V-NM

Team UnnA

Epameinondas Kontothanasis [ekon@kth.se](mailto:ekon@kth.se)

Sikandar Khan [sikkha@kth.se](mailto:sikkha@kth.se)

Maria Haider [mariahai@kth.se](mailto:mariahai@kth.se)

Peiyue Zhao [peiyue@kth.se](mailto:peiyue@kth.se)

Jing Jin [jji@kth.se](mailto:jji@kth.se)

Ganapathy Raman [grma@kth.se](mailto:grma@kth.se)

Acknowledgements

We would like to thank everyone who has helped in the last five months to accomplish this project.

Our sincere thanks to Professor Dejan Kostic for planning and organizing the course and also for providing necessary guidance to learn and progress in the project.

A special thanks to our coach Georgios Katsikas for helping us throughout the project by providing constant feedbacks in every team meeting and also for the extra meeting which helped us to improve our pace of our project.

Last but not least, we would like to KTH administration for providing such as a course which helped us to improve our skills over networking domain and also made us to learn to work as a team.

# Abstract

The project builds on the idea of integrating the emerging concept of Software Defined Networking (SDN) with Network Function Virtualization (NFV). The basic notion of SDN is separation of Data and Control plane whereas the NFV is to interchange dedicated network appliances with the software running on servers. The main concept behind project NFV is to replace the traditional network management by modern approach using the said techniques. OpenFlow protocol and OpenStack framework are used in the implementation of this project, which allows to manage SDN and NFV with great flexibility. This project aims to demonstrate the usage of OpenStack, OpenDaylight and Mininet with the existing functionalities and some additional features built on top of OpenStack and OpenDaylight to suit the requirements provided. The project tries to depict various real world scenarios where the functions such as Webserver, Fileserver and various service infrastructures are realized using Openstack instances. The ISP network topology is virtualized using Mininet and the flows to control and route packet through the switches in Mininet are achieved through OpenDaylight. This project leverages the advantages of integrating SDN with NFV, thereby setting a trend to meet the real world challenges and provides an easy and cost effective way for the researchers.

# Background

Over the last year, the hottest topics in networking have been software defined networking (SDN) [1] and Network Function Virtualization (NFV) [2]. There is, however, considerable confusion amongst enterprise IT organizations regarding these topics. The confusion arises from the fact that there are a sheer number of vendors out there who are offering solutions that solve different problems using different solution architectures and technologies and all of them claim to be offering SDN and/or NFV solutions.

As noted, the traditional data network has been largely hardware-centric. For example, in the mid to late 2000s, network appliances such as WAN Optimization Controllers (WOCs) [3] and Application Delivery Controllers (ADCs) [4] were purpose-built, hardware appliances. However, over the last few years the adoption of virtualized network appliances and the burgeoning interest in software defined data centers (SDDCs) [5] have lead a movement towards an increased reliance on software-based network functionality.

A SDDC can be looked at as the complete opposite of the traditional data center network that was previously described. For example, one of the key characteristics of a software-defined data center is that all of the data center infrastructure is virtualized and delivered as a service. Another key characteristic is that the automated control of data center applications and services is provided by a policy-based management system. One of the characteristics that is often associated with any fundamentally new approach to technology is that there is confusion about the opportunities that can be addressed by that new approach. In order to successfully evaluate and adopt a new approach to technology such as SDN, IT organizations need to identify which opportunity or opportunities that are important to the organization are best addressed by that new approach.

The Open Networking Foundation (ONF) [6] is the group that is most associated with the development and standardization of SDN. According to the ONF [6], *“Software-Defined Networking (SDN) is an emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of today’s applications. This architecture decouples the network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services. The OpenFlow™ protocol is a foundational element for building SDN solutions.”*

According to the ONF [6], the SDN architecture is:

* *Directly programmable* - Network control is directly programmable because it is decoupled from forwarding functions.
* *Agile* - Abstracting control from forwarding lets administrators dynamically adjust network-wide traffic flow to meet changing needs.
* *Centrally managed* - Network intelligence is (logically) centralized in software-based SDN controllers that maintain a global view of the network, which appears to applications and policy engines as a single, logical switch.
* *Programmatically configured* - SDN lets network managers configure, manage, secure, and optimize network resources very quickly via dynamic, automated SDN programs, which they can write themselves because the programs do not depend on proprietary software.
* *Open standards based and vendor neutral* - When implemented through open standards, SDN simplifies network design and operation because instructions are provided by SDN controllers instead of multiple, vendor-specific devices and protocols.

On the other hand Network virtualization isn’t a new topic as network organizations have a long history implementing techniques such as virtual LANs (VLANs), virtual routing and forwarding (VRF) and virtual private networks (VPNs). One way to implement network virtualization is as an application that runs on a SDN controller, leverages the OpenFlow protocol and defines virtual networks based on policies that map flows to the appropriate virtual network using the L1-L4 portions of the header. This approach is often referred to as fabric-based network virtualization.

Another way to implement network virtualization is to use encapsulation and tunneling to construct multiple virtual network topologies overlaid on a common physical network. This approach is often referred to as overlay-based network virtualization. IT organizations have been implementing network virtualization via overlays for the last few years based on protocols such as VXLAN. But the drawback of this method is that controller-less solutions typically used flooding as a way to disseminate information about the end systems, these solutions didn’t scale well.

The most recent way of realizing Network Function Virtualization features a controller and the network elements are either vSwitches or vRouters. One of the primary roles of the controller is to provide tunnel control plane functionality. This functionality allows the ingress device to implement a mapping operation that determines where the encapsulated packet should be sent to reach its intended destination VM. The primary benefit of an overlay-based network virtualization solution is that it provides support for virtual machine mobility independent of the physical network. If a VM changes location, even to a new subnet, the switches at the edge of the overlay simply update their mapping tables to reflect the new location of the VM.

SDN and NFV being the hot topics over the recent years, there has been significant amount of research and development being carried out in these areas. OpenDaylight [7], a controller software for installing and monitoring the flows on the SDN switches has gained significant amount of attention in the market. Many companies have deployed SDN controlled switches and uses OpenDaylight as there controller. Also another SDN controller called Floodlight controller [8] is being used in certain research works. OpenFlow [9], the de facto protocol for SDN are used on both the controllers. On the other hand to virtualize the network functions, OpenStack [10], a cloud managing software is used to deploy and manage multiple companies infrastructure through virtual instance has gained its popularity recently. Also another controller for NFV is developed by Wisconsin Madison which is called OpenNF [11]. But OpenStack is being used widely and known for its complete roll-out of the infrastructure in few hours.

Some significant measures have been taken to integrate SDN with NFV as an application running on top of SDN. A notable effort has been done by Madhu Venugopal et al from RedHat systems [12]. Here the authors have integrated the SDN with NFV as an application over SDN. In their design, the author has Mininet [13] switches in substitution of Physical switches, OpenDaylight for installing and controlling flows over Mininet and OpenStack controller for NFV. The author launches instances on OpenStack compute nodes. The Mininet acts as a core layer in networking domain or can be considered as set of switches belonging to a certain ISP. When the instance from one compute node wants to communicate with another instance residing in the other compute node, the packet flows through Mininet switches over the path instructed by OpenDaylight installed flows. Here the authors have taken the initial steps to combine OpenStack and OpenDaylight through customization. This project is available on the author’s website which contains a virtual machine with version of OpenStack and OpenDaylight pre-installed. But it contains only limited features to roll-out the services in real world.

# Introduction

To extend the idea of Madhu et al. further, this project focuses to integrate OpenStack (NFV) as an application on top of SDN switches. The switches act as core switches from an ISP where the flows are controlled by OpenDaylight. To extend the scenario and to make it realistic, the project tries to depict a simple scenario of two companies. The host of the two companies are created in the Mininet and are connected to the access layer of the Mininet topology which are in turn are connected to core switches. The core switches are then connected to the external world. Moreover there is a pool of servers created in the Mininet that are commonly used by both the companies. The packets from the host in the Mininet that are destined to the external world are routed using the pre-assigned flows installed on the Mininet switches by the OpenDaylight. It is to be noted that Mininet switches route packet by default using ‘Flooding’. But to demonstrate the purpose of SDN flows, default flooding has been disabled in Mininet switches. Packets that correspond to a particular flow are routed on per-hop-behavior based routing. At any point of time, if the switch does not contain any flows, packets are dropped. On the OpenStack side, each company represents one tenant so that they can be managed separately. Each company contains multiple instances placed and managed on various networks in the network topology created in OpenStack. For this project the instances are considered as servers that belong to particular a company. Dedicated servers are placed under one tenant that is allocated to a particular company but are distributed over various compute nodes. The compute nodes are OpenStack nodes with large storage space mainly used for holding the instances of the two companies which are centrally administered by OpenStack controller node. Each tenant contains multiple networks for administrative purposes. Access security and policy are applied for each server but mostly for the whole tenant. In this project three tenants CompanyA, CompanyB and RemoteA are used to demonstrate various scenarios. The host of a particular company from Mininet is made to communicate with the servers on the OpenStack using the flows assigned on Mininet switches that take the packets to OpenStack from the Mininet. Inside the OpenStack, packets flow using virtual routing. Various access policies are applied for security purposes in the OpenStack network. Various scenarios such as NAT using floating IP (IP address assigned to particular instances from public network range), Routing inside OpenStack, Firewall to allow certain packet to and from certain clients, VPN with OpenStack tenants are created and demonstrated in this project. The project demonstrates simple routing in Mininet switches using the pre-assigned flows, routing using primary flows and re-routing of packets through secondary paths (High Availability) in the event of link/node failure, monitoring of switches and links to demonstrate load-balancing i.e. when the flows on certain link/switch exceed threshold then certain packets are re-routed through the nearby links for load balancing. Also a new software called ‘FlowGlance’ is developed for visualizing the flows. The software displays the path taken by the packets and dynamically updates the changes in the flows by monitoring all the switches and links. With all the scenarios mentioned above, the project aims to depict real time deployment and services to provide a new focus to solve the burning industrial issues. The following sections elaborate the project in detail.

# Problem statement

In the current era of networking, an introduction of a new service requires deployment of infrastructure across the network. Each of the networking devices need to be manually customized to deliver its purpose and it is not easy to reconfigure them without the risk of producing inconsistencies and inaccessibility in the network. The traditional approach is also not cost efficient where new devices need to be added to reflect any changes in the network.

# Problem

Implementation of a virtual network management system using modern technologies is required to eliminate the manual configuration and customization of the infrastructure in order to improve the performance and efficiency.

Hypothesis

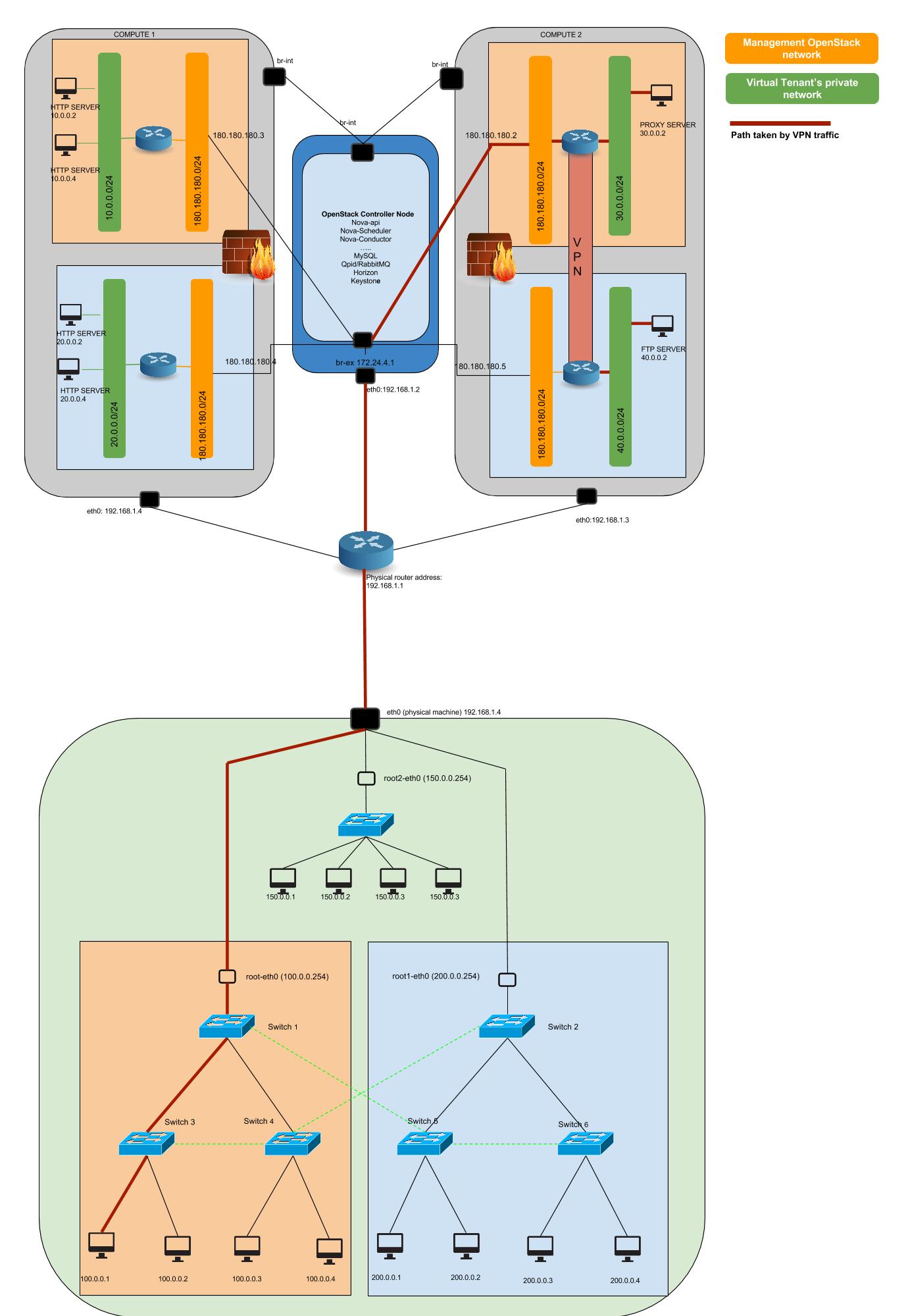
It is possible to build a virtualized environment over commodity physical infrastructure to simplify the management of network functions.

# Goal

Setup a Virtual Network Management system that consists of a virtual topology and implement functionalities to test and monitor the network. Usage of various scenarios to analyze how the topology would respond in different settings.

# Measurable objectives

The measurable objectives are to isolate the clients while guarantying QoS and security, and to handle the traffic in different load scenarios and to counter errors by providing redundant solutions.

Project Architecture *Figure 1: Overall project architecture*

The architecture of the project can disseminated into three major part

1. Mininet topology – Represents emulate the physical topology with virtual switches and host as in real world.
2. Controller [7] – For controlling the flow of packets though the Mininet with the help of pre-assigned flows.
3. Network Function Virtualization (NFV) [10] – To virtualize and manage various network function such as Load Balancing, Web Service etc.

Below sections describe in detail about Mininet, OpenDaylight and OpenStack in detail. It also contains the method(s) to install the three software. Also various issues that were faced and resolved are also mentioned.

# Mininet

Mininet is a *network emulator*. It runs a collection of end-hosts, switches, routers, and links on a single Linux kernel. It uses lightweight virtualization to make a single system look like a complete network, running the same kernel, system, and user code. A Mininet host behaves just like a real machine; you can ssh into it (if you start up sshd and bridge the network to your host) and run arbitrary programs (including anything that is installed on the underlying Linux system.) The programs you run can send packets through what seems like a real Ethernet interface, with a given link speed and delay. Packets get processed by what looks like a real Ethernet switch, router, or middlebox, with a given amount of queueing.

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

## Installation

If the system is running Debain or RedHat distribution, then Mininet can be downloaded from the repositories. To run on other OS, it is highly recommended to download the Mininet pre-installed image (.ovf or .vmdk) from [13]. It also contains the self-explanatory guide to start up. Mininet is more convenient to write and customize own topology using python code. The code used to setup the topology for this project can be downloaded from [14].

Mininet Architecture

The Mininet topology used for this project is a miniature replica of the Stanford topology [15]. This entire topology is emulated under a single machine. The topology consist of 7 switches and 8 hosts and 4 servers in total. 7 switches are grouped into four category. Top switch is where the pool of servers are connected. Next two switches (Switch1 and Switch2) forms the core layers, Switch 3, 4, 5 and 6 forms Access layer in the topology where the hosts of the company are connected.

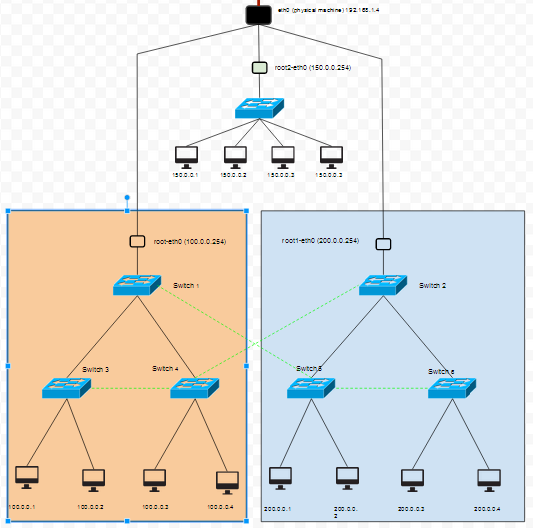


Figure : Mininet Architecture

The set of switches and hosts which are part of two different colors represent the two different companies’ infrastructure. As per assumption, the entire Mininet topology belongs to a single ISP. The redundant links are added to provide a seamless connectivity of the hosts in the Mininet to the OpenStack even at cost of link or node failures. The top core switches are connected to virtual interfaces (root-eth0, root1-eth0, root2-eth0) which are created on the root-space in the kernel. These interfaces are configured to receive and forward all packets to eth0 which act as an external gateway interface.

# OpenDaylight

OpenDaylight is a collaborative, open source project to advance Software-Defined Networking (SDN). OpenDaylight is a community-led, open, industry-supported framework, consisting of code and blueprints, for accelerating adoption, fostering new innovation, reducing risk and creating a more transparent approach to Software-Defined Networking.

SDN enables users to program network layers, separating the control plane and data plane. By enabling programmability, SDN can enable users to optimize their network resources, increase network agility, service innovation, accelerate service time-to-market, extract business intelligence and ultimately enable dynamic, service-driven virtual networks.

SDN presents both significant challenges and unlimited opportunities for the future of transferring information and communicating across the globe. The companies coming together understand that the best way to address this historical moment in their industry is to do it together. Collaborative open development and open source software are the driving force behind modern architectures and well recognized for accelerating technology innovation and adoption.

OpenDaylight will provide a common platform on top of which vendor products and services can be built, giving vendors the room to innovate and compete and provide users with the best solutions at a rapid pace.

## **Installation**

Version-Hydrogen Edition

OpenDaylight was installed from [16]. It includes a virtual machine containing a pre-configured OpenDaylight software. The minimum requirement for memory to run the ODL is 4GB. The OpenFlow version used for the OpenDaylight is 1.3, the project used other versions of OpenDaylight too. Primarily the work was done with a Devstack VM which has OpenDaylight integrated with Openstack.

## **OpenDaylight - Architecture used for this project**

Mininet Topology View from OpenDaylight GUI

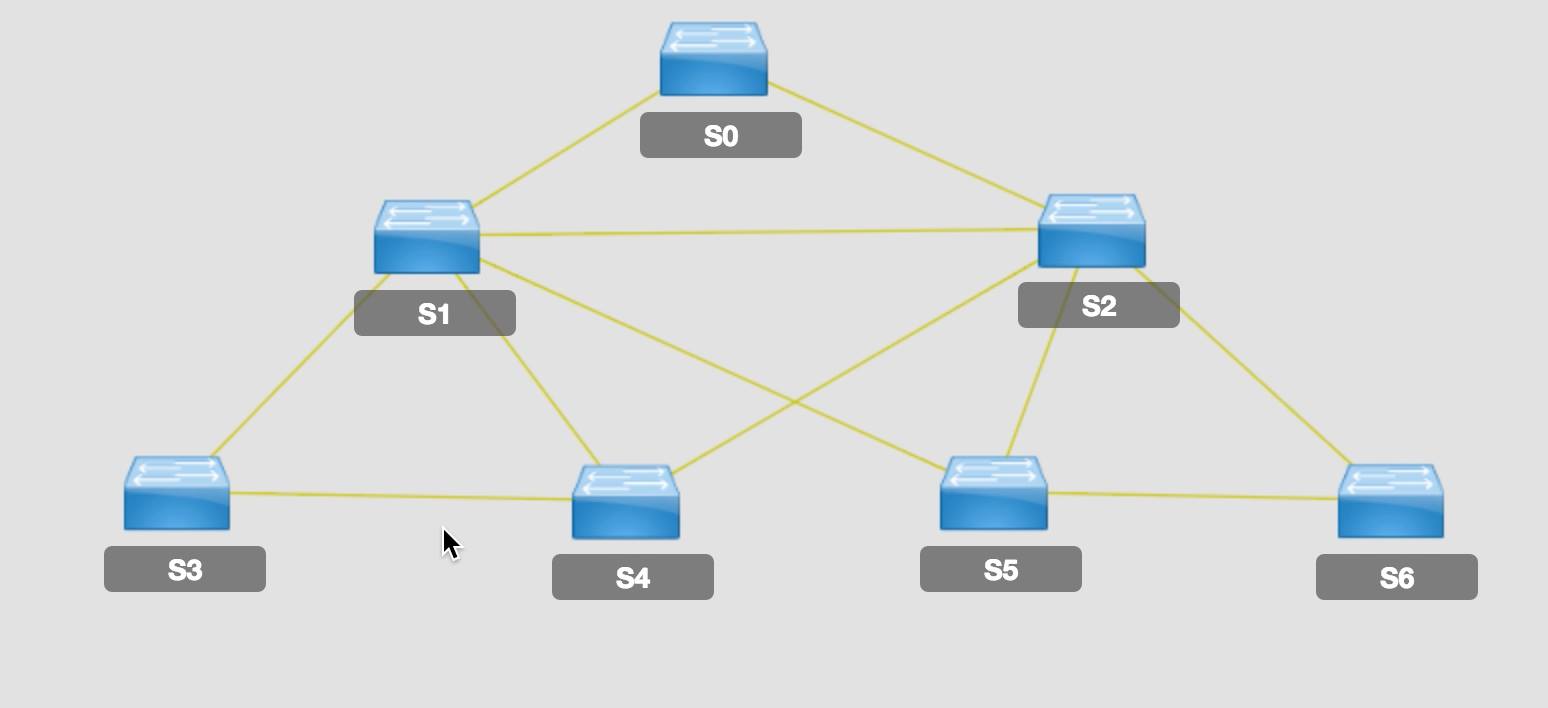


Figure : Mininet Topology view from ODL

The whole topology has been divided to accommodate two companies. Each company constitutes of three switches (two access and one distributor switch) and four hosts. From the picture above, S1, S3 and S4 belong to company A and S2, S5 and S6 belong to company B. In case of a link failure between the access switch and hosts, link between the second access switch and the hosts from the same company are used. In case, all of the links of a company fail, the redundant link between the two companies is used for recovering the connection.

Third part of our topology consists of a single switch with pool of servers. It is denoted as S0 in the picture above. The purpose of the pool of servers is twofold. It shows that local servers are accessible by both the companies which are equivalent to physical servers and not necessarily virtual ones. The pool of servers also protect from flooding the internal company networks with external data.

The motivation of the topology is to show an ISP with isolated companies. The design makes it simpler to display different network functions and services to model a complete view of a functional and practical network.

## **Scenarios Demonstrated in the project**

The main function in the Opendaylight is to control the traffic in the network. Though assigning flows in Opendaylight, the network packets could be controlled and forwarded flowing some specific paths. Basically, flows in Opendaylight could be understand as the forwarding rules of packets based the conditions like source address, destination address, protocol type etc. Through the REST API, flows can be added, edited, removed by different Json methods like post and get. By manipulating flows properly in Opendaylight, services like redundancy and load balancing could be implanted. In this project, SDN Redundancy and load balancing service are implemented as third-party applications for the Opendaylight platform.

Scenario 1 – SDN Redundancy Service: For the efficiency of the network, primary flows were designed to forward the packets in the shortest paths. By assigning primary flows, the packet will only be forward through the shortest paths instead of flooding. However, the links contained in the shortest paths may be broken or down. The aim of SDN Redundancy service is to ensure the network connectivity as much as possible in terms of link failure. The SDN Redundancy detects the link conditions dynamically and it will push flows of the backup paths to the Opendaylight when necessary. In the demo Scenario, some links in the Mininet will be turned off to test the redundancy and the FlowGlance will be used to monitor the link changes and traffic shifts.

Scenario 2 – SDN Load balancing Service: The purpose of SDN Load balancing service is to balance the traffic between different physical links. The load balancing can help make full use of the network links and prevents the some links being jammed while some links are idle. The load balancing is implemented by a load balancer which follows some pre-designed load balancing rules. A load balancing rule contains two parts: trigger conditions and actions. According to the trigger conditions, the Load balancer will check the data rate of a specific port in the network periodically. When the data is higher than the upper bound or lower than the lower bound, the load balancer will finish the actions defined in the load balancing rules. In practical, the some load balancing rules are not independent: the trigger of one rule is based the trigger of another rules. In this projects, several rules forms a load balancing chain which works as a simple state machine. The load balancing rules in a load balancing chain will be triggered in order. The load balancer support several load balancing chain at same time. In the demo scenario, a host downloads a huge file from a remote file server. The load balancer balances the downloading traffic between two links. The FlowGlance presents the traffic and path shifts.

**FlowGlance – A software to display the packet flows on Mininet**

In the perspective of network administration, GUI based network monitor could ease the network management and troubleshooting. Besides the topology provided in the Opendaylight or the HyperGlance, a dynamic view of the ongoing traffic is needed. FlowGlance is a python program designed in this project to present the topology and the traffic in a SDN network. Links in different conditions (up or down) and links of different data rates are dynamically colored differently in FlowGlance. FlowGlance is implemented by proceeding the network data from Opendaylight REST API. FlowGlance will be used in the demonstration scenarios as a network monitor.

**Issues faced and resolved in OpenDaylight**

* Memory inadequacy

The Devstack version of OpenDaylight requires 4GB. Some of our computers were not compatible memory wise to run it. Finally the RAM was replaced and computers for running ODL smoothly.

* Switch ID problem

The OpenDaylight from the Devstack shows switch IDs in different formats which were not compatible with our codes for running the flows.

* Version problem

The OpenDaylight version used is Hydrogen. The new releases Helium cannot work for our project because the flow installing python file cannot work on this version.

# OpenStack

OpenStack is a free and open-source cloud computing software platform. Users primarily deploy it as an infrastructure as a service (IaaS) solution. The technology consists of a series of interrelated projects that control pools of processing, storage, and networking resources throughout a data center—which users manage through a web-based dashboard, command-line tools, or a RESTful API.

OpenStack lets users deploy virtual machines and other instances which handle different tasks for managing a cloud environment on the fly. It makes horizontal scaling easy, which means that tasks which benefit from running concurrently can easily serve more or less users on the fly by just spinning up more instances. For example, a mobile application which 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.

## **Installation**

Version: IceHouse Edition

Installing OpenStack may be cumbersome and time consuming process. So for quick start it is advisable to download Devstack, where most of the parts are customized and pre-configured by which one can make OpenStack up and running in few minutes. For this project the latest and stable version ‘*ICEHOUSE*’ of OpenStack has been chosen. This can be downloaded from the GitHub and the step can be followed as mentioned in this site [17] for installation. Devstack can be installed on Virtual Machine or can be directly installed on the Linux machines. For this project DevStack has been installed over Ubuntu 14.04 LTS machine.

Generally OpenStack nodes can be classified into two, Controller Node which can be considered as a managing node of the overall enterprise. Compute Node, which act as storage nodes for the virtual instance launched. These compute nodes are administrated and controlled centrally by the compute nodes. So for this project three laptops were used where one laptop will act as a Control Node and rest two will act a Compute Node. Devstack has to be installed on all three machines. But controller and compute uses different ‘*localrc*’ files. The localrc for compute and control node can be downloaded from [18].

**OpenStack - Architecture used for this project**

The below picture depicts the setup made for this project. It consist of two tenants CompanyA and CompanyB. Also third tenant RemoteA is also used to demonstrate VPN service. The subnet 10.0.0.0/24, 30.0.0.0/24 belongs to CompanyA. The subnet 20.0.0.0/24 belongs to CompanyB. And subnet 40.0.0.0/24 belongs to RemoteA, where the clients on the network 30.0.0.0/24 and 40.0.0.0/24 interacts through VPN. All the instance are distributed among the compute nodes. It is to be noted that two companies’ instances can be on the same compute node since they are virtually separated and managed by control node. Also the instance belongs to same network can also be distributed among the compute nodes. The instance that belong to the same network are tagged with a specific VLAN. By this way instance that are physical separated i.e. that are in different compute nodes, but which are logical under same subnet can communicate with each other same as the host under same network in real world. The virtual routers are created and communicate using namespace. These routers helps to bridge the private networks to the public external network. Also most of the services such as NAT, VPN, Firewall, and Load Balancing are done with on these virtual routers.

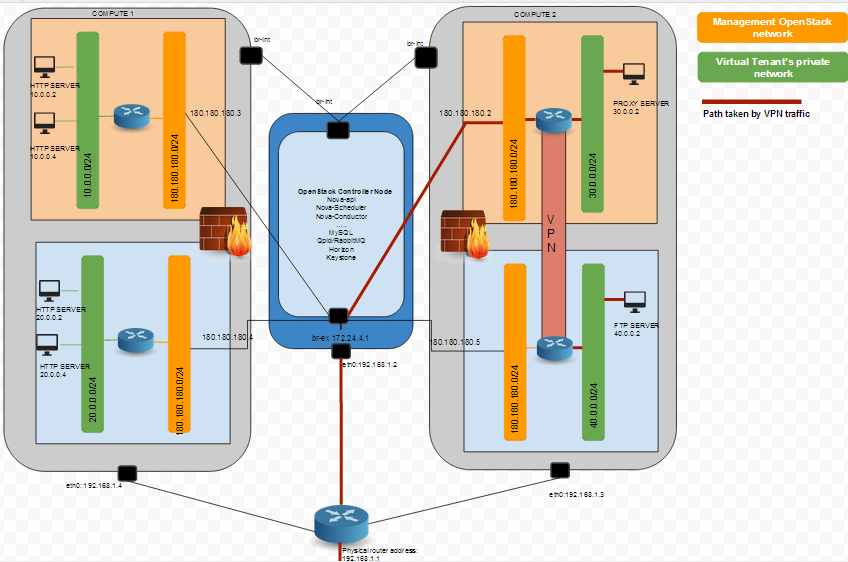


Figure : OpenStack Architecture

The packets that are destined to same network travel through internal port ‘*br-int*’, while the packet destined to other networks are to the external world takes the port ‘*br-ex*’. In this project the public network 180.180.180.0/24 is considered as the public network which is connected to the external world. For q-routers (virtual routers) inside the OpenStack has default gateway to 180.180.180.1 which is the IP address of br-ex.

CompanyA-Dashboard view

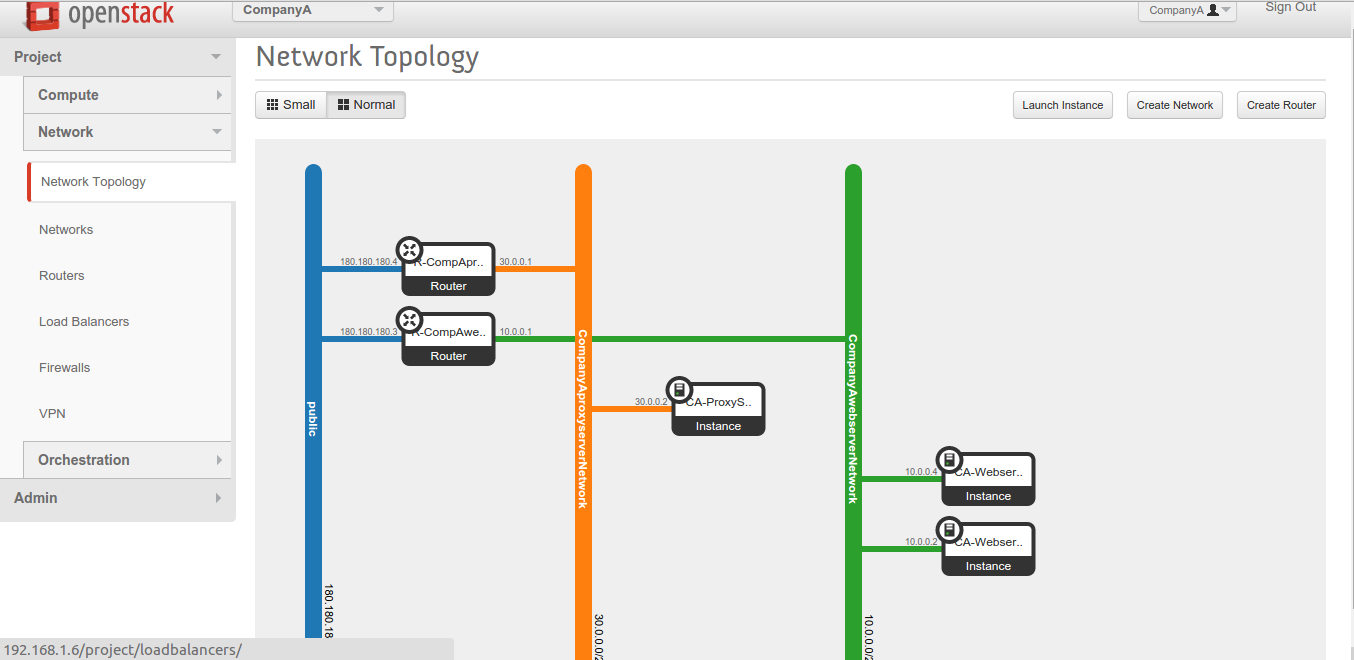


Figure : OpenStack dashboard view of CompanyA

CompanyB-Dashboard view

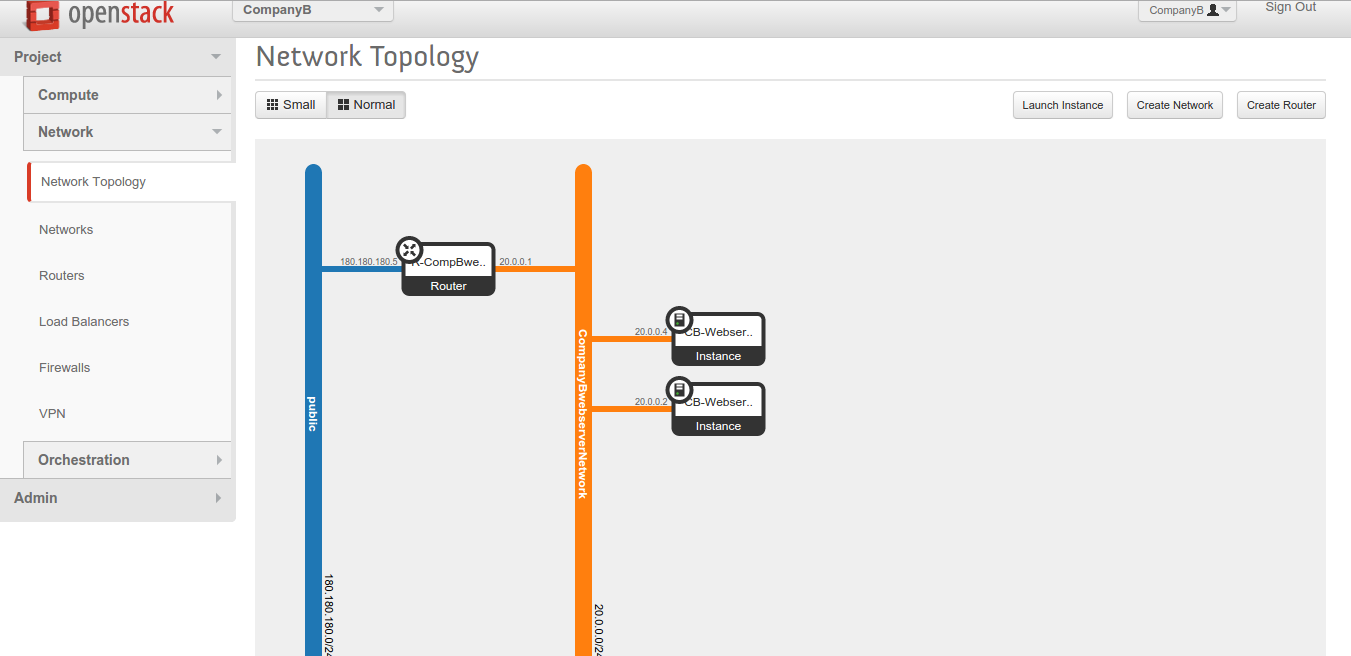


Figure : OpenStack dashboard view of CompanyB

RemoteA-Dashboard view

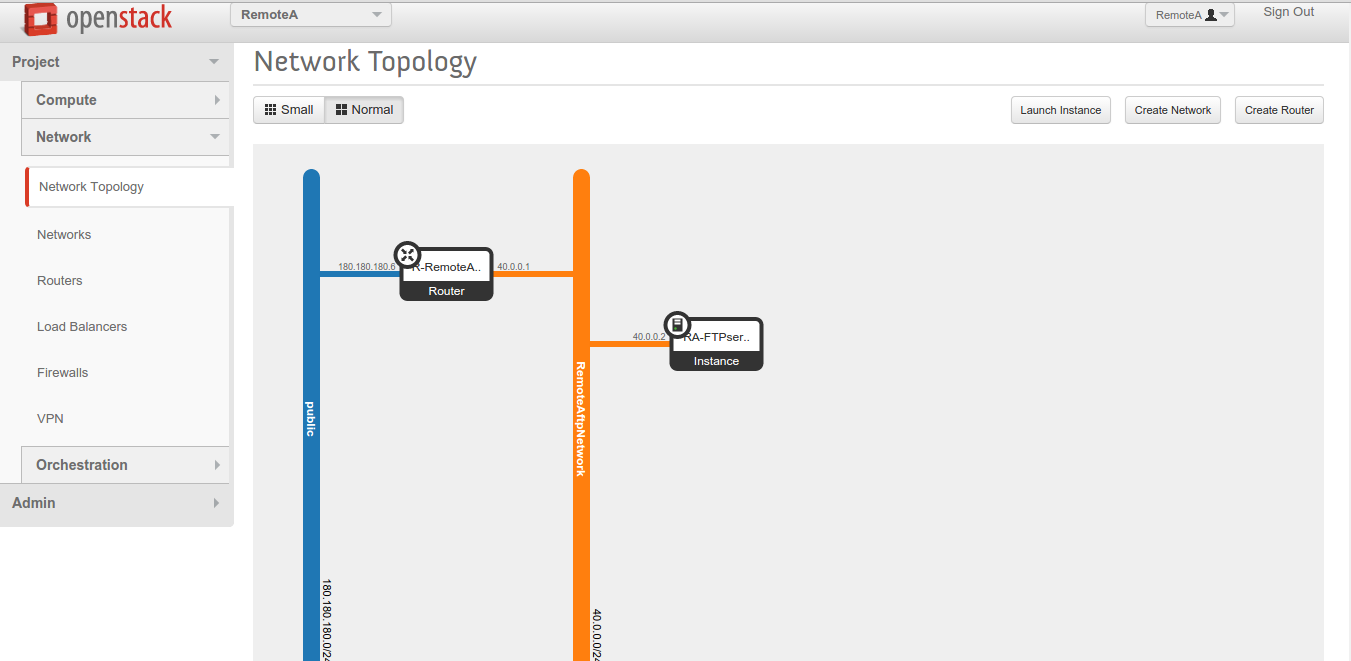


Figure : OpenStack dashboard view for RemoteA

**OpenStack Services**

### NAT

In OpenStack the NATing functionality can be ached through Floating IPs. The instance that are behind the private network and want to be NATed can be associated with a floating IP. This floating IP is provided in the range of public network. By which the entities in which are in the outside world can communicate with the instance which are behind the NAT through floating IP. This service is enabled by default.

### Routing

The routing is achieved through the virtual router (q-routers) that are present in the OpenStack. These routers enables the routing by interconnecting two internal networks or to the external world by internal network with the public network via br-ex. This service is enabled by default.

### Firewall

It is to be understand the there are two kinds of policy based firewall/access list. *‘Access Security’* which is applied for each instances. The rules may include to allow or deny ICMP, TCP, UDP, SSH etc. packets either ingress or egress to the instances. But using Firewall as a service can be applied to the entire tenant. Firewall rules can be applied specific to a network either allowing or denying ICMP, TCP, UDP or any packet. To enable this service Firewall drivers have to be present. To enable Firewall as service, neutron service needs to be enabled in localrc file by adding *‘q-fwaas’* under ENABLE\_SERVICE.

### Load Balancing

In OpenStack, one can have multiple Load Balancers. Various instances can be grouped into pools. And for each pool can be associate with a type of load balancing to be achieved. Either load balancing based on *‘Round Robin’* or *‘SOURCE\_IP’* or *‘Least Connection’.* Persistence can be provided based on *‘SOURCE\_IP’ or ‘HTTP\_Cookie’ or ‘APP Cookie’*. Also the load balancing can be applied based on various protocols such as HTTP, ICMP, UDP etc. To enable this service Firewall drivers have to be present. If not it can be downloaded using *‘apt-get install neutron-lbaas-agent’*. To enable Load Balancing as service, the neutron service needs to be enabled in localrc file by adding *‘q-lbaas’* under ENABLE\_SERVICE.

### VPN

In OpenStack, the VPN as service can be provided to the networks that are on different tenants or belongs to different controller. To setup VPN it is necessary to setup IPSec, IKE Configuration, IPSec-site-connection and then VPN service configurations. It requires the peer’s router’s IP address on the public interface. Also the remote private network and the secret pass passphrase. VPN requires bidirectional setup. To enable VPN as service, neutron service needs to be enabled in localrc file by adding *‘q-vpn’* under ENABLE\_SERVICE.

**Scenarios Demonstrated in the project**

Scenario 1 - Load Balancing of Webservers:

This scenario demonstrates the load balancing as a service with the help of Webserver as an example. In OpenStack, the two companies i.e. the tenant CompanyA and CompanyB has a pool of Webservers on the networks 10.0.0.0/24 and 20.0.0.0/24 networks. When the client on the Mininet of the tries to access their corresponding companies’ webserver, then request must be send load balanced on the OpenStack and then request must be forwarded to one of the webserver. Also the requests are assigned in ‘ROUND ROBIN’ fashion i.e. the first request from Host1 is forwarded to WebServer1 and processed. When the second host Host2 request for the webpage, based on the ROUND ROBIN method, the request is forwarded to the WebServer2. Once the all the servers are processing at least one request, then the new request is forwarded to the WebServer1 and the cycle continues. Also for the persistence in processing the request, *‘SOURCE\_IP’* is used i.e. when a request from a host is assigned to a certain Webserver the future request are also forwarded to the same server. So all request originating from the same IP are processed by the same Webserver. It is to be noted that the clients/host on the Mininet tries to establish HTTP connection with the VIP (Virtual IP) on which the load balancer is listening on port 80. On arrival of request to this IP on the HTTP port, the request are then forwarded to the Webserver internally.

To extend the scenario, on CompanyB’s Webserver servers renders webpage based on Source\_IP. In this scenario, a host on Mininet is considered that it belongs to Financial Department. So when the request arrives from the host on Mininet, they are send to load balancer then they are forwarded to any one of the available Webserver. Then based on the source IP the relevant webpage is rendered. For Financial host, financial webpage is rendered and for other client normal webpage is rendered. The source IP based request processing is handled by a program written on python which running on top of the webserver.

Scenario 2 – File Server using VPN:

This scenario demonstrates VPN as a service in OpenStack. The goal of this scenario is the client/host for the CompanyA on the Mininet has to receive a file from the Fileserver on the OpenStack through file transfer protocol. But the interesting fact is that client cannot directly interact with the server having the file. Instead the client contacts the proxy server on OpenStack which in turn fetch the file from the remote file server using VPN service. Client on Mininet initially tries to establish a TCP connection with the Proxy Server on network 30.0.0.0/24 listening on port 50000 under the tenant CompanyA. On receiving the request from the client, the Proxy Server suspends the current TCP request temporarily. Now the Proxy Server creates a new TCP request on behalf of the client and tries to contact the real Fileserver residing network on 40.0.0.0/24 under another tenant RemoteVPN through VPN connection. On Successful connection through VPN with the Fileserver, Proxy Server downloads the request file and terminates the new request with the Fileserver. Now the Proxy Server resumes the old TCP connection between the client on the Mininet and returns the file to the clients and client successfully downloads the request the file if the file exist else and error is returned to the client. All the program for client that request the file, the intermediate proxy server that receives the file download request and forwards it to Fileserver and the Fileserver that is listening for the file download request are written in Python language.

Scenario 3 – Inter-Company communication:

This scenario demonstrates inter-company communication between the companies. This is achieved by en-routing the request via Proxy Server on the Mininet. The client that belongs to CompanyA requests an employee search portal page of CompanyB from the proxy server. Then Proxy Server returns an employee search portal webpage. When the client search for an employee using the name of the employee the query is posted using GET method to the Proxy Server. Then Proxy Server suspends the HTTP request temporarily. Then Proxy Server tries to connect to the server running *‘Sqlite3’* database. On successful connection, Proxy Server request the employee details from the database server. On receiving the employee name database quires with the employee name the returns the details of the employee to the Proxy Server. Proxy Server upon receiving the details from the database server terminates the new TCP connection with the database server and resumes the old HTTP connection with the client and returns a HTML page with the employee details to the client. The program that is listening for the HTTP request on Proxy Server and the program that quires and returns the employee details are written in Python language.

**Major Issues faced in OpenStack and their solutions**

* Unable to Ping/SSH the instance launched in OpenStack

Initial work in this project was started with installing the *‘DevStack’* using the Virtual Machines as provided by the NetworkStatic forum [12] and followed the instruction as mentioned in the website. These virtual machine installed on Virtual Box Manager running over Window8 machine. The virtual machine has both OpenStack and OpenDaylight and they are integrated. So the both the topologies on Mininet and on the OpenStack together can be visualized on the OpenDaylight GUI. But the problem with the VM was, after launched the instance on OpenStack they are unreachable from the external world. Even with proper configuration and routes ping or SSH does not works.

This issue was resolved by directly installing DevStack on the system running Linux distribution. For this project demonstration Devstack was installed directly on the machine running Ubuntu 14.04 LTE.

* PIP package not found

This is a common error that occurs after running *‘clean.sh’.*  It remove all OpenStack dependencies. Simple solution would be install the pip from the repositories using command *‘apt-get install pip’* or *’yum install pip’.*

* Permission error

Sometime while stacking, process terminated abruptly due to insufficient permission of certain directories like neutron, nova, ceilometer etc. So to best method is to take the ownership of the entire directory recursively using chown <username> -R /opt/stack/

* Dependencies errors

Sometimes stack may fail due to various dependencies. The reason might be due to floating binaries and residual configuration from previous setup. The best solution is to remove the previous OpenStack and reinstall it. By doing so may lead to floating binaries. So it is a good practice to remove the folders installed by the OpenStack such as /opt/stack and all the OpenStack folders in /etc/ before re-installing.

* Unauthorized to access Admin Page on OpenStack GUI

This is a common error that occurs when the OpenStack dashboard opened and the user the tries to unstack and stack the OpenStack again. When the user tries to login with a valid credentials an error is thrown on the webpage ‘Unauthorized to access to admin page’. This is because of the cookies hold by the previous session. The simple solution is to access the OpenStack board through another browser.

* Internal Server Error

This error forbids the user from accessing OpenStack dashboard even when the user tries to login using valid credentials. This error is caused by apache2 webserver due to permission error in reading the file under the folder. /opt/stack/horizon/secret/. This error can be resolved by providing necessary permission to the folder or by modifying the content on the file /opt/stack/horizon/openstack\_dashboard/local/local\_settings.py.

In the file search for the line

*SECRET\_KEY = secret\_key.generate\_or\_read\_from\_file(os.path.join(LOCAL\_PATH, '.secret\_key\_store'))*

and edit it to ‘SECRET\_KEY = <anyNumber>’. Then restart the webserver using the command ‘sudo service apache2 restart’. Now the user can access the horizon OpenStack dashboard through GUI.

* Unable to access VNC console

This error occurs mostly when the command line interface of the instances launched on the compute node are inaccessible through the horizon dashboard. This may be due to fact the VNC server is not running on the compute machine or the VNC port on the compute machine is blocked to firewall settings or due misconfigurations of VNC server and client on the ‘localrc’ file. The best solution is to configure the VNC server and console port properly in the compute nodes in the ‘localrc’ file before stacking the OpenStack.

* Ping works but unable to wget or ftp from the OpenStack instance launched on Compute nodes – MTU size issue.

This is one of the most interesting and peculiar issue that is related to networking. From instances that are launched on compute node with proper access security the user can ping the external. But when the user tries to download any file using FTP or using Wget the instance command hangs. This error occurs due to fragmentation of packet mismatch. If neutron is used as a DHCP agent but the dhcp-option-force=26,1400 was not [set as [instructed](http://docs.openstack.org/admin-guide-cloud/content/ch_networking.html#openvswitch_plugin) on top of OpenVSwitch and GRE, all instances will get an MTU of 1500 which](http://docs.openstack.org/admin-guide-cloud/content/ch_networking.html#openvswitch_plugin) would require fragmentation. When OpenVSwitch uses such interfaces for GRE, it sets the [DF](http://en.wikipedia.org/wiki/IPv4#Header) flag which is [obeyed by interface](http://www.tinc-vpn.org/pipermail/tinc-devel/2013-December/000631.html) and the packet is not fragmented and never reaches destination.

The best solution is to lowering the MTU size of the interfaces. Since all the packets from instance pass through *‘br-ex’*, the MTU should be least. MTU size of 1500 bytes (default) has to be reduced as follows

MTU size of the interface attached to the external gateway on controller – 1454 bytes

MTU size of the interface connecting to control node on compute – 1454 bytes

MTU size of the ‘br-ex’ virtual interface on control node – 300 bytes

Conclusion

From the above section it can its evinced that the project has opened a new dimension to build the network infrastructure in the real world and has increased the scope for research. Integration of SDN with NFV provides a greater flexibility in rolling out the network infrastructure on the NFV quick and efficiently. Also it provide a great control over the routing of packet over the SDN controlled switches. SDN and NFV being an open source provides a freedom to expand the current networking world to take a better dimension instead of being dependent on vendors. From the demonstration by using OpenDaylight as controller for SDN switches on Mininet and OpenStack for rolling out NFV platform along with the various miniature of real time scenario on both OpenDaylight and OpenStack has reduced the burden on ISP provider in setting up the network topology and monitoring the switches has been reduced. Also adopting SDN technology save the network operation teams time in setting up and running complex routing protocols to adopt to the situation dynamically. Big companies like Google have shifted the focus towards SDN and have already deployed infrastructure using SDN controlled switches. On the other hand improvement over NFV and its easy of rolling out the network infrastructure quickly and the provision to expand when need has made the cloud based companies like Amazon and Microsoft has stated offering cloud based services using NFV. FlowGlance, the software built for this project can be used as quick tool for monitor the flows over Mininet switches mainly on the areas where running HyperGlance (heavy monitoring tool) is difficult. By using the concepts of both (SDN and NFV) cutting edge technologies has increased the importance of this project manifolds. The project has also provided a suitable method for the researches to build, deploy, test and expand the network infrastructure for their research in efficient and reliable way.

Future Work

Further the project can be extended to a big scenario with more companies and infrastructure manage.

OpenDaylight

In particular, mechanism for load balancing and redundancy can provided as in-built add-on solution by optionally installing them as Java bundles or provided a mechanism to configure these functions via REST APIs.

OpenStack

Apart from contributing a stable OpenStack release, high availability of moving the instances from one compute node with all its states intact to a new compute node will make OpenStack a more reliable solution for cloud based NFV.

References:

[1] Cisco’s, *“SDN, Transformation Through Innovation”.* Available at: <http://www.cisco.com/web/solutions/trends/sdn/index.html>

[2] F5 white paper, *“Network Functions Virtualization—Everything Old Is New Again”*. Available at: <http://www.f5.com/pdf/white-papers/service-provider-nfv-white-paper.pdf>

[3] EMC2, *“WAN Optimization Controller Technologies”.* Available at: <http://www.emc.com/collateral/hardware/technical-documentation/h8076-wan-optimization-tb.pdf>

[4] Citrix white paper. *“Application Delivery Controller”.* Available at:<https://www.citrix.com/content/dam/citrix/en_us/documents/products-solutions/what-is-an-application-delivery-controller-adc.pdf>

[5] Forbes *“Two Vital Keys For The Future Of The Software-Defined Data Center”.* Available at: <http://www.forbes.com/sites/symantec/2014/09/25/two-vital-keys-for-the-future-of-the-software-defined-data-center/>

[6] ONF, *“Software Defined Networking”.* Available atL <https://www.opennetworking.org/sdn-resources/sdn-definition>

[7] The OpenDayLight Project. Available at: <http://www.opendaylight.org/project>

[8] Project FloodLigt *“Open Source Software for Building Software-Defined Networks”.* Availabe at: <http://www.projectfloodlight.org/floodlight/>

[9] ONF, *“Openflow protocol”.* Available at: <https://www.opennetworking.org/ja/sdn-resources-ja/onf-specifications/openflow>

[10] OpenStack. Available at: <http://www.openstack.org/>

[11] OpenNF. Available at: <http://opennf.cs.wisc.edu/>

[12]Network Static, Brent Salisbury’s Blog. Available at: <http://networkstatic.net/opendaylight-openstack-integration-devstack-fedora-20/>

[13] Mininet, *“An Instant Virtual Network on your Laptop (or other PC*)”. Available at: [www.miniet.org](http://www.miniet.org)

[14] Bitbucket, Group2UnnA. Avaible at: <https://bitbucket.org/group2unna/ik2200/src/4e9828a5fe33861bebcd49b3fdde3adf0c4a524d/Mininet/?at=Final>

[15] Mini-Stanford topology. Availabe at: <https://bitbucket.org/peymank/hassel-public/wiki/Mini-Stanford>

[16] Fedora 20, All-in-One Image. Available at: <https://wiki.opendaylight.org/images/HostedFiles/Fedora20_ODL_OpenStack.zip>.

[17] Openstack Icehouse install using Devstack. Available at: <http://sreeninet.wordpress.com/2014/05/11/openstack-install-using-devstack/>

[18] Midokura Github. Available at: <https://github.com/midokura/odp-ovsdb/tree/master/resources/openstack>