Université Catholique de Louvain

Master Thesis

Cloud Network Performance Analysis: An OpenStack Case Study

Supervisor:
Marco Canini
Author:
Tuan-Anh BUI
Reader 1:
Viet Hoang Tran
Reader 2:
Ramin Sadre

A thesis submitted in fulfilment of the requirements for the degree of Master in Science (120 credits)

Option: Software engineering and programming systems

January 2016

UNIVERSITE CATHOLIQUE DE LOUVAIN

Abstract

Faculty Name
Department of Computer Science and Engineering (INGI)

Master in Science (120 credits)

Cloud Network Performance Analysis: An OpenStack Case Study

by Tuan-Anh BUI

The last decade has witnessed the fast growth of Cloud Computing (CC) paradigm in the ICT world, drawing lots of attention from academia and industry. The increasing popularity of cloud operating systems, supported by the vastly decreased cost of commodity hardware, makes deploying and managing a CC data centre more feasible than ever. So far, OpenStack has been the main character behind some of the most successful stories in the Cloud service market nowadays. Academic literature, however, has little insight in how OpenStack could empower raw hardware infrastructure and turn it into scalable, on-demand pools of computing resources that can satisfy even the most computationally intensive applications and services. This thesis work aims at deploying a fully functional OpenStack cluster with different settings and operational environments, followed by an in-depth analysis of its network performance and explanation of what happens behind the scene. Using a methodological approach and having carried out numerous experiments, we present various scenarios where OpenStack Virtual Machines perform at their best and worst with regards to network performance. Eventually we are able to draw conclusions on the impacts that the Networking module places on the overall OpenStack network performance.

Acknowledgements

I would first and foremost like to express my deepest gratitude to my supervisor, Professor Marco Canini, for his precious guidance and invaluable suggestions, for sharing his knowledge and professional experience with me, and beyond all, for being so inspirational and sympathetic to me.

Also, I would love to send my special words of thanks to Professor Ramin Sadre, and Viet-Hoang Tran, who spend their valuable time reading my work and provide me with highly constructive feedbacks.

I am much obliged to my dear friend, Quynh Le, who sticks by me through the thick and thin, giving me countless support in my personal life. My gratefulness also goes to my other two friends Nhung Nguyen and Alice Egan, whose proofreading effort is profoundly essential to my work. And I wholeheartedly appreciate all other friends for their inspirational ideas and discussions, as well as for the time spent with me.

Last but not least, I owe extreme indebtedness to my beloved family who always encourage me and help me get through difficult times.

Contents

Abstract i

Acknowledgements ii

Contents iii

List of Figures vi

List of Tables viii

Abbreviations ix

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 2

2 Background 4
- 2.1 Cloud Computing . . . 4
  - 2.1.1 Definition . . . 4
  - 2.1.2 Cloud Computing models . . . 5
- 2.2 OpenStack . . . 6
  - 2.2.1 The OpenStack Project . . . 6
  - 2.2.2 OpenStack Software Components . . . 7
    - 2.2.2.1 Compute service . . . 8
    - 2.2.2.2 Storage service . . . 8
    - 2.2.2.3 Networking service . . . 8
    - 2.2.2.4 Dashboard service . . . 9
    - 2.2.2.5 Shared services . . . 10
  - 2.2.3 Service Communication and Integration . . . 10
    - 2.2.3.1 RESTful API . . . 11
    - 2.2.3.2 Remote Procedure Call . . . 11
    - 2.2.3.3 RabbitMQ . . . 11
  - 2.2.4 KVM . . . 12
- 2.3 OpenStack Networking: Neutron . . . 12
  - 2.3.1 The Neutron Project . . . 12
  - 2.3.2 Architecture Overview . . . 13
    - 2.3.2.1 Neutron Server . . . 13

2.3.2.2 Plug-in Architecture . . . 14
2.3.2.3 Message queue . . . 16
2.3.2.4 L2 Agent . . . 17
2.3.2.5 L3 Agent . . . 18
2.3.2.6 DHCP Agent . . . 19
2.3.2.7 Virtual bridges . . . 19
2.3.3 Network Architecture . . . 20
2.3.4 Neutron Network Traffic Types . . . 20
2.3.5 Traffic Flow In Neutron Networking . . . 21
2.3.5.1 Neutron network with legacy routing . . . 22
2.3.5.2 Network with distributed routing . . . 23
2.4 Open vSwitch . . . 24
2.5 Network Tunnelling With VXLAN . . . 24
2.5.1 VXLAN Overview . . . 25
2.5.2 VXLAN Tunnel End Point (VTEP) . . . 26
2.5.3 VXLAN Tunnel In OpenStack . . . 26

3 Methodology **27**
3.1 Network Performance Measurements And Evaluations . . . 27
3.1.1 Overview . . . 27
3.1.2 Throughput Evaluation . . . 28
3.1.3 Latency Evaluation . . . 28
3.1.4 CPU Statistic And Profiling . . . 29
3.2 Experimental Scenarios Based On Traffic Flow . . . 29
3.2.1 Network Architecture With Legacy Router . . . 30
3.2.1.1 Intra-node Intra-subnet . . . 30
3.2.1.2 Intra-node Inter-subnet . . . 31
3.2.1.3 Inter-node Intra-subnet . . . 32
3.2.1.4 Inter-node Inter-subnet . . . 32
3.2.2 Network with DVR router . . . 32
3.2.2.1 Intra-node Inter-subnet (with DVR router) . . . 33
3.2.2.2 Inter-node Inter-subnet (with DVR router) . . . 34

4 Experimental Setup **35**
4.1 Overview . . . 35
4.2 Single-host Installation . . . 36
4.3 Amazon AWS Testbed . . . 36
4.4 Local Cluster Test-bed . . . 38
4.5 Other Configurations . . . 39
4.5.1 KVM And VHostNet . . . 40
4.5.2 MTU Size . . . 41
4.5.3 Security Group . . . 41
4.6 Logical Network Architecture . . . 41

5 Experiment Result **44**
5.1 Amazon AWS . . . 44
5.1.1 Thoroughput Measurement . . . 44

5.1.2 Latency measurement . . . 45
5.1.3 Conclusion . . . 45
5.2 Local Test-bed . . . 46
5.2.1 Tenant Network . . . 46
5.2.1.1 Throughput measurement . . . 47
5.2.1.2 Latency measurement . . . 48
5.2.1.3 Performance comparison: DVR versus non-DVR . . . 50
5.2.1.4 Performance comparision: KVM/VHostNet . . . 51
5.2.2 External Network . . . 51
5.2.2.1 Throughput measurement . . . 52
5.2.2.2 Latency measurement . . . 52
5.2.3 Observation . . . 53

6 Further System Analysis **55**
6.1 CPU Usage . . . 55
6.2 CPU Profiling . . . 56
6.2.1 Receiving Data . . . 57
6.2.2 Sending Data . . . 58
6.2.3 Discussion . . . 58

7 Conclusion **60**
7.1 Future Work . . . 61
7.2 A Final Thought . . . 61

A Installation and Configuration **63**
A.1 Vagrant script for OpenStack installation on AWS . . . 63
A.2 OpenStack Module Configurations . . . 65
A.3 Miscellaneous scripts . . . 70

Bibliography **72**

List of Figures

2.1 OpenStack conceptual architecture . . . 7
2.2 OpenStack Dashboard: Network Topology . . . 9
2.3 Neutron architecture overview . . . 13
2.4 The Modular Layer 2 plug-in . . . 15
2.5 Neutron ML2 Plugin start-up diagram . . . 15
2.6 Neutron-server communication with OVS agents using RPC . . . 17
2.7 OpenStack Neutron L3 Agents [1] . . . 18
2.8 OpenStack Network with centralised (legacy) router . . . 22
2.9 OpenStack Network with DVR router . . . 23
2.10 VXLAN packet header . . . 25

3.1 Intra-node Intra-subnet VM traffic flow (HNR) . . . 31
3.2 Intra-node Inter-subnet VM traffic flow (H-NR) . . . 31
3.3 Inter-node intra-subnet VM traffic flow (-HNR) . . . 32
3.4 Inter-node inter-subnet VM traffic flow (-H-NR) . . . 33
3.5 Inter-node inter-subnet VM traffic flow (with DVR router) . . . 33
3.6 Inter-node inter-subnet VM traffic flow (with DVR router) . . . 34

4.1 Basic component of OpenStack . . . 35
4.2 Amazon AWS Network architecture . . . 37
4.3 Test-bed Network architecture . . . 38
4.4 KVM and VHostNet . . . 40
4.5 Logical network . . . 42

5.1 AWS throughput measurement . . . 44
5.2 AWS latency measurement . . . 45
5.3 Network throughput with legacy router (TCP) . . . 47
5.4 Network throughput with distributed router (TCP) . . . 47
5.5 Network throughput with DVR router (UDP) . . . 48
5.6 Network latency with legacy router . . . 49
5.7 Network Latency with distributed router . . . 49
5.8 Comparison: Network throughput with and without DVR . . . 50
5.9 Comparison: Network Latency with and without DVR . . . 50
5.10 Comparison: Network throughput with and without VHostNet/KVM . . . 51
5.11 Comparison: Network latency with and without VHostNet/KVM . . . 51
5.12 Network throughput with distributed router (inter-node traffic) . . . 52
5.13 Network latency with distributed router . . . 53

6.1 CPU usage of Compute node for data sending of host and guest machine 55

6.2 CPU usage of Compute node for data receving of host and guest machine 56

List of Tables

2.1 Scenarios under which VM traffic needs to pass through Network node . . 21

4.1 AWS Configuration 36
4.2 Test-bed server configurations 39
4.3 Local testbed VM Configuration 39
4.4 VM distribution to virtual networks and Compute nodes 42
4.5 VM IP addresses 43

Abbreviations

API	**A**pplication **P**rogramming **I**nterface
CC	**C**loud **C**omputing
CLI	**C**ommand-**L**ine **I**nterface
OVS	**O**pen**VS**witch
DVR	**D**istributed **V**irtual **R**outing
VM	**V**irtual **M**achine
VN	**V**irtual **N**etwork
VNI	**V**irtual **N**etworking
VXLAN	**V**irtual E**X**tensible **L**ocal **A**rea
NAT	**N**etwork **A**address **T**ranslation
VTEP	**V**XLAN **T**unnel **E**nd **P**oint
AMQP	**A**dvanced **M**essage **Q**ueue **P**rotocol
PM	**P**hysical **M**achine

Chapter 1

Introduction

During the last few years, *Cloud computing* has become one of the hottest trends in the ICT domain that attracts lots of attention from researchers in both industrial and academic fields. The computing model is considered to evolve the way in which different technologies collaborate to change organisation's approach to build and manage their IT infrastructure as well as computing services. Similar to other scientific and technological advancement, especially those evolving from existing technologies, the value of Cloud Computing does not lie in the technology itself, but rather in the operational changes upon its deployment and application.

1.1 Motivation

So far there have been several platforms that attempt to popularise cloud environment to the market. These include both software solutions like *Apache CloudStack* or *OpenStack*, and service providers like Amazon (EC2) or Google (Google Cloud Platform). In meeting with the fast-growing demand for cloud computing, OpenStack has come up as a software stack that can help to quickly deploy a cloud cluster at low cost and overheads.

In order to provide the services with best Quality-of-Service (QoS) possible, guaranteeing high performance of the operating cloud cluster is always among the main concerns of cloud service providers. As such, achieving a thorough comprehension and detailed analysis of OpenStack architecture, its core technologies and operation is a necessary step prior to any enhancement made to improve the overall OpenStack performance. This

thesis work aims to achieve a thorough understanding of the OpenStack architecture, especially its Netwoking module, and to study the network performance of a OpenStack-based cloud cluster.

1.2 Problem Statement

In the academic literature, to the best of author's knowledge, there has been no recent publication that deals with an in-depth performance analysis of OpenStack with a special interest given to its networking module - Neutron. Under OpenStack's renown 6-month release cycle, Neutron has been under massive redesign and improvement lately. In the 10th OpenStack release (codename *Juno*), Neutron comes up with, among other new features, the implementation of Distributed Virtual Routing (DVR) that primarily aims at enhancing network performance and mitigating the single-point-of-failure issue as in the architecture with centralised router. OpenStack Juno is expected to be more efficient than previous releases, both overall and network performance. In a nutshell, this thesis comes up with the following goals:

1. *Deployment of OpenStack cloud software into different environments consisting of Amazon AWS and local test-bed*
2. *A methodological approach to analyse OpenStack network performance with different operational settings, including Distributed Virtual Routing (DVR)*
3. *In-depth system analysis, based on the CPU profiling, to explain the impact of Neutron on OpenStack network performance*

In general, we are able to run and experience a fully functional OpenStack cluster and evaluate the Neutron's impacts on the underlying operating system and hardware infrastructure, which accordingly affects the network performance of OpenStack. We also see how different settings and environment influence behaviours of the designated VMs. As the experiment goes on, it can be observed that network performance is considerably impacted by the encapsulation mechanism that is used to isolate tenant networks. This results in bandwidth utilisation of less than 30% of the measured VM traffic and heavily consumed CPU usage of the host machines.

The work is divided into 7 chapters: The introductory one briefs the problem statements and motivations behind the work. Chapter 2 provides background of OpenStack and related technologies which construct the software stack. Chapter 3 outlines the methodologies with which experiments are conducted. Chapter 4 describes the installation processes carried out with different platforms including the Amazon Web Service (AWS) EC2 and local cluster test-bed . Chapter 5 presents the results collected from the various tests and chapter 6 gives an in-depth analysis based on system profiling, which helps to explain result from chapter 5. The last section concludes the work and suggests some ideas as potential future work.

The thesis only mentions the most critical configurations and scripts. A full resources related to the deployment, evaluation scripts and experiment results of the work can be found at https://github.com/tab87vn/sinf_master_thesis.git

Chapter 2

Background

2.1 Cloud Computing

2.1.1 Definition

Cloud computing (CC) has been in the market for a while and is praised by consumers and enterprises for its provision of on-demand access to scalable computing resources, to meet the need services and applications with growing complex. Yet there has not been any standardised definition of what Cloud computing is and as a matter of fact, different companies and institutions tend to have their own definitions for this new technology/business model [2–4].

Among many ways in the literature of how the term *Cloud Computing* is defined, the one in the published work of the U.S. National Institute of Standards and Technology (NIST)[4] has been taken as the *de-facto* definition:

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.".

CC introduces a new way of optimally utilising and computing power (CPU, memory, storage), in which cloud resources are not only shared among multiple users, but also able to be dynamically supplied (on demand). Provisioned to users on a pay-for-use

basis, CC offers an attractive environment for users and enterprises to develop and/or run Internet-based applications and services, with little concerns over upfront costs as well as infrastructure maintenance costs.

2.1.2 Cloud Computing models

The ICT industry has defined the three main forms of Cloud Computing including Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS). These three models are commonly referred to as SPI model.

SaaS

SaaS, short for *Software-as-a-Service*, is the most basic form of cloud service where users are able to run their favourite applications and services (such as emails, office, or even video games and so on) in the cloud. As such, users use the resources effectively regardless of constraints on IT implementation problems. This model also helps to minimise upfront cost in operation as well as maintenance. Typical examples of SaaS are Google apps, Salesforce, Cisco WebEx, and so on.

PaaS

PaaS, short for *Platform-as-a-Service*, provides a development platform (i.e. development kits and a number of supported programming languages, database or other software components) on which cloud users can leverage to develop, manage and run their own applications and services. With PaaS, cloud users are given more control over the environments for managing applications. Typical examples of PaaS include Windows Azure, Heroku, Google App Engine, and so on.

IaaS

IaaS, short for *Infrastructure-as-a-Service*, provides cloud users with physical resources or virtual machines in terms of CPU, storage, load balancers or operating system. Some IaaS service providers also provide disk image library and file-based storage.
Typical examples of IaaS include Amazon EC2, Google Compute Engine, and so on.

XaaS

SaaS, PaaS and IaaS as listed above are the most common forms of XaaS with the provisioned resources being referred to software, platform and infrastructure, respectively.

Everything-as-a-Service (also known as *Anything-as-a-Service*), or XaaS, refers to the growing diversity of services provided over the Internet rather than locally or on-premise.

2.2 OpenStack

This section gives a brief overview of the OpenStack project, its core components and some of the key enabling technologies.

2.2.1 The OpenStack Project

OpenStack[5] is a free and open-source cloud computing software platform that enables rapid deployment, management and development of a cloud infrastructure in a data centre. OpenStack was jointly launched by NASA[6] and Rackspace Hosting[7] in July 2010 and is managed by the OpenStack Foundation. OpeStack Foundation is a non-profit organisation formed in September 2012 to promote the development, distribution and adoption of the software stack. Currently, the OpenStack project is supported by more than 500 companies.

OpenStack platform provides cloud computing services running on standard commodity hardware and is primarily deployed as an Infrastructure-as-a-Service (IaaS) model. The software stack consists of a group of interrelated projects that control pools of processing (*Nova*), storage (*Swift*, *Cinder*) and networking (*Neutron*) resources throughout a data centre. Management and control over these pools are exposed to users through a web-based dashboard (*Horizon*), command-line tools, or a RESTful API. By utilising a massive collection of popular enterprises and open-source technologies, OpenStack becomes an ideal solution for heterogeneous infrastructure.

The OpenStack project currently has a 6-month release cycle. Up to the point of writing this thesis, there have been 11 stable releases, among which the latest one (code name *Kilo*), was released in April 2015 while the soon-to-be-released version (code name *Liberty*) is planned to come out on 15 Oct 2015 [8]. Having been the most recent stable release and had a reasonable time-on-market when this thesis is started, *Juno* is considered the most suitable for the deployment and experiment purposes of the thesis.

2.2.2 OpenStack Software Components

The OpenStack project consists of several interrelated sub-projects that help to manage different aspects of hardware resources including computing, storage, networking and other related services, each of which offers its own set of APIs to facilitate the integration of the whole software stack. Figure 2.1 illustrates the OpenStack conceptual architecture with interactions among its software components[9]. As an IaaS-focused cloud platform, OpenStack has VMs at its centre, provisioned by the *Nova* module. VMs are surrounded by other services including network connectivity handled by *Neutron*; operating system images stored by *Glance*; storage services provided by *Swift* and *Cinder*. *Keystone* is responsible for the authentication of the whole OpenStack system while, at a high level, *Horizon* provides a web-based management interface to all the other services.

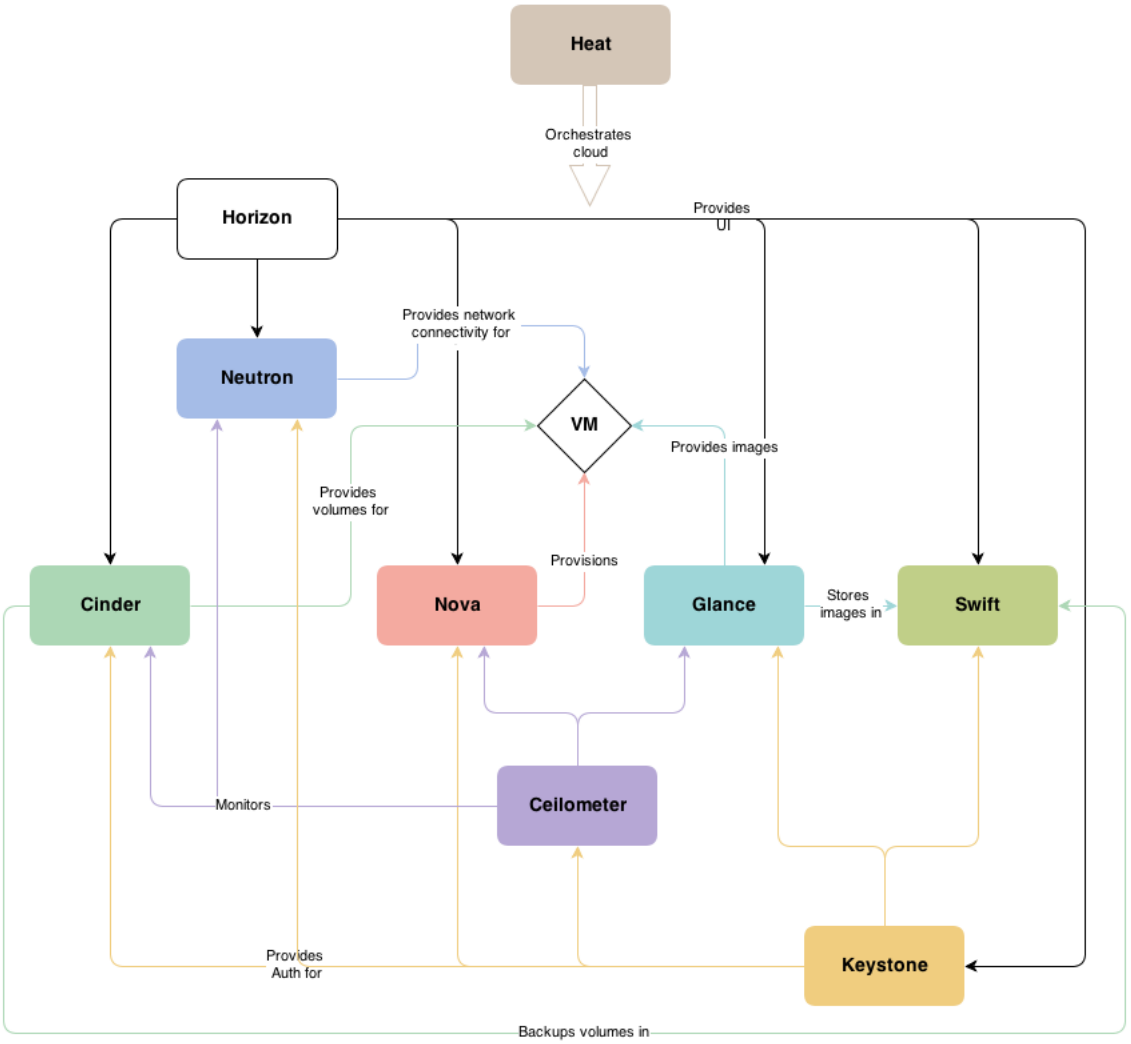

FIGURE 2.1: OpenStack conceptual architecture

2.2.2.1 Compute service

OpenStack Compute (*Nova*) is designed to manage pools of computing resources and provide access to these pools via either graphical user interface tools (dashboard) or command-line tools or the rich native API sets. Nova works with most popular virtualisation technologies such as KVM (default) [10], VMware[11], Xen[12] or Hyper-V[13] as well as Linux Container technologies like LXC[14].

Nova can be considered the main part of an IaaS system, in which cloud users have access to VMs hosted by nodes running Nova service. Within OpenStack platform, Compute nodes can be added and integrated with the existing nodes, making the resource pool horizontally scalable on standard hardware.

2.2.2.2 Storage service

Besides traditional storage technology (that comes along with computing resources managed by Nova), OpenStack also supports two additional types of storage, namely Object Storage and Block Storage.

Object storage (Swift) is a scalable redundant storage system in which objects and files are stored, replicated, and distributed throughout multiple servers in the cluster. As an example, Amazon runs its storage service *S3* via its public cloud platform at massive scale.

Block storage (cinder) manages (creates/attaches/detaches) virtualised block storage pools and provide OpenStack users with access to these pools. Block storage is fully integrated into compute (Nova) and dashboard (Horizon) services via APIs, enabling users to consume these storage resources even without any knowledge of the technology of the underlying storage devices.

2.2.2.3 Networking service

The Networking service (*Neutron*, formerly *Quantum*) provides an abstraction of Virtual Network Infrastructure (e.g: network, subnets, ports, routers, etc.) and services (e.g.: firewall, load balancer, virtual private network, etc.) within OpenStack-based cluster.

Neutron essentially provides VMs (created and managed by Nova) with networking service which, prior to the existence of Neutron, used to be handled by `nova-network`. Being capable of providing only basic networking service, development and deployment of `nova-network` became gradually lessened in later releases, with a long-term plan to remove this module from OpenStack code base [15, p. 317]. While `nova-network` still was not deprecated in Juno release, Neutron has stepped up as the default networking module for OpenStack with more flexible and full-fledged abstractions of network infrastructure and services. Section 2.3 gives a more insightful discussion on Neutron network module.

2.2.2.4 Dashboard service

OpenStack dashboard (Horizon) enables users to access and manage VMs, VNs and other OpenStack resources via a web-based graphical users interface (written in Python using Django). Figure 2.2 exemplifies the Dashboard service, presenting different virtual machines, the virtual networks they belong to and virtual routers that connect these networks. Besides dashboard, users (particularly developers) can also interact and perform administrative tasks by using sets of native OpenStack APIs or the EC2 compatibility APIs.

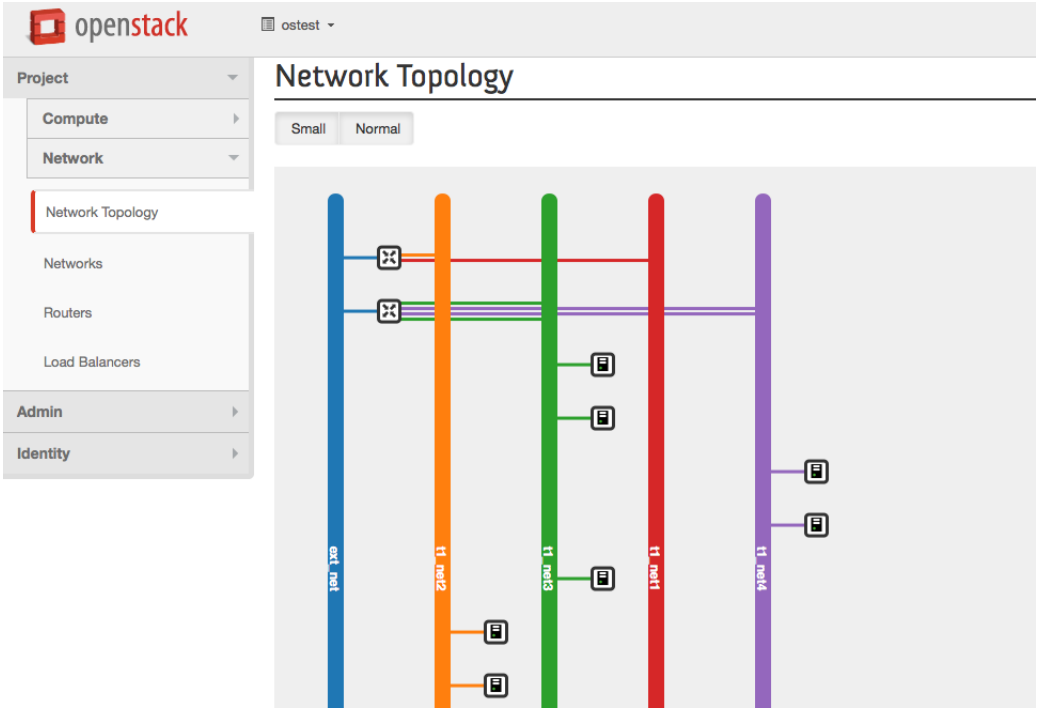

FIGURE 2.2: OpenStack Dashboard: Network Topology

2.2.2.5 Shared services

OpenStack has several other services that are commonly used by the above core projects, making it easier to implement and operate your cloud. These services — including identity, image management and a web interface - integrate the OpenStack components with each other as well as external systems to provide a unified experience for users as they interact with different cloud resources.

Identity service

OpenStack Identity service (*Keystone*) provides a central authentication and authorisation mechanism for other OpenStack services. Keystone also provides a catalog of endpoints for all OpenStack services.

Image service

The OpenStack Image service (*Glance*) enables creating, storing and retrieving disk images for VMs, which is used by Compute service during the provisioning of VM instances.

Telemetry service

The OpenStack Telemetry service (*Ceilometer*) can measure and track the services used by OpenStack users and provide billing accordingly.

Orchestration service

OpenStack Orchestration (*Heat*) provides the ability to define and automate the deployment of infrastructure, services and applications using flexible templates. It can scale up or scale down the OpenStack cluster.

2.2.3 Service Communication and Integration

OpenStack is a fully distributed system, consisting of multiple smaller projects, or modules, as discussed in the previous section. Each module is designed with the "Share Nothing Architecture" principle in mind, and is functionally independent from the others. A module like Nova or Neutron is comprised of multiple components that together bring on its functionalities. As with any other distributed systems, the functionality of OpenStack as a whole depends heavily on how its inner services are integrated, which in turn relies on the capability of its modules and components to communicate between

them. There are 3 main mechanisms that enable the communication and service integration of OpenStack: RESTful API, Remote Procedure Call and RabbitMQ.

2.2.3.1 RESTful API

REST (*REpresentational State Transfer*) is an architectural style, and an approach to communications that is often used in the development of Web services. Each of the OpenStack core modules exposes one or more RESTful interfaces to interact with the outside world. By using RESTful APIs, OpenStack provides access to users in different ways, either by Command-Line Interface (CLI), cURL or via REST client. Each of these major projects has an API service as endpoint for client to access (e.g. `openstack-nova-api`, `openstack-glance-api`) so it can accept REST request from its clients, either users or other modules. As such, RESTful API is an effective way to let different OpenStack modules communicate.

2.2.3.2 Remote Procedure Call

Remote Procedure Call, or RPC, enables inter-process communication that allows its clients to trigger the execution of subroutines in a remote location. OpenStack modules like Nova (`nova-compute`, `nova-api`, `nova-scheduler`), Neutron (`neutron-server`, `neutron-openvswith-agent`) or Cinder (`cinder-scheduler`, `cinder-volume`) make heavy use of RPC for its intra-module communication, to the extent that almost everything happening in these modules is triggered by RPC calls. For example, after Neutron's `neutron-server` receives a (RESTful) request to create a new network, it asks the available plug-in (e.g `ml2plugin`) to in turn send an RPC call to the corresponding agent (e.g. `neutron-openvswith-agent`).

2.2.3.3 RabbitMQ

RPC calls rely on a channel, or a messaging mechanism through which they are delivered to the consuming processes (i.e. consumers). RPC requests are packaged into messages that are sent to a message broker which then forwards them to the consumers. This is where a messaging broker like RabbitMQ fits into the picture. RabbitMQ [16] is an open-source implementation of the *Advanced Message Queue Protocol* (AMQP)[17]

standard. AMQP is designed to facilitate the brokering of messages between different processes, applications of the same system, or even between systems that communicate by message passing. In OpenStack platform, AMQP is utilised to establish an efficient internal communication mechanism between components of the same OpenStack module, for instance, Nova, Neutron or Cinder.

2.2.4 KVM

Kernel-based Virtual Machine, or KVM, is a full virtualisation solution for Linux and has been shipped with Linux kernel since kernel version 2.6.20. KVM enabled by running QEMU-based hardware emulation with KVM-acceleration mode enabled. KVM is a special operating mode of QEMU that utilises processor's hardware-assisted virtualisation capability (Hardware Virtual Machine, or HVM) to perform hardware virtualisation via its processor-specific kernel modules. KVM is among several hypervisor platforms compatible with OpenStack.

2.3 OpenStack Networking: Neutron

This section discusses in detail the OpenStack Networking module - *Neutron* - and the relevant software packages that altogether construct the Neutron-based network infrastructure.

2.3.1 The Neutron Project

The OpenStack Neutron project, having its premiere in Havana release (October 2013), replaces *nova-network* to provide OpenStack with a full-featured abstractions of the Virtual Network Infrastructure as well as basic and advanced network services. Thanks to Neutron, cloud users have access to essential networking infrastructure and resources like network, subnet and router objects. The elements simulate functionalities of real-world corresponding physical components: network consists of subnets connected to routers, which route traffic between different subnets and networks. Besides the provision of such basic network services as NAT, DHCP or routing, Neutron also enables users to

create advanced virtual network topologies including services such as firewalls (Firewall-as-a-Service, or FWaaS), load balancers (LoadBalancer-as-a-Service, or LBaaS), and virtual private networks (VPN-as-a-Service, or VPNaaS).

2.3.2 Architecture Overview

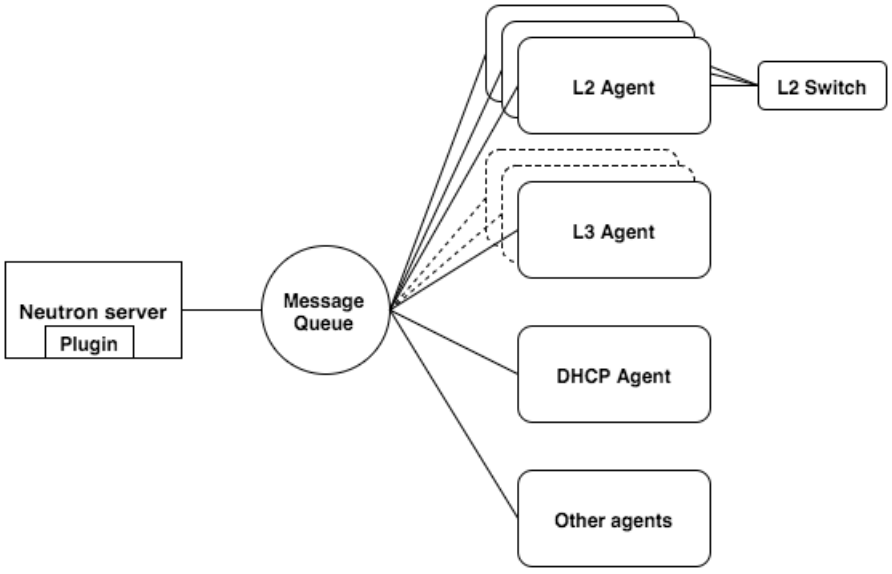

FIGURE 2.3: Neutron architecture overview

Neutron Networking is a standalone component in the whole OpenStack architecture, working closely with other components (Identity, Compute, Storage, etc.) to provide networking connectivity and services to VMs. At a high level, the Neutron Networking consists of a central server daemon which exposes, receives and dispatches API requests. Neutron clients uses these APIs for building flexible policies and sophisticated networking topologies. The Neutron server administers several agents responsible for host and network configuration. Communication between Neutron server and these agents relies on RPC (over RabbitMQ) or through the standard Networking API. This section provides an overview on the architecture and main components of Neutron, as illustrated in Figure 2.3.

2.3.2.1 Neutron Server

The Neutron server daemon (`neutron-server`) starts up, reads the configuration files then loads all configured plug-ins and extensions. It also exposes APIs to Neutron clients

(via Dashboard, CLI or API calls), and forwards requests from the clients to configured plug-ins. In particular, these client requests are placed into a message queue (using the RabbitMQ messaging system) and dispatched to corresponding agents (L2, L3, DHCP or other agents for advanced services).

2.3.2.2 Plug-in Architecture

Neutron's ability to integrate with different underlying infrastructure and other networking services is implemented by a variety of plug-ins. In other words, while Neutron server provides its users with sets of resourceful APIs to manage and customise networks, it is the plug-ins that do the actual configuration tasks and enable Neutron to support fast-changing network technologies from various vendors as well as to efficiently deploy the *Software-Defined Networking* paradigm. With plug-ins, advanced networking capabilities such as *L2-in-L3* tunnelling, load balancing, virtual private networks or firewalls can be supported and plugged into Neutron stack. As such, the plug-in architecture brings a powerful and flexible way of customising a network's capabilities. There is only one Neutron plug-in running at a time and it is configured as followed:

```
core_plug-in = neutron.plug-ins.ml2.plug-in.Ml2Plugin
```

Modular Layer 2

Modular Layer 2 (ML2) plug-in provides a framework to simultaneously manage a variety of Layer-2 technologies, each with an individual mechanism driver. ML2 helps to address the problem of implementation redundancy in which different switching technologies and vendors (e.g. OpenvSwitch, Linux bridge or Cisco) bring in their own monolithic plug-ins and associated agents while still providing similar features and getting plugged into the same environment (Neutron stack). This results in duplication of database, code base, and so on, along with development and maintenance efforts. ML2 is intended to eliminate these duplication issues and to simplify the scalability potential (e.g. development of new plug-in/agent for new switch vendor). Besides, with ML2, there is still only one single plug-in allowed but multiple switching technologies can be run simultaneously thanks to ML2's mechanism driver. Figure 2.4 provides a high-level view of the ML2 framework in association with the Neutron server.

The ML2 plug-in essentially consists of Type Manager and Mechanism Manager:

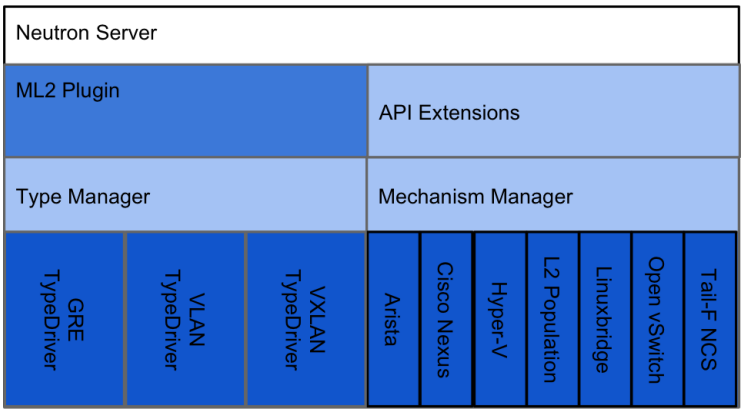

FIGURE 2.4: The Modular Layer 2 plug-in

- ***Type manager:*** A type driver manages network state of a specific type and also performs provider and tenant network validation. OpenStack Juno's supported type drivers includes: `local`, `flat`, `vlan`, `gre` and `vxlan`. Within the scope of this thesis, VXLAN will be discussed in detail as the overlay technology for management of tenant networks.

- ***Mechanism manager:*** The mechanism manager manages drivers for different underlying technologies (from different vendors) used to manipulate the underlying infrastructure and makes sure that they are applied in accordance with available type drivers.

ML2 startup procedure

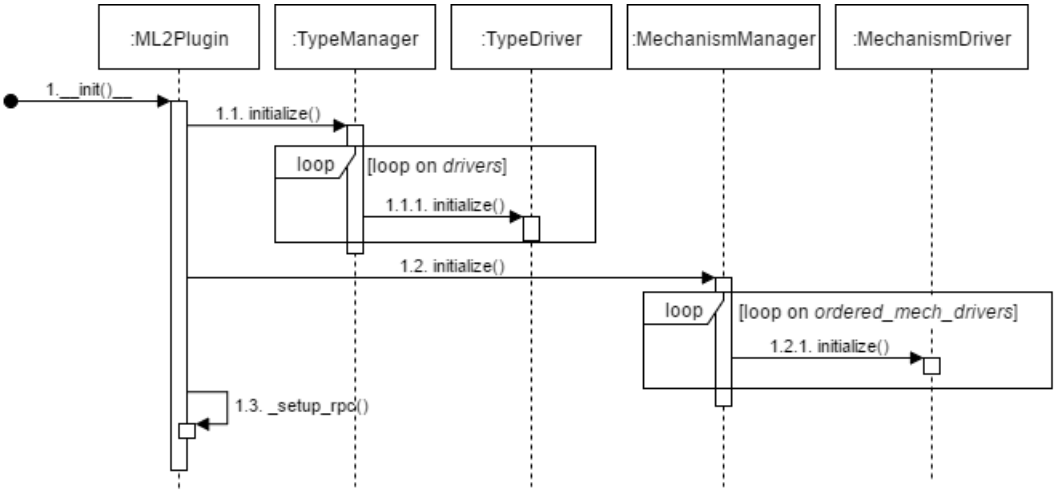

FIGURE 2.5: Neutron ML2 Plugin start-up diagram

The sequence diagram from Figure 2.5 describes how the ML2 plug-in starts and enables the available type drivers and mechanism drivers. The start-up procedure consists of the following steps:

1. When Neutron server starts, it loads its only plug-in, which in this case is the ML2
2. When loaded, ML2 requests its type manager (`TypeManager`) class to read its configuration file(s) (`ml2.ini`) and load all supported network types specified
3. For each of the loaded network types, `TypeManager` creates an instance of driver (`TypeDriver`) class to handle that specific type. In doing so, there can be different types of network running simultaneously
4. Once all the drivers for all network types have been loaded and initialised, a request is sent to the mechanism manager (`MechanismManager`) class to load all supported networking mechanisms specified in the configuration file (`ml2.ini`)
5. For each of the configured mechanisms, the `MechanismManager` creates a driver (`MechanismDriver`) instance to handle. This enables multiple networking mechanisms for underlying technologies from different vendors (i.e. Open vSwitch, Cisco, etc.) to run simultaneously
6. After all configurations have been loaded and necessary handling processes are created, the ML2Plugin sets up and RPC requests to communicate with the agents for according host configuration

2.3.2.3 Message queue

Similar to other OpenStack modules, Neutron uses RabbitMQ as a messaging broker for communication between its internal components by exchanging *Remote Procedure Call* (RPC) over RabbitMQ message queueing mechanism. Figure 2.6 illustrates the communication between Neutron components including `neutron-server`, the OVS agent and the OVS (already exemplified in section 2.2.3.2).

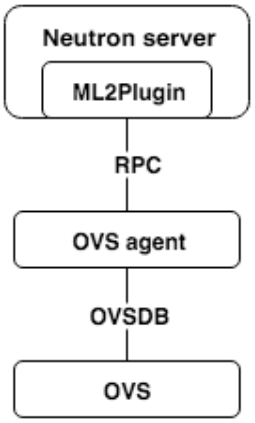

FIGURE 2.6: Neutron-server communication with OVS agents using RPC

2.3.2.4 L2 Agent

L2 agents run on hypervisors (Compute nodes) and communicate with Neutron server using RPC. An L2 agent is responsible for monitoring its hosting hypervisor and informing the `neutron-server` of events occurring with the new or removed devices

In OpenStack, L2 connectivity can be provided using various mechanism drivers. L2 agents need to be present in all compute/network nodes to make sure that L3 services are reachable by tenant VMs and subnets.

Open vSwitch Agent

As discussed above, Neutron requires plug-in agents (`neutron-openvswitch-agent` in this case) to be present in all hypervisor and networking nodes to provide local Open-vSwitch configuration.

An OVS agent receives requests from `neutron-server` and acts accordingly to configure OVS. This mainly involves setting up the integration bridge (`br-int`), to which all internal network services and tenant VMs are attached. The `neutron-openvswitch-agent` particularly relies on an OVS-specific API (`ovs_lib`) to configure OVS and manipulate flow entries via two utilities `ovs-vsctl` and `ovs-ofctl`, respectively. Despite being an OpenFlow-compatible switch, OVS operates within Neutron networking as a regular L2 switch with both normal and flow modes.

2.3.2.5 L3 Agent

Figure 2.7 captures an overview on L3 services in collaboration with virtual components configured by L2 agents. The L3 agent creates different internal ports with prefix `tap` for DNS service, `qr-` for virtual router or prefix `qg-` for gateway to the External (public) network.

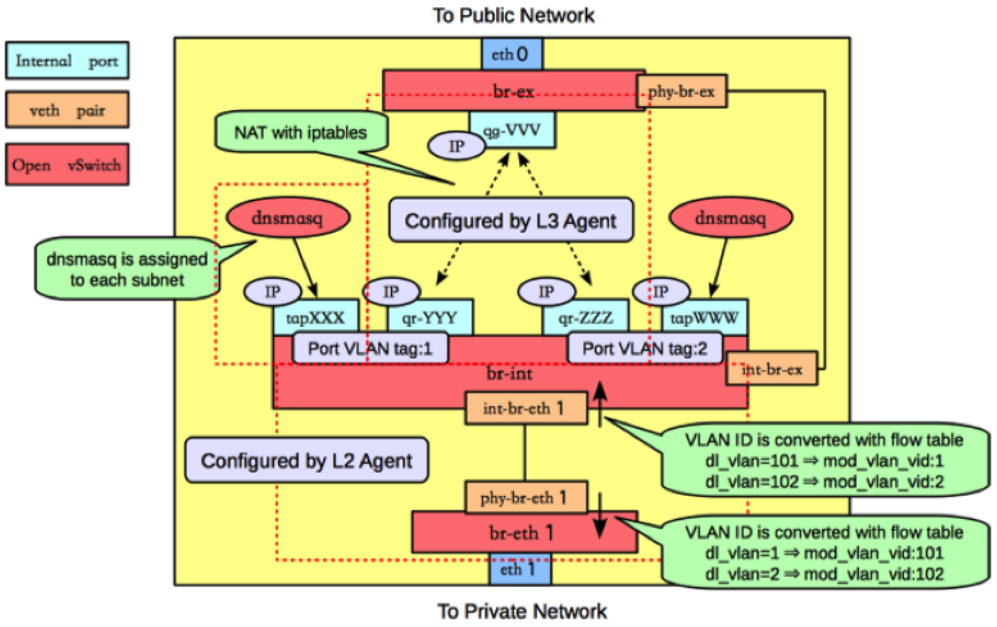

FIGURE 2.7: OpenStack Neutron L3 Agents [1]

As its name suggests, the Neutron L3 agent (`neutron-l3-agent`) configures its hosting node with different Layer-3 networking services including *routing*, *NAT*, and *Floating IP*. Traditionally, such L3 services run on Network node and rely on the L2 agent to provide layer-2 connectivity to VM instances running on Compute nodes. Neutron L3 agents use the Linux IP stack and `iptables`[18] to perform L3 forwarding and NAT. In order to support multiple routers with potentially overlapping IP addresses, `neutron-l3-agent` defaults to using Linux network namespaces to provide isolated forwarding contexts.

Routing

Similar to conventional networking, routing is required if packets are sent from one subnet to another, including traffic between VMs belonging to different subnets, or between VMs and hosts reachable via the External network. Assume that there are multiple tenant networks and one shared external network and each of the former is connected to the latter via a Neutron-based virtual router. Such a router will be:

- Connected to the internal (tenant) network via *"qg-"* (gateway) interface on `br-ex`
- Connected to the external network via port via *"qr-"* (router) interface on `br-int` integration bridge
- Having a namespace (*"qrouter-"* prefix) associated with router name to avoid IP conflicting between networks

Section 2.3.5 will discuss Neutron routing in more detail, with particular attention given to the distributed virtual routing capability.

NAT

The `neutron-l3-agent` implements its router's NAT, or *Network Address Translation*, functionality using Linux kernel `iptables`, enabling packets from internal (tenant) networks to reach external network before going out to the Internet. Similar to routing, NAT rules of a router need to be executed under a specific router's namespace to isolate them from host's network and other tenants' networks.

Floating IP

Virtual router provides Floating IP by NAT and iptables. This L3 service allocates and associates IP addresses from external network to internal tenant VMs to make them directly reachable from external network. The neutron-l3-agent implements Floating IP association also by using `iptables` to perform NAT as described above.

2.3.2.6 DHCP Agent

Neutron relies on its DHCP Agent, `neutron-dhcp-agent`, located in the Network node to provide *Dynamic Host Configuration Protocol* (DHCP) services to tenant networks, thus allocating IP addresses to VMs. In particular, `dnsmasq` [19] is used as back-end service for this purpose. For each subnet created, there is a running `dnsmasq` daemon attached to the `int-br` via port with "`tap-`" prefix under a DHCP namespace.

2.3.2.7 Virtual bridges

A basic architecture of OpenStack Network requires the setup of the following virtual bridges:

- *br-int*: The *Integration bridge* connects VMs and other virtual devices/services (DHCP, Routing, etc.) with host machines
- *br-ex*: The *External bridge* acts as a gateway, mapping traffic between internal tenant VNs and the External physical network. `br-ex` is connected to virtual routers via ports with `qg-` prefix
- *br-tun*: The *Tunnelling bridge* (de-)encapsulates traffic sent (received) by VMs via the Tunnel network

2.3.3 Network Architecture

There are three types of network in a standard OpenStack configuration: Management, Tunnel (or Tenant) and External networks.

- ***Management***: Is used for administrative communications and OpenStack internal operations such as authentication, access to internal databases (on Controller node), configurations, and so on. When the cluster is being set up, all configurations that require multi-node connectivity use the Management network.
- ***Tunnel/Tenant***: Is reserved for communications of tenant networks, specifically traffic data exchanged between VM instances. This is where the network overlay standard like VXLAN is applied, to essentially isolate tenant's virtual networks and create a multi-tenancy network environment (i.e. multiple users can have their network run on a shared physical network). As such packets sent through this network is encapsulated. This is also referred to as private network.
- ***External***: Is essentially the gateway that allows traffic from VM instances to reach physical networks; As such, VM traffic must go through the node or nodes that have routing capability. This is also referred to as public network.

2.3.4 Neutron Network Traffic Types

With regard to the flow pattern of network traffic in Neutron networking, there are 4 different types: intra-subnet, inter-subnet, SNAT and DNAT.

- ***Intra-subnet***: Intra-subnet type refers to traffic between VM instances belonging to the same subnet of a tenant network. As each subnet is an L2 network segment configured by OVS, traffic within subnet does not need any routing mechanism. It is instead handled by OVS and forwarded with the MAC learning capability.

- ***Inter-subnet***: Inter-subnet type refers to traffic between VM instances belonging to different networks/subnets of the same tenant given that all subnets are connected to the same router. In this case, routing mechanism is required, thus traffic need to pass through the Network node if DVR is not enabled.

- ***SNAT***: SNAT, or *Source Network Address Translation*, type refers to traffic originating from VMs, reaching external network via a centralised router. This traffic type requires sent packets to go through Network node where the virtual router in SNAT mode is placed.

- ***DNAT***: DNAT, or *Destination Network Address Transation*, type, refers to traffic originating from VMs and reaching external network via a distributed router. This routing mechanism allows packets to be sent directly from the Compute node to the outside world with a dedicated floating IP associated to each VM. This traffic type is only available if DVR is enabled (since OpenStack Juno release).

Table 2.1 summarises different scenarios when traffic between VMs needs to pass through the Network node.

	DVR router	Legacy router
Intra-subnet	No	No
Inter-subnet	No	Yes
DNAT	No	Yes
SNAT	Yes	Yes

TABLE 2.1: Scenarios under which VM traffic needs to pass through Network node

2.3.5 Traffic Flow In Neutron Networking

In a Neutron-based architecture, packets sent by VMs need to adhere the routing rules configured by Neutron agents and virtual devices. Traffic routing in OpenStack is implemented with two different of network architecture: legacy router and distributed router.

2.3.5.1 Neutron network with legacy routing

The architecture with legacy router places the Network node at its centre: All packet flows, as long as routing is required (i.e. inter-subnet, SNAT, DNAT traffic types), need to go through the Network node where there are specific Neutron agents handling routing and other Layer-3 services. This architecture potentially turns the Network node into a *single-point-of-failure*: Should Network node become unreachable, routing service will be unavailable for all VMs. Additionally, the fact that VM traffic always needs to visit Network node for routing information worsens the network performance.

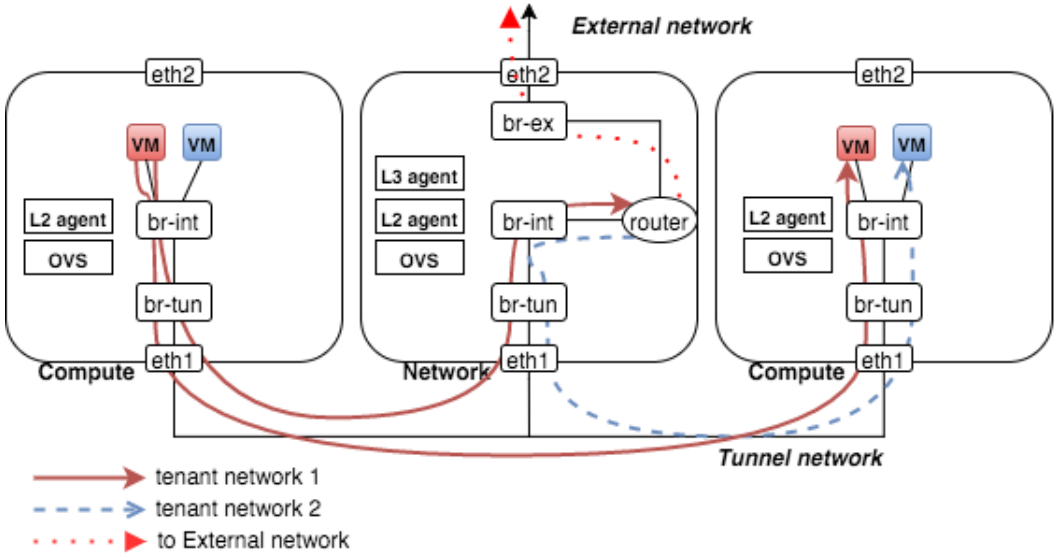

FIGURE 2.8: OpenStack Network with centralised (legacy) router

As can been seen in Figure 2.8, an L3 agent is installed on the Network node and configures a router. This virtual router is responsible for forwarding packets coming from and to the tenant networks with which the router is interfaced. L2 agents are present on the Network node and all Compute nodes where OVS daemons are running.

If the two VMs are in the same subnet then it is the job of L2 agent and OVS to handle the packet exchange between two VMs. Traffic leaving one VM simply enters the tunnel bridge of the sending host machine and arrives at the tunnel bridge of the receiving host before being delivered to the other VM. No routing is required here.

On the other hand, if the two VMs are from different subnets that are reachable to each other via a virtual router, traffic sent by a VM has to go through the router in order to arrive at the other VM. As packets leave the sending VM, they enter `br-int` and,

since receiving VM is not in the same Compute node, are passed to `br-tun` and start their journey in the tunnel network. Those packets are destined to the *virtual router* located inside the Network node before being forwarded either to the External network via `br-ex`, or to the `br-tun` bridge of the Compute node where the receiving VM reside.

2.3.5.2 Network with distributed routing

As one of the key features in OpenStack Juno release, DVR, or Distributed Virtual Routing, redesigns the network architecture by placing routers into Compute nodes in addition to the Network node. As such, the burden of routing is shared among all participating Compute and Network nodes, which helps not only to avoid a single point of failure as in legacy (centralised) architecture, but also to significantly improve network performance and scalability.

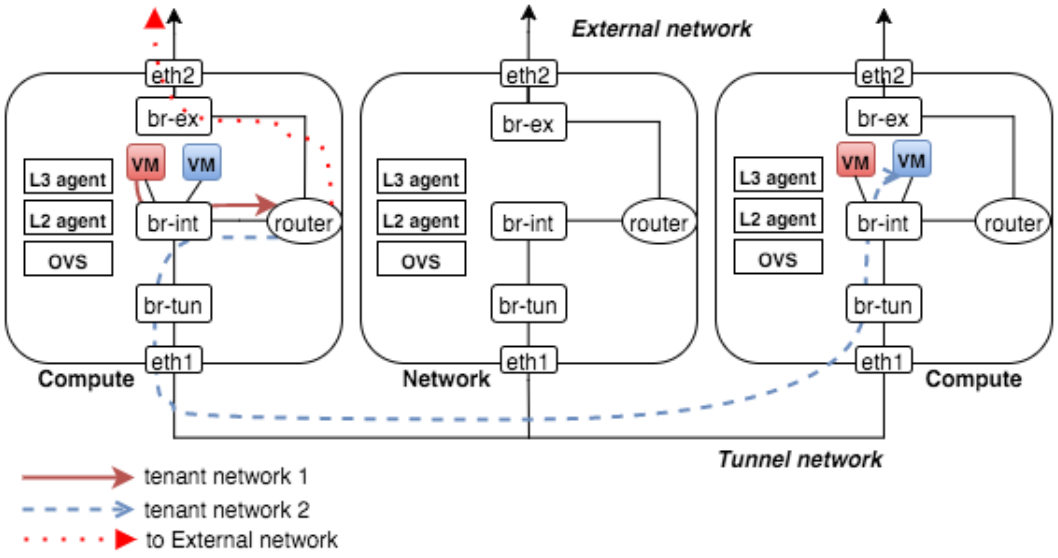

FIGURE 2.9: OpenStack Network with DVR router

What DVR brings to Neutron Networking, from an implementation-wise point of view, is that each Compute node has its own `neutron-l3-agent` running. This enables a Compute node to handle routing requests locally instead of offloading traffic over the central Network node as with legacy router.

In distributed mode, a virtual router makes its appearance in all participating Compute/Network nodes: every node has the same router identity and same interfaces as well as MAC address. Now that a virtual router is present in Compute nodes, it can

route traffic originating from local VM instances to either Tenant network (via the Tunnelling bridge `br-tun`) or External one (via the External bridge `br-ex`). It is of crucial importance that `br-ex` needs to be properly configured in Compute node for DVR to be operable.

2.4 Open vSwitch

Open vSwitch (OVS)[20] is a production-quality open-source implementation of a distributed virtual multi-layer switch. OVS relies on virtual network bridges and flow rules to forward packets between hosts.

OVS has three main components including the server module (`ovsdb-server`), the main daemon (`ovs-vswitchd`) and a kernel module. OVS provides high-level interfaces including, among the others, `ovs-vsctl` and `ovs-vsctl` that allow OpenStack to configure and use it as the underlying L2 switch. Specifically, Neutron's OVS agents (`neutron-ovs-agent`) store switch-level configurations into the OVS server module. The `ovs-vswitchd` daemon will read these configurations and set up virtual networks accordingly.

The kernel module of OVS, also known as "fast-path", plays an important role in packet forwarding. While the forwarding decisions are mostly made by the OVS daemon, it is the kernel modules that handle the majority of the traffic. When OVS experiences a new traffic flow, the first packet of the flow, considered a cache miss, is handled in user space by the OVS daemon. Subsequent packets of the flow are forwarded by kernel module for better performance. As user-space processing is much slower than kernel-space processing, we would normally experience some delay with the first few packets. The network performance evaluation is carried out with this in mind.

2.5 Network Tunnelling With VXLAN

OpenStack Neutron effectively creates and manages a multi-tenancy environment in which many cloud users share the same physical network infrastructure but their individual VNs and VMs data remain isolated from each other. There are different standards that facilitates virtual networks and tenant isolation, among which VLAN, GRE,

VXLAN are the examples. Throughout this study, the VXLAN protocol (RFC7348) [21] is chosen to be deployed in the tunnel network. This section gives a brief introduction on VXLAN and how it is integrated into Neutron infrastructure.

2.5.1 VXLAN Overview

Traditionally, data centres use VLANs to facilitate virtual networks, essentially isolating layer-2 traffic from physical network. As data centres may scale, the number of virtual networks increases, and the limit of 4096 VLANs (with some reserved ones) becomes insufficient. *Virtual eXtensible Local Area Network*, or VXLAN, is a network overlay technology that encapsulates MAC frame at layer 2 into a UDP header. In general, VXLAN allows users to create logical networks for VMs across different networks via packet encapsulation. In other words, it is able to create layer-2 network on top of a lay-3 connection.

Within a host machine, packets sent by VMs are tagged with the local VLAN ID. These tags are stripped off as traffic heads out of that host machine, when it is encapsulated with VXLAN header. Traffic between VM instances is sent through VXLAN-based tunnels set up between participating Compute/Network nodes. For different tenant networks, OpenvSwitch implements different tunnel ID header field in the VXLAN header.

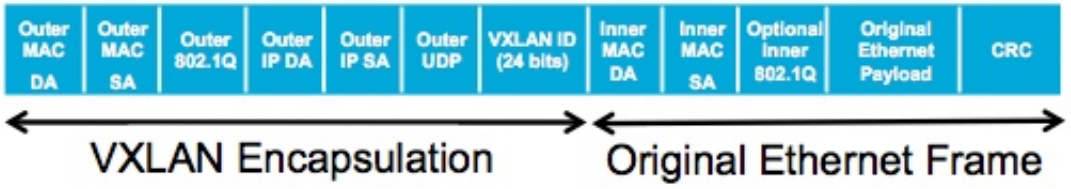

FIGURE 2.10: VXLAN packet header

Figure 2.10 illustrates the header of a VXLAN packet: The original Ethernet frame is encapsulated a VXLAN ID (recognised as VNI), UDP header and other fields to become a VXLAN-encapsulated frame. When this frame arrives at the receiving host machine, it needs to go up the network stack for the first time to extract the original frame. This original frame is then fed to the network stack for the second time in order to be delivered to the its intended recipient (inner MAC DA). According to [22],

a software implementation of this packet processing may result in an overhead of 30% CPU resources.

According to VXLAN protocol design, different from other types of overlay network like VPN, the VXLAN-based tunnelling does not provide any encryption mechanism, causing the tunnelled traffic to be exposed insecurely while travelling through the network.

2.5.2 VXLAN Tunnel End Point (VTEP)

With the introduction on VXLAN protocol, the RFC7348 also defines a new entity named *VXLAN Tunnel End Point* (VTEP), which connects an access switch to the IP network. A VTEP is located in a hypervisor where guest machines (VMs) are hosted, such that it will be able to encapsulate packets sent out by VMs within IP header and deliver them across the IP network. This encapsulation is transparent to the VMs.

2.5.3 VXLAN Tunnel In OpenStack

OpenStack establishes a full mesh of VXLAN tunnels between the all nodes running OVS agent (`neutron-openvswitch-agent`), namely, the Network node and Compute nodes. Systematically, every time an OVS agent starts up (i.e. a Compute node has finished its installation), it uses the AMQP-based message queue to notify the Neutron server about its presence. The controller then informs all pre-existing Compute and Network nodes of the new node, causing new VXLAN tunnels to be formed between the former and the latter. As a host-to-host tunnel might be shared among several tenant networks, a tunnel ID header field in the VXLAN header is used to distinguish packets belonging to different tenant networks.

A VTEP is responsible for encapsulation and de-encapsulation of VXLAN-based packets before they enter and after they leave the tunnel, respectively. In the studied OpenStack environment, VTEP is implemented as the OVS's tunnelling bridge (`br-tun`). This implementation, known as soft VTEP, utilises OVS kernel module's tunnelling capability.

Chapter 3

Methodology

Since this thesis puts its focus into the network performance analysis of OpenStack, a fully functional OpenStack software first needs to be deployed. To experience its deployment and operation in different environments, OpenStack is going to be installed on both an Amazon EC2 test-bed and a local server test-bed. Installation scripts are adapted in accordance with configuration details provided by each environment so as the deployment process can be automated. As discussed above, the 10th OpenStack release - *Juno* - is selected for the experimental deployment and evaluation.

3.1 Network Performance Measurements And Evaluations

3.1.1 Overview

This section discusses how experiments are carried out in the constructed OpenStack test-beds under different scenarios. In general, there are numerous tests conducted to measure bi-directional traffic, in terms of network *throughput* and *latency*, between provisioned VM instances. These cover all different scenarios discussed in Section 3.2. Tests are automated to run with various bash scripts designed for different experiment types and scenarios. Their results are stored into plain-text file for further processing (filtering, graph generating and so on). The final graphs are generated based on tests that are conducted over 10 days, calculated as average value. Result of each day is derived from the median value of 3 rounds measurement.

In order to perform planned performance evaluation, all evaluated networks are configured with security policies that allows `ssh` traffic (default to port 22) for connection to VMs. The rules also need to allow `ICMP` packets and traffic targeting port 5001, to which `iperf` in server mode listens. Tools used for system, network monitoring and traffic evaluation include:

- `ping`: Used to exchange ICMP packets between two designated VMs
- `iperf`[23]: Used to generate data exchanged between two designated VMs
- `perf`[24]: Used to perform system profiling
- `top`, `atop`, `atopsar`: Used to track system usage and provide activity report
- `tcpdump`[25], `wireshark`[26]: Used to monitor and capture network traffic for further analysis

3.1.2 Throughput Evaluation

To evaluate traffic throughput, `iperf` is used as the data generation tool. Each VM instance in a pair is configured to in turn run `iperf` in server and client mode, respectively, so it can receive/send data to/from its peer. `iperf` in server mode listens for TCP connection on port 5001, awaiting for client request. In client mode, the configured VM continuously sends data to the VM running `iperf` in server mode. To better observe variability, the `iperf` client outputs throughput value every 1 second.

To mitigate the impact of initial packet transfer delay (due to OVS's flow entry missing), we run `iperf` in 60 seconds and discard results from the first 10 seconds. Thus, the result is generated by observing throughput values within 50 seconds.

3.1.3 Latency Evaluation

The latency measurement is conducted using the `ping` tool. Similar to the throughput test, each VM in a pair in turn pings the other VM in 60 seconds. To achieve better consistency of the overall measurement, we strip out 10 first pings and only consider the remaining 50 ones. The latency measurement simply considers the packet delay time between two VMs within these 50 ICMP packets.

3.1.4 CPU Statistic And Profiling

By running some system monitoring tools like `top` or `atop`, we can easily observe system processes associated to OpenStack operations and their resource consumption. In order to understand better what happens with the underlying infrastructure, we will be doing system profiling with the `perf` tool. Being part of the Linux kernel that runs underlying operating system, `perf` is able to provide us with accurate event counting with respect to the function level.

In order to collect detailed CPU statistic of the taken measurements, we run `perf` in recording mode in 60 seconds, with focus on the 'cpu-cycles' event. The command is as followed:

```
/usr/bin/perf record -e cpu-cycles -c 100000 -ag -- sleep 60
```

In order to view the report of collected statistic, following command is used:

```
/usr/bin/perf report -n --sort comm --stdi
```

This allows us to see threads/processes with their CPU consumption details and to trace them down to the invoked functions.

3.2 Experimental Scenarios Based On Traffic Flow

Designated experiments consider various types of traffic with regards to the placement of VMs on Compute host, the (virtual) network they belong to and the (virtual) router they are connected to. Throughout the study, the following terms are used to refer to different types of evaluated network traffic:

- *Intra-node* refers to the traffic exchanged between two VMs that reside in the same Compute host
- *Inter-node* refers to the traffic exchanged between two VMs that reside in different Compute host

- *Intra-subnet* refers to the traffic exchanged between two VMs that belong to the same (virtual) subnet
- *Inter-subnet* refers to the traffic exchanged between two VMs that belong to different (virtual) subnet, of either the same or different tenant.
- *DVR-based* refers to traffic in a network based on distributed virtual router
- *Non-DVR/legacy/centralised router* refers to traffic in a virtual network that operate with conventional centralised (virtual) router

In order to accurately specify the flow of VM traffic, we need to track how packets are sent and received by different ports and bridges within physical hosts. On one hand, a VM of the observed pair is set to continuously ping its peer. On the other hand, we observe the traffic flowing through all available network interfaces (displayed using `netstat -i` command) and decide those participating in sending and/or receiving packets. After that `tcpdump` is run to capture the transferred packets so that the packet headers and patterns can be learnt.

```
tcpdump -i <interface> -n icmp -e -v
```

Based on the analysis of monitored traffic, this section specifies various experimental scenarios based on the observed traffic flow patterns. Two architecture options are considered according to the implementation of Neutron-based virtual routing feature: Legacy router and distributed router.

3.2.1 Network Architecture With Legacy Router

3.2.1.1 Intra-node Intra-subnet

This scenario examines the traffic passing between two VMs (denoted A1 and A3) residing in the same Compute node and belonging to the same subnet. Under this scenario, traffic simply goes from one VM to the other VM through Compute node's internal `br-int`. As such, exchanged traffic remains inside the Compute host and no tunnelling is required.

Let *HNR* denote the VM pair exchanging this traffic category (Figure 3.1).

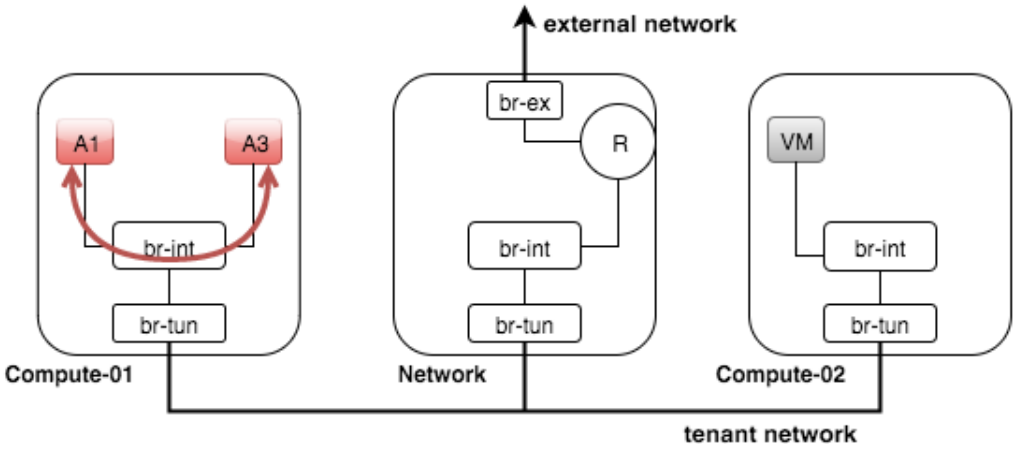

FIGURE 3.1: Intra-node Intra-subnet VM traffic flow (HNR)

3.2.1.2 Intra-node Inter-subnet

This scenario examines the traffic passing between two VMs (denoted A1 and B1) residing in the same Compute node but belonging to different networks. Under this scenario, as the two VMs reside in different subnets, traffic from A1 is firstly sent to a virtual router (denoted R) located in Network node. R then forwards this traffic to B1's subnet before sending it back to the same Compute node where it is received by B1. This scenario requires VXLAN-based tunnelled connection between Compute Node and Network node. As A1 and B1 belong to two different tenant networks, there would be two packet flows, each with a unique VXLAN ID.

Let *H-NR* denote the VM pair exchanging this traffic category (Figure 3.2).

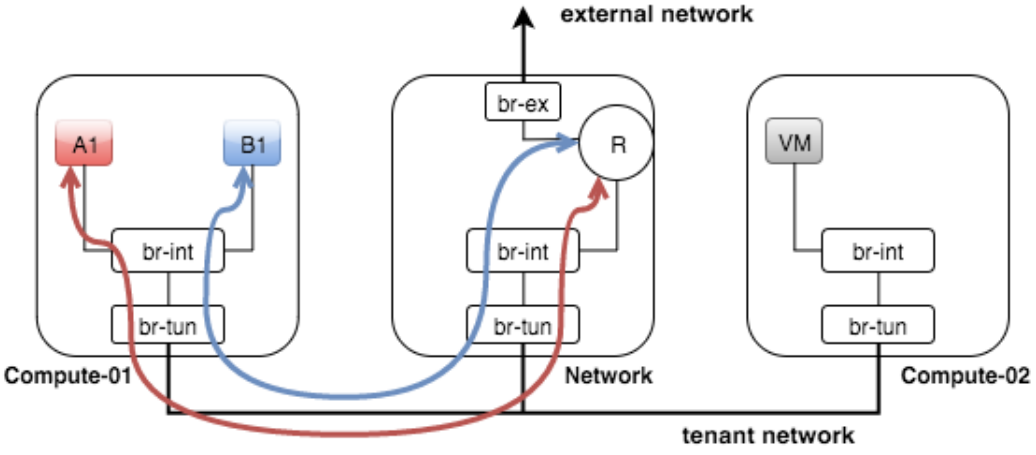

FIGURE 3.2: Intra-node Inter-subnet VM traffic flow (H-NR)

3.2.1.3 Inter-node Intra-subnet

This scenario examines the traffic passing between two VMs (denoted A1 and A2) residing in different Compute nodes and belonging to the same subnet. Packets leave A1, get transferred via `br-int`, then `br-tun` and packaged with VXLAN encapsulation before entering the tunnel network. After being received and de-encapsulated by `br-tun` on node Compute-02, packets are forwarded to `br-int` before being arriving at VM A2. No routing is required here.

Let *-HNR* denote the VM pair exchanging this traffic category (Figure 3.3).

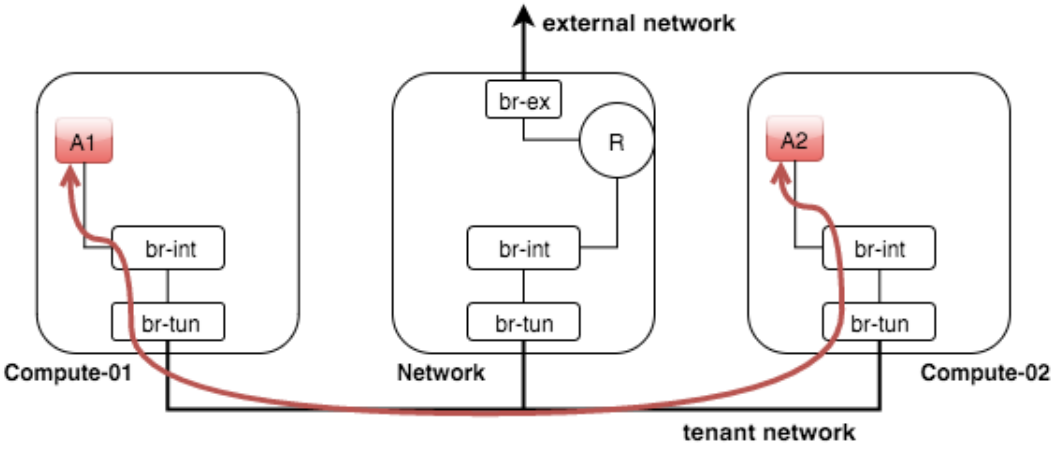

FIGURE 3.3: Inter-node intra-subnet VM traffic flow (-HNR)

3.2.1.4 Inter-node Inter-subnet

This scenario examines the traffic passing between two VMs residing in different Compute nodes and belonging to different networks. Under this scenario, traffic from one VM is passed to the virtual router located on Network node which then forwards it to the other VM located on another Compute node.

Let *-H-NR* denote the VM pair exchanging this traffic category (Figure 3.4).

3.2.2 Network with DVR router

DVR router equips each configured Compute node with routing capability, thus allowing VM traffic to be passed directly between different virtual networks without the need to

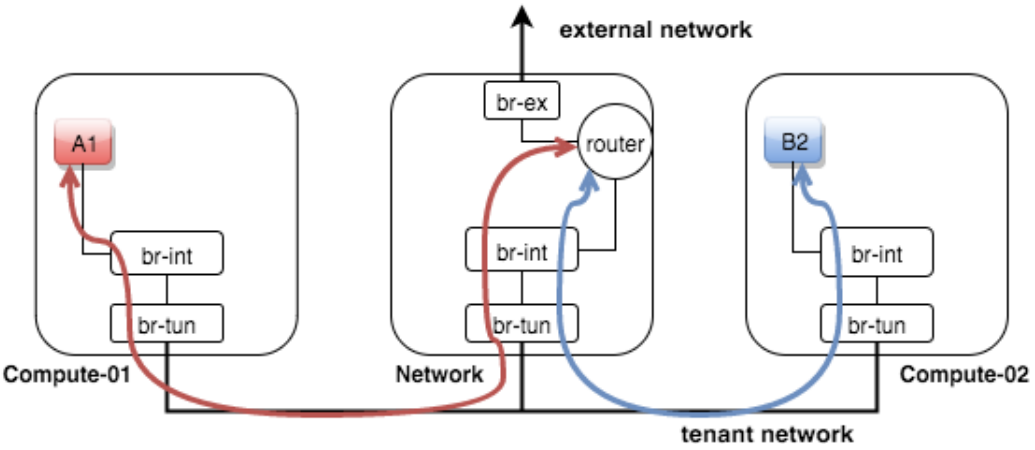

FIGURE 3.4: Inter-node inter-subnet VM traffic flow (-H-NR)

flow through Network node. With the DVR enabled, while the flow patterns for intra-subnet traffic (`HNR`, `-HNR`) remain the same, the two inter-subnet traffic scenarios, `H-NR` and `-H-NR`, experience some changes and are accordingly re-illustrated in 3.2.2.1 and 3.2.2.2, respectively.

3.2.2.1 Intra-node Inter-subnet (with DVR router)

With the presence of a router, traffic between VMs is simply forwarded by the router and remains inside the Compute host. As such there is no impact from the VXLAN encapsulation.

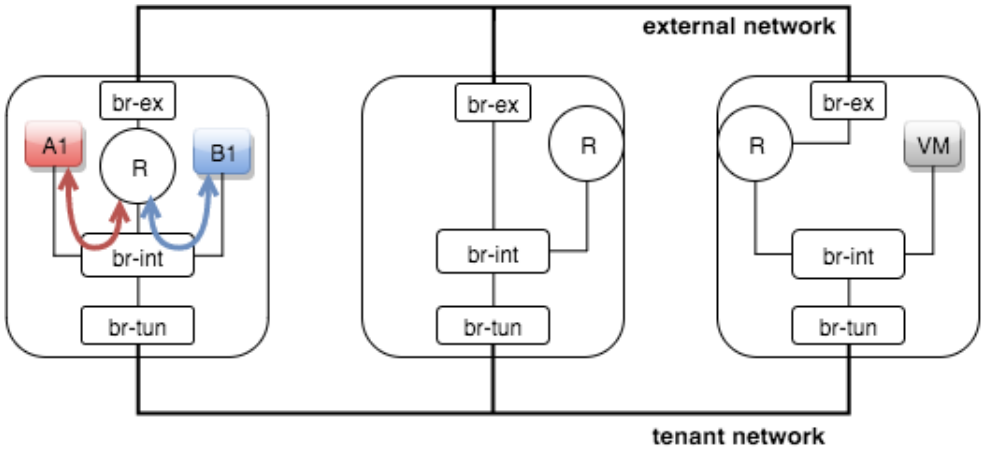

FIGURE 3.5: Inter-node inter-subnet VM traffic flow (with DVR router)

3.2.2.2 Inter-node Inter-subnet (with DVR router)

As in Figure 3.6, traffic between VMs are sent directly from one Compute node to the other one, with each virtual router being responsible for its own local subnet.

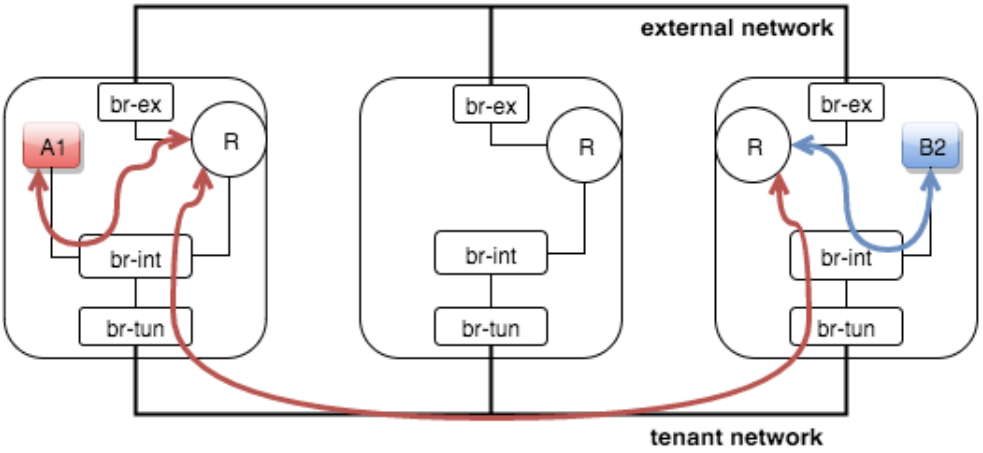

FIGURE 3.6: Inter-node inter-subnet VM traffic flow (with DVR router)

Chapter 4

Experimental Setup

4.1 Overview

Within the scope of this thesis project, an OpenStack cluster is deployed and consists of 4 nodes: 1 Controller, 1 Network and 2 Compute nodes. Each node runs specific software packages in order to perform its designated role, as illustrated in Figure 4.1. Figure 4.1 also depicts a simplified architecture of the cluster where there are four nodes, namely 1 Controller, 1 Network and 2 Compute nodes, connected through their three network interfaces, forming three distinct networks: *Management*, *Tunnel (Tenant)* and *External* networks. Each node comes up with certain software packages in order to perform expected functionalities.

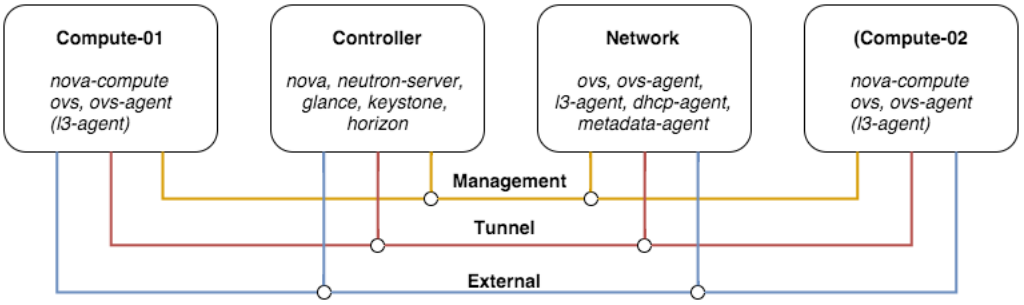

FIGURE 4.1: Basic component of OpenStack

The installation of OpenStack is carried out in three different platforms: Local virtual environment using VirtualBox, remote virtual environment in Amazon's AWS platform and the local Test-bed located in server room of our department.

4.2 Single-host Installation

OpenStack is installed on a personal computer running Mac OSX operating system and VirtualBox is used to create Virtual Machine as guest OS. The installation is based on the Devstack[27] script in its all-in-one mode. This single-host installation brings an initial hands-on experience with OpenStack functionalities while simplifying the installation process and remaining suitable for a computer with limited hardware resources. Under all-in-one mode of Devstack, basic OpenStack modules and services (*Identity*, *Nova*, *Neutron*, *Compute*, *Dashboard*) are installed and run on the same host machine.

While Devstack effectively provides an instant and effortless way of experiencing OpenStack, it does not fully expose the installation process and is less customisable. This unfortunately makes it harder to understand the system, as well as to debug its components should any problems occur during either installation or operation stage. Moreover, DevStack is considered more suitable for starters rather than an ideal solution for a full-featured OpenStack cluster.

4.3 Amazon AWS Testbed

To overcome the limitations discussed in section 4.2, a multi-node setup of OpenStack is approached, essentially involving deployment of required software components into different nodes within a cluster (consisting ideally of 4 servers). As a personal computer does not provide sufficient resources, while the local cluster test-bed (of 4 physical servers) is yet to be ready, Amazon Web Service (AWS)[28] Elastic Cloud Computing (EC2) platform is utilised to facilitate this multi-node OpenStack installation. Table 4.1 depicts the configurations to initialise EC2 instances in AWS.

Cloud image type	ami-3d50120d/Ubuntu 14.04
Instance type	T2.Medium
Availability region / zone	us-west-2 / us-west-2a
Private IP addresses	Controller: 172.31.28.200 Network: 172.31.28.201 Compute-01: 172.31.28.202 Compute-02: 172.31.28.203

Table 4.1: AWS Configuration

Since AWS does not support associating IP addresses with EC2 instances' virtual bridges, OpenStack cluster on AWS is installed with basic configuration using a single `eth0` interface for both *Management* and *Tunnel* networks. This interface has a throughput of approximately 1 Gbps (as measured with `iperf`). Experiment with *External* network is not covered in AWS test-bed. Network architecture of AWS test-bed is depicted in Figure 4.2. The deployment of OpenStack on AWS necessitates four *t2.medium*[29] EC2 instances corresponding two four OpenStack nodes: Controller, Compute (x2) and Network. The *t2.medium* instances are considered sufficient in providing computing and network resources required by the experiments. For software part, the *OpenStack Juno* release is used in the experiment as being a stable version in the market.

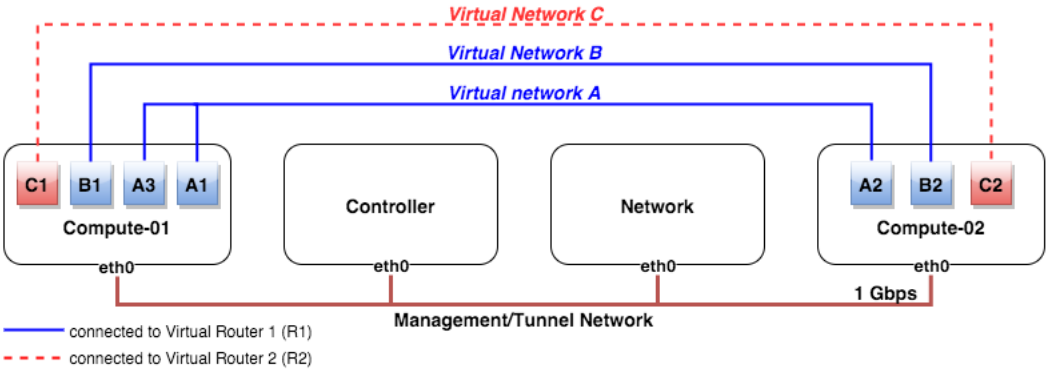

FIGURE 4.2: Amazon AWS Network architecture

The experimental cluster consists of 3 virtual networks A, B and C, among which A and B are connected to Virtual Router 1 (R1) while C is connected to Virtual Router 2 (R2). 7 VMs are created such that for every virtual network, there is at least 1 VM located on each of the two Compute nodes 01 and 02. Figure 4.2 illustrates the distribution of VMs into virtual networks and compute nodes.

To avoid potential hassles from manual installation, a bash script, derived and adapted from the OpenStack Cookbook[30], is used to facilitate the setting up of the OpenStack cluster. Furthermore, we decided to use Vagrant[31] as an automation tool for building complete development environments. The AWS API [32] adds an AWS provider, allowing Vagrant to synchronise all local installation and configuration files to the newly instantiated EC instances and execute them at boot time. In so doing,we can automate the installation process that takes place remotely between the client computer and the EC2 instances hosted on Amazon cloud. As AWS API provided for Vagrant is relatively limited, the installation process is not fully automated: Additional network interfaces

can only be manually attached into VMs after they have been instantiated. This shortcoming causes any configuration related to the second (*Tunnel*) and third (*External*) network interfaces unavailable at the time of installation. Alternatively, Amazon provides a python API, boto[33], that supports better remote instantiation of EC2 instances including configuring additional network interfaces.

4.4 Local Cluster Test-bed

Compared to the AWS platform, an OpenStack testbed consisting of multiple physical servers provides more privileged control over the hardware resources and operating system, thus makes it more accessible to the setup of the cluster. With this deployment, Juno release is selected due to its better stability and extended features, including particularly Distributed Virtual Routing (DVR), compared to its predecessors. The installation script is upgraded and adapted to match with Juno and new test-bed environment.

Figure 4.3 depicts the architecture of the OpenStack cluster deployed on the local Test-bed, consisting of 4 servers connecting to each other via 3 network interfaces with different bandwidth capacities.

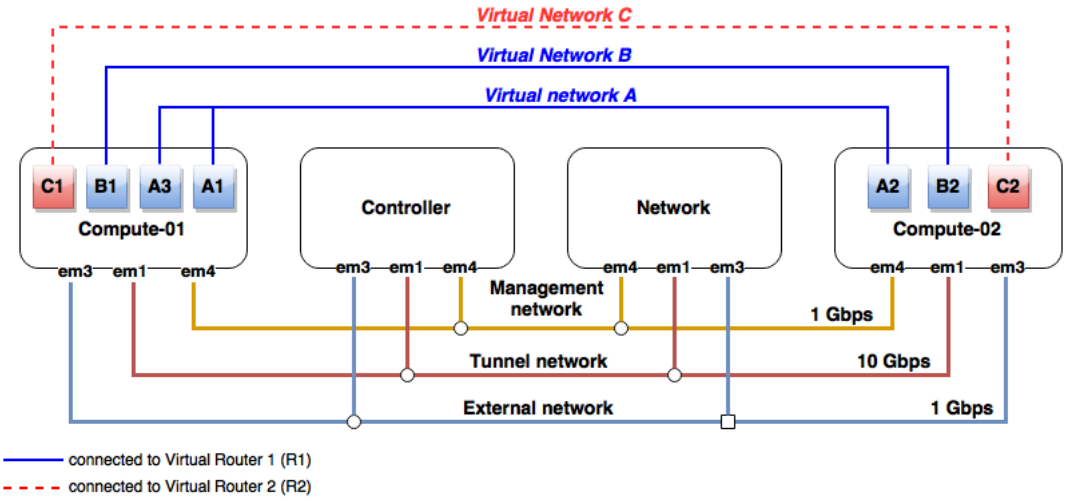

FIGURE 4.3: Test-bed Network architecture

Table 4.2 summarises general server configurations used to deploy the Test-bed.

A standard OpenStack deployment requires 3 distinct network interfaces. In terms of underlying infrastructure, the test-bed has its Management network run on a dedicated

CPU	16x physical cores with HyperThreading enabled
Memory	128 GBs
Hard disk	1 TB
Image	Ubuntu Cloud 14.04
Network interfaces	em1 (10 Gbps), em3 (1 Gbps), em4 (1 Gbps)
Network	OpenFlow switch controlled by Ryu

TABLE 4.2: Test-bed server configurations

physical switch with 10 Gbps interface while the Tunnel (Tenant) and External network interfaces run on OpenFlow [34] switch controlled by Ryu [35].

As illustrated in Figure 4.3, VNs and VMs are created and distributed in a way similar to two the AWS cluster setup discussed in section 4.3. Each VM is configured as specified in Table 4.3.

Memory	512 MB
Hard disk	2.2 GB
Image	Ubuntu Cloud 14.04
Hypervisor	KVM (QEMU with KVM acceleration)
Network	Neutron + OpenvSwitch + VXLAN Tunnel
Network	Public/private key

TABLE 4.3: Local testbed VM Configuration

The deployment of OpenStack cluster on the local test-bed requires extra caution. As a matter of fact, the test-bed is part of our faculty's infrastructure, and is thus less capable of tolerating faulty operations than virtual servers (i.e. if anything goes wrong they cannot be instantly cleaned out like with a VM). With that in mind, although the installation script has been well tested on virtual environment, the cluster is deployed with separate bash commands rather than by a straightforward run of the whole script files.

4.5 Other Configurations

The section discusses certain important configurations made before the installation is finalised to make sure that OpenStack can be smoothly integrated into the host machine environments.

4.5.1 KVM And VHostNet

Both KVM and VHost-Net are Linux kernel modules that, when enabled, can significantly boost up the performance of guest machines in a host environment. Figure 4.4 illustrates how KVM and VHostNet modules interact, as well as how the latter acts as a userspace interface for QEMU-based guests.

As the QEMU-based guest machines' access to memory is very expensive due to context switches between kernel space and user space. The VHostNet module runs in kernel as a kernel thread and interrupts the guest with much less overhead. As such it provides near native performance.

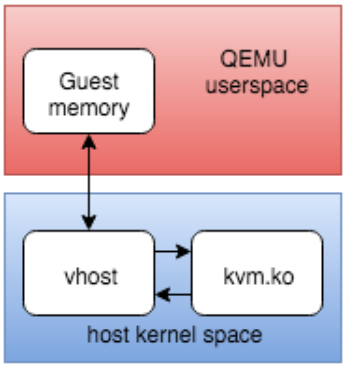

FIGURE 4.4: KVM and VHostNet

Since KVM and VHostNet are not enabled by default when OpenStack is installed, this section discusses some configurations that can help VMs benefit from those two modules. To explicitly enable KVM for OpenStack Nova, following is configured on Compute nodes:

```
compute_driver = libvirt.LibvirtDriver
[libvirt]
virt_type = kvm
```

As the Compute nodes are running Intel processors, their KVM module can be enabled as follows:

```
modprobe -a kvm-intel
```

VHostNet (`vhost-net`) is another kernel module that needs to be loaded in order to improve network performance of VMs. This provides better latency and much greater throughput for network. In all Compute nodes where VMs will be instantiated, VHost-Net can be enabled as follows:

```
modprobe vhost_net
```

There is a huge different in terms of network performance before and after the `vhost-net` module is enabled. A close comparison in Chapter 5 will show us the gap in performance that the duo bring to OpenStack VMs.

4.5.2 MTU Size

The *maximum transmission unit* (MTU) of VMs is set at 1450 bytes, taking into consideration the additional 50 bytes of VXLAN header (added by `br-tun`) when transmitted via VXLAN tunnels. As such, frames leaving VMs and getting transferred in physical network will have the standard MTU size of 1500 bytes and thus will not cause any further frame decomposition, which might worsen the network performance.

4.5.3 Security Group

In order to allow `ssh`, `ICMP` and `iperf` traffic types within OpenStack tenant networks, these following security rules are added to the system:

```
nova secgroup-add-rule default tcp 22 22 0.0.0.0/0
nova secgroup-add-rule default icmp -1 -1 0.0.0.0/0
nova secgroup-add-rule default tcp 5001 5001 0.0.0.0/0
```

4.6 Logical Network Architecture

This section depicts the logical network architecture and the distributions of VMs into VNs and Compute nodes.

As can be seen in Figure 4.5, there are 3 virtual networks, in which A (11.0.0.0/8) and B (12.0.0.0/8) are attached to virtual router 1 (R1) while C (13.0.0.0/8) is attached to

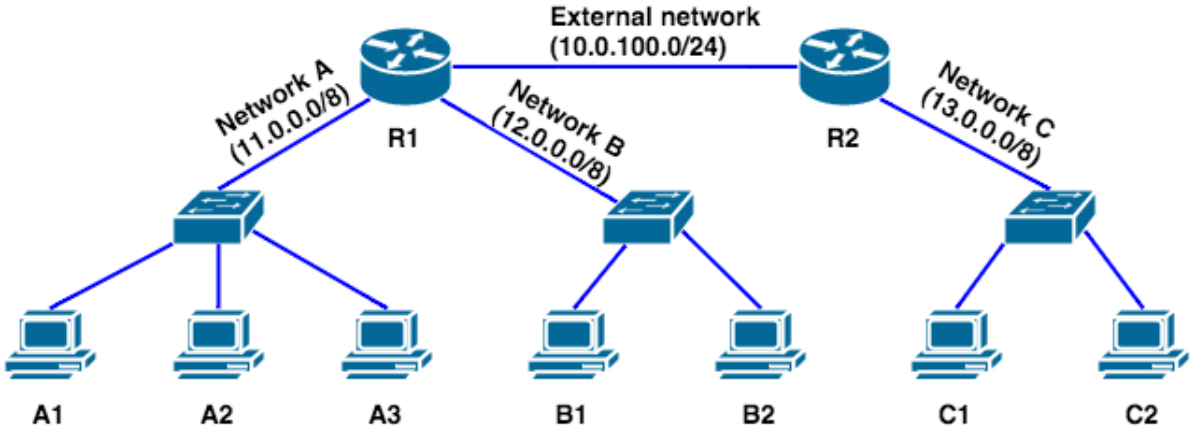

FIGURE 4.5: Logical network

virtual router 2 (R2). For the conducted experiments, R1 and its attaching VNs (A, B) and VMs (A1, A2, A3, B1, B2) belong to *tenant 1* while R2 and its attaching VN (C) and VMs (C1, C2) belong to *tenant 2*. R1 and R2 are connected via the External network, belonging to the same physical subnet address (10.0.100.0/24) of all servers in the cluster via `em1` interface.

	Compute-01	Compute-02
Network A	A1, A3	A2
Network B	B1	B2
Network C	C1	C2

TABLE 4.4: VM distribution to virtual networks and Compute nodes

Table 4.4 summarises how 7 VMs are distributed among 3 designated VNs and 2 Compute nodes such that different types of traffic can be observed from the experiments. In general, one VN has two VMs, each of which is scheduled on of the two Compute nodes. Network A exceptionally has 3 VMs, among which A1 and A3 reside in the same node. Details of these traffic categories will be discussed in Section 3.2.

Denotation	**VM name**	**Private IP**	**Floating IP**
A1	VM1	11.0.0.17	10.0.100.111
A2	VM2	11.0.0.18	10.0.100.102
A3	VM3	11.0.0.21	10.0.100.103
B1	VM4	12.0.0.5	10.0.100.104
B2	VM5	12.0.0.6	10.0.100.105
C1	VM6	13.0.0.7	10.0.100.116
C2	VM7	13.0.0.10	10.0.100.117

TABLE 4.5: VM IP addresses

List of 7 instantiated VMs with their name and public (floating) IP and private (tenant network) IP addresses are given in Table 4.5.

Chapter 5

Experiment Result

5.1 Amazon AWS

This section briefly presents the results of the experiments conducted on Amazon AWS test-bed. Since the EC2 instance's network interfaces only work with their default ports, additional networks are not available for a dedicated Tunnel and External networks. Consequently, the results only concern evaluation taken through a Tunnel network which is shared with the Management network.

5.1.1 Thoroughput Measurement

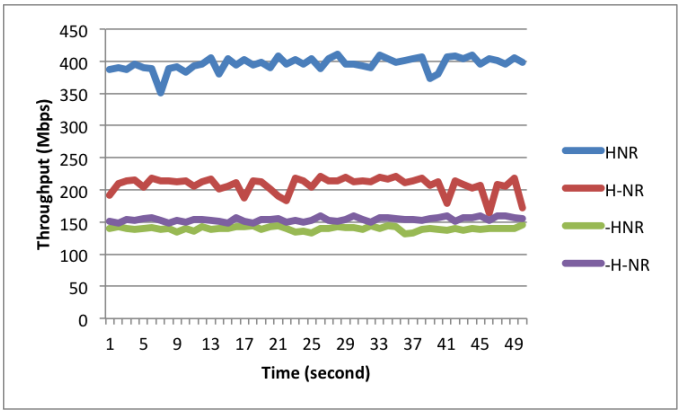

FIGURE 5.1: AWS throughput measurement

Figure 5.1 illustrates the results of throughput measurement conducted with different VM pairs without DVR enabled. The case of `HNR`, undoubtedly has the highest throughput, reaching 400 Mbps. Meanwhile, two cases in which VMs are in different Compute

nodes experience the data rate of around 150 Mbps, lowest among the 4 cases. `H-NR` implies that two VMs are in the same node but their traffic needs to visit the Networking for routing service. Under this circumstances, the achievable network throughput is roughly 200 Mbps. In general, the network performance in all cases are relatively poor (no ore than 200 Mbps throughput in most cases), even if traffic only stays inside the same host and does not suffer from distance or packet encapsulation overheads.

5.1.2 Latency measurement

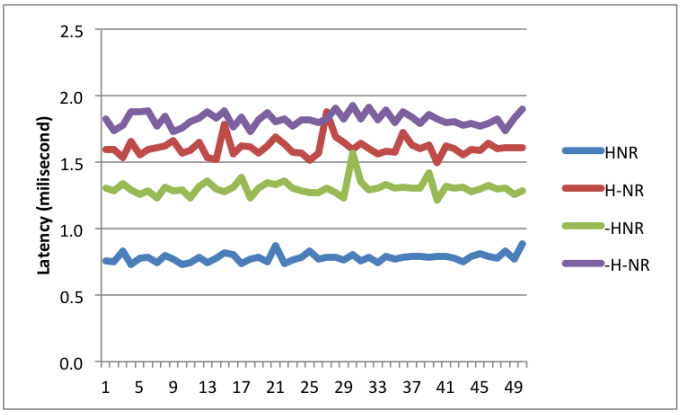

FIGURE 5.2: AWS latency measurement

Figure 5.2 shows the packet delay of the designated VM pairs. As can be seen from the result, the inter-node inter-subnet traffic type (`HNR`) is the most stable and achieves the lowest latency at around 0.75 ms. It is of no surprise that `-H-NR` might have the highest packet delay due to the overheads from both packet encapsulation and routing. In two cases where the VM traffic has to visit the Network node from one Compute node, if it comes back to Compute node (`H-NR`), the measured latency is higher than if traffic is forwarded to the other Comptute node (`-HNR`). Again, except for the `HNR` case, all 3 other cases have their packet delay significantly higher than the host-to-host delay (less than 0.5 ms).

5.1.3 Conclusion

In short, deploying a cloud platform onto an already-a-cloud environment like Amazon AWS is obviously not efficient. In particular, an EC2 instance is virtualised hardware, so it does not provide an ideal infrastructure to run a proper hypervisor. Running

a hardware emulator like QEMU surely results in a poor system performance because direct access to underlying hardware is not allowed (performance-critical kernel modules like kvm or vhost-net are not supported).

On the bright side, though, we can experiment how bad it is to run a run a cloud inside another cloud. More importantly, this deployment on AWS provides a preliminary experience with OpenStack, in order to study its features, operation, architecture, and observe its traffic pattern.

5.2 Local Test-bed

The experiment scheme carried out on the local Test-bed is similar to the one on the Amazon AWS but there are different settings applied. As experiments are to be conducted in dedicated servers and networking environment, it is reasonable to assume that a more stable network performance could be achieved. Moreover, the full control of the network infrastructure allows the deployment of a full-featured OpenStack Juno with Distributed Virtual Routing, the virtue of which allows direct traffic exchange between Compute nodes that hosts VM instances (instead of indirect traffic exchange via the Network node when routing is required). As a consequence, Juno on local test-bed is expected to offer higher throughput and lower latency in tenant's network traffic compared to the AWS.

Results are presented with regards firstly to the designated network interfaces, namely, Tenant Network (private address) and External Network (public address). With each type of interface, we will consider the throughput and latency measurement with and without the use of DVR router.

5.2.1 Tenant Network

This section presents the result of network performance evaluation when tests are conducted using VM's private networks. This means traffic flows through the Tenant network that has the theoretical bandwidth of 10 Gbps.

5.2.1.1 Throughput measurement

Network throughput with legacy router

Figure 5.3 and Figure 5.5 illustrate the data transfer throughput between two VMs via tenant network when traditional centralised router is used.

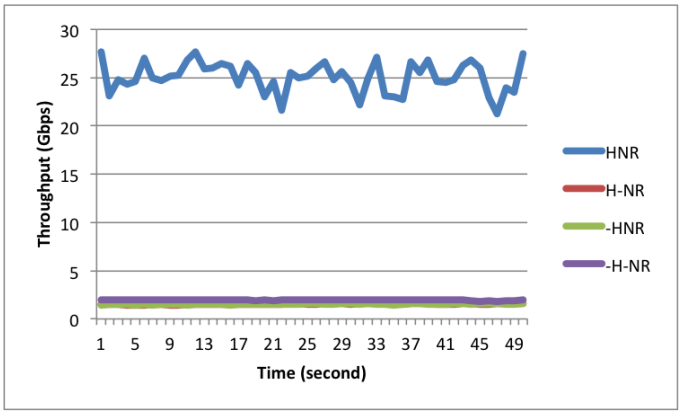

FIGURE 5.3: Network throughput with legacy router (TCP)

In Figure 5.3, it is obvious that VM pair being in both the same physical host and the same subnet can achieve very high throughput (16 Gbps), much higher than any other 3 pairs whose performance gets stuck at around 2 Gbps. It is understandable that in the former case (`HNR`) traffic simply travels inside the same Compute node and is subjected to a very high data rate (around 25 Gbps). Meanwhile, in all latter cases (H-NR, -HNR and -H-NR), sent packets need to pass through the Tenant network to reach the recipient VM, be it in the same or different Compute host.

Network throughput with distributed router

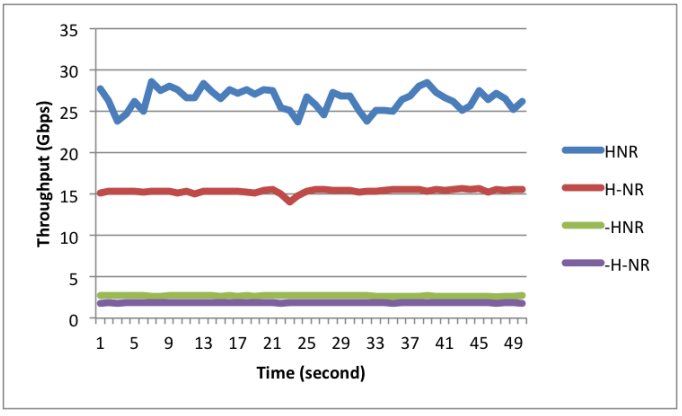

FIGURE 5.4: Network throughput with distributed router (TCP)

Figure 5.4 illustrates the network throughput when distributed router is used in Neutron network. Similar to the case without distributed router, intra-node intra-subnet (HNR) VMs are able to achieve a very high data transfer rate of more than 25 Gbps. Meanwhile the data transfer rate of VM pairs that are in different Compute nodes, `-HNR` and `-H-NR`, is limited at approximately 2.5 Gbps or less. VMs residing in the same host but in different network are able to achive a throughput of 15 Gbps. This implies that there are certain impacts of network routing even in the same physical node, especially when a large amount of data is handled.

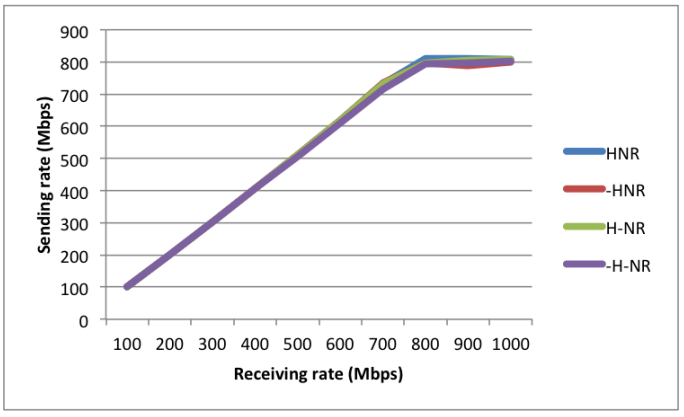

FIGURE 5.5: Network throughput with DVR router (UDP)

Figure 5.5 shows the data throughput between VMs if UDP is used. In general, the receiving rate is proportional, and roughly equivalent, to the sending rate up until the former reaches 800 Mbps. This means even if the sending VM tries to send UDP at higher rate than 800 Mbps, the receiving data rate will not be able to cross its limit. This holds for all participating VM pairs. As this issue is caused by the host machines that involves data generation capability, further research is not in the scope of this study.

5.2.1.2 Latency measurement

Network latency with legacy router

Figure 5.6 represents the network latency measured in legacy network with non-DVR router. We can easily observe the difference in how traffic is delayed when VMs are placed in different physical servers as well as when they belong to different virtual networks. It is not surprising to see VMs in the same network and same subnet experience the lowest latency (below 0.4 ms). On the contrary, VMs belonging to neither the same compute

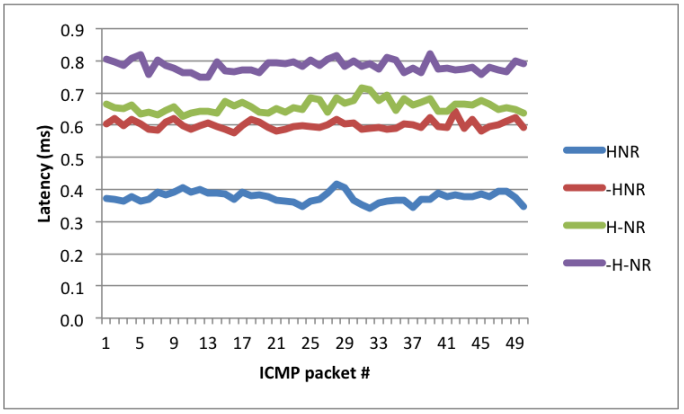

FIGURE 5.6: Network latency with legacy router

host, nor the same subnet, endure twice as much of the latency (at 0.8 ms). There is a subtle difference (about 0.05 ms) in latency if the VMs belong to either same subnet or same Compute host. In these two cases, `H-NR` and `-HNR`, the latter's latency is roughly 0.6 ms while the former's latency fluctuates around 0.65 - 0.7 ms.

Network latency with distributed router

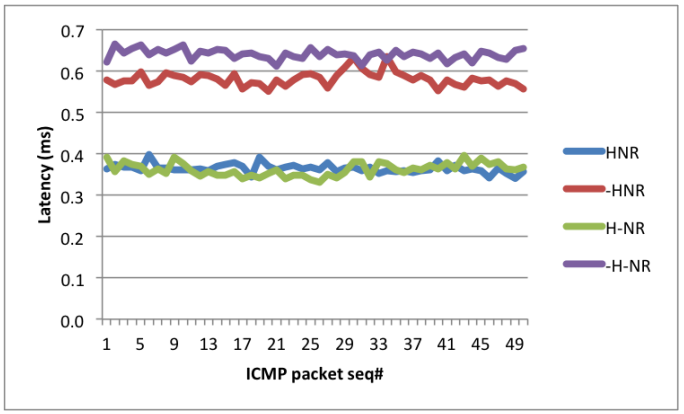

FIGURE 5.7: Network Latency with distributed router

Figure 5.7 depicts network latency measurement when DVR is enabled. From the first glance, we can see there is a certain performance gap between VM pairs exchanging intra-node traffic (`HNR`, `H-NR`) and pairs exchanging inter-node traffic (`-HNR`, `-H-NR`). Similar to the case with legacy router, inter-node VM pairs tend to have higher (at roughly 50% or 0.2 ms) latency compared to intra-node VM pairs. As the Compute node can now perform routing capability itself, whether VMs are in the same network or not does not cause any significant overhead. Meanwhile, packets travelling between two Compute host suffer a delay of 0.6 ms.

In both cases, legacy and distributed router, traffic is received without any packet loss.

5.2.1.3 Performance comparison: DVR versus non-DVR

Figure 5.8 and 5.9 give, respectively, comparative views of the network throughput and latency with regards to the utilisation of DVR under 4 different discussed scenarios.

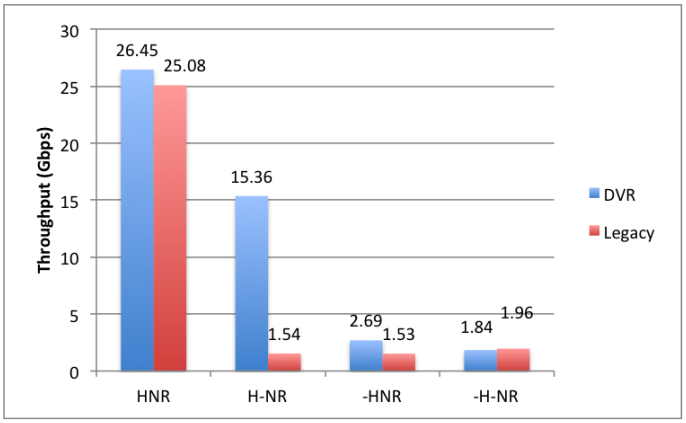

FIGURE 5.8: Comparison: Network throughput with and without DVR

Figure 5.8 shows that while there are no considerable gaps in achievable throughput with most of the considered scenarios, the `H-NR` VM pair experience a huge difference brought in by DVR: 15.36 Gbps vs. 1.54 Gbps. This shows the difference when the traffic is limited by the Tenant network interface which also runs the VXLAN encapsulation.

As can be seen from Figure 5.9, in most cases the network latency is not significantly different with and without DVR being used, except when two VMs reside in physical node but belong to two different networks. This particular scenario witnesses almost twice as much (0.66 ms and 0.36 ms) in latency that the non-DVR network has compared to that of DVR-based network.

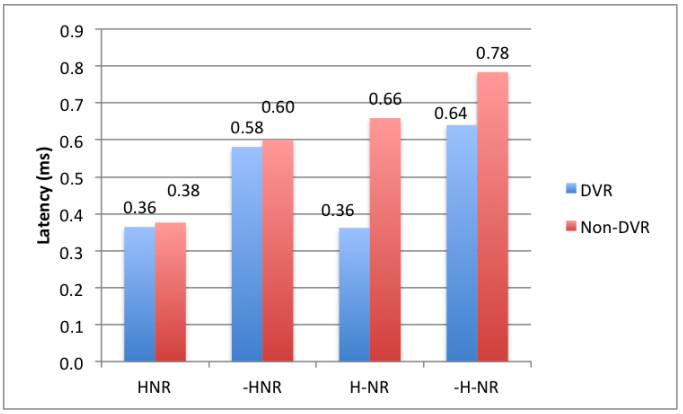

FIGURE 5.9: Comparison: Network Latency with and without DVR

5.2.1.4 Performance comparision: KVM/VHostNet

Figure 5.10 and 5.11 illustrate the difference in network throughput and latency before and after the KVM, and particularly VHostNet modules are enabled.

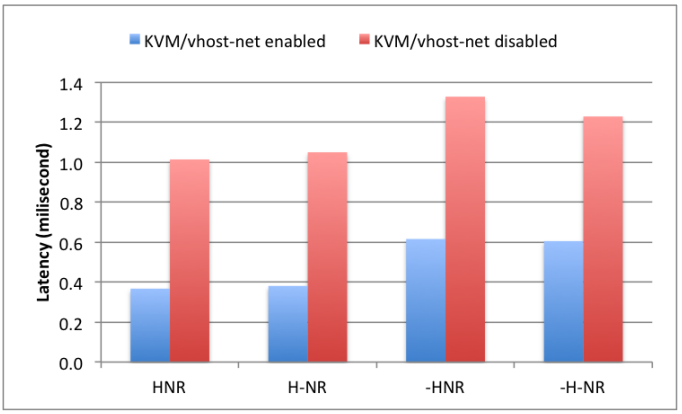

FIGURE 5.10: Comparison: Network throughput with and without VHostNet/KVM

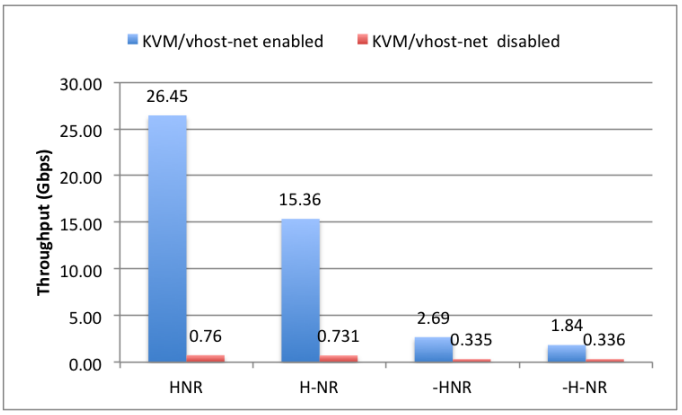

FIGURE 5.11: Comparison: Network latency with and without VHostNet/KVM

As can be seen from the two figures, without KVM acceleration and vhost-net driver enabled, all VMs have a very very poor performance. With KVM/VHostNet both enabled. While the latency is more than twice better, the performance gap that these two bring to the measured throughput is insanely big. We can simple say that is a Megabit-vs-Gigabit difference.

5.2.2 External Network

To exchange experimental traffic via External interface, VM instances use their floating or public IP addresses for the connection. It needs to take into account that, in legacy

routing mode, the VMs that want to use their public IP addresses for connection have to send data to the Network node via the Tunnel network (as explained in section 2.3.5.1). If this is the case then it becomes unnecessary to take the experiment because the data will not flow through the External network interfaces of Compute nodes. Consequently, we only consider in this section the experiment with VM pairs running in distributed routing mode where each of the VMs has its traffic sent directly for its hosting Compute node (as discussed in section 2.3.5.2). Eventually this enables us to see how much (of the 1 Gbps) bandwidth of the External network interface is utilised, compared to the Tunnel/Tenant network.

5.2.2.1 Throughput measurement

Network throughput with distributed router

For the purpose of measuring the External network throughput, we only consider cases where two VMs are placed in two different Compute nodes, regardless of their virtual networks or routers. Compared to Tunnel network traffic, the throughput measured via the External interface (`em3`) are relatively stable and comparable between 3 types. As these VMs are connected via a physical bandwidth of 1 Gbps, the achieved throughput of around 940 Mbps results in a pretty high bandwidth utilisation (94%).

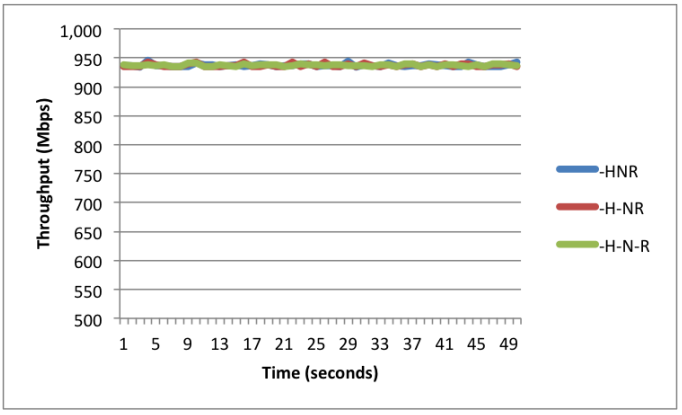

FIGURE 5.12: Network throughput with distributed router (inter-node traffic)

5.2.2.2 Latency measurement

Similar to throughput measurement, only VM pairs which are in DVR mode and have their peers in different Compute nodes are required to take part in the evaluation.

Network latency with distributed router

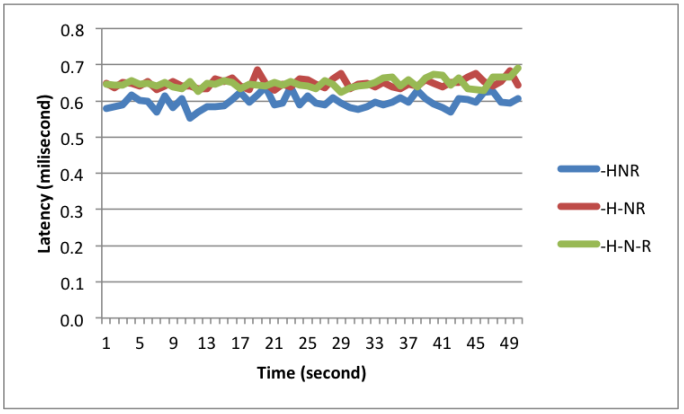

FIGURE 5.13: Network latency with distributed router

Figure 5.13 depicts the network latency with distributed router enabled. As can be seen from the figure, if two VMs are in the same network and same router, it is straightforward for the packets to be sent from one VM to another. It is thus reasonable to see -HNR (different hosts, same network, same router) has the lowest delay with the ping. On the other hand, when VMs fall into different networks and/or get plugged into different routers then a routing effort is needed. This makes two latter cases (-H-NR and -H-N-R) have slightly longer packet delay.

5.2.3 Observation

At this point when we have gone through the results of all the conducted experiments, the following points can be drawn:

- The highest achievable throughput belongs to VMs who are residing in the same Compute node and same tenant network. A data rate of 25 Gbps is beyond both the External and Tunnel network interface theoretical bandwidth (1 Gbps and 10 Gbps, respectively). This data rate is only second to the data rate of a VM when it sends data to itself (51 Gbps).
- The kernel module pair KVM/VHostNet make a huge difference in network performance once they are enabled thanks to the direct kernel access

- Enabling distributed virtual routing helps to significantly improve network performance when traffic routing is required.
- When VMs are to send data via the Tunnel network, the achievable throughput is between 1.6 Gbps and 2.7 Gbps, which is less than 30% of the theoretical bandwidth. If we use the External interface, the utilisation percentage is, as discussed in previous section, 94%. It can be inferred that VXLAN encapsulation has certain impacts on the network performance of OpenStack.
- It is the physical distance and on which host a VM is placed that highly likely influences the network performance between VMs

Chapter 6

Further System Analysis

As Neutron heavily influences the network performance of OpenStack, it is worth analysing its performance and properly understanding what actually happens with the system CPUs under the hood. This section expands the discussion on the results from previous chapter, with further analysis based on CPU profiling.

6.1 CPU Usage

Figure 6.1 illustrates a Compute node's CPU usage measured when the node itself runs `iperf` in client and when one guest machine runs `iperf` in client mode. As an `iperf` client, the Compute node continuously generates a large amount of data to send out and consumes approximately 25% CPU usage. Meanwhile, a VM doing similar things consumes 170% CPU usage.

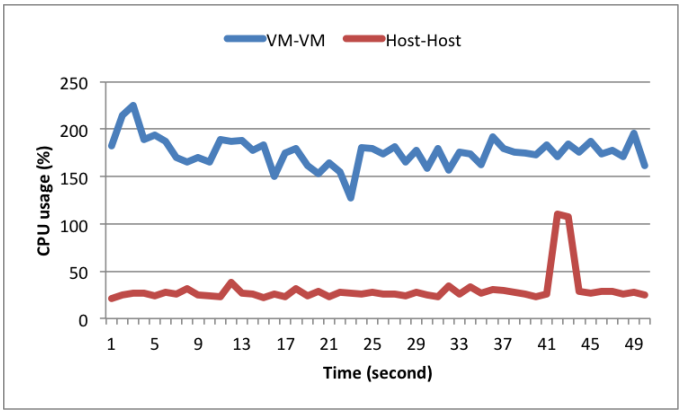

FIGURE 6.1: CPU usage of Compute node for data sending of host and guest machine

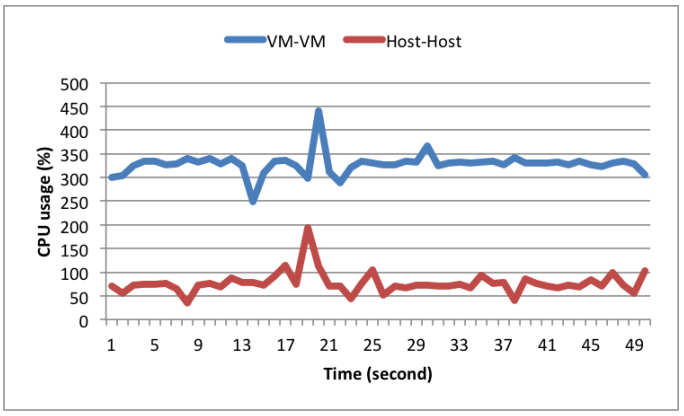

FIGURE 6.2: CPU usage of Compute node for data receving of host and guest machine

Similarly, Figure 6.2 compares CPU usage of a Compute node between when the node itself receives data and when a guest machine receives data (as running `iperf` in server mode). As presented in the figure, the host spends roughly 70% of CPU usage while the guest machine's consumption is almost 5 times higher, at approximately 350% CPU usage. At this point, it can safely be assumed that it costs significantly more computing resources to receive data than to send them, especially with VMs.

Given that the host system has 16 physical cores with hyperthreading enabled, making a total of 32 virtual CPUs (or 3200%, to be quantified as CPU usage) and total CPU consumption of other irrelevant processes being negligible (less than 5%), it takes full computing resources of almost 2 CPUs to send data, and of more than 3 CPUs to receive data.

6.2 CPU Profiling

This section uses the `perf` tool to analyse the CPU utilisation with regards to the number of CPU cycles rather than the usage percentage (of total 32 CPUs) as in section 6.1. `perf` essentially bases its CPU profiling on the counting of events that occurred when functions are invoked. As such CPU consumption is calculated based on the CPU cycles one process consumes over the total CPU cycles of the host system as a whole during the designated network performance evaluation.

6.2.1 Receiving Data

CPU statistic for data sending is calculated from 1,000,000 samples of "cpu-cycles" event, with an event count of approximately 140,869,100,000.

When the host machine receives the stream of data, it can be observed, from cpu statistic, that three most resource-intensive threads include `ksoftirqd`, `vhost-xxxxx` (with xxxxx being a particular process ID for the `vhostnet` module) and `qemu-system-x86`. While the `qemu-system-x86` (with KVM acceleration enabled) is the hypervisor platform and already consumes a lot of CPU usage, it is not the most resource-intensive and accounts for only 3% of total CPU cycles . Instead, it is the `ksoftirqd` and the `vhost-net` that account for 50.59% and 39.79% of the total CPU cycles. These two are the main reasons behind the surprisingly high CPU usage as discussed in the previous section.

`ksoftirqd` is a per-CPU kernel thread that runs when the machine is under heavy software-interrupt load. In OVS-based networking, the process of delivering a VXLAN-encapsulated packets to recipient guest machines takes several steps, including two times of injecting the packet into the host's network stack. As a software interrupt is scheduled right when a packet is fed into network stack, for each and every received VXLAN-encapsulated packet there are at least two times software interrupt occurs. In fact, according to the profiling results, it can be seen that the function `run_ksoftirqd` which triggers a soft interrupt is called many times, especially when the `iptables` is consulted (function `ipt_do_table` with 14.28%) or when data is copied into memory (function `memcpy` with 5.45%).

The second most resource-hungry thread is created by the kernel module `vhost-net`, which can enhance the guest networking performance by moving packets between the host and the guest environments (i.e. VM and Computer node) using Linux kernel rather than QEMU. This helps to avoid context switches between user-space and kernel space, which can improve overall performance including higher throughput and lower latency. As the VM is in data receiving mode, it is not surprising to see the `vhost-net` thread has to spend considerable processing power to handle a terribly large amount of data. In particular, `vhost-net` makes call to functions that perform copying received data (`copy_user_enhanced_fast_string` with 10.61%) or iptables policy look-up (`ipt_do_table` with 10.32%).

As the hypervisor platform for OpenStack, the `qemu-system-x86` surely gets invoked every time a VM performs an operation. As the Compute node has Intel processor with Intel VT-x architecture, it allows the use of Linux `kvm.o` module as an interface for QEMU to access CPU to run the virtualised resources. As such, the main job of `qemu-system-x86` is to invoke KVM module functions like `vcpu_enter_guest` or `vmx_vcpu_run`.

6.2.2 Sending Data

CPU statistic for data sending is calculated from 755,000 samples of "cpu-cycles" event, with an event count of approximately 75,523,300,000.

When a VM is dedicated to data sending, CPU consumption seems less than when it is in receiving mode (according to section 6.1). From the profiling result, the thread created by `vhost-net` module accounts for 75,91% of total CPU cycles consumed. Going down the calling stack, we would see that kernel-level functions like `__copy_skb_header` or `copy_user_enhanced_fast_string` are the hottest points (most commonly invoked).

6.2.3 Discussion

From the previously observed results and profiling details, we come up with the following points:

- Firstly, as already seen in the experiment results from the Chapter 5, the VMs network performance measured in cases with (1) VXLAN encapsulation (VM's Tunnel network traffic) is far worse than (2) those without encapsulation (VM's External network traffic) or (3) those with neither virtualisation nor encapsulation (host-to-host traffic). Specifically, while (2) and (3) can utilise approximately 95% of the physical bandwidth, that of (1) remains below 30%. As such we can assume that (VXLAN) encapsulation potentially contributes to the degradation of network.

- Secondly, as discussed in Section 6.2.1, each time a VXLAN packet arrives at the Compute host, it needs to be fed into the network stack of the host system two times. This implies that the processing effort (iptables look-up, memory-copy,

etc.) is doubled. The Linux's soft-interrupt is scheduled each time the packet is about to go up the network stack, which means that host system is under really heavy load. Having to process a large volume of data sent by `iperf`, the host CPU soon gets saturated. Thus, it is the VXLAN processing, especially when receiving encapsulated packets, that makes host system spend a considerably high amount of CPU resource.

- Thirdly, besides packet processing, CPU throttling is also contributed by the `vhost-net` module that copies a large stream of data from user-space into kernel-space. This is what mainly happens to a sending VM, but it is clear that without VXLAN packet processing, it would take much less CPU resource (as seen from the difference between sending and receiving data of VM, presented in Figure 6.1 and Figure 6.2)

From the 3 above mentioned points, we argue that VXLAN packet processing greatly contributes to the surprisingly high CPU usage experienced during the network performance evaluation. Since a network-related operation is normally single-threaded (we can actually see that there is only one thread for each of the invoked modules like `vhost-net`, or `ksoftirqd`), once a single CPU is saturated, the network performance would be undoubtedly throttled. It is also reported in other studies [22] that using VXLAN would add certain overhead to CPU, especially when the VXLAN Tunnelling EndPoint is completely software-based implementation.

To sum up, VXLAN could be an an effective solution for Neutron to implement network virtualisation, owing to its ability to isolate tenant networks within OpenStack environment. However this tunnelling standard in fact holds back the network performance with the burden of packet encapsulation/de-encapsulation suffered by the host system. Furthermore, the nature of the experiments, generating a stream of data using `iperf`, has forced the host system to saturate its processing resource. Without any support from hardware (e.g. *Network Interface Card*) to aid the specific-type (e.g. VXLAN) packet processing, the overheads will heavily influence system's CPUs. Consequently the network performance will be degraded.

Chapter 7

Conclusion

The OpenStack project has been on the market for more than 5 years, delivering one of the most successful open-source software platforms to deploy Cloud Computing. The software itself has been gradually accommodated with increasing development efforts from the community to be more stable and to have more features so as to satisfy growing needs from Cloud providers and users. As OpenStack is a relatively new and still growing cloud computing solution, it is of utmost importance to learn and evaluate its core technologies and performance so that a correct understanding over the platform can be achieved before we reach further improvement and/or deployment at a larger scale.

This thesis work aims at providing a big picture of OpenStack in general and placing its focus on the networking module - *Neutron* - in particular. This study comes up with, beyond a high-level understanding of the software architecture, a detailed deployment strategy along with a properly planned experimental and evaluative methodology in order to give an insightful observation on OpenStack operation. We are able to present the traffic patterns and the correspondingly measured network performance (in terms of throughput and latency) under the Neutron-based architecture. According to the experiment results, while packet encapsulation guarantees network isolation in the OpenStack Cloud environment, its bandwidth utilisation is limited to below 30% of the underlying physical channel. Further system analysis based on CPU profiling indicates that the use of VXLAN encapsulation potentially causes CPUs to throttle and thus degrade the network performance.

7.1 Future Work

As a growing open-source platform, OpenStack leaves rooms for a plenty of improvements, be they feature-wise or performance-wise. Although the lists of development issues are maintained by the OpenStack community itself, we recognise from this study that certain potential enhancements could be made to aid OpenStack generally and Neutron particularly function better.

One of the performance issues observed from the study rises from the fact that VMs belonging to the same tenant network are placed in different physical hosts. Current OpenStack schedulers (which is out of this thesis's scope) mostly consider the balance of hardware resources (CPU, memory, disk space) when selecting which Compute node to initialise a new VM. A potential network-aware scheduler would take into account the physical distance and connection such that VMs provisioned to the same Cloud tenant (thus belong normally to the same virtual network) can benefit from the intra-host traffic.

Besides, reducing the overheads of encapsulation is another issue worth considering. Current OpenStack relies on network overlay standards like VXLAN, which consumes significant processing power and thus reduces system performance as discussed in section 6. Offloading such encapsulated packet processing effort to a hardware-assisted module would help a great deal.

7.2 A Final Thought

OpenStack is on its way to potentially be a leading open-source Cloud platform. From the technological point of view, the value of OpenStack does not lie in any new technologies that it brings, just as the Cloud Computing paradigm itself. Instead, OpenStack is able to gather a variety of existing standards and technologies into a single software system that ease the headaches of Cloud service providers in deployment and management of a full-featured Cloud Computing cluster or data centre. This thesis provides the author with an opportunity to learn about a variety of technologies and standards prior to the understanding of OpenStack's philosophy itself.

As a collection of various technologies collaborating to create a Cloud platform altogether, OpenStack has made itself is a very complex system. Over time, it has been learnt that OpenStack is susceptible to various sources of performance problems that are hard to diagnose. From the measurements, we have experienced that performance is heavily affected by many different factors and that, while our study contributes to explore performance in some conditions, the study is not exhaustive and applies to the configuration that we used. Repeating the experiments after the servers had experienced some software and configuration changes showed performance variations. However, the methodology is a general contribution and a similar analysis of performance in different settings can then be repeated.

Appendix A

Installation and Configuration

A.1 Vagrant script for OpenStack installation on AWS

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"


nodes = {
    "controller"  => [1, 200],
    "network"  => [1, 201],
    "compute"  => [2, 202],
    #"cinder"   => [1, 211],
    #"test"    => [1, 222],
}


Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
    config.vm.box = "dummy"
    config.vm.box_url = "https://github.com/mitchellh/vagrant-aws/raw/master/dummy.box"
    config.vm.synced_folder "../juno_aws", "/vagrant", id: "vagrant-root"

    nodes.each do |prefix,(count, ip_start)|
        count.times do |i|
            if prefix == "compute"
```

```
        hostname = "%s-%02d" % [prefix, (i+1)]
    else
        hostname = "%s" % [prefix, (i+1)]
    end

    config.vm.define "#{hostname}".to_sym do |aws2|
        aws2.vm.hostname = "somename"

        aws2.vm.provider :aws do |aws, override|
            aws.access_key_id = ENV["ACCESS_KEY_ID"]
            aws.secret_access_key = ENV["SECRET_ACCESS_KEY"]
            aws.keypair_name = ENV["KEYPAIR_NAME"]

            aws.ami = "ami-5189a661" #Ubuntu 14.04 LTS # ami-3d50120d

            if prefix == "network" or prefix == "compute" or prefix == "controller"
                aws.instance_type = "t2.medium"
            end

            if prefix == "cinder" or prefix == "test"
                aws.instance_type = "t2.micro"
            end

            if prefix == "test"
                aws.private_ip_address =  "172.31.21.#{ip_start+i}"
            else
                aws.private_ip_address =  "172.31.20.#{ip_start+i}"
            end

            # an IP for network
            #if prefix == "network"
            #   aws.elastic_ip = true
            #end

            aws.region = "us-west-2"
            aws.availability_zone = "us-west-2a"
            aws.security_groups = ["tab87vn_master_thesis"]

            #if prefix == "controller"
            aws.block_device_mapping = [{ "DeviceName" => "/dev/sda1", "Ebs.VolumeS
            #end
```

```
                override.ssh.username = "ubuntu"
                override.ssh.private_key_path = ENV["MY_KEY"]
                aws.tags = {
                  "user" => "tab87vn-node#{i}"
                }
            end
            #aws2.vm.provision :shell, :path => "#{prefix}.sh"
        end
    end
end
```

A.2 OpenStack Module Configurations

Nova

```
[DEFAULT]
dhcpbridge_flagfile=/etc/nova/nova.conf
dhcpbridge=/usr/bin/nova-dhcpbridge
logdir=/var/log/nova
state_path=/var/lib/nova
lock_path=/var/lock/nova
root_helper=sudo nova-rootwrap /etc/nova/rootwrap.conf
verbose=True

use_syslog = True
syslog_log_facility = LOG_LOCAL0

api_paste_config=/etc/nova/api-paste.ini
enabled_apis=ec2,osapi_compute,metadata

# Libvirt and Virtualization
libvirt_use_virtio_for_bridges=True
connection_type=libvirt
#libvirt_type=${LIBVIRT}
compute_driver = libvirt.LibvirtDriver
libvirt_type=kvm
```

```
# Database
sql_connection=mysql://nova:openstack@${MYSQL_HOST}/nova

# Messaging
rabbit_host=${MYSQL_HOST}

# EC2 API Flags
ec2_host=${MYSQL_HOST}
ec2_dmz_host=${MYSQL_HOST}
ec2_private_dns_show_ip=True

# Network settings
network_api_class=nova.network.neutronv2.api.API
neutron_url=http://${CONTROLLER_HOST}:9696
neutron_auth_strategy=keystone
neutron_admin_tenant_name=service
neutron_admin_username=neutron
neutron_admin_password=neutron
neutron_admin_auth_url=https://${KEYSTONE_ENDPOINT}:5000/v2.0
libvirt_vif_driver=nova.virt.libvirt.vif.LibvirtHybridOVSBridgeDriver
linuxnet_interface_driver=nova.network.linux_net.LinuxOVSInterfaceDriver
#firewall_driver=nova.virt.libvirt.firewall.IptablesFirewallDriver
security_group_api=neutron
firewall_driver=nova.virt.firewall.NoopFirewallDriver
neutron_ca_certificates_file=/etc/ssl/certs/ca.pem

service_neutron_metadata_proxy=true
neutron_metadata_proxy_shared_secret=foo

#Metadata
metadata_host = ${CONTROLLER_HOST}
metadata_listen = ${CONTROLLER_HOST}
metadata_listen_port = 8775

# Cinder #
volume_driver=nova.volume.driver.ISCSIDriver
enabled_apis=ec2,osapi_compute,metadata
volume_api_class=nova.volume.cinder.API
iscsi_helper=tgtadm
iscsi_ip_address=${CINDER_ENDPOINT}
```

```
# Images
image_service=nova.image.glance.GlanceImageService
glance_api_servers=${GLANCE_HOST}:9292

# Scheduler
scheduler_default_filters=AllHostsFilter

# Auth
auth_strategy=keystone
keystone_ec2_url=https://${KEYSTONE_ENDPOINT}:5000/v2.0/ec2tokens

# NoVNC
novnc_enabled=true
novncproxy_host=${CONTROLLER_HOST}
novncproxy_base_url=http://${CONTROLLER_HOST}:6080/vnc_auto.html
novncproxy_port=6080
#
xvpvncproxy_port=6081
xvpvncproxy_host=${CONTROLLER_HOST}
xvpvncproxy_base_url=http://${CONTROLLER_HOST}:6081/console

vnc_enabled = True
vncserver_proxyclient_address=${MNG_IP}
#vncserver_proxyclient_address=${EXT_IP}
vncserver_listen=0.0.0.0

[keystone_authtoken]
admin_tenant_name = ${SERVICE_TENANT}
admin_user = ${NOVA_SERVICE_USER}
admin_password = ${NOVA_SERVICE_PASS}
identity_uri = https://${KEYSTONE_ADMIN_ENDPOINT}:35357/
insecure = True
```

Neutron

```
[DEFAULT]
verbose = True
debug = True
```

```
state_path = /var/lib/neutron
lock_path = \$state_path/lock
log_dir = /var/log/neutron

bind_host = 0.0.0.0
bind_port = 9696

# Plugin
core_plugin = ml2
service_plugins = router
allow_overlapping_ips = True
#router_distributed = True
#dvr_base_mac = fa:16:3f:01:00:00

# auth
auth_strategy = keystone
nova_api_insecure = True

# RPC configuration options. Defined in rpc __init__
# The messaging module to use, defaults to kombu.
rpc_backend = neutron.openstack.common.rpc.impl_kombu

rabbit_host = ${CONTROLLER_HOST}
rabbit_password = guest
rabbit_port = 5672
rabbit_userid = guest
rabbit_virtual_host = /
rabbit_ha_queues = false

# ============ Notification System Options =====================
notification_driver = neutron.openstack.common.notifier.rpc_notifier

[agent]
root_helper = sudo

[keystone_authtoken]
auth_host = ${KEYSTONE_ADMIN_ENDPOINT}
auth_port = 35357
auth_protocol = https
admin_tenant_name = ${SERVICE_TENANT}
admin_user = ${NEUTRON_SERVICE_USER}
```

```
admin_password = ${NEUTRON_SERVICE_PASS}
signing_dir = \$state_path/keystone-signing
insecure = True

[database]
connection = mysql://neutron:${MYSQL_NEUTRON_PASS}@${CONTROLLER_HOST}/neutron
```

ML2 Plugin

```
[ml2]
type_drivers = gre,vxlan
tenant_network_types = vxlan
mechanism_drivers = openvswitch,l2population

[ml2_type_gre]
tunnel_id_ranges = 1:1000

[ml2_type_vxlan]
vni_ranges = 1:1000

[agent]
tunnel_types = vxlan
l2_population = True
enable_distributed_routing = True
arp_responder = True

[ovs]
local_ip = ${ETH1_IP}
tunnel_type = vxlan
enable_tunneling = True
l2_population = True
enable_distributed_routing = True
tunnel_bridge = br-tun

[securitygroup]
firewall_driver = neutron.agent.linux.iptables_firewall.OVSHybridIptablesFirewallDriver
enable_security_group = True
```

OpenvSwitch

```
# OpenVSwitch Configuration

#br-int will be used for VM integration
sudo ovs-vsctl add-br br-int

# Neutron Tenant Tunnel Network
sudo ovs-vsctl add-br ${VMN_BR}
sudo ovs-vsctl add-port ${VMN_BR} ${VMN_IF}



sudo ifconfig ${VMN_IF} 0.0.0.0 up
sudo ip link set ${VMN_IF} promisc on
# Assign IP to br-eth2 so it is accessible
sudo ifconfig ${VMN_BR} ${VMN_IP} netmask 255.255.255.0

# Neutron External Router Network
sudo ovs-vsctl add-br ${EXT_BR}
sudo ovs-vsctl add-port ${EXT_BR} ${EXT_IF}
#

sudo ifconfig ${EXT_IF} 0.0.0.0 up
sudo ip link set ${EXT_IF} promisc on
# Assign IP to br-ex so it is accessible
sudo ifconfig ${EXT_BR} ${EXT_IP} netmask 255.255.255.0
```

A.3 Miscellaneous scripts

Collecting iperf results

```
#!/bin/bash
#sleep 12000
hosts=("11.0.0.18" "11.0.0.21" "12.0.0.5" "12.0.0.6")
for i in "${hosts[@]}"
do
    j="0"
```

```
    while [ $j -lt 10 ]
    do
            iperf -c $i -t 60 -i 1 >> "result_iperf_legacy_local_20160101.txt"
        j=$[$j+1]
    done
done
# external interface
sleep 10
hosts=("10.0.100.102" "10.0.100.103" "10.0.100.104"
        "10.0.100.105" "10.0.100.116" "10.0.100.117")
#hosts=("10.0.100.116" "10.0.100.117")
for i in "${hosts[@]}"
do
        j="0"
        while [ $j -lt 10 ]
        do
                iperf -c $i -t 60 -i 1 >> "result_iperf_legacy_ext_20160101.txt"
                j=$[$j+1]
        done
done
```

Regular Expression to filter ping/iperf output

```
64 bytes from [0-9]+\.[0-9]\.[0-9]\.[0-9]+: icmp_seq=[0-9]+ ttl=[0-9]+ time=
\[[\s]+[0-9]\]\s[0-9]+\.0\-[0-9]+\.0 sec[\s]+([0-9]+|[0-9]\.[0-9]+)[\s]+(MBytes|GBytes)[\s]
```

Bibliography

[1] Layer 3 networking in neutron. URL http://docs.openstack.org/developer/neutron/devref/layer3.html. [Online; accessed 27-July-2015].

[2] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lidner. A break in the clouds: Towards a cloud definition. [Online; accessed 27-July-2015].

[3] A. Kalapatapu and M. Sarkar. Cloud computing: An overview. [Online; accessed 20-December-2014].

[4] Grance Mell. The NIST Definition of Cloud Computing. URL http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. [Online; accessed 05-December-2014].

[5] OpenStack, . URL https://www.openstack.org. [Online; accessed 27-July-2015].

[6] National Aeronautics and Space Administration. URL http://www.nasa.gov. [Online; accessed 20-July-2015].

[7] Rackspace Hosting. URL https://www.openstack.org. [Online; accessed 27-July-2015].

[8] Openstack releases, . URL https://wiki.openstack.org/wiki/Releases. [Online; accessed 27-July-2015].

[9] Openstack installation guide, . URL http://docs.openstack.org/juno/install-guide/install/apt/openstack-install-guide-apt-juno.pdf. [Online; accessed 21-November-2015].

[10] Kernel-based Virtual Machine. URL http://www.linux-kvm.org/. [Online; accessed 215-November-2014].

[11] VMware. URL https://www.vmware.com/. [Online; accessed 20-December-2014].

[12] Xen project. URL http://www.xenproject.org/. [Online; accessed 20-December-2014].

[13] Hyper-V. URL https://en.wikipedia.org/wiki/Hyper-V/. [Online; accessed 20-December-2014].

[14] Linux Containers. URL https://linuxcontainers.org/lxc/introduction/. [Online; accessed 20-July-2015].

[15] Openstack operations guide, . URL http://docs.openstack.org/openstack-ops/openstack-ops-manual.pdf. [Online; accessed 21-November-2015].

[16] RabbitMQ. URL https://www.rabbitmq.com/. [Online; accessed 27-July-2015].

[17] Advanced Message Queue Protocol. URL http://www.amqp.org/. [Online; accessed 27-July-2015].

[18] Iptables. URL https://en.wikipedia.org/wiki/Iptables. [Online; accessed 19-February-2015].

[19] Dnsmasq. URL http://www.thekelleys.org.uk/dnsmasq/doc.html. [Online; accessed 26-July-2015].

[20] Open vSwitch. URL http://openvswitch.org. [Online; accessed 27-July-2015].

[21] [RFC7348] M. Mahalingam. Virtual extensible local area network (vxlan) A framework for overlaying virtualized layer 2 networks over layer 3 networks. URL https://tools.ietf.org/html/rfc7348. [Online; accessed 27-July-2015].

[22] Andre Pech. Running OpenStack over a VXLAN Fabric. URL https://www.openstack.org/assets/presentation-media/OpenStackOverVxlan.pdf. [Online; accessed 20-December-2015].

[23] Iperf. URL https://iperf.fr/. [Online; accessed 27-November-2014].

[24] perf: Linux profiling with performance counters. URL https://perf.wiki.kernel.org/index.php/Main_Page. [Online; accessed 21-December-2014].

[25] Tcpdump. URL http://www.tcpdump.org/. [Online; accessed 27-July-2015].

[26] Wireshark. URL https://www.wireshark.org/. [Online; accessed 27-July-2015].

[27] Devstack - An Openstack Community Production. URL http://docs.openstack.org/developer/devstack/. [Online; accessed 19-October-2014].

[28] Amazon Web Services, . URL http://aws.amazon.com. [Online; accessed 28-November-2014].

[29] OpenStack, . URL https://aws.amazon.com/ec2/instance-types/. [Online; accessed 28-November-2015].

[30] E. Sigler K. Jackson, C. Bunch. OpenStack Cloud Computing Cookbook. URL http://openstackcookbook.com/. [Online; accessed 27-July-2015].

[31] Vagrant, . URL https://www.vagrantup.com. [Online; accessed 27-July-2015].

[32] Use vagrant to manage your ec2 and vpc instances, . URL https://github.com/mitchellh/vagrant-aws. [Online; accessed 27-July-2015].

[33] boto: A Python interface to Amazon Web Services. URL https://boto.readthedocs.org/en/latest/. [Online; accessed 25-March-2015].

[34] OpenFlow, . URL https://www.openstack.org. [Online; accessed 27-July-2015].

[35] Ryu SDN Framework. URL http://osrg.github.io/ryu/. [Online; accessed 15-December-2015].