and flexibility to the administrator. In the near future, we can expect OpenDaylight to support all the extensions of OpenStack to achieve the perfect integration.

4.9 OpenStack with VTN

This section describes using OpenDaylight with the VTN manager feature providing network service for OpenStack. VTN manager utilizes the OVSDB southbound service and Neutron for this implementation. The below diagram depicts the communication of OpenDaylight and two virtual networks connected by an OpenFlow switch using this implementation.

odl-vtn-manager-neutron feature provides the integration with Neutron interface.

Functional Behavior

- The ML2 implementation for OpenDaylight will ensure that when Open vSwitch is started, the ODL_IP_ADDRESS configured will be set as manager.
- When OpenDaylight receives the update of the Open vSwitch on port 6640 (manager port), VTN Manager handles the event and adds a bridge with required port mappings to the Open vSwitch at the OpenStack node.
- When Neutron starts up, a new network create is POSTed to OpenDaylight, for which VTN Manager creates a Virtual Tenant Network.
- Network and Sub-Network Create: Whenever a new sub network is created, VTN Manager will handle the same and create a vbridge under the VTN.
- VM Creation in OpenStack: The interface mentioned as integration bridge in the configuration file will be added with more interfaces on creation of a new VM in OpenStack and the network is provisioned for it by the VTN Neutron feature. The addition of a new port is captured by the VTN Manager and it creates a vbridge interface with port mapping for the particular port. When the VM starts to communicate with other VMs, the VTN Manger will install flows in the Open vSwitch and other OpenFlow switches to facilitate communication between them.

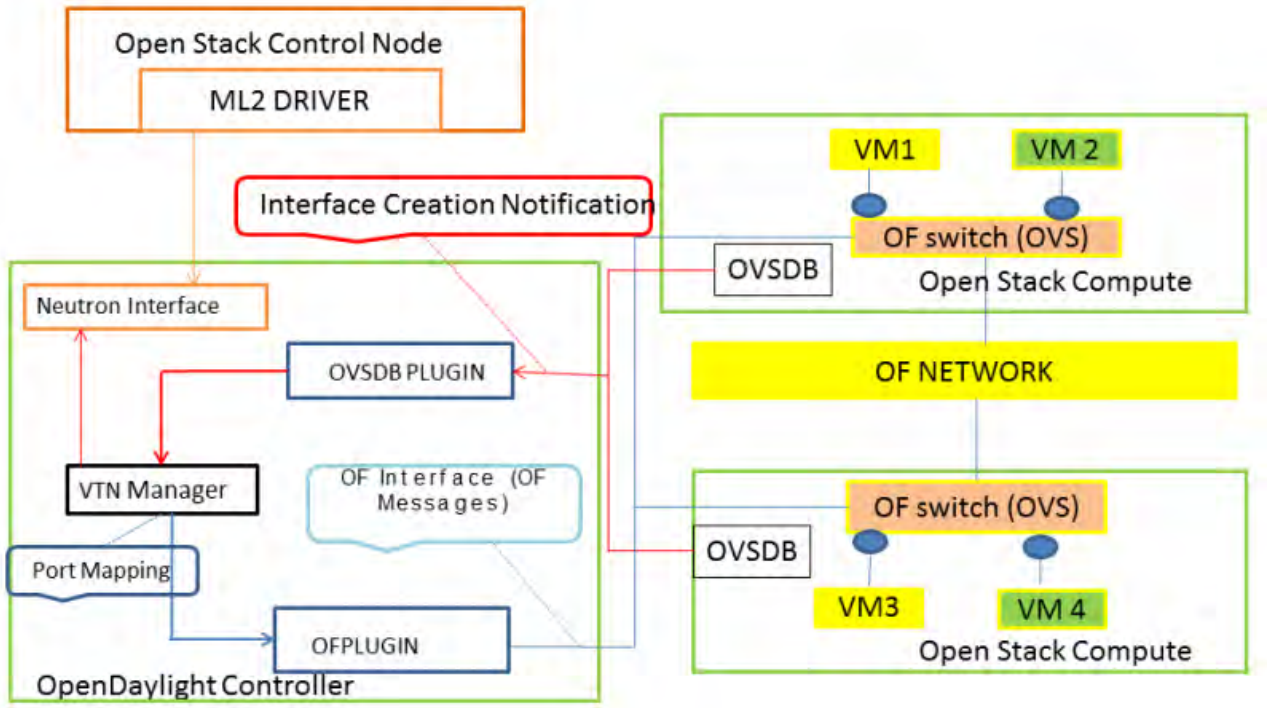

Figure 21: OpenStack Architecture

5 SDN assist of Openstack

5.1 Introduction

Open Source MANO (OSM) is focused on delivering an open source Management and Orchestration (MANO) stack for NFV capable of meeting the requirements of production NFV networks while consuming an established common Information Model (IM) { all defined, implemented and released in open source software.

The way SDN assist works, succinctly, is as follows. OSM will deploy the VMs of a NS with Passthrough and/or SRIOV interfaces, then will get from the VIM (Openstack) the compute node where the VM was deployed and the physical interface assigned to the VM (identified by its PCI address). Then, OSM will map those interfaces to Openflow ports in the switch making use of the mapping that you should have introduced in the system, and finally OSM will create the dataplane networks by talking to the SDN controller and connecting the appropriate Openflow ports to the same network.

The module in charge of this is the OSM-RO (Resource Orchestrator). It uses an internal library to manage the underlay connectivity via SDN. The library relies on Openflow pro-active rules to configure the connectivity in the switch. The current library includes plugins for FloodLight, ONOS and OpenDayLight.

5.2 General requirements

The general requirements are:

- A dataplane switch with Openflow capabilities that will connect the physical interfaces of the VIM compute nodes.
- An external SDN controller controlling the previous dataplane switch.

- The mapping between the switch ports (identified by name) and the compute node interfaces (identified by host-id and PCI address)
- Some VIMs as Openstack requires admin credentials in order to be able to get the physical place of the SRIOV/passthrough VM interfaces

In addition to the general requirements, every VIM will have to be properly configured.

5.3 Using SDN assist

NFV MANO enables flexible on-boarding, design, deployment, operation and maintenance of NFV services. NFV MANO comprises three functional blocks:

- **NFV Orchestrator (NFVO):**
  - { On-boarding of new Network Service (NS) and VNF Packages
  - { NS lifecycle management
  - { Resource management
  - { Policy management for NS instances
- **VNF Manager (VNFM):**
  - { Lifecycle management of VNF instances
  - { Coordination of configuration and event reporting between NFVI and the E/NMS
- **Virtualized Infrastructure Manager (VIM):**
  - { Controlling and managing the NFVI compute, storage and network resources
  - { Collection and forwarding of performance measurements and events

6 Software Development and Deployment in the NITOS testbed

6.1 Introduction

We wanted to experiment with the communication between the control node and the compute node of an OpenStack deployment over a random/unknown network infrastructure. We simulated the infrastructure network with the use of Mininet. So, for this communication to work the infrastructure network had to be managed and configured based on the OpenStack needs. For this reason, we developed a software that uses the VTN feature of ODL to manage and configure the infrastructure network. The software architecture is pretty simple as shown in the Figure 22, using RESTful commands it takes the information needed from ODL-OS and then based on this information it creates a VTN in ODL-NET. ODL-OS is the OpenDaylight controller which manages the OpenStack and ODL-NET the OpenDaylight controller which manages the infrastructure network (mininet network).

The demo was deployed in the NITOS testbed, using two nodes. One node was the controller node(node084) and the other one was the compute node(node085) of the OpenStack. Each node in the NITOS testbed has two interfaces eth0 and eth1. We used eth0 interface to deploy our OpenStack. The eth1 interface was used as internal data network for tenant traffic.

The networks that have been used in the demo are vlan networks.

6.2 Devstack

DevStack is a series of extensible scripts used to quickly bring up a complete OpenStack environment based on the latest versions of everything from git master. It is used interactively as a development environment and as the basis for much of the OpenStack project's functional testing. It is used interactively as a development environment. Since DevStack installs all-in-one OpenStack environment, it can be used to deploy OpenStack on a single VM, a physical server or a single LXC container. Each option is suitable depending on the hardware capacity available and the degree of isolation required. A multi-node OpenStack environment can also be deployed using DevStack.

Figure 22: Software Architecture

For either kind of setup, the steps would involve installing a minimal version of one of the supported Linux distributions and downloading the DevStack git repository. The repo contains a script stack.sh that must be run as a non-root user and will perform the complete install based on configuration settings.

The official approved and tested Linux distributions are Ubuntu(LTS plus current dev release), Fedora(latest and previous release) and CentOS/RHEL 7(latest major release). The supported databases are MySQL and PostgreSQL. RabbitMQ and Qpid are the recommended messaging service along with Apache as the webserver. The setup defaults to a FlatDHCP network using Nova Network or a similar configuration in Neutron.

The default services configured by DevStack are Keystone, Swift, Glance, Cinder, Nova, Neutron, Horizon and Heat. DevStack has a plugin architecture to include additional services that are not included directly in the install.

DevStack is not and has never been intended to be a general OpenStack installer. It has evolved to support a large number of configuration options and alternative platforms and support services. However, that evolution has grown well beyond what was originally intended and unfortunately many of the configuration combinations are rarely, if ever, tested.

DevStack configuration is modified via the file local.conf. It is a modified INI format file that introduces a meta-section header to carry additional information regarding the configuration files to be changed. The file is processed strictly in sequence; meta-sections may be specified more than once but if any settings are duplicated the last to appear in the file will be used.

The new header is similar to [[‘ <phase> ‘|’ <config{file{name> ‘]]’, where <phase> is one of a set of phase names defined by stack.sh and <config-file-name> is the configuration filename. If the path of the config file does not exist, it is skipped. The file is processed strictly in sequence and any repeated settings will override previous values.

The defined phases are:

- local - extracts localrc from local.conf before stackrc is sourced
- post-config - runs after the layer 2 services are configured and before they are started
- extra { runs after services are started and before any files in extra.d are executed
- post-extra { runs after files in extra.d are executed

A specific meta-section local|localrc is used to provide a default localrc file. This allows all custom settings for DevStack to be contained in a single file. If localrc exists it will be used instead to preserve backward-compatibility.

The local.conf files that used in this demo are listed below.
For control node:

```
[[local|localrc]]
VERBOSE=True
LOG_COLOR=True

ADMIN_PASSWORD=secret
DATABASE_PASSWORD=$ADMIN_PASSWORD
RABBIT_PASSWORD=$ADMIN_PASSWORD
SERVICE_PASSWORD=$ADMIN_PASSWORD

# disable everything so we can explicitly enable only what we need
disable_all_services

# here
enable_service placement-api

# Core compute (glance+keystone+nova+vnc)
enable_service g-api g-reg key n-api n-crt n-obj n-cpu n-cond n-sch n-novnc n-xvnc n-cauth

# dashboard
enable_service horizon

# neutron services. Recognize neutron-agt and neutron-l3 is not set which means ODL is the
# l2 agent and l3 provider.
enable_service neutron neutron-dhcp neutron-meta neutron-svc

# enable one of the two below:
# the first is external which assumes the user has ODL running already
# make sure to set the ODL_MGR_IP and ODL_PORT because we run in manual mode
# the second is allinone where devstack will download (if online) and start ODL
enable_service odl-compute odl-neutron
# or use the allinone
#enable_service odl-server odl-compute

# additional services. rabbit for rpm-based vm.
enable_service mysql rabbit

HOST_IP=$(ip route get 8.8.8.8 | awk '{print $NF; exit}')
HOST_NAME=$(hostname)
SERVICE_HOST_NAME=$HOST_NAME
SERVICE_HOST=$HOST_IP
Q_HOST=$SERVICE_HOST

enable_plugin networking-odl http://git.openstack.org/openstack/networking-odl stable/queens
ODL_MODE=manual
ODL_PORT=8080
ODL_MGR_IP=$HOST_IP
ODL_PROVIDER_MAPPINGS=public:eth1
ODL_INSTALL=False

NEUTRON_CREATE_INITIAL_NETWORKS=False
SKIP_OVS_INSTALL=True
```

For compute node:

```
[[local|localrc]]
VERBOSE=True
LOG_COLOR=True
```

```
PIP_UPGRADE=True

ADMIN_PASSWORD=secret
DATABASE_PASSWORD=$ADMIN_PASSWORD
RABBIT_PASSWORD=$ADMIN_PASSWORD
SERVICE_PASSWORD=$ADMIN_PASSWORD

# disable everything so we can explicitly enable only what we need
disable_all_services

# here
enable_service placement-api placement-client

# Core compute (nova+vnc)
enable_service n-cpu n-novnc n-api-meta

# next line enables odl as the neutron backend rather than the l2 agent
enable_service neutron odl-compute

# additional services. rabbit for rpm-based vm.
enable_service rabbit

HOST_IP=$(ip route get 8.8.8.8 | awk '{print $NF; exit}')
HOST_NAME=$(hostname)
SERVICE_HOST_NAME=control
SERVICE_HOST=10.0.1.84
Q_HOST=$SERVICE_HOST

enable_plugin networking-odl http://git.openstack.org/openstack/networking-odl stable/queens

ODL_MODE=compute
ODL_PORT=8080
ODL_MGR_IP=$SERVICE_HOST

# public network connectivity
ODL_PROVIDER_MAPPINGS=public:veth2
SKIP_OVS_INSTALL=True
```

Both local.conf files are self explanatory, the only thing worth to mention is the option ODL_PROVIDER_MAPPINGS=<phys_net>:<interface>, which tells the OpenStack how to connect to the pre-existing network. The provider physical_network name we provided when creating the network is mapped to an actual interface on the compute node via OVSDB other_config:provider_mappings. This port may have a different name across compute nodes, but it is assumed all compute nodes are connected to this physical network. We can verify it using the following command:

```
stack@control:~/devstack$ sudo ovs-vsctl get Open_vSwitch . other_config:provider_mappings
"public:eth1"
```

In the local.conf for control node we configured other_config:provider_mappings=public:eth1, this causes eth1 port to be added to the br-int bridge. We will talk more about br-int bridge later in this chapter.

6.3 Software tools

The following software tools have been used for this demo.

- Devstack for the deployment of OpenStack
- OpenDaylight as SDN controller
- Mininet to simulate a infrastructure network
- Python code to implement the software to achieve connection between the control and compute nodes

6.4 Topology

The Figure 23 below shows the topology that has been used for the demo. It is a basic OpenStack deployment with one control and one compute node. The connection between them is achieved via the mininet network. There are 2 ODL instances one for the OpenStack (ODL-OS) and one for the mininet network (ODL-NET). ODL-NET is responsible for the connection between control and compute node.

The mininet network contains two switches connected in linear way, the S3 switch, which has eth1 port and S4 switch, which has veth1. We used veth pair ports because of a lack of a third interface (eth2). So, when the VM1 tries to ping VM2 the packets will go through eth1 interface and respectively for VM2 the packets will go through veth2 interface.

The goal here is to somehow the minine network know how to forward the packets receive, either from port eth1 or veth1. In this simple case all we have to do is just to add a couple of flows in S3 and S4 but we want to achieve a more generic approach to this problem. In real case scenarios the infrastructure network may contain more than two switches and the topology may be more complicated.

The solution we propose is simple, using the VTN feature of OpenDaylight we can hide the complexity of the infrastructure network and then all we have to do is to map our physical resources to the virtual resources that we have created using the VTN feature, more details will be discussed later on this chapter.

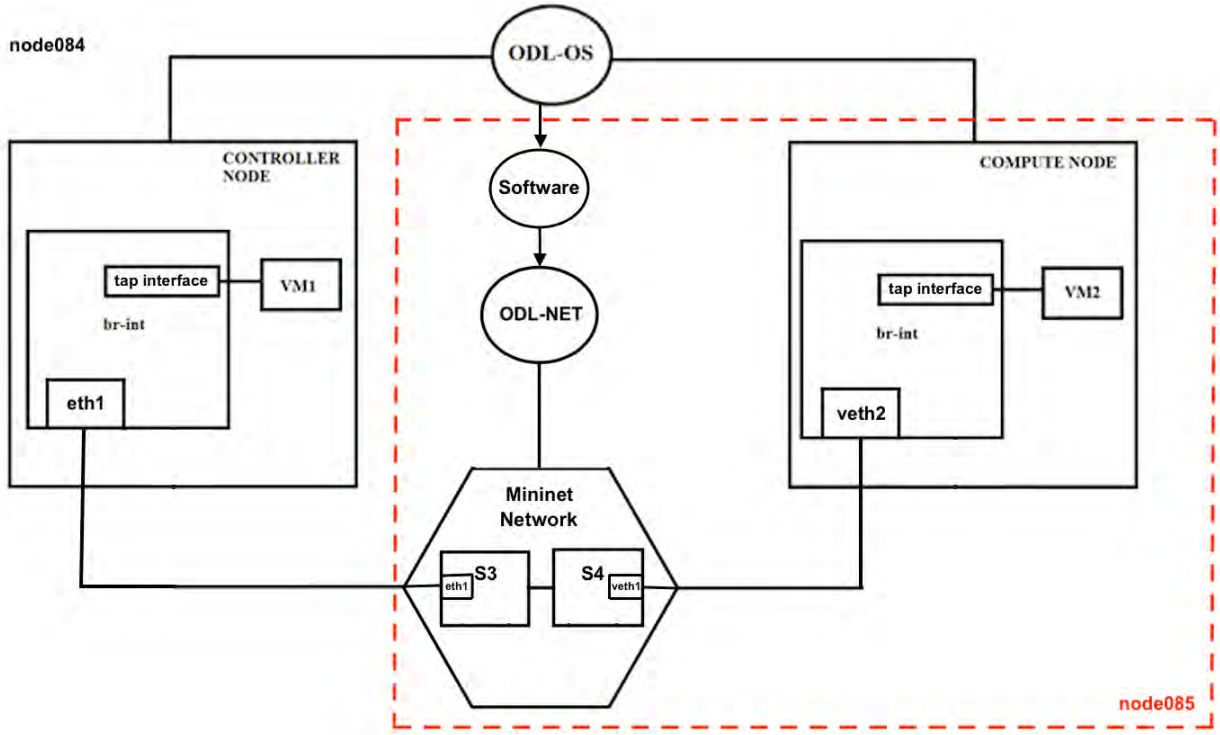

Figure 23: Demo Topology

6.5 Demo

6.5.1 Overview

The demo is as follows, first we create a vlan network with vlan id 10, then on this network we deploy two Virtual Machines (VMs). One VM is located on the control node and the other on the compute node. After the successful creation of the two VMs we try to see if there is a connection between them, if the one can ping the other. The ping is unsuccessful, because the packets dropped when they arrive at the underlying network. The underlying network has no flows and thus it does not know what to do with the packets.

Next step is to execute our software and then check again if the VMs can ping each other. What the software does is to communicate with the ODL-OS and check if there are any networks. In case there are, send to ODL-NET to create a vBridge with name vbr-{subnetwork_id}-{vlan_id}. Where network_id and vlan_id the network_id obtained from ODL-OS with the vlan_id it belongs to, in our case vlan_id 10. The final step of

our software is to vlan mapping, map the physical network resources to the vbr created using vlan_id of the incoming L2 frame, and the ping is successful now.

As a last step of our demo is to check if the software can work with multiple vlan networks. Thus, we created two more networks with vlan_id 20 and 30 and two VMs on each network(one VM is located one control node and one on compute node). Then execute the software and check if the VMs on the same vlan can ping each other. The ping is working, also we can observe that the ODL-NET has three vBridges, vbr-{network_id}-10, vbr-{network_id}-20 and vbr-{network_id}-30 and each vBridge is mapped with the appropriate vlan_id.

6.5.2 Details

Using the ovs-vsctl show command we can see for each node the ovs-vswitchd configuration database(known as ovs-db). So for control node we have:

```
stack@control:~/devstack$ sudo ovs-vsctl show
9474ae2e-8c46-4ce3-95a8-436c1eb6b142
    Manager "tcp:10.0.1.84:6640"
        is_connected: true
    Manager "ptcp:6641:127.0.0.1"
        is_connected: true
    Bridge br-ex
        Port br-ex
            Interface br-ex
                type: internal
    Bridge br-int
        Controller "tcp:10.0.1.84:6653"
            is_connected: true
        fail_mode: secure
        Port "tapdd919248-48"
            Interface "tapdd919248-48"
        Port "eth1"
            Interface "eth1"
        Port br-int
            Interface br-int
                type: internal
        Port "tap36d92a10-90"
            tag: 4095
            Interface "tap36d92a10-90"
                type: internal
        Port "tun330ffd41d40"
            Interface "tun330ffd41d40"
                type: vxlan
                options: {key=flow, local_ip="10.0.1.84", remote_ip="10.0.1.85"}
    ovs_version: "2.9.0"
```

The bridge br-int is created and managed by OpenStack. We add the port eth1 on this bridge since we want to use the eth1 interface in order to communicate with our control node. When we create a network in OpenStack then a port added in br-int with a name tap followed by a random string the same goes with VM. So the two tap interfaces are one for our vlan network and one for our VM. Now the tun interface is created automatically and is used for vxlan networks. When we create a VM on a vxlan network the packets will go through this tun interface which is actually a tunnel and thus the packets will not go through eth1 but will use this tunnel instead. The bridge br-ex is created by OpenStack and is used for external network connectivity and since in this demo we do not need the external network connectivity we did not configure it. For compute node we have:

```
stack@compute:~/devstack$ sudo ovs-vsctl show
3e3e1a6b-d916-4fd3-beda-61ebd6c5f5eb
    Manager "tcp:10.0.1.84:6640"
        is_connected: true
    Bridge "s4"
        Controller "ptcp:6655"
        Controller "tcp:127.0.0.1:6653"
```

```
            is_connected: true
        fail_mode: secure
        Port "s4-eth1"
            Interface "s4-eth1"
        Port "s4"
            Interface "s4"
                type: internal
        Port "veth1"
            Interface "veth1"
    Bridge br-int
        Controller "tcp:10.0.1.84:6653"
            is_connected: true
        fail_mode: secure
        Port br-int
            Interface br-int
                type: internal
        Port "veth2"
            Interface "veth2"
        Port "tap1cc35942-1d"
            Interface "tap1cc35942-1d"
        Port "tunad397de4e6b"
            Interface "tunad397de4e6b"
                type: vxlan
                options: {key=flow, local_ip="10.0.1.85", remote_ip="10.0.1.84"}
    Bridge br-ex
        Port br-ex
            Interface br-ex
                type: internal
    Bridge "s3"
        Controller "ptcp:6654"
        Controller "tcp:127.0.0.1:6653"
            is_connected: true
        fail_mode: secure
        Port "s3-eth1"
            Interface "s3-eth1"
        Port "s3"
            Interface "s3"
                type: internal
        Port "eth1"
            Interface "eth1"
    ovs_version: "2.9.0"
```

The mininet network is on the same physical node as the compute node (node084 as shown in the topology figure). We can see that s3 bridge contains the interface eth1 and s4 bridge the interface veth1, the other two interfaces s3-eth1 and s4-eth1 are used for the connection between them. The bridge br-int is similar with our control node except that we have only one tap interface, which is our VM, since the network is managed by the control node.

Now if we execute our software in this topology the following vtn will be created. This vtn is controlled by the ODL-NET.

```
stack@compute:~/devstack$ curl --user "admin":"admin" -H "Content-type: application/json" \\
-X GET http://localhost:8181/restconf/operational/vtn:vtns/ | python -m json.tool
```

```
{
    "vtns": {
        "vtn": [
            {
                "name": "vtn1",
                "vbridge": [
                    {
```

```
                        "bridge-status": {
                            "path-faults": 0,
                            "state": "UP"
                        },
                        "name": "vbr_b7bda431_10",
                        "vbridge-config": {
                            "age-interval": 600,
                            "description": "creating vbridge"
                        },
                        "vlan-map": [
                            {
                                "map-id": "ANY.10",
                                "vlan-map-config": {
                                    "vlan-id": 10
                                },
                                "vlan-map-status": {
                                    "active": true
                                }
                            }
                        ]
                    }
                ],
                "vtenant-config": {
                    "description": "creating vtn",
                    "hard-timeout": 0,
                    "idle-timeout": 300
                }
            }
        ]
    }
}
```


As we can see from the json output a vtn with name vtn1 is created. Furthermore, we see that this vtn contains one vbrdige with name vbr-{subnetwork_id}-{vlan_id} and used vlan mapping to map our resources.

So, now that we have our vtn setup we can try to ping VM2 from VM1. The ping will be successful and the following flows will be added on each switch. For S3 switch:

```
stack@compute:~/devstack$ sudo ovs-ofctl dump-flows s3 -O OpenFlow13
 cookie=0x7f56000000000005, duration=125.419s, table=0, n_packets=6, n_bytes=556,
  send_flow_rem priority=10,in_port="s3-eth1",dl_vlan=10,dl_src=fa:16:3e:a5:78:c6,
  dl_dst=fa:16:3e:7f:f6:8d actions=output:eth1

 cookie=0x7f56000000000006, duration=124.420s, table=0, n_packets=5, n_bytes=468,
  idle_timeout=300, send_flow_rem
  priority=10,in_port=eth1,dl_vlan=10,dl_src=fa:16:3e:7f:f6:8d,
  dl_dst=fa:16:3e:a5:78:c6 actions=output:"s3-eth1"

 cookie=0x7f57fffffffffff, duration=2109.070s, table=0, n_packets=5475, n_bytes=458039,
  send_flow_rem priority=0 actions=CONTROLLER:65535
```

The first flow is for when the packets arrived in S3 from S4. When a packet arrive in port s3-eth1 and the vlan id(10), the mac source and the mac destination match the values above then the action is to output to interface eth1. The second flow is for when the packets arrived in S3 from VM1. Similarly with first flow, when a packet arrive in port eth1 and the vlan id (10), the mac source and the mac destination and the action is to output to interface s3-eth1.
For S4 switch:

```
stack@compute:~/devstack$ sudo ovs-ofctl dump-flows s4 -O OpenFlow13
 cookie=0x7f56000000000005, duration=145.250s, table=0, n_packets=14, n_bytes=1260,
  idle_timeout=300, send_flow_rem priority=10,in_port=veth1,dl_vlan=10,dl_src=fa:16:3e:a5:78:c6,
  dl_dst=fa:16:3e:7f:f6:8d actions=output:"s4-eth1"
```

```
cookie=0x7f56000000000006, duration=144.251s, table=0, n_packets=13, n_bytes=1200,
 send_flow_rem priority=10,in_port="s4-eth1",dl_vlan=10,dl_src=fa:16:3e:7f:f6:8d,
 dl_dst=fa:16:3e:a5:78:c6 actions=output:veth1

cookie=0x7f57ffffffffffff, duration=2128.903s, table=0, n_packets=557, n_bytes=50506,
 send_flow_rem priority=0 actions=CONTROLLER:65535
```

The flows are in the same logic as S3.

6.6 Future Work

As future work, we would like to add some extra features to our software. In addition to VTN feature of OpenDaylight we want to try out the Service Funciton Chaining (SFC) feature. SFC, also know as Network service chaining is a capability that uses SDN capabilities to create a service chain of connected network services (such as L4-7 like firewalls, network address translation [NAT], intrusion protection) and connect them in a virtual chain. This capability can be used by network operators to set up suites or catalogs of connected services that enable the use of a single network connection for many services, with different characteristics.

The primary advantage of network service chaining is to automate the way virtual network connections can be set up to handle traffic flows for connected services. For example, an SDN controller could take chain of services and apply them to different traffic flows depending on the source, destination or type of traffic. The SFC capability automates what traditional network administrators do when they connect up a series of physical L4-7 devices to process incoming and out coming network traffic, which may require a number of manual steps.

The software based on the information received by both controllers (ODL-OS and ODL-NET) will be able to manage the infrastructure network in more efficient way.

References

[1] "devstack". https://docs.openstack.org/devstack/latest/.

[2] "industry leaders collaborate on opendaylight project, donate key technologies to accelerate software-defined networking". https://www.linuxfoundation.org/press-release/industry-leaders-collaborate-on-opendaylight-project-donate-key-technologies-to-accele

[3] "mininet". http://mininet.org/.

[4] "network functions virtualisation". http://www.etsi.org/technologies-clusters/technologies/nfv.

[5] "open source mano". https://osm.etsi.org/.

[6] "opendaylight". https://www.opendaylight.org/.

[7] "opendaylight: A big step toward the software-defined data center". https://www.infoworld.com/article/2614152/sdn/opendaylight--a-big-step-toward-the-software-defined-data-center.html.

[8] "openstack". https://www.openstack.org/.

[9] "software-defined networking (sdn) definition". https://www.opennetworking.org/sdn-definition/.

[10] "what are sdn controllers (or sdn controller platforms)?". https://www.sdxcentral.com/sdn/definitions/sdn-controllers/.

[11] "what is virtualized infrastructure manager (vim)? definition". https://www.sdxcentral.com/nfv/definitions/virtualized-infrastructure-manager-vim-definition/.

[12] "what you must know about the new open-source opendaylight sdn controller". https://www.sdxcentral.com/articles/news/spotlight-on-daylight-sdn-consortium-open-source-controll 2013/02/.

[13] Benzekki Kamal, El Fergougui Abdeslam, and Elbelrhiti Elalaoui Abdelbaki. Software-defined networking (sdn): a survey. *Security and Communication Networks*, 9(18):5803–5833.

[14] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.