Universidade de Aveiro
2014

Departamento de Eletrónica,
Telecomunicações e Informática

Igor
Duarte Cardoso

Controlo de Infrastrutura de Rede para Campus Virtuais

Network Infrastructure Control for Virtual Campuses

Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2014

Igor Duarte Cardoso

Controlo de Infrastrutura de Rede para Campus Virtuais

Network Infrastructure Control for Virtual Campuses

"*We convince by our presence*"

— Walt Whitman

Universidade de Aveiro
2014

Departamento de Eletrónica,
Telecomunicações e Informática

Igor
Duarte Cardoso

Controlo de Infrastrutura de Rede para Campus Virtuais

Network Infrastructure Control for Virtual Campuses

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor João Paulo Silva Barraca, Professor assistente convidado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Rui Luís Andrade Aguiar, Professor associado c/ agregação do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Dedico este trabalho ao meu Pai, Mãe, namorada e amigos.

o júri / the jury

presidente / president

Prof. Doutor Osvaldo Manuel da Rocha Pacheco
professor auxiliar da Universidade de Aveiro

vogais / examiners committee

Doutor Pedro Miguel Naia Neves
investigador na PT Inovação e Sistemas

Doutor João Paulo Silva Barraca
assistente convidado na Universidade de Aveiro

agradecimentos /
acknowledgements

Agradeço aos orientadores da dissertação, ao colaborador da mesma, Prof. Doutor Diogo Gomes, ao Carlos Gonçalves pelas discussões no âmbito deste trabalho, e ao Filipe Manco pelo trabalho antecedente a este.

Palavras Chave

cloud computing, virtualização de redes, redes heterogéneas, redes legadas, nfv

Resumo

Esta dissertação apresenta uma forma de juntar infrastruturas Cloud Computing com redes tradicionais ou legadas, trazendo o melhor de ambos os mundos e possibilitando um controlo logicamente centralizado. Uma arquitetura é proposta com o intuito de extender implementações de Cloud Computing para que possam gerir também redes fora da infrastrutura de Cloud Computing, extendendo os segmentos de rede internos, virtualizados. Isto é útil para um variado conjunto de casos de uso, tais como migração incremental de redes legadas para a Cloud, redes híbridas virtuais/tradicionais, controlo centralizado de redes já existentes, aprovisionamento de *bare metal* e até mesmo a passagem de serviços tipicamente fornecidos por um *home gateway* para o lado do operador, melhorando o controlo e reduzindo custos de manutenção. Uma implementação da solução é apresentada e testada em cima do OpenStack, a principal solução Open-Source de Cloud Computing disponível. A implementação inclui alterações à API, à interface de linha de comandos e aos mecanismos já existentes, que apenas suportam implementações homogéneas, para que possam suportar qualquer equipamento e automatizar o aprovisionamento dos mesmos. Através daquilo que se chamam *drivers* externos, qualquer organização (seja um fabricante de equipamentos de rede, um fornecedor de Cloud ou uma operadora de telecomunicações) pode desenvolver o seu próprio drivers para suportar novos, específicos equipamentos de hardware. Para além da facilidade de desenvolvimento e extensibilidade, dois *drivers* são também fruto deste trabalho: um para *switches/routers* OpenWrt e outro para os módulos de *switching* Cisco EtherSwitch, sistema operativo IOS. Testes efetuados indicam que há baixas penalizações na latência e largura de banda, e ainda que os tempos de aprovisionamento são reduzidos em comparação com semelhantes operações de manutenção em redes informáticas tradicionais.

Keywords

cloud computing, network virtualization, heterogeneous networks, legacy networks, nfv

Abstract

This dissertation provides a way to merge Cloud Computing infrastructures with traditional or legacy network deployments, leveraging the best in both worlds and enabling a logically centralized control for it. A design/architecture is proposed to extend existing Cloud Computing software stacks so they are able to manage networks outside the Cloud Computing infrastructure, by extending the internal, virtualized network segments. This is useful in a variety of use cases such as incremental Legacy to Cloud network migration, hybrid virtual/traditional networking, centralized control of existing networks, bare metal provisioning and even offloading of advanced services from typical home gateways into the operator, improving control and reducing maintenance costs. An implementation is presented and tested on top of OpenStack, the principal Open-Source Cloud Computing software stack available. It includes changes to the API, command line interface and existing mechanisms which previously only supported homogeneous vendor equipment, such that they support any hardware and be able to automate their provisioning. By using what is called External Drivers, any organization (an hardware vendor, a Cloud provider or even a telecommunications operator) can develop their own driver to support new, specific networking equipment. Besides this ease of development and extensibility, two drivers are already developed in the context of this work: one for OpenWrt switches/routers and one for Cisco EtherSwitch IOS switching modules. Test results indicate that there are low penalties on latency and throughput, and that provisioning times (for setting up or tearing down networks) are reduced in comparison with similar maintenance operations on traditional computer networks.

Contents

List of Figures

List of Tables

Glossary

AAA	Authentication, Authorization and Accounting
AD	Attachment Device
AP	Access Point
API	Application Programming Interface
ARP	Address Resolution Protocol
ASIC	Application-Specific Integrated Circuit
BGP	Border Gateway Protocol
bps	bits per second
BRAS	Broadband Remote Access Server
CapEx	Capital Expenditure
CBSE	Component-Based Software Engineering
CIDR	Classless Inter-Domain Routing
CLI	Command Line Interface
CNa	Campus Network agent
CNc	Campus Network controller
CPE	Customer Premises Equipment
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
DevOps	Development and Operations
DHCP	Dynamic Host Configuration Protocol
DMZ	Demilitarized Zone
DNAT	Dynamic Network Address Translation (NAT)
DNS	Domain Name Server
DPI	Deep Packet Inspection
DSL	Digital Subscriber Line
DSLAM	DSL Access Multiplexer
EPE	External Port Extension
FCS	Frame Check Sequence
FOSS	Free and Open-Source Software
FWaaS	Firewall as a Service
GRE	Generic Routing Encapsulation
HA	High Availability
HGW	Home Gateway
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
ICMP	Internet Control Message Protocol
IoT	Internet of Things
IP	Internet Protocol
IPC	Inter-Process Communication
IPFIX	IP Flow Information Export
IPsec	Internet Protocol Security
IT	Information Technology
ISP	Internet Service Provider
JSON	JavaScript Object Notation
L1	OSI Layer 1
L2	OSI Layer 2
L2GW	Layer 2 Gateway
L3	OSI Layer 3
L7	OSI Layer 7
LAN	Local Area Network
FWaaS	Load Balancer as a Service
LLDP	Link Layer Discovery Protocol
MAC	Media Access Control
Mbps	Megabits per second

ML2	Modular Layer 2
MTU	Maximum Transmission Unit
NaaS	Network as a Service
NAP	Network Attachment Point
NAT	Network Address Translation
NEP	Network External Port
NFV	Network Function Virtualization
NIC	Network Interface Controller
NPC	Network Point Controller
NRP	Network Report Point
NSEP	Network Segment Extension Process
NVS	Network Virtualization Stack
OOP	Object-Oriented Programming
OpEx	Operating Expenditure
OS	Operating System
OSI	Open Systems Interconnection model
OSPF	Open Shortest Path First
OSS	Open-Source Software
OVS	Open vSwitch
OVSDB	Open vSwitch Database Management Protocol
P2P	Peer-to-Peer
PC	Personal Computer
PaaS	Platform as a Service
PM	Physical Machine
PoC	Proof of Concept
PTL	Project Technical Lead
QoS	Quality of Service
RAM	Random-Access Memory
RGW	Residential Gateway
RIP	Routing Information Protocol
RPC	Remote Procedure Call
SaaS	Software as a Service
SDN	Software-Defined Networking
SFC	Service Function Chaining
SNAT	Static NAT
SNMP	Simple Network Management Protocol
SOA	Service-Oriented Architecture
SSH	Secure Shell
SSID	Service Set Identifier
sTIC	Serviços de Tecnologias de Informação e Comunicação
TC	Technical Committee
TCG	Tiny Code Generator
TCP	Transmission Control Protocol
ToR	Top of Rack
UDP	User Datagram Protocol
UI	User Interface
vCPE	Virtual Customer Premises Equipment (CPE)
vHGW	Virtual Home Gateway (HGW)
VIF	Virtual Interface
VLAN	Virtual LAN
VM	Virtual Machine
vNIC	Virtual Network Interface Controller (NIC)
VPN	Virtual Private Network
VPNaaS	VPN as a Service
VTEP	VXLAN Tunnel End Point
VXLAN	Virtual Extensible LAN
XML	Extensible Markup Language
WAN	Wide Area Network

CHAPTER 1

INTRODUCTION

This work proposes a method for extending virtualized networks (provided to clients via, for instance, an Infrastructure as a Service (IaaS) Cloud Computing software stack) with segments that live outside the virtualized datacenter. There are multiple real-world use cases that thrive from such a feature: incremental migration from legacy networking to Cloud networking; private Clouds with critical legacy equipment; on-demand campus networking control and flexibility; home gateway services and control offloading; bare metal provisioning; service chaining to entities external to the Cloud; Virtual Private Network (VPN), amongst others. The biggest problem face to achieve these real-world use cases is the fact that most of them make use of heterogeneous networking equipment, complicating matters. Consequently, heterogeneity support is a crucial characteristic and an always-present concern throughout the entire work. The end result is a working IaaS stack (having network resources - Network as a Service (NaaS)) with the capability of extending virtual networks to the outside world, all controllable via an a logically centralized API which leverages existing Cloud advantages like multi-tenancy as well as the features provided by this work. Because this "outside world" is heterogeneous by nature, a way of dealing with it is also presented, thus leveraging many benefits regarding legacy networking equipment. To achieve the defined objectives, an abstract way of dealing with all disparate components of such a scenario is addressed and a reference implementation on top of OpenStack is presented and tested.

1.1 MOTIVATION

Recent trends, like Cloud Computing, have invaded the market and made the deployment of services easier and more flexible. Furthermore, service-hosting providers have found a way to simplify their datacenters by recurring to this new concept. Even Telecommunications Operators increasingly make use of Cloud Computing with the purpose of simplifying tasks and operations and eventually decrease time-to-market of new services and products. When Cloud Computing makes use of Network Virtualization to provide advanced Network features to customers, a whole new set of use cases can start to be explored. Cloud tenants or administrators are now able to design and implement their own network topologies, with considerable flexibility, and attach them to their own virtual machines. This is true both for private, public or hybrid Clouds.

What happens nowadays for such Cloud Computing stacks is that, to achieve the desired level of elasticity and ease of maintenance, underlying network resources to be virtualized need to be as

homogeneous as possible. As is detailed furthermore in this work, homogeneity of resources by itself can, indeed, alleviate a set of problems improve efficiency in multiple aspects. However, it also limits what kind of use cases can be fulfilled by the Cloud Computing provider, in comparison to a deployment consisting of heterogeneous kinds of network resources.

As a result, Cloud Computing may actually be dropped altogether in favor of a more traditional deployment, depending on the set of requirements presented by the client, effectively dismissing modern technology in favor of something else.

Given these premises, this work tries to find the midpoint between typical Cloud Computing infrastructures with NaaS based on homogeneous networking resources and traditional network deployments that rely on heterogeneous networking equipment (different types, brands, models). An example of this heterogeneity can be found at the University of Aveiro.

1.1.1 UNIVERSITY OF AVEIRO

The University of Aveiro includes over 30 departments spread over a campus with 65 buildings and a total area of around 92 football fields [1]. Participation in research projects, either local, in collaboration with other national entities, or in the context of European projects, is very active. The number of students, faculty members and various kinds of staff is around a couple tens of thousands. Multiple kinds of administrative services, from academic and financial ones to social support must be supported. Following the considerably large number of departments, also comes a large amount of different classes, some of them with special requirements, e.g. networking equipment in a computer networks lab. In addition, people can access the Internet and internal services provided by the university's network infrastructure through two main networking interfaces: either via Ethernet ports or the Eduroam wireless infrastructure deployed across the campus, each one providing different access rights depending on predetermined characteristics of the host or user.

For these and other reasons (consider, e.g., all requirements imposed by videoconferencing services), the core network infrastructure must have attributes that guarantee reliable levels of stability, monitoring, controllability and flexibility. With such an heterogeneous deployment and a plenitude of services that will mostly grow and increase, and factors which include more and greater research challenges, the attributes mentioned become increasingly harder to leverage.

Regarding actual background on the university's networking infrastructure and equipment, it can be estimated that around 4500 Personal Computers (PCs) are deployed across the university's domain. Besides that, there are about 200 servers, both virtual and physical, again demonstrating the heterogeneity of the network. These servers support the service portfolio in the university's for education, research and administration purposes. At a lower level, the different departments are connected to the network infrastructure via optical links. Inside these departments, traffic is distributed mostly recurring to Virtual LANs (VLANs), provided via switches and Access Points (APs).

Making a change in the current networking infrastructure is not an easy task because it is complex and lacks flexibility. Taking, for instance, computer networking labs, there is a great difficulty in preparing lessons due to the underlying network and administrative tasks that may need to be carried out. It may be needed to ask the central IT services department, Serviços de Tecnologias de Informação e Comunicação (sTIC), to proceed with specific network changes for special computer networking lab classes. More importantly, research testbeds are also affected in terms of the previous points mentioned

[1] `http://www.ua.pt/campusdaua`

and also in terms of their reach. Usually, these testbeds will be carried out over an isolated network, so as to prevent any threat to the network connectivity and stability at the university's production network.

Such an ossified infrastructure, which is present nowadays in most entities and inclusively affecting the Internet as well (although its ossification lies in other aspects), is an obstacle that requires a non-trivial undertaking.

1.2 OBJECTIVES

The principal objectives of this work can be summarized in three parts.

Firstly, existing Cloud Computing software stacks, specifically those with NaaS, or another kind of networking virtualization software stack with enhanced features, aim to be integrated with the work presented throughout this document. In other words, there is a clear objective of keeping features already provided by these software stacks while leveraging new ones by this work. Thus, this works aims to extend on existing work to provide a better, more complete solution as a whole, with the best of two worlds.

Secondly, there is the core focus of work. The objective here is to actually implement what is called an Network Segment Extension Process (NSEP), which consists in making it possible to extended a network segment (usually virtual and deployed inside a Cloud Computing / Network Virtualization software stack) with another network segment that can live anywhere outside the first deployment. These network segments can live "at the other side of the Internet".

Thirdly, the always-present objective of keeping heterogeneity a core property. Without this property, the work essentially becomes impossible to deploy in the real world, precluding the accomplishment of any use case that justifies its own development.

1.3 CONTRIBUTIONS

This work led to the re-submission of a blueprint initially developed by Filipe Manco [1], named "Provider Router Extension", with contents significantly changed to align with a new vision for implementation and deployment, which eventually molded into the work presented in this dissertation. The changed provider-router blueprint sent for review (prior to converting to the newly-recommended Spec format [2]) in OpenStack is available at the appendix A.

Another contribution, actually accepted to upstream, was made to the OpenStack Documentation project, with new steps on how to upgrade releases from Havana to Icehouse, including Neutron and ML2: two critical parts of OpenStack for the development of this work.

Moreover, one person from Cisco [3] showed initial interest in this work and one person from Big Switch Networks [4] started developing a similar work, although trying to leverage a broader set of objectives whilst not being concerned with heterogeneity. Future contribution based on this dissertation to Big Switch Network's work has been accepted by its author. A new work from HP is now starting

[2] `https://wiki.openstack.org/wiki/Blueprints`

[3] `http://www.cisco.com/`

[4] `http://www.bigswitch.com/`

to be defined, which has some similarities with the previous work by Big Switch Networks, and this dissertation's author has begun participating in discussions with the intent of leveraging the work currently being specified by HP [5] with some of the advantages of the work presented in this dissertation. One Convergence [6] and PT Inovação e Sistemas [7] have also demonstrated interest in this work.

1.4 STRUCTURE

In order to familiarize the reader with the most important broad concepts throughout this document, the next chapter (Chapter 2) presents the background necessary. Inside, both conceptual and technical background can be found, the latter mainly about OpenStack. Afterwards, Chapter 3 provides a view on State of the Art technology and developments, and discusses what they fail to achieve taking into account the work presented in this dissertation. Having all background and State of the Art technology been presented, Chapter 4 is introduced, dedicated to stating some problems faced today, taking into account State of the Art technology as well, and what new use cases could be leveraged by overcoming these problems and obstacles. Chapter 5 starts defining the solution, as a generic design, to attain or at least ease the materialization of the use cases previously explored. With a complete design specified, Chapter 6 maps it to an implementation, based on the IaaS Cloud Computing OpenStack software stack, and defines multiple aspects, properties and components regarding it. With the implementation addressed, tests are carried out and results measured and discussed, as presented in Chapter 7. Finally, a conclusion is provided in the last chapter, Chapter 8, where future work is also discussed. Appendices and References come afterwards.

[5] http://www.hpcloud.com/
[6] http://www.oneconvergence.com
[7] http://www.ptinovacao.pt

CHAPTER 2

Background

2.1 Conceptual Background

2.1.1 Service-Oriented Architecture (SOA)

Service-Oriented Architecture is an architecture model that takes services as its main components. There is no single definition of SOA and no single governing standards body defining service-orientation. It is a concept originating from different sources: be them Information Technology (IT) organizations, vendors or consulting firms [2].

However, it can be traced back to a software engineering theory known as "separation of concerns". Other models have emerged from this same theory: Object-Oriented Programming (OOP) or Component-Based Software Engineering (CBSE), for example. The objective is to decompose a problem into a collection of smaller ones, each one addressing a specific concern.

A typical set of principles most associated with service-orientation, also based on [2], can be summarized as such:

- Services are reusable, even if there is no immediate opportunity for reusing them, so they should be designed with this requirement in mind;
- Services share a formal contract, so they are inherently prepared to interact with other services (the formal contract specifies an interface that describes exactly what is the service and how information exchange should occur);
- Services are loosely coupled, meaning that there should be no tight couplings and/or cross-service dependencies;
- Services abstract underlying logic, by sharing a formal contract and nothing else;
- Services are composable, so they may compose other services to create different levels of granularity while leveraging reusability and the creation of abstraction layers;
- Services are autonomous, meaning that they should be able to exist, whenever possible, as standalone and independent services;
- Services are stateless, because otherwise it would be difficult to remain loosely coupled, so state management should be deferred to another entity;

- Services are discoverable, meaning that they should provide enough information so that both humans and service requestors understand and make use of their logic.

2.1.2 CLOUD COMPUTING

Cloud Computing is a broad concept that has been conquering datacenters all around the world for the last few years. It used to be a daily "buzzword", even for non-technical users. Nowadays it still progresses even though the overall hype is lower than ever [3]. However, if expectations are correct, it is slowly becoming more interesting again.

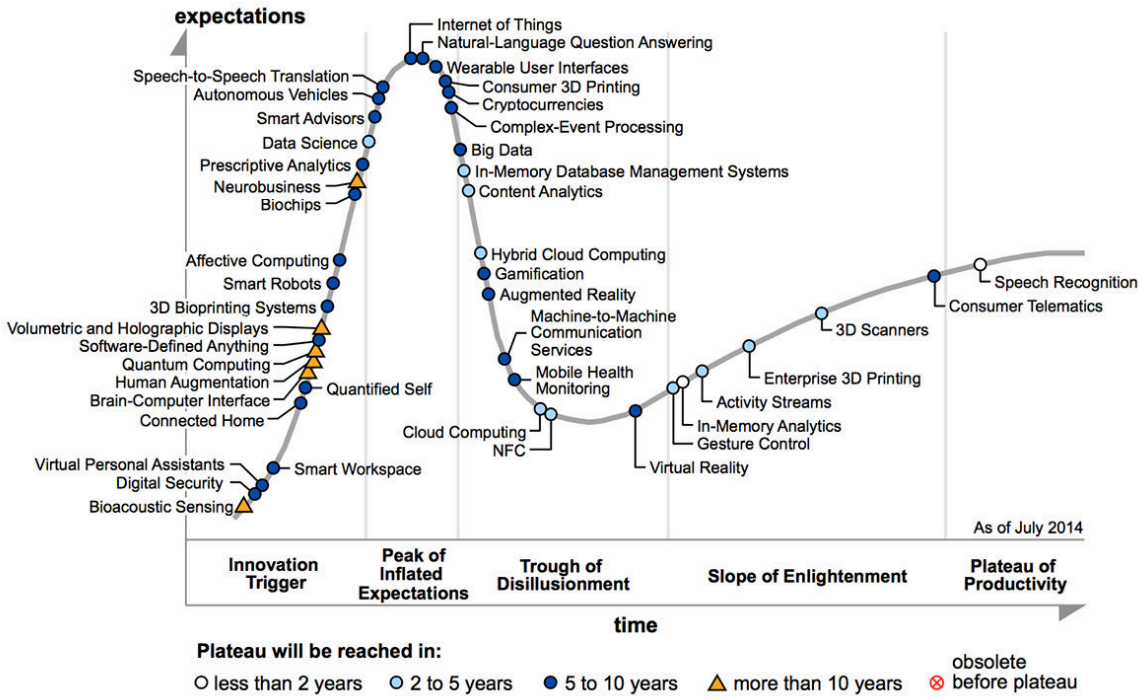


Figure 2.1: Hype Cycle for Emerging Technologies, 2014 [3] shows Cloud Computing is about to get more interesting.

One of the underlying concepts of Cloud Computing, resource offloading, can be traced back to the era of mainframes [4], when computing power and storage were remotely available. With the trends of ever more compact, more powerful and cheaper computers, they eventually became personal and the mainframe began its downfall.

More recently, the concept of Cloud Computing started emerging, bringing back some properties of the classic mainframe computing. Yet, it is much more than resource offloading. Per the NIST definition of Cloud Computing [5], it presents itself in different service and deployment models, and comprises multiple essential characteristics. A requirement that tightly follows Cloud Computing is the ability to consume resources on-demand in a self-service fashion, meaning that end users must be able to choose what resources they want to use, whilst not being concerned about where they are (somewhere in the "Cloud"), and how they will be provisioned to them. To achieve this, a SOA must be in place allowing the provisioning of these resources as services. Furthermore, these resources are served in a multi-tenant manner, where they are dynamically assigned to different users (tenants)

through the use of resource virtualization. Authentication, Authorization and Accounting (AAA) is usually also a part of Cloud Computing.

SOA is a part of the overall Cloud Computing infrastructure, necessary to carry out a plenitude of advantages regarding integration, interoperability and flexibility.

The infrastructure mentioned consists of a physical layer that joins together all resources and an abstraction layer that allows them to be provided taking into account essential characteristics.

The essential characteristics of Cloud Computing, according to the NIST definition, can be summarized as:

- On-demand self-service, or the ability for users to provision themselves with resources without requiring any other human interaction with service providers;
- Broad network access, or the ability for Cloud Computing services to be provided to any kind of client (mobile, workstations, etc.);
- Resource pooling, where resources are available in a pool so as to stay ready for future provisioning, serving multiple consumers using a multitenancy model;
- Rapid elasticity, which allows resources and capabilities to be increased based on consumers' demand, sometimes automatically;
- Measured service, or the metering and control of resources' usage, helping accounting services and providing transparency between providers and consumers.

There are three major kinds of industry-accepted Cloud Computing service models:

- Software as a Service (SaaS);
- Platform as a Service (PaaS);
- Infrastructure as a Service (IaaS).

An SaaS provides users with end applications that run on top of a Cloud Computing infrastructure, while PaaS provides the ability to deploy new applications developed by users, which will also be run by the Cloud Computing infrastructure. Finally, an IaaS goes even deeper by providing users with low-level resources which are typically, but not limited to, VMs, storage and specific networking components.

More recently, and thanks to the SDN explosion, IaaS Cloud Computing stacks have also started to offer more advanced networking resources, like private IP networks, mainly for interconnecting VMs. As is the case with VMs, these networking resources are also, typically, virtualized. Software-Defined Networking (SDN).

Resource virtualization (coupled with a SOA) is what makes their provisioning to clients possible and automatic, abstracting the physical layer of the Cloud Computing infrastructure. By decoupling resources provided (as a service) from the physical ones, a Cloud Computing provider does not need to carry out any physical action to give (new) resources to a (new) user. Besides that, automatic elasticity of resources depending on users' needs can be easily leveraged.

Finally, there are four different industry-accepted deployment models:

- Private Cloud, when the Cloud Computing infrastructure is exclusively used by a single organization;
- Community Cloud, like the private Cloud but for a set of organizations and consumers;

- Public Cloud, a Cloud Computing infrastructure that is available everywhere and to everyone willing to consume its services;
- Hybrid Cloud, a composition of at least two of the previous deployment models.

Currently, many IaaS and PaaS Cloud Computing providers exist, as well as multiple SaaS services.

A few examples of IaaS providers include: Amazon AWS, Bluelock Cloud Services, CloudScaling's Open Cloud System, Google Compute Engine, HP Enterprise Converged Infrastructure, IBM SmartCloud Enterprise, Rackspace Open Cloud, Windows Azure and Joyent.

A few examples of IaaS Open-Source Software (OSS) projects include: OpenStack, Open Nebula, Eucalyptus and Apache CloudStack.

A few examples of PaaS providers include: Amazon AWS, Appfog, Caspio, Engine Yard, Google App Engine, Heroku, Red Hat OpenShift and Windows Azure.

A few examples of PaaS OSS projects include: Cloud Foundry, OpenShift Origin, AppScale and Marathon.

Finally, some SaaS services that can be found today are: Adobe Creative Cloud, Dropbox, Gmail, Google Drive, GoToMeeting, iCloud, Microsoft Office 365, Salesforce CRM, Taleo's Talent Management Cloud, and the Docker Hub.

2.1.3 Virtual Private Networks (VPNs)

VPNs, as the name implies, are meant for private use. Because a public telecommunications infrastructure may be the only way a certain entity has to communicate with another entity (or another site), and given the requirements in terms of security, isolation and other guarantees, a Virtual Private Network is a way of fulfilling these requirements in such a public, shared network infrastructure. Per the VPN Consortium, there are three important VPN technologies [6]: Trusted VPNs, Secure VPNs and Hybrid VPNs.

Trusted VPNs are the ones where a VPN customer trusts the VPN provider with guarantees regarding its traffic, namely in terms of paths reservation and snooping prevention.

Secure VPNs do not trust any relationship with providers. As a result, traffic must be secured: encrypted and authenticated.

A Hybrid VPN mainly includes both basic VPN technologies simultaneously: Trusted and Secure, providing path reservation/assurance, snooping prevention and security.

Besides these recognized VPN technologies, they can also be distinguished in terms of the service type they provide. Venkateswaran, R. summarized the kinds of VPN services in three primary ones [7]: Local Area Network (LAN) Interconnect VPN services; Dial-Up VPN services; and Extranet VPN services. These services can be established in multiple Open Systems Interconnection model (OSI) layers, usually at OSI Layer 1 (L1), L2, OSI Layer 3 (L3) or even OSI Layer 7 (L7).

Furthermore, a VPN is a network that besides being private, is also virtual. It is virtual because it sits on top of a shared, public telecommunications network infrastructure (a form of network virtualization).

LAN Interconnect VPN services occur when a LAN, usually belonging to the same entity, is spread over multiple geographically dispersed sites. This was classically made possible through the use of leased lines or dedicated links.

Dial-Up VPN services are the most used VPN service kind due to the number of customers who use it: they are usually employees of a company, students from a university, and other individuals.

The Dial-Up VPN services enables users to connect to a remote network, e.g. the company where they work, as if they were physically present. Consequently, they can work and access resources local to the endpoint, but remotely. In terms of underlying technology, it usually sets up an L2 or L3 end-to-end link. The first effectively bridges the client machine to the site local network, resulting in a shared broadcast domain. The second establishes a network layer tunnel, usually IP, where the site's endpoint to this IP tunnel acts as a router to the rest of the site's network. Internet Protocol Security (IPsec) [8], [9], an extension to the IP protocol, allows advanced security to guarantee traffic isolation and authenticity while in the public shared network.

Finally, Extranet VPN services facilitate dedicated connections between different entities for accessing specific resources available through a Demilitarized Zone (DMZ).

2.1.4 NETWORK VIRTUALIZATION

Network Virtualization is a broad concept with the fundamental objective of allowing multiple networks to coexist where only a single, physical network exists [10]. One of the reasons Network Virtualization is used and trending, is the fact that it enables services to be decoupled from infrastructure, thus improving flexibility and customization of end-to-end services for end users.

An early technology considered part of Network Virtualization is the VLAN [11]. VLANs permit coexistence of multiple LANs via software, where otherwise only a single LAN would exist due to the physical equipment in use (for instance, a network switch would always aggregate a single LAN).

Another kind of Network Virtualization consists in VPNs [7]. A VPN, as the name implies, is meant for private use. Because a public telecommunications infrastructure may be the only way a certain entity has to communicate with another entity (or another site), and given the requirements in terms of security, isolation and other guarantees, a Virtual Private Network is a way of fulfilling these requirements in such a public, shared network infrastructure.

Overlay Networks are another kind of Network Virtualization which can be implemented in a variety of protocols. The basic idea is that a (virtual) topology is created on top of a another (main) topology. Given this, nodes and links between them are created on top of existing nodes and paths. Some examples of overlay networks include, but are not limited to, Peer-to-Peer (P2P) networks.

Finally, and most recently, Active [12] and Programmable Networks have been trending as a means to decouple networking hardware from control software [13]. Consequently, decoupling network infrastructure from service provisioning has also become easier, and a greater focus of research.

Under the umbrella of Programmable Networks we can find SDN (a paradigm), OpenFlow (a protocol) and other concepts and technologies that are empowering the latest trends in Network Virtualization.

2.1.5 Software-Defined Networking (SDN)

Computer networks can be looked at three different planes: data, control and management [14]. Classically, the planes of data and control have always been tightly coupled, e.g. network switches forwarding traffic based on internal criteria only.

SDN comes as a result from the efforts of Active and Programmable Networks groups, and it primarily aims at decoupling data and control planes in computer networks. The main advantage in

decoupling these is that now the control plane can be logically centralized (although it can still be physically distributed), thus simplifying actual devices by making them only deal with the data plane. Moreover, flexibility is enhanced while operational and capital costs are reduced by keeping maintenance mostly at the control plane [15]. Thanks to the these benefits, innovation can be accelerated as well.

The actual term of SDN came in order to represent the work and concepts around the OpenFlow protocol created at Stanford University [16].

The SDN architecture is organized in a way that molds against the three planes of a computer network. At the management plane, network applications are found. At the control plane, controller platforms are present. Lastly, regarding data plane, data forwarding elements, e.g. OpenFlow switches, are encountered [14]. The controller platform communicates with the data forwarding elements via an API, named Open Southbound API, whilst receiving new rules to be enforced from network applications via another API, named Open Northbound API.

Current projects around SDN include:

- OpenFlow [1], a protocol for manipulating the control plane in network switches or routers [16];
- OpenDaylight [2] [14], an OSS framework "to foster innovation and create an open and transparent approach to SDN" by the Linux Foundation;
- Open vSwitch [3], an OSS implementation of a virtual switch compatible with OpenFlow [17];
- Open vSwitch Database Management Protocol (OVSDB) [18], a management protocol for Open vSwitch, allowing programmatic extension and control;
- Project Floodlight [4], by Big Switch Networks, includes multiple OSS sub-projects related to SDN and OpenFlow;

2.1.6 LEGACY NETWORKS

Legacy network is not a well defined concept, as it depends on what is being considered as modern. For the sake of this work, legacy networks are mostly non-virtualized computer networks, interconnected by classic and/or heterogeneous network equipment (multiple vendors and protocols). However, legacy networks can also be seen as virtualized computer networks although not fully integrated into a logically centralized control that enables homogeneous control.

2.1.7 FOG COMPUTING

Fog Computing, a term coined by Bonomi et al. [19], extends the Cloud computing paradigm to the edge of the network. Based on the existence of Cloud Computing, which helps to centralize business decisions, data processing and storage for the recent Internet of Things (IoT) trend, there is an increasing need for a secondary level of data aggregation and processing, in between the Cloud infrastructure and the end devices that make up an IoT architecture.

[1] `https://www.opennetworking.org/sdn-resources/onf-specifications/openflow`
[2] `http://www.opendaylight.org/`
[3] `http://openvswitch.org`
[4] `http://www.projectfloodlight.org/floodlight`

As described by Bonomi et al. [19], the main motivating examples of deploying a Fog Computing architecture are as follows:

- Edge location, location awareness, and low latency;
- Geographical distribution;
- Large-scale sensor networks;
- Very large number of nodes;
- Support for mobility;
- Real-time interactions;
- Predominance of wireless access;
- Heterogeneity;
- Interoperability and federation;
- Support for on-line analytics and interplay with the Cloud;

2.2 TECHNICAL BACKGROUND

2.2.1 OPENSTACK

OpenStack [5] is an Open-Source IaaS Cloud Computing software stack or, alternatively, Cloud Operating System. Its development began in 2010 as a joint effort from Rackspace Hosting and NASA. Both entities already had experience in developing Cloud platforms, so they based initial work on their own projects: NASA's Nebula [6] and Rackspace's Cloud Files [7]. The project was setup as Open-Source since the very beginning, under the Apache 2.0 license. Today it is part of the Open Stack Foundation, a non-profit corporate entity established in September 2012 [20]. More than 100 architects from 25 different companies began defining the roadmap for OpenStack as well as new code during the project's inception [21].

OpenStack is comprised of multiple projects, each one with a clear focus on a specific Cloud Computing service. They are developed in a semi-independent way, in order to avoid dependencies between projects as much as possible, leveraging a high degree of modularity, powered by a SOA. Such an architecture allows Cloud administrators to only deploy Cloud services which are actually required for the specific scenario involved.

In a survey conducted by Linux.com and The New Stack [8] in August 2014, OpenStack was voted the top IaaS Free and Open-Source Software (FOSS) project of 2014 and the overall best FOSS Cloud Computing project [22].

The list of official OpenStack projects as of the Icehouse release (April 2014) [23]:

- Swift, or OpenStack Object Storage, is a highly available, distributed, eventually consistent object/blob store;

[5] `http://www.openstack.org`
[6] `http://nebula.nasa.gov`
[7] `http://www.rackspace.com/cloud/files`
[8] `http://thenewstack.io`

- Nova, or OpenStack Compute, is a Cloud computing fabric controller, the main part of an IaaS system, which, at its core, allows the provisioning of VMs;
- Glance, or OpenStack Image Service, provides services for discovering, registering, and retrieving VM images;
- Horizon, or OpenStack Dashboard, provides a web based user interface to other OpenStack services including Nova, Swift, etc,;
- Keystone, or OpenStack Identity, provides Identity, Token, Catalog and Policy services for use specifically by projects in the OpenStack family.
- Neutron, or OpenStack Network Service, provides "Network Connectivity as a Service" between interface devices (e.g., vNICs) managed by other OpenStack services (e.g., Nova);
- Cinder, or OpenStack Block Storage, provides "Block Storage as a Service" which can be used for storing VMs' volumes;
- Ceilometer, or OpenStack Telemetry, aims to deliver a unique point of contact for billing systems to acquire all of the measurements they need to establish customer billing;
- Heat, or OpenStack Orchestration, is a service to orchestrate multiple composite Cloud applications;
- Trove, or OpenStack Database Service, is a "Database as a Service" that allows customers to quickly and easily utilize features of a relational database without the burden of handling complex administrative tasks;
- OpenStack Documentation, although not really a Cloud service, is still an essential project to OpenStack's success.

Besides these projects, there are other ones meant to interact with the main projects' APIs by recurring to CLIs, using their APIs under the hood.

Other programs include Development and Operations (DevOps) for joining together developers and Telecommunications or Cloud Operators, to accelerate development that goes towards major market requirements while simplifying and amplifying feedback from the operations personnel.

Code contributions to OpenStack are provided to the Continuous Integration system used, Jenkins, and need to be tested in an infrastructure dedicated for that purpose, the Test infrastructure. The objectives are to increase the probability of discovering new bugs and problems when a new patch is made, either caused or made visible by it, and making sure the code is internally correct per the tests defined. There are multiple ways tests can be run (test run styles), each with different trade-offs between cost, reliability and code coverage. Depending on the way tests are run, contributions can be automatically voted by the Test infrastructure in a positive or negative way, flagging the contribution as ready or unready to be reviewed by the core developers, respectively. There are five kinds of test run styles or jobs: Experimental, Silent, Third party, Check and Gate [24]. The first test run style is meant for Experimental test jobs, which are manually triggered by a developer and reliability can be low. Jobs of this kind are not able to vote. Silent test jobs are normal test jobs, i.e. cannot be manually run by a developer, that would otherwise be able to vote but cannot due to a temporary technical difficulty that impairs its credibility. Third party jobs are run by third parties when new code is merged, and are able to vote. Check jobs not only run when code is merged, but for each patch set being proposed, and are maintained by the OpenStack community. This kind of jobs must be highly available. Finally, Gate jobs have the same importance and administration of Check jobs,

but are run after a contribution is merged so failures can be detected after a patch is approved (and against all other patches merged so far).

OpenStack is governed by a number of bodies. Besides the corporate members of the OpenStack Foundation and its board of directors, there are other positions one can achieve [25]. For each program/project, like Neutron or Nova, there is a Project Technical Lead (PTL) who is responsible for overseeing the project status and taking project-level technical decisions. A PTL is elected by the community of the project in regard. Furthermore, the Technical Committee (TC) works with the PTLs to set technical policies that cross different projects, including new ones still in incubation. TC members are elected by individual corporate members of the foundation. Inside each project there are also developers with enhanced power, named core developers, which have earned that status by meritocracy (community votes taking into account contributions made from the past to the present). Core developers are able to, collectively, decide what patches should be merged or not.

Every 6 months a new major version is released, marking the principal milestone both in terms of release management and development roadmap. All official projects are simultaneously updated during each major version, offering a new complete OpenStack software suite without further delays.

Figure 2.2 presents the logical architecture of OpenStack as of the Havana release (taken from [26]).

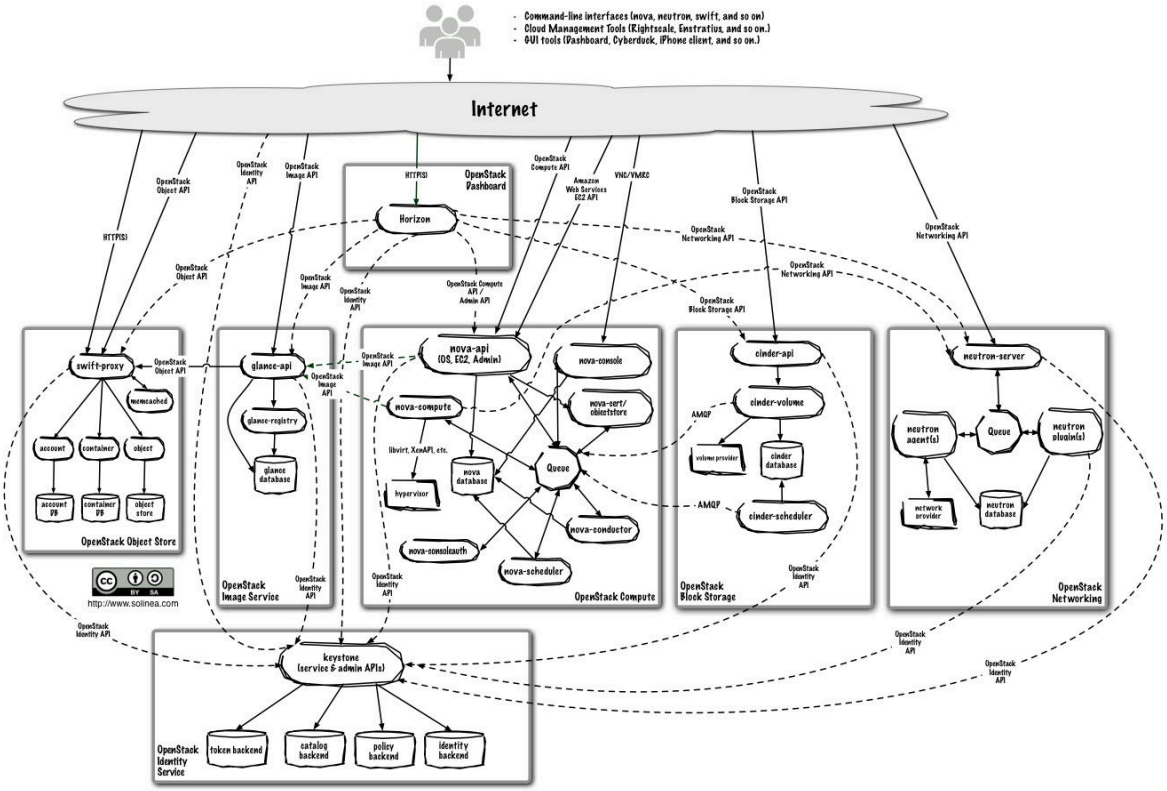


Figure 2.2: OpenStack Logical Architecture for The Havana release [26]

2.2.2 NEUTRON

The most important OpenStack project regarding the work presented, which must be introduced, is Neutron. If otherwise stated, all information about its architecture applies to the Icehouse release (from April 2014), against which the dissertation's work has been tested.

Neutron is the Network Service of OpenStack. It allows interface devices, like vNICs from Nova-managed VMs, to be connected between each others. Neutron has evolved from the networking part of Nova, named Nova Network. Nova Network made use of a basic model of network isolation via VLANs and linux firewall IP tables. There was the need to create a dedicated project for networking to avoid excessive complexity in the Nova project while improving the ability to develop new, and more advanced, networking features and services. Nowadays, Neutron is already very complex by itself but is a crucial service for the deployment of modern IaaSs, including the ones that can be used by Telecommunications Operators (Network Function Virtualization (NFV)).

Neutron was designed from scratch to be an extensible plugin-based "NaaS" Server. In other words, Neutron in its basic form is simply a logically-centralized API for networking requests, supporting only fundamental networking resource types that allow VMs to be connected: networks, subnets and ports. These are called Neutron core resources, and the API is the Core Networking API.

The fundamental resource types available are:

- *Network*: A Neutron Network consists in an abstraction of a typical L2 computer network segment, i.e. a broadcast domain. It is the basis for describing an L2 network topology.
- *Subnet*: A Neutron Subnet is associated to a Network on a many-to-one fashion. This abstraction enables IPv4 or IPv6 address blocks to be associated to Networks. Other network configuration that can be attached to Subnets are default gateways, Domain Name Server (DNS) servers, and other L3-specific attributes.
- *Port*: A Neutron Port is the fundamental interface between Networks and what is intended to be connected to them. It is analogous to an attachment port on an L2 network, e.g. a classic network switch. When a Neutron Port is created on a Neutron Network, an available fixed Internet Protocol (IP) address (per each IP version) is automatically assigned from an available Subnet. When a Port is deleted, its addresses are reinserted into the Subnet's pool of available IPs addresses.

The core resource types introduced need to be implemented so they can actually be used in a datacenter, a Cloud-based datacenter, by mapping them to real networking equipment (hardware or software). As previously stated, Neutron is plugin-based. A core/L2 plug-in provides the back-end implementation Neutron needs to execute the network topology described by the set of core resources created through the Core Networking API. Usually, these will be vendor plug-ins, i.e. pieces of software that at least include code to communicate with the specific vendor's equipment and translate the core resources into real, probably physical, network topologies. This is somewhat similar to an Operating System (OS) driver for a hardware device.

However, there are other kinds of Neutron plug-ins, named Service plug-ins, which provide additional functionality besides core L2 connectivity [27]. Also, core Neutron plug-ins may implement Service plug-in functionality as well. An example of a Service plug-in is the L3 plug-in to provide routing, NAT, floating IP functionality in Neutron networks, e.g. the L3 agent which makes use of Linux iptables and network namespaces [28]. Other examples include Firewall as a Service (FWaaS), Load Balancer as a Service (FWaaS) and VPN as a Service (VPNaaS).

A Neutron core plug-in can, thus, utilize a variety of technologies to implement logical API requests, be it either OpenFlow, simple Linux VLANs, Linux IP Tables, tunnel set-ups, or some other obscure technology.

It must be noted that these resources, even though they are the core, can be extended to support more attributes, configurations and actions.

Besides extending core resources and, consequently, the Neutron Core Networking API, other aspects of Neutron can be extended as well. For instance, new kinds of resources and actions can be created to leverage new functionality.

However, when something in Neutron is extended, each core plug-in must also follow and implement code that carries out the extensions' changes. Otherwise, core plug-ins that do not support some extension will not be able to leverage the advantages of that extension (but will not break the rest of the networking functionality).

Neutron can be extended by recurring to at least one of three different types of extensions:

- *Resources*: Resource extensions introduce new entities, or resources, to the Neutron Core Networking API, similarly to the Networks, Subnets and Ports core resources.
- *Actions*: Action extensions can be seen as predicates, or operations, on what to do with resources. There are no core actions, but extensions can create their own ones.
- *Requests*: Request extensions allow new attributes to be added to existing resources, core or not. They are called request extensions because they extend on what API requests can provide and acquire.

The Neutron group develops and supports a reference core plug-in, called ML2. The ML2 plug-in is not vendor-specific but rather open, generic and even extensible by itself (besides being able to implement Neutron extensions, per the plug-in architecture).

One of the fundamental objectives when the development of ML2 started, was to try to reduce code and effort duplication each time a new core plug-in was developed. It was observed that every plug-in made used similar resources. For instance, data structures to define, populate and use VLANs or tunnels. Each new plug-in needed new code for its specific implementation, even though part of it could be shared (interface-wise).

ML2 made it possible since its very inception to separate code for data types from code of actual functionality. The former is called a Type Driver while the latter is a Mechanism Driver.

Type Drivers are responsible for handling each available network type and their correspondent data structures.

The network types supported by the ML2 core plug-in's Type Drivers are:

- Local, which is a kind of network implemented on a single Linux host without recurring to special technologies;
- Flat, which allows reusing the underlying network on which Neutron is installed;
- VLAN, for networks that are implemented and isolated by recurring to VLAN technology available at the operating system;
- GRE, for networks to be isolated via the GRE tunneling technology [29] available at the operating system;

- Virtual Extensible LAN (VXLAN), similarly to GRE, this network type allows networks to be implemented and isolated using the VXLAN tunneling technology [30], in this case.

The great level of modularity in ML2 follows the trend in the rest of the project. Besides that, its first release as the default Neutron plug-in, back in Icehouse, came with support for at least two important Mechanism Drivers: LinuxBridge and Open vSwitch [31].

Previously, LinuxBridge and Open vSwitch were available directly as core plug-ins for Neutron network deployments consisting only in typical Linux-running computers. With ML2, they were decorated with the Mechanism Driver interface to, on one hand, be utilized under ML2 without any internal change and, on the other hand, allowing ML2 to be used as the default core plug-in. Figure 2.3 presents a vertical view on the architecture of Neutron with ML2.

In conclusion, vendors can now make use of ML2 to develop new Mechanism Drivers for their equipment with reduced effort and better consistency due to Type Drivers. Still, Neutron extensions must be supported by each Mechanism Driver, like they must be supported by other plug-ins.

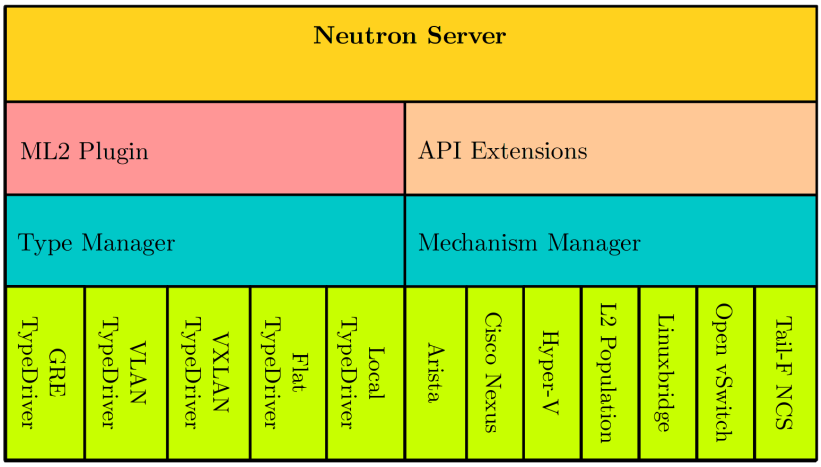


Figure 2.3: Neutron architecture taking into account ML2 (originally from [31])

OpenStack enables network engineers and datacenter administrators to deploy and make use of a multi-node Cloud Computing IaaS, to provide a highly available, efficient and flexible service. There are typically three kinds of OpenStack nodes: the Cloud Controller Node, the Network Node and the Compute Node. Because OpenStack is designed to be massively horizontally scalable, these nodes can be distributed to provide enhanced availability, more performance and more capacity.

NODES

An OpenStack deployment is usually spread amongst different nodes: computers running OpenStack services. Services can be assigned to any node because there is no hard restriction for it. However, there are typical deployment scenarios, where each node is considered as belonging to a certain kind, depending on the set of services it runs. The major three kinds of nodes, when using Neutron, are called: Cloud Controller Node, Network Node and Compute Node. Figure 2.4 exemplifies a typical deployment with the three kinds of nodes and what services are usually run on each one.

Cloud Controller Node. The Cloud Controller Node is a logically centralized node, which provides the central management system for OpenStack deployments. It is, at least, responsible for managing

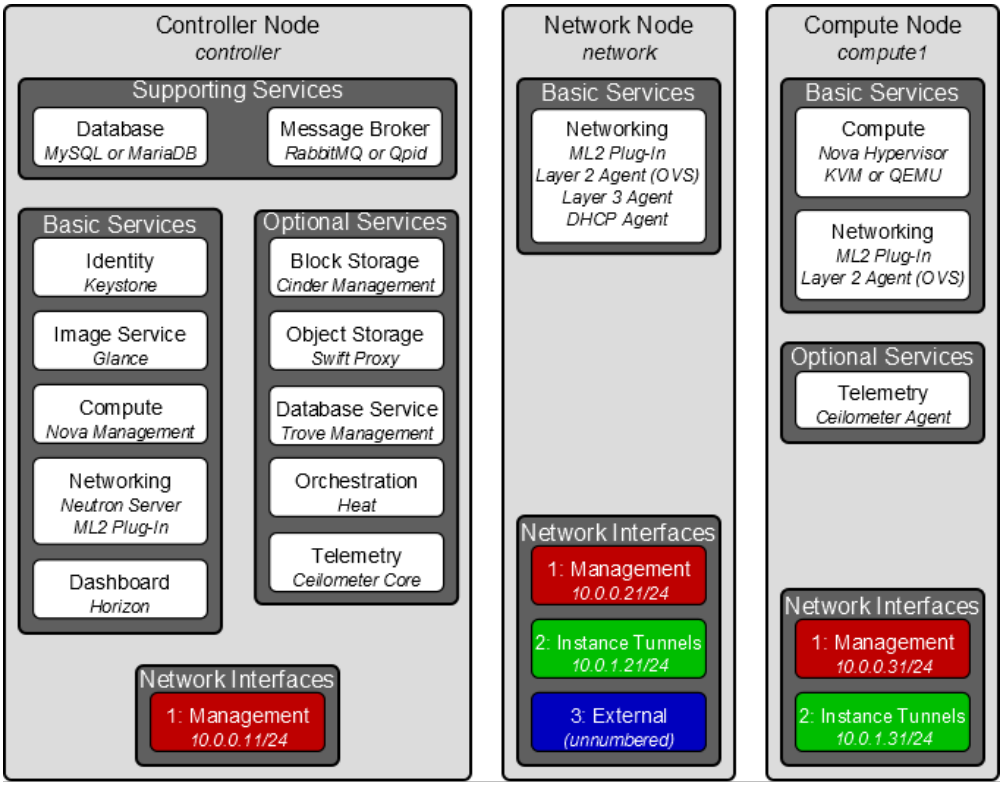


Figure 2.4: An OpenStack deployment on three nodes, with Neutron [32].

authentication aspects and sending/receiving messages to other systems via an Remote Procedure Call (RPC) message queuing system over RabbitMQ[9]. It usually is a single node, but to achieve high-availability its services must be carefully split.

The list of services the Cloud Controller Node is entitled to manage are the following [33]:

- Databases;
- Message queue services;
- Conductor services;
- Authentication and authorization for identity management;
- Image-management services;
- Scheduling services;
- User dashboard;
- API endpoints.

Network Node. An OpenStack Network Node is a dedicated node for networking operations. It runs services such as Dynamic Host Configuration Protocol (DHCP) servers and agents, L3 agents which provide virtual routing entities, etc. It is supposed to be directly connected to an External Network in order to provide outside connectivity to VMs running on Nova. Traffic from these VMs should reach the network node via a previously configured technology, usually enabled by an L2 agent

[9] `https://www.rabbitmq.com/`

like the Open vSwitch (OVS) agent which, e.g., sets up GRE tunnels to interconnect VMs spread in multiple Compute Nodes which are part of the same network.

This is also most important node of this work, since that is where Neutron main services are run, as well as the OVS agent which already provides some of the functionality required to leverage this solution.

Just like with the Cloud Controller Node, this kind of node can be distributed in multiple hosts to achieve high-availability and enhanced performance.

Compute Node. An OpenStack Compute Node mainly hosts VMs managed by the Nova service. They provide all resources necessary to run instances (VMs), like processing, memory, network and storage. This is the kind of node that is usually distributed in a higher degree because the number of VMs can escalate quickly when there are multiple tenants with multiple instances running.

Other nodes. Due to OpenStack's flexibility and modularity, different services can be split into different nodes, even if they do not really fit a self-contained objective. What services can be split and through which internal networks depend. However, besides Cloud Controller, Network and Compute Nodes, the following kinds of (named) nodes can usually be found:

- Block Storage nodes which contain disks to provide volumes, via the Cinder service;
- Object Storage nodes, which provide a distributed, replicated object store, via the Swift service;
- Proxy nodes, which take requests and look up locations for other services, routing the requests correctly, while also handling API requests.

NETWORK DEPLOYMENT

In a typical OpenStack deployment featuring the OpenStack Networking service, Neutron, a network architecture similar to the one presented in Figure 2.4 can be obtained. It must be noted that the actual architecture to be deployed is very flexible, so multiple solutions can be thought of. However, for the sake of this work, only the network architectures presented in Figure 2.4 (from Icehouse) and Figure 2.5 (from Havana) are explained because they are simple and easily translatable to different ones, all supporting the work presented in this dissertation. The networks presented in these figures are very similar, to the point that it is safe to assume they are the same network deployment. By focusing on a single network deployment and assuming its existence, any further match between specific implementation aspects presented and deployment characteristics of an OpenStack installation becomes easier to assimilate by the reader.

The example architecture with OpenStack Networking (neutron) requires one Cloud Controller (controller) node, one Network Node, and at least one Compute Node. The Cloud Controller Node contains one network interface on the Management Network. The Network Node contains one network interface on the Management Network, one on the Instance Tunnels Network, and one on the External Network. The Compute Node contains one network interface on the Management Network and one on the Instance Tunnels Network [32].

There are, thus, at least three kinds of networks in a typical OpenStack deployment (with Neutron), and they are defined as such:

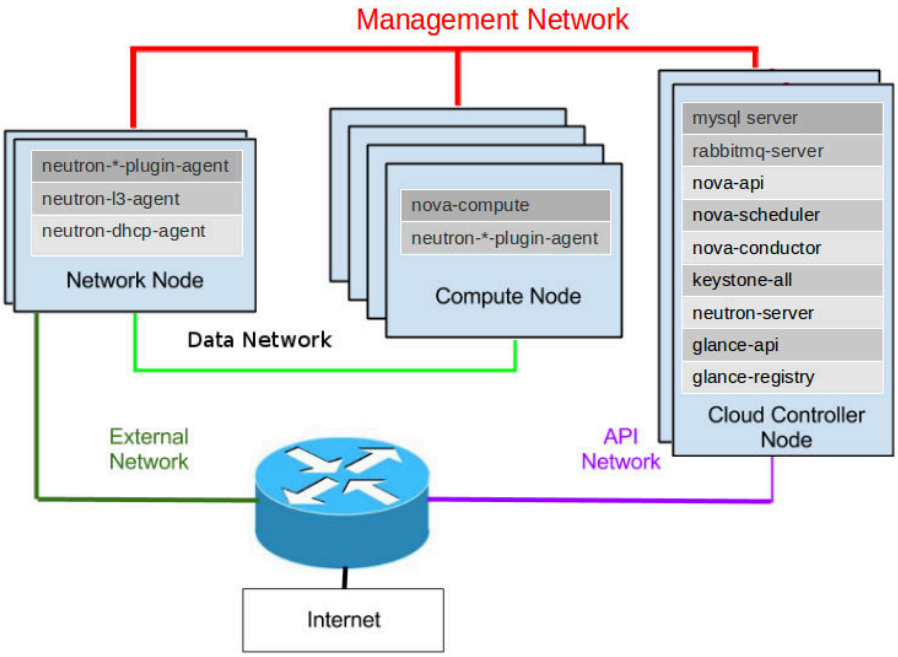


Figure 2.5: An Havana deployment on three nodes, with Neutron [34] (from Havana docs).

Management Network. The Management Network is a private network which interconnects all OpenStack nodes in the same broadcast domain. Having a segregated network for this purpose prevents administration and monitoring of the systems to be disrupted by tenants' traffic, while also implicitly improving security by having less hosts exposed to public or semi-private access.

This network can be further split in other private networks for dedicated communication between specific internal components of OpenStack, e.g. message queues.

There will usually exist a dedicated network switch to interconnect all nodes' physical NICs, which may also employ VLANs for the purpose of segregating the private network into smaller, more specific ones, as mentioned above.

Data Network. Sometimes called the Instance Tunnels Network as well. This network is deployed when joint Compute and Network (logical) nodes are more than a single physical node. Also, it should be noted that the existence of this network depends on the active Neutron core plug-in. In a scenario where ML2 is used as the Neutron core plug-in, with GRE as the tunneling technology for virtual networks, the Instance Tunnels Network (or Data Network) should exist.

The Data Network is conceptually an isolated network dedicated for traffic between different Compute and Network Nodes. By having a dedicated network for this purpose, traffic from/to/between VMs running on the Compute Nodes will not interfere with control traffic usually present at the Management Network, or external traffic present in other networks. Full throughput can be achieved this way. The Data Network can actually be the same as the Management Network, although in that case the advantages of having a dedicated network for the Compute and Networks nodes are lost.

External Network. An External Network is a network which is directly connected to the Network Node(s) and usually provides access to the Internet. This network is connected to a virtual router running inside Network Nodes, which then allow tenants' VMs to access the Internet (or just the external Network) by having these also connected to the virtual router. This router effectively acts as

a NAT server for the tenants' networks with Dynamic NAT (DNAT) support so instances can reach the outside with a public IP from the External Network (usually the IP of the Network Node. It is provided to Neutron as an L3 plug-in. The Network Node provides a pool of public IP address, known as floating IPs, which may be assigned to instances, thus supporting Static NAT (SNAT), enabling them to be accessed from the Internet (if the tenant so desires).

Traffic from instances, having Internet as the destination, will come from the Instance Tunnels Network, as previously explained, reach the relative Network Node, cross the virtual router, and finally head out to the External Network. By having a dedicated network to access to exterior of OpenStack, security concerns in terms of malicious tenants' attacks can be alleviated. It also gives space to a dedicated API network, which may or may not reach the Internet as well, but is secure from tenants' attacks and or other disadvantages of sharing a network for tenancy and administration purposes.

API Network. Another kind of network should be defined: the API Network. Even though not present at the Icehouse example deployment (only seen in Figure 2.5). This network is similar to the External Network in the fact that it can also be accessed from the outside. However, it is meant only for administration purposes, i.e., outside API calls. By having it segregated from the rest of the networks, security is inherently improved. Typical Cloud Computing requests, like requesting the creation of a new tenant with specific quota requirements, will reach the Cloud Controller Node via the network.

CHAPTER 3

State of the Art

This chapter exposes the latest, and most similar technology related to this work, and points out the most important missing aspects of each one. Some work is very practical and not scientifically-backed, although an effort has been made to acquire the principal research trends which align with the work proposed in this dissertation.

3.1 Physical networks in the virtualized networking world

Bruce Davie presents in [35] a way for extending virtual networks from the datacenter to the outside, spanning both virtual and physical resources. An year later Bruce Davie et al., in [36], presented an update on the original implementation further improving the work.

They were able to achieve this by using a set of modern technologies: a pair of a hypervisor and a virtual switch per host providing VMs; VXLANs overlays to tunnel the traffic towards endpoints somewhere outside the datacenter; OVSDB as the protocol to carry networking information to configure the network according to the specific use cases they had in mind; VMware NSX[1] as the network virtualization controller that receives API requests and in turn configures the desired topology via OVSDB, while also mapping physical ports (and their VLANs) to logical networks; and finally VXLAN-capable network switches, called VXLAN Tunnel End Points (VTEPs).

The topology is instantiated over a set of Top of Rack (ToR) switches that support both VXLAN termination and the OVSDB protocol.

The simplest use case achievable with the proposed implementation is extending network segments, at the OSI Layer 2, that encompass VMs deployed, towards the other side of the physical network that encompasses physical machines. Furthermore, the developers have gone even further and implemented services in higher layers, namely Distributed Logical Routing [36] (L3).

[1] `http://www.vmware.com/products/nsx`

Despite the facts that this solution is flexible, allows multiple levels of services to be deployed across the physical and virtual networks, and control is logically centralized, it has its disadvantages as well.

First, it has no integration with an existing Cloud Computing network and compute infrastructure stack, which may already deploy hypervisors, virtual switches, tunneling technologies, and even integrating with other Cloud Computing services. And second, it requires specialized hardware at the physical side of the network. Specifically, it requires switches that support the modern OVSDB protocol, a fact that precludes integration with legacy networks or incremental legacy to Cloud network migration scenarios.

3.2 A PROPOSAL MANAGEMENT OF THE LEGACY NETWORK

Farias et al. proposes in "A proposal management of the legacy network environment using OpenFlow control plane" [37], a way to use existing legacy infrastructures simultaneously with new and experimental/future network architectures, software and protocols, by making use of OpenFlow.

The solution presented intends to keep the legacy part of the network intact, while leveraging new network functionality and concepts through modern technologies like OpenFlow, making it possible for entities to incrementally migrate to new technologies or towards an all-virtualized network infrastructure, but reusing legacy equipment until they are not further necessary. Moreover, it enables new approaches to be tested without sacrificing current production infrastructures, by carrying them out simultaneously with these but in an isolated way.

The core problem lies in the fact that pure legacy networks will not be able to work alongside OpenFlow.

Given these issues and objectives, the authors came up with a design called LegacyFlow which aims at maintaining a legacy network, usually a core one, intact, while being surrounded by modern networks based on OpenFlow (see Figure 3.1). LegacyFlow includes typical parts of an SDN network: an OpenFlow controller and OpenFlow switches. Then, in order to integrate with non-OpenFlow switches, i.e. create a link between OpenFlow and the legacy network, a new OpenFlow datapath was created, dedicated to applying legacy configurations through special OpenFlow actions. This new datapath, called Legacy Datapath, will communicate with special controllers for each kind of legacy switch. These controllers are pieces of software, spawned as processes, with which the Legacy Datapath communicates via Inter-Process Communication (IPC). They will know how to translate Legacy Datapath actions into each legacy switch's language: CLI commands via telnet or Secure Shell (SSH), web services via Hypertext Transfer Protocol (HTTP), Simple Network Management Protocol (SNMP), etc. The overall architecture can be seen in Figure 3.2.

When a packet is recognized by the datapath, it will check for a matching flow. If it is not recognized, it is sent upwards to the OpenFlow controller, which will then install new flows via the Legacy Datapath and consequently have the corresponding legacy switches configured. Traffic will transit through tunnels or circuits based on VLANs between legacy edges, because the production network must stay intact.

This solution is especially useful because it enables legacy networks to continue their existence while modern networks improve and replace the former. Logical control centralization is also a plus,

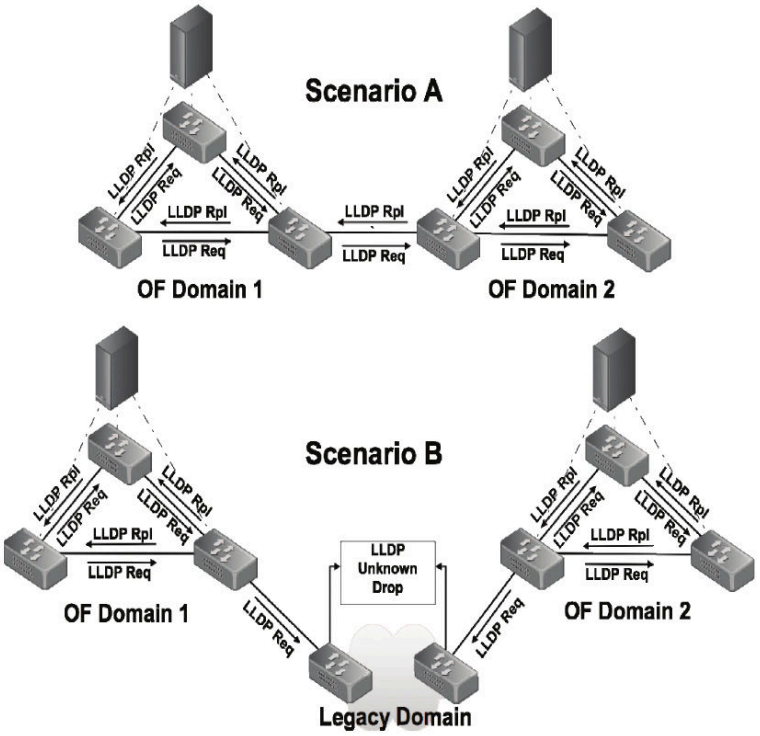


Figure 3.1: Scenarios with the OpenFlow and Legacy network [37].

comparing at how legacy networks are usually controlled (distributed in a mostly manual manner).

However, it aims at a network infrastructure where modern networking equipment is already in place, so the need is to circumvent the remaining part of the network (the legacy part) in case it cannot easily be replaced. If we take as an example a company having its network completely legacy, with the objective of making that network available as a new part of a virtual network, this solution does not really apply. Also, Cloud Computing integration has not been addressed or discussed.

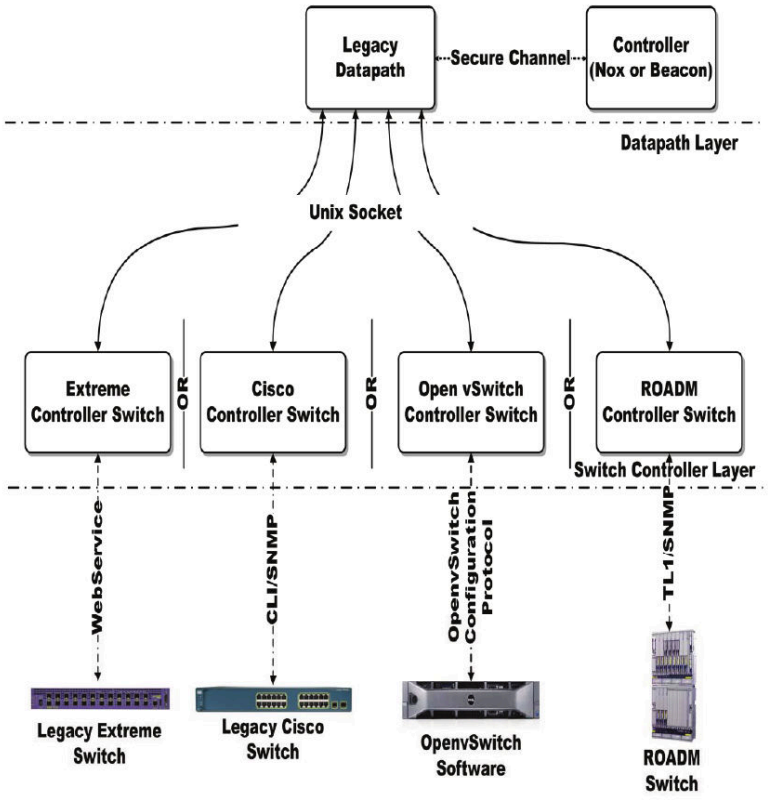


Figure 3.2: LegacyFlow architecture [37].

3.3 VIRTUAL AND PHYSICAL NETWORK FOR A NETWORKING LABORATORY

Chan and Martin propose in "An integrated virtual and physical network infrastructure for a networking laboratory" [38], a way to leverage a virtualized networking environment on top of a physical, legacy network, with the objective of optimizing computer networking lab classes in multiple fronts. They claim that not only will traditional computer networking classes' topics, like Routing Information Protocol (RIP), Open Shortest Path First (OSPF), Border Gateway Protocol (BGP) and VLANs, benefit from using a combination of physical and virtual networking devices, but also classes of modern and virtualized computer networks, including server virtualization, will be easily achievable by lecturers.

It is important to first define what are virtual and physical network infrastructures in terms of a networking laboratory. A physical network infrastructure is the basis for a physical networking laboratory, where students acquire hands-on experience with the traditional topics and equipment (like routers, switches, etc.). Similarly, a virtual network infrastructure is the basis for a virtual networking laboratory.

With the principal definitions in mind, they expose their design objectives as such (quoted directly from [38]):

- To create a flexible and scalable virtual networking environment to support laboratory work;
- To facilitate hands on networking experience with physical routers, physical switches, virtual routers, and virtual switches;

- To introduce new IT technologies: infrastructure, system virtualization, network virtualization, and Cloud computing;

Necessary hardware equipment consists only of traditional routers, switches and workstations running Linux.

Furthermore, a purpose-built VMware ESXi[2] server is deployed, providing core network virtualization as well as actual VMs, which then integrates with the rest of the physical equipment. Inside the core network virtualization, virtual switches can be found, making it possible to achieve a high degree of flexibility only typical of virtualized networks. This flexibility is important for experiments in a large range of technologies, making it ideal to teach the newest technologies without additional Capital Expenditure (CapEx) and Operating Expenditure (OpEx). On top of the server, other networking services can be deployed and attached to the network, for instance IP routers (like Vyatta[3].

Finally, actual integration between the ESXi server and the rest of the networking equipment, to be used by the students, is done by simply connecting the server's interface to a physical switch and configuring VLANs accordingly. The rest of the networking equipment connected to the physical switch, usually used by students, will be automatically connected to the correct network (defined by the lecturer) and the lab can be carried out smoothly (e.g. without dependencies between different groups).

This solution is an example of how to drastically improve networking lab classes in terms of flexibility, technologies offered, CapEx and OpEx. However, instead of having a dedicated server where to deploy the main networks, Cloud Computing integration would be a great plus which would allow to reuse an eventually existing Cloud Computing infrastructure. Let's consider that a university, providing network lab classes, already has a Cloud Computing infrastructure set up as the core for the overall campus' network. A tenant could be created specifically for the setup of virtual networks and machines to be used in lab classes. Another disadvantage is that this solution has a limited view on the global network. In other words, it is not possible to have a view of the network from the virtual server towards the physical devices. Also, physical devices cannot be made "controllable" by the server. In summary, logical centralized control of physical devices is not really addressed or discussed.

3.4 NETWORK INFRASTRUCTURE CONTROL FOR VIRTUAL CAMPUS

In his Master's Dissertation, Filipe Manco envisioned an advanced manner for virtualizing an entire legacy network infrastructure and attaching it to a Cloud Computing provider [39].

He created four different kinds of entities that map to one or more physical network elements, which are then the basis for creating a network overlay on top of the legacy network. The four entities are:

Nodes. Nodes represent any kind of devices that can be configured by the system in order to achieve virtualization. Examples include switches, routers, amongst others.

[2] `http://www.vmware.com/products/vsphere/features/esxi-hypervisor.html`
[3] `http://www.brocade.com/launch/vyatta`

Ports. Ports are points of access to the network and are provided by Nodes. They are used by end devices to achieve connectivity.

Segments. Segments interconnect Nodes using some technology, like VLANs or GRE tunnels.

Links. Links are actual connections leading to each Node and are part of Segments.

The architecture was split in two main parts: a Campus Network controller (CNc) for managing what networks are deployed, and a Campus Network agent (CNa) for communicating with each device.

This solution effectively addresses virtualization of legacy networks composed by heterogeneous networking elements and technologies, enabling new L2 networks on top of existing datacenter resources.

Cloud Computing integration is also a plus, having OpenStack been chosen as the proposed software stack to partly act as a CNc and thus integrate with Neutron's tenant networks. Implementation-related details can be found in the following OpenStack blueprints: [40] and [41], also available as appendices in his Master's Dissertation.

However, the architecture may become administratively complex, which difficults problem tracing and fixing, consequently keeping OpEx not the lowest possible. Moreover, no final implementation was developed and tested to ascertain the feasibility of this design.

3.5 EXTERNAL ATTACHMENT POINTS FOR OPENSTACK

A very recent proposal to OpenStack has been from Big Switch Networks, to enable what they call "External Attachment Points" [42]. The objective is to automate assigning points of the network, which are external to the Cloud, to the internal Cloud networking infrastructure. The main use cases presented that justify this undertaking are the ability the create L2 gateways that make it possible to extend existing broadcast domains of Neutron networks, via existing datacenter switches. The root motivation for this work lies in the fact that there is no well-defined way to connect devices not managed by OpenStack directly into a Neutron network. Usually, it is necessary to manually create Neutron ports to match the Media Access Control (MAC) address of external equipment that want to be integrated to the network, besides the fact that VLANs still need to be changed to the proper ones if the device is to communicate with the correct Neutron network, for which there is currently no way to do so.

The proposed workflow is that administrators create attachment points mapped to devices in the datacenter that can extend an L2 network, which can then be used by tenants to extend their networks.

Three formats of attachment points are included in this proposal: L2 switches supporting VLANs, OVS gateway and bounded port groups which can be multiple instances of the previous formats.

This proposal is very interesting taking into account the dissertation's motivations and objectives. It also has some similarities with one of Filipe Manco's blueprints: ML2 External Ports Extension [41]. However, the proposal has limitations in the following aspects: limits itself to attachment points created by administrators which difficults legacy to Cloud migration use cases, especially for public Clouds; does not predict how distantly can network attachments function properly; even though it has a high degree of heterogeneity by allowing the use of switches' VLANs or OVS gateways, it still is not heterogeneous enough for clients that make extensive use of legacy networking equipment, whose access network may even mask or difficult reaching internal VLANs; and finally, it is only a blueprint proposal for OpenStack and some preliminary code that do not yet make up a usable implementation.

CHAPTER 4

Problem Statement

Cloud Computing paired with Network Virtualization provides very interesting and useful features to customers. Cloud tenants or administrators are now able to design and implement their own network topologies, with considerable flexibility, and attach them to their own virtual machines. This is true both for private, public or hybrid Clouds.

However, in order to achieve the desired level of elasticity and ease of maintenance, the resources to be virtualized need to be as homogeneous as possible. Although that is not really a disadvantage per se, there are undesirable consequences when providing a service like this to clients, namely overhead and performance concerns [43] due to virtualization, and no backwards compatibility with existing equipment.

Furthermore, it may be necessary to provision bare metal machines, again due to overhead and performance concerns. Gordon et al. [44] sum up the major causes of performance losses in virtual machines and further present an innovative method for reducing these losses, in terms of I/O interrupts.

As a result, there may be the need to avoid Cloud Computing altogether, depending on the set of requirements presented by the client, effectively dismissing a modern technology in favor of something else. Therefore, to avoid this scenario and make both sides benefit (Cloud Computing providers and service consumers/clients), the advantages of Cloud Computing must coexist with other requirements like performance. One way of achieving this objective is by having special and heterogeneous kinds of resources that serve the sole or principal purpose of guaranteeing important requirements.

To illustrate, and picking up on the performance concern again, a Cloud customer may be interested in having a high-performance Deep Packet Inspection (DPI) as a service. It is not important if such a service is currently available on Cloud Computing stacks. What is important is that the DPI service may not guarantee enough performance, per the customer requirements. If this customer was able to externally provide a special machine for this purpose (which he/she knows will meet performance requirements), and attach it to the Cloud, the problem would be solved. This is what was meant by "having special and heterogeneous kinds of resources". The special machine is the special resource here. It is heterogeneous because it will be different than the rest of machines or services provided in-Cloud due to its ability to meet special criteria and/or requirements, and it will be connected in an external, exceptional way.

A new set of use cases can be enabled by the possibility of attaching points of the network, including up to the Internet, external to a Cloud Computing infrastructure (with a networking stack available to

tenants), into the infrastructure itself. That way, all benefits of Cloud Computing services (including multi-tenancy) can be leveraged for these external points as well.ok

4.1 USE CASES

4.1.1 INCREMENTAL LEGACY TO CLOUD MIGRATION

Consider, for instance, a company (called C1) that has its own datacenter, being responsible for its maintenance and having complete control over all equipment, which are very heterogeneous amongst them, with different kinds, brands, technologies, requirements and interoperability aspects. Besides that, some nodes have special functionality (like DPI) but are very hard (or costly) to replace by analogous, yet different, technology. Now, this company has its reasons for moving into the Cloud, but it faces multiple worries and blockers:

1. Strict requirement to keep special functionality where it currently is, due to physical control, latency and performance requirements, or other aspects;
2. Desire to keep some existing equipment to maximize investments and avoid major expenses;
3. Difficulty in replacing the current network deployment as a whole;
4. Some services currently provided by the infrastructure may not be tolerant to downtimes, meaning that they must be migrated only when strictly necessary and as quickly as possible, with the Cloud replacement already in place and properly tested;
5. There may be resources hampering or making the transition to Cloud more difficult, at present time.

By moving to a Cloud Computing provider that supports the work presented in this dissertation, C1 can move all resources but the ones mentioned as worries or blockers. Then, Cloud-provided networks would be extended to merge legacy resources that were not moved to the Cloud. This network extensibility only requires calling special operations on the Cloud Computing provider, and making sure there is a reachable point in the legacy network that is externally configurable/manageable, in order to provide the linking point between the two.

Figure 4.1 shows what can be accomplished in this use case: making legacy resources, already part of a legacy network, become part of a Cloud-managed virtualized network which is an extension of the legacy network, and vice-versa.

Afterwards, when/if no other legacy resources are necessary, the link can be brought down without any loss of functionality or down time, simply by calling tenant API methods. However, the administrator must have already provisioned all replacements for legacy resources inside the Cloud-provided network(s).

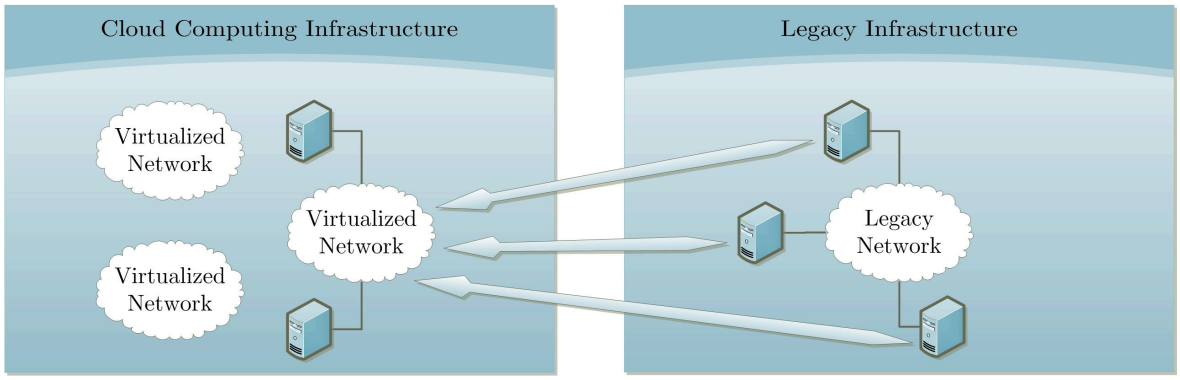


Figure 4.1: Legacy resources being attached to a virtualized network.

4.1.2 CENTRALIZED NETWORK CONTROL AND FLEXIBILITY

By having parts of the networking Cloud spread across different physical/legacy segments (not in the Cloud, as to say), while also offering the ability to reconfigure these parts via a logically centralized API with the benefits of Cloud Computing, network administration becomes easier and more flexible in scenarios where infrastructure keeps changing.

One of the most useful realizations of centralized networking control and flexibility is the creation of a virtual campus. This enables a centralized control of a campus network infrastructure, either by an administrator or through multi-tenancy, along other Cloud Computing advantages.

Consider a modern university campus scenario, where there is some heterogeneity of network connections. Users may connect to the network either by Ethernet links, usually available in classrooms, or via Wireless. There will usually be multiple broadcast domains, as to prevent performance issues originated from excessive traffic. These broadcast domains may be associated to specific locations and/or connectivity types. Besides, they may also be associated to specific users' roles, for instance: students, teachers, administration staff, specific classes' students, amongst others. Based on these premises, there is a need to manage the network as a whole with as least effort and most effectiveness as possible.

Some use cases that come out as more specific instances of the general use case are the ability to change network attributes in a global manner (deployed across the whole location, company, campus), e.g. L3 addressing. That way administrators have maximum control over addressing, easily avoiding any conflicts and issuing changes by executing non-ambiguous configuration commands. Applying advanced services provided by the Cloud Computing framework to the physically deployed networks, e.g: policy control, firewall, etc, also contribute to the centralization of control and services because nothing else needs to be provisioned for the networks which typically are not under the umbrella of the Cloud Computing framework. Central management of wireless APs, either to attach them to virtualized networks, or networks already accessible by Ethernet ports or even to create temporary Service Set Identifiers (SSIDs) for specific, punctual occasions, is definitely a use case possible to achieve. The ability to merge different physical networks in distant physical locations into the same broadcast domain is an undertaking that makes all sense to be done in a centralized manner, ideally without manually connecting any equipment, and this is made possible by this work as well. Education is another use case benefiting from centralized network control, where the ability to setup networks for use in the classroom, especially useful for Computer Networking lab classes, leveraging most, if not all, of the use cases presented in [38], is a considerable benefit. Finally, easily setting up networks for guests or for research testbeds, deployed on top of any connectivity interface available, just by

accessing a centralized control interface, can be made possible. Figure 4.2 illustrates this use case.

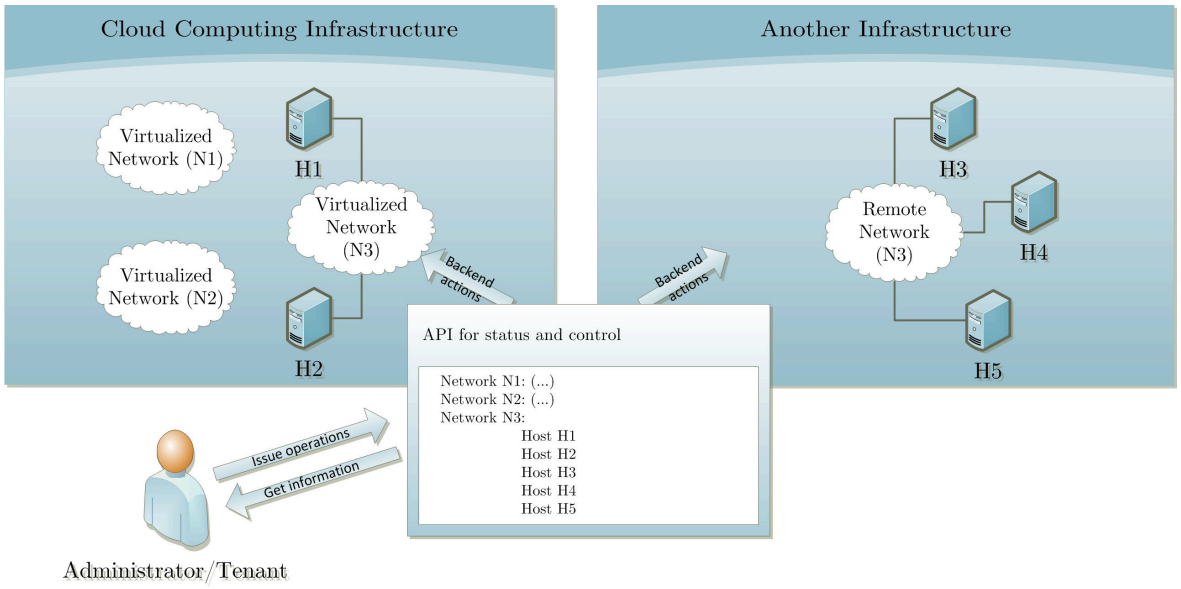


Figure 4.2: Centralized Network Control and Flexibility via unified API

4.1.3 virtual Customer Premises Equipment

Home Gateways (HGWs), Residential Gateways (RGWs), or some kinds of Customer Premises Equipments (CPEs) are used in households to provide users with broadband Internet access via an Internet Service Provider (ISP) as well as modern services like television's channel streams' rewinding. There have been progresses and changes to the typical HGW during the last years to accommodate different access network technologies and new services, although not very substantial. Offloading services usually present in the HGWs to an upper entity, or simply the provisioning of Virtual HGWs (vHGWs) or Virtual CPEs (vCPEs) are possible use cases to leverage by this work.

Some have defended the idea of making an HGW more complex, like Rückert et al. [45] who present an SDN-enabled HGW to enable a new traffic management architecture for advanced services in Digital Subscriber Line (DSL)-based broadband access networks, reducing the amount of total traffic in the core network by setting up proper flows that prevent the Broadband Remote Access Server (BRAS) from sending needlessly duplicated traffic, from ISP-provided custom services, to all users of the same DSL Access Multiplexer (DSLAM).

Others have defended work in an opposite direction, such as [46] which mainly focus on network monitoring and control in order to detect and prevent security issues that may threaten end users of a service, by "outsourcing" HGW management and monitoring to the operator. Cruz et al. [47] present a detailed framework for a fully virtualized HGW/RGW running at the operator side, physically removing the RGW from the customer premises. Per the own words: "This solution potentially reduces deployment, maintenance and operation costs, whilst improving overall flexibility, reliability and manageability, both for the access network infrastructure and for provided services".

With the work presented in this dissertation, HGWs can become simpler, offloading some services to the ISP side, e.g. the LAN DHCP server. Thus, ISPs acquire more control of HGWs' maintenance

and state, which may prove helpful to fight widespread security threats, reduce OpEx, speed up the introduction of new services, etc. Other advantages of this scenario are the lower cost of consumer equipment acquisition, increased energy savings due to the simplistic hardware and reduced CapEx, to name a few. All of this while integrating with a Cloud Computing infrastructure.

A Cloud Computing provider can thus provide Internet connectivity to external hosts of the External Network like any other host, such as a Nova VM instance. Consequently, Neutron can act as an Internet provider to External Ports, a use case that might especially useful for Telecommunications Operators betting on NFV.

Besides HGWs, enterprise CPEs can likewise be integrated in the same scenario, with the same advantages and features like NFV. Figure 4.3 shows a deployment where two kinds of customers are linked to the same Cloud Computing infrastructure.

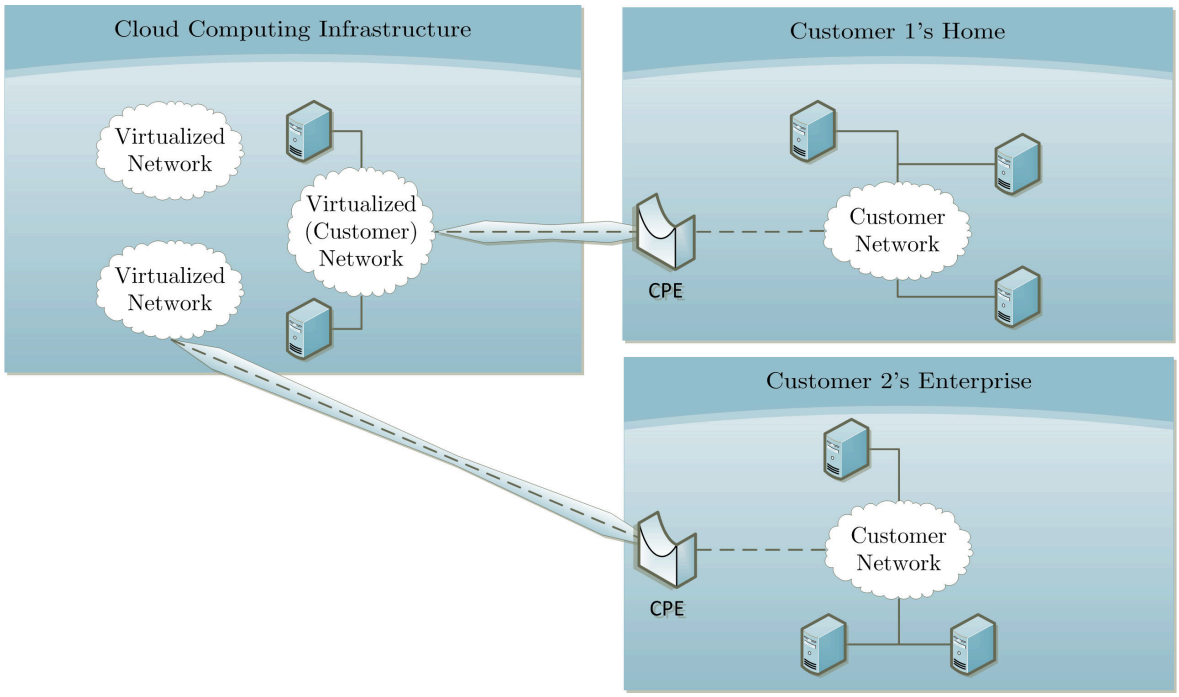


Figure 4.3: Deployment where two kinds of customers are linked to a Cloud Computing infrastructure.

4.1.4 BARE METAL PROVISIONING

A new trend in Cloud Computing stacks is what's called "Bare Metal Provisioning". Due to the high overhead in provisioning virtual machines in an IaaS infrastructure (there have been improvements such as [44]), the ability to achieve heavy workloads, with high performance, becomes limited or compromised. Therefore, on scenarios where an IaaS infrastructure is in place, the need to provide bare metal machines, i.e. operating systems installed on top of physical machines, has increased.

Moreover, bare metal machines require network connectivity as well. Instead of implementing and deploying a specific way to reach and boot these machines via the network, this work allows to reuse the same functionality from VM networking for that end. By adding bare metal nodes via the Cloud Computing administration API with proper configurations, the process of interconnecting these to the rest of the network and making them visible to their respective tenants has been improving but

still lacks major integration, namely with Neutron. The Ironic project from OpenStack as well could benefit from this work by automatically creating attachment points with the correct MAC addresses and switch ports of the connected switch, keeping everything else transparent in Neutron.

4.1.5 SERVICE CHAINING TO EXTERNAL ENTITIES

Since this work allows extending Cloud-provided networks into the outside world (and vice-versa), many different kinds of services may, as a result, be provided externally. Take the case of service chaining, an important NFV use case that can be leveraged. By providing service chains, Service Function Chaining (SFC), to external points, existing equipment and services can be reused (which goes towards the incremental legacy to Cloud migration).

Initially, service chains were deployed in a hard way, i.e., specialized hardware to provide specific services, which were then interconnected in predefined ways to achieve a desired purpose [48]. Recently, with the advent of Network Virtualization and SDN, service chains have become more flexible, with less associated OpEx, while easier to deploy and manage. However, legacy service chains are still the dominant portion of telecommunications operators' core network infrastructures because it is impracticable to easily, and quickly replace them with modern technology [48].

Consequently, a compromise between modern service chains and legacy service chains becomes an interesting possibility. Existing hardware, providing specific services, can thus be reused in a Cloud context alongside "software-defined" service chaining features implemented by recurring to e.g. traffic steering [49], with minimum effort. Depicted in Figure 4.4 is a scenario where services both inside the Cloud Computing infrastructure and the external infrastructure are being chained together. The arrows, along their numbered labels, show the direction and order of the chain amongst hosts.

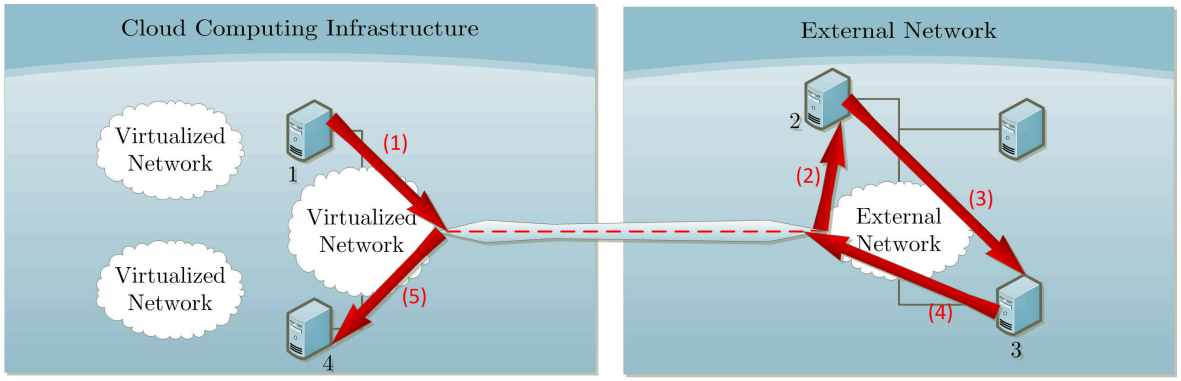


Figure 4.4: Service Chaining with External Entities.

4.1.6 AN ALTERNATIVE TO VPN

In fact, this work has some similarities with the end-goal of a L2 VPN. However, the main difference, and quite a significant one, lies in the flexibility with which tenants can set up extensions to their main networks. By providing all functionality via an API there is less operational hurdle and costs for the service client. It must be noted that this work is not strictly about setting up tunnels

for customers, like modern VPNs without security. Rather, it is a package that improves the array of functionality offered by a Cloud Computing service. A possible end scenario is, thus, the deployment of a VPN with all the major advantages offered by this work: flexibility, control, Cloud integration and compatibility with heterogeneous equipment and technologies.

Bringing the LAN to the Wide Area Network (WAN) is a possible use case for L2 VPNs which allows, for instance, to play a LAN-only multiplayer video game over the Internet.

4.1.7 OTHER USE CASES

Certainly, other use cases can be extract from this work, although some may not differ much in relation to the previous ones. Although, in summary, the main points that this work leverages are that:

- it allows typically virtualized networks to be extended to the outside (and vice-versa);
- it fits into a Cloud Computing infrastructure that provides other desirable advantages like multitenancy or, maybe, service chaining features;
- it allows bringing external services to the Cloud;
- it opens up the possibility to develop advanced services on top of it, like legacy network state monitoring.

CHAPTER 5

Design

In this section, a design is proposed as the solution to the problem previously stated, where technical features are outlined, establishing a clearer interpretation of actual functionality that is provided and how it matches against the end goals, use cases and scenarios depicted and described in Problem Statement 4.

The solution is not standalone, in the sense that it is not supposed to be used by itself to achieve the desired use cases. Rather, this solution is designed to be modular and to integrate as smoothly as possible with existing network virtualization stacks, especially complete Cloud Computing IaaS software stacks that support the former.

There are already multiple Cloud Computing IaaS software stacks that provide network connectivity to tenants, at least for connecting virtual machines between each other, as seen in the Background section 2. Adoption of IaaS Clouds has been increasing, both in providers and consumers [50], meaning that there is a responsibility to keep these services as excellent as possible, while providing new features to users with ever newer requirements. Thus, envisioning a new feature to be added to an IaaS software stack has a higher probability of commercial adoption, whilst avoiding duplicated efforts with existing projects and products. Another, more technical justification for creating a solution that extends a different one is the fact that multiple FOSS IaaS and other network virtualization stacks exist, thus accelerating the development of a solution ready for testing (by implementing on top of them).

5.1 concepts and features

5.1.1 concepts

Network Segment Extension Process (NSEP)

NSEP is the process of extending a network, which typically leaves inside the datacenter, with a remote network. It is a process coordinated by two specific entities which are introduced furthermore in this chapter.

Network Virtualization Stack (NVS)

NVS is the existing software piece. It is responsible for dealing with network virtualization and other aspects. For instance, it can be a complete Cloud Computing IaaS software stack that provides NaaS, multitenancy, and other Cloud Computing attributes.

NVS is the component with which the proposed solution will integrate, leveraging all technical features described for the use cases presented earlier, while providing the rest of the Cloud's features.

External Port Extension (EPE)

EPE covers the entire software piece that results as the implementation of this solution. It is plugged into a modular NVS to leverage the end goals of this work. It extends anything necessary in NVS to fulfill all requirements.

In Figure 5.7, EPE is actually separated from other components which are part of this solution. The reason for this is that, when implemented and deployed, the External Port Extension is both an interface as well as an implementation. Looking at the figure, it is presented as an interface, which respects NVS' requirements. However, implementation-wise, this extension includes all other software pieces necessary to achieve the end goals of adding functionality to the NVS, excluding the latter itself. Also implementation-wise, this extension can be looked at the same level as specified in Figure 5.7, if the subject of discussion is EPE's business logic.

Network Point Controller (NPC)

NPC is part of EPE and manages a set of other resources related to this work, namely Network Attachment Point (NAP) and Network Report Point (NRP), sending and receiving information against them to perform any desired operation in the scope of the work.

NPC can be run in different nodes other than the NVS' one, allowing it to be distributed by design.

It is also the software piece that enforces flexibility and heterogeneity, taking into account information provided by the NVS (extended with EPE's business logic).

It carries out communication, configuration and state reporting functionalities against NAPs and NRPs, being responsible with one side of the NSEP.

Network Attachment Point (NAP)

NAP is the entity that addresses the other side of the NSEP. It is the network's element with which NPC will communicate and configure according to what may be defined on the NVS and EPE's business logic. Flexibility will mostly depend on this entity, which may be limited or empowered by its heterogeneity.

Network Report Point (NRP)

NRP is an entity that reports to an NPC so that the latter can maintain an updated network state, including, e.g, any clients that may connect to it (see Network External Port (NEP)). Like NAP's, the NVS will communicate and configure it accordingly, but will not use it to enforce the NSEP.

Depending on the NAP's capabilities, it may also be an NRP, although this design allows them to be decoupled from each other.

Network External Port (NEP)

NEPs are external ports of the NVS's managed networks. They attach, through NAP and posteriorly NPC, to existing networks. Through these ports normal devices can be connected: computers, servers, etc.

An example of what can be reported via NRP to NPC is when a new NEP is detected, or when they become online/offline.

DRIVER

A driver in this work is what tells an NPC how to translate configurations applied at the NVS to an NAP or a NRP. Besides that, it also presents what options are available, for instance: can the network be extended via an access point, is security supported, is Quality of Service (QoS) adjustable? This solution is designed so that drivers can be developed for any kind of hardware with the potential to extend network segments.

5.1.2 FEATURES

L2 NETWORK SEGMENT EXTENSION

L2 network segments "living" in a virtualized network infrastructure, possibly managed by an IaaS provider, can be extended beyond that basic infrastructure. Taking as basis Figure 4.1, another one, more generic, is presented: Figure 5.1. The interpretation for this figure is that the virtualized network segment is extended beyond the Cloud by bridging it with the remote network segment. Consequently, all hosts, either connected to the virtualized or remote part of the network, will be in the same broadcast domain. This functionality can be achieved in any kind of underlying network, with any number of (underlying) hops. The two infrastructures can be separated by the whole Internet. This is the process called NSEP.

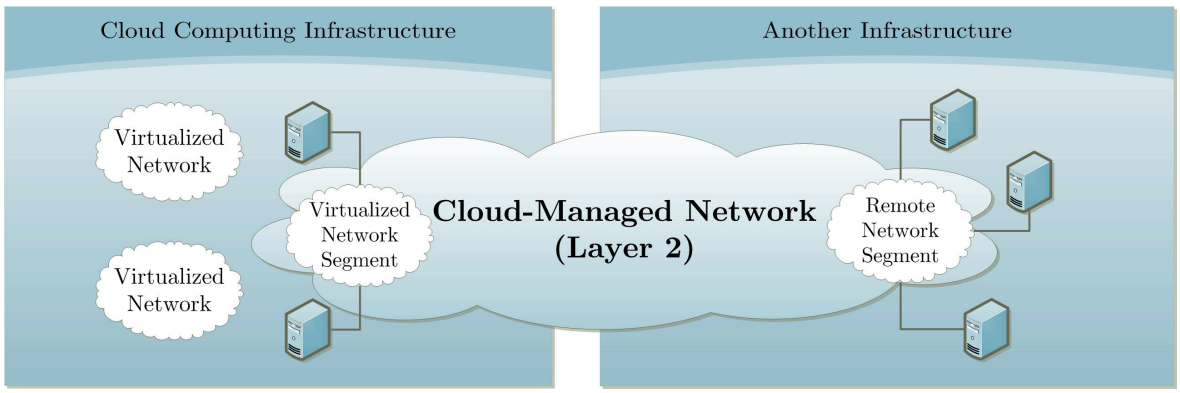


Figure 5.1: A virtualized network segment extends into a remote network.

In order to extend virtualized network segments, the proposed solution integrates and extends existing Cloud technology to send and receive data towards and away from the remote network, while making sure that the remote underlying network carries out reciprocal actions.

LOGICALLY-CENTRALIZED NETWORK AWARENESS

Because this work builds on top of existing network management solutions, like network-enabled Cloud Computing services, it is beneficial to keep the state of the whole network up to date, not just the virtualized part of the network. Given that, this work is designed to be aware of all main entities "connected" to the Cloud infrastructure, beyond the ones traditionally known. Specifically, it has knowledge about end devices which are externally connected, i.e. L3 hosts in the remote network segment, even though they are transparently processed as typical devices (like virtual machines) by the rest of the Cloud Computing software stack (which means no existing functionality is broken). Figure 5.2 demonstrates this transparency by what an administrator should generally be able to see. The work also has knowledge about what remote devices are responsible for the second half of the NSEP, being the first half the responsibility of the Cloud provider. Although named concepts have not yet been formally introduced, this is an important moment to keep in mind the name of two very important kinds of entities in this work: NEP, which represents the L3 devices mentioned, and NAP, which represents remote devices that fulfill half of the process of attaching the remote network segment to the Cloud.

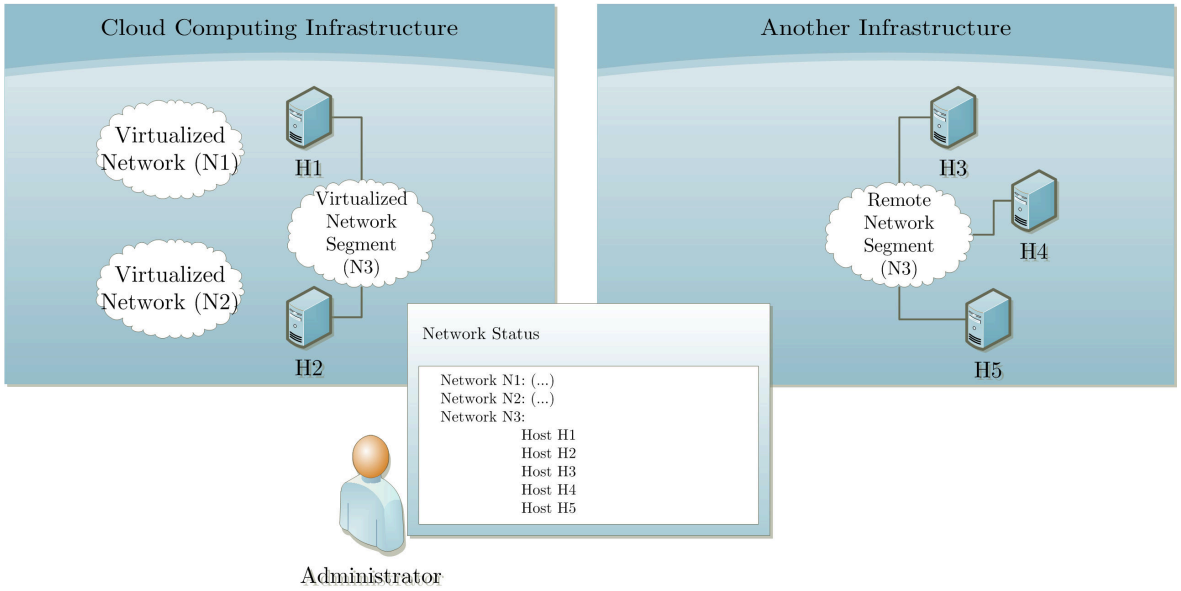


Figure 5.2: An administrator is able to have a complete view over the network.

AUTOMATIC REMOTE NETWORK CONFIGURATION

This solution takes care of necessary configuration actions that need to be taken on NAPs, which are responsible for fulfilling network segment extension at the remote infrastructure's side, so they effectively become operational.

In other words, when a customer requests that a remote network segment be attached to a virtualized network segment managed by the Cloud Computing infrastructure, this solution (which is plugged as a module in the Cloud Computing software stack) automatically follows multiple steps, one of which is connecting to the NAP and executing all necessary actions to attain mutual network segment extensibility. Figure 5.3 this, where the Controller element represents the controller of the Cloud Computing software stack plus the new module.

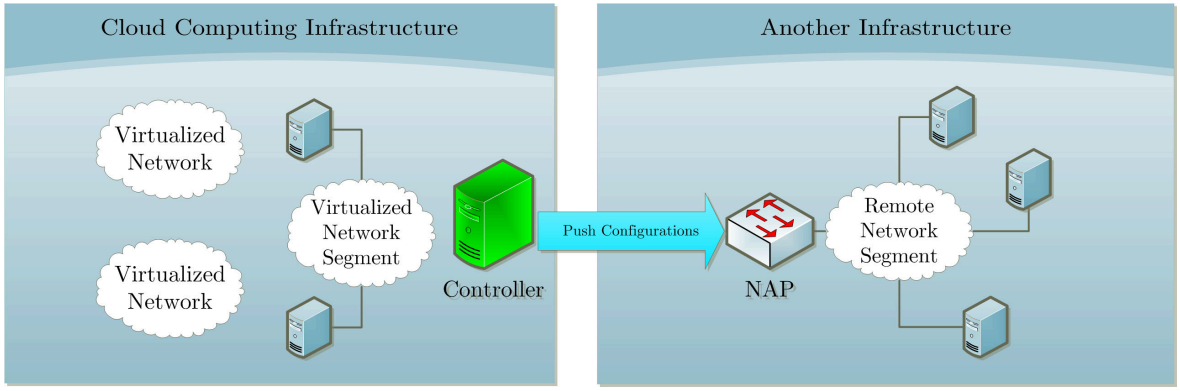


Figure 5.3: Configurations are "pushed" to the NAP so that its network segment becomes part of Neutron.

NETWORK STATE REPORTING

This solution encompasses a network state reporting functionality which allows the report of specific events towards the IaaS provider, as can be seen in Figure 5.4. These reports are useful for reactive changes in the network, security policies, or simply for other administrative purposes. One of the clearest advantages of having this features is the ability to automatically detect that a new host has connected to the remote network segment, and provision any necessary resources (like reserving an IP address).

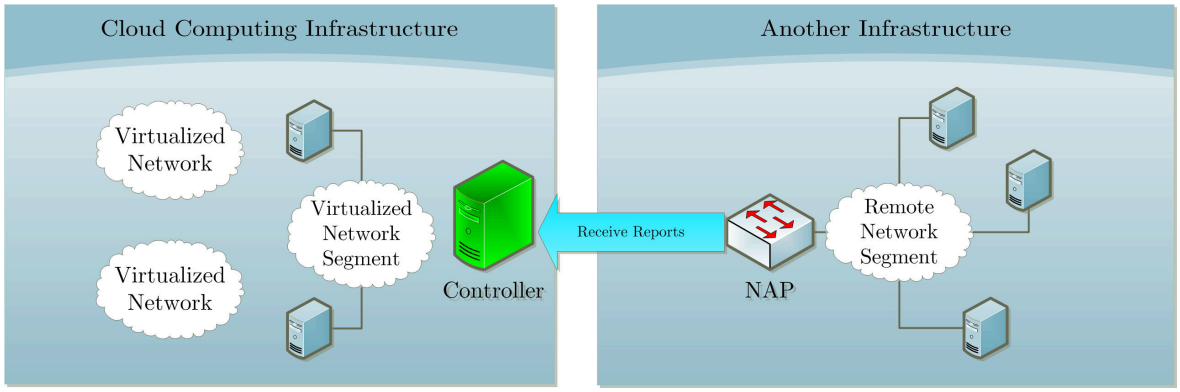


Figure 5.4: Network state reports are obtained from the NRP to be processed by Neutron.

IMPLICIT FLEXIBILITY

This solution is implicitly flexible because it makes available a generic interface for customizing each new network segment, depending on the technologies and configuration options available in each NAP.

An example of this flexibility occurs when extending networks via an NAP that is actually an wireless AP. This solution is flexible to the point of setting up different SSIDs with different security configurations. Another example that can be thought of is modifying a router's routes when it is used as a NAP, so existing networks at the remote infrastructure's side are kept in such a way that coexistence with the new connection towards the Cloud Computing infrastructure is allowed and sometimes improved. Figure 5.5 shows two examples.

Existing advanced services that may be deployed inside the remote infrastructure are also possible to be reused by the IaaS provider.

Segments may be of different networks or the same network.

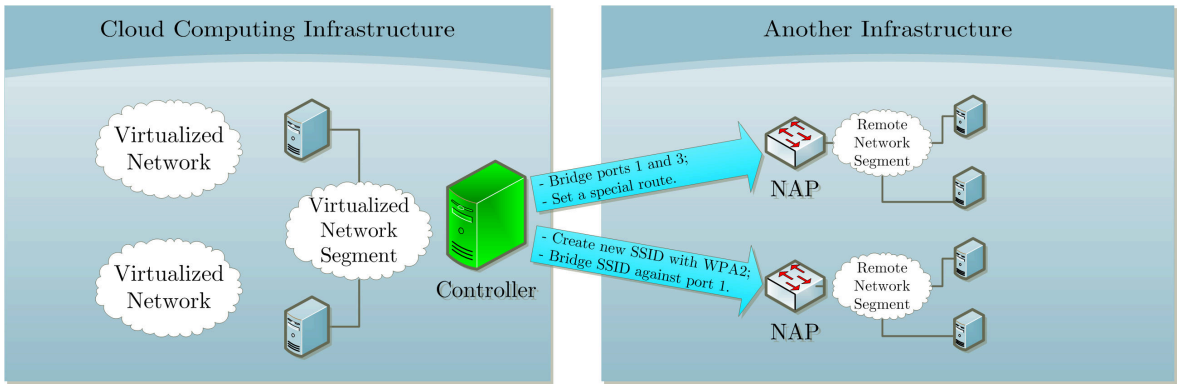


Figure 5.5: Flexibility attained by configuring the NAP with any option supported by its driver.

IMPLICIT HETEROGENEITY

Heterogeneity in this context essentially means that this solution is devised so that it is compatible with virtually any NAP. It is a core feature of this solution. Moreover, it is also designed to be compatible with any Cloud Computing software stack, although implementation-wise specific criteria would need to be analyzed: interoperability and modularity of existing stacks, standards, etc.

The main idea is that the architecture's part which supports NAPs is driver-based and pluggable, as depicted in Figure 5.6. Given that, there is a clear separation between the overall features and their implementation. This enables reusing legacy equipment. Finally, because each driver may announce different customization support, the previous technical feature of "Implicit Flexibility" is leveraged.

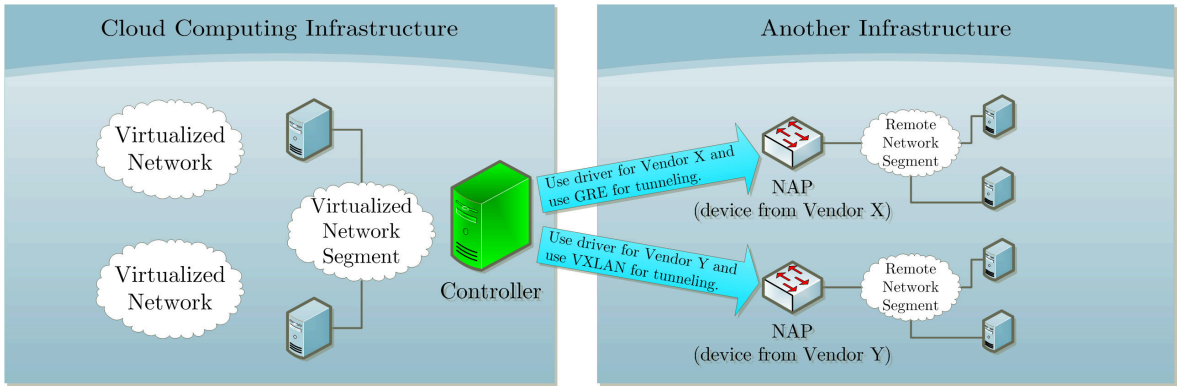


Figure 5.6: Heterogeneity attained by developing drivers for different NAPs.

5.2 ARCHITECTURE OVERVIEW

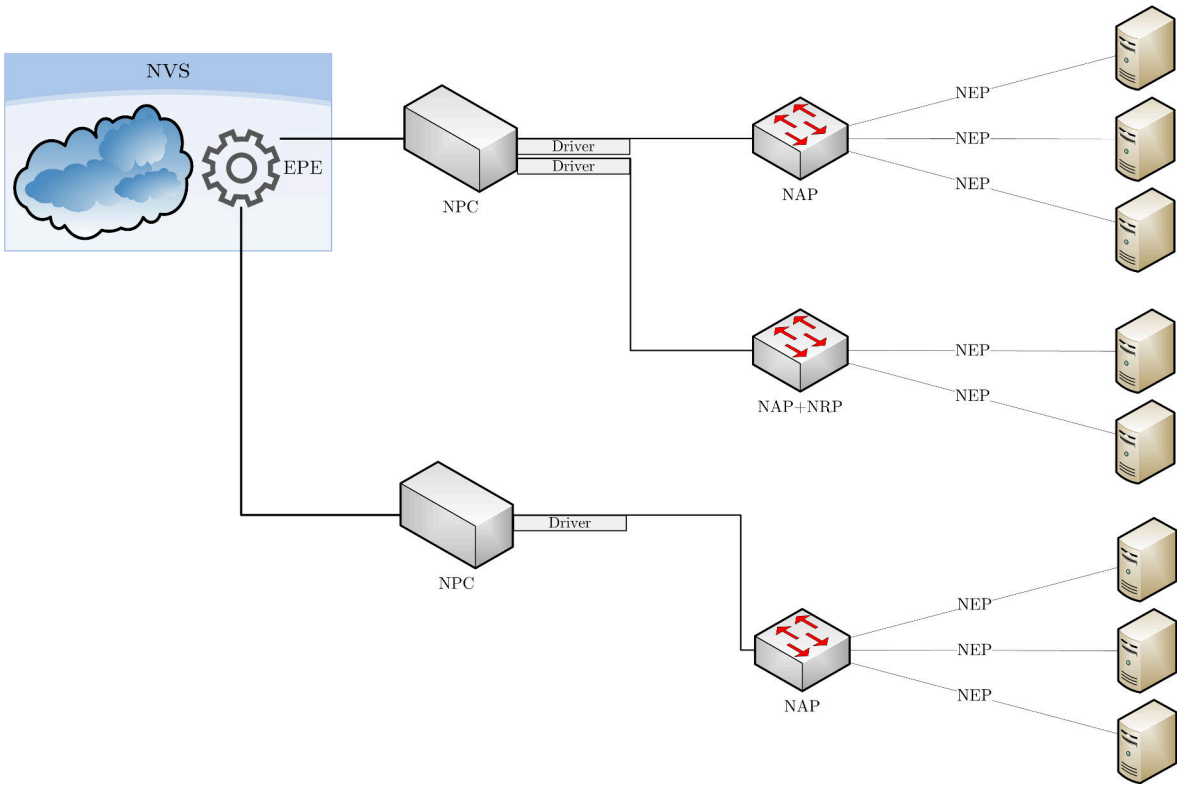


Figure 5.7: An overview of the architecture.

5.3 PROCESSES

CREATING A NEW NAP

The objective of creating a new NAP is to map some device, which will be responsible for providing a network segment to extend an existing network managed by the NVS, to relevant properties like

what network should it extend, how should it be configured and how to communicate with it, how will traffic flow between the two endpoints (NVS and NAP) and any other information that is sufficient for leveraging the NSEP.

In order to create a new NAP users issue a request on an API or express their intention in a User Interface (UI) (which will usually translate into an NVS API call) informing that they intend to create a new NAP resource. Important attributes that need to be provided are what driver to use and where is the NAP located, network-wise. When the resource is created it will be kept in the existing NVS or EPE database (depending on the implementation). Note that the NAP is still not attached to any network in particular, although it may be attached to a specific tenant (again, depending on the implementation or the existing NVS solution).

Having the NAP been created, it is ready to be used in the future, when the need to extend a network segment emerges. However, until then, nothing is ever communicated to the NAP.

To illustrate the process, Figure 5.8 shows a generic sequence operations to be carried out.

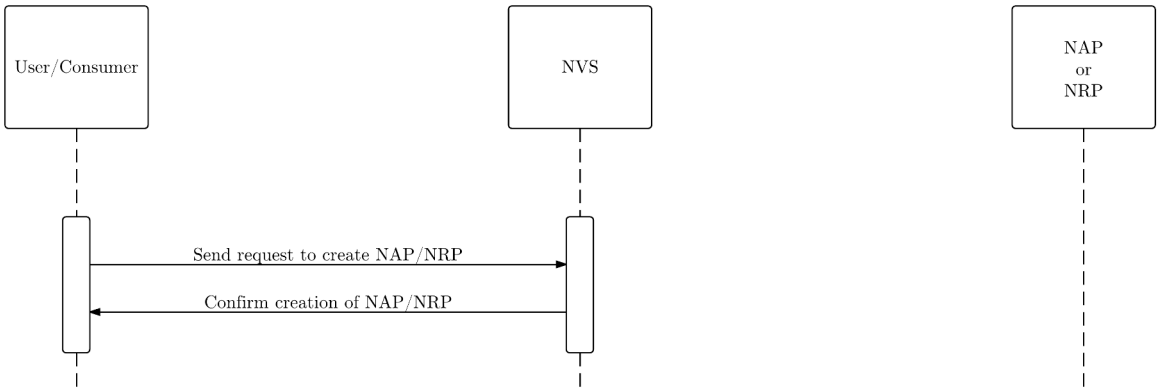


Figure 5.8: Sequence diagram illustrating a NAP being created.

CREATING A NEW NRP

The NRP is essentially a NAP although not being responsible for extending the network, only reporting its state (in some manner). All other aspects from creating a new NAP apply here. NAPs and NRPs can be derived from the same device, simultaneously. Given the similarities in workflow when creating a NAP or a NRP, Figure 5.8 illustrates this process as well.

EXTENDING A NETWORK SEGMENT

Users, via the available user interface, request that some network be extended. They specify what NAP should be attached to the network, so that the latter is extended through the former. What happens then is that the NPC will initiate a communication with the NAP, via whatever method is allowed and configured in the driver, and execute any necessary remote operations to make the NAP ready for extending the desired network segment. Finally, communication is brought down and the NAP becomes responsible for extending the network segment (at its side). Approximately at the same time, EPE's mechanisms (either existing or extended ones) will manipulate NVS so that it carries out its responsibility of extending the network segment against the NAP as well.

Like previous processes, this one also requires users to explicitly request the operation (or an orchestrator to automate this process). Figure 5.9 illustrates what generic operations are involved with extending a network. It all starts with the operation requested by the user, which becomes aware of its effectiveness as soon as the attachment process initiates. NVS gives the order of attachment to the NPC and, based on the properties assigned to the NAP previously created (at the process described in 5.3), NPC instantiates the correct driver for communicating with and configuring the NAP. The driver does exactly that and, when ready, traffic naturally starts flowing between NVS and NAP, directly. To clarify, this traffic is part of the network whose segment(s) has/have been extended. In other words, traffic between hosts connected to NVS hypervisors (the first L2 segment of the network) and hosts (NEPs) connected to the NAP (a second L2 segment of the network). Grey arrows in Figure 5.9 indicate the ability to detect exactly when a network segment extension becomes operational, which is a secondary objective to be leveraged by this design.

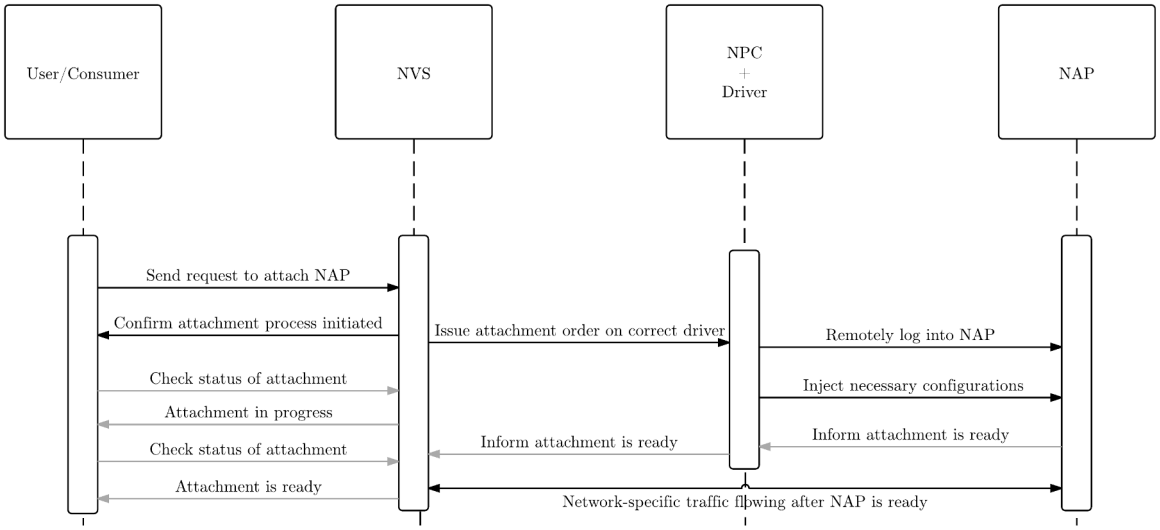


Figure 5.9: Sequence diagram illustrating a network being attached to a NAP, extending the former.

DETECTING A NEW NEP

One of the main reasons for the existence of NRPs is to detect new NEPs and act in NVS so as to carry out tasks like allocating new resources, updating a firewall, or creating a port similar to what would be created for a virtual machine. It all depends on the implementation. In terms of actual process, it is done in the following way: after the NRP has been properly created and configured, it will either notify the NPC or respond to it periodically (by polling), sending new events. One of these events may be the detection of a new host, which is then processed by the NPC and further sent upwards, so EPE and NVS apply any necessary operations to make the new host effective in the network. Figure 5.10 illustrates this scenario, where a NEP connects for the first time to the NAP and it alerts this event back to the NPC.

Another, completely different manner of detecting a new NEP, anticipated by this design, is to provide functionality internal to the NVS (or EPE) to automatically detect new NEPs and provision any necessary resources and actions related. This method is illustrate at Figure 5.11.

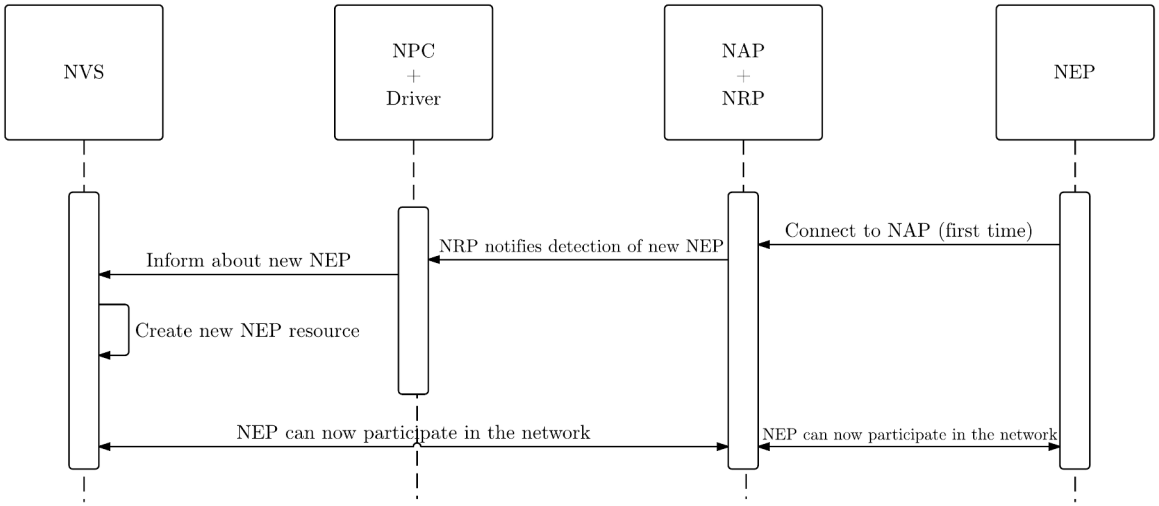


Figure 5.10: Sequence diagram illustrating how a NEP is detected, recurring to the NAP driver.

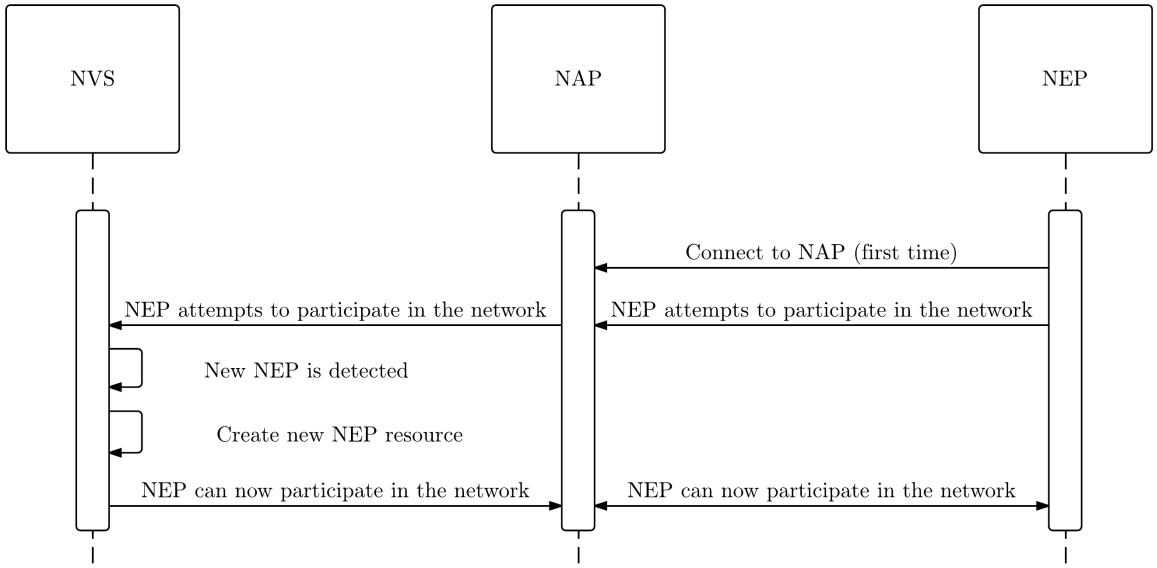


Figure 5.11: Sequence diagram illustrating how a NEP is detected, recurring to internal detection.

CHAPTER 6

Implementation

This chapter presents the work beyond design. An implementation is proposed based on the design discussed in the previous section. OpenStack [1] has been chosen as the platform on top of which this solution is implemented. More specifically, the OpenStack NaaS project, Neutron, is used as the code-base for everything developed and hereby presented.

6.1 OpenStack

Before proceeding, it must be clear why there was a decision on using OpenStack/Neutron as the underlying enabler for the work presented in this section. OpenStack is a FOSS project, meaning that, besides other advantages, access to current code is provided for free. It is the most popular FOSS Cloud Computing software stack [22], driven by a very active community, where people can also contribute to it individually. Besides that, it is designed to be extended and extensible.

6.1.1 A deeper look at ML2

In section 2.2.2 a basic and general description of Neutron and ML2 was presented. In this sub-section the focus is to acquire more knowledge about the inner works of the ML2 plug-in and the OVS mechanism driver, on top of which the work presented was developed. By providing a deeper look at these components, it is expected that the solution presented becomes easier to understand.

The ML2 Neutron core plug-in can be activated like the rest of Neutron's core plug-in, via the main Neutron configuration file, `neutron.conf`. What the plug-in will effectively do is translate the logical API requests that Neutron receives into a back-end implementation that the plug-in symbolizes. So, when extending the Neutron API it is expected that the plug-ins' interfaces will also be extended, so they implement the new features added to Neutron by translating the new API requests to new back-end operations. However, due to the number of available plug-ins and different staff responsible

[1] `http://www.openstack.org/`

by their maintenance, not all plug-ins can support every available extension at any time. Consequently, each plug-in explicitly announces what Neutron API extensions they support.

It has already been addressed what a non-extended plug-in must implement in its back-end: networks, subnets and ports. The plug-in must make sure that these abstractions are actually mapped to a real deployment, so tenants can get the service they requested for. Each plug-in implements the core networking resources and API calls, each one at its own way: be it directly on top of a physical network with Network and Compute Nodes interconnecting virtual machine instances by some form of network virtualization (VLANs, GRE tunnels, etc) or through specialized, proprietary hardware and software from specific vendors (again, some form of network virtualization), e.g.: the VMware NSX platform plus Cisco Nexus [2] switches.

When using the ML2 plug-in, the back-end implementation is actually inside the mechanism driver. So, the mechanism driver must, too, implement any new API requests provided by an API extension, if it is going to support these new operations.

What can be seen so far is a consecutive forwarding of operation calls between each entity from top to bottom. Until reaching the mechanism driver, no extremely important operations take place internally, besides calling each modular entity in the level exactly below. When the mechanism driver is called, however, important aspects which relate to the actual implementation take place, and these differ between mechanism drivers.

Open vSwitch mechanism driver

Only the OVS mechanism driver is presented in here because it is the basis for the solution developed and is one of the mostly deployed Neutron core plug-ins, especially on experiments and testbeds, due to its low difficulty in setting up and scarce requirements (computers running relatively new versions of the Linux kernel and average hardware specifications).

Neutron OVS-based network deployments have the set of Neutron processes/services, like `neutron-server`, which are usually present in the Network Node, as well as new ones like the OVS agent. This new agent is a process which manages OVS resources and is present both at Network Nodes and Compute Nodes. A Neutron network deployed on top of multiple nodes using the ML2 OVS mechanism driver may have instances of the same Neutron-managed network spread on different Compute Nodes. As such, communication inside that network may cross these different nodes, including the Network Node when the hosts want to reach a virtual router or the External Network.

The OVS agent is responsible for dealing with OVS bridges, which in some ways are similar to Linux bridges but provide more advanced functionality, configuration interfaces and support for OpenFlow. Linux bridges are also supported in ML2 via the LinuxBridge mechanism driver. In a typical deployment of a core Neutron network deployed on multiple nodes, from now on assumed to consist on 1 Cloud Controller, 1 Network and 2 Compute Nodes, using GRE as the tunneling technology for connecting different Compute Nodes, the following OVS bridges are created within Network and Compute Nodes to form the virtual network topologies requested by tenants:

- `br-int` or the *Integration Bridge*: The first bridge, `br-int`, is the fundamental one regarding network connectivity between instances of the the same network. Conceptually speaking, this bridge is analogous to a typical network switch which interconnects different hosts to the same broadcast domain via its Ethernet ports. In this case, these hosts are actually virtual machine

[2] `http://www.cisco.com/c/en/us/products/switches/cisco_nexus_family.html`

instances or other resources that can be bounded by OVS, e.g. a virtual DHCP server or a virtual router interface provided by Neutron and its plug-ins. Being bounded by OVS in this sense means that the resources will be attached to the OVS bridge, by software, so they effectively become part of the network the bridge refers to. More detail on how the resources are actually attached to the bridge, and which ones can be bounded via OVS, is presented afterwards. In order to differentiate between networks, `br-int` uses internal VLANs in each of its ports depending on the network each instance is connected to.

- `br-tun` or the *Tunneling Bridge*: The second bridge, `br-tun`, exists solely when using the GRE tunneling technology, though otherwise an analogous method would be carried out (such as when using VLANs to interconnect different Compute Nodes). The objective of this bridge is to extend `br-int`'s scope to multiple nodes. As such, what happens is that `br-int` becomes distributed amongst the multiple Network and Compute Nodes, so that all the networks which interconnect all the instances can be spread amongst these nodes, being only isolated via the internal VLANs used by `br-int`. The way `br-tun` extends `br-int`'s scope to other hosts is by bridging it with the rest of the `br-int`s present at other nodes. Each node has a `br-int` and a `br-tun`, where the latter is the aggregating element of the former into the rest of the `br-int`s present in the network, by having a virtual patch interface connecting them both. How the technology for bridging all `br-int`s via `br-tun`s actually works is explained afterwards in a detailed manner.
- `br-ex` or the *External Bridge*: The final bridge, `br-ex`, provides access to the External Network. This bridge does not directly connect to any other, like what happens between `br-int` and `br-tun` and their virtual patch interfaces. Rather, this bridge connects to a virtual router provided by a Neutron L3 plug-in so the latter can provide access to the External Network, as well as NAT and features like floating IPs.

TRAFFIC FLOW

Figure 6.1 provides a comprehensible starting point to understand the underlying mechanism of bridging different virtual networks on top of a physical network with multiple network and Compute Nodes by making use of the previously introduced OVS bridges deployed by OVS agents. There are OVS agents running on each Network or Compute Node, in order to manage a set of OVS bridges for the purposes previously reported. In the case of Network Nodes, all kinds of bridges will usually be created unless a) there is no External Network and, as such, no `br-ex` or b) the set of Network or Compute Nodes are just a single node in which case there is no need to instantiate a `br-tun`.

Starting from the top-right corner of Figure 6.1 the Network Node is found, which has at least two native network interfaces, `eth1` and `eth2` in the example provided by the figure. Interface `eth2` connects to an External Network to provide external connectivity to the Network Node itself, as well as to any virtual machine instances that require access to it. Although managed by the operating system, the OVS bridge `br-ex` is set up to connect "directly" to the network which is provided by `eth2`. In order to enable this, `eth2` must be set up to act in promiscuous mode, so that its traffic does not get consumed by itself, but rather sent to another level of the operating system's networking stack, which then forwards everything to OVS's control, reaching `br-ex`. That is the reason why some connectors are plain while others are dashed when connecting bridges. The plain ones match exactly

where each bridge is connected, while the dashed ones point to interfaces which forward traffic related to the bridge in regard.

With external traffic having reached `br-ex` via an OVS port prefixed by `qg-`, it is then received by the virtual router set up to interconnect with the External Network. There are other possible setups in OpenStack, for instance when multiple, distinct External Networks exist, or when an external router is used (called a Provider Router), or even multiple routers in a per-tenant fashion. All of these special cases could be further explained, however their relevancy for this work is limited so the reader is invited to obtain more information via the OpenStack Manuals [51]. The virtual router is implemented by a Neutron L3 plug-in and is usually presented with an L3 agent which sets up iptables [3] according to the routing setup, Neutron networks created and any floating IPs created, fulfilling both SNAT and DNAT. Eventually, traffic (now NAT-translated to the correct Neutron network) gets received by `br-int` via an OVS port prefixed by `qr-`. At this point, traffic already has the correct destination IP for the internal Neutron network, so from here on the actual Compute Node which will receive the traffic (and GRE tunnel used throughout) will depend on the destination.

At this point, `br-int` must make a decision on where to forward the traffic received. To simplify, consider from now on that it is a single packet. It was received via a specific `qr-` interface, which is assigned to a specific VLAN tag inside `br-int` (one virtual router interface per Neutron network, consequently one `qr-` port in `br-int` per each virtual router interface, with a specific VLAN tag). Network Node's `br-int` knows that to reach the destination of the packet (after using Address Resolution Protocol (ARP)), it must go through the `patch-tun` port, so the packet is sent there. This port is a virtual patch interface that "bridges" both `br-int` and `br-tun` together and is a VLAN trunk port. It reaches `br-tun` via `patch-int` (equivalent to `patch-tun`), which has mappings from internal VLAN tags to GRE tunnel IDs (in OpenFlow). Based on the VLAN tag of the packet received at `br-tun` (remember that the patch interfaces are trunk ports), the packet is encapsulated with a GRE header and a specific key (based on the internal VLAN tag) via OpenFlow actions and sent through the `br-tun`'s GRE tunnel port that matches the specified GRE tunnel key. Traffic will finally exit Network Node via `eth1`. The destination of this tunnel is the Compute Node that hosts the destination of the packet.

Moreover, the packet cross the Data Network, encapsulated by GRE until it reaches the correct Compute Node. It must be noted that the kind of GRE used in this OVS deployment is meant for Transparent Ethernet Bridging [29] which mainly allows any protocol to be encapsulated on top of Ethernet. That makes GRE an important candidate for bridging remote Ethernet segments. At the Compute Node, the packet first crosses `eth1` and is then matched by one of its `br-tun`'s ports, after the operating system processes the Ethernet layer and passes control to the application responsible for dealing with the GRE protocol, in this case OVS. Now, `br-tun` carries out the opposite of what Network Node's `br-tun` did. It matches the received packet with the GRE port that is set up for the same GRE Key [52], which makes the packet effectively enter `br-tun`. Then, the proper flow is matched by OpenFlow, and the actions of de-encapsulating the real packet from GRE and forwarding it to a specific VLAN via `br-tun`'s' `patch-int` port (a trunk port) are carried out. The packet is then received in `br-int` and, knowing what VLAN it belongs to, is then forwarded to the port belonging to the correct virtual machine instance (a port prefixed by `qvo`). Reaching the instance, however, is also not atomic. The `qvo`-prefixed port leads to a Virtual Interface (VIF) which is provided by a virtualization driver enabling OVS to reach the Virtual NIC (vNIC) of the virtual machine instance.

Port binding consists in linking together a `br-int`'s `qvo` port with the instance's port it represents,

[3] `http://www.netfilter.org/projects/iptables`

making sure traffic flows both ways. ML2 has specific methods for dealing with port binding, which then calls the mechanism driver, so that the binding becomes effective. Internally, it matches the type of network and host running the virtual machine and, if successful, the binding details are created and the VM's VIF, as exposed via a VIF driver (e.g. Libvirt), becomes linked to the OVS's port. Kukura and Mestery provide deeper technical insight into these details [53].

In addition, due to security groups there are other "hops" between `br-int`'s `qvo` ports and actual virtual machines due to the fact that iptables cannot be applied directly to these ports. However, this fact is not related to the work presented.

A final scenario for traffic flow that needs to be looked at is how internal VM traffic exits its Neutron network and heads out the External Network. Until reaching the Network Node, the process is the same as already explained. Nevertheless, at the Network Node, a packet heading to the External Network will reach `br-int` and then sent to the virtual router. There, NAT rules matching any floating IP configurations will be applied and the traffic will exit with the pre-configured floating IP assigned to the source instance.

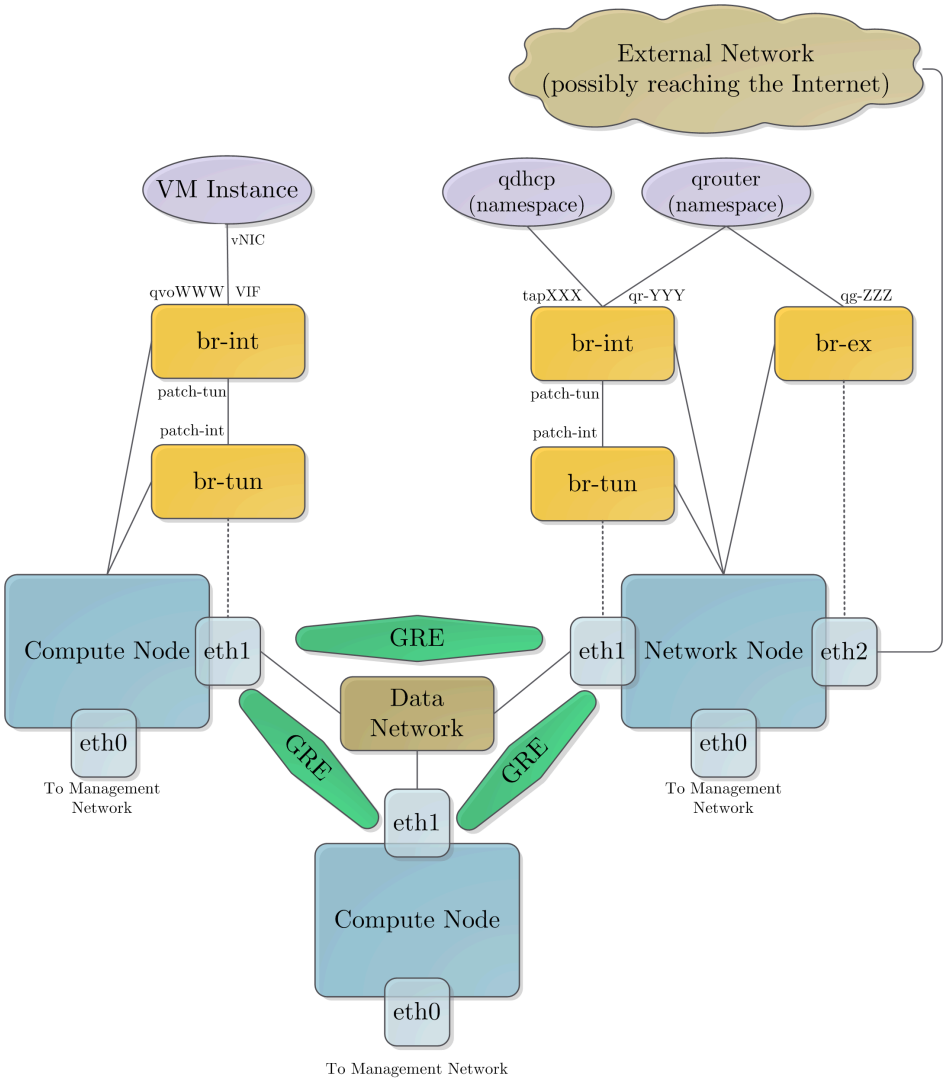


Figure 6.1: An L2 network deployment using Open vSwitch with GRE

COMMUNICATION

All communication between different nodes in a Neutron deployment is conceptually made through the Management Network. In the case of an ML2-deployed network with the OVS mechanism driver active, and other typical plug-ins active, like the L3 plug-in for routing to the External Network, some processes are expected to be running and dealing with communication between other nodes and their processes are. Figure 6.2 shows the most important processes/agents and what kind of communication occurs between each node.

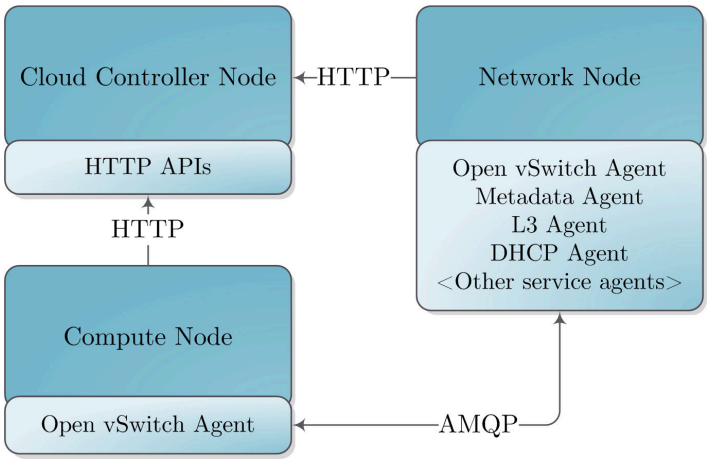


Figure 6.2: Principal OpenStack nodes and agents.

Not all agents communicate with other agents/nodes. All of them actuate in some way in the node they run, for example the DHCP agent instantiates `dnsmasq`[4] to provide IP addresses to the instances connected in each network. However, the OVS agents present in the Compute Nodes need to communicate with the Network Node by message queuing via RPC, so they can fetch the state of the network as mandated by the Neutron API, accessible via the Cloud Controller Node, evidencing another method of communication between nodes.

6.2 REQUIREMENTS

Besides fulfilling or providing a base to leverage all use cases presented earlier, the specific implementation on top of OpenStack should satisfy some requirements.

6.2.1 GENERAL REQUIREMENTS

In general, the implementation should satisfy five important, mainly non-functional requirements.

Reuse existing functionality:

Existing functionality should be reused at Neutron, by keeping compatibility with existing advanced services, APIs and other Cloud Computing features. Virtual routing, DHCP server or access

[4] `http://www.thekelleys.org.uk/dnsmasq/doc.html`

to the External Network are a few services and use cases that usually exist in Neutron deployments, and likewise should exist for this implementation as well. Future functionality, for instance to enable SFC should be able to make use of this work, without carry out any future modifications to it. Consequently, this work must make use of existing interfaces as much as possible so it becomes as transparent as possible to the rest of Neutron. Furthermore, all other OpenStack services should be able to work transparently against this solution.

Achieve average traffic performance:

Achieving an average/good level of traffic performance and latency when compared with a traditional solution is important or, otherwise, it cannot be deployed for production except in scenarios of very low performance requirements. Specific numbers for this are not provided, although throughput of at least half of what can be achieved between VM instances is considered a positive result. Similarly, a latency of at most double what can be achieved between VM instances is a positive result.

Provide a clear UI:

Providing a clear, succinct and light UI to tenants and administrators is an import aspect because it avoids misunderstandings and improves efficiency and agility for administration tasks, potentially reducing OpEx costs. OpenStack CLI clients follow human interface guidelines [54], therefore this work should respect it as much as possible, elevating consistency and predictability especially for existing OpenStack administrators and operators.

Meanwhile, it must be complete and flexible so tenants and administrators can explore all advantages and use cases from the CLI, to the extent that providing a clear UI is not endangered.

Guarantee quick action times:

It is important to guarantee quick times for setting up and destroying (tearing down) network segment extensions. Traditional networks need some minutes to have their topology changed (by manually swapping cables) or their addressing changed (by manually configuring devices), in the best case scenario. They need some days or weeks for more complex topology changes, in the worst case scenario. With the work presented, each action that changes network segment extension should be kept and the order of seconds or tens of second, in the worst case scenario.

Respect core OpenStack:

The work should follow architecture modularity and extensibility as evidenced by the existing OpenStack core and its components. Code styles [55] and Neutron development guidelines [56] should be respected.

6.2.2 DEVICE REQUIREMENTS

Devices, like Switches, to be used for the NSEP require special functionality. First of all, they must be able to operate at L2, like all typical Switches and some Routers configurable at L2. They need to be remotely reachable, from where OpenStack is deployed. Remote administration must be possible, by making use of some communication and configuration protocols. The ability to use a tunneling technology, or otherwise another technology that allows traffic to be kept in the same broadcast domain while traversing the Internet, is necessary inside the Device. It is required to allow reconfiguring the

internal Switch to manipulate what subnets are to be provided for each physical interface (Ethernet, Wireless SSID, etc.), usually by the use of VLANs although not necessarily. Finally, devices must be able to associate the tunnels created to the subnets, so internal traffic meant for the other part of the network (at the OpenStack side) can be sent there (and vice-versa).

6.2.3 PROVIDER ROUTER

Provider Router is a concept originally proposed by Filipe Manco in one of his OpenStack blueprints [1] to register external routers to OpenStack and make use of them. Afterwards, this concept was further developed in the context of this dissertation and resubmitted to OpenStack in the new Spec format [5]. Appendix A contains the latest version prior to converting it to the Spec format. This blueprint aims at providing a way to map and register distant non-OpenStack routers to the Neutron database, allowing them to be used by tenants with complete API support. Such functionality eases integration and on-demand connectivity between legacy network infrastructures and Cloud-provided virtual networks, also useful for incrementally migrating from the former to the latter, amongst other use cases.

CONCERNS

This blueprint effectively was the basis for the rest of the development proposed in this dissertation. During this dissertation's timespan, some concerns and difficulties were raised and mentioned regarding the reachability and deployability of the solution, which are still valid for the work that emerged afterwards. These concerns are addressed in the next topics, with all concepts adapted to the final work's ones (see also 6.3.1), followed by conclusions and a final decision on what was done.

Attachment Device (AD) Configurations:

The Driver needs configuration parameters to each of the ADs it will be communicating to, e.g. credentials. Making these configuration parameters part of the Neutron API extension for External Ports is not feasible or, at least, a good-practice as device-specific configuration options should not pollute the global API. Having the configuration parameters, for each AD, inside the actual External Agent's configuration file means duplicated effort when adding an AD to Neutron. The reason for this is that it is going to be added both by the API and by editing the External Agent configuration file with its configurations. Furthermore, this automatically hinders the possibility of adding these devices per tenant. In conclusion, this is situation with multiple possibilities and clear trade-offs against each one. The decision for the dissertation's work was to set AD's configurations via the API, thus leveraging more flexibility and functionality.

Classless Inter-Domain Routing (CIDR) Overlapping:

This work does not take into account CIDR (IP network ranges) overlapping (at the AD side) and eventually loss of reachability from Neutron to the AD when tunnels are established end to end (necessary for connecting the remote network segments with Neutron). For this concern not to be a problem, one must assume there is already a network, at the remote side where the AD is, dedicated

[5] `https://review.openstack.org/91925`

to reach it and exchange traffic. For an old legacy network, this might mean that some part of the network requires manual changes, possibly being removed altogether so the AD has a dedicated port to remotely communicate with Neutron, although this might break addressing and reachability inside the network requiring even more changes. In case there is no need to keep addressing and reachability between legacy networks unchanged, the AD only needs to be already reachable. In the original blueprint, the assumption was that the AD was a router, so there was the concern that virtual networks could not overlap any other legacy networks' CIDRs as well. However, with the generalized work further developed, the ADs are usually Switches which support CIDR overlapping by making use of, e.g. VLANs. The network through which the AD is accessed is called "Migration Network" (for the use case of migrating legacy network to the Cloud). It is the network through which tunnels are established (at the AD), and through which Neutron reaches the AD. This network has a different CIDR from the legacy network (virtual afterwards), and from any other networks connected to the AD either virtual or legacy, and is directly connected to the AD.

NAT:

If NAT is used at the legacy network, it may be difficult to or just undesirable to reconfigure the network in a manner that then allows it to be extended into OpenStack. This is a problem that requires a work-around but does not preclude this work from being deployed.

DHCP:

DHCP addressing can be provided either internally (from OpenStack) or externally (from the legacy network). However, care must be taken to avoid conflicts and make sure control is logically centralized, ideally inside Neutron.

Broadcasting:

Broadcast packets may carry expensive overhead given the disparate locations of the virtual and legacy parts of the network, plus the tunneling technology used to connect them.

6.3 ARCHITECTURE

6.3.1 ENTITIES

This implementation is split in multiple entities, which relate back to design elements. Table 6.1 summarizes what entities have been developed for this work and how they map to design elements.

Design	Implementation
NVS	OpenStack Neutron.
EPE	External Port API Extension developed for Neutron and ML2.
NPC	External Agent, a Neutron agent.
NAP	Attachment Point plus (Attachment) Device.
NRP	Reporting Point.
NEP	External Port.
Driver	Driver

Table 6.1: Implementation components mapped to design elements

OPENSTACK NEUTRON

Neutron is the NaaS part of OpenStack, an FOSS Cloud Computing software stack. It is flexible, modular and extensible. This work sits on top of Neutron by extending in a recommended way: by creating an API extension.

EXTERNAL PORT EXTENSION

External Port Extension has been developed for Neutron and the ML2 plug-in. It starts as a Neutron API extension, which allows a plug-in to extend the basic Neutron API to provide more functionality. It includes all code necessary to make this work a reality, from the API to internal interfaces to actual functionality. So, all following implementation components are part of this, more general, component. In terms of ML2, the Open vSwitch mechanism driver was used to leverage the process of extending a network segment (at Neutron's side).

EXTERNAL AGENT

The External Agent is a new Neutron agent and, like other Neutron agents, communicates via RPC and is saved as a new agent resource in the database as well, which is available for querying via the API. This agent is, thus, an individual process which may theoretically run at any OpenStack node assuming it can reach remote Attachment Points and vice versa, given that the only other constraint is the fact that it needs to communicate with OpenStack's controller node via RPC. Because this is the component which communicates with Attachment Points to exchange state and configurations, it is the one having access to drivers that further translate abstract operations into devices' configurations.

Attachment Device

An Attachment Device (AD) is a new type of Neutron resource. The actual device that is pointed by this resource can be any switch, router or other equipment with network bridging capabilities and a remote management interface. Resource-wise, it symbolizes a specific device, assigned to a specific driver in order to translate any operations, that is reachable via the IP network. The AD is through where Attachment Points are provided.

ATTACHMENT POINT

It must be clarified that an Attachment Point is a logical entity, not a physical one. In other words, this entity is provided by another, physically traceable entity: an AD. The AD does not need to be physical, but is not plainly logical neither. Attachment Points provided by ADs can also, sometimes, be seen as physical, traceable entities (like a network switch port) although, in an abstract manner, it is simply a logical point where a Neutron network can be extended (into another segment that broadens the broadcast domain). This entity is also known as a Layer 2 Gateway (L2GW) [57].

REPORTING POINT

Any switch, router or other equipment supporting a reporting protocol, like SNMP and a remote management interface. There is not any new resource type created for this entity. Rather, Attachment Points will have a reporting attribute to fulfill the same objective, in this implementation.

EXTERNAL PORT

An External Port can be any external IP host: a computer, server, smartphone, sensor, camera, etc. It is a new kind of Neutron resource. In this work, these resources are mapped to core Neutron ports. The main reason, in a summarized manner, can be attributed to the fact that administrators or tenants should have a clear control over their External Ports, without dealing directly with core Neutron ports, improving flexibility whilst keeping existing Neutron resources as simplistic as possible.

A note on tunneling External Port's traffic:

When using GRE as the tunneling technology, External Ports must be configured with a custom Maximum Transmission Unit (MTU) of (at most) 1458 bytes. Otherwise, packets larger than that will automatically be dropped as they do not fit inside the GRE tunnel. Using this implementation, the correct MTU can be sent via Neutron's DHCP server so External Ports automatically set that in their interfaces, enabling packet fragmentation beyond that threshold.

DRIVER

An actual software driver that the External Agent instantiates given what is defined in Neutron's database. The database keeps track of what Attachment and Reporting Points have been created, and whether they are attached to some network. These "Points" are provided by ADs, which require a driver to translate configurations and operations to the interfaces they support. They also announce what technologies and configuration options the ADs supports. Finally, a Driver will also keep network state monitoring running properly, in regard to each Reporting Point managed by it. Two drivers have been developed and tested in this work: Cisco EtherSwitch IOS Driver (`etherswitch_driver`) and OpenWrt OVS Driver (`openwrt_driver`).

Figure 6.3 provides an overview of the architecture in a deployment perspective. It builds upon Figure 5.7 presented earlier at the Design chapter, although with improved detail, and its components now mapped to the implementation.

Starting from left to right in Figure 6.3, or top-bottom in terms of actual architecture, the first element to be encountered is OpenStack itself or, more precisely, a set of its abstractions. Given that OpenStack as a whole is not a major concern regarding this work, besides the fact that existing services should continue to exist (authentication, multitenancy, amongst other services and general operations), focus can be provided directly to Neutron. Neutron is the project that is actually extended by the solution proposed, and consequently lies at the top of the architecture hierarchy.

The first change made to Neutron is an API extension, providing support for new network operations. These new operations rest on top of Create, Read, Update, Delete (CRUD) calls on new resources of the API. The new resources are the components Attachment Device (AD), Attachment Point and External Port.

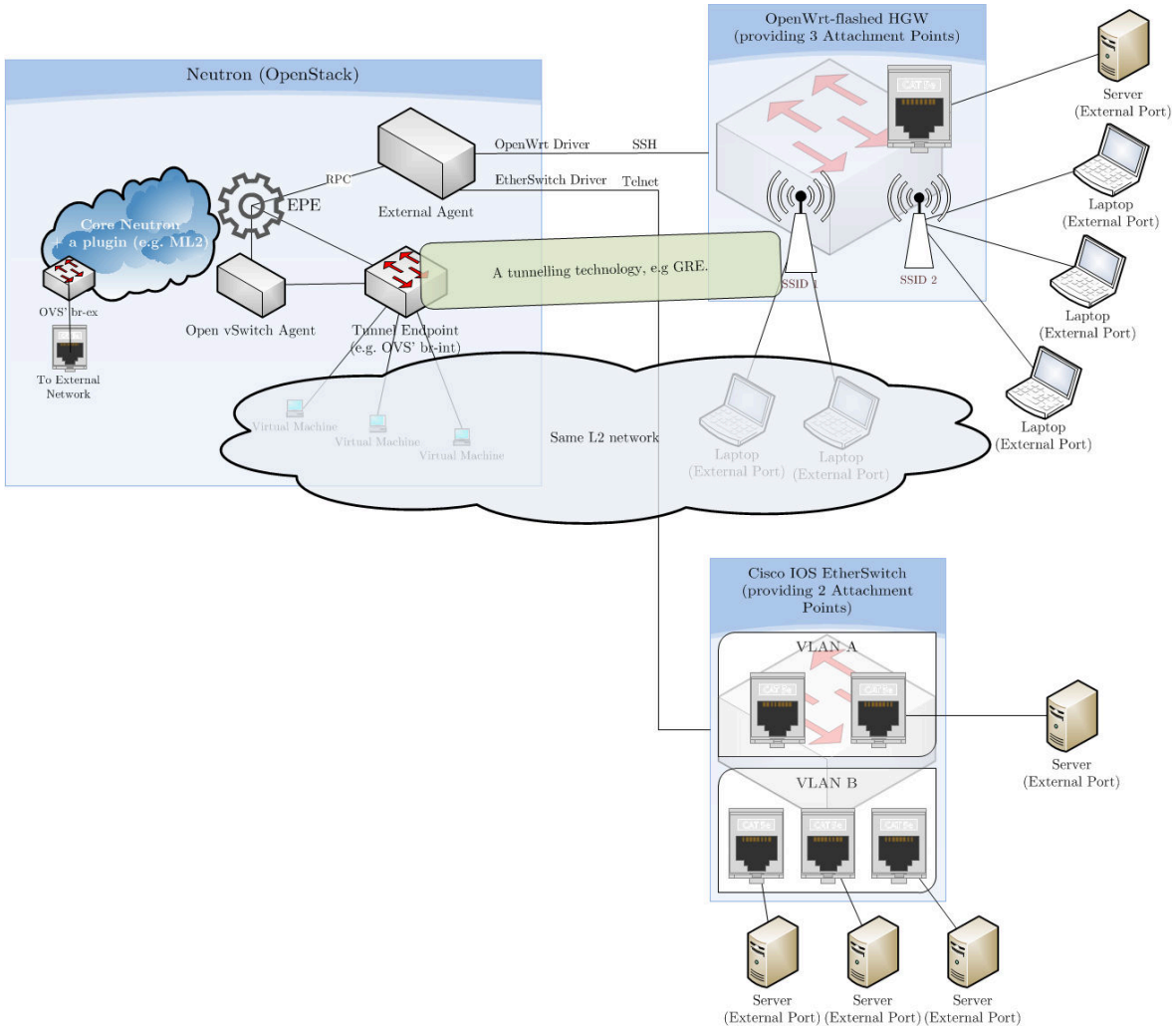


Figure 6.3: An example of a Neutron External Port Extension deployment.

Based on these new operations supported on top of the Neutron API extension, there are two major operations, disguised as use cases, that besides changing the data model require new functionality to be implemented on the existing architecture:

- Attaching a network to an Attachment Point;
- Detaching a network from an Attachment Point;

It should be reinforced, at this point, that this work is developed on top of the ML2 OVS mechanism driver. It can be extended to other mechanism drivers with minimum to medium effort, given the level of abstraction already put inside ML2 extension developed. A possible one to support is the OpenDaylight mechanism driver.

The operations presented are opposite, so they can be explained in an unified way. When attaching a network to an Attachment Point, the existing OVS agent executes a new method that sets up a special GRE tunnel at `br-tun` or `br-int` present at the Network Node, depending on the nodes' setup. When detaching a network, the opposite is carried out by destroying the tunnel created. How the OVS agent receives these requests has not been changed from other operations that the agent carries out, which is by RPC against new methods developed in the RPC server, enforced by the External Port Extension. The tunnel is put up by creating new OVS ports on `br-int` (which may be `br-tun` on

multi-node setups) of type GRE. To these ports, unique GRE Keys are assigned and an IP endpoint is provided, matching the public IP of the desired Attachment Point to be reached by that tunnel. Afterwards, OVS itself fulfills the rest of the process, by sending the GRE traffic (which comes from some Neutron network currently attached to that Attachment Point) to the Network Node's NIC that is able to reach the Attachment Point IP (taking into account the routes setup). As such, in the case of a multi-node OpenStack deployment, traffic originating in some Neutron network that reaches the Network Node, either because it is broadcast traffic or is headed to some External Port provided by an Attachment Point, leaves `br-tun` and is sent via a different interface than the one usually meant for tunneling traffic (the interface that connects to the Data Network). Usually, this kind of traffic will be sent via the interface that connects to the External Network, without passing through `br-ex` because virtual routing does not apply to this feature.

Besides setting up Neutron to fulfill network segment extension, by resorting to the OVS agent, the Attachment Point itself must be brought up, by resorting to what is called the External Agent. The External Agent is just like any other Neutron agent, it is manageable by the API and is instantiated at the Network Node as an individual process. Furthermore, it communicates with the RPC server at the Cloud Controller Node to obtain all necessary data, and is notified by it when other changes occur. This agent is responsible for dealing with ADs, which provide Attachment Points for extending network segments, complementing the tunnel setups made by the OVS agent at the Network Node side. So, when the operation of attaching a network is requested by a tenant or administrator, the RPC server will notify the External Agent, communicating the operation and any data relevant for carrying it out. When the agent receives the operation request it extracts all data, including the name of the driver to configure the AD where the Attachment Point is to be established. Then, it instantiates the correct driver and establishes communication via the AD mentioned by the Neutron notification, by connecting to its public IP address with whatever protocol is defined or supported by the driver (e.g. SSH, telnet or SNMP). Having a session been established with the desired AD, specific configurations are injected which match the complement of what the OVS previously did at the Network Node: setting up the second endpoint of the GRE tunnel, with the same GRE Key (corresponding to the Attachment Point). The operation of detaching an Attachment Point is analogous, although the configurations injected intend to destroy the tunnel and rollback the to its previous state, instead of setting up new options.

Figure 6.3 shows a deployment state where there are a total of five Attachment Points spread amongst two ADs. The first AD is an OpenWrt-flashed HGW providing three Attachment Points, two of them materialized in different wireless SSIDs and the third one materialized in a specific Ethernet port. A Cloud shape shows how the clients of the first Attachment Point: laptops connected to the wireless AP's SSID, are connected to the rest of the network elements living inside OpenStack as Nova instances (VMs), at the same broadcast domain. Figure 6.3 also demonstrates a tunnel established between the OVS' br-int and the Open-Wrt flashed HGW AD, fulfilling a network segment extension. Analogous Clouds and tunnels can be drawn for the remaining Attachment Points of Figure 6.3, including for the ones provided by the Cisco IOS EtherSwitch AD, but have been omitted to keep the figure legible.

The procedure explained for the External Agent is in line with what is developed and tested. OVS is chosen as the mechanism driver and GRE tunnels were used against the Attachment Points. However, the architecture is also built in a generic, flexible way, so using e.g. VXLANs is perfectly possible, assuming drivers support it. The same applies to the running version of OVS as well. Furthermore, when presenting the data model it becomes clearer how different technologies (GRE, VXLAN, etc., are inherently supported).

6.3.2 COMPONENTS

Figure 6.4 shows the newly developed code components (excluding the definition of the Neutron extension itself and any CLI code). Besides those, other related components are presented as well. Components with horizontal stripes as their background are the newly developed ones, while those with vertical stripes are slightly changed versions of Neutron's existing components. All others remain unchanged besides having new Python mix-ins applied. The stereotype "metadata" in this context means that the receiving component is based on multi-inheritance by means of Python mix-ins, where the other component is one of the mix-ins which applies to the former component.

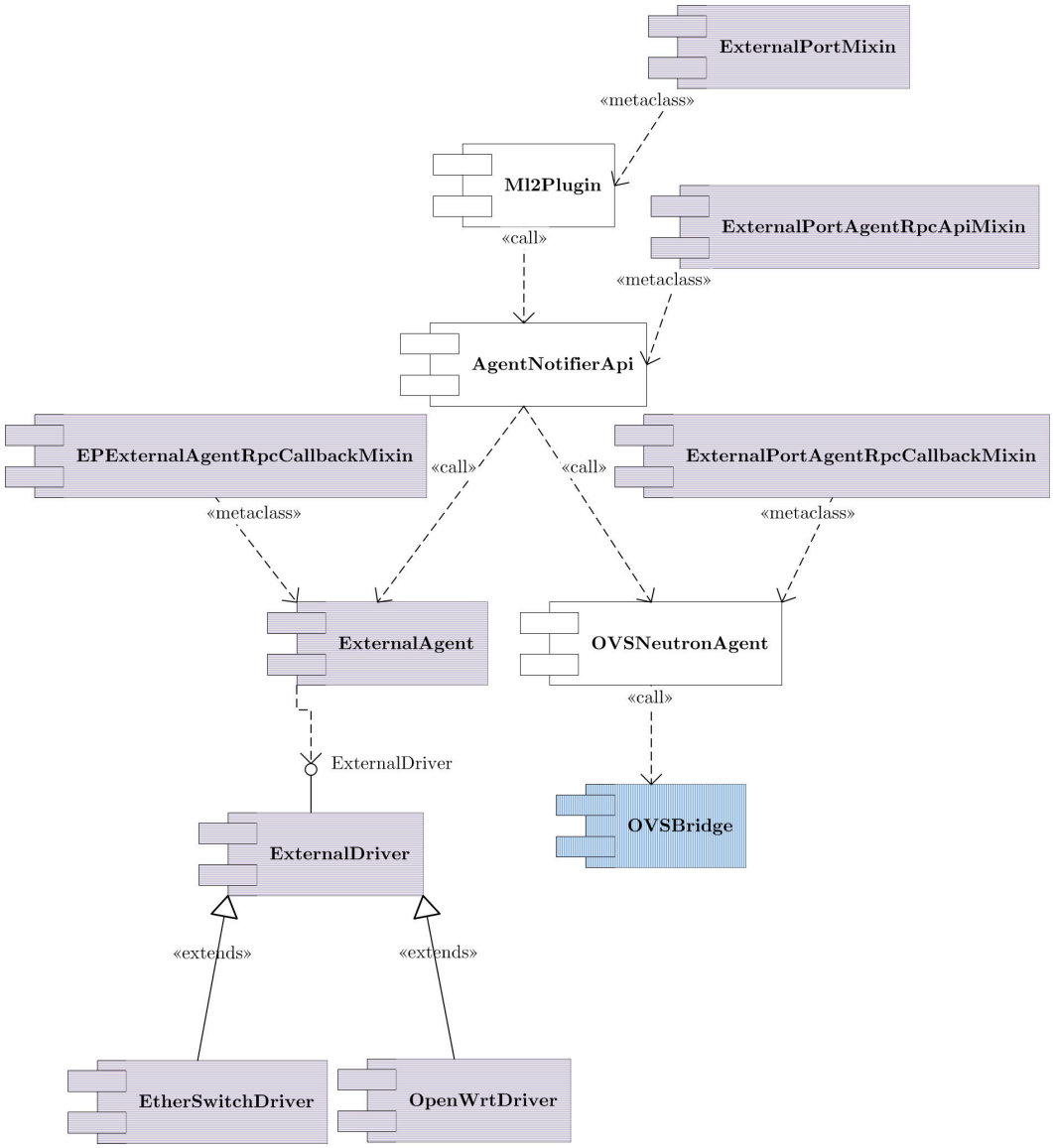


Figure 6.4: Component Diagram of code components and their relationships.

Each time there is a Neutron API request that has consequences in the EPE, for instance when attaching an Attachment Point, `Ml2Plugin` will be the first to receive that request given that it is the active core Neutron plug-in. Afterwards, it calls `AgentNotifierApi` (which has new methods from `ExternalPortAgentRpcApiMixin`), usually present at the same node, so it alerts any agents that may be expecting calls from the RPC server. Two kinds of agents, which may be at different, remote nodes,

receive the request: `ExternalAgent` and `OVSNeutronAgent`. The latter configure the OVS bridges accordingly so they leverage half of the NSEP. The former requests the correct `ExternalDriver` and calls the desired method for configuring the remote AD.

6.3.3 EXTERNAL DRIVERS

Support for heterogeneous network equipment is achieved by the use of (External) drivers (for ADs). These drivers are used by EPE to interface with the respective ADs and configure them according to what networks are attached to which Attachment Points. External Agent is the entity which instantiates and controls these drivers, serving mainly as a mediator between what is defined in Neutron's database and the implementation of these definitions by the ADs.

The `ExternalDriver` component, depicted in Figure 6.4, is the abstraction of these drivers. The methods `ExternalDriver`'s children must implement are the following:

- `driver_name(cls)`: A class method that must be implemented by each External Driver to explicitly and unequivocally state the driver's name. Each AD present in Neutron's database has the `driver` attribute, which is used to find the correct driver, by searching throughout this method in all available External Driver's children.
- `__init__(self, os_ip_addr, ap_ip_addr, identifier, technology, index, report_point)`: This method, although not technically a constructor, is the most approximate there is to it given that it prepares new objects instantiated from a class. All parameters received are mandatory. `os_ip_addr` is the reachable/public IP address of the OpenStack/Neutron deployment. `ap_ip_addr` is the reachable/public IP address of the AD which provides the Attachment Point. The `identifier` parameter is meant to receive the `identifier` attribute from the database's AD definition, which must respect driver-dependent syntax and semantics. `technology` is used to match with the Neutron's side NSEP technology (e.g. GRE). Finally, `index` is an integer used to clearly separate different Attachment Points to be provided by the same AD.
- `attach(self)`: The object method which triggers the attachment of a network to an Attachment Point, enabling the NSEP. Based on all parameters passed to `__init__` beforehand, this method is able to start a connection with the AD, reconfigure it, and close the connection.
- `detach(self)`: The object method inverse to `__attach__`. It triggers a detachment of a network from an Attachment Point, destroying the NSEP. Based on all parameters passed to `__init__` beforehand, this method is able to start a connection with the AD, reconfigure it, and close the connection.
- `monitor(self)`: This method is meant to be called by the External Agent for the Attachment Points that are also Reporting Points. It is responsible for dealing with whatever notifications the Reporting Point sends back to OpenStack, or poll it for new information. Actual functionality is defined by the Attachment Driver itself as well as its `identifier` field.

Two working drivers are developed in this work: OpenWrt and Cisco EtherSwitch. Just like the OpenStack-side implementation, only the GRE tunneling technology is developed, for both drivers.

OPENWRT

The OpenWrt driver (named `openwrt`) interfaces with typical switches or routers running the latest stable version of OpenWrt, currently 14.07 Barrier Breaker [6]. Compatibility with this driver must be checked for each device, because OpenWrt is an operating system that runs on a wide array of switches and routers. A NETGEAR WNDR3700v1 HGW was used for all debugging and testing regarding this driver. Connections made by the driver to these ADs can either be through `ssh` or `telnet`, depending on the identifier. The syntax of the `identifier` field for the OpenWrt driver is a simple key-value list of configurations: `<property1>=<value1>;<property1>=<value1>;...`. Table 6.2 presents what keys were developed and example values:

Key	Description	Example Value
iface	Access interface	ssh
usr	Login username	root
pwd	Login password	secretpass
ssid	SSID to instantiate	Staff
ssid_pass	Password for SSID	65846531
eth_ports	Ethernet ports to bridge	1,3,4

Table 6.2: OpenWrt Driver identifier syntax

Internally, the OpenWrt External Driver makes heavy use of the UCI system [58] to inject network configurations into the AD. These configurations primarily create new VLANs, reconfiguring the integrated switch and wireless radios to assign ports and SSIDs to these VLANs. Besides UCI configurations, a new Linux interface of type `gretap` (packets become intrinsically encapsulated in GRE) is always created and connected to a virtual LinuxBridge switch (not the integrated switch) meant for a specific VLAN. Furthermore, traffic coming from a specific VLAN will always arrive at the correct LinuxBridge switch, either via integrated switch's ports, radio interfaces or the `gretap` interface and, afterwards, exit through another of LinuxBridge port (per the forwarding table).

Matching with the AD requirements stated in 6.2.2, devices using the OpenWrt driver in fact have a reachable IP address from within Neutron. Remote administration is possible thanks to SSH or telnet for getting access to the devices' CLIs. The Python module used for SSH was paramiko [7] and for telnet was telnetlib [8]. GRE tunnels are possible to be established thanks to the `gre` kernel module available in OpenWrt. These devices allow reconfiguring the integrated switch for assigning VLANs and any combination of physical interfaces (Ethernet, wireless SSIDs, etc.). Finally, GRE tunnels can be associated to the internal VLANs created.

CISCO ETHERSWITCH IOS

The Cisco EtherSwitch IOS driver (named `etherswitch`) interfaces with the Cisco EtherSwitch line of switching modules compatible with some Cisco equipment running IOS. Connections established by this driver to reconfigure the switch are made through telnet only. A Cisco C3640 with an EtherSwitch module (NM-16ESW) was used for all debugging and testing regarding this driver. The syntax of the `identifier` field for the Cisco EtherSwitch IOS driver is a simple key-value list of configurations: `<property1>=<value1>;<property1>=<value1>;...`. Table 6.3 presents what keys were developed and example values:

[6] `https://openwrt.org`
[7] `http://www.paramiko.org`
[8] `https://docs.python.org/2/library/telnetlib.html`

Key	Description	Example Value
usr	Login username	root
pwd	Login password	secretpass
eth_ports	Ethernet ports to bridge	fe0/0,fe0/1,fe0/2"

Table 6.3: Cisco EtherSwitch IOS Driver identifier syntax

Internally, VLANs are created and then associated to a bridge group [59]. Besides that, GRE tunnels are created and associated to the bridge group as well.

Matching with the AD requirements stated in 6.2.2, devices using the Cisco EtherSwitch IOS driver in fact have a reachable IP address from within Neutron. Remote administration is possible thanks to telnet for getting access to the devices' IOS CLIs (again making use of the telnetlib Python module). GRE tunnels are possible to be established because built-in support in the Cisco EtherSwitch modules is available [60]. These devices allow reconfiguring the integrated switch for assigning VLANs and any combination of Ethernet ports by making use of bridge groups. Finally, GRE tunnels can be associated to the bridge groups.

6.3.4 DETECTION OF EXTERNAL PORTS

As presented in the Design chapter, specifically at section 5.3, detection of External Ports can be accomplished by one of two possible manners: by making use of Reporting Points (via the Attachment Driver), or internally by analyzing traffic inside Neutron. Although this process has not been developed to its fullest, an experimental work/Proof of Concept (PoC) was undertaken to automate the creation of External Ports and consequently core Neutron ports.

The experimental work consists in launching a new process from within the External Agent which makes use of the libpcap library [9] meant for sniffing/capturing network traffic, more specifically the Python module for making use of it, pylibpcap [10]. This process should be launched inside the Linux namespace where the DHCP server is running. It listens for UDP packets headed towards port 67 (the default DHCP serve port) and extract the source MAC address, then executing the necessary series of CLI commands to create and attach an External Port. Figure 6.5 shows the general architecture of the PoC based on the general component diagram presented in Figure 6.4.

However, a better way of achieving this detection would be to actually modify the DHCP server, by default dnsmasq, so it becomes reactive to specific DHCP packets, executing custom actions.

[9] `http://www.tcpdump.org`
[10] `http://pylibpcap.sourceforge.net/`

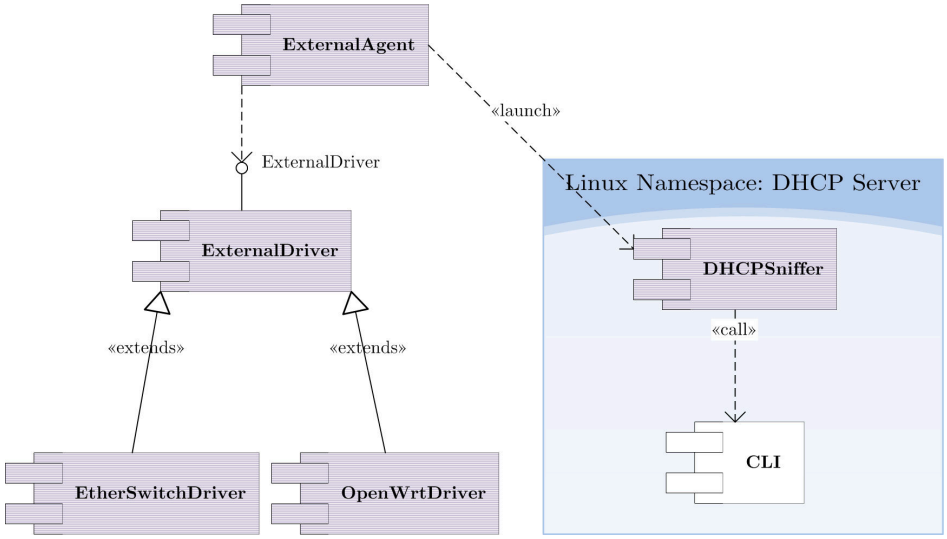


Figure 6.5: The added components for experimental port creation.

6.4 DATA MODEL

The data model for OpenStack resources has essentially been changed by providing three new entities, or classes. The first is called an Attachment Point, the second an External Port, and the third a AD. These entities are related to design concepts, as can be seen in Table 6.1.

6.4.1 DATA CLASSES

ATTACHMENT DEVICE

The first new data entity developed to support this work is the Attachment Device, which maps back to the design's NAP. Specifically, it is the data resource that describes a remote device able to leverage the NSEP. However, it does not specify anything about the internals of the segment extension like which network they are mapped to. Its two main uses are reachability and compatibility: how to reach the device that makes NSEP possible (by providing an Attachment Point) and how to communicate with that device (by making use of drivers). The data model for this entity, as developed for Neutron, is presented in Table 6.4.

Name	Type	Access	Default	Validation	Description
id	string	RO, all	generated	N/A	identity
name	string	RW, all	empty	string	name
description	string	RW, all	empty	string	description
tenant_id	string	RW, all	from auth	N/A	tenant
ip_address	string	RW, all		ip_address	IP address to reach Attachment Point
driver	string	RW, all		string	an available driver's name
admin_state_up	string	RW, all	True	convert_to _boolean	force up or down state
status	string	RO, all			general status
error	string	RO, all			error state

Table 6.4: Data model for Attachment Devices

Some of the attributes from Table 6.4 along with their descriptions are quite straightforward

and self-explicative: `id`, `name`, `description`, `tenant_id`, `admin_state_up`, `status` and `error`. These attributes are also standard throughout other OpenStack and Neutron resource types.

The attributes that effectively need a bit more of an introduction are: `ip_address` and `driver`.

- *ip_address* This field stores the IP address that is used to reach the Attachment Point. Given that it may live at the other side of the Internet, crossing multiple hops until reaching the device that really provides the Attachment Point, the responsible OpenStack Network Node and the device must be mutually reachable. Traffic must be mutually routable between them. As such, this IP address allows the first part, OpenStack's Network Node, to reach the second part (device providing the Attachment Point) whilst simultaneously telling it how to reach back the Network Node. The specific circumstances or scenarios where the IP address is used occur when either: configuring the device; retrieving network state from it; and forwarding traffic to/from it. In the first scenario, a driver-defined protocol will be used to carry on configuration actions towards the Attachment Point. In the second scenario, if the Attachment Point was configured as a Report Point as well, it may be able to notify OpenStack's Network Node without prior request (depending on the driver and chosen reporting protocols), in which case the device will be the one initiating a connection towards OpenStack's Network Node IP address. Finally, for the third scenario, some kind of tunneling technology will be in place, defined by the Attachment Point and driver-supported, whose endpoints will be the IP addresses of OpenStack's responsible Network Node and the Attachment Point's IP address.
- *driver* The `driver` specifies what Attachment Point driver should be loaded for itself. It is a string that must uniquely identify one of the installed drivers at the OpenStack deployment in regard. These drivers are pluggable and completely decoupled from the rest of the External Port Extension. Considering that, they can be developed for any device, by anyone. A company hosting an OpenStack deployment but not participating in its development can develop an in-house driver by itself, for their own equipment. Thanks to OpenStack's design and development decisions, including Python's choice of base programming language, and analogous decisions in this work, these drivers can be copied to OpenStack's installation path without rebuilding any of its parts or this extension's, and without even needing to restart current OpenStack's services. This results in reduced downtimes that would otherwise affect the company's availability and reliability. It is the driver that specifies what technologies are supported by the end device, what configurations can be applied to it (via the `identifier`), if network state reporting is supported (as well as how to interpret these state reports or schedule report requests, via polling e.g.). It will set up the device, reachable through its public IP address, and unset it up when it should no longer provide an Attachment Point. One thing that must be clear, though, is that it is not the driver that provides Neutron's endpoint for the established network traffic tunnel, that must be provided by the core plug-in enabled in Neutron (e.g. ML2), which are responsible for materializing the network defined via the Neutron Core Networking API.

ATTACHMENT POINT

The first new data entity developed to support this work is the Attachment Point, which maps back to the design's NAP. Think of an Attachment Point as the crucial part in the whole network that deals with merging (or bridging, in computer networks' terms) a pair of network segments. It is a

L2GW. Besides NAP, this entity also maps back to the design's NRP by configuring one of its fields. The data model for this entity, as developed for Neutron, is presented in Table 6.5.

Name	Type	Access	Default	Validation	Description
id	string	RO, all	generated	N/A	identity
name	string	RW, all	empty	string	name
description	string	RW, all	empty	string	description
tenant_id	string	RW, all	from auth	N/A	tenant
device_id	string	RW, all		N/A	an AD ID
identifier	string	RW, all		string	identifies how the Attachment Point extends the network
technology	string	RW, all		string	what technology both sides must use
network_id	string	RW, all		uuid_or_none	network to which this Attachment Point attaches
index	int	RO, all	generated	int	automatic index to differentiate Attachment Points living on the same physical device, e.g.
report_point	string	RW, all	True	convert_to _boolean	if it should report network state as well
admin_state_up	string	RW, all	True	convert_to _boolean	force up or down state
status	string	RO, all			general status
error	string	RO, all			error state

Table 6.5: Data model for Attachment Points

Some of the attributes from Table 6.5 along with their descriptions are quite straightforward and self-explicative: `id`, `name`, `description`, `tenant_id`, `admin_state_up`, `status` and `error`. These attributes are also standard throughout other OpenStack and Neutron resource types.

The attributes that effectively need a bit more of an introduction are: `identifier`, `technology`, `network_id`, `index` and `report_point`. The `device_id` attribute is used to select an existing AD, which already defines its reachable IP address and driver name.

- *identifier* The `identifier` attribute is a string used to pass configuration options to a driver and, consequently, to a device hosting an Attachment Point. This attribute must respect a syntax imposed by the Attachment Point's assigned driver. What configuration options, and how and when these will be used in the driver depend on the operations called as only the driver knows when each one should be used. So, the `identifier` provides a full set of options which goes against what the administrator/tenant intends to provide to/in these devices, in order to fulfill any desired objectives. Because identifiers' syntax is dependent on the driver, a prior study must be carried out. An example of what an `identifier` can configure at the Attachment Point is when remotely setting up wireless SSIDs which extends a Neutron's network segment via an wireless AP. The administrator can, thus, pass the name of the SSID, security options, or other settings via the `identifier` field. Like previously said, the identifier syntax depends on the driver, but the semantics depend as well because they are tightly related to what features the device can provided. In the example given, the device must have wireless interfaces and be able to reconfigure them via a management interface (to be leveraged by the driver).
- *technology* The `technology` field is the one that puts both sides of the network segment extension process in agreement in terms of the technology used to forward network traffic. That technology will usually be a tunneling technology, e.g. GRE, VXLAN, IPsec, although it is not strictly required. Computer networks' technologies evolve so do the mechanisms to bridge traffic over the Network Layer. It could be argued that this field should be provided via the `identifier` attribute. Although merging these attributes into a single one would make configuration options

completely, and clearly, defined in a single place, they have quite a bit of differences and have thus been split. The reasons for this is that while `identifier`-provided options are flexible, dependent on the driver and have no impact on OpenStack, the `technology` field is not flexible, not directly dependent on the driver and has actual meaning for OpenStack. There is a loose dependency on the `technology` and the driver, though, as it must support the desired technology or the network segment extension will not work. However, the attribute itself is agnostic to the driver, mainly because the latter does not enforce any syntax. The meaning of `technology` to OpenStack is that it defines what technology should be used by Neutron to pass traffic via the Attachment Point, so it would not be wise to keep a Neutron-relevant configuration inside a driver-only attribute like `identifier`.

- *index* This attribute is what differentiates between Attachment Points of the same device, in terms of the technology used to forward traffic. Because there can exist many Attachment Points with the same public IP address (same device), and the underlying technology to forward traffic may be the same, there must be a way to uniquely identify each Attachment Point. So, the `index` attribute was created. It is an integer number that automatically increments for each new Attachment Point. This number is then used to uniquely identify each different network segment extension towards the same device, e.g. associating different keys in different GRE tunnels.

- *report_point* An Attachment Point becomes a Reporting Point as well via this boolean attribute. Going back to the design chapter, there are NAPs and NRPs. Setting the `report_point` field to true yields that both an NAP and NRP are present and reachable in the same point. When this field is set to true, the driver carries on two aspects: it should provide network state reporting configurations on the remote device; it should, itself, initiate a process of network state monitoring. The first aspect is applied during the normal operation of remotely configuring the Attachment Point. The second aspect starts right after the first, posterior to finalizing the device's configuration. The driver will keep a special internal process running to receive and process network state reports and/or request them, in what constitutes the whole network state monitoring activity. Whether network state information will be acquired via polling or just waiting for notifications from the Reporting Point depends on the actual driver and eventually a configuration option passed via the `identifier` field.

EXTERNAL PORT

The second new data entity to support this work is the External Port, which maps back to design's NEP. This is the entity that translates to the external host which participates in Neutron's networks, whereas the Attachment Point is merely the resource used to bring these External Ports into participating hosts of Neutron's networks. The data model for this entity, as developed for Neutron, is presented in Table 6.6.

Most attributes present in Table 6.6 are quite straightforward. However, three of them are best if clearly defined now, hopefully removing any doubts or ambiguities regarding their existence and the choices that led to it.

The attributes that need a bit more of an introduction are: `mac_address`, `attachment_point_id`, and `port_id`.

Name	Type	Access	Default	Validation	Description
id	string	RO, all	generated	N/A	identity
name	string	RW, all	empty	string	name
description	string	RW, all	empty	string	description
tenant_id	string	RW, all	from auth	N/A	tenant
mac_address	string	RW, all		mac_address	the MAC address of the port
attachment_point_id	string	RW, all	uuid_or_none	string	Attachment Point that provides connectivity to this External Port
port_id	string	RW, all	uuid_or_none	string	Neutron core port resource associated with this External Port
admin_state_up	string	RW, all	True	convert_to_boolean	force up or down state
status	string	RO, all			general status
error	string	RO, all			error state

Table 6.6: Data model for External Ports

- *mac_address* A MAC address field is already present at the core Neutron port resource. A decision was made to have a separate data type for External Ports, instead of just extending the already existing core one. The attribute itself, it identifies the MAC address of the host that will be directly connected to the Attachment Point's device, which is also the MAC address that will be used by that host's abstraction when participating in Neutron's networks. The host can be a traditional, physical computer, connected via Ethernet to a traditional, physical network switch. The reason why the address must be added beforehand to an External Port is that, otherwise, this host which is connecting to the a Neutron network would not be "official" and, as such, essentially invisible. In other words, a Neutron port must be created for any host participating in a network, otherwise the DHCP service is not able to serve it with an IP address, amongst other undesirable consequences. A core Neutron port requires specifying a MAC address, so the one set up in the corresponding External Port is used. If an External Port has not yet been created when a host tries to communicate with other hosts (e.g. Nova instances), and even if it has manually set a legal IP address, it may be blocked from communicating with these other hosts, or the rest of the network for that matter, depending on what security measures are enforced in OpenStack. Typically, it will work (currently), but that is not a guarantee. Besides, this is not the procedure that would be carried out on a production scenario. The ARP protocol may work properly even if no External Port has been created, but that is limited connection, besides the fact that the host may be blocked from communicating altogether, as referred before. So, in summary, the MAC address mainly allows external hosts to acquire a valid, official IP address and to be made visible in the network, clearing any security measures that may be in place.

- *attachment_point_id* External Ports can be created independently. Posteriorly, they can be associated to a port in a network, where the port will mimic External Port's MAC address so it indeed becomes part of the network. However, the External Port must also be associated to an Attachment Point, itself associated (or, in this case, "attached") to a network. The network must be the one providing the port which is associated to the External Port. The Attachment Point where each External Port is associated to is given by this field. A database consistency mechanism is enforced so that each External Port associated to a core Neutron port must also be associated to an Attachment Point of the same network. The reason why this consistency mechanism is enforced, instead of just inferring what network each External Port is associated to, through the port common, lies in the fact that knowing what is that network is not the only objective. The network administrator or tenant must be able to know what is the Attachment

Point providing that External Port, at current time. Due to the fact that multiple Attachment Points can be associated to the same network, it is not possible to infer the Attachment Point currently providing an External Port. Rather, a dedicated relationship must be made and, as such, this attribute defined.

- *port_id* As previously noted, a core Neutron port gets associated to an External Port in some phase of its life cycle. External Ports do not need to be associated with ports. However, if the host External Port symbolizes needs to be used in some network, which is attached via an Attachment Point, that External Port must also be attached to the Attachment Point (by setting its `attachment_point_id`). That process will, in turn, create an associated core Neutron port (if one does not already exist) and associate it with this External Port.

6.4.2 CLASS DIAGRAM

The class diagram present in Figure 6.6 illustrates how the data model changes relate to existing resource types:

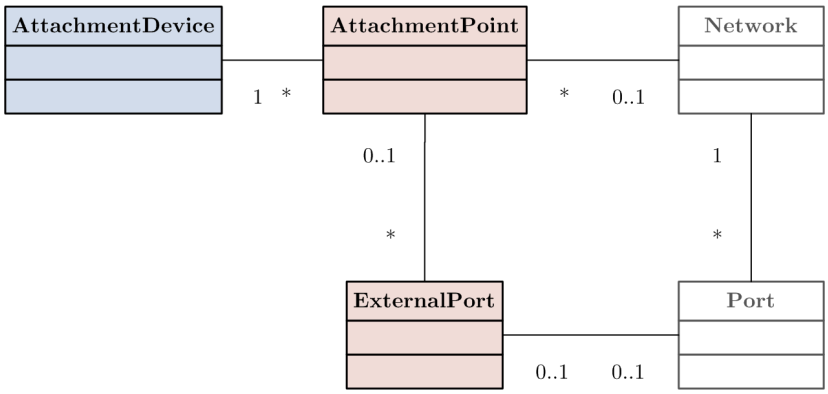


Figure 6.6: Class Diagram of solution and directly-related classes.

Core Neutron networks are associated one to many with core Neutron ports, which is an accepted fact in Neutron. It also makes sense when comparing this with traditional physical networks.

An Attachment Point has a relationship of many to one relatively to a Neutron core network because each one of the former need to attach to a specific network segment. Otherwise, the network segment provided by the Attachment Point would be forcefully merged with distinct broadcast domains, creating a conflict and making the network unreliable. Each network. however, can be attached to different Attachment Points at the same time. This allows an existing virtual network to be extended with multiple new segments, i.e. a concatenation of L2 network segments forming a single, larger, broadcast domain.

Each Attachment Point is associated to multiple External Ports because the latter are reachable (and reach others) through the former, and they have no conceptual limit in existence (a network is meant to have multiple ports, otherwise only the port needed to exist). These External Ports can only be associated to a single Attachment Point because, differently, the way to reach the rest of the network (or the opposite) would be ambiguous. Nevertheless, nothing prevents these External Ports to

change Attachment Points during their lifetime, provided that they stay associated to at most a single Attachment Point.

Finally, External Ports have a relationship of one to one with Ports. This means that an External Port effectively matches a core Neutron port, but they support different objectives.

Technologically speaking, an External Port could be an extension of the existing Port or, in Object-Oriented Programming (OOP) terms, it could inherit from Port, becoming a subtype of it. In the OpenStack database, the existing Port would be extended with new attributes to approximate the External Port. However, this implementation tried to keep the two as separate as possible to improve manageability by the administrator or tenant, preventing configuration errors or other unwanted behaviors. Understandably, this decision would need to be backed by clear reasons from key people (that benefit from this work), so would the opposite decision. The main objective when creating an External Port is to reserve some desired host to some network, by specifying its MAC address and having it assigned to an Attachment Point. If the host is accepted and the Attachment Point attached to a network, then a core Neutron port can be created and activated, which will acquire the External Port's MAC address.

Another thing that must be noted when interpreting the diagram is the relationship loop around the Attachment Points, External Ports and the existing resource types. It may seem that there is a redundancy of associations, though in reality the multiplicities are not equivalent. Besides improved flexibility (by managing resources and associations in a tiny scoped way) and clearer administrative control (although this would require concrete justifications beyond speculation), the main advantage of having relationships arranged in such a manner is to allow multiple Attachment Points to extend a network, which would otherwise be impossible if the relationship between these was inferred by relationships of Attachment Point, External Port and port.

6.5 INTERFACES

The interfaces considered for this work are both a programming interface and a user interface. The first presents itself as an extension to the Neutron core API, while the UI is presented in the form of a CLI, extended from the Neutron CLI project, `python-neutronclient`.

6.5.1 PROGRAMMATIC INTERFACE

The API for the data model developed comes as a direct mapping of the latter into either JavaScript Object Notation (JSON) or Extensible Markup Language (XML), per the OpenStack Networking API v2.0 [11]. As such, there is no need to repeat content.

The API is usually accessed for UI purposes either via a CLI (like the `python-neutronclient` project), Horizon (the web interface) or for orchestration purposes (via the OpenStack Heat) project. For the sake of this work, `python-neutronclient` was extended to make use of the developed Neutron API extension, whereas Horizon and Heat projects were left unchanged.

[11] `http://docs.openstack.org/api/openstack-network/2.0`

6.5.2 User Interface

The Neutron API is usually accessed either via Horizon (OpenStack's web-based dashboard project), or through a CLI.

In this work, the web dashboard was not extended as it is essentially out of scope. However, full support is developed for the CLI project for Neutron, `python-neutronclient`. Tables 6.8 and 6.9 show all the new CLI operations and their arguments, that can be request to the API. It also provides examples for each command. Dependencies between resources are enforced by the database itself and, if an operation is not possible due to those reasons, it is unsuccessful and the user is alerted with a relevant message.

Command	Arguments	Description	Example
`attachment-device-create`	`IP_ADDRESS DRIVER [PARAMS]`	Create a new AD.	`neutron \ attachment-device-create \ 10.0.0.100 openwrt \ --name openwrt_hgw`
`attachment-device-delete`	`DEVICE`	Delete a given AD.	`neutron \ attachment-device-delete \ openwrt_hgw`
`attachment-device-update`	`DEVICE [PARAMS]`	Update a given AD.	`neutron \ attachment-device-update \ openwrt_hgw \ --name home_ap \`
`attachment-device-show`	`DEVICE`	Show information of a given AD.	`neutron \ attachment-device-show \ openwrt_hgw`
`attachment-device-list`		List ADs that belong to a given tenant.	`neutron \ attachment-device-list`

Table 6.7: CLI Commands for ADs

Command	Arguments	Description	Example
`attachment-point-attach`	`ATTACHMENT_POINT NETWORK`	Attach an Attachment Point to a given network.	`neutron \ attachment-point-attach \ openwrt_hgw_ssid1 guest_net`
`attachment-point-detach`	`ATTACHMENT_POINT`	Detach an Attachment Point from a network.	`neutron \ attachment-point-detach \ openwrt_hgw_ssid1`
`attachment-point-create`	`DEVICE IDENTIFIER TECHNOLOGY [PARAMS]`	Create a new Attachment Point.	`neutron \ attachment-point-create \ openwrt_hgw \ "proto=ssh;ssid=guests" \ gre --name openwrt_hgw_ssid1`
`attachment-point-delete`	`ATTACHMENT_POINT`	Delete a given Attachment Point.	`neutron \ attachment-point-delete \ openwrt_hgw_ssid1`
`attachment-point-update`	`ATTACHMENT_POINT [PARAMS]`	Update a given Attachment Point.	`neutron \ attachment-point-update \ openwrt_hgw_ssid1 \ --name guests_port \ --identifier "proto=ssh;eth=1"`
`attachment-point-show`	`ATTACHMENT_POINT`	Show information of a given Attachment Point.	`neutron \ attachment-point-show \ openwrt_hgw_ssid1`
`attachment-point-list`		List Attachment Points that belong to a given tenant.	`neutron \ attachment-point-list`

Table 6.8: CLI Commands for Attachment Points

Command	Arguments	Description	Example
external-port-attach	EXTERNAL_PORT ATTACHMENT_POINT	Attach an External Port to an Attachment Point, creating a port during the process if the AP is attached to a network.	neutron \ external-port-attach \ computer1 \ openwrt_hgw_ssid1
external-port-detach	EXTERNAL_PORT	Detach an External Port from an Attachment Point, potentially deleting a port.	neutron \ external-port-detach \ openwrt_hgw_ssid1
external-port-create	MAC_ADDRESS [PARAMS]	Request the creation of an External Port.	neutron \ external-port-create \ 11:22:33:44:55:66 \ --name computer1
external-port-delete	EXTERNAL_PORT	Delete a given External Port.	neutron \ external-port-delete \ openwrt_hgw_ssid1
external-port-update	EXTERNAL_PORT [PARAMS]	Update a given External Port.	neutron \ external-port-update \ openwrt_hgw_ssid1 \ --name main_server
external-port-show	EXTERNAL_PORT	Show information of a given External Port.	neutron \ external-port-show \ openwrt_hgw_ssid1
external-port-list		List External Ports that belong to a given Attachment Point.	neutron \ external-port-list

Table 6.9: CLI Commands for External Ports

CHAPTER 7

Evaluation and analysis

Tests in different levels are required to evaluate the feasibility of the work and other factors that must be known beforehand when deploying a scenario that makes use of it.

The tests carried out measure different characteristics of the implementation and its deployment on a simplified real world scenario. These are not tests to ascertain whether code is correct. Rather, they measure the solution's potential value in a technical perspective, when deployed.

All hosts connected to the Neutron network as External Ports have access to the External Network like any other host, such as a Nova instance. Consequently, Neutron can act as an Internet provider to External Ports.

7.1 SCENARIO

The evaluation scenario used for all tests aims at replicating a real world scenario, where two independent infrastructures are apart but mutually reachable, usually via routing (e.g. over the Internet). Figure 7.1 presents that scenario.

However, for testing purposes, the scenario presented in Figure 7.1 is simplified to focus on measurements which are only, as much as possible, related to the systems which make up the NSEP. As such, there is no routing of packets between the computer running OpenStack and the Device that provides Attachment Points. Having hops between these endpoints would increase the number of factors that contribute to the test results, even though these are not related to the work itself. Furthermore, routing between where OpenStack is deployed and where the Attachment Point is located is usually part of another administrative domain which may or may not have a great impact on the end results. For these reasons, the scenario where routing between the two endpoints occur is excluded from the Test Scenario. So, Figure 7.1 morphs into Figure 7.2, the final Test Scenario where the OpenStack host is directly connected to the Device which will provide Attachment Points.

Finally, it must be noted that, for each physical and VMs that form the Test Scenario, a specific amount of main memory, Random-Access Memory (RAM), is assigned so as to prevent memory swapping during all tests. In fact, no memory swapping to disk was performed by any of the operating systems involved.

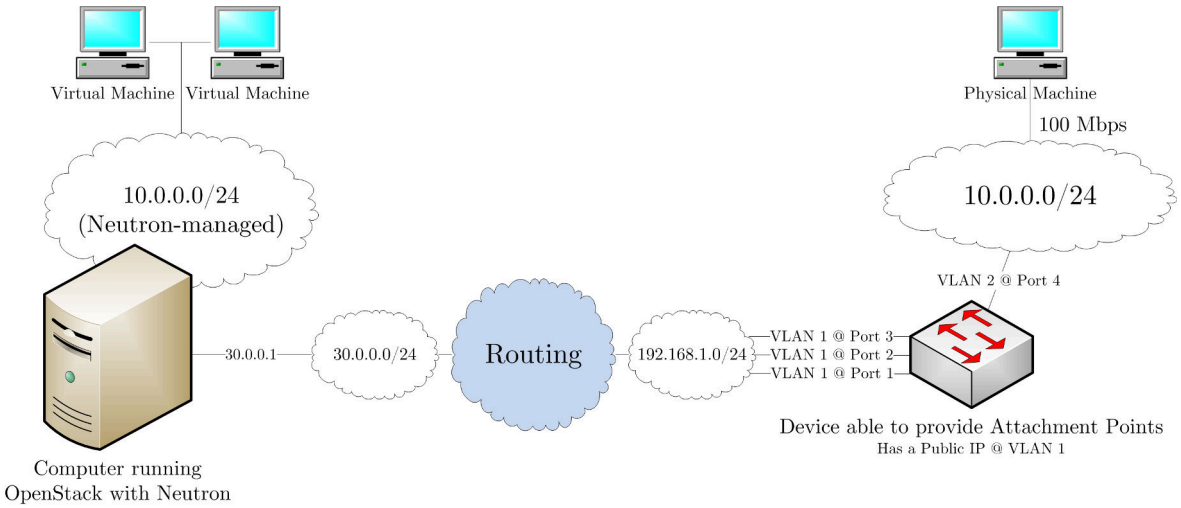


Figure 7.1: Generalization of the Test Scenario

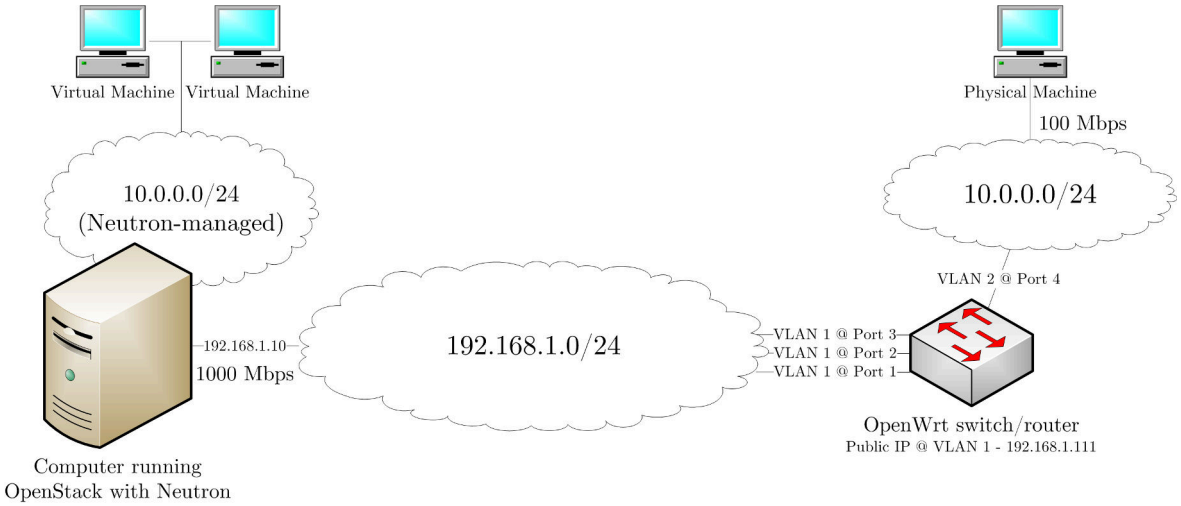


Figure 7.2: The actual Test Scenario

7.1.1 MAIN ELEMENTS

Figure 7.2 shows all main elements of the Test Scenario. At the left-hand side, there is a computer running OpenStack. This machine is actually a VM inside another, physical computer, which is not depicted in Figure 7.2. A Neutron network is already provisioned inside OpenStack, with two Nova instances running. At the right-hand side, an OpenWrt switch/router is depicted. The specific networking device used initially has a default configuration which consists in having all Ethernet ports (a total of 4) connected to a default VLAN. In this case, the device is initially connected to the `192.168.1.0/24` network, as well as the computer running OpenStack and the physical machine that can be seen at the top-right corner of Figure 7.2. The OpenWrt switch/router, when attached to a Neutron network, is set to change its fourth port to a different VLAN, effectively changing the broadcast domain of the physical machine (to the Neutron-managed `10.0.0.0/24` network) and fully justifying Figure 7.2. Specific details for each mentioned element follow along.

Computer running OpenStack with Neutron:

The computer running OpenStack has an Intel Core i5-2450M Central Processing Unit (CPU) clocked at 2.5 GHz (up to 3.1 GHz via Turbo Boost). Main memory totals 8 GB at 1333 MHz (dual-channel). The operating system in use is Arch Linux running on top of the Linux kernel 3.17.1 (64-bit). Finally, it uses VirtualBox 4.3.18 to run an Ubuntu VM, on top of which OpenStack is installed.

OpenStack itself is run inside a VM hosted by VirtualBox, deployed using DevStack [1], a distribution of OpenStack meant to ease development. This VM is set to acquire network connectivity by bridging against the host's NIC, effectively making it connect to the same network as the host computer. All code developed in this work has been applied to DevStack, Neutron and the Neutron Client projects on top of the `stable/icehouse` git branch as of mid-October 2014, which is the basis for all tests.

The VM which runs DevStack is set up as an Ubuntu 14.04.1 LTS on top of Ubuntu's linux-generic kernel 3.13.0-37 (64-bit). One CPU core is assigned to this VM and a total of 5 GB of main memory are allocated to it. Both Intel VT-x and Intel EPT hardware virtualization technologies [2], compatible with the host's CPU, are enabled under VirtualBox.

VM instances in OpenStack's Nova:

This VM instances to be run in Nova are Ubuntu 14.04.1 LTS images taken from Ubuntu Cloud Images [3], daily build 20141016. Depending on the test, either one or two of these machines were provisioned under Nova. Each VM has a total of 512 MB of main memory allocated to it and the kernel is Ubuntu's linux-generic kernel 3.13.0-37 (32-bit). The vNIC driver in use by the instance is the `virtio_net` [4] for improved network performance in virtualized guest/instances. Because these instances are already provisioned inside a virtualized environment, the default hypervisor (KVM[5]) does not make use of hardware acceleration, instead defaulting to Tiny Code Generator (TCG)[6].

OpenWrt switch/router:

This entity is the Device which is able to provide Attachment Points. It consists in a NETGEAR WNDR3700v1 having a main memory amount of 64 MB. The device has been flashed with the latest stable version of OpenWrt, Barrier Breaker 14.07, which uses the Linux kernel 3.10.49.

OpenStack interfaces with this device via the `openwrt` External Driver that has been developed, to test and debug this implementation (alongside the `etherswitch` External Driver).

Remote computer (External Port):

The remote computer is a traditional Desktop computer with the latest Ubuntu 14.04.1 LTS installed and all packages updated to the latest version as of 20th October, 2014. The CPU is an Intel Celeron D with a clock speed of 2.66 GHz. Main memory totals 960 MB at 400 MHz. The operating system in use is Ubuntu 14.04.1 LTS on top of Ubuntu's linux-generic 3.13.0-37 (32-bit), in an attempt to be as similar as possible with the VM instance in Nova.

[1] `http://devstack.org/`
[2] `http://ark.intel.com/Products/VirtualizationTechnology`
[3] `http://cloud-images.ubuntu.com/trusty/current`
[4] `http://www.linux-kvm.org/page/Virtio`
[5] `http://www.linux-kvm.org`
[6] `http://wiki.qemu.org/Documentation/TCG`

7.2 TESTS

7.2.1 SETUP AND TEARDOWN

The first aspect to test in the proposed solution is how long administrators/tenants need to wait before their orders for attaching or detaching Attachment Points become operational. That said, two measures have been created: Setup time, or time to attach, and Teardown time, or time to detach.

SETUP TIME

Setup time, or time to attach an Attachment Point and have bidirectional connectivity working measures the amount of time until an Attachment Point becomes operational. Time starts counting as soon as an administrator/tenant triggers an API request to attach an Attachment Point to a network. Only when network connectivity is acquired by a host at the network segment provided by the Attachment Point's owning Device does the time counter stop.

In order to faithfully measure these times, with a guaranteed upper bound of precision, a timer is started in a testing computer directly connected to the Device at the exact moment the API request to attach a network is sent to Neutron. This timer has been programmed to keep sending network pings to Neutron's virtual router IP address in the network to be attached. Pings are sent every 100 milliseconds and, as soon as a reply is obtained, the timer is stopped and total time obtained. Thus, the upper bound in precision when measuring the Setup times is 100 milliseconds.

TEARDOWN TIME

Teardown time, or time to detach an Attachment Point, is essentially the opposite of the Setup time. It measures the amount of time since the first API request made by an administrator/tenant to Neutron until network connectivity in the testing computer (inside the Neutron network as an External Port) is lost. A total of two API requests are sent: a) detaching the Attachment Point and b) removing the Attachment Point from the database. Time starts counting as soon as the API request to detach the Attachment Point is made.

Also like the previous case, this test is undertaken and measured recurring to periodic pings, with a 100 milliseconds period. As soon as an Internet Control Message Protocol (ICMP) Echo Reply (ping response) fails to be received in a maximum of 100 milliseconds, the timer stops. To circumvent eventual packet losses (although extremely rare in the isolated testbed used), the timer keeps running after a ping response fails to be received in case posterior responses are received, replacing the previous time in this case. All other unanswered questions regarding this specific test are equivalent to the Setup time test.

7.2.2 LATENCY

LOCAL LATENCY

Local Latency measures the latency between two VMs hosted by Nova in a standard out-of-the-box DevStack deployment. The relevancy of this test lies in the overall comparison of latency between

different VMs hosted in a single node, and latency between one of these VMs and an External Port physically distant from the Nova node.

The way latency is measured is by executing a series of pings (ICMP Echo Request/Response) between one of the VMs and the other. Ping properties are set up as the default ones used by the iputils' `ping` utility [7], version `20121221`, available in the Ubuntu VMs used. These properties consist in a 1 second delay between each ping, with each ICMP packet having a total of 48 bytes of ICMP data. This translates to a total of 102 bytes, including ICMP, IP and Ethernet headers, plus the Frame Check Sequence (FCS). It must be noted, though, that for packets traversing the tunnel that leads to/from Attachment Points an additional 42 bytes of overhead are inserted (totalizing 144 bytes).

An extra test carried out in the context of Local Latency is measuring how long does it take for the host running OpenStack to reach one of its Nova VMs. It also consists in exchanging a series of pings, as for the previous case.

REMOTE LATENCY

Remote Latency measures the latency between a VM host by Nova and a physical machine provided as an External Port. Carrying out this test enables comparisons with other kinds of deployments which do not provision External Ports. Namely, it is compared to the Local Latency test where two VMs in the same Nova node have their latency measured. The procedure for testing the latency is similar to the Local Latency test, save for the fact that the machines are in physically disparate locations.

7.2.3 TRAFFIC THROUGHPUT

Traffic Throughput tests aim to measure the overall throughput between different combinations of the kinds of nodes involved: Nova instances and External Ports.

Unless otherwise stated:

- UDP buffer size is 160 KiB
- UDP datagram size is 1470 KiB
- TCP client's window size is 48.3 KiB
- TCP server's window size is 85.3 KiB

LOCAL TRAFFIC THROUGHPUT

Local Traffic Throughput measures the average throughput between two VMs hosted by Nova. This test is analogous to the Local Latency test, except that it is meant to measure throughput instead of latency. Besides, it is decoupled in two different sub-tests: one for TCP and another for UDP.

For the testing procedure, `iperf` [8] is used. One of the instances is set listening for incoming connections with the following command (for the TCP test):

[7] `http://www.skbuff.net/iputils/`
[8] `https://iperf.fr/`

`iperf -s`, where `-s` specifies that iperf should act as a server.

In the case of UDP, the following command is used:

`iperf -u -s`, where `-u` specifies that the server is listening for UDP datagrams.

At the other VM, `iperf` is set as a client to send traffic to the first instance with the following command (for TCP):

`iperf -c 10.0.0.5`, where `-c` specifies that iperf should act as a client, and the IP address given is the server's one.

In the case of UDP, the following command is used:

`iperf -u -c 10.0.0.5 -b 1000M`, where `-b` specifies how much UDP bandwidth should try to be used when sending (in bits per second (bps)).

The reason for specifying a UDP bandwidth of 1000 Megabits per second (Mbps) is to make use of as much bandwidth available as possible. This value is not even greater because, as tests results later demonstrate, VMs cannot make use of all the bandwidth specified on the `iperf` client. Furthermore, the link with the highest speed in the Test Scenario, when using External Ports, has a speed of 1000 Mbps (Gigabit Ethernet cable).

The test are carried out without swapping VMs because they are twin and all network connectivity is symmetrical, so there are no influences in this regard.

REMOTE TRAFFIC THROUGHPUT

Remote Traffic Throughput measures the average throughput between a VM hosted by Nova and a physical machine provided as an External Port. The test is similar to Local Traffic Throughput, save for being performed between hosts in physically disparate locations. The testing procedure is also exactly the same. However, because network configuration is not symmetrical and the hosts are not twins anymore, this test is actually undertaken two times: one having `iperf` listening in the Nova VM and the other having `iperf` listening in the computer hosted as an External Port, each one for TCP and UDP. This way, throughput can be analyzed both in an upstream-manner and a downstream-manner.

The `iperf` commands for each of the upstream and downstream tests are analogous with the ones presented in Local Traffic Throughput, except for UDP when VMs act as clients, thus requiring the following command instead:

`iperf -u -c 10.0.0.5 -b 1000M -l 1430`, where `-l 1430` specifies a custom datagram size.

The reason for adding a non-default datagram size is that OVS would otherwise drop the datagrams instead of fragmenting them to fit the GRE tunnel towards the Attachment Point.

7.2.4 TRAFFIC OVERHEAD

Traffic Overhead is an implicit test. Although it is important to (explicitly) measure traffic overhead in this implementation, the only driver tested relies on the GRE protocol to achieve the NSEP. As such, traffic overhead (for the total packet size) is always fixed.

7.3 RESULTS

This section presents the results of the tests previously defined. Moreover, these results are evaluated and subject to discussion by the author, including additional information for subjective or unmeasurable results, which nevertheless are important to take into consideration when deploying this work.

7.3.1 SETUP AND TEARDOWN

Both tests of Setup and Teardown times have been carried out with a total of 10 iterations each.

Figure 7.3 shows the results of both tests, with the *xx* axis stating the iteration number and the *yy* axis stating the total time until network connectivity was attained. It must be remembered that the measuring precision has an upper bound of 100 milliseconds. Besides, setup and teardown iterations have not necessarily been made in alternating order. In other words, Teardown iteration 1 is not necessarily the tearing down of Setup iteration 1, so any correlation between these can only be evaluated as coincidental.

Tables 7.1 and 7.2 present average and standard deviance variables of the Setup and Teardown times' tests, respectively.

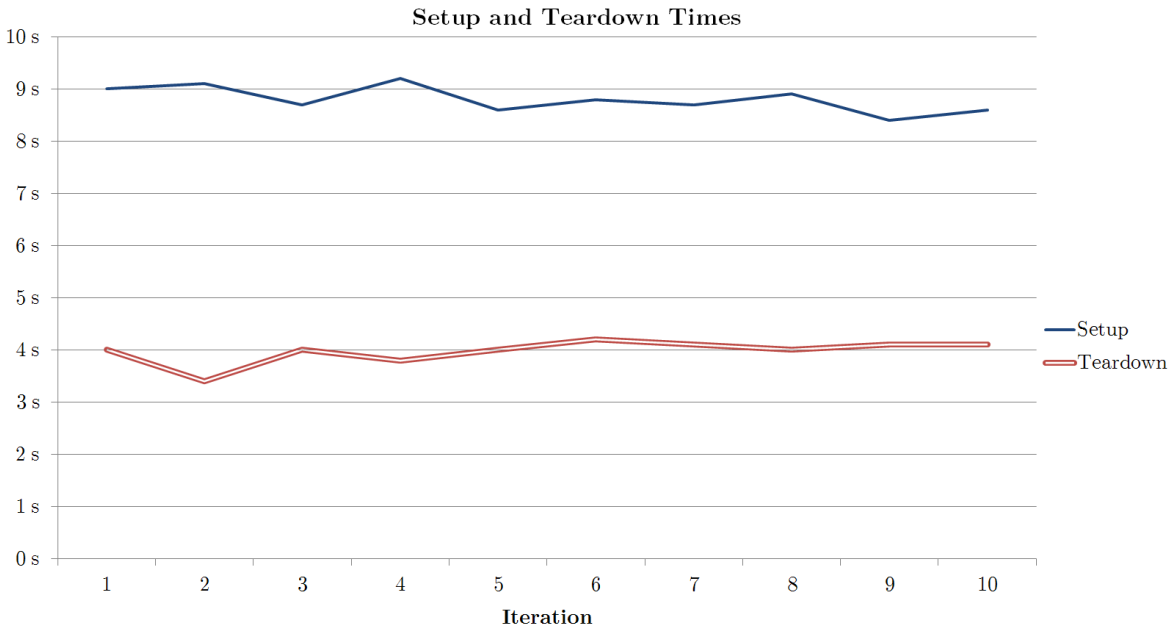


Figure 7.3: Setup and Teardown times per iteration.

Average	8.80 s
Standard Dev.	0.25 s

Table 7.1: Setup time statistics.

Average	3.97 s
Standard Dev.	0.23 s

Table 7.2: Teardown time statistics.

DISCUSSION

Setup times have been observed to be around the 9 seconds mark, which is spread between establishing an SSH session with the OpenWrt Device, executing specific commands and then committing all changes to the Device. In a production scenario where CPU usage may reach higher levels and network may get congested, the setup time will certainly increase. However, the operations carried out and the network traffic exchanged with the Device are low, so these measurements are not likely to increase greatly. Waiting 9 seconds, or even 10 times that, to insert a new, external network segment to a virtual network managed by OpenStack is a significant achievement when looking at the workarounds that usually need to be made. For instance, and considering that this work is best suited for existing network equipment, usually without support for SDN protocols such as SDN, rearranging networks so that a special host becomes available in a specific network may take anywhere from one day to weeks.

About the Teardown time, it is around the 4 seconds mark, less then half of setup's time. What has been said for the Setup time also applies in this case. Much like Setup time, having an "instant" Teardown time would be better, although in most real world scenarios an "instant" network redeployment speed is not as important as being plainly faster than traditional solutions, save for critical networks with critical components. A specific downside of having a non-instant Teardown time is that, if a network administrator or monitoring software finds that a tenant is exploiting the infrastructure for illegal purposes by making use of this extension, it cannot immediately block it just by using the public API methods available. Still, it is better than typical solutions, especially considering the heterogeneity support of the work (which influences, in a negative or positive way, Setup and Teardown times).

7.3.2 LATENCY

Both tests of Local and Remote Latency (including from OpenStack itself to one of its Nova VMs) have been carried out by sending a total of 100 pings between hosts. In order to prevent, or at least reduce, any initial "warm-up" influence over the `ping` delays (for instance due to the ARP protocol, the first 20 `ping` times were discarded (in each of both tests).

Local latency tests are hereafter named "VM - VM" based on the fact that communication takes place between Nova VMs, bidirectionally. Similarly, Remote latency tests are named "VM - PM".

The *xx* axis of Figure 7.4 shows the `ping` number, while the *yy* axis shows the amount of time that the respective ICMP Echo Reply took to arrive at the first host.

Tables 7.3 and 7.4 present average and standard deviance variables of the Local and Remote Latencies' tests, respectively.

Average	1.10 ms
Standard Dev.	0.34 ms

Table 7.3: Local latencies' statistics.

Average	1.42 ms
Standard Dev.	0.24 ms

Table 7.4: Remote latencies' statistics.

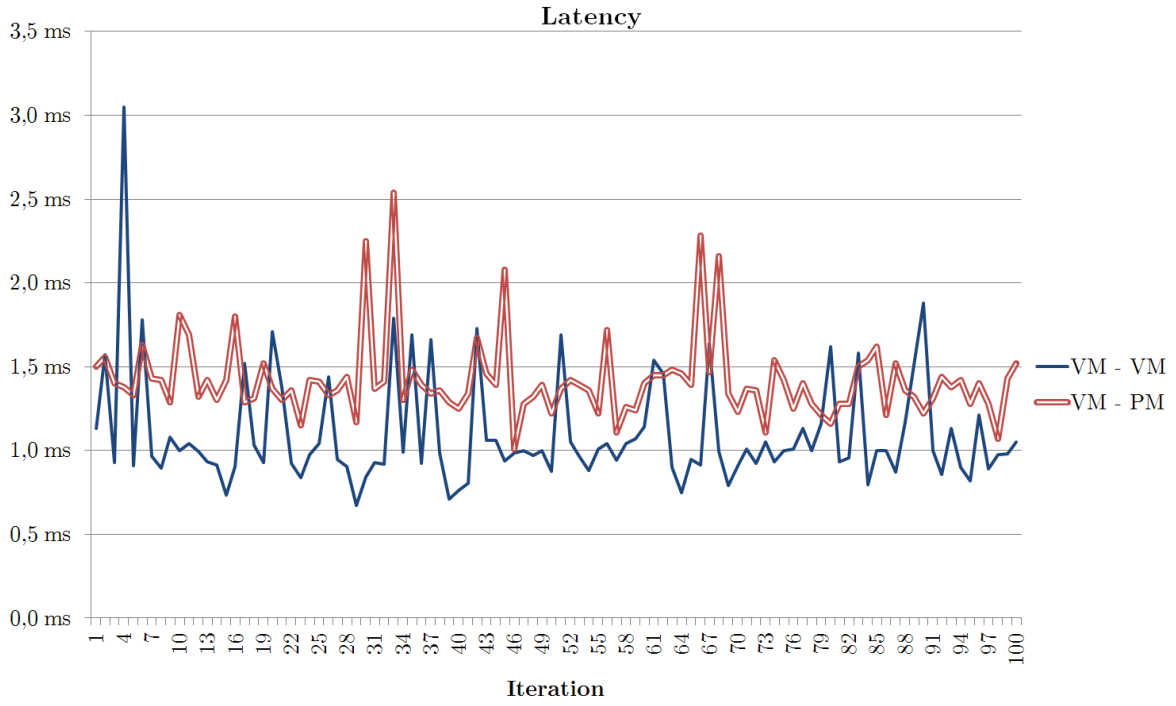


Figure 7.4: Latencies obtained in both Local and Remote cases.

In addition, the extra test of measuring how long does it take for the OpenStack host to reach one of its Nova VMs, has its results presented in Table 7.5.

Average	0.59 ms
Standard Dev.	0.14 ms

Table 7.5: Latency between host and VM.

DISCUSSION

Even though the latency tests were performed after a "warm-up" period, no memory swapping was undertaken by any operating system and the physical network was free of traffic, ping delays oscillate considerably. This can be seen in Figure 7.4 and Tables 7.3 and 7.4 which state the standard deviances. Nevertheless, they never oscillate above 3.05 milliseconds for the Local Latency test or 2.54 milliseconds for the Remote Latency test.

What is most important to notice in these tests is how the average latency changes when moving from a VM to VM scenario to a VM to External Port scenario. The average of the former is 1.10 milliseconds while the latter is 1.42 milliseconds, an increase of approximately 29%. The first result is not influenced by the work proposed as there are no dependences between them. The average of 0.59 milliseconds that the OpenStack host requires to exchange an ICMP Echo Reply/Request with a VM, without anything else deployed, also demonstrates this independence. Given that both VMs are run by the same Compute/Nova node and the External Port computer is physically located outside this node and its network (and behind an OpenWrt Ethernet switch), this delay increase is not very significant and is within expectations for such a network deployment scenario. Furthermore, applications with medium latency requirements can still be provided by recurring to an External Port as deployed

for these tests. For applications with critical latency requirements (such as voice), they can still be leveraged for the measured values, although if more hops are introduced between the Cloud Computing infrastructure and the AD, latency will surely increased and latency-critical applications cannot be deployed. Nevertheless, the introduction of new hops is a problem that affects other scenarios or network architectures and is strictly an external factor to this work. In essence, the increase in latency imposed by this work is thus negligible.

It is expected that these latencies can be significantly reduced, for latency-critical scenarios, by properly setting up QoS in the underlying network as well as inside the Network Node (by using OpenStack extensions and Operating System's changes dedicated for that purpose), as well as decreasing the number of hops.

7.3.3 TRAFFIC THROUGHPUT

Both tests of Local, Remote downstream and Remote upstream Traffic Throughput have been iterated 20 times.

Local traffic throughput tests are hereafter named "VM > VM" based on the fact that traffic is sent from one Nova VMs to another one, which one is not important given their twin nature and network symmetry as described in the test specification 7.2.3.

There are two remote traffic throughput tests, one for each traffic direction. Therefore, when traffic is sent from a Nova VM to a computer acting as an External Port, it is a Remote downstream traffic throughput test and the test label is "VM > PM". Similarly, for the opposite traffic direction, the test is an upstream traffic throughput test and the test label is "PM > VM". Consequently, naming follows a perspective centered in the External Port.

The *xx* axis of Figure 7.5 shows the iteration number (from 1st to 20th), while the *yy* axis shows traffic throughput (no headers, data only) in Mbps.

Tables 7.6, 7.7, 7.8, 7.9, 7.10 and 7.11 present average and standard deviance variables of TCP and UDP's Local, Remote Downstream and Remote Upstream Throughput tests, respectively.

Average	290,1 Mbps
Standard Dev.	20,0 Mbps

Table 7.6: Local TCP traffic throughput statistics.

Average	26,7 Mbps
Standard Dev.	1,6 Mbps

Table 7.7: Local UDP traffic throughput statistics.

Average	90,0 Mbps
Standard Dev.	1,0 Mbps

Table 7.8: Remote upstream TCP traffic throughput statistics.

Average	58,9 Mbps
Standard Dev.	4,6 Mbps

Table 7.9: Remote upstream UDP traffic throughput statistics.

Average	77,5 Mbps
Standard Dev.	2,3 Mbps

Table 7.10: Remote downstream TCP traffic throughput statistics.

Average	63,1 Mbps
Standard Dev.	3,3 Mbps

Table 7.11: Remote downstream UDP traffic throughput statistics.

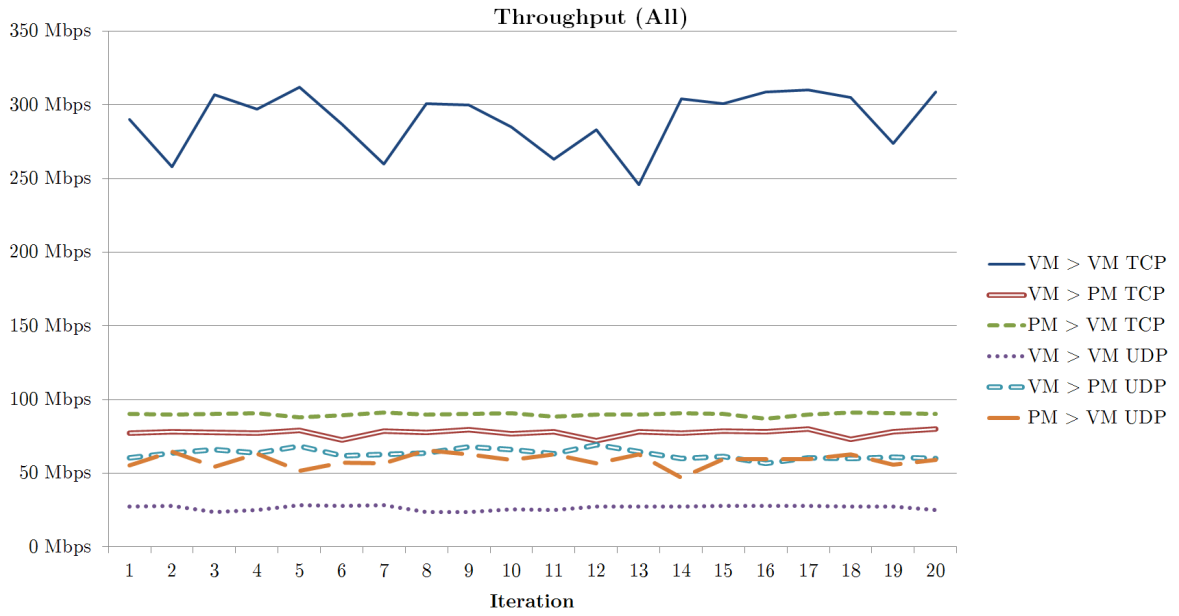


Figure 7.5: Traffic throughput obtained for TCP and UDP.

DISCUSSION

Local and Remote Traffic Throughput are split in six different tests, aiming to measure throughputs in the three possible scenarios when there are two VMs and one External Port deployed, twofold: one for UDP and one for TCP.

Figure 7.5 presents an interesting finding: throughput between VMs is the highest for TCP but, conversely, is the lowest for UDP. The cause for this lies in the fact that, for traffic between VMs, there is a higher processing load. Because there are 2 VMs involved instead of just one as in the Remote Traffic Throughput tests, this load only becomes apparent for the Local Traffic Throughput tests. They both run inside the same Compute node, which stresses CPU usage and traverses the virtual network stack twice. Consistent with this reasoning is the throughput obtained for the the Remote Traffic Throughput tests, where only 1 of the VMs are involved, and is roughly 2 times the throughput of the test between VMs. The actual reason for UDP being this heavy when packets are being sent/received between VMs, to the point it becomes a bottleneck, can be attributed to `iperf`'s default UDP buffer size of 160 KiB which fills up very quickly and stresses the machine.

Both TCP and UDP throughput measures were not very distant for each remote direction tested. In the case of TCP, the upstream direction is, in average, 16.1 % faster than the downstream direction. For UDP, the downstream directions wins by 7.1 %. This discrepancy can be attributed to some security rules imposed by Neutron to the traffic, as well as other non-symmetric mechanisms of sending and delivering traffic from/to Nova instances.

All tests have shown sustained network speed, with quite low standard deviances. Although standard deviance is not high for the TCP test between VMs and the UDP throughput test from the Physical Machine (PM) (External Port) to the VM, sustained speed for these specific cases demonstrated to be more volatile.

What these results mean for the deployability of the work is that throughput will be inferior than having only VMs inside one same machine, although that is not how medium to large virtualized datacenters are made of. For datacenters with multiple compute nodes, there is likewise an impact on the throughput between VMs there provisioned. In addition, these VMs are usually interconnected (if

part of the same virtualized network) via a tunneling technology, for instance GRE. In other words, the use of tunnels does not not negatively impact network performance/throughput more than it does today. The AD being used is the one that may instill limitations, due to processing overhead and available internal bandwidth, but that is dependent on the hardware used.

7.3.4 TRAFFIC OVERHEAD

The absolute traffic overhead per packet is the sum of Ethernet (14 bytes), IP (20 bytes) and GRE (8 bytes) headers which total 42 bytes. In a typical network deployment with NIC interfaces set to use an MTU of 1500 bytes, the actual amount of data that can be transferred in a single packet (including transport layer headers) is 1480 bytes (when using the IP protocol because its header occupies 20 bytes from the MTU). In the same conditions, the implementation proposed in this work allows for an actual amount of data of 1438 bytes to be transferred in a single packet ($1480 - 42 = 1438$ bytes).

Therefore, traffic overhead is approximately $1480/1438 = 2.92$ % for full packets, which results in the same increase of packet count when transferring sequential data.

However, for traffic that consists in smaller packets, the traffic overhead proportion is greater. Sinha et al. provide a technical report on the distribution of packet sizes on the Internet in October 2005 [61]. They observed a strong mode of 1300 B packets (L2 data length) in some cases. Besides, packet sizes seemed to follow a mostly bimodal distribution for 40 B packets and 1500 B packets (at 40 % and 20 % of packets, respectively). Even though the abundance of 1500 B packets as reported by [61] are very likely to be tied with sequential data transfers, it is still interesting to analyze the overhead of individual packets with this size. It may seem that the packet overhead is very similar to the one obtained for the traffic overhead in sequential data transfer (because all length of the MTU is used) but what actually happens is that a second packet must be sent to transfer the whole 1500 B. Ignoring the processing overhead implied by a second packet being necessary and focusing only on total packet size overhead, the first packet is able to transmit 1458 B of IP data and the second transmits the rest (42 B). It then results in a total overhead of $(1518 + 18 + 42)/1518 = 3.95$ %. The other two numbers are important to analyze due to their frequent activity on the Internet The packet overhead proportion for packets with a size of 1300 B (1280 B of IP data, 1318 B with FCS and all headers included) would translate in: $1360/1318 = 3.19$ %. For 40 B packets instance, the overhead is $100/58 = 72.41$ %. Table 7.12 summarizes traffic/packet overheads for the cases described as well as for other typical packet kinds, with Ethernet data sizes specified. Moreover, Figure 7.6 presents a graph with the packet overhead per individual packet size (defined for Ethernet data lengths between 1 and 1458 bytes) as given by the GRE overhead formula present in Equation 7.1, where n is the Ethernet data length, in bytes. The reasoning behind this equation is the ratio of a packet with GRE overhead against a normal packet, minus the packet itself (100 %). A normal packet has 18 more bytes due to the Ethernet and FCS, while the same packet via GRE has an additional 42 bytes of overhead, totaling 60 bytes.

$$f(n) = \frac{n + 60}{n + 18} - 1 \quad , n \in [1, 1458] \tag{7.1}$$

Packet description	Size	Overhead	Comments
Sequential transfer	1500 B	2.92 %	Packet count increases with the same ratio.
1500 B packet	1500 B	4.95 %	Requires a second packet, more processing overhead.
1300 B packet	1300 B	3.19 %	
DHCP Discover	328 B	12.14 %	A possible size for a DHCP Discover.
ICMP Echo Request	84 B	41.18 %	48-byte ICMP data.
TCP Ack IPv6	72 B	46.67 %	Without TCP data.
DNS Query	60 B	53.85 %	For a domain with 14 characters.
TCP Ack IPv4	52 B	60.00 %	Without TCP data.
40 B packet	40 B	72.41 %	

Table 7.12: Examples of packets and associated GRE overhead.

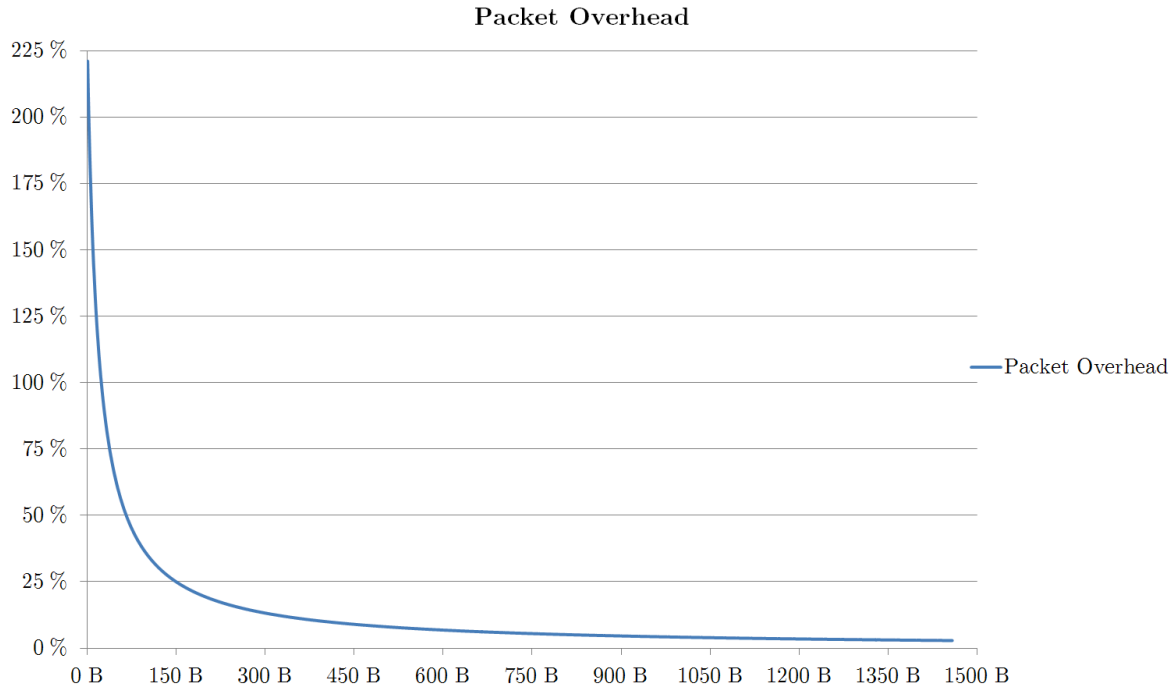


Figure 7.6: Packet overhead per individual packet size, using GRE.

DISCUSSION

Usually, packet overhead is fixed due to the GRE tunneling technology used, which imposes an additional 42 bytes of headers. However, the proportion of overhead varies for each packet size, which results in a variable traffic overhead when multiple packets are being sent through the network. It has been shown that the overhead beyond L3, i.e. IP data length, is approximately 2.92% for a typical network deployment relying on an MTU of 1500. What this means is that, for a sequential file transfer over the network, approximately 2.92% more packets will be sent (and probably acknowledged, depending on the transport protocol). Besides the total increase in traffic, CPU, networking hardware Application-Specific Integrated Circuits (ASICs) and main memory usage at the endpoints and intervening nodes, such as switches and routers, will increase, resulting in lower overall performance. However, this increase is (arguably) negligible, especially when taking into account the rest of the benefits and potential benefits of this solution.

Another important point is that packets with a size within a 42-byte boundary of the MTU require a second packet to send the rest of the data (assuming fragmentation like this is not an issue, which most likely is thus requiring External Ports to customize their MTU to 1458 bytes either manually of via the Neutron DHCP server). This increases both traffic overhead and processing overhead (highly

dependent on hardware and operating system). Furthermore, for the remaining packet sizes, it can be seen that as the size decreases, the proportional overhead increases.

Even though packet overhead may be too high for some packets, proportionally, it does not invalidate the deployability of this work with the GRE technology, unless in very specific scenarios with tight requirements. For large packets, they are usually followed by more packets of the same kind, usually for stream transfers, keeping overhead below 10 %. For medium packets, overhead will be more significant but still usually below 10 %. For small packets, where packet overhead (and traffic overhead for that matter, as multiple packets may be transfered in a short time span) is proportionally more significant, it may not be a concern anyway because the packets are small. In other words, small packets with the GRE overhead are still small packets (only slightly larger) so they will most likely not disrupt packet latencies and consume all of the link's capacity. However, one scenario where the high overhead of small packets is noticeable and undesirable occurs when they are very frequent and the dominating kind of packets, contributing to the exhaustion of the link's capacity, effectively contributing to a steep decrease in Traffic Throughput in comparison to a traditional link. For exactly the same conditions, traffic without the GRE overhead would be much swift and lighter. Nevertheless, this is a very specific scenario and the GRE overhead is expectedly compatible with the majority of scenarios and use cases. Plus, the advances in networking equipment and computers have led to processing power and bandwidth higher than ever, cheaper than ever.

CHAPTER 8

Conclusion

This work presented a way of joining the advantages of modern Cloud Computing software stacks, mainly their ability to provide network connectivity as a service to tenants coupled with inherent Cloud Computing properties like flexibility and on-demand service, with the benefits of having an heterogeneous, bare-metal or eventually legacy network deployed, by making both parts integrate and inter-operate in a seamless way, with control and administration logically centralized. Results have shown that these advantages are not eclipsed by major problems, again empowering the feasibility of the work.

Starting with the premise that Cloud Computing has become, and is still becoming, more popular, an important disadvantage of the concept was identified: the necessity for homogeneous resources. Therefrom, the University of Aveiro's network infrastructure was picked as an example of a motivation for turning existing, heterogeneous networks into Cloud-managed ones, with the ability to be transparently controlled and administrated in a centralized and Cloud-like manner. Moreover, there was a commit to create a solution that would be able to respond to the motivation, essentially by leveraging a set generalized set of objectives. The first objective was to integrate with an existing Cloud Computing solution without breaking its own functionality. The second was to be able to connect to existing, remote networks, and making them belong to the Cloud Computing network infrastructure. Finally, leverage a foundation for heterogeneity in terms of networking equipment and technologies supported.

With relevant conceptual and technical background provided, five major scientific and technical endeavors were presented as the State of the Art in respect to the motivation of this work. Whilst all of them offer solutions with clear advantages over prior technology, they come with their own drawbacks, which are stated likewise.

Furthermore, and after understanding the limitations in the State of the Art technology, the general problem to solve was stated. Alongside it, a list of the most important use cases was outlined and discussed. Some of these use cases relate to the drawbacks in state of the art technology, others come as new overhauls for what can generally be achieved with Cloud Computing or traditional computer networks.

Taking into account motivation, objectives, State of the Art technology limitations, and use cases to be leveraged with the problem solved, a design for the solution was presented. Aiming to be as generic and technology-agnostic as possible, the design specified the main features to be provided by the work, which can also be seen as lower-level use cases. It also introduced the basic concepts of the

components involved and a general architecture, also in a generic way so mapping these to a future implementation technology can be accomplished with less effort. Finally, it described what functional processes should be leveraged and how they work, or can work, internally.

A natural development for the solution design was a proposed implementation. This design fueled and originated an implementation on top of OpenStack, an existing FOSS Cloud Computing software stack, mapping design details to implementation components. With the major objectives in mind, which had their basis on the motivation, this solution implementation effectively leveraged all of them. Where design was abstract and scarce in details, the implementation chapter explained each component, requirement, and other details like data structures or APIs, with a high degree of depth.

Finally, an evaluation of the solution's implementation was carried out, results presented and analyses made, discussing results obtained. Given the external nature of the networks which are connected to the implemented system, results are satisfactory in the sense that these networks do not perform poorly and at the same time are enabling new features and use cases (as presented).

Subjective results are results which cannot be measured and, by nature, are not as important as the objective, measurable ones, when discussing test results. Nonetheless, it is worthwhile to note them. This work provides positive results in administrative and network engineering flexibility. It enables heterogeneous network equipment to be integrated into a Cloud Computing infrastructure and make it as easy as requesting a few API requests. In this regard, OpEx are reduced, which is a measurable result although not attainable by this dissertation alone. Potentially, CapEx can also be reduced, by reducing and/or postponing acquisition of new equipment to fulfill new use cases or integrate with modern network infrastructures, probably based on Cloud Computing. When talking purely about network administration, the number of operations carried out by an administrator can be reduced if most of it can be achieved by an API. When talking about a Telecommunications Operator, this opens doors for gradual migration into NFV, for instance, by reusing dedicated hardware instead of fully relying on modern software that replace the old hardware. Migration benefits also apply to other sorts of companies, even those that only require a private Cloud. For a Cloud Computing provider, it may mean, for example, that there is a pool of high-performance bare-metal machines already assigned to the system and the only thing the Cloud consumer does is requesting a new External Port to access one of these machines. It will be up and running inside its tenant network automatically, without any human intervention.

8.1 FUTURE WORK

Although the proposed implementation can already be deployed and its use cases harnessed with reduced or no additional effort, multiple paths of future work can be traced which further enrich and strengthen this solution and the feasibility of its use cases. As such, it can be extended, in the future, with new features and capabilities for notable scenarios with specific requirements that present new challenges and define a new generation in State of the Art.

The first interesting addition to this work is to implement a robust manner of detecting new External Ports, via an Attachment Driver (whose architecture already expects this functionality), for instance by recurring to Link Layer Discovery Protocol (LLDP) (described in [62]) or SNMP (described in RFC 1067 [63]). Alternatively, detection could be made by developing an internal mechanism (analogous or built on top of the experimental one already created) that analyses traffic

and detects when new hosts have joined, for instance by recurring to OVSDB (described in RFC 7047 [18]) notifications, IP Flow Information Export (IPFIX) (described in RFC 5101 [64]) notifications, or by reacting to client DHCP packets like the experimental one, although probably without having to sniff traffic.

Another interesting addition to this work is inherent support for High Availability (HA), which proves especially helpful for Telecommunications Operators, but not limited to them.

Extreme robustness and security improvements (not just by relying on External Driver's capabilities) are amongst other desirable characteristics not explored in this dissertation, which may be a strict requirement for critical deployments as well as for Telecommunications Operators.

Another interesting future work is the ability to integrate this work with an orchestration project, such as OpenStack's Heat, to accelerate and streamline multi-Cloud deployments that make use of hybrid networks based on this work. Naturally, there are other projects that might be interesting to further integrate with this work, materializing the use cases presented. An example is SFC with External Ports, which might be implemented by integrating the Traffic Steering work presented in [49].

Finally, the ability to create external ports in a cross-Cloud manner, i.e. merging network segments from disparate mutual Cloud Computing deployments, is an interesting use case currently not achievable by this work.

Appendices

Appendix A

Blueprint: Provider Router Extension

- (WIP) -

Neutron Provider Router Extension

Table of contents

- (WIP) -

History

Version	Date	Authors	Content Changes
0.1.0	2013/06/30	~fmanco	- As discussed for Havana.
0.2.0	2014/04/07	~igordcard	- General formatting, sectioning, writing changes and corrections; - Some changes in content that don't change the overall picture; - Removed PHY networks; - Updated distant provider router DB and API representation; - Replaced L3 agent with DPR agent, updating dependencies accordingly;
0.2.1	2014/04/09	~igordcard	- Nomenclature of the distant provider router now consistent throughout all document; - Added merge objective (milestone target) of juno-1; - Added DPR driver configuration parameters information.
0.2.2	2014/04/11	~igordcard	- Created Concerns section; - Added 5 concerns to the Concerns section;
0.2.3	2014/05/03	~igordcard	Blueprint specification has moved to the new neutron-specs repository: https://review.openstack.org/#/c/91925/

Now under review on neutron-specs:

https://review.openstack.org/#/c/91925/

Scope

Specify a Neutron API extension that provides a way to map and register distant non-OpenStack routers to the Neutron database, allowing them to be used by tenants with complete API support.
Such functionality will ease integration and on-demand connectivity between legacy network infrastructures and cloud-provided virtual networks, also useful for incrementally migrating from the former to the latter.
In the context of this blueprint: provider router, distant router, distant provider router, physical router and legacy router all share the same meaning. To avoid confusion with the existing "provider router" term (http://docs.openstack.org/trunk/install-guide/install/yum/content/section_networking-provider-router_with-provate-networks.html), the "**distant provider router**" or "**DPR**" terms will be preferred throughout this document.

Use cases

Currently, the integration of existing physical/legacy components with a virtual environment is of vital importance to allow the migration of currently deployed (legacy) infrastructures to new networking clouds (public or private).
The main use cases are thus related to connecting physical environments to virtual ones through a legacy network that cannot be easily changed.

- Connect a virtualized environment to distant physical networks on demand;
- Have physical hosts on a private physical network being able to reach VMs on a private virtualized environment;
- Have a router connected to at least two different networks: a directly connected one and a distant virtualized one (located at the cloud) but nevertheless seen as directly connected;
- Maintain addressing and reachability unchanged from the legacy network's perspective, i.e., everything looks as though the (now virtual) physical network is still there (and still physical).

Although deploying a VPN, possibly with Neutron's VPNaaS, could serve a similar purpose, the objective is to maintain the legacy network as untouched as possible, configuring everything from the OpenStack side and making sure the 4th use case above is possible.

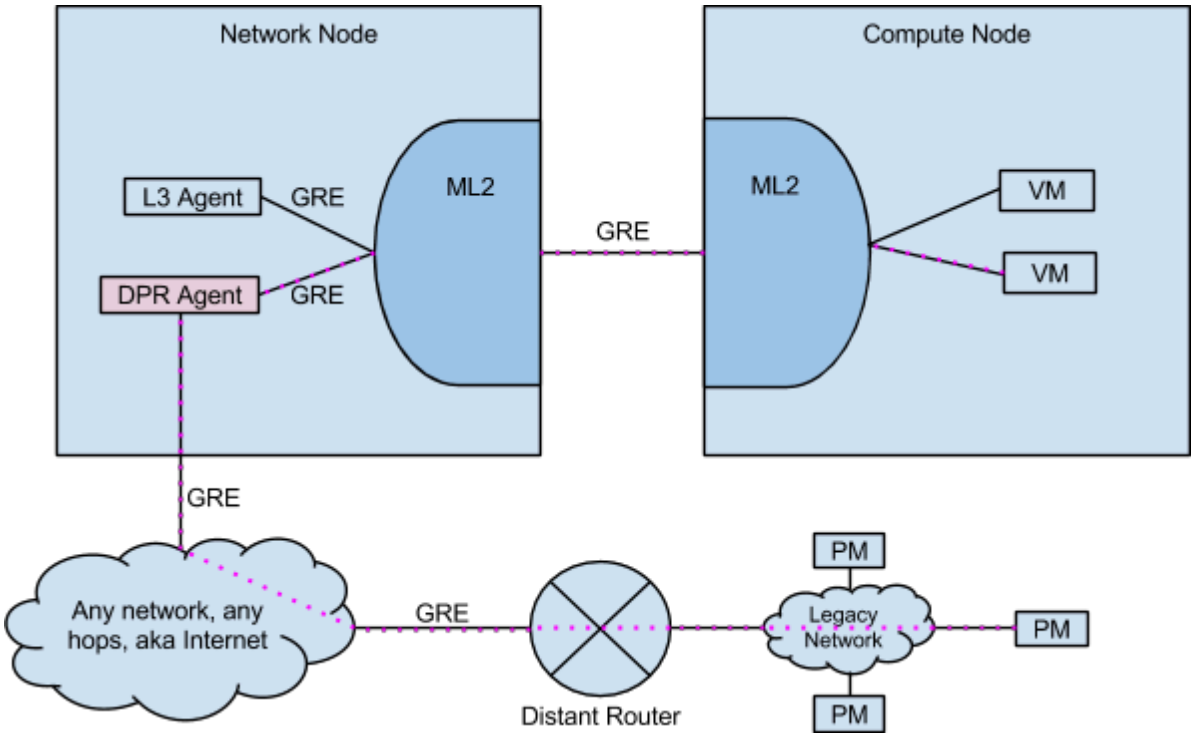
Implementation Overview

There are three main points to consider for the implementation of this extension:

- Who/what is responsible for communicating changes to the physical routers;
- How to communicate with physical routers:
    - This should be extensible in order to support any router;
- How router interfaces will be connected to tenant networks:
    - The traffic between the VMs and the physical routers should be isolated;
    - The traffic must stay on the same IP network;

Regarding the first point, I'm thinking about having a special agent, built as an agent per https://blueprints.launchpad.net/neutron/+spec/quantum-l3-routing-plugin, called DPR Agent (DPR: distant provider routing, not to be confused with just DPR: distant provider router).
When an operation is executed (e.g. `router-interface-add`) the agent will communicate

that change to the correspondent physical router.
For the second point, the management of the physical routers will be done through a driver implementing a generic interface. This will keep the software easily extensible and enable the use of any equipment from a linux box to a carrier grade router by implementing the respective driver. The driver implementation will be up to its creator and dependent on the physical router characteristics. It may use a management interface, through telnet or ssh, the SNMP protocol, or any other mechanism that is appropriated. This driver is connected to the DPR agent plugin, implementing its interface for drivers.
For the third point, the technology used to connect the router to the tenant network (e.g. GRE) *is determined by the DPR agent plugin* in use **must be supported both by the L2 plugin used in the compute nodes (e.g. ML2) and the distant provider router**.
An example of a physical machine at the legacy side of the distant provider router communicating with the virtual side of the distant provider router, using ML2 and network type GRE, is presented in the following diagram (pink dots follow the communication path):



Physical networks

Reachability with the physical networks at the other side of the distant provider router should be straighforward, as machines will be configured with some gateway IP address either manually or by DHCP, exactly the same as in a typical environment.

Implementation plan

The objective is to merge this extension into OpenStack (milestone target) by **`juno-1`** (around the beginning of June).
The implementation will be divided in the following work items:

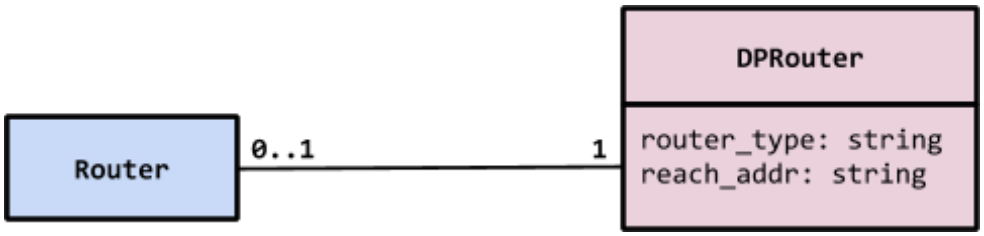
1. API modifications;
2. Changes to the database model and related code;
3. Sketch the driver interface;
4. Implement DPR Agent;

a. Tenant network support;
b. `router:external` network support;
c. Floating IPs support;

These will be performed in individual commits to be sent for review.

Data Model Changes

The database model extension is shown in the diagram below. It adds a new table to keep track of the distant provider router's specifics. This table contains the IP address used to reach the router, which might be somewhere on the Internet, and router type, to select the driver used by the DPR agent to decide how to apply configurations to the router (as explained previously).
TODO



Configuration Variables

A new configuration variable is necessary to provide the mapping between the `router_type` and the correspondent driver. The variable `provider_router_driver_mapping` will go to the `dpr_agent.ini` file.

This will be further analysed before a milestone is set for this blueprint.

APIs

The provider-router blueprint extends the Neutron API by creating new attributes. No new resources or actions are defined.

Extended attributes

The router structure will be extended with 2 new attributes in the provider namespace:

- `provider:router_type` [CR]:

  Specifies the type of router. This will be used to choose the driver that will handle the communication with the physical router.

- `provider:reach_addr` [CRU]:

  Specifies the IP address used to reach the router, which might be somewhere on the Internet.

Authorization

By default, distant provider routers can only be created or deleted by admin users. The

provider attributes are also only visible, and can only be updated by admins.
This routers can nonetheless be assigned to any tenant. The tenant will then be able operate the router the same way it operates a virtual router.

Plugin Interface

TODO: Clarify how ML2 will be able to establish GRE tunnels with the distant provider router.

Required Plugin Support

At least ML2.

Dependencies

Neutron components

If using ML2, at least the Open vSwitch mechanism driver.

Python dependencies

The extension itself doesn't have any specific requirements. However it is possible that the used DPR agent drivers introduce new requirements. It is expected that, to manage the routers, drivers will use telnet (https://docs.python.org/3.4/library/telnetlib.html) , SSH (https://wiki.python.org/moin/SecureShell) , SNMP (http://pysnmp.sourceforge.net) or other technologies.

New Neutron Blueprints

[There are some blueprints currently in development, targeting the Juno release, that may impact the implementation of this blueprint.]

Previous Neutron Blueprints

Some previous Neutron blueprints are worth checking out as they may conflict or boost development of the provider-router blueprint.

Migrate L3 router service from mixin to plugin

https://blueprints.launchpad.net/quantum/+spec/quantum-scheduler

This blueprint moves L3 functionality to a separate plugin. Implementation will take that in consideration, but it doesn't entail many differences since the changes will still be done mainly to the `L3NATAgent` and `L3_NAT_db_mixin` classes.

Multiple L3 and DHCP agents for Neutron

https://blueprints.launchpad.net/quantum/+spec/quantum-scheduler

The most direct consequence of this blueprint is that the physical router will be connected to multiple compute nodes instead of just one network node.

Neutron Multi-host DHCP and L3

https://blueprints.launchpad.net/quantum/+spec/quantum-multihost

Make Authorization Orthogonal

https://blueprints.launchpad.net/quantum/+spec/make-authz-orthogonal

Implementation will conform to the improved authorization mechanism.

Concerns

Router configurations

The DPR agent driver also needs configuration parameters to each of the routers it will be communicating to, e.g. credentials. Making these configurations parameters part of the Neutron API extension for distant provider routers is not feasible (see Salvatore Orlando's comment start with "I would like to not see appear anywhere in the quantum API"). Having the configuration parameters, for each router, inside the actual DPR agent config file means duplicated effort when adding a distant provider router to Neutron (add it both through the API and by editing the DPR agent configuration file with its configurations) while automatically disallowing the possibility of adding them per tenant. Another possibility is by making use of service types: https://review.openstack.org/#/c/29750/, as Salvatore Orlando recommended.

CIDR overlapping

The way this blueprint is specified doesn't take into account CIDR overlapping (at the DPR side), and eventually loss of reachability from Neutron to the DPR, when GRE tunnels are established end to end. For now, I will just assume someone has already gotten rid of the old legacy network in the DPR, and replaced it with some other network reachable from Neutron (or the DPR's loopback address reachable from Neutron). Or, in case there's just no need to keep addressing and reachability between legacy network unchanged, the DPR only needs to be already reachable and the virtual networks to attach to it must not overlap any other legacy networks' CIDRs.
However, this is a serious concern that must be well discussed. For the sake of having at least a PoC working soon I will do what I told in the first paragraph, for now.
The network through each the DPR is accessed is called "**Migration Network**". It is the network through which tunnels are established (at the DPR side), and through which Neutron reaches the DPR. This network has a different CIDR from the legacy network (virtual afterwards), and from any other networks connected to the DPR either virtual or legacy, and is directly connected to the DPR.

NAT

If NAT is used at the legacy network, it may be difficult to or just undesirable to reconfigure the network in a manner that then allows it to be extended into OpenStack.

DHCP

DHCP addressing can be provided either internally (from OpenStack) or externally (from the legacy network). However, care must be taken to avoid conflicts and make sure control is logically centralized, ideally inside Neutron.

Broadcasting

Broadcast packets may carry expensive overhead given the disparate locations of the virtual and legacy parts of the network, plus the tunneling technology used to connect them.

CLI Requirements

The `router-create` and `router-update` operations will need to support the new provider attributes. The Neutron client application already handles this, so no changes needed.

- (WIP) -

Horizon Requirements

TODO

Usage Example

Creation of a distant provider router with the Neutron client will look like:

```
neutron router-create distant_r1 \
       --provider:reach_addr 10.13.1.1
       --provider:router_type IOS12
```

Test Cases

TODO: add more, better tests.
Unit->Functional->Unit+Functional->Integration

New unit tests will be added to test the manipulation of distant provider routers. The following components must be tested:

- Database manipulation code for provider attributes;
- DPR Agent operation with distant provider routers:
    - Internal networks;
    - External networks;
    - Floating IPs;

As a physical router is not available for testing, a dummy driver can be used to check the correct operation of the DPR Agent.

Contributors

Igor Duarte Cardoso (igordcard) is currently the main contributor of this blueprint, although it was originally written by Filipe Manco (fmanco).

Rui Aguiar, Diogo Gomes (dgomes), João Paulo Barraca and Carlos Gonçalves (cgoncalves) are active advisors.

References

[1] F. Manco, *Appendix C - Provider Router Extension, Blueprint*, Network infrastructure control for Virtual Campus, 2013.

[2] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. 2005, p. 760, ISBN: 0131858580.

[3] H. LeHong and J. Fenn, "Hype Cycle for Emerging Technologies, 2014", *Gartner*, 2014.

[4] A. Baritchi, *Cloud Computing: The Mainframe Reincarnated*, 2008. [Online]. Available: `http://de.sys-con.com/node/723583`.

[5] P. Mell and T. Grance, "The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology", *Nist Special Publication*, vol. 145, p. 7, 2011, ISSN: 00845612. DOI: `10.1136/emj.2010.096966`.

[6] V. Consortium, "VPN Technologies: Definitions and Requirements", *VPN Consortium*, 2008. [Online]. Available: `http://www.vpnc.org/vpn-technologies.html`.

[7] R. Venkateswaran, "Virtual private networks", *IEEE Potentials*, vol. 20, 2001, ISSN: 0278-6648. DOI: `10.1109/45.913204`.

[8] S. Kent and K. Seo, *Security Architecture for the Internet Protocol*, RFC 4301 (Proposed Standard), Updated by RFC 6040, Internet Engineering Task Force, Dec. 2005. [Online]. Available: `http://www.ietf.org/rfc/rfc4301.txt`.

[9] J. Gilmore, *Network Encryption - history and patents*, 1997. [Online]. Available: `http://www.toad.com/gnu/netcrypt.html`.

[10] N. M. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization", *Computer Networks*, vol. 54, pp. 862–876, 2010, ISSN: 13891286. DOI: `10.1016/j.comnet.2009.10.017`.

[11] D. McPherson and B. Dykes, *VLAN Aggregation for Efficient IP Address Allocation*, RFC 3069 (Informational), Internet Engineering Task Force, Feb. 2001. [Online]. Available: `http://www.ietf.org/rfc/rfc3069.txt`.

[12] J. Smith, K. Calvert, S. Murphy, H. Orman, and L. Peterson, "Activating networks: a progress report", *Computer*, vol. 32, 1999, ISSN: 0018-9162. DOI: `10.1109/2.755003`.

[13] N. Feamster, J. Rexford, and E. Zegura, "The Road to SDN: An Intellectual History of Programmable Networks", *ACM Sigcomm Computer Communication*, vol. 44, pp. 87–98, 2014, ISSN: 01464833. DOI: `10.1145/2602204.2602219`.

[14] D. Kreutz, F. M. V. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey", p. 49, 2014. arXiv: `1406.0440v1`.

[15] B. Naudts, *Evaluating the impact of SDN on CapEx and OpEx*, Italy, 2013. [Online]. Available: http://sites.ieee.org/sdn4fns/files/2013/11/SDN4FNS%5C_panel%5C_presentation%5C_Bram%5C_Naudts.pdf.

[16] N. Mckeown, T. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, J. S. Turner, and S. Louis, "OpenFlow: enabling innovation in campus networks", *Computer Communication Review*, vol. 38, pp. 69–74, 2008, ISSN: 01464833. DOI: 10.1145/1355734.1355746.

[17] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending Networking into the Virtualization Layer.", in *8th ACM Workshop on Hot Topics inNetworks*, vol. VIII, 2009, p. 6.

[18] B. Pfaff and B. Davie, *The Open vSwitch Database Management Protocol*, RFC 7047 (Informational), Internet Engineering Task Force, Dec. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc7047.txt.

[19] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog Computing and Its Role in the Internet of Things", *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pp. 13–16, 2012, ISSN: 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513.

[20] L. Sell, *OpenStack Launches as Independent Foundation, Begins Work Protecting, Empowering and Promoting OpenStack*, 2012. [Online]. Available: http://goo.gl/8FKI5T.

[21] J. Curry, *Introducing OpenStack*, 2010. [Online]. Available: http://www.openstack.org/blog/2010/07/introducing-openstack/.

[22] A. Williams, *The Top Open Source Cloud Projects of 2014*, 2014. [Online]. Available: http://www.linux.com/news/enterprise/cloud-computing/784573-the-top-open-source-cloud-projects-of-2014.

[23] Community, *OpenStack 2014.1 (Icehouse) Release Notes*, 2014. [Online]. Available: https://wiki.openstack.org/wiki/ReleaseNotes/Icehouse.

[24] ——, *Test infrastructure Requirements*, 2014. [Online]. Available: http://ci.openstack.org/test-infra-requirements.html.

[25] ——, *Governance*, 2014. [Online]. Available: https://wiki.openstack.org/wiki/Governance.

[26] ——, *Get started with OpenStack - Logical architecture*, 2013. [Online]. Available: http://docs.openstack.org/admin-guide-cloud/content/logical-architecture.html.

[27] A. Migliaccio and S. Orlando, *How to write a Neutron Plugin - if you really need to*, Hong Kong, 2013. [Online]. Available: http://pt.slideshare.net/salv%5C_orlando/how-to-write-a-neutron-plugin-if-you-really-need-to.

[28] J. Denton, "Creating Routers with Neutron", in *Learning OpenStack Networking (Neutron)*, K. Harvey, R. Harvey, S. Panda, S. Poojary, R. Banerjee, S. Chari, and K. Narayanan, Eds., 1st ed., Packt Publishing, 2014, ch. 6, pp. 145–182, ISBN: 978-1783983308.

[29] S. Hanks, T. Li, D. Farinacci, and P. Traina, *Generic Routing Encapsulation (GRE)*, RFC 1701 (Informational), Internet Engineering Task Force, Oct. 1994. [Online]. Available: http://www.ietf.org/rfc/rfc1701.txt.

[30] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*, RFC 7348 (Informational), Internet Engineering Task Force, Aug. 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7348.txt.

[31] R. Kukura and K. Mestery, *Modular Layer 2 In OpenStack Neutron*, Atlanta, 2014. [Online]. Available: https://www.openstack.org/assets/presentation-media/ML2-Past-Present-and-Future.pptx.

[32] Community, *OpenStack Networking (neutron)*, 2014. [Online]. Available: http://docs.openstack.org/icehouse/install-guide/install/apt/content/basics-networking-neutron.html.

[33] ——, *Designing for Cloud Controllers and Cloud Management*, 2014. [Online]. Available: http://docs.openstack.org/openstack-ops/content/cloud%5C_controller%5C_design.html.

[34] ——, *Architecture (OpenStack Havana)*, 2013. [Online]. Available: http://docs.openstack.org/havana/install-guide/install/yum/content/ch%5C_overview.html.

[35] B. Davie, *Network Virtualization Gets Physical*, 2013. [Online]. Available: http://blogs.vmware.com/cto/network-virtualization-gets-physical/.

[36] B. Davie, R. Fortier, and K. Duda, *Physical Networks in the Virtualized Networking World*, 2014. [Online]. Available: http://blogs.vmware.com/networkvirtualization/2014/07/physical-virtual-networking.html.

[37] F. N. N. Farias, J. J. Salvatti, E. C. Cerqueira, and A. J. G. Abelem, "A proposal management of the legacy network environment using OpenFlow control plane", in *Proceedings of the 2012 IEEE Network Operations and Management Symposium, NOMS 2012*, 2012, pp. 1143–1150, ISBN: 9781467302685. DOI: 10.1109/NOMS.2012.6212041.

[38] K. C. Chan and M. Martin, "An integrated virtual and physical network infrastructure for a networking laboratory", in *2012 7th International Conference on Computer Science & Education (ICCSE 2012)*, IEEE, 2012, pp. 1433–1436. DOI: 10.1109/ICCSE.2012.6295333.

[39] F. Manco, "Network infrastructure control for Virtual Campus", MSc Thesis, Universidade de Aveiro, 2013, p. 122.

[40] ——, *Appendix D - Campus Network Extension, Blueprint*, Network infrastructure control for Virtual Campus, 2013.

[41] ——, *Appendix E - ML2 External Port Extension, Blueprint*, Network infrastructure control for Virtual Campus, 2013.

[42] K. Benton, *Neutron External Attachment Points*, 2014. [Online]. Available: https://review.openstack.org/%5C#/c/87825/13/specs/juno/neutron-external-attachment-points.rst.

[43] M. A. Vouk, "Cloud Computing — Issues, Research and Implementations", in *ITI 2008 - 30th International Conference on Information Technology Interfaces*, IEEE, Jun. 2008, pp. 31–40, ISBN: 978-953-7138-12-7. DOI: 10.1109/ITI.2008.4588381. arXiv: 0307245 [hep-th].

[44] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "ELI: bare-metal performance for I/O virtualization", in *ASPLOS '12 Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems4*, 2012, pp. 411–422, ISBN: 9781450307598. DOI: 10.1145/2150976.2151020.

[45] J. Ruckert, R. Bifulco, M. Rizwan-Ul-Haq, H.-J. Kolbe, and D. Hausheer, "Flexible traffic management in broadband access networks using Software Defined Networking", in *2014 IEEE Network Operations and Management Symposium (NOMS)*, 2014, pp. 1–8, ISBN: 978-1-4799-0913-1. DOI: 10.1109/NOMS.2014.6838322.

[46] N. Feamster, "Outsourcing home network security", in *Proceedings of the 2010 ACM SIGCOMM workshop on Home networks - HomeNets '10*, 2010, pp. 37–42, ISBN: 9781450301985. DOI: 10.1145/1851307.1851317.

[47] T. Cruz, P. Simoes, N. Reis, E. Monteiro, and F. Bastos, "An architecture for virtualized home gateways", in *Proceedings of the 2013 IFIP/IEEE International Symposium on Integrated Network Management, IM 2013*, 2013, pp. 520–526, ISBN: 9783901882517 (ISBN).

[48] W. John, K. Pentikousis, G. Agapiou, E. Jacob, M. Kind, A. Manzalini, F. Risso, D. Staessens, R. Steinert, and C. Meirosu, "Research directions in Network Service Chaining", in *SDN4FNS 2013 - 2013 Workshop on Software Defined Networks for Future Networks and Services*, 2013, ISBN: 978-1-4799-2781-4. DOI: 10.1109/SDN4FNS.2013.6702549. arXiv: 1312.5080.

[49] C. Gonçalves and J. Soares, *Traffic Steering blueprint*, 2014. [Online]. Available: https://review.openstack.org/#/c/92477/7/specs/juno/traffic-steering.rst.

[50] RightScale, "State of the Cloud Report", RightScale, Tech. Rep., 2014.

[51] Community, *Chapter 3. Neutron Use Cases*, 2014. [Online]. Available: http://docs.openstack.org/training-guides/content/module002-ch003-neutron-use-cases.html.

[52] G. Dommety, *Key and Sequence Number Extensions to GRE*, RFC 2890 (Proposed Standard), Internet Engineering Task Force, Sep. 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2890.txt.

[53] R. Kukura and K. Mestery, *Modular Layer 2 In OpenStack Neutron*, 2014. [Online]. Available: https://www.openstack.org/assets/presentation-media/ML2-Past-Present-and-Future.pptx.

[54] Community, *Open Stack Client - Human Interface Guidelines*, 2013. [Online]. Available: https://wiki.openstack.org/wiki/OpenStackClient/HumanInterfaceGuidelines.

[55] ——, *OpenStack Style Guidelines*, 2013. [Online]. Available: http://docs.openstack.org/developer/hacking/.

[56] ——, *Neutron Development*, 2013. [Online]. Available: https://wiki.openstack.org/wiki/NeutronDevelopment.

[57] Y.-H. C. Y.-H. Chang, "Wide area information accesses and the information gateways", *Proceedings of lst IEEE International Workshop on Community Networking*, 1994. DOI: 10.1109/CN.1994.337369.

[58] OpenWrt, *The UCI System*, 2014. [Online]. Available: http://wiki.openwrt.org/doc/uci.

[59] Cisco, *Configuring Transparent Bridging*, 2005. [Online]. Available: http://goo.gl/Yu3jyD.

[60] ——, "Switch Virtual Interface for Cisco Integrated Services Routers", 2012, [Online]. Available: http://goo.gl/y6hMYy.

[61] R. Sinha, C. Papadopoulos, and J. Heidemann, "Internet Packet Size Distributions: Some Observations", USC/Information Sciences Institute, Tech. Rep. ISI-TR-2007-643, May 2007. [Online]. Available: http://www.isi.edu/~johnh/PAPERS/Sinha07a.html.

[62] P. Congdon, "Link Layer Discovery - Protocol and MIB - v0.0", IEEE, Tech. Rep., 2002, pp. 1–20. [Online]. Available: http://www.ieee802.org/1/files/public/docs2002/lldp-protocol-00.pdf.

[63] J. Case, M. Fedor, M. Schoffstall, and J. Davin, *Simple Network Management Protocol*, RFC 1067, Obsoleted by RFC 1098, Internet Engineering Task Force, Aug. 1988. [Online]. Available: http://www.ietf.org/rfc/rfc1067.txt.

[64] B. Claise, *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information*, RFC 5101 (Proposed Standard), Obsoleted by RFC 7011, Internet Engineering Task Force, Jan. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5101.txt.