Pós-Graduação em Ciência da Computação

HOTOM: A SDN BASED NETWORK VIRTUALIZATION FOR DATACENTERS

Por

LUCAS DO REGO BARROS BRASILINO DA SILVA

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
<www.cin.ufpe.br/~posgraduacao>

RECIFE
2015

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Lucas do Rego Barros Brasilino da Silva

"HOTOM: A SDN BASED NETWORK VIRTUALIZATION FOR DATACENTERS"

ESTE TRABALHO APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA A UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR: KELVIN LOPES DIAS

RECIFE
2015

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

S586h Silva, Lucas do Rego Barros Brasilino da
HOTOM: a SDN based network virtualization for datacenters / Lucas do Rego Barros Brasilino da Silva. – 2015.
101 f.: il., fig., tab.

Orientador: Kelvin Lopes Dias.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2015.
Inclui referências.

1. Redes de computadores. 2. Computação em nuvem. I. Dias, Kelvin Lopes (orientador). II. Título.

004.6 CDD (23. ed.) UFPE- MEI 2016-104

Dissertação de Mestrado apresentada por **Lucas do Rego Barros Brasilino da Silva** ao programa de Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **HotOM: A SDN based Network Virtualization for Datacenters**, orientada pelo **Prof. Kelvin Lopes Dias** e aprovada pela banca examinadora formada pelos professores:

Prof. José Augusto Suruagy Monteiro
Centro de Informática/UFPE

Prof. Glauco Estácio Gonçalves
Departamento de Estatística e Informática/UFRPE

Prof. Kelvin Lopes Dias
Centro de Informática/UFPE

Visto e permitida a impressão.
Recife, 27 de julho de 2015

Profa. Edna Natividade Da Silva Barros
Coordenadora da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

I dedicate this dissertation to everybody that struggles
day-by-day to make networking even better.

Acknowledgements

First of all, I'd like to thanks God for allowing me be present here, endowed with health, intelligence and persistence to get this work fulfilled. Also, to my surgeon Dr. José Alberto Guerra, for recover me from a deep and painful illness in recent days.

Second, many thanks to my family, for their excitement and support. To my father, Carlos Alberto, and my mother, Dalva, my infinite gratitude for teaching me that achievements comes from hard work. A special acknowledgment to my wife, Simone Cabral Vilaça, for her eternal love and never-ending patience and encouragement, things that makes me stronger every day. My big "I love you!!". An enormous recognition to my brother-in-soul Aldrey Galindo for motivating me, always with a smile in his face.

I'd like to thanks my advisor Kelvin Lopes Dias, for guiding me in the sinuous path of research. Also, many thanks to all Kelvin's students, in special to Edivaldo Cavalcante de Albuquerque Júnior, for the promptitude when configuring and troubleshooting the testbed's components (server hosts and network switches).

To my co-workers at Public Ministry of Labor, in special to my boss Márcia Parga, my acknowledgment for being consentible and for always accept whenever I needed to leave erlier from work to be present in any Cin related activities.

Finally, I'd like to thanks the examining board's components, for accepting to review this work in a very short time.

No nosso livro, a nossa história,
é faz de conta ou é faz acontecer?
—O TEATRO MÁGICO

Abstract

Although datacenter's server hosts have embraced virtualization, the network's core itself has not. A virtual network (VN) is an instance (slice) of network resources such as links and nodes that is built on top of a physical network. Indeed, virtual networking is of paramount importance for multi-tenant datacenters, since it makes management easier.

However, VLANs continue to be used nowadays, driving virtualized datacenters to scalability constraints. VLAN isolates layer 2 (L2) address spaces and indexes them by a 12-bit value, which imposes the hard limit of only 4,096 VNs. Modern Cloud Computing-aware datacenters have being required for delivering IaaS, and are willing to go beyond these scalability restrictions. Even modern tunneling schemes, such as STT, come at a price of overhead because frames are encapsulated by higher layer protocols (UDP, IP). In addition, current virtualized datacenters demand specialized switching hardware (layer 3), increasing datacenter's CAPEX, and require huge computing resources in order to precompute virtual link's states.

Recently, the Software-Defined Networking (SDN) appears as a potential solution for fulfilling those needs by enabling network *programmability*. SDN decouples the network control from the data plane, placing the former in a central controller that exposes an API for developers and vendors. As a consequence, controllers have a unified network's view and are able to execute custom network applications, reaching an unprecedent flexibility and manageability. OpenFlow is currently the most prominent SDN technology.

Even with SDN, many questions remain unanswered. For instance, how to provide scalability and dynamics to a network while preserving legacy core devices? If a datacenter operator can preserve its previous investments, surely he will adopt SDN easier.

This dissertation presents HotOM (**HotO**at**M**eal), a new virtualized datacenter network approach that, by leveraging SDN, overcomes the traditional scalability constraints, enables network programmability while still using legacy network devices, therefore preserving CAPEX. The logic part of HotOM was implemented in Python programming language as a component of the POX OpenFlow controller. HotOM was deployed and evaluated in a real testbed. Analyses were done, from throughput, RTT, CPU time usage to scalability. These metric results were compared against plain VLAN Ethernet network. In addition, a validation of isolation between tenants was performed, as well as a study on protocol overhead. It was confirmed that HotOM scales up to 16.8M tenants, while achieving 47%, 44%, 41% less overhead than STT, VXLAN, and NVGRE, respectively. Finally a qualitative analysis between HotOM and state of the art datacenter virtual network (DCVN) proposals was carried out, showing by comparison that HoTOM consolidates advantages in many functional features: it fulfills almost all evaluated characteristics, more than any other presented technology.

Keywords: Datacenter. Virtual Network. Software-Defined Network. Cloud Computing. Infrastructure-as-a-Service.

Resumo

Apesar dos hosts servidores de datacenters terem abraçado a virtualização, o núcleo de redes não o fez. Uma rede virtual (VN - *virtual network*) é uma instância de recursos de rede, como enlaces e nós, construída sobre uma rede física. De fato, VNs é de suma importância para datacenters multi-inquilinos porque facilitam o gerenciamento.

Porém, VLANs ainda continuam sendo utilizadas, conduzindo datacenters virtualizados a restrições de escalabilidade. Uma VLAN isola espaços de endereçamento da camada 2 e os indexa através de um valor de 12 bits, o que impõe um limite de apenas 4.096 VNs. Datacenters modernos de Computação em Nuvem têm sido requisitados, cada vez mais, a dar suporte à IaaS e, portanto, devem suplantar estas restrições de escalabilidade. Mesmo nos novos esquemas de tunelamento, como STT, há um efeito colateral do *overhead* acrescentando ao quadro das máquinas virtuais, uma vez que estes são encapsulados por protocolos de camadas mais altas (UDP, IP) para transmissão pela rede. Além disso, os atuais datacenters virtualizados exigem dispositivos de comutação especializados, aumentando assim o CAPEX, e necessitam de enormes recursos computacionais para calcular os estados dos links virtuais.

Recentemente, as Redes Definidas por Software (SDN - *Software-Defined Networking*) surgiram como uma solução para atender a tais requisitos ao permitir programabilidade da rede. SDN desacopla o controle da rede do plano de dados, colocando-o em um controlador central que expõe uma API para desenvolvedores e fornecedores. Como consequência, os controladores têm uma visão unificada e são capazes de executar aplicações de rede customizadas, alcançando flexibilidade e gerenciabilidade sem precedentes. OpenFlow é atualmente a tecnologia SDN mais proeminente.

Mesmo com SDN, várias questões permanecem sem resposta. Por exemplo, como prover escalabilidade e dinamicidade a uma rede enquanto se mantém os dispositivos legados no núcleo?

Esta dissertação apresenta o HotOM (**HotO**at**M**eal), uma nova abordagem para redes de datacenters virtualizados que, utilizando SDN, supera as restrições tradicionais de escalabilidade, permite a programabilidade da rede enquanto utiliza dispositivos de rede legados, preservando, assim, o CAPEX. A parte lógica do HotOM foi implementada em Python no controlador OpenFlow POX. O HotOM foi implantado e avaliado em um *testbed* real. Análises da vazão, RTT, tempo de uso de CPU e escalabilidade foram realizadas. Os resultados foram comparados com Ethernet. Adicionalmente, uma validação sobre isolamento entre inquilinos foi realizada, bem como um estudo sobre o *overhead* da proposta. O HotOM escala até 16.8M VNs e obtém 47%, 44% e 41% menos overhead que STT, VXLAN e NVGRE. Finalmente foi conduzida uma análise qualitativa entre HotOM e estado da arte em redes virtuais de datacenter, demonstrando-se comparativamente que o HotOM agrega vantagens: ele atende a praticamente todas as características avaliadas, mais que qualquer outra tecnologia apresentada.

Palavras-chave: Datacenter. Redes Virtuais. Redes Definidas por Software. Comutação em Nuvem. Infraestrutura-como-Serviço.

List of Figures

List of Tables

List of Acronyms

OS *Operating System* 24

VDC *Virtualized Datacenter* 18

EGRE *Ethernet over Generic Routing Encapsulation* 36

GRE *Generic Routing Encapsulation* 27

IaaS *Infrastructure-as-a-Service* 18

CAPEX *Capital Expenditure* 19

OPEX *Operational Expenditure* 19

ToR *Top-of-Rack* 22

Pod *Points of Delivery* 22

RAID *Redundant Array of Inexpensive Disks* 24

LU *Logical Units* 24

STT *Stateless Transport Tunneling Protocol* 25

VXLAN *Virtual Extensible Local Area Network* 25

NVGRE *Network Virtualization using Generic Routing Encapsulation* 25

TSO *TCP Segmentation Offload* 25

LRO *Large Receive Offload* 25

SEQ *Sequence Number* 25

ACK *Acknowledgment Number* 25

STP *Spanning Tree Protocol* 26

VTEP *VXLAN Tunnel End Point* 26

VNI *VXLAN Network Identifier* 26

VSID *Virtual Subnet Identifier* 27

NVE *Network Virtualization Edges* 27

PMAC *Pseudo MAC* 44

AMAC *Actual MAC* 44

PIP *Pseudo IP* 54

LDP *Location Discovery Protocol* 45

LDM *Location Discovery Message* 45

NLA *NetLord Agent* 48

VIF *Virtual Interface* 48

LLDP *Link Layer Discovery Protocol* 50

DHCP *Dynamic Host Configuration Protocol* 50

CARP *Controller ARP update* 55

Contents

1 Introduction

Datacenters and networking, despite been treated distinctly for years, now have reached a need for joint advance in pursuance of addressing huge and complex demands in virtualized environments. Indeed, new demands on providing more flexible and better computing, storage and networking services to clients are appealing datacenters to embrace modern paradigms, such as *Infrastructure-as-a-Service* (IaaS). These innovative ways to think how resources are delivered to *tenants*, are advancing datacenters to create "*slices*" (virtual shares) of these resources, thus allowing tenants to directly configure and manage them. To assure isolation, security, deployability and manageability on each share, some kind of *supervisioning* must be applied. A hypervisor system has to guarantee resource's multiplexing seamlessly, introducing the concept of *Virtualized Datacenter* (VDC)(SOUNDARARAJAN; GOVIL, 2010).

Although virtualization of computing (CPU time, memory) and storage (disk, backup) are currently well supported(VMWare Inc., 2015a; Xen Project, 2015; LVM Project, 2015), networking are not. Datacenters still relying upon limited and inelastic network virtualization mechanisms, such as *Virtual Local Area Network* (VLAN)(IEEE Computer Society, 2014a), that are not flexible enough to provide desirable properties, resulting in limitations like narrow scalability, poor performance isolation, reduced security risk avoidance, faulty application deployment/migration, terrible resource management and *almost no* innovation(BARI et al., 2013).

In order to circumvent the hardness and constraints of traditional network technologies, new paradigms emerged in recent years. One of them in particular has predominantly attracted attention: the *Software-Defined Networking* (SDN)(FEAMSTER; REXFORD; ZEGURA, 2013; KREUTZ et al., 2015). SDN introduces an architecture where network's control plane is separated from data forwarding plane and placed in a centralized controller. With the introduction of OpenFlow(MCKEOWN et al., 2008; OpenNetworking Foundation, 2014), the most prominent SDN architecture nowadays, a myriad of cutting-edge applications can be built by changing network's behavior through *programmability*, attaining an unpreceded flexibility on how networks operate.

Additionally, VDCs are being challenged on how to effectively step forward and push virtualization into network field while properly managing the related resources. Investigations on how to better lead a SDN adoption in datacenters were done recently. Some of them argue that the best way is by deploying edge SDN-aware access switches, while preserving legacy devices in core's network(LEVIN et al., 2013). Others support tat in a near future most of vendors might update their products and datacenters would widely adopt SDN (MYSORE et al., 2009). In an already established and operational datacenter, the

author of this work believes that the former is the best approach because physical topology was already designed and (legacy) network devices were bought, configured and in operation. Requiring datacenters to discard their switches and replace with new SDN-aware ones will surely postpone SDN deployment to the next investment's cycle, and it can take years.

That approach is also endorsed by the "bleeding-edge" network virtualization technologies available to VDCs a this moment. They leverage the use of *Layer 3* (L3) tunnels interconnecting every endpoint, that are usually server hosts. This means that these proposals maintain traditional L3 switches/routers operating in network's core, thus avoiding new SDN-aware device's acquisition. But, L3 tunnels faces a trade-off. It demands a high computational power to maintain network's state and imposes a huge encapsulation overhead.

In summary, VDCs demands a *Virtual Network* (VN) technology that can be scalable enough to overcome the current VLAN limitations, guarantee flexibility, programmability and isolation between tenants, eases virtual infrastructure migration/deployment, diminishes protocol overhead, while still uses legacy network switches and so preserving *Capital Expenditure* (CAPEX) and *Operational Expenditure* (OPEX).

1.1 Proposal and motivations

With all aforementioned problems in mind to deal with, this work introduces **HotOM**[1]: a proposal that leverages network virtualization and programmability on datacenter's network in a straightforward and pragmatic way. Its main objective is fairly simple: *achieve high number of tenants being hosted and better network resource's utilization while allowing the use of legacy network core's devices, improving flexibility and diminishing costs, thus maximizing datacenter's profitability.*

Profitability is the most important facet to maintain the datacenter's stand in a long run. In fact, it can be defined as an "umbrella" aspect(EBERT; GRIFFIN, 2014): any other are beneath, and depends on, profitability. Profits can be simple defined as the difference between revenues and costs, and profitability is the ability to earn a profit. Simply put, the less costs a datacenter has, the more profitable it is.

Datacenter's owner company might try to maximize revenues and minimze costs. A path to obtain the former is increasing the level of resource's utilization, by expanding the number of tenants that are hosted simultaneously in the infrastructure. The latter, in sequence, can be accomplished in many ways, were employing simpler and less expensive network forwarding devices must be seriously considered. Network devices costs about 15% of the total CAPEX of a new deployed datacenter(GREENBERG et al., 2008). HotOM, then, was designed to reliably provide network multi-tenancy along with support legacy physical infrastructure.

In a glance, HotOM redefines the purpose of a L2 field, employs address translation mechanisms and introduces a new L2.5 protocol header to implement network virtualization by leveraging SDN on datacenters.

At the time of this writing, HotOM is a *prototype*, designed to prove its usefulness in terms of VN *instantiation*, *Virtual Machine* (VM) *connectivity* and *network isolation*. It uses OpenFlow as its enabling

[1] There is a popular saying in Brazil related to **HotO**at**M**eal, alluding that complex problems should be attacked first by the edges.

technology but, looking ahead to Chapter 4, Section 4.6, it does not meet all HotOM's requirements. To circumvent this OpenFlow's deficit, a non-performance prone design was taken, introducing throughput penalty. This issue is discussed later on, as well as some feasible solutions.

1.2 Organization

This work is organized as follows. Chapter 2 discuss the theoretical foundations. Chapter 3 explains some of the "bleeding-edge" proposals in datacenter network virtualization arena. Chapter 4 deeply discuss the proposal. Chapter 5 shows the testbed and experimental analyses of a initial (and current) implementation of HotOM, as well as compares it with the considered proposals. Finally, Chapter 6 concludes the dissertation.

2

Theoretical foundations

This work is supported by many foundations, and this chapter is devoted to discuss them. First, HotOM was designed to be appropriately deployed in a datacenter. So, this facility and their network architecture might be understood.

Into its objective, one of the key HotOM's goal is to advocate the use of legacy *Layer 2* (L2) switches for practical reasons. These reasons can be summarized as *(1)* they might be installed and managed for a long time; *(2)* they are less expensive; *(3)* they protect OPEX and lower CAPEX. So, how these switches work should be known for better understanding how HotOM benefits from them.

Another key goal is to minimize protocol overhead. This fact mitigates the waste of bandwidth for protocol (signaling and addressing). It also contributes for lesser power consumption because less protocol headers are necessary to move the same amount of useful, real application data payload.

Finally, since it is a SDN application, HotOM uses OpenFlow architecture as discussed in Chapter 4.

2.1 Datacenters

Datacenters are buildings that physically house three key class of apparatus: server hosts, storage equipments and network devices. Along with them, additional systems like power distribution and cooling are available. It is worthy to mention that all these systems, but mostly the latter, have attracted so much attention in last years due to their noticeable share in OPEX, that a new kind of research, *green* datacenters(ZHANG; ANSARI, 2012), have emerged.

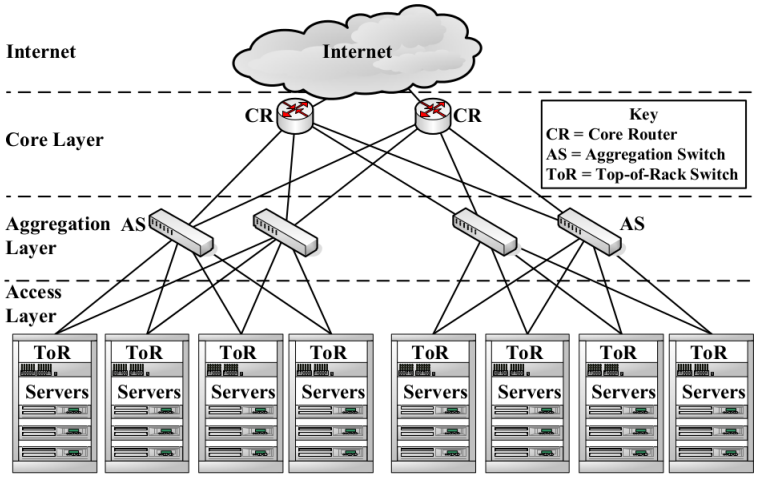
Those types provide, respectively, three basic resources:

- **computing:** comprehends server host's CPU time and memory portions;
- **storage:** composed by whole disks, disk arrays, filesystem extents and disk blocks;
- **network:** comprised by links and bandwidth.

The datacenter network, in particular, is the infrastructure in charge of providing communication between servers, either over protocols at L2 by switches or L3 by routers, as well as implementing policies such as firewalling, load-balancing, traffic shaping and so forth. It can be arranged in many topologies, i.e., ways to interconnect networking devices.

Some years ago, a datacenter network topology was proposed by a well known network player(Cisco Systems, Inc., 2004), seen in Figure 2.1. This topology was thereafter named as *traditional datacenter network topology*. It is composed by racks with server hosts or storages equipments connected to a local (intra-rack) switch called *Top-of-Rack* (ToR), which provides the access layer. Its names resembles its position within the rack: it is usually placed on top of the rack. In turn, ToRs are connected to a smaller number of switches called *aggregation layer switches*. It is aggregation switches' responsibility to provide routing service between L2 switching from ToR to L3 subnets toward *core layer switches*. This last layer (core switches) allows traffic interchange between sets of racks and the Internet.

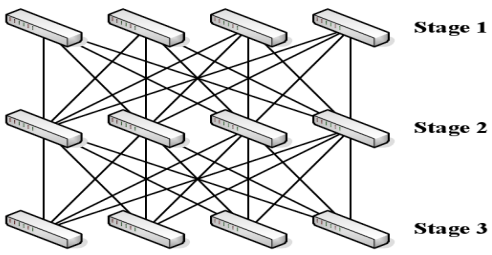
Figure 2.1: Traditional datacenter network topology



Source: (BARI et al., 2013)

Another common topology used in datacenter networks is the Clos topology(DALLY; TOWLES, 2003). It is arranged in stages of switches where each one in a given stage is fully connected to others in the next stage. Figure 2.2 depicts it. This topology has its roots on early telephony switching networks. Its most interesting characteristic is to be *nonblocking*, which means that it guarantees that an unallocated network ingress point (source switch's port) will always have a path to an unallocated egress point (destination switch's port). In other words, two points willing to communicate would not be put in a hold state (thus, nonblocking).

Figure 2.2: Clos network topology

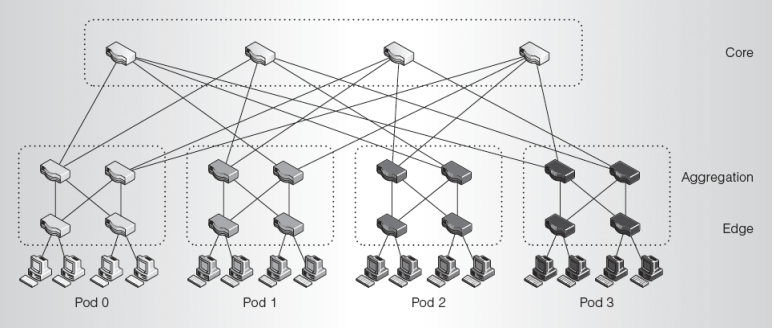


Source: (BARI et al., 2013)

Fat-tree topology(LEISERSON, 1985) is a special case, indeed a subset, of Clos one. As its name suggests, this topology is built in a tree-like structure with layers of switches. The central idea is to provide *Points of Delivery* (Pod), which are tree's leaves where servers are connected to — as seen in Figure 2.3.

Suppose a datacenter deployed with Fat-tree topology that hosts n Pods. Each one are composed by two layers, *edge* and *aggregation*, of $\frac{n}{2}$ switches. The switches within a Pod have n ports used, where $\frac{n}{2}$ ports are connected to lower level layer (edge→server host; aggregation→edge) and the same quantity connected to upper layer (edge→aggregation; aggregation→core). Each Pod is connected to every core switches, so there is n of these switches in the infrastructure. With this arrangement, every server host has a highly redundant path to the other one, enabling fault-tolerance and traffic distribution.

Figure 2.3: Fat-tree topology



Source: (VAHDAT et al., 2010)

Topologies are not restricted to those discussed above. They are mentioned here because of their suitability to datacenter environments. Others are available too, like BCube and Flattened Butterfly(ABTS et al., 2013) topologies, to be used in some cases that are out of the scope of this work.

In early days, datacenters have provided resources in a non-favorable way. Services were deployed into production by allocating dedicated servers and disks, then sharing network without proper traffic isolation. As a result, datacenters have terrible server utilization, large disk-stored data fragmentation, difficult to predict network utilization, high power consumption and, thus, huge operational costs. To overcome such drawbacks, with a special attention to lowering the waste of resources, the *machine virtualization* paradigm was adopted, creating a new kind of datacenter: a *Virtualized Datacenter* (VDC).

2.2 Machine virtualization

Machine virtualization has been a *hot-topic* nowadays because it addresses the poor server host utilization when they are supporting applications(GREENBERG et al., 2009). Although it is currently attracting many attention and effort, virtualization is not a new topic: it was first introduced by IBM in the 60's, when developers provided a way to "split" a mainframe hardware into many so-called *Logical Partitions* (LPAR)(SAHOO; MOHAPATRA; LATH, 2010).

As computing resources are provided by server hosts, they are virtualized by a software layer denominated *hypervisor* — also called *Virtual Machine Monitor* (VMM) — that manages CPU time allocation, RAM memory usage and network interface multiplexing. Grouping together these virtual resources creates a VM. Each VM has some identifiers that are unique, like its *Universally Unique Identifier* (UUID) and, normally, *Media Access Control* (MAC) addresses. Traditionally hypervisors implement profiles associated to a VM type, adding flexibility on its provisioning. This means that the datacenter administrator can create VM templates, with the benefit of a easy deployment of multiple complex architectures like, for instance, multi-tier applications(BI et al., 2010).

Some kinds of machine virtualization came up as result of its development. They differ on how a guest *Operating System* (OS), i.e, the OS that is executed inside a VM, interacts with the hypervisor and hardware resources. The most common types are:

- **Full virtualization**: In this type of virtualization, the guest OS runs unmodified over a virtual hardware provided by the hypervisor. Moreover, the hypervisor itself runs as an application on top of host OS that creates a hardware abstraction.
- **Paravirtualization**: Here, the guest OS is modified and is aware that it runs over a hypervisor. In turn, the latter runs directly over the server host's hardware. This type of virtualization has the advantage that both guest and hypervisor cooperate to achieve a better performance and resource's sharing.
- **Container-based virtualization**: In this type, the host OS contains the virtualization functions and the VMs also executes a copy of it. In fact, every OS instance is seen as a VM, even that one running by the server host. Finally, this instance that is executed by the server host is called **root context**.

2.3 Virtualized datacenter

VDC is a emerging class of facilities that leverage the virtualization of almost, or even total, devices that once were physical. Along with machine — or server — virtualization as discussed in Section 2.2, others resources like storage and networking are also being pushed to the virtualization field.

Storage resource's are virtualized by equipments called disk storage or simply, *storage*. They arrange physical disks in arrays called *Redundant Array of Inexpensive Disks* (RAID). As it name suggest, RAID has the objective of raising data availability through redundancy. There are many types of it, from no-redundancy to high level of data duplication, hence achieving less net space. The hypervisor within storages creates virtual disks called *Logical Units* (LU), then allows server hosts to access them through many technologies like, for example, iSCSI, Fibre-Channel, AoE or even directly attached.

Network virtualization is leveraged by the virtualization of paths, links, switches, routers, firewalls, load-balancers and so forth. Paths are virtualized mainly by adding a label to packets or flows for identification, thus allowing them to be decoupled from physical topology. Links are virtualized by tunneling, i.e, by encapsulating data link protocol over another data link or routing protocols. *Virtual Switch* (VS) and *Virtual Router* (VR) are achieved mostly by virtualizing forwarding tables of data link layer or routing layer. One interesting thing worth to clarify is that virtual switches and routers can be implemented within physical network devices, but also in server hosts by their hypervisors. This fact increases network flexibility because VMs are direct connected to physical network through a capable, full network stack, VS.

Getting deeper, the virtual topology is deployed by interconnecting many of those virtual switches or routers by virtual links or paths creating a VN. Most of nowadays virtual network proposals for multi-tenant datacenters advocates the use of a mesh of tunnels to create virtual links and upon them construct the VN topology. This technique is called *overlay networks*. From a protocol perspective, overlay networks are a L2 domain built over L3 connections, plus some *overlay header* for VN identification.

This imposes a large protocol overhead, since the entire VM generated frame (with L2, L3, L4 and payload) are encapsulated over *outer headers*, which are overlay header plus L3 plus L2. The three most accepted overlays protocols currently in use are *Stateless Transport Tunneling Protocol* (STT), *Virtual Extensible Local Area Network* (VXLAN) and *Network Virtualization using Generic Routing Encapsulation* (NVGRE). The following subsections describe each of them.

2.3.1 Stateless Transport Tunneling Protocol - STT

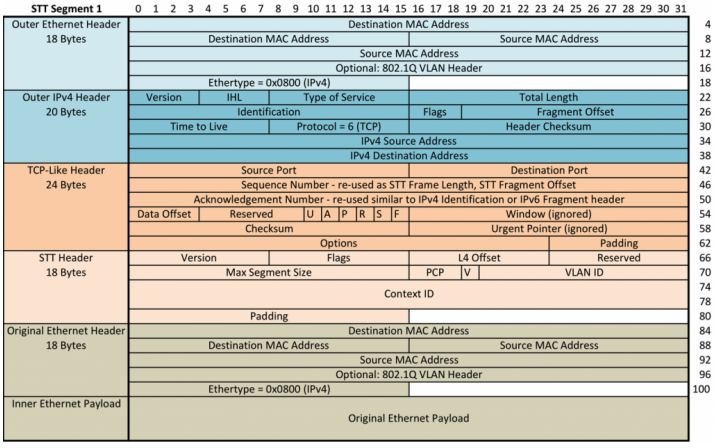
The rationale behind STT is to take advantage of the *TCP Segmentation Offload* (TSO), on the sending end of the tunnel, and *Large Receive Offload* (LRO), on the receiving side of the tunnel. These two offloading techniques are provided by some high-end Ethernet NICs and they release host's CPU to do any other task it needs to.

Tunnels are over L3 (IP) and Ethernet NICs do not support offloading in the presence of any IP encapsulation in the packet. In fact, the hypervisor and its VS are in charge of getting a large frame from a VM, breaking it in nearly MTU-sized payloads and then encapsulating the pieces over IP before forwarding throughout the tunnel.

But, there is a way to accomplish offloading by a class of Ethernet NICs that support TSO and LRO. This two offloading techniques demand a traditional *Layer 4* (L4) TCP protocol header to be able to get a large payload (about 64Kb) from a VM and then chop it in MTU-sized ones. So, STT is a standard, but *fake*, TCP header just to enable these offloading process. Figure 2.4 depicts the STT header.

Despite the fact that STT packets resembles standard TCP packets, STT protocol is stateless and it does not demands three-way handshake between tunnel's endpoints. For so, *Sequence Number* (SEQ) and *Acknowledgment Number* (ACK) fields were re-purposed in such a way that they do not confuse Ethernet NICs that expects them to be used in TSO and LRO. The ACK field is used as a packet identifier due the fragmentation process - it must be constant for all STT packets belonging to the same original frame. In turn, the SEQ field was splitted in two subfields. Its first 16 bits carries the length of the encapsulated frame in bytes, while the latter 16 bits carries the offset, in bytes, of the current fragment within the original large frame.

Figure 2.4: STT header



Source: <http://www.plexxi.com/2014/01/stateless-transport-tunneling-stt-meets-network/>

TCP's source and destination ports were re-purposed too. Destination port was requested for IANA to be in user range, between 1024 and 49151. It will be used to identify, from a network debugging software/appliances for instance, that a given traffic is using STT. Source port is used in such a way that to each flow is assigned one particular value, mainly picked up from ephemeral port range. This fact enables the use of *Equal-Cost Multipath Protocol* (ECMP) by the network core routers, because a path is chosen by a hashing calculation over destination's address, source's and destination's port.

Finally, STT provides a way to identify to whom the flow belongs to. Instead of explicitly determinating the belonging tenant, as some other protocol does, STT uses a generic Context ID field to distinguish a more general entity. This entity can be a tenant, a network in a tenant (in a multi-tier deployment), a single point-to-point communication and so forth.

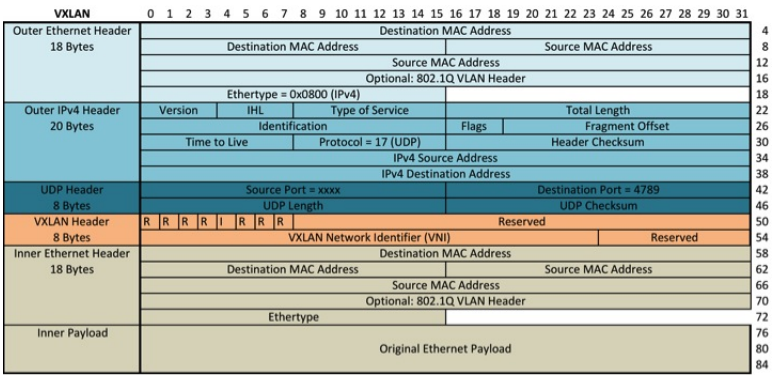
2.3.2 Virtual Extensible Local Area Network - VXLAN

Another tunneling protocol is the *Virtual Extensible Local Area Network* (VXLAN). It was conceived to ease overlay networks deployment and to overcome VLAN and *Spanning Tree Protocol* (STP) scale limitations.

In VXLAN, the overlay tunnels are stateless and established between each VXLAN-capable switch, no matter if it is physical or virtual. However, the most common scenario is using virtual switches run by the host's hypervisor. These switches are called *VXLAN Tunnel End Point* (VTEP). VTEP has the duty of entire VM's frame encapsulation and of choosing which tunnel and destination VTEP to send the VXLAN packet. VM's are not aware that an encapsulation process is taking place. The main VXLAN header field is the *VXLAN Network Identifier* (VNI), which is in charge of identifying each virtual network - also called *VXLAN segment* in VXLAN terminology. This field is 24-bit long, allowing 16.8M virtual networks to be instantiated. Traditionally, there is a one-to-one mapping between tenants and VXLAN segments. Figure 2.5 depicts VXLAN header.

When a VM sends a packet to another, the VTEP then finds which VNI the origin VM is associated to. After that, it determines if the destination VM is on the same virtual network and if its destination VTEP is known. If so, a outer header is added to the original frame before it is being sent to the network. This outer header comprehends an outer Ethernet header, with both origin and destination VTEP's MAC addresses, an outer IP header, also with both origin and destination VTEP's IP addresses,

Figure 2.5: VXLAN header



Source: <http://www.plexxi.com/2014/01/overlay-entropy/>

an outer UDP header and then the VXLAN header (Figure 2.5).

Once the encapsulated VM's frame arrives at the destination VTEP, it *learns* the mapping from inner source MAC (VM MAC) to outer source IP address. In other words, the destination learns on which VTEP the origin VM is reachable and so it caches this information for future use. But, to send a frame to an unknown destination, VXLAN uses other procedures to determine its location, such as being programmed by an external controller or using multicast as an auxiliary mechanism.

VXLAN uses multicast groups to implement both broadcast and multicast services to overlay virtual network. The basics is by associating a multicast group address to a VNI. This association is specially useful when a VM issues an *Address Resolution Protocol* (ARP) request. After figuring out which VNI the VM belongs to, if the origin VTEP also does not have the VM's MAC↔VTEP's IP mapping, it will encapsulate the ARP packet with an outer header, where the destination IP is the musticast group IP address associated to the VNI. Using such way, all VTEPs that terminates a given VNI will receive the ARP within a multicast packet. In addition, these VTEPs will learn the origin VM position.

Finally, in terms of physical infrastructure requirements, VXLAN demands two properties: *(1)* multicast support, due the mechanism above described, and *(2)* VTEPs must not fragment VXLAN packets. If an intermediate router fragments them, destination VTEP may silently discard these fragments, leading for a retransmission due to timeout. To avoid such situation, VM's frame must be reduced to be accommodated in a frame with depicted outer header.

2.3.3 Network Virtualization using Generic Routing Encapsulation - NVGRE

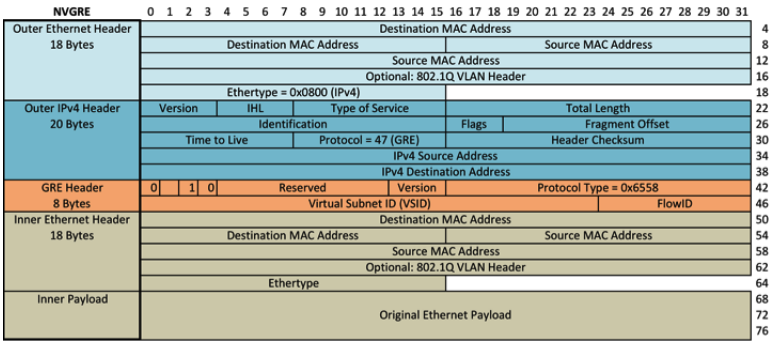
The last tunneling protocol widely in use is the NVGRE. Its objective, similar to others tunneling protocols visited above, is to provide a seamless path for enterprises to expand their VNs hosting capabilities towards multi-tenancy and cloud networks.

NVGRE uses *Generic Routing Encapsulation* (GRE)(FARINACCI et al., 2000) protocol to add a *Virtual Subnet Identifier* (VSID) to VM's traffic. Simply put, NVGRE provides an association between every virtual L2 network and a VSID, which is a 24-bit long field. The tenant↔VSID mapping is outside NVGRE scope and traditionally the management plane is in charge of it. Due to the length, VSID can address up to 16.8 M virtual networks.

Similar to VXLAN, NVGRE tunnels also terminate in an endpoint called *Network Virtualization Edges* (NVE). It also can be a physical host or switch, but traditionally it is a virtual switch within host's hypervisor. Again, VMs are totally unaware about encapsulation/decapsulation performed by NVE. In addition to these functions, NVE can participate in routing acting as a gateway in the virtual topology as well. During the encapsulation, origin NVE needs information about to which destination NVE the target VM is connected to. This information is provided by a control plane or learned from traffic itself.

The NVGRE frame format is depicted in Figure 2.6. It is basically an original VM's frame encapsulated in an outer Ethernet header, an outer IP header, both addressing origin and destination NVEs, and a specially crafted GRE header. This GRE header has the *Protocol Type* field set to `0x6558`, the *Checksum Present* and *Sequence Number Present* bits set to 0 and the *Key Present* field, which is 32-bit long, re-purposed and transformed in two fields (*i*) VSID field, being 24-bit long and (*ii*) Flow ID, being 8-bit long. The Flow ID field must have the same value for packets from the same VSID and same flow. It is used to provide *entropy* for flows. This means that different flows in the same VSID have different Flow

Figure 2.6: NVGRE header



Source: <http://www.plexxi.com/2014/01/overlay-entropy/>

ID values, and this data can be used by NVGRE-aware switches/routers to choose different paths to them. Such mechanism creates a multipath environment, increases throughput and fault-tolerance within the physical network.

NVGRE also provides an additional way to support multipathing. It is allowed to a given NVE to have more than one IP address associated to it. So, the origin VTEP can also spread traffic for multiple paths within the physical network by calculating some flow-based hash to determine which VTEP's IP to send frames to.

Also similar to VXLAN, it is anticipated in NVGRE protocol the use of multicast of physical network to support broadcast and multicast on virtual network, by tying up a given VSID to a multicast group IP address. Alternatively, both broadcast and multicast can be supported by using N-Way unicast, where the origin NVE sends one encapsulated frame to every destination NVE that would receive the broadcast or multicast packet.

Regarding IP fragmentation, neither origin NVGRE neither network devices that forwards/routes NVGRE tunnels must apply fragmentation. Destination NVGRE may discard fragmented NVGRE packets. So, VM's frames MTU must be reduced to be accommodated within a physical network compatible MTU size.

Finally, since GRE is also used in a number of VPN implementations, NVGRE takes advantage of it and leverages VPN's usage to allow secure traffic within the same virtual network that are splited in between two geographically separated datacenters.

2.3.4 Considerations on overlay networks

Overlay networks have the advantage on entirely decoupling the physical network topology to the virtual network one. This decoupling is accomplished by "simulating" links, so making them *virtual*, over IP connections (tunnels) between every server hosts or endpoints. Doing so, the L2 virtual network can be deployed in a managed environment, with isolation and performance guarantees.

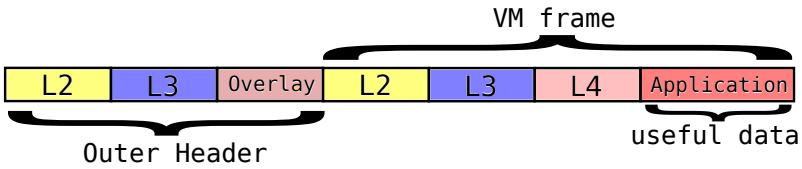
But, that advantage has to face two main problems. The first problem is the amount of computational resources that must be employed to compute the state of every virtual link. Initial researches that came up recently endorse this need by demonstrating that to calculate and program the entire network takes hours(KOPONEN et al., 2014) to create a mesh of virtual links.

The second main problem is the protocol overhead. Overlay networks impose a huge overhead

by encapsulating the entire VM frame into an outer L2 and L3 headers, as depicted in Figure 2.7. This shrinks the VM application payload, i.e. useful data, and augments fragmentation. Recent researches point out that so much overhead is not desirable in production environments(GUENENDER et al., 2015).

In summary, although overlay networks are being adopted by many network players today, they open a wide area of research and experimentation on how to minimize such overhead.

Figure 2.7: Overlay header overhead



2.4 Switches and forwarding tables

Datacenter networking generally relies on switches for data transportation. L3 routing services are optionally used, depending on network topology and scale. But, looking ahead to Chapter 4, this section discuss how switches internally behaves, because it is a key aspect on how this proposal works on forwarding frames.

Switches forwards Ethernet frames based on both destination MAC and 802.1q(IEEE Computer Society, 2014a) VLAN tags. In other to decide to which port a given frame must be forwarded, a switch maintains a *forwarding table* associating MAC address(es), VLAN tag(s) and output port.

For a better clarification, supposing a switch is operational in a VLAN network its main actions in presence of traffic are:

1. A frame arrives through a given port;
2. The switch does a lookup into its forwarding table to realize if the destination MAC is known and through which port it is reachable;
3. If the lookup misses, i.e, the destination MAC is not found, the switch stores in the forwarding table an association between VLAN tag, source MAC address and the arriving port, then copies the frame and floods it out to every port. In other words, the switch *learns* through which port to reach the source MAC address and *spreads out* the frame throughout all its ports, hoping to reach destination.
4. If the lookup hits, i.e. MAC is found, the switch just forwards the frame through the associated port.

Table 2.1 shows a forwarding table excerpt that might be stored in a switch. Indeed, VLAN tag and MAC address are grouped together in a tuple that is used as key in table lookup when searching for the associated output port.

There are some exceptions to the above mechanism. Broadcasting, for example, is done when frame's destination address is `FF:FF:FF:FF:FF:FF` (all bits set to 1). This destination address signals

Table 2.1: A forwarding table excerpt

tuple=(tag,MAC)	output port
(256,AA:BB:CC:DD:EE:FF)	8
(256,00:00:00:01:02:03)	10
(256,00:A2:D3:FD:00:A3)	10
(128,00:11:22:33:44:55)	5
(128,56:78:90:12:34:AA)	6
(100,AA:BB:CC:DD:EE:FF)	10

the switch not to do a lookup in forwarding table, but just to copy and flood out the frame to every port. However the switch still learns the source MAC address, i.e., it stores this address and tag along with the ingress port in forwarding table. This feature is very useful when network management wants to *teach* the switch how to reach a particular address.

The forwarding table is stored within the switch's hardware in a *Content Addressable Memory* (CAM). The CAM is a high-speed, specialized and very expensive memory that matches an input and points an output typically in one clock cycle. Due its price and rareness, it is small in size, often storing about 8K tuple entries. If a network has a large number of different MAC address frames, the switches will have to constantly evict some of those addresses to make room to new ones, increasing both in-hardware removing-inserting-lookup delays and in-network floods. So, it is highly desired to use as less forwarding table as possible in order to avoid *pressure* over it. In the current context, pressure over CAM memory means filling out the forwarding table and imposing switch to constantly do removing-inserting-lookup procedures.

Those physical switches are still widely in use today, even in modern datacenters. But their behavior, dictated by the control plane, are somewhat ossified. There are just little configurations that modify their actions. For example, one can configure a port to be part of a given VLAN, make it forward tagged or untagged frames, join ports together in a group to use *link aggregation*(IEEE Computer Society, 2014b), and so forth. But, the learning and forwarding process, in essence, cannot be modified. Further, new virtualized datacenter demands needs for flexibility. To circumvent this hardness problem, the concept of *Software-Defined Networking* (SDN) was introduced.

2.5 Software-Defined Networking

In the last decades, computer networks have evolved slowly when compared with other areas of computer science. Networks involve using many different kinds of devices like switches, routers, gateways, firewalls, load-balancers, intrusion-detection systems, and so forth, that are interconnected together. Connecting this myriad of options, vendors and technologies is troublesome. So, that slowness can be better understood by how networks and its devices are developed.

Network devices are commonly provided by different vendors, each one with its own proprietary and closed controlling software (operating system) that implements network protocols. It is almost impossible to anyone to modify these device's behavior in a way other than with little configuration options, because they are vertically integrated: *specialized* network applications that runs over an *specialized*

operating system that, in turn, manages a *specialized* hardware. Moreover, these network protocols takes years to be proposed, evaluated, standardized and to have their interoperability tested.

Central and easy management is another problem. Each device's operating system has its own way to configure, to exposes its own interfaces, *Application Programming Interface*s (APIs) (if any), command set and *Management Information Base*s (MIBs). These facts difficult the deployment and operation of a centralized network management tool.

All those facts used to slow down network innovation. To circumvent them, SDN was introduced by changing the way networks are designed and managed.

SDN is defined by two simple characteristics. First, the *control plane* is separated from the *data plane*. The former is in charge of deciding how to handle a traffic, while the latter properly forwards the traffic according to decisions from the former. Second, the control plane is centralized in a single software that rules one or more data planes. This centralization has a range, i.e., its locality can be just one data plane, a local network, a campus network, a multi-datacenter network and so forth, depending on network architecture. Moreover, the centralized control plane software can indeed be a distributed system. This fact is not controversial because the *centralized* aspect of the control plane actually means that it is not tight coupled to the data plane within a network device.

To enable programmability of the control plane, one of the bigger SDN advantage, a software service and framework called *controller* exposes an API that makes possible to dictates the network behavior programmatically. So, this new paradigm allows the opportunity of developing, testing, marketing and deploying new applications from many software vendors, breaking the vertical rigidity of traditional network devices.

In the pragmatical sense, SDN technologies also defines a communication protocol between the control and data planes. The events and messages specified by the SDN protocol are unveiled to application through the controller's API. This is the way that the network application performs direct control over data plane elements (programmable switches and routers). The most proeminent SDN protocol in use nowadays, being supported by an active community and many well established vendors (HP, Dell, NEC, Juniper, etc) is the OpenFlow protocol.

2.6 OpenFlow

To fulfill the need of a pragmatic SDN architecture, OpenFlow was proposed. It was conceived and introduced by the Clean Slate Program at Stanford University. It emerged as an evolution of Ethane (CASADO et al., 2007) work, which came up with a centralized way to apply security network policies through data plane dynamic adaptation, feature that resembles programmability.

Since its inception, around year 2008, OpenFlow has attracted attention from a large number of universities, research groups, users and vendors. Many of the latter have released firmwares that turns their physical switches compatible with OpenFlow. Due to OpenFlow success, a non-profit organization, *Open Networking Foundation* (ONF), was established to guide its development, along with others relevant effort towards SDN. The first standardization, version 1.0, came up in late 2009. The latest available at the time of this writing is version 1.5.1.

The OpenFlow architecture is composed by centralized controllers, programmable data planes

— commonly called *OpenFlow-enabled switches* —, and a communication protocol between them. The former controls the latter through a secure TCP connection where *OpenFlow protocol* messages are exchanged. There are many controller implementations, written in several programming languages. Pox[1], written in Python, Nox[2], written in C++ and OpenDaylight[3], written in Java, are examples of the most known controllers.

On the other hand, there are a few OpenFlow-enabled switches implementations. Production-ready distribution code is barely limited to Open vSwitch(PFAFF et al., 2009). Others like OpenFlow 1.3 Software Switch(CPqD; FERNANDES, 2015) and Indigo Virtual Switch(Project Floolight, 2015) are examples of switches that are not well suited for real world network operations.

2.6.1 Flow tables

The core of an OpenFlow-enabled switch, hereafter in this section just referred as "switch", is its *flow table*. A flow table is a set of *flow entries*. The way these switches work is basically by comparing protocol headers against flow entries and, in the case of a match, an associated action is performed over the packet or flow.

Each flow entry encompass three section (Table 2.2):

Table 2.2: Flow entry representation

Header Fields	Counters	Action Set

- **header fields**: Protocol header fields from ingress packets;
- **activity counters**: Per-flow counters, tracking received packets and received bytes that matches the entry, along with duration in seconds plus nanoseconds from entry insertion into table;
- **actions**: a set (list) of actions to apply to matching packets.

The matching header fields section of a flow entry consists in a static list of protocol headers values from L2 to L4, where the packet is compared against. Each value can be specific, for an equality match, can be a subnet mask, for partial match (for example, *long-prefix matching* on L3 headers), or ANY, which matches any value. In the OpenFlow version 1.0 which was used on this work, the matching protocol headers are shown in Table 2.3.

The counters section is quite straightforward. It stores the quantity of matched packets and the sum of their bytes. As seen in Table 2.4, it also stores the time in seconds and nanoseconds since the flow entry's insertion.

OpenFlow also defines per-table, per-port and per-queue counters. They are mostly used for management and statistical reasons. On hardware switches, per-port and per-queue counters might be maintained by theirs PHY[4] chipsets.

[1] <http://www.noxrepo.org/pox/about-pox/>
[2] <http://www.noxrepo.org/support/about-nox/>
[3] <http://www.opendaylight.org/>
[4] Ethernet physical transceivers

Table 2.3: Matching protocol header fields

Fields	Description	Bits length
Ingress port	Ingress switch's port number starting from 1	depends
Ether src	Source MAC address	48
Ether dst	Destination MAC address	48
Ether type	Ethernet payload type	16
VLAN id	VLAN tag ID	12
VLAN priority	VLAN PCP	3
IP src	Source IP address	32
IP dst	Destination IP address	32
IP proto	IP protocol field	32
IP ToS	IP type of service	6
Transp. src port	Source port from transport protocol or ICMP type	16
Transp. dst port	Destination port from transport protocol or ICMP code	16

Table 2.4: Per-flow counters

Counters	Description	Bits length
Matched packets	Number of packets that matched the flow entry	64
Matched bytes	Sum of matched packet's length	64
Duration seconds	Number of seconds since entry insertion	32
Duration nanoseconds	Number of nanoseconds since entry insertion	32

The last flow entry section is a list of zero or more actions that imposes how switch must handle the matched packets. Actions must be performed in the same order that they were specified. If the set of actions is empty, i.e., a flow entry without an action, the flow's packets must be dropped. OpenFlow defines some *required* actions, that every switch must support, and some *optional* actions, that, as its name suggests, are highly desirable but not enforced to support. Further, if a controller tries to add an unsupported action, the switch should immediately return an *unsupported flow error*. Table 2.5 summarizes the defined actions.

Table 2.5: Actions

Action	Description	Type
Forward	Forward the flow to port(s)	Required
Drop	Flow must be dropped	Required
Enqueue	Send the flow to a queue associated to a port	Optional
Modify-Field	Modify values in protocol header(s)	Optional

The *forward* action needs to specify to which port the flow must be delivered to. There are physical and virtual ports. The physical port is specified by its number, beginning from 1. In turn, virtual ports are specified by their labels. Some of this kind of ports are required, some are optional. Table 2.6 lists the defined virtual ports.

Table 2.6: Virtual ports

Label	Description	Type
ALL	Packets are sent out to all interfaces, excluding incoming one	Required
CONTROLLER	Packets are diverted to the controller	Required
LOCAL	Packets are sent to switch's local networking stack	Required
TABLE	On a packet-out message, perform actions in flow table	Required
IN_PORT	Packets are sent out through incoming interface	Required
NORMAL	Packets are processed using the traditional forwarding path	Optional
FLOOD	Flood the packet respecting the minimum spanning tree	Optional

2.6.2 *In-wire* protocol

The OpenFlow specifies a protocol that is used *in-wire* through the secure TCP connection and allows the communication between controller and switch. In fact, this connection splits apart the control plane and data plane, enabling the latter to be ruled by the former.

Three types of messages are defined: *symmetric*, *asynchronous* and *controller-to-switch* messages. Each message type has its own multiple sub-types that will be discussed later on.

Symmetric messages are sent without previous solicitation from switch to controller and vice-versa. Its sub-types are:

- **Hello**: Message exchange when the secure TCP connection is established;
- **Echo**: Primary used to test if the controller↔switch connection is alive. One side issues an Echo request and the other side must answer with an Echo reply. It can also be used to measure secure TCP connection's latency and bandwidth;
- **Vendor**: Used to offer vendor specific functionalities to the other end. Not well defined yet.

Asynchronous messages are sent by the switch without the controller asking for them. It is used to notify the controller about some kind of event, like a packet arrival, a state's change or even an error. Its sub-types are:

- **Packet-In**: Notify the controller about a packet that have not matched to any flow entry *or* have been matched and forwarded to **CONTROLLER** virtual port. Part of the packet (default 128 bytes) is added to message body, allowing controller to decide what to do;
- **Flow-Removed**: When a flow entry is installed, two timeouts are set: idle and hard timeout. When any timeout is reached, the flow entry is removed and controller is notified through this message;
- **Port-Status**: Whenever a port state changes, like set to down or up state or even blocked by spanning tree protocol, the controller is notified through this message;
- **Error**: Notifies the controller when in the existence of an error.

Finally, the last defined message is the controller-to-switch. It is used when the controller wants to query about switch's features, to insert flow entries in flow table - thus *programming* switch's behavior -, to demand switch to send a packet out and so forth. Its sub-types are:

- **Features**: After the secure TCP connection establishment and the first **Hello** message exchanged, the controller issues this message to be aware about switch's capabilities;
- **Configuration**: Allows the controller to query and/or set configuration in the switch, depending on its capabilities. The switch only answer this message if it is a query;
- **Modify-State**: Allow the controller to manage switch's state, which comprehends adding, removing and modifying flow entries in flow table, as well as setting port properties (bandwidth, for example);
- **Send-Packet**: An arbitrary packet created within the controller can be sent out through a port to the network by this message;
- **Barrier**: This message instructs switch to completely process all previous messages before try to process newer ones.

With those three types of messages and their sub-types, the controller receives all information about switches' capabilities, configuration and state and, programmatically using useful algorithms, creates a logical representation of the network. Then, it can tracks state's changes and dynamically modify switches' behavior by inserting flow entries in their flow tables. These mechanisms breaks the hardness of traditional control planes available within physical legacy network devices.

2.7 Chapter summary

This chapter discussed the theoretical foundations that are the basis for the operation of HotOM. As it was designed to fit into a datacenter network, the most used types of datacenter network topologies were examined.

Moreover, overlay networks are being well accepted by vendors and researches, but they have the inconvenience of demanding too much computing power and employing too much overhead. The most proeminent overlay network protocols were presented along with a discussion on how they waste payload area to protocol itself.

How switches operates, mainly due to their forwarding table related activity and learning process were exposed as a ground to demonstrate how HotOM chooses paths for data movement within the physical network.

Finally, Software-Defined Networking and OpenFlow were introduced, because they are a key enabling technology for HotOM.

3
Related work

This chapter is dedicated to review the latest researches in datacenter network virtualization field. Although network virtualization is not really dependent upon SDN, *i.e.*., the former can happens without the latter, its implementation using such new approach is definitively easier, with the advantage of enabling network programmability.

Some of the researches discussed here are not based directly on SDN, but since they are successful network virtualization implementations, they provide a great comparison standard to the proposal of this thesis.

3.1 Trellis

Trellis(BHATIA et al., 2008) is a software platform for hosting either virtual networks and virtual machines on shared commodity network and server host's hardware. It starts from a predefined set of properties and requirements to define which technologies to use. On one hand, Trellis synthesizes two container-based machine virtualization technologies named VServer(VServer Project, 2014) and NetNS(BIEDERMAN, 2007), and on the other hand it uses *Ethernet over Generic Routing Encapsulation* (EGRE) as a tunneling mechanism. In some sense Trellis goes beyond HotOM, because it embraces virtual machines as well. However, it is its network virtualization mechanism that will be discussed.

3.1.1 Properties

As Trellis proposes a "*network hosting*" platform that can run a number of multiple programmable virtual networks, it aims at some properties to be addressed:

- **Isolation:** For virtual networks mutual interference avoidance, the infrastructure should enforce system resources (*e.g.*, files, CPU, processes) and network resources (*e.g.*, link bandwidth, forwarding tables) isolation.
- **Speed:** Each virtual network should be able to switch and forward packets up to multi-Gigabit speeds.
- **Flexibility:** A tenant or service running over a virtual network should be capable to use a routing protocol or scheme of choice, along with its own application logic.

- **Scalability:** The proposal should be able to support an acceptable number of virtual networks.
- **Low Cost:** Costs is a key factor in network budget. It should be very low, by using commodity servers and network infrastructure hardwares.

Trellis uses and synthesizes existing virtualization technologies - both on virtualizing hosts and network stack - to fulfill the above properties. To do so, it uses a new tunneling protocol (EGRE), a new fast software bridge kernel module, called *shortbridge*, and a known container-based host virtualization implementation: VServer with NetNS.

Using those technologies, Trellis design defines two basic constituent components in its topology as:

- **virtual hosts:** Where high-level services and applications are run and packets are forwarded.
- **virtual links:** Where packets are transported between virtual hosts.

A number of requirements are also defined. These requirements are shown in Section 3.1.2.

3.1.2 Design Requirements

Trellis authors have identified four high-level requirements that the proposal should take care of.

First of all, Trellis must connect *virtual hosts* with *virtual links* so the topology can be deployed. This design requirement joins together to two basic constituent components.

Second, Trellis must run on *commodity hardware*. This design requirement is crucial to support its deployability.

Third, a *general purpose operating system* must be run inside the virtual hosts that can support existing routing software, just like XORP(HANDLEY; HODSON; KOHLER, 2003) and Quagga(Quagga Project, 2015), as well as provide an efficient platform for developing new network services.

Fourth and finally, Trellis should be capable of doing *packet forwarding* inside the general purpose operating system kernel. This design requirement due to packet forwarding in userspace introduces significant latency, thus reducing packet forwarding rate.

These four high-level design requirements are the keystones that guided Trellis development.

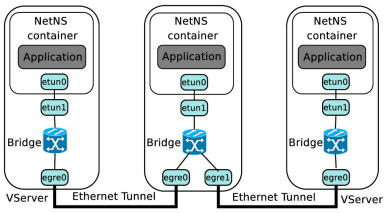
3.1.3 Implementation

Trellis achieves its proposal by synthesizing host and network virtualization techniques in a single and coherent system that satisfies the four high-level design requirements described in section 3.1.2.

In order to address the properties discussed in section 3.1.1 — speed, isolation, flexibility and scalability —, Trellis authors have taken some decisions.

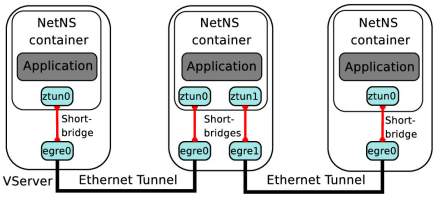
The first decision is using *Container-based virtualization* instead of full virtualization. This choice is based on the fact that the Xen platform, which implements a variant of full virtualization called *paravirtualization*, has performance penalties on network stack, sacrificing the property of speed. Container-based virtualization provides a good level of isolation through namespaces(KERRISK, 2013) (processes, files, memory, network interfaces, network addresses and forwarding tables) and resources

Figure 3.1: Point-to-multipoint links



Source: (BHATIA et al., 2008)

Figure 3.2: Point-to-point links



Source: (BHATIA et al., 2008)

(CPU time, memory and network bandwidth) isolation. The authors also argues that it is more scalable than full virtualization because only a subset of the operating system resources and functions are virtualized. The chosen virtualization technologies for Trellis were VServer along with NetNS.

The second decision is implementing virtual links by sending Ethernet frames over GRE(FARINACCI et al., 2000), i.e. EGRE, tunnels. In order to virtual links be fast, the overhead of transporting a packet through a virtual link must be minimal comparing with the traditional, native network link. The virtual links also must be flexible enough to allow multiple virtual hosts on the same physical network to use overlapping address space as long as to provide support for transporting non-IP packets.

The third decision is to terminate the EGRE tunnel in the root context, not in the virtual host container context. In a container-based virtualization, the root context is the hypervisor. Doing so, Trellis has the ability to impose authoritative restrictions over virtual network resources usage, like bandwidth, scheduling policies and so forth. Trellis also must allow flexibility on topology's configurations by permitting the deployment of point-to-point and point-to-multipoint topologies. It also enables containers running on the same host to be connected directly, rather than being forced to use an EGRE tunnel.

Due to explained third decision, one or more root context network interfaces must be connected with the virtual network interface that lies inside the virtual host container. One way to implement it is with *software bridging*. Using such kind of bridging in Trellis is natural since Linux kernel natively supports it. Just like a traditional Ethernet bridge or *switch*, the software bridge performs a lookup based on the destination MAC address and decides through which port or interface to send the packet.

The fourth decision is to use the native Linux bridge kernel module when and where the network topology requires point-to-multipoint links. Using this kind of links enforces the use of many terminating tunnels network interfaces within the same root context. Some network technologies require a bus-like topology where a set of interfaces might have the behavior of being on the same local area network. In this situation, a multicast or broadcast packet sent from a single interface might reach some or all interfaces on the network. The native Linux bridge module supports such requirements easily.

The Figure 3.1 depicts a topology that uses point-to-multipoint links and thereafter enables the use of broadcasts and multicasts packets. A pair of connected interfaces, `etun`, is created to allow communication between virtual host container and the root context. This pair of interfaces is necessary since the software bridge lies in the root context. The `egre` interfaces are terminating EGRE tunnels.

The fifth decision is to use a new Linux bridge kernel module developed by the Trellis authors, named *shortbridge*, when and where the network topology requires point-to-point links. In such links, the interfaces that lies inside the virtual host container (`ztun`) is directly connected to the `egre` interfaces in the root context, as depicted in Figure 3.2. The native Linux bridge module, used in point-to-multipoint

links, has some performance penalties due to some operations like copying the frame header, learning MAC addresses and doing the MAC address table lookup. Such operations are not necessary when using a point-to-point link. So shortbridge is a special Linux bridge implementation that removes these operations, achieving much higher switching speeds.

3.1.4 Concluding Remarks

By encompassing virtual hosting, Trellis proposes a system that goes somewhat beyond what is normally dealt with by others technologies.

Trellis implements virtual network based on EGRE tunnels. Such tunnels provides great isolation. Its decision to terminate a tunnel within the root context apply the ability of controlling by constraining bandwidth usage.

But there are costs. The first one is a notable level of overhead, since a GRE header is added after the original L2 frame. The second cost is that Trellis imposes the use of expensive L3 switches because tunnels lays over L3 protocols. This somewhat goes against the initial stated property of achieving low cost. Finally, the third cost and more important is the virtual link topology itself. If a virtual network is created using point-to-point links, the number of links (EGRE tunnels) increases by the power of two. Even if point-to-multipoint links are used, the number of links increases notably. So Trellis does not scale very well. The authors states Trellis can create around 60 virtual networks, a low number in datacenter environments.

3.2 VL2

The VL2(GREENBERG et al., 2009) architecture aims to achieve a high resource utilization of a datacenter network. Indeed, datacenter network's CAPEX is gigantic - it can easily drain beyond hundreds of thousands or tens of millions of US dollars. To be cost effective and profitable, such network must allow dynamic resource allocation across large server pools, hold tens to hundreds of thousands of server hosts and allow performance isolation between a large number of services, such as Web searching, E-mail, Map-Reduce computations and utility computing. In the VL2 context, each service is executed in an instance of a virtual network, i.e., each service is a different tenant.

The key aspect of VL2 to achieve high utilization is the property of *agility*, which is defined by its authors as *the capacity to assign any host to any service*. Agility leverages cost savings and improves risk management. With agility, datacenter can meet the variations in demands of individual services from a large shared server pool, which results in higher server utilization. Without agility, each service must previously allocate an enough number of servers to meet the difficulty of predicting failures or demand spikes, lowering datacenter efficiency and increasing costs.

3.2.1 Properties

The starting point of the VL2 architecture is to implement a functionality that gives each service the illusion that all hosts are assigned to it, and they are connected by a single, totally isolated, Ethernet switch. This virtual Ethernet switch was named *Virtual Layer 2* (VL2).

In order to build that completely service-isolated network, VL2 might meet the following three properties:

- **Performance isolation:** Traffic data and patterns of one service might not be affected by the traffic of any other service, just like if each service was connected by a separate physical switch.
- **Uniform high capacity:** The maximum throughput of a server-to-server communication might be limited only by the available capacity of the network adapters on both hosts, as well as assigning them to a service should be independent of network topology.
- **Layer-2 semantics:** Datacenter operator or management software should be able to assign any server to any service, configuring the server with whatever IP address the service expects. Services should be able to migrate to any host while keeping the same IP address along with the traditional network configurations parameters such as netmask, broadcast address, IP gateway address and so forth. Additional features like broadcast should also work as it would if server hosts were connected to a common switch.

VL2 was built to overcome today's datacenter network lack of agility, as defined in section 3.2. First, conventional datacenter architectures rely over a tree-like network configuration(Cisco Systems, Inc., 2007) built using expensive hardware, for example, core and aggregation switches. These architectures do not provide enough capacity between communicating servers due to a link oversubscription on the higher level branches of the tree (branches towards the root). Oversubscription can vary from 1:5, on paths near the access switches, up to 1:240, on paths near to core switches. Furthermore, a conventional network does very little to prevent that a traffic flood in one service affects other service operation, so it's common that services that are sharing the same network sub-tree suffer some collateral effect in the available throughput. Finally, services' migration is normally constrained to a topological location, since the network is traditionally divided in VLANs and IP address space is somehow "location-aware".

3.2.2 Design Requirements

Along with the basic properties discussed in section 3.2.1, the VL2 authors defined some design requirements in other to guide the proposal's development.

First, VL2 uses a flat addressing for each service. This means that a continuous IP address space is assigned to a network no matter where the server is located, allowing the service instances to be placed anywhere in the network.

Second, the service traffic must be spreaded uniformly in the network. For such requirement, VL2 authors have used *Valiant Load Balancing* (VLB)(KODIALAM; LAKSHMAN; SENGUPTA, 2004) for both spreading packets throughout many paths *and* increasing resilience in the case of some network failure.

Third, the end-systems must continue to use its normal address resolution mechanism but without introducing complexity and significant delay to the network control plane.

With these three requirements and using commodity L3 switches arranged in a Clos topology (not in a tree topology) that provides extensive path diversity between servers, VL2 addresses high datacenter network resource usage then lowering costs and increasing operation profitability.

3.2.3 Implementation

VL2 archieves its goal by using a set of technologies and joining them together in a way that can be easily deployed in a datacenter. Its operation is based in (a) spreading out the network traffic, (b) using a shim layer 2.5 in the hosts and (c) dealing with a distributed Directory System in charge of mapping two different types of IP addresses as it will be discussed below.

At least in theory, VLB ensures a *non-interfering* packet switched network if two basically conditions are met: first, the link bandwidth should be greater than traffic load. This means that the communication channel (link) must be adequate to support all traffics sharing the channel. Second, the allocation of resources, i.e. buffers and link bandwidth, must be done in a way that no single flow denies service to another for more than a certain amount of time. Since VL2 deploys a Clos topology where a single host have multiple equal-cost paths to reach the destination, VLB ensures those conditions by randomly spreading traffic across multiple intermediate nodes. VLB per se works in a packet level granularity, so making each packet transverse the network in a different path. But VL2 implements VLB in *flow* level granularity, i.e., to each flow is randomly chosen a different path, but all packets from the same flow uses the same path. This avoids out-of-order delivery and the necessity of packet reordering.

In addition to the randomly spreading traffic, VL2 uses some IP routing and forwarding technologies already present in commodity switches. Those technologies are link-state routing, ECMP, IP anycast and IP multicast. Link-state routing protocol is used to maintain the network topology in switch level terms, but without host's information such as MAC or IP addresses. Doing so the switches are not required to learn and update a huge and frequently changing information.

The IP packet forwarding in VL2 is based on traditional L3 forwarding decision, but the packet itself has some particular characteristics. Two different IP addresses spaces are used.

The first IP address space is called *Location-specific Address* (LA). All interconnected switches — ToR, access, aggregate and core switches — have a LA address assigned to it. Those switches must support link-state routing protocol, such as OSPF, and ECMP. So the entire network topology is known by switches through the former protocol. This feature allows switches to forward host's data encapsulated in a LA address packet along the shortest path to the destination.

The second IP address space is called *Application-specific Address* (AA). Each host associated to a service, i.e. application, has an AA address. It is very common that a service is built on top of a set of hosts which are in the same IP network. In order to create the illusion that they are connected to a single Ethernet switch, VL2 provides a mapping mechanism between the AA and the LA of the ToR switch they are hooked to. This mapping mechanism is provided by the Directory System.

VL2 does data transportation from a host that is assigned an AA IP through a LA-based network by encapsulating AA in LA packets. Prior to transmitting data, the host issues an ARP broadcast performing a destination AA resolution. The VL2 agent running on the server is in charge of listening and answering accordingly such ARP packets. It also queries the Directory System to get the destination server AA IP and its MAC address, where both information allows the agent to create an ARP response.

When sending data, the agent traps the packet and add the ToR LA IP where the destination server is connected to. It does so by querying the Directory System to get the destination ToR LA IP and so do the encapsulation. Finally, the encapsulating LA IP header is referred by VL2 authors as the shim layer 2.5.

If the Directory System refuses to provide an AA-to-LA mapping information to a host, it will not be capable of sending packets through the network. This means that the Directory System can enforce access control and isolation policies.

The Directory System provides three primary functions. The first one is looking up and answering for AA-to-LA mappings. The second function is updating AA-to-LA mappings and the third one is a reactive cache updating, so the latency-sensitive updates (like service live migration) can happen quickly. Its main goal to provide scalability, reliability, and high performance.

The expected lookup and update pattern and workload, that require high throughput and low response time, led VL2 authors to deploy a two-tiered Directory System architecture. On the bottom layer there is a modest number — 50 to 100 servers for each 100K server hosts in datacenter — of read-optimized, replicated *directory servers* that cache AA-to-LA mapping and handle queries from VL2 agents. The upper layer is formed by a small number — 5 to 10 hosts — of asynchronous, *Replicated State Machine* (RSM), write-optimized hosts that offer a strongly consistent AA-to-LA mappings.

Each bottom layer directory server caches the entire set of AA-to-LA mappings stored at upper layer RSM servers and replies to lookups from VL2 agents. As strong consistency is not required, directory server synchronizes its local mappings with RSM servers every half a minute. A VL2 agent spreads lookups to a defined number of directory servers in order to achieve high availability and low latency. If many replies are received, the agent chooses the fastest one and stores the resulting mapping in its cache.

To perform updates, a network provisioning system sends mapping updates to a randomly chosen directory server, which forwards the update to a RSM server. Then, this RSM server reliably replicates it to every RSM server deployed and then replies the directory server with an acknowledgment. As a last job, the directory server forwards acknowledgments back to the originating client.

The reactive cache updating takes place when there is an AA-to-LA mapping inconsistency in directory servers and VL2 agents local mapping cache. A stale mapping needs to be corrected only when it is used to deliver traffic. When such stale mapping is used, some packets arrive to a LA ToR switch which does not connect the destination application (AA) anymore. So the ToR switch forwards a sample of the non-deliverable packet to a directory server, which corrects the stale mapping in the source's cache via a unicast connection.

3.2.4 Concluding Remarks

VL2 allows a large number of services to be hosted in a datacenter without interference between them. It also provides a great scalability and redundancy due to the use of VLB, along with ECMP, to scatter the traffic and achieve reliability. A Directory System provides the logical view of each hosted network, by mapping AA-to-LA, while its agents creates an illusion to the service that it is connected to a large and isolated Ethernet switch.

But VL2 has some drawbacks. The first one is that it is an *application-driven* architecture. It means that each host can run many *applications*, and there is an AA IP assigned to every service in a host. So, if a host is running a large number of applications, an equal number of AA must be assigned to it.

This leads to management hardships. It also promotes a significant packet overhead due to the mechanism of IP-over-IP (AA packet encapsulated in a LA packet), which is not well supported in some commodity switches. The routing protocol must be IP, since VL2 uses ECMP and LA address for forwarding. This fact eliminates the possibility of using non-IP routing or forwarding schemes. Finally, in datacenters nowadays server hosts relies heavily over virtualization, and VL2 architecture does not deal, in a first moment, with virtual servers running on a host.

3.3 PortLand

Portland is a datacenter network proposal which promises to be a "plug-and-play", scalable, fault-tolerant, easily manageable and efficient network fabric. The "plug-and-play" characteristic is defined as the capability of Portland to be deployed without any intervention or configuration on switch or router. In turn, the proposal claims to be efficient in terms of packet switching, routing, broadcasting (by diminishing it) and fast convergence in an event of network failure.

While trends in multi-core processors, end-host virtualization, and high-throughput network devices are suggesting that future datacenters will host millions of VMs, existing L2 and L3 protocols are limited in regards of scalability, difficult management, inflexible communication and limited support for VM's migration. Those facts are addressed by Portland through a structured distribution of L2 addresses in a hierarchical tree-like network topology, a mapping between the original VM's address and the hierarchical one, and a translation mechanism.

3.3.1 Properties

The main property tackled by Portland is scalability. Due to the natural behavior of Ethernet network, i.e., its flat addressing and broadcast dependency, it is not doable for large scale datacenter deployment. Contemporary large scale datacenters can host a server's number that can easily reach thousands of units. Considering that each server hosts many VMs, millions of IP and MAC address would be in use, imposing a huge resource's allocation and overhead by network's devices. Below the Portland's properties summary are described:

- **Scalability:** The datacenter's network must be able to support thousands of server hosts, each one running dozens of VMs.
- **Flexibility:** A VM must be able to migrate from one host to another whithout the need for L3 address reassignment.
- **Speed:** Achieving high throughput in a datacenter's network is essential, so using hardware-based switching capabilities is desirable.

3.3.2 Design Requirements

Large scale datacenters are traditionally built with a predefined and specialized network topology. Among them, Fat-tree is one of the most used on real world deployment. Portland's authors have chosen

that topology and have decided the following requirements, denoted by R*x* - where *x* is the requirement number:

- **R1** *Virtual machine migration:* Any VM may migrate to any server host and this operation should maintain VM's IP addresses, avoidind pre-existing TCP connections and application-level state interruptions.
- **R2** *Zero switch configuration:* Before deployment in datacenter's network, the network engineer should not need to configure any switch.
- **R3** *Efficient communication:* Any server host should efficiently communicate with any other by using any physical path available.
- **R4** *No loops:* No forwarding loops should exist in the network.
- **R5** *Efficient failure detection:* Failure detection should be quick and efficient. Unicast and multicast sessions should proceed unaffected.

3.3.3 Implementation

As the design of Portland aims to deliver a scalable L2 routing, forwarding and addressing for datacenter's networks, all data lookup and actions, switching decisions and translations are done over Ethernet headers. Scalability demands modularity, because such property opens an opportunity for stretching the network. By adopting a tree-structured topology, like Fat-tree, it is possible to apply a network's expansion by adding more "leaves", i.e. rows of server hosts.

Portland Fat-tree topology is depicted in Figure 3.3. At the bottom of the network are placed *Points of Delivery* (Pod). They are a modular set of computing, network, and storage resources available for tenants. Regarding the original Portland paper and this work, the focus is on network resources. The core switches interconnects each Pod downwards by linking one aggregation switch at a time. In turn, aggregation switches connects upwards only half of the core switches. Since each Pod has at least two of those switches, the Pod itself is connected to all core switches. Inside the Pod the aggregation switches are redundantly connected to the edge (access) switches.

The Portland's key aspect is encoding the Pod's position, host's and VM's position as well, in a L2 address called *Pseudo MAC* (PMAC) that are assigned to every connected host. PMACs is a 48-bit value shaped as *pod.position.port.vmid*. *pod* is a 16-bit number that indexes the Pod itself, where 0 is the leftmost one. *position*, a 8-bit value, is the edge switch's position within a given Pod. *port* is the switch's local view of the host, i.e, to which switch's port a host is connected to. It is also a 8-bit value. Finally, *vmid* is used to multiplex various physical hosts on the other side of a bridge or to identify multiple VMs running by the connected host. For each new MAC address observed on a given port, the edge switches assign a different *vmid*.

Portland names the factory assigned host's MAC address as *Actual MAC* (AMAC). The main purpose of edge switches besides switching itself is to map AMAC to PMAC, and vice-versa. Every time an ingress edge switch receives an Ethernet frame never seen before, this frame is sent to the switch's software that acts somewhat like a local control plane. Then the software creates an entry in a local

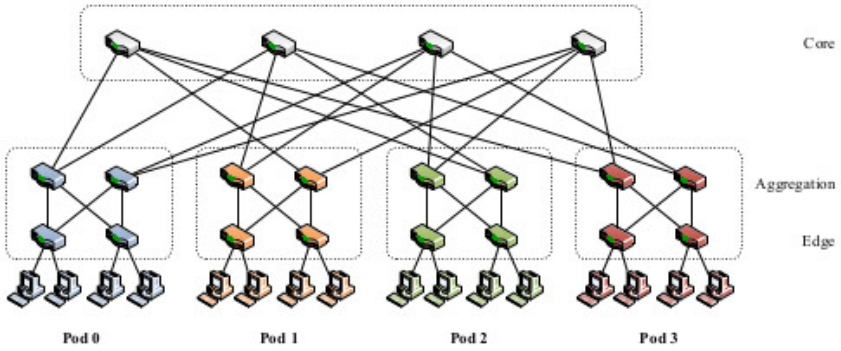
PMAC table mapping the host's AMAC to its PMAC - the IP address is mapped as well. This entry in PMAC table is reported to the Fabric Manager, which is the centralized controller that has the entire datacenter's network view. It also creates a flow table entry rewriting AMAC to PMAC address. Lastly, the edge switch creates the appropriate flow table entry in the inverse direction, i.e., rewriting the PMAC destination address to the AMAC for the traffic destined to the host. So, in essence, the edge switch performs a deterministic AMAC$\longleftrightarrow$PMAC rewriting.

The Fabric Manager is a logically centralized controller that stores, tracks and maintains the network configuration's state and information, such as topology. Being a user-space process running in a dedicated server host, its main purpose is to help edge switches with ARP resolution, fault tolerance and multicast communication. Portland's implementation of Fabric Manager do not impose any particular way on how it is connected to switches. It may run in a separate control network or simply be redundantly connected to a switch. Regarding of how information is tracked, in Portland there are some restrictions of knowledge, limiting it to a soft state. This fact means that it is not demanded by the Fabric Manager a strong consistency, deviating the need for any administrator to configure it with number of switches, their locations, their identifiers, etc. That fact also eliminates a strict consistency among replicas, if Fabric Manager is deployed as a distributed application.

Since Portland creates a L2 domain, it must deal with ARP broadcasts. The Fabric Manager is used to store necessary IP and MAC information for creating ARP replies. When a host issues an ARP request, the edge switches trap and forward it to the Fabric Manager. Then, the Fabric Manager searches its PMAC table trying to find an entry mapping the target IP address to the requested PMAC. If there's such entry, the Fabric Manager returns the PMAC to the edge switch, which in turn creates an ARP reply and returns it to the originating host. A missing entry tells the Fabric Manager that it does not know the PMAC related to the IP address, so it must discover such mapping. Thus, it demands a core switch to issue an ARP request. This ARP request is distributed downwards through the tree until it reaches the edge switch. The target host will reply with its AMAC, and then the ingress switch creates a flow table rewriting PMAC to AMAC addresses, for egress packets, and send back the PMAC to Fabric Manager and originating host. Since PMAC reveals host's position within the topology, there's an efficient forwarding based on parts (*pod*, *position*, *port*, and *vmid* bits) of the destination PMAC address.

One of the main Portland's premises is to make switches' configuration unnecessary. Another objective is that switches' use their position in the topology to perform more efficient forwarding and

Figure 3.3: Portland Fat-tree topology



Source: (MYSORE et al., 2009)

routing based on L2 header (PMAC's). To avoid the work of manually assign a switch position, Portland employs a *Location Discovery Protocol* (LDP), so a switch can automatically find its position. From time to time Portland's switches send a *Location Discovery Message* (LDM) containing its *identifier*, *Pod number*, *position* and *tree level*. The tree level can be 0, for edge, 1 for aggregation, and 2 for core. A key LDP factor is that edge switches just receives LDM packets from part of their ports, since the other part is connected to hosts. So edge switches easily can set their tree level and send LDM messages upwards to aggregation. Aggregation switches receives in part of their ports messages from edge, so they can easily set their level as well. Finally, core switches receive LDM messages on *all* ports, so they can also discover their levels. The position within a Pod is chosen randomically. An edge switch gets a position value and sends LDM messages telling it. If no other switch also claims that position, it is hold. Finally, Portland leverages the Fabric Manager to assign values to Pods.

Once switches establish their position using LDP, they populate their forwarding tables based on neighbors positions. This means that, for example, core switches will forward frames based only on *pod* value within the PMAC address. In the same way, aggregate switches will forward those frames based on *position* value, whereas edge switches will deliver frames based on *port* and *vmid* values. This tree-like forwarding, where each group of bits in PMAC is checked for switching decision downwards guarantees a loop free forwarding.

3.3.4 Concluding Remarks

Portland proposes a scalable, easily manageable, fault-tolerant and efficient datacenter network by leveraging a Fat-tree topology along with a well defined switching mechanism and MAC address translation. The switching mechanism is based upon forwarding on some positional values in a Pseudo MAC header. The MAC address translation is performed by edge switches, mapping the AMAC to PMAC addresses.

The main advantage of Portland is its scalability. A datacenter administrator can easily expand the network by adding more Pods horizontally. A second advantage is its fault-tolerance, by also using LDP to detect dead switches, and its simplicity by using OpenFlow technology to insert flow table's rules that translates AMAC's to PMAC's, casts ARP replies, etc.

But Portland has some problems. First of all, the architecture is host aimed rather than VM aimed. This means that it is much more concerned in interconnecting hosts than VMs. Every given communication example were done upon hosts, not VMs.

Furthermore, another Portland problem is its heavily dependence on OpenFlow hardware switches. Forwarding frames based on pod, position and port values of PMAC is not industry-standard, i.e., long prefix matching is done over L3 addresses, not on L2. By these terms, core and aggregate switches *must* be OpenFlow-enabled, which lowers Portland's feasibility in real-world implementations. Edge switches must be OpenFlow-enabled as well, since it have to communicate with the Fabric Manager, send ARP replies on its behalf, and install flow table entries that translates AMAC's to PMAC's. LDP is not also a standard protocol, forcing even more the usage of OpenFlow-enabled switches in the entire topology.

Finally, Portland authors do not make clear how they create an illusion of a completely separated network to tenants. The author of this work guesses that Portland only forward authorized PMAC's source $\leftrightarrow$ destination frames.

3.4 NetLord

Multi-tenant datacenters usually face many challenges. One of the most important is how to maintain profitability. A natural way to obtain economical achievements during operation is maximizing resource's sharing. Thus, multiplexing is the foundation of reaching an optimal scenario of such sharing. In that sense, the larger the number of tenants and the larger the number of VMs, the higher the chance for multiplexing.

NetLord is a proposal to allow a datacenter to deliver *Infrastructure-as-a-Service* (IaaS). Basically, IaaS explores multiplexing to provide multi-tenancy, along with scale and easy operation. By fully abstracting both L2 and L3 address spaces, NetLord provides to tenants a flexible network virtualization.

To maintain low deployment and operational costs, NetLord was designed to lay over commodity network devices. Nevertheless this feature, an initial setup and configuration is required. NetLord's authors argues that this one-time, off-line configuration does not have significant impact in its deployability.

3.4.1 Properties

The authors of NetLord have defined a small, but highly desirable, set of properties. It primary aims in creating a multi-tenant environment that enables an economical advantage. They are:

- **Scale maintaining low cost:** To reduce cost, the network *must* use commodity switches. This class of switches typically holds a few tens of thousands of L2 forwarding table's entries, which constraints the address' number that can be learned at once. To scale at such situation, VM's MAC addresses must not be exposed to switches.
- **Easy configuration and operation:** A network that is easy to configure and operate reduces costs, by lowering OPEX. Forwarding devices should, as much as possible, be automatically configured and operated. If both these actions could not be avoided, they must be at minimum in network deployment time.
- **Flexible network abstraction:** In a multi-tenant environment, tenants can demand quite different network requirements. For instance, a tenant running Map-Reduce application would need just a set of connected VMs in the same virtual network, while another tenant, say a hosting provider, would deploy a three-tier Web application that needs three different L3 subnets. So, the network infrastructure must be done flexible enough to provide customized services to different tenants.

The elementary mechanism behind NetLord is the encapsulation of tenant's VM L2 frames on another packet, by adding a specially assigned L2 *and* L3 headers. Then, this whole frame is transferred through a scalable network fabric. To provide these header manipulation services, NetLord uses a local agent executed by the hypervisor that encapsulates, routes, decapsulates and delivers frames from and to VMs.

3.4.2 Design requirements

Although datacenter's network are traditionally designed in a structured way, with layers of switches — access, aggregation and core —, NetLord does not narrow down which design to use. It is concerned to provide a scalable, low cost and flexible network architecture. To do so, some design requirements were sketched.

First of all, VM L2 address must be hidden from network forwarding devices. Such hiddenness is the primary key to allow commodity switches to be used, since the number of MAC address exposed to them are just equal the number of server hosts working in the datacenter, lowering pressure on switch's forwarding table. Thus, VM frames must be encapsulated in a L3 packet.

Second, the network topology should be reliable. To construct a multipath, high-bandwidth, resilient network fabric using commodity Ethernet switches, NetLord relies on SPAIN(MUDIGONDA et al., 2010). SPAIN works basically by *(1)* pre-computing k distinct paths between pairs of edge switches; *(2)* pre-allocating VLANs tags to identify these disjoint paths and *(3)* using an end-host agent to spread traffic among paths.

Third, the *outer* L2 and L3 headers, that encapsulate VM frames, must facilitate forwarding. This requirement is accomplished by carrying in L2 headers the source and destination switches' MAC addresses and by placing in the VLAN tag the SPAIN path. In turn, the L3 header carries the tenant ID and destination edge switch egress port, so the core network devices are configured to forward packets based on those crafted headers.

Fourth, the usage of local agents, which in NetLord context is called *NetLord Agent* (NLA), is a must. This agent is in charge of *(1)* VM packet encapsulation as briefly described above; *(2)* routing; *(3)* decapsulation in destination host and *(4)* choose a per-flow VLAN tag to address the SPAIN path to be used in that flow.

Finally, a configuration repository must be present in order to store tenant's network related data. This repository gives a "broad view" of the network. It must be accessible to all NLAs for queries and updates and is used to configure both switches and end-host parameters.

3.4.3 Implementation

To fully implement NetLord, it is necessary to understand how tenants information, ingress source edge switch's MAC address, egress destination edge switch's MAC address and port, and path identifier are encoded in L2 and L3 headers that encapsulates the entire VM's frame. As previously stated, this frame itself isn't modified at all.

NetLord transports data back and forth between VM's *Virtual Interface*s (VIFs). Each VIF is identified by the tuple $(Tenant\_ID, MACASID, MAC\text{-}Address)$. This tuple is stored in the configuration repository along with others information like the egress switch's port the VM is connected to, and which VLAN tag will be used by SPAIN to reach this VIF.

Each tenant hosted in a datacenter must be uniquely identified. For this purpose, NetLord uses the *Tenant_ID* value. It is a 24-bit identifier that allows NetLord to scale up to 2^{24} tenants. Some ID's are reserved for NetLord internal use, as will be described soon.

Since one of the design requirements is to provide a flexible network abstraction, NetLord allows

each tenant to use multiple L2 networks. This means that a tenant can create many L2 networks with overlapping addresses, using a VM as a router or even deploying a virtual router service. To identify each L2 address-spaces that belongs to the same tenant, NetLord uses the *MACASID* value.

Finally, the last value of the tuple is the *MAC-Address*. It is the destination VIF layer 2 address. Since the entire VM's frame is encapsulated, this information is available into the original VM frame.

Every time a frame is originated from a source's VIF to a destination one, the NLA takes place and encapsulates the packet, unless both VMs are executed by the same host. The encapsulating L2 header is built in the following way:

1. **MAC.src**: Equals to source edge switch MAC address;
2. **MAC.dst**: Equals to destination edge switch MAC address;
3. **VLAN.tag**: Equals to chosen SPAIN path based on both destination edge switch and flow.

In turn, the encapsulating L3 header is built as:

1. **IP.src**: Equals to MACASID value;
2. **IP.dst**: Equals to a function `encode()`, that encodes destination edge switch port and Tenant_ID;
3. **IP.id**: Same to VLAN.tag
4. **IP.flags**: Equals to *Don't Fragment*

As can be seen, almost all information are exposed in both headers in order to NLA identify from which tenant the packet belongs to. Tenant_ID is encoded in IP.dst field, the MACASID is available in IP.src and the original frame header, which is encapsulated, carries the MAC address of the destination VIF. The SPAIN path identifier is placed twice. This happens because the destination edge switch will strip the VLAN tag before sending the packet to the destination NLA, but this NLA needs such information due to some SPAIN-related fault-tolerance mechanism.

The most tricky aspect on creating both L2 and L3 headers is the function `encode()`. This function receives as input the egress edge switch port number p and the Tenant_ID *tid* and returns an IP address that is placed on IP.dst. The encoded IP address is in such a form that the port p is a prefix and the Tenant_ID *tid* is a suffix. So, the IP address is returned as p.*tid*[16:23].*tid*[8:15].*tid*[0:7]. Remembering that the egress edge switch is the last hop traversed by the frame, this address encoding allows this switch to use *longest-prefix match* (LPM) on a forwarding table with addresses of the form (p.*.*.*/8 $\rightarrow p$). For instance, if a frame of Tenant_ID 10 is sent to a VIF that is connected to a destination egress switch in port 12, the `encode()` function will return the encoded IP as 12.0.0.10 and, when it reaches the egress switch and the forwarding mechanism matches it to the L3 forwarding table entry `(12.*.*.*/8 → 12)`, this frame will be sent throughout port 12.

As stated before, edge switches needs two one-time configurations defined on deployment and applied whenever the switch boots. The first configuration is the above explained IP forwarding table entries. To be more accurate, the entries are on the form (prefix,port,next-hop) = (*encode()*,p,p.0.0.1) for each switch port. The NLA is the "owner" and *listens*

to the IP *p*.0.0.1. Table 3.1 depicts a forwarding table example for a 48-port edge switch. The second configuration is the VLAN tag for SPAIN compliance. Each switch must forward a VLAN-tagged frame which ID is designed by the NLA during SPAIN multipath computation. Both configurations can be pushed to switches by SNMP, CLI or remote scripting.

The NLA deals with many already discussed informations to provide encapsulation, decapsulation, SPAIN path selection and routing. NetLord's authors assume in their original paper that some of those information are provided by a system that *(1)* manages Tenant_ID allocation to tenants, *(2)* manages VM placement on hosts, *(3)* informs VIF parameters for every VM that is created/booted and *(4)* make this set of information available to hypervisors.

Nevertheless, some of those information are learned from network. When a NLA is started, it listens to the *Link Layer Discovery Protocol* (LLDP) messages sent by the edge switch, which contains switch's MAC address and port number. Thus, the NLA assumes the IP address `encode(`*p*`,1)` = *p*.0.0.1 as its address, and then responds any ARP queries for that address from the local edge switch. By self-assigning this IP address, the NLA takes advantage over the forwarding table entries illustrated in Table 3.1.

NetLord Agents has to communicate with the configuration repository, VM manager system and hypervisors, but the IP address *p*.0.0.1 is local-only (edge switch $\leftrightarrow$ NLA). Therefore, the agent needs a globally-usable address. For that end, NetLord reserves an address space for Tenant_ID=1 where the NLA will broadcast a *Dynamic Host Configuration Protocol* (DHCP) request for the globally-usable IP address.

Since NetLord is transparent to VMs, it would send out packets exactly as they would have in on a traditional Ethernet network. If a packet is an ARP query, the agent captures it and calls a procedure called *NL-ARP subsystem*.

The NLA's NL-ARP subsystem maintains a VIF IP address $\leftrightarrow$ MAC address table entries. It is also in charge of communicating with other NLAs informing that a given entry is in place. To do so, the NL-ARP uses a small protocol with three messages types: `NLA-HERE`, `NLA-NOTHERE` and `NLA-WHERE`.

When a VM starts or migrates, the NLA broadcasts VM's location using a `NLA-HERE` message. The NL-ARP table entry never expires, but broadcast messages can be lost resulting in either missing or stale entries. When a VM issues an ARP query that NLA does not have the accordingly entry, due to a prior missing message, it broadcasts among NLAs using the `NLA-WHERE` message and the target NLA replies with a unicast `NLA-HERE`. If a stale entry causes packet to reach a NLA that does not have the destination VM, the receiving NLA unicast the sender with a unicast `NLA-NOTHERE`, which will

Table 3.1: Last hop forwarding table

prefix	port	next hop
1.*.*.*/8	1	1.0.0.1
2.*.*.*/8	2	2.0.0.1
⋮	⋮	⋮
47.*.*.*/8	47	47.0.0.1
48.*.*.*/8	48	48.0.0.1

force the sender to broadcast a `NLA-WHERE` query. All these messages are exchanged using the reserved Tenant_ID=1.

If the packet sent by a VM is not an ARP query, the Tenant_ID of the receiving VIF is always the same as that of the sender VIF. That is because NetLord does not allows direct traffic between two tenants, providing network isolation. The only exception is via the public address space with Tenant_ID=2. Going further, the NLA needs to determine other two information to complete the tuple: MACASID and MAC-Address.

Traditionally the source VIF MACASID *and* destination VIF MACASID are the same, since the most common communication takes place within the same L2 VN. The destination VIF MAC-Address is found in the L2 header of the encapsulated frame.

NetLord provides a virtual router function between virtual networks from the same tenant, but with different MACASID. A given VIF is allocated as a virtual router. When a packet is MAC-addressed to the virtual router, the NLA gets the destination IP address from the original packet, then using the N-ARP table entries it gets the destination VIF's MACASID and MAC-Address. Along with both information, NLA also gets the destination edge switch's MAC address and port number.

After collecting those required data, no matter if VM is sending a packet to the same MACASID or not, NLA invokes the SPAIN's VLAN selection algorithm to choose the path that best fits to the given flow. At this point, NLA holds all information needed, thus adding a L2 header with a VLAN tag and a L3 header to the encapsulated frame. Then it forwards the whole packet out to the network.

The destination egress switch parses the L2 header on the received frame and recognizes the destination MAC as its own, strips that header and looks up the destination IP address in its IP forwarding table to figure out the destination NLA next-hop information, which gives the port number of the switch and the IP address of the destination NLA. Then, the egress switch forwards the packet to the NLA.

Finally, the receiving NLA strips out the L2 and L3 headers, extracting the Tenant_ID from destination IP address, the MACASID from the IP source an the VLAN tag (SPAIN path) from the IP ID field. Using its local table, the NLA then determines the destination tenant VM's VIF, using the L2 destination address in the inner packet. After all, the NLA delivers the original frame to the destination VM.

3.4.4 Concluding Remarks

NetLord is a very complex system, dealing with lots of information, some of them are exposed by the VM management system, some of them are learned directly from network. One of its the most notable strength is the ability to provide a very flexible network abstraction to tenants. It is possible, for example, to a tenant instantiate, say, three different L2 virtual networks with MAC address overlapping, each virtual network with its own L3 addresses and connect them with virtual routers. This fact provides a huge capability to scale. NetLord can host 2^{24} tenants (Tenant_ID), each one with up to 2^{32} virtual networks (MACASID), each virtual network with up to 2^{48} VM's VIF (MAC-Address).

But those scalable capabilities comes with costs. First of all, there is a huge header overhead. The original VM payload is encapsulated by a L3 and then a L2 header, as normally would. Then, this entire frame is encapsulated inside another L3 and L2 headers. This means that two L2 and two L3 headers are placed before the actual payload.

Other subtle drawback is that the *NetLord Agent* (NLA) always stores tuples (Tenant_ID,MACASID,MAC-Address) for every VM VIF by demand. Part of tuple's information are stored in table entries by the NL-ARP subsystem. That means that those table entries are duplicated in *every* NLA, which tends to be space inefficient.

NetLord relies over SPAIN. This multipath proposal must calculate every multiple path between two edge switches, and allocate VLAN tags to index them. The calculation's algorithm is very expensive. Furthermore, in a event of a topology change, the SPAIN might need to recalculate all over again, which can lead to a time hiatus in forwarding frames.

Edge switches must be L3-aware, since they have to forward the egress packet to the local NLA using a forwarding table as depicted in Table 3.1. This requirement undermines the objective of lowering CAPEX, since this class of switches are more expensive than just L2 ones.

Finally, NetLord does not provide any kind of network programmability, neither by the agent or by exposing an API. This fact limits, mostly denies, its deployment in datacenters that aims to provide a dynamic virtual network.

3.5 CrossRoads

As datacenters companies grow, undoubtedly many facilities alike begin to be built in different geographic locations. In some cases, datacenter complexes are incorporated and merged to a central, probably larger, acquiring company operation. This means that it is necessary to allow communication and traffic interchange between distant datacenters. Moreover, since they have evolved separately, there might be different topologies, different L2 and L3 addresses and routing schemes, heterogeneous platforms, and so on. Amazon, for example, maintains at least six complexes around the globe, located in Palo Alto, Ashburn, Dublin, Tokyo, Singapore and Sydney — probably there are non-disclosed others.

Running applications on VMs is a trend that enterprises cannot deny. Most of them have already heavily adopted virtualization. In regards of geographically dispersed datacenters, VM's migration among them provides necessary operation's flexibility and a path for lowering OPEX costs. In this regards, CrossRoads(MANN et al., 2012) proposes a network fabric focusing on a seamless live and offline VM's migration among multiple datacenters by using SDN and providing agnostics L2 and L3 networks.

3.5.1 Properties

Some CrossRoads' properties are shared with others proposals, some are not. In general, there are foundations for VM's migration - both live and offline between datacenters - without hassles. It means that VM's locally network configuration, i.e. L2 and L3 addresses, must be preserved during migration, no matter how is routing designed on the origin or destination's datacenter network. With these facts in mind, CrossRoads properties are:

- **Layer 2 semantics:** Tenants must be able to assign an IP address and netmask to their VMs as well as other configuration such as gateway and broadcast addresses, for instance. So, a virtualized tenant's network must perform as a traditional L2 network. Furthermore, VMs need to preserve their L3 configuration upon migration between datacenters.

- **East-west communication:** Connectivity across datacenters for east-west communication should be maintained at L2 to avoid issues like having to reconfigure firewalls. East-west traffic is originated from one datacenter to another.
- **North-south communication:** Connectivity with networks and/or clients outside the datacenter must be possible, i.e., traffic from a VM must transverse the datacenter's L3 boundary. This kind of traffic, in CrossRoads context, is called North-South communication.

3.5.2 Design requirements

CrossRoads' design requirements were mainly established to give support to its principal goal: VM mobility. Due to the complexities associated with live and offline migration across subnets, such kind of migration is still limited to within a local network. These complications arises from the hierarchical addressing used by L3 routing protocols. There is a hierarchy breakage when a cross subnet or datacenter's L3 boundary VM's migration takes place. L4 connections, for instance, are interrupted.

Getting into details, the first design requirement is to allow live VM migration. This kind of migration is very desirable because it is a key precondition for server consolidation, which in turn lowers OPEX costs. Associated to server consolidation, the dynamic workload balance among server hosts is crucial since it can spread usage across hosts. This mitigates problems imposed by a VM that has a high-level resource usage during a, say, short period of time. Finally, live migration can provide high availability, fault tolerance and disaster recovery.

Second, CrossRoads must provide off-line VM migration. This migration model is important because it allows an entire multi-tier application running in a given datacenter to be backed up and then restored at another datacenter in an opportune time. Thus, this allows any complex application, the one that runs in many VMs, to be moved to another location when convenient.

Third, the firewall reconfiguration throughout datacenters might be avoided in order to ease network management. The several firewalls, acting at L3, can block communication from a given VM when it migrates to another datacenter. So, it is highly desirable that the traffic between VM placed in different datacenters be handled at L2.

Fourth, a state management must takes place. Commonly, live VM's migration requires that source and destination host must be on the same L2 network. In other words, they must be on the same broadcast domain. Habitually the announcement of the new VM location is done by broadcasting *Gratuitous Address Resolution Protocol* (GARP) or *Reverse Address Resolution Protocol* (RARP) packets. Since these packets are not transmitted through L3 boundaries, a different mechanism that maintains the current location of a VM, no matter in which datacenter runs it, must be used.

Finally, the last design requirement is that CrossRoads might be deployed without demanding modifications in existing datacenter infrastructures. With this requirement, CrossRoads can be adopted without much capital expenses.

3.5.3 Implementation

CrossRoads uses OpenFlow paradigm to deploy every single functionality. Its architecture is arranged in such a way that every datacenter has its own OpenFlow controller, and they know and

communicate with other controllers.

The physical network topology adopted is the traditional one, as depicted in Figure 3.3 where edge switches are in fact ToR switches. In order to interconnect geographically dispersed datacenters through the Internet, CrossRoads uses a special purpose switch called *Encapsulation switches*. Its duty is to "route" frames from one datacenter to another by encapsulating them and, then, decapsulating at the destination datacenter. Unfortunately, CrossRoads' authors do not provide much information about Encapsulation switches. They basically cite its presence in the architecture.

CrossRoads works similarly to Portland and VL2. It also assigns to each VM a PMAC, plus a *Pseudo IP* (PIP). The idea behind them is that they carry information about network positioning within the topology. PMACs are used mainly to assure connectivity across datacenters for a migrated VM. It has a pre-formatted form, where the 48-bit addresses are in the form *dcid.pod.position.port.vmid*. *dcid* is a 8-bit length field and identifies the datacenter. *pod* is 16-bit length and identifies the point of delivery within a datacenter. *position* defines the port number of a ToR switch that connects a host to the aggregation switches. Finally, *vmid* identifies a VM running in a particular host.

PIP also uses fields from an IP header to encode position information of a VM. It is defined as *privateNetworkId.dcid.subnetid.hostid*. To be compliant with RFC 1918(REKHTER et al., 1996), *privateNetworkId* field can be 8, 12 or 16 bit long and is a private network identifier. So, *privateNetworkID* can be "10" for class A private addresses, thus being a 8-bit field, or a range "172.16 - 172.131" for class B private addresses being a 12-bit field and finally it can be "192.168" for class C private addresses, being a 16-bit field. Just like in PMAC, *dcid* is a 8-bit length field to identify which datacenter the VM is placed. The last two fields, *subnetid* and *hostid* design for identify respectively (sub)network and host, can occupy the remaining 16, 12 or 8 bits as required on using CIDR.

PMACs and PIPs assigned to hosts or VMs changes based on their current location, whereas PMACs and PIPs assigned to routers and gateways remains fixed. Within a datacenter, all traffic are forwarded based on their L2 addresses. Eventually a flow has to cross a L3 boundary to reach an outside component. At this point it is routed using PIPs and may again be delivered to the destination using PMACs.

Every time a new ToR switch or router are deployed in the network, the centralized OpenFlow controller figures out if the switch is a ingress switch. If so, the controller installs a rule to redirect all ARP requests to itself. There are two objectives using ARP resolution. First, if a VM will communicate with another in the same L3 subnet, it will query for the destination's L2 address. Second, if there is a cross subnet communication, it will do a resolution for the gateway's L2 address.

When an ARP resolution occur in the same subnet, after crafting a reply packet and replying source VM with it, the controller installs a set of OpenFlow rules in origin and destination switches. The first rule is placed in the destination ToR to translate the PMAC to the actual VM's MAC. The second rule is also placed in destination ToR to replace PIP address with the real IP of destination VM. The third and fourth rules are placed in origin switch, to translate from VM's actual MAC to PMAC and translate from VM's real IP to PIP.

An ARP resolution could also be done to reach another subnet, when demanding routing services. In this case, VM will issue an ARP asking for its *default gateway*. Due to its mapping table, the controller knows that the VM is asking for gateway's MAC address, and so it does a little trick responding back

with a special MAC address. When the first packet reachs the ingress ToR, it sends this packet to the controller which installs a rule in ToR matching packets with the special MAC address and related IP, replacing them with the destination PMAC, as well as replacing the destination real IP with its related PIP. These rules acts as a virtual router and ensures that following packets in a flow will also be matched and translated accordingly.

Chances are that packets reach an access router. If this router realizes that packet's destination is an external datacenter, it does not use L3 address for routing and sends packets directly to the corresponding encapsulation switch based on a pre-installed static rules that matches PMACs and PIPs.

During a North-South traffic exchange, the core router receives packets with the real IP. If it is an unmatched IP, i.e., this traffic is not known by the core router, it redirects the first traffic packet to the controller to find out the PIP and PMAC related to destination IP. Again, based on information available in its mapping table or fetched from others datacenter's controllers, the local controller install a rule translating destination real IP and actual MAC to the correspondent PIP and PMAC. If the destination IP is located in other datacenter, the rule will route the packets to the encapsulation switch.

In a event of a VM migration to a destination host, its hypervisor announces VM's new location by broadcasting a GARP or RARP frame. This frame is sent to the controller by the previous installed ARP related rule. So, the controller checks if the VM has migrated within its own datacenter.

In case of a same datacenter's migration, the controller assigns a new PMAC and PIP based on the new location and updates all previous added rules in both ingress and egress switches to be compliant with new addresses.

When a cross datacenter migration takes place, the destination datacenter's controller broadcasts the GARP/RARP frame to other controllers. The previous datacenter's controllers replies it with the old PMAC, actual MAC, real IP and IP and removes the old rules in ingress and egress switches. The destination controller then installs new rules on the directly connected ToR (egress) switch to replace PMAC and PIP to actual MAC and real IP. Finally, it sends a special frame called *Controller ARP update* (CARP) to all other controllers specifying the new PMAC, new PIP, actual MAC and real IP. This frame will allow other controllers to remove any remaining rule in their datacenters.

Summarizing, CrossRoads is heavily based on translating actual MAC and real IP to PMAC and PIP, as well as based on communication between various datacenter's controllers that maintains a mapping table between these addresses. These various controllers exchange mapping table's entries, adding and removing them according to the VM location. In network's core, the frames are "routed" based on its PMACs using long prefix matching on L2 addresses, similar to Portland.

3.5.4 Concluding Remarks

CrossRoads is a doable proposal based on addresses replacement. It works by changing the VM's actual MAC and real IP to a PMAC and PIP addresses, that carry topological placement information. Within a datacenter, frames are forwarded by long prefix matches over L2 address. Cross datacenter communication are achieved by using an encapsulation switch. Finally, external communication are done using traditional L3 routing using destination PIP. For its goals, live and offline VM mobility, it has proven its usefulness.

The main obstacle to CrossRoads deployment is that it imposes that all network devices within

a datacenter must be OpenFlow-enabled switches to allow long prefix match forward at L2, to allow support addresses translation rules, to allow frames' redirection to encapsulation switches and to allow encapsulation/decapsulation processes per se.

Another important shortcoming is the huge number of OpenFlow rules that must be maintained on switches. For instance, if an egress ToR switch connects 40 hosts running 16 VMs each, it will connect 640 VMs total. For each VM there must be two rules on egress traffic, one for replacing PMAC with actual MAC and another for replacing PIP with real IP. So, there must be 1280 rules on it. Furthermore, on an ingress traffic for each destination VM communication a local VM wants to exchange data, two rules must be added - again one for replace actual MAC with PMAC and another to replace real IP with PIP. If each local VM communicates with another, say, 50 VMs, there will be 100 rules per VM. If a host runs 16 VMs, it will be 1600 rules for ingress traffic, plus 1280 rules for egress traffic totaling 2880 rules in a single ToR switch.

Finally, the state management is dependent upon the hypervisor type each datacenter uses, because CrossRoads architecture was designed in a way that datacenters controllers expects a RARP frame sent by the hypervisor to annotate the new VM locality. VMWare ESX/ESXi(VMWare Inc., 2015a) is one a few that uses this protocol. In other words, CrossRoads are mainly dependent only over VMWare, not allowing other types of hypervisor within datacenter.

3.6 NVP

The Network Virtualization Platform(KOPONEN et al., 2014), or NVP for short, is one of the most recent proposal to deal with multi-tenant datacenter's network virtualization. The authors brings to debate that different workloads, whatever they are from one or multiple tenants, require different network topologies and services. Tenants usually want the ability to migrate workloads from their home or enterprise networks to service provider datacenters without modifying it and thus retaining the same networking configurations.

The service provider datacenters, in turn, must be ready to meet those new demands on running migrated workloads seamlessly. To preserve operational flexibility and efficiency, these migrations must be accomplished with a minimal effort and without operator intervention.

Finally, traditional networking technologies have failed to provide those tenant and service provider requirements. NVP then was proposed to tackle such problems.

3.6.1 Properties

NVP defines some properties which are in a higher level than other already discussed proposals. Standard properties like isolation, throughput, scalability and flexibility are faced as a "must-have". Others, like low cost, is not in NVP radar, because it uses advanced networking hardware capabilities, like *TCP Segmentation Offload* (TSO), or can uses newer network protocols, like VXLAN.

The two distinct properties defined by NVP's authors are:

- **Virtual topology:** Since different workloads require different topologies and services, the virtual network's topology can be anyone needed, completely decoupled from physical

topology. For instance, a service discovery protocol based workload requires a L2 network - with broadcast semantics and flat addressing -, while large analytic workloads demands L3 service - such as multi-tier applications. Finally, many other applications are based upon services on L4 to L7. Nowadays it is difficult to a single physical topology to support the configuration requirements of all these workloads.

- **Virtual address space:** Each workload must maintain its original L2 and/or L3 addresses. Workloads today are often using the same address space as the physical network. This leads to a number of problems, like VMs cannot be placed on arbitrary locations (hosts), tenants can't use their own IP address management or even cannot change the addressing type, from IPv4 to IPv6, for instance.

On an ideal scenario, networking layer might allow arbitrary network topologies and addressing architectures to be overlayed on top of the same physical network, with a minimal or even no operator hands-on configuration. Its inspiration comes from server virtualization, where VMs are created, migrated, destroyed, turned into a template, etc, by the hypervisor. So, those properties are the focus of NVP.

3.6.2 Design requirements

As mentioned before, NVP raises its approach to a next step towards virtualization of a network by ruling the system that manages and extends the physical infrastructure, introducing abstractions over network resources. These abstractions are used as a path to assign control over virtualized resources to each tenant. In other words, a tenant interacts with a management system, which has control over resource's abstraction, asking for services like forwarding, routing, and so forth. Between the management system and the physical network there is a *network hypervisor* which arbitrates on physical network usage.

First, a *control abstraction* must be introduced. This abstraction allows tenants to create and define a set of logical network elements, also called logical datapaths. Logical datapaths are defined by a set of pipeline stages, each one covering a packet forwarding pipeline interface that contains a sequence of lookup tables capable of matching on packet headers and metadata established by earlier pipeline stages. Thus, tenants have the ability of configure these logical network elements as they would do on physical devices, so they have their own control plane - a *virtual* control plane. The network hypervisor consecutively has the duty of managing the various tenant's virtual control planes, multiplexing them and configuring down the physical network through its own control plane.

Second, a *packet abstraction* must be provided. This abstraction gives the same switching, forwarding, routing and filtering services to a tenant's packet flow in the provider datacenter as they would have in the tenant's home network. This can be accomplished within a logical datapath. An example, a tenant's control plane might want to provide basic L2 forwarding service to flows from one VM to another. So, the control plane populates a single logical forwarding table entry explicitly matching on destination's VM MAC address, sending the flow to the port this VM is connected to.

Those two abstraction exploit the two ways how a tenant interacts with a network: the tenant configures network elements for forwarding and the tenant's VM send packets through these elements. Summarizing, those two abstraction, along with logical datapath, are the foundation in which NVP operates.

3.6.3 Implementation

The network hypervisor designed by NVP supports control and packet abstractions by implementing tenant's logical datapaths over the service provider datacenter's physical network forwarding infrastructure.

The logical datapaths are implemented in the virtual switches running on each host - by the host's hypervisor - through a set of overlay tunnels *and* a set of flow table entries in logical flow tables. Tunnels are point-to-point traffic over IP connecting every pair of host. This means that the logical datapaths are almost entirely implemented on the origin virtual switches, thus network physical devices do not "see" anything else than traditional IP traffic. A SDN controller - in fact, a cluster of SDN controllers, so high-availability can be reached - is in charge of configuring virtual switches with the necessary logical forwarding rules serviced by tenants. Furthermore, NVP does not control physical network devices neither how IP packets are routed. On its point of view, this kind of control is a home network's or service provider datacenter's operator responsibility. NVP only assumes that physical network provides uniform capacity across servers and multiple paths, both built on ECMP-based load-balancing, for instance.

NVP's tunnels provide a clever way for unicast communication. But, they are not feasible for broadcast or multicast traffic. NVP deals with these kind of traffic pattern in two ways. The first way is that the sending hypervisor copy the packet and deliver it to each involved destination. This mechanism is doable for low broadcast/multicast volume. The second way is to employ a dedicated server host, called *service node*, also managed by the centralized SDN controller. So, the broadcast/multicast packet is sent to the service node, who will copy and deliver the packet to each destination.

A tenant also would be able to reach an outside network other than a virtual network. This outside network can be, for instance, a tenant's remote physical network or even the Internet. This routing service is done by a *gateway node*. This node is also a dedicated server host running a virtual switch, managed by the SDN controller as well.

Any packet entering the virtual switch must be sent through a logical pipeline corresponding to the logical datapath which the packet belongs. These packets can ingress coming from a virtual network interface card (vNIC) attached to a VM or from a tunnel from a different host, service node or gateway node. The SDN controller cluster knows which logical pipeline the packet must be assigned by which vNIC or tunnel ID it came from. Finally, NVP leverages two protocols to configure virtual switches. Tunnels are created and managed by *Open vSwitch Database* (OVSDB) Protocol(PFAFF; DAVIE, 2013), whereas logical datapaths are programmed by adding OpenFlow flow table's entries.

The number of logical datapaths created in a service provider datacenter is quadratic, following the function $f(x) = \frac{n*(n-1)}{2}$ where n is the number of hosts. If there are 10000 hosts, would be created almost 50 million tunnels. So, it would be a tremendous computational overhead to create and manage such large number of tunnels and worse: would be barely impossible to react to a new packet by adding flow table's entries in logical table's pipeline. For this reason NVP does not dynamically responds to packet flows, but precomputes the entire network state and then programs down all virtual switches in the datacenter. Just for knowledge, evaluations made by NVP's authors with 3000 hosts and 7000 logical datapaths, a number way less than required, a three host's Xeon cluster with 12 cores each took about *one hour* to compute the entire network state. But after such time, tracking virtual network topology changes, such as VM migration, requires much less computing power.

In order to ease control plane programmability by tenants as well as faster compute of network's state, NVP's authors developed an domain specific language called *nlog*. With nlog, tenants can ask for the use of a tunnel, set hosts and virtual ports origin and destination, choose an encapsulation, and so forth.

NVP's tunnels are created over IP using the service of a tunneling protocol. It supports three options: STT, VXLAN and NVGRE. These protocols were already visited in Section 2.3.

3.6.4 Concluding Remarks

Network Virtualization Platform is one of the most novel proposals for multi-tenant datacenter network virtualization deployment. It is inspired on many other investigations, like Trellis, when using tunnels to decouple virtual topology from physical, previous work on network forwarding plane virtualization(CASADO et al., 2010), SDN, distributed control plane(KOPONEN et al., 2010) and programmable dataplanes(PFAFF et al., 2009). Furthermore, NVP has successfully glued together those myriad of technologies, but through a huge trade-off.

Trellis has shown that using tunnels is a conceivable way to implement virtual topologies. But this technique lacks of scalability - managing point-to-point tunnels between each hosts in datacenter is hard. To circumvent this drawback, NVP's authors decided to compute the state of all virtual networks before programming physical forwarding devices, working proactively. A NVP cold start, i.e. starting the entire datacenter (SDN controllers, hosts, switches, and so forth) from the turned-off state or after a major disaster to a steady state, takes hours of computation, even using a large cluster of distributed SDN controllers. This means that a huge amount of resources, that is reflected to CAPEX, must be dedicated to them.

NVP's tunnels lays upon IP, which demands specialized network switches that can route based on L3 information. This fact increases both CAPEX and OPEX. Moreover, encapsulating the entire VM's frame in a L3 header expands protocol overhead. This means that data payload size is squashed and fragmentation is increased, which can lead to a higher probability of retransmission in a event of one packet in a flow is dropped or missed. In case of using STT as encapsulating protocol, it is worth to state that, although increasing fragmentation, host's CPU cycles is saved due to TSO and LRO techniques. But, to take advantage of these techniques, datacenter's hosts must be equipped with higher-end Ethernet NICs, also increasing CAPEX.

Finally, in order to use NVP virtual switches must support STT, VXLAN or NVGRE. They also have to support the OVSDB Protocol. Not every SDN-aware virtual switch works with them. This fact narrows down deployment choices what can lead to problems like datacenter's operator having to make its physical topologies or resources compatible with a single virtual switch type or implementation.

3.7 Chapter Summary

This chapter discussed the latest researches in datacenter networks arena: Trellis, VL2, PortLand, NetLord, CrossRoads and NVP.

Trellis proposes the usage of a mesh of virtual links where any virtual topology can sit upon.

This is the most influential aspect of it. Many others technologies, like NVP, leverages this idea in their proposals.

VL2 uses an IP-in-IP encapsulation to allow many services to be executed by hosts. Although such encapsulation was not new, the VLB aspect of VL2 was used in many other researches.

PortLand and CrossRoads are similar. At least on L2 they are identical. The difference is that CrossRoads leverages the PIP usage, influenced by VL2. Their main contribution to other proposals was the study of using a large number of OpenFlow rules in datapath's flow table.

NetLord, along with many other aspects, has played with encoding tenant information within L3 header. This fact pushed others proposals, again like NVP, to dedicate a custom header to carry tenant's data.

Finally, NVP is today the most prominent "bleeding-edge" technology for virtualized datacenter networks. It is supported by a leader in virtualization area (VMWare) and is the base of a production-ready, commercial platform called NSX(VMWare Inc., 2015b). By these facts it is accepted as a "definitive" solution in SDN for datacenter. The author of this work disagree.

4

HotOM Proposal

In this chapter, the HotOM proposal is presented. A detailed explanation is disclosed on every single aspect that disposed together makes HotOM.

As seen in Section 1.1, HotOM leverages network virtualization and programmability on datacenter's network with the objective of achieving high network resource's utilization while still using legacy switches.

4.1 Goals

The previously stated HotOM main objective was split in five goals. They were designated as **G*x***, where ***x*** is an index ranging from 1 to 5.

G1: *HotOM must work on network core's legacy switches.* This goal is directly related to the main objective. In a previously deployed datacenter, a set of network devices, a collection of management system, a topology, etc, were already chosen and are in production. Maintaining existing investments helps to avoid OPEX boost. In addition, there is the advantage of promoting the infrastructure to support IaaS, and consequently increasing the number of hosted tenants.

G2: *HotOM must be L3 routing agnostic.* By not being tied to IP protocol's routing schemes, HotOM enables two important factors. First, the datacenter network's administrator can experiment and use newer protocol stacks on its infrastructure. There are many efforts in academia proposing L3 protocols other than IP. Furthermore, such new schemes can be offered to tenants as well, turning into a competitive advantage over ossified architectures. Second, it is not imposed to network core's forwarding devices to support IP protocol. In other words, they have not to be L3 switches. This fact alleviates datacenter's owner the burden of buying specialized and expensive forwarding devices. Only simpler L2 devices are required, helping to maintain CAPEX as lower as possible.

G3: *HotOM must provide complete L2 network isolation to tenants.* This goal enables each tenant to define its own L2 and L3 addresses, as well as its virtual topology. For instance, a tenant may want to deploy a three-tier application, each tier with their own L3 subnet, and interconnect them through a virtual router. These tiers might have L2 addresses that would therefore overlap with another tenant's

virtual network and, of course, HotOM must insure complete isolation between them. Moreover, this goal facilitates the off-line migration of a network, no matter if it is physical or virtual, from another infrastructure or platform to the HotOM-enabled datacenter.

G4: *HotOM must lay on a small L2.5 shim protocol header.* This shim header must not only be small, but *effective*. One doable way to virtualize traffic between two endpoints is by employing the encapsulation of the original packets. Some proposals (GREENBERG et al., 2009) use an IP-over-IP scheme. Others (BHATIA et al., 2008),(MUDIGONDA et al., 2011),(KOPONEN et al., 2014) go further and encapsulates the entire VM frame into a L3 and, consequently, L2 packet. They boost protocol overhead, where part of the encapsulated frame is used by protocols themselves, not for useful application payload. To overcome this drawback, HotOM was design to encapsulate VM's L3 data and up into a small L2.5 header, diminishing overhead.

G5: *HotOM must provide scalability beyond VLAN's limit.* Although VLAN is still widely in use today, virtualized datacenters requires support to instantiate hundreds of thousands of VNs, or even more. So, 802.1q(IEEE Computer Society, 2014a) technology is not feasible anymore. Overtaking this limitation allows the achievement of high network resource's usage, and thus profitability. Bleeding-edge proposals, such as HotOM, allows 16.8M VNs over the same physical infrastructure.

Those goals were defined as the foundations that drove HotOM's development. Some rationale behind each of them were already discussed above, and will be later expanded in further Sections along with each HotOM's aspect.

4.2 Topology and physical infrastructure

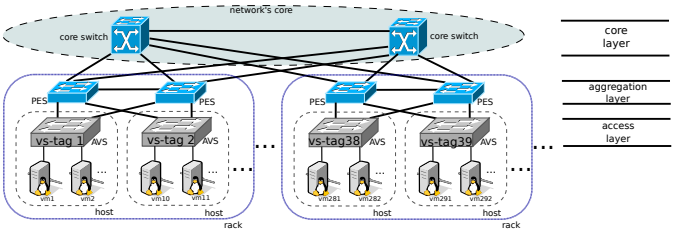
Datacenter's topology is a subject that can bring up many discussions. Each one has its own strengths and weaknesses. In general, it is up to the network's engineer to decide which topology is more qualified to his needs.

Foreseeing this situation, HotOM does not stick to any particular topology. It can be used in an already designed datacenter network. HotOM focus its functionality on network edges by the employment of OpenFlow-enabled virtual switches.

Moreover, the architectural aspect of HotOM was designed to fit flawlessly into the so-called traditional datacenter's organization(Cisco Systems, Inc., 2007). This means that a migration process from former technologies to HotOM is made in a smooth way.

The traditional datacenter's topology was proposed by Cisco Inc and was heavily adopted world-wide. In this topology, hosts are connected to network's core through a physical switch called *access switch*. In turn, the access switches are connected to a upper layer called *aggregation* or *distribution* switches. In HotOM, access switches are pushed down to virtualization layer and are called *Access Virtual Switches* (AVS). Moreover, AVS' must be *OpenFlow-enabled*. It is AVS' responsibility to provide VN abstraction and VM connectivity by applying HotOM mechanisms, on behalf of an OpenFlow controller. The aggregation layer is provided by *Physical Edge Switches* (PES), which are in charge of connecting hosts and AVS' to network's core (Figure 4.1).

Figure 4.1: HotOM datacenter topology



HotOM requires only a simple physical switches' configuration: every link must be configured as *trunk*. This simple on-time configuration is necessary since, looking ahead for Section 4.3.1, HotOM uses VLAN ID tag for frame "routing". There would not be a great effort for network's administrator to accomplish such configuration, since it can be conveniently done using switches CLI, scripting or even via SNMP.

4.3 Key aspects

HotOM technology is a group of concepts and network functions that are coordinated to work together. For a better understanding, each of its key aspects is presented individually in this section.

4.3.1 VLAN ID based forwarding

Any multi-tenant network technology needs a way to create a virtual topology upon physical infrastructure to provide connectivity among VMs. The most accepted way to accomplish this task is to create L3 tunnels interconnecting every virtual switch(CASADO et al., 2010). Surely this approach is uncomplicated to deploy, because its needs only an IP route between endpoints (hosts), with the advantage of easily reach hosts outside L2 boundaries.

But, the tunnel approach has some drawbacks. First, the number of tunnels increases by the power of two of hosts quantity. For instance, in a datacenter with only 4094 endpoints, it would be about 8.3M tunnels to supervise. It is hard to manage this huge mesh of connections. Second, network forwarding devices must be L3 compliant. Also, these forwarding hardware must be more complex, with CAMs to store L2 forwarding tables, *Ternary Content Addressable Memories* (TCAM) to store L3 forwarding tables, and so forth. So, they are much more expensive than simple L2 ones. Indeed, CAPEX would increase significantly.

Packet switching in network's core must be as fast as possible. The process of searching and matching packets in forwarding tables impacts in backplane latency. Using devices that needs to referer to two tables (L2→CAM and L3→TCAM) before deciding how to switch out the packet always introduces delays.

In turn, HotOM uses the concept of *virtual path*, where an identifier singles out each path. This concept has been used in other protocols such as *MultiProtocol Label Switching* (MPLS)(ROSEN; VISWANATHAN; CALLON, 2001). In MPLS, a L2.5 header with a label (among a few other fields) specifies which path a frame must transverse. So, it allows a MPLS-compliant's network device to forward a frame based only in a small field, hence simplifying searching/matching procedures.

On the datacenter's network point of view and with the vision of MPLS forwarding approach, HotOM was designed to take advantage of fast switching capabilities provided by L2 technologies. HotOM uses frame's VLAN ID tag as an unique key (an index) in a forwarding decision. To each AVS is assigned an unique VLAN ID tag called *vs-tag*. Summarizing, HotOM pushes network's forwarding decision function to a label (VLAN ID↔vs-tag) in L2 header, with the advantage of neither using uncommon MPLS or expensive L3 switches.

The reason for using VLAN ID for routing is to accomplish goals **G1**, **G2**, and **G5**. This forwarding mechanism demands simpler L2 devices. **G2**, in particular, avoids the employment of expensive L3 switches. Moreover, **G2** also permits tenants to use arbitrary L3 protocols other than IP, turning HotOM into an open technology for new protocol stacks above L2.

Current switches forwards frames based on VLAN ID tag *and* destination MAC address. HotOM creates a possibility for developing a switch that forwards frame *only* based on VLAN ID tag. This kind of switch would be much faster and cheaper than current ones, because it might use a very small CAM portion, just to store [`tag` → `port`] entries, accelerating searching/matching in forwarding tables.

Using VLAN ID tags as an index to identify AVS has a drawback. They are a 12-bit long header field, so it is possible to address up to 4096 AVS'. HotOM architecture overcome this problem by reserving two *special purpose* vs-tags and the remaining 4094 AVS' are grouped in a logical arrange called *cluster*.

The first special purpose vs-tag reserved, number 1, is used to tenants reach public IP address space, i.e., to send traffic to the Internet. This means that the border router, the one that interconnects the datacenter to the Internet, is connected to the AVS which vs-tag is 1.

The second special purpose vs-tag, number 2, is used to reach a VM placed in another cluster. For instance, if AVS' vs-tag 2 is a dedicated switch with 48 ports, the entire datacenter would host 196.512 AVS' (48 times 4094). If, say, 48 VMs are executed by one hypervisor and therefore connected to each AVS, the entire datacenter would host 9.4M VMs. Special purpose AVS' and clusters provide a high level of scalability in compliance with goal **G5**.

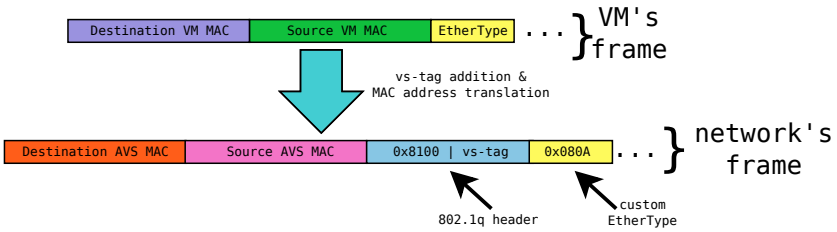
4.3.2 MAC address translation

As discussed in Section 4.3.1, HotOM implements routing based on VLAN tags. But actual legacy L2 switches forwards frames based on this tag *and* destination MAC address. For this reason, a MAC address translation must be performed. When a local VM sends a frame to a remote one, i.e. to a VM connected to another AVS, the frame's source address is translated to the local AVS' MAC address, while the destination address is translated to the destination AVS' MAC address. In addition, the destination vs-tag is inserted into 802.1q header. So, the final frame that is actually forwarded to network core has on its outermost L2 header the destination AVS' address, the source AVS' address and the vs-tag in VLAN header. Figure 4.2 depicts the translation mechanism.

Besides VLAN ID, the 802.1q header has three additional fields. The *Priority Code Point* (PCP) is used to signals the frame's priority level and usually is mapped to multiple QoS queues in switches. The *Drop Eligible Indicator* simply point out if the frame should be dropped if there is some network congestion. These two fields are not dealt by HotOM itself, but it can be ruled by an additional management system, with the advantage of adding QoS management to HotOM.

The third 802.1q header's in focus is the *EtherType*. It has the same semantics that EtherType

Figure 4.2: MAC address translation



from Ethernet (802.3) header, being used to identify the type of the immediate frame's upper layer. Looking ahead from Section 4.3.4, HotOM uses a L2.5 shim protocol header. Therefore, EtherType must be translated accordingly to a value that expresses that a HotOM header is in place. The value of `0x080A` was chosen in the current HotOM implementation as temporary because it is unused, i.e., it is not reserved for any organization as shown in IEEE EtherType Public List(IEEE Standards Association, 2015). A definitive value can be assigned as soon as HotOM becomes standard. Finally, in order to not miss the original EtherType value, this field is saved by a mapping into HotOM's *type* field, as further discussed in Section 4.3.4 as well.

There is a key advantage on using MAC address translation mechanism: physical switches (both PES and core) are not aware that a number of VNs are in place. They just deal with the AVS' vs-tag and MAC addresses. For example, if there are 48 VMs in a traditional Ethernet network connected to a virtual switch, each physical network device would populate 48 similar forwarding table's entries as `[(VM_MAC) → port]`. When using HotOM, only *one* entry as `[(vs-tag,AVS_MAC) → port]` would be created. This is an important factor to increase scalability, to maintain a low pressure over physical switches' forwarding table (CAM) size and to allow the employment of simpler and cheaper network devices, as goal **G1** requires, while providing a fast table lookup and, thus, switching speed.

Finally, virtual switches implementations assumes some MAC address when instantiated. Open vSwitch, for example, creates a random UUID and uses it as its L2 address. So, it would be hard to register associations between vs-tags and MAC address, as well as tracking down frames when doing a network debugging. But as an OpenFlow-enabled virtual switch, it is possible to program AVS to react to any MAC address. HotOM then uses this capability to well define which address a particular AVS uses. It was defined that the AVS' MAC address is a 48-bit encoding of its vs-tag value. For instance, if vs-tag is `4052` (`0x0FD4` in hexadecimal, the associated MAC address will be `11:22:33:00:0F:D4`. There are advantages with this design choice. For example, the populated forwarding table's entries within physical switches would be consistent, i.e. not dependent on random actual AVS MAC address, and predictable. Moreover, network debugging, by capturing and analyzing packets, surely will be facilitated.

4.3.3 Gratuitous ARP

As discussed in Sections 4.3.1 and 4.3.2, an ingress frame from a VM has its destination MAC translated to the destination AVS address, then a vs-tag is added into 802.1q header. In order to correctly deliver a frame, physical switches must *learn* a virtual path that leads to a destination AVS. This means

that they must know through which port an AVS is reachable by adding entries to their forwarding table. To accomplish this, HotOM uses Gratuitous ARP (GARP) broadcast packets to force switches to set L2 "routes" (forwarding table's entries).

GARP is a special crafted ARP broadcast packet sent, from time-to-time, by the AVS to *announce* its vs-tag and MAC address. Every physical switch on the network, starting by the PES, receives that GARP packet on a port, adds an `[(vs-tag,AVS_MAC) → port]` entry in their forwarding table and then floods the packet to others physicals switches throughout their ports. Using this mechanism, all switches learn paths to every single AVS.

Figure 4.3 depicts a GARP packet captured using the `tcpdump`(TCPDump Project, 2015) tool while evaluations were being performed in testbed. This packet was sent by the AVS' which vs-tag was designed as `4052` and its MAC source address was set accordingly (`11:22:33:00:0f:d4`). The main characteristic that makes a GARP packet different from an ordinary ARP is that the sender and target IP addresses are set to the same value of `0.0.0.0`. This has no meaning outside from GARP scope because does not make sense a given device with IP address `0.0.0.0` querying for who has the same `0.0.0.0` address. Indeed, physical network switches does not takes the ARP payload into account, it just deals with data available on L2 header.

Being a broadcast, GARP tends not to be scalable. It should be done less frequently as possible. A typical physical switch holds forwarding table entries for 300 seconds in their CAM. This timeout is configurable and should be increased when using HotOM. So, sending GARP frames a little less than this time frame is sufficient to populate the desired entries while avoiding network congestion.

4.3.4 HotOM Protocol Header

To achieve high scalability in terms of VN's quantity, some mechanism to add a layer of indirection is needed. This indirection defines orthogonal address *namespaces*. So, it is possible to run many different VNs over the same physical network using the same L2 and L3 addresses without overlap between them. Furthermore, another desired property is to identify a huge amount of VMs on each VN. These are the main purposes of the *HotOM Protocol Header*.

The HotOM protocol header is a L2.5 shim one placed as a VLAN payload in a frame that transverses the network's core. It is composed by four fields, three of them are 24-bit length and one is

Figure 4.3: GARP packet

```
Frame 1: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
Ethernet II, Src: 11:22:33_00:0f:d4 (11:22:33:00:0f:d4), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
802.1Q Virtual LAN, PRI: 0, CFI: 0, ID: 4052
Address Resolution Protocol (request/gratuitous ARP)
    Hardware type: Ethernet (1)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    [Is gratuitous: True]
    Sender MAC address: 11:22:33_00:0f:d4 (11:22:33:00:0f:d4)
    Sender IP address: 0.0.0.0 (0.0.0.0)
    Target MAC address: Broadcast (ff:ff:ff:ff:ff:ff)
    Target IP address: 0.0.0.0 (0.0.0.0)
```

8-bit length. They have an Ethernet-like semantics.

Figure 4.4 depicts field's placement within the header. The first field is the *net_id*, being an *index* that identifies which VN a given flow belongs to. It is represented as a hexadecimal value. For instance, a value like `0x00FFBA` defines a VN. Since this field's length is 24 bits, a datacenter can instantiate up to 16.8M VNs, far beyond the 4K VLAN's limit and thus achieving goal **G5**. Summarizing, *net_id* applies the desired layer of indirection *and* identifies the VN.

The following two fields are ***HotOM addresses*** and they accomplish the other HotOM Protocol Header's purpose: VM addressing. The second field is the HotOM *destination* address, related to the destination VM MAC address, and the third is the HotOM *source* address, related do the source VM MAC address as well. The field's values are the last 24 bits from VM MAC addresses. Thus, each VN can address up to 16.8M of VMs. Similar to Ethernet, HotOM represents these addresses as three colon-separated hexadecimal bytes. For example, values like `00:00:01` or `AD:DA:FF` are valid HotOM addresses.

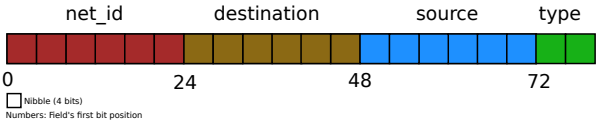
The fourth field is the ***HotOM type*** field. This field is a simple mapping from original EtherType. Since this field is 8-bits length, it is possible to map 256 different types of Ethernet frames. This number is sufficient for a real datacenter's deployment because only a few L2 payloads, like IPv4, IPv6, AoE, HyperSCSI and FCoE, are effectively used. Table 4.1 shows the mapping between EtherType and HotOM's type field.

Table 4.1: Type field mapping

L3 Protocol	EtherType	HotOM's type
IPv4	`0x0800`	`0x00`
IPv6	`0x88DD`	`0x01`
AoE	`0x88A2`	`0x02`
HyperSCSI	`0x889A`	`0x03`
FCoE	`0x8906`	`0x04`
FCoE Init. Prot.	`0x8914`	`0x05`

HotOM assigns VM's MAC addresses in a predefined way. The first 24 bits, known as *Organization Unique Identifier* (OUI), *should be* equal to `00:00:00`. This definition eases VN's migration from another virtualization platform. For instance, VMWare uses `00:50:56` as OUI, Hyper-V uses `00:15:5D`, Xen uses `00:16:3E` and VirtualBox uses `08:00:27`. Using a simple script, it is possible to convert this 24-bit value to `00:00:00`. The last 24 bits of VM's MAC address, just like any other platform, are freely chosen as desired. This policy is enforced by the Local Agent Service (later discussed) when it replies ARP requests. Finally, HotOM addresses are allocated by the tenant, achieving goal **G2**.

Figure 4.4: HotOM Protocol Header's fields



HotOM Protocol Header field's length were chosen to fulfill two aspects. First, some widely accepted network virtualization technologies, like VXLAN and NVGRE, uses a 24-bit field as a VN identifier. So, the author of this work realizes that 16.8M VNs are a extensive number enough, even for the largest datacenters. Second, understanding that Ethernet address are 48-bit long and desiring a fast translation between it and HotOM address, it is doable to use the remaining 24 bits for VM addressing. These decisions were taken to meet goals **G3**, **G4**, and **G5**.

Network forwarding is performed in a switch, no matter if it is physical or virtual, by executing many repetitive tasks on every packet in a flow. These tasks might be designed to be as fast as possible or data movement would have a poor throughput. With this in mind, the address translation between VM MAC address and HotOM address was designed to be done quickly. The task is simple. Considering `Ethernet[0:95]` as the bit array of destination and origin VM's addresses in an Ethernet header, destination and origin HotOM addresses would be `Ethernet[24:47]` and `Ethernet[72:95]`, respectively. These operations are easy to do in software, by a virtual switch, as in hardware, by a dedicated ASIC or FPGA.

Finally, the concept of locality of a VM in network topology is defined by the vs-tag, since it identifies which AVS a VM is connected to. The concept of identity, in turn, is defined by the virtual network identifier (net_id) and the HotOM addresses, i.e., the net_id *and* the HotOM address when grouped together turns into a tuple that are, by definition, *unique across the entire datacenter*.

4.4 Management and control planes

As any other OpenFlow-based network virtualization technology, HotOM also depends on a control plane to work. But, as not much others, it also has a management plane. The control plane is centralized on the point of view from AVS, but distributed in the point of view from the management plane. The management plane, in turn, is implemented by a central simple process. Both planes are described in following subsections.

4.4.1 Network Coordinator Service

The HotOM *Network Coordinator Service* (NCS) is a central service that is in charge of tracking, computing and maintaining the VNs topologies and managing which VMs are connected in a particular VN. NCS has a "broad view" of virtual networks. Its primary task is to provide data to the *Local Agent Service* (LAS).

Actually, the NCS performs a simple management plane. It is the central database that maintains the entire datacenter's primary network data, such as information about VNs, VMs and AVS. Further, in the early HotOM's implementation, NCS is executed in a host that can be reached from every LAS in the datacenter. But for real large scale datacenters, it might be deployed as an distributed application, taking care of maintaining redundancy for better resiliency and workload distribution.

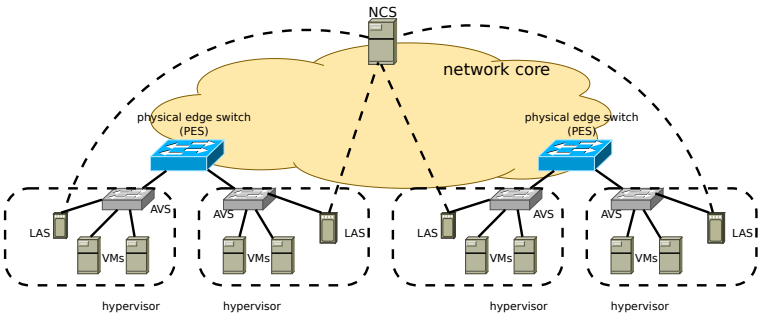
In current HotOM's code, NCS is in charge of storing the following collection of data and their relationships:

- Associations between AVS' and their vs-tags;

- VN related data
    - VN tenant's name/identifier
    - Network identifier (net_id field)
    - A textual description
- AVS related data
    - Virtual Switch Tag (vs-tag)
    - A textual description
- VM related data
    - VM name
    - which VN it belongs
    - which AVS it is connected to
    - which AVS port it is using
    - MAC addresses
    - IP addresses
    - A textual description

Figure 4.5 depicts the relationship between HotOM's NCS, LAS, AVS' and VMs. The dashed line connecting NCS and every LAS outline some kind of InterProcess Communication (IPC) between them.

Figure 4.5: HotOM architecture



4.4.2 Local Agent Service

The *Local Agent Service* (LAS) is indeed the main HotOM's OpenFlow application. It runs on every hypervisor's host. This means that each AVS is connected and controlled by one and only one OpenFlow controller that runs the LAS code. In order words, the LAS implements the HotOM's control plane.

The main LAS's source of information is its local cache table. The data set stored in local cache table is directly provided by NCS. Also, this data set is actually a subset of what is stored and managed by NCS. NCS just pushed down to LAS *only* the data it really needs. For instance, if a given AVS (controlled by LAS) has VMs that belongs to four VNs, only the data (AVS, vs-tags, VMs addresses and so forth) about these four VNs will be stored in the LAS' local cache table. Finally, the local cache was introduced to avoid LAS to continually query NCS about information that probably not changes so much.

The LAS does the following tasks:

- searches local (in-host) cache table about VM and VN's information;
- queries, when there is a cache miss, NCS for VM and VN's information;
- demands, from time-to-time, AVS' to send GARP packets (Section 4.3.3);
- receives and replies, based on local cache table data, ARP queries sent by VMs;
- installs flow table rules in AVS, allowing two local VMs to communicate;
- when a frame is sent from a local VM to a remote one, using data in its local cache, it:
    - adds the destination vs-tag (Section 4.3.1);
    - translates both destination and origin MAC addresses (Section 4.3.2);
    - encapsulates the VM payload into HotOM L2.5 shim header (Section 4.3.4);
    - demands AVS to forward frame to network;
- when a frame arrives from the network to a given VM, using data in its local cache, it:
    - check if the vs-tag is equal to assigned to it;
    - removes the vs-tag;
    - decapsulates payload from HotOM L2.5 shim header;
    - translate both destination and origin MAC addresses back to the original one (from data placed in shim header);
    - demands AVS to forward the frame to destination VM;

Finally, due to OpenFlow limitations, looking ahead Section 4.6, the current LAS implementation does the above listed tasks of applying/removing vs-tags and adding/removing L2.5 shim protocol header *to every packet in a flow* that are transmitted from/to VMs. This means that packets must be transmitted to the controller and so there is an associated performance penalty. Again, HotOM is in a *prototyping* stage to investigate its usefulness on VN instantiation, VM connectivity and network isolation.

4.4.3 Communication between NCS and LAS

The proper HotOM operation relies on the communication between NCS and LAS. In this information exchange, which in fact is an IPC, LAS queries NCS for data, or NCS pushes down data to LAS.

There are many IPC techniques available in Unix-like operating systems like Linux. This IPC could be a plain TCP connection, which is bidirectional, or UDP messages exchanged between them. It could also use a high level application layer protocol, like JSON over HTTP (TCP based) or RPC (UDP based). HotOM itself does not impose any kind of communication method.

In current HotOM implementation a very simple IPC is used. In a cold boot, NCS must be started before LAS and it populates the LAS' local cache table with its related data subset. The local cache table is a simple SQL file in hypervisor. When LAS then starts, it queries the local cache table every time it needs to.

It would be possible that NCS needs to update LAS' local cache table, so it directly updates the SQL file. As any other that is written, the operating system updates the file's time stamp. The LAS, in turn, has a timer that every 5 seconds tests if the local cache table's file timestamp has changed. If so, the LAS discards any query buffer (a small data buffer from table's SQL file) and queries the local cache table. This case probably happens in a event of a live VM's migration (Section 4.5.3).

Finally, if LAS need some data not present in a local cache table, it sends a small UDP packet containing the net_id and IP address of the VM it needs information, asking for the vs-tag and MAC address. This case happens when a local VM issues an ARP query. The NCS performs the same way as described above: it directly write the information in local cache table through a SQL update command.

4.5 Virtual network behavior

This section describes how HotOM behaves when an ARP query is issued by the VM, a VM migration takes place and during a a communication between VMs.

4.5.1 VM's ARP query

In traditional virtualized datacenter network, VM's ARP queries are sent throughout the entire network, forcing physical switches to populate a huge amount of forwarding table's entries among them. These switches can typically hold up to 8K MAC addresses in their tables.

If they are exposed to a higher number of addresses than that limit, there will be many *table misses*[1]. In a event of a table miss, the physical switch will flood out the ARP packet through all ports, wait for the response, search and evict older entries, and insert the new one. All these operations imposes scalability problems into the network.

In a HotOM-enabled datacenter, the AVS is responsible for capturing and sending VM's ARP queries to LAS, that in turn parses and replies them to the issuing VM based on information available in its local cache table. This means that no VM's ARP queries are flooded to network's core, helping to maintain low resource's allocation (CAM) and processing power[2] in physical switches.

[1] Lookups to addresses not found in table

[2] Switch's CPU cycles due search and evict procedures

4.5.2 Local VM communication

Two or more VMs, that belongs to the same VN, can be executed by the same hypervisor and, therefore, are connected to the same AVS. When they communicate to each other, no frame needs to be switched out to the network. This type of traffic exchange is called *local communication* or *intra-AVS communication*.

Assuring that a local VM communication is taking place is really simple. LAS has just to validate that the destination vs-tag is equal to AVS' vs-tag it controls. If so, the LAS installs an OpenFlow rule in AVS' flow table directly connecting both source and destination's ports. This rule is pretty much like connecting the two (or more) VMs in an isolated switch. Thus, their traffic are not subject to interference from another VN.

4.5.3 Live VM's migration between hosts

In any datacenter, no matter its size, live VM's migration is a desired feature that has many ends. It is used for spreading workload purposes, where VMs are conveniently placed in previously chosen hosts for better workload distribution among hosts, for better traffic distribution among switches and so forth. Live VM's migration is also used on planning host downtime and maintenance, where the VMs that a host executes are migrated to another one, allowing the host to be turned off. This assures seamless continuity of tenant VN's operation.

Live VM's migration is performed by a series of steps, most of them issued by the VM administration system. Initially, two basic requirements must be fulfilled. First, both origin and destination hosts might communicate with each other through the network. Second, VM disk image file must be accessible and shared between them.

It is not the purpose of this work to describe details on live VM's migration. It will focus on how network behaves. So, in a glance, when the VM administration system ensures the above two requirements and starts a live migration, first the VM's memory image is copied from origin host to destination. Then, the VM is put in a suspended state for a short period of time, and its CPU state (registers and so forth) is also copied to destination. Further, the VM administration system falls back the VM to running state. After that, it must somehow interact with the physical or virtual network in order to inform new VM's position.

Generally, in traditional virtualization platforms, when a VM is migrated, its new location is announced to the core network. This announcement is necessary because switches *see* the VM's MAC addresses and must *learn* the new VM location to be able to forward frames to the right switch's port destination. The announcement is normally done by broadcasting a *Reverse Address Resolution Protocol* (RARP) packet.

In HotOM, live VM's migration is much more simpler because network core's switches does not deal with VM's MAC address. The VM administration system just notifies NCS the destination host - actually the destination AVS - the VM was migrated to. NCS then updates its own database and pushes this new information to all LAS's local cache table which AVS' must know the new location.

After that "new location" data transfer from NCS to LAS, an origin AVS, on behalf of its LAS, will label frames with the vs-tag of the AVS the VM migrated to. Again, the network physical switches

does not realize that just happened a VM's migration.

4.5.4 Packet's life

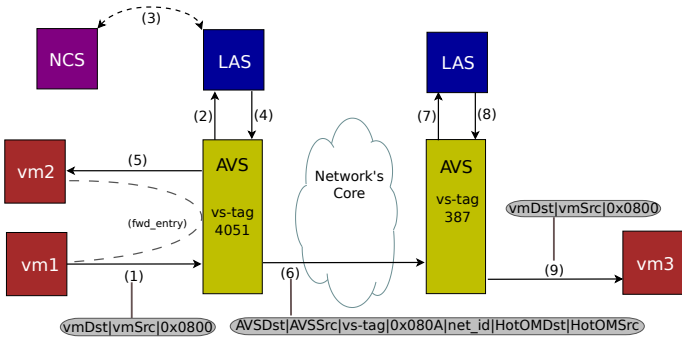
For a better understanding how HotOM works, Figure 4.6 shows every performed steps on a unicast communication within a given VN. A frame originated from VM_1 has two possible destinations: VM_2 or VM_3. During the entire process, all actions are done over L2 protocol headers (Ethernet and 802.1q).

Suppose VM_1 wants to send an IP packet to other VM. First, it broadcasts an ARP query packet on *(1)* asking for a MAC address for that IP address. The corresponding AVS captures this query and sends it to LAS on *(2)*. The LAS makes a lookup in its cached data searching for an association between MAC address and IP. If there's a cache miss, the LAS retrieves that association from the NCS on *(3)*, caching it. Then, it creates an ARP reply and demands AVS to send it back to the VM (reverse direction of *(1)*).

After the ARP address resolution, VM_1 sends a frame in *(1)* to the destination VM. The AVS captures this frame and sends it to LAS through an OpenFlow message *(2)*. At this point, LAS can perform two actions. If the destination VM is local, i.e., it is connected to the same AVS, *(4)* is an OpenFlow message installing a forwarding table entry in AVS direct connecting VM_1 and VM_2 *(5)* (dashed line). If the destination VM is remote, LAS labels the frame with destination AVS' vs-tag, changes EtherType to 0x080A, translates the source and destination MAC addresses to its MAC and remote AVS' MAC, adds the L2.5 shim protocol (by adding *net_id*, *HotOM addresses* and *type*) and demands by OpenFlow message *(4)* to AVS to forward the frame *(6)* to network's core through PES.

Once the frame arrives at the destination AVS, it sends the frame to its LAS by the OpenFlow message *(7)*. The receiving LAS then removes the vs-tag and reads the L2.5 shim protocol header. With the information available within that header (net_id field), the LAS knows to which VN that frame belongs to and reading the HotOM source and destination addresses, it translates the MAC addresses back to the original one, also translating the original EtherType back using the *type* field. Finally, LAS demands the AVS to deliver the frame by issuing the OpenFlow message *(8)*. A frame, identically to the original one, reaches the destination VM_3 in *(9)*.

Figure 4.6: Packet's life



4.6 OpenFlow applied to HotOM

Although OpenFlow introduces flexibility to networks, it fails badly on providing a way to introduce arbitrary protocol headers in flows while accomplishing performance. As discussed in Section 2.6, OpenFlow works by creating a set of matching-action flow table's entries in data plane (switch) based on pre-defined protocol header's fields. It is, somehow, rigid.

OpenFlow can add a rule in AVS matching frames from a VM, adding a VLAN ID tag and translating MAC addresses. But it *cannot* add or remove a HotOM Protocol Header or any other custom one. So, the current HotOM implementation faced a trade-off between functionality and performance: the LAS had to be in charge of inserting/removing the HotOM header (function), but with a drawback of inserting delays on every frame (performance penalty).

4.7 Chapter summary

This chapter presented the HotOM proposal, its goals and how it works.

The main HotOM's goal of high utilization of network resources in addition of using legacy network's core switches is achieved by employing a set of techniques. First, frames are forwarded based on their VLAN ID tag and, by now, the destination AVS' MAC address. This fact ensures simpler and less expensive L2 switches. The source and destination AVS' MAC addresses are in L2 header due to an address rewriting mechanism. VM's addresses, in turn, are placed along with the virtual network identification (net_id) in a small L2.5 shim header - the HotOM Protocol Header.

HotOM's mechanisms are mostly done by the AVS running in each host, on behalf of LAS. As discussed, OpenFlow has some limitations that prohibits insertions and removalsls of an arbitrary header.

Finally, a detailed packet's life was discussed, in order to give a better understanding on every step during a virtual network communication.

5

Analyses and Comparisons

This chapter describes the proposal analyses and comparisons. First, practical analyses are discussed along with their results. Then, comparisons between HotOM and other technologies that were visited in Chapter 3 are performed.

5.1 Testbed description

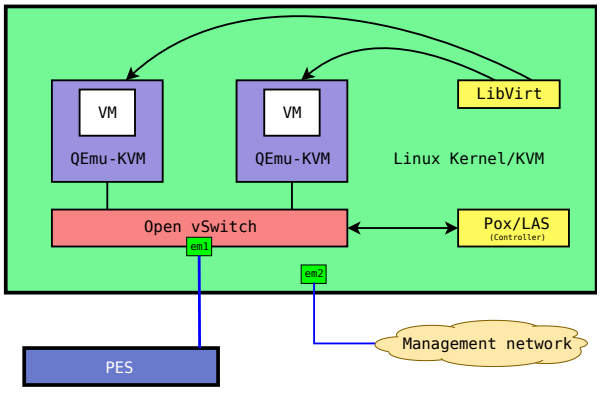
For a real implementation and analysis of HotOM with physical devices (server host and legacy switches), a testbed was made available within the datacenter of *Núcleo de Tecnologia da Informação* (NTI) at Federal University of Pernambuco. This testbed consist in two host servers and three network switches, one playing a role as *core* switch and two acting as *physical edge switches* (PES). The device models are:

- **Server:** Dell PowerEdge R310
  - One Intel Xeon Processor X3470 2.93GHz
  - 2Gb of RAM
  - One Hard Drive SATA 250Gb 7.2K RPM
  - Two 1Gbps Ethernet NICs with Broadcom BCM5716 chipset
- **PES:** ExtremeNetworks Summit x430t
  - 24 ports
  - Up to 1Gbps per port
  - Forwarding table holds up to 16K MAC addresses
  - 56 Gbps total bandwidth
  - 41.5 Mpps
- **Core:** ExtremeNetworks Summit x440t
  - 24 ports
  - Up to 1Gbps per port

- Forwarding table holds up to 16K MAC addresses
- 88 Gbps total bandwidth
- 65 Mpps

On each server was installed the Linux operating system, more precisely Fedora 19 64-bit distribution with kernel 3.14.17-100. Linux supports a number of virtualization technologies. The chosen machine virtualization technology was KVM[1], mainly because it is well integrated with Linux kernel, has low overhead and thus good performance. Moreover, it is extensively supported by Linux community and vendors. KVM needs a modified version of QEMU, called QEMU-KVM[2], as a userspace application for running the VMs themselves. It was used QEMU-KVM 1.4.2. Moreover, the VMs were executing CentOS 6.6 64 bits Linux distribution with kernel 2.6.32-504. This distribution was used because the VM image was already built and loaded with all need tools for evaluations. Another and important reason is that a virtualized datacenter houses a wide variety of VMs with different operating systems. So, its doable and incremental to learning process performing experiments with different Linux kernel versions. Figure 5.1 depicts softwares running in a host.

Figure 5.1: Softwares running in a server host



During the evaluation some administrative operations over VMs like instantiation, connecting to virtual switch, startup, shutdown and so forth were needed. For these VM administration's purposes Libvirt[3], version 1.0.5, was installed. Libvirt makes VM configuration and manipulation easy. Also, it is a fundamental component in well known cloud computing solutions like OpenStack, Eucalyptus, oVirt and RHEV, among others.

For SDN support networking, Open vSwitch version 2.1.2 was employed as the OpenFlow enabled virtual switch, i.e. the AVS'. As described before, the AVS connects to its instance of OpenFlow controller running the LAS component. The LAS component was developed over Pox controller[4] version carp. They were executed by the Python 2.7.5 interpreter. Pox was the only piece of software that wasn't installed from Fedora 19 repository, but from its GitHub repository[5].

[1] <http://www.linux-kvm.org>
[2] <http://wiki.qemu.org/KVM>
[3] <http://www.libvirt.org>
[4] <http://www.noxrepo.org/pox/about-pox/>
[5] <https://github.com/noxrepo/pox>

In turn, the NCS was developed in Python as well and was placed in one host. LAS' local cache table, which are populated by NCS, was implemented using SQLite 3.8.3. Besides Pox, all softwares were installed from official Fedora 19 repositories using the `yum` tool.

5.2 Topology

The five devices described in the Section 5.1 above were interconnected in a traditional datacenter network topology(Cisco Systems, Inc., 2007). This topology was chosen because it is well known, widely used and highly supported by big players in networking market.

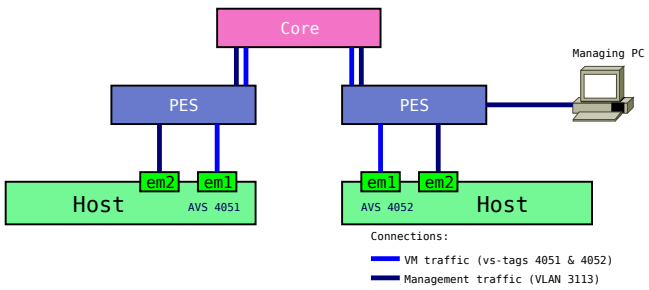
Figure 5.2 details the topology. Each server host has two network interfaces `em1` and `em2` associated with the two gigabit NIC. As shown in Figure 5.1, `em1` interface is added as an AVS' port acting as an *uplink* to the PES it is connected to. This interface has no associated IP address. In turn, the `em2` interface is connected to management network and has an IP address. Through this address SSH connections are open to access and manage the Linux operating system on the server host from the managing PC.

Furthermore, PES' are connected to a central core switch. In a real datacenter, there be at least two PES connected to server host as well as at least two core switches for the sake of redundancy and traffic distribution. But, for HotOM's evaluation purposes, the presented topology is enough. In regards of links, there are two different kinds of, one for VM's traffic and the other one for management traffic. The VM's traffic links (blue links) are configured as 802.1q *trunk* allowing *all* VLANs to go through, but untagged frames are prohibited to be forwarded. The other links are for management traffic (dark blue) and their related ports are *access ports* to VLAN 3113. This means that in these links there are only untagged frames. These are fairly simple configurations, on PES and core switches, that can be done using the CLI interface, scripting or even SNMP.

There are two vs-tags associated to AVS' executed by hosts. To one is designated vs-tag 4051 and to the other vs-tag 4052. The NCS is not running in a dedicated host, but in one of these two hosts. It stores and updates data in the LAS' local cache table through the management network.

Finally, to support practical experiments with the available physical components, 16 VMs were instantiated and equally distributed on both server hosts. Since they have 2GB of RAM, it was necessary to limit VM's number on each host to 8, because a VM needs about 256Mb of RAM to boot. Moreover, server host's CPUs allow the Linux kernel to deliver 4 virtual processors to hypervisor (KVM), which means that it is highly advisable to run tests with up to 4 VMs simultaneously because each VM would

Figure 5.2: Testbed topology



allocate one virtual processor at a time.

5.3 Evaluation

The evaluations were done primarily by taking VM's point of view. This approach was chosen because in a virtualized datacenter the majority of data movement occurs between VMs from the same tenant. Other traffics, like management, VM's deployment, migration and so forth, are a very small percentage of the total traffic.

In terms of metrics, the evaluations were done measuring throughput and *Round-Trip Time* (RTT). Throughput was measured using the iperf[6](HSU; KREMER, 1998) network measurement tool, while RTT was collected using the traditional `ping` Linux command. Each experiment were performed in a timeframe of 4 minutes.

Iperf can do measurements using TCP or UDP protocols, but in this evaluation TCP protocol was used. It must be run on both origin and destination VM. On the origin VM, iperf executes in *client mode*. On the destination VM, consecutively, it runs in *server mode*. The used options were plain simple. None options like TCP window size or maximum segment size were touched just because VM's Linux kernel has feasible default TCP parameters for server environments. Paramenters like TCP slow start, TCP window size ramping up, send and receive buffer size and so forth are self-adjusted on-demand by kernel during traffic. The only two options was `-f`, to display results in Mbps, and `-t`, to define measurement time's length. Figure 5.3 shows the iperf execution in both mode, where `IP` is the destination VM's IP address.

In addition to metrics above described, OpenFlow controller's CPU usage was also evaluated. Section 4.6 discussed about encapsulation/decapsulation performed by the controller and why it had to take place. These operation are CPU intensive, so server host's CPU usage was analyzed.

5.3.1 Network evaluation

The first set of performed analyses was about network throughput. A typical VN connects VMs placed in the same host or in different host. For example, suppose a VN connecting 2 VMs that can be allowed to run in 2 hosts, say $host_1$ and $host_2$. There are two possible different scenarios (*a*) all VMs running on the same host or (*b*) one VM running on each host. These two options are both limits in the space of possible throughput results, since in (*a*) no frame is address rewritten and vs-tag labeled neither forwarded to network, it is just connected directly by an AVS datapath forwarding table entry, while in (*b*) all frames are rewritten, vs-tag labeled, forwarded to the network, then their destinations are address rewritten again, removed the vs-tag and forwarded to VM. Thus, throughput experiments were done using both scenarios, named respectively "*local* communication" and "*remote* communication".

Figure 5.3: Iperf execution options

Server mode:
```
# iperf -f m -s
```

Client mode:
```
# iperf -f m -t 240 -c IP
```

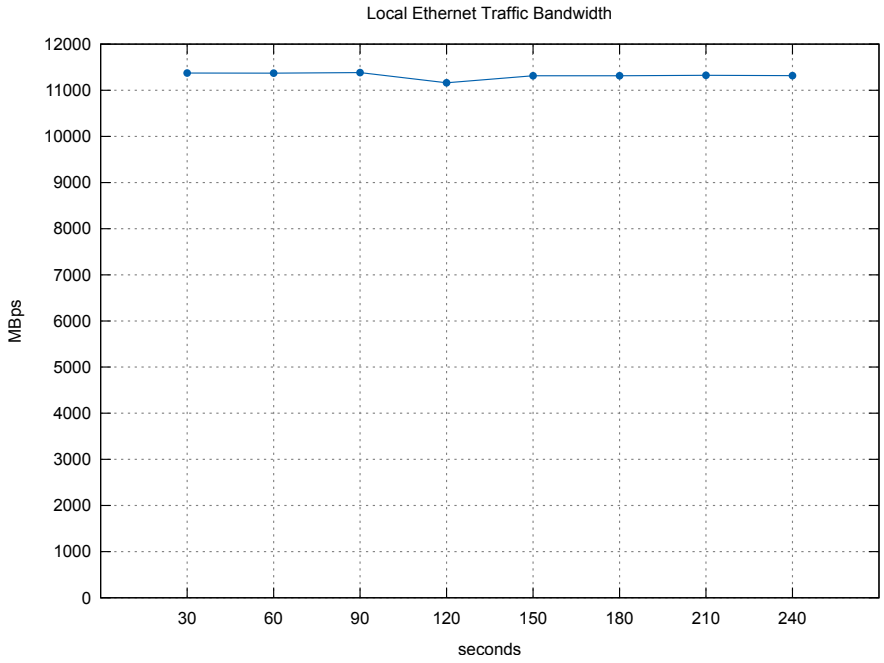
[6]<http://www.iperf.fr>

5.3.1.1 VLAN Ethernet Evaluation

At the beginning, one VN with two VMs was instantiated. This VN was created as a simple VLAN and AVS was not ruled by the OpenFlow Controller, but configured to make the VM's interfaces as an *access* port to VLAN. So, the AVS was acting as a traditional learning switch. This arrange is, in this document, simply called "Ethernet".

Figure 5.4 depicts the throughput results of an "Ethernet" while connecting two locals VMs, i.e., connected to the same AVS and executed by the same server host. The throughput is quite high, about 11500Mbps. The graph is plane, demonstrating that the hypervisor and the AVS can maintain flows without major interruptions or variations.

Figure 5.4: Local Ethernet throughput



The RTT experiment also was done and the results are shown on Figure 5.5. The first RTT was about 1.5ms due the delay associated to the ARP resolution. After this resolution, RTT were between 0.25 and 0.68 ms. Figure 5.12 also shows a bargraph of this experiment with minimum, maximum and average RTT time.

Moreover, a remote communication measurement was done. Figure 5.6 shows the results.

Again, the graph is quite plane, maintaining a throughput of about 930Mbps. Another equal experiment from one host to the other at "bare-metal" level (i.e. not in VM) has shown that there is a 939Mbps of host-to-host bandwidth. This means that the hypervisor imposes a very low overhead of about 1%.

In terms of RTT, results on remote evaluation, low values were also achieved as shown in Figure 5.7. Again, the first RTT is high, about 3ms, due the ARP resolution. After that the results are between 0.5ms and 1ms.

Those four results are mainly limited by both hypervisor and physical network (server's NIC, switches) performance. So, they are considered an upper bound that any other virtualization layer built

Figure 5.5: Local Ethernet RTT

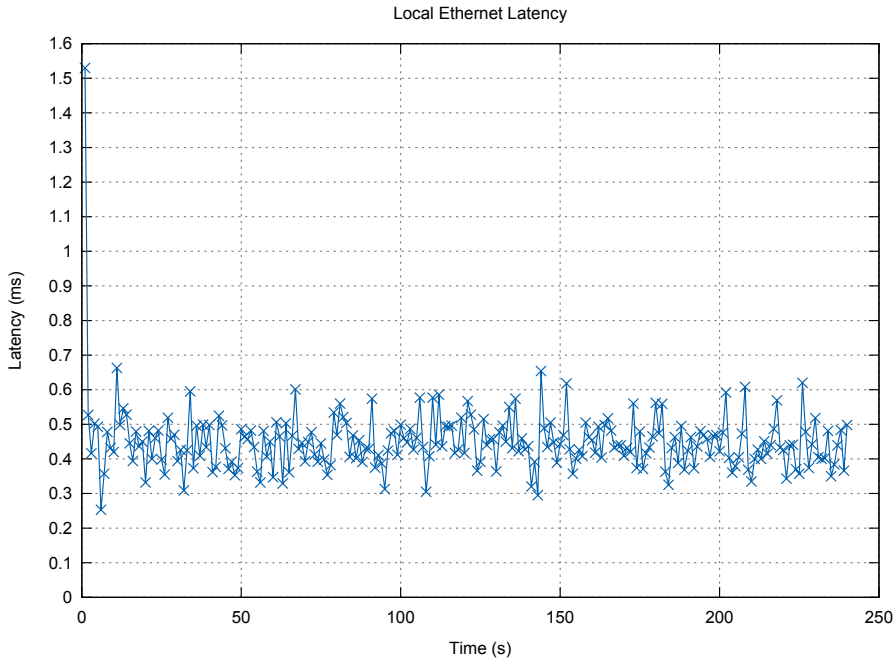
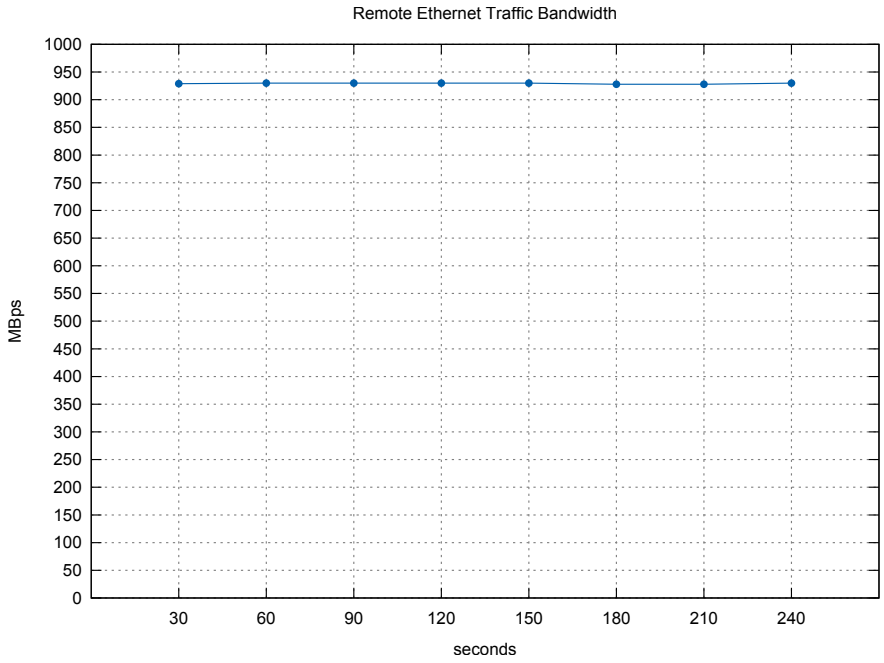


Figure 5.6: Remote Ethernet throughput

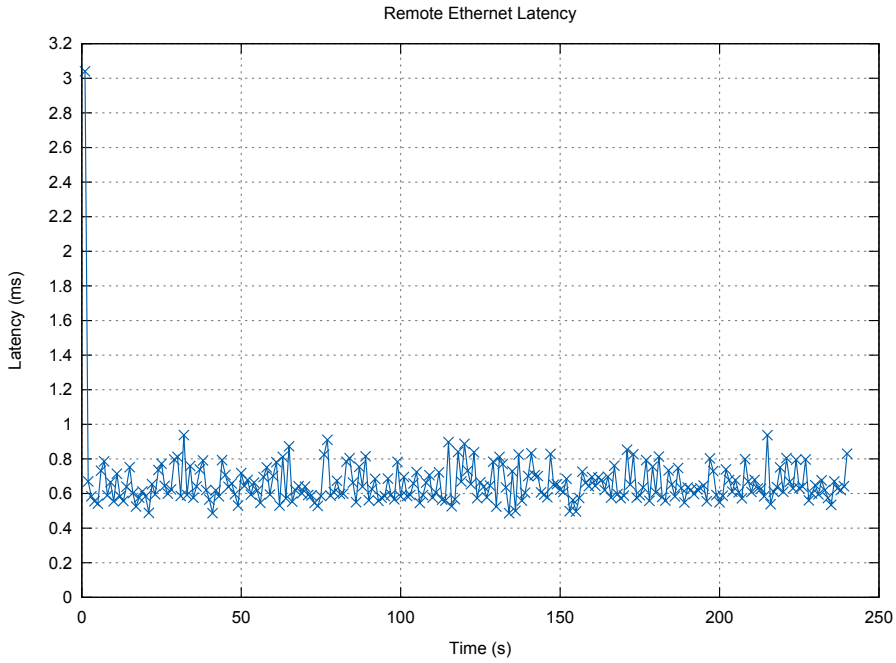


upon it will, in the best of the worlds, be *equal* with those results. A performance degradation is expected and will be evaluated.

5.3.1.2 HotOM Evaluation

After the initial collected data from "pure" VLAN Ethernet as discussed in Section 5.3.1.1, HotOM was executed. The AVS in each server host started to be controlled by their own OpenFlow

Figure 5.7: Remote Ethernet RTT



controller instance running the LAS component. Moreover, NCS was also started and populated in advance LAS local caches with its database entries (VNs, VMs and AVS).

Following the same steps from previous tests, the first experiment done was the local communication throughput. At this moment only one VN was instantiated, connecting two VMs. The results are shown in Figure 5.8. The achieved throughput was similar to local Ethernet communication, maintaining around 11500Mbps. This result is expected, since in HotOM the local traffic is routed by a flow table entry in AVS connecting the two VM's interfaces.

RTT measurements were also done and the results are plotted in Figure 5.9. The first RTT is about 17ms due the time of AVS sending the ARP request to the controller who finds out the IP associated with the MAC address in its local cache, crafting an ARP reply and sending it back to the VM. After that operation, RTT drops to in-between 0.25 and 0.68 ms.

Regarding HotOM remote throughput results, the achieved bandwidths were between 4.0Mbps and 4.3Mbps, as Figure 5.10 shows.

In turn, RTT results depicted in Figure 5.11 also has shown a comparable high values (between 24ms and 157ms) and variance. These high RTT and low remote throughput is due that traffic's packets must be diverted to controller for encapsulation/decapsulation process as discussed in Section 4.4.2.

For a better visualization on RTT measured results, Figure 5.12 shows the average RTT as a box level and the minimum and maximum RTT values as the vertical line endpoints. Ethernet local, HotOM local and Ethernet remote communications are in the same y-axis range, while HotOM remote is plotted with another range, about 10 times higher.

An evaluation were also done in terms of performance on various VN's instantiations. In the first experiment, all VNs were running in the same server host, thus configuring a *local communication*. Up to 4 VNs were possible to be created because the available servers are capable to run up to 8 VMs, whereas each VN connects two VMs. The results are shown in Figure 5.13. Just as demonstrated in Figure 5.4,

Figure 5.8: Local HotOM throughput

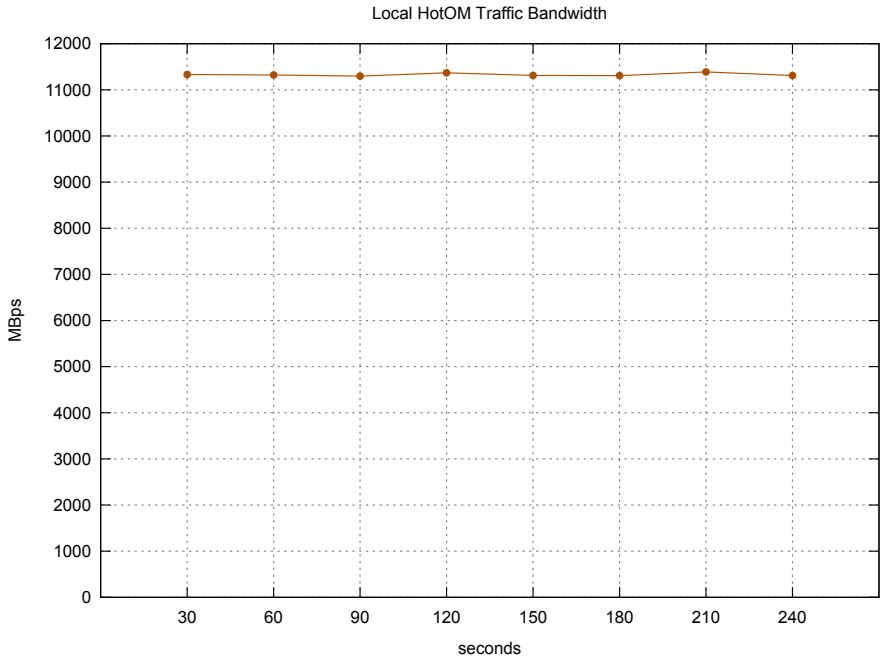
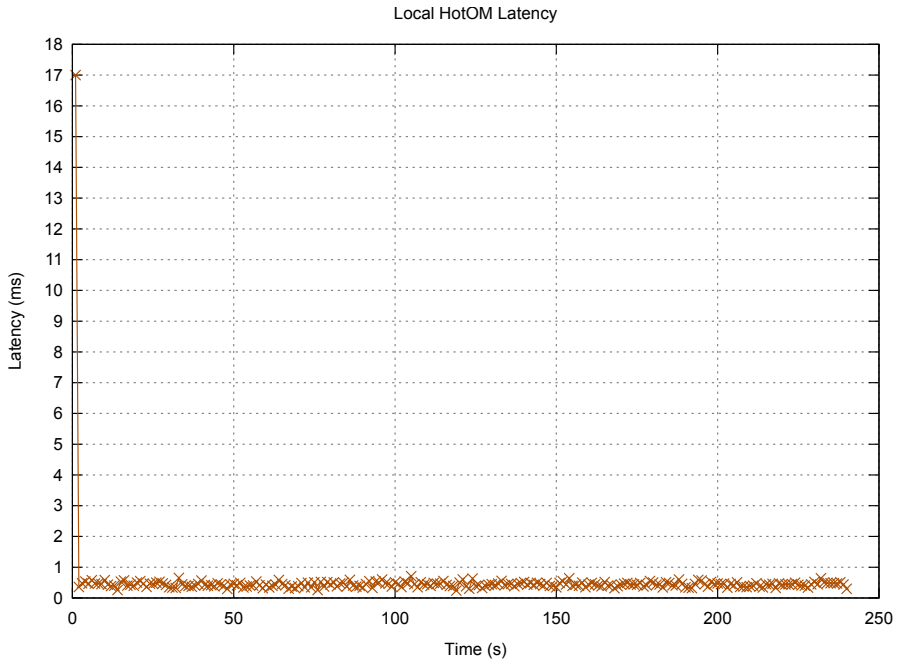


Figure 5.9: Local HotOM RTT



when there is only one VN running a throughput of around 12000Mbps was achieved. Ranging VN quantity from 1 to 4 shows a hyperbolic decay of each VN throughput, but the aggregate throughput is maintained quite constant. So, the hypervisor and AVS is capable of split bandwidth somewhat equally between VNs.

In turn, the remote communication test comprises ranging from 1 to 8 VNs where again each VN has two VMs. As expected, one VN achieves around 4.3Mbps of throughput, which decays hyperbolic while scaling VN's number up to 8. Figure 5.14 depicts the results. In this experiment, the controller is

Figure 5.10: Remote HotOM throughput

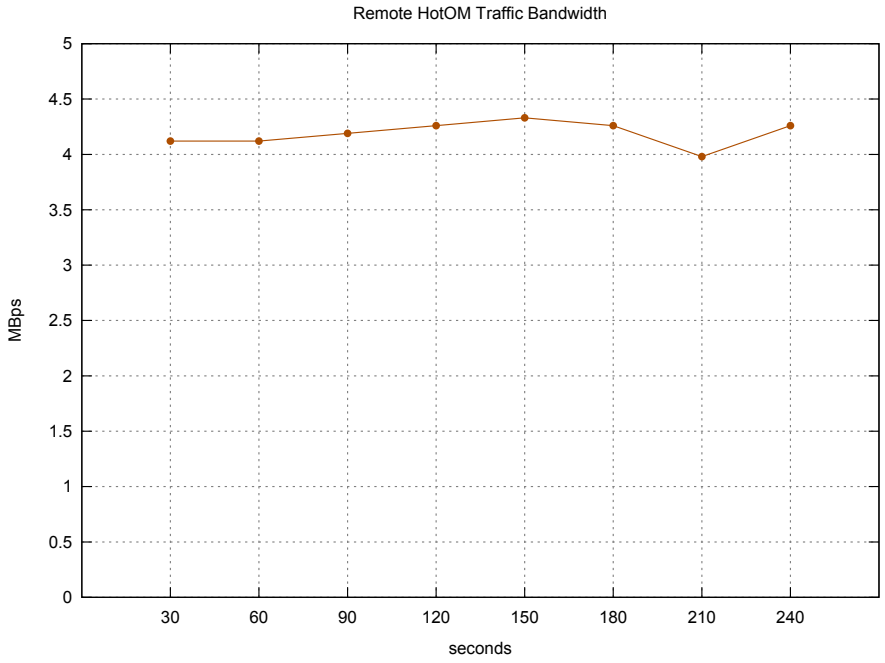
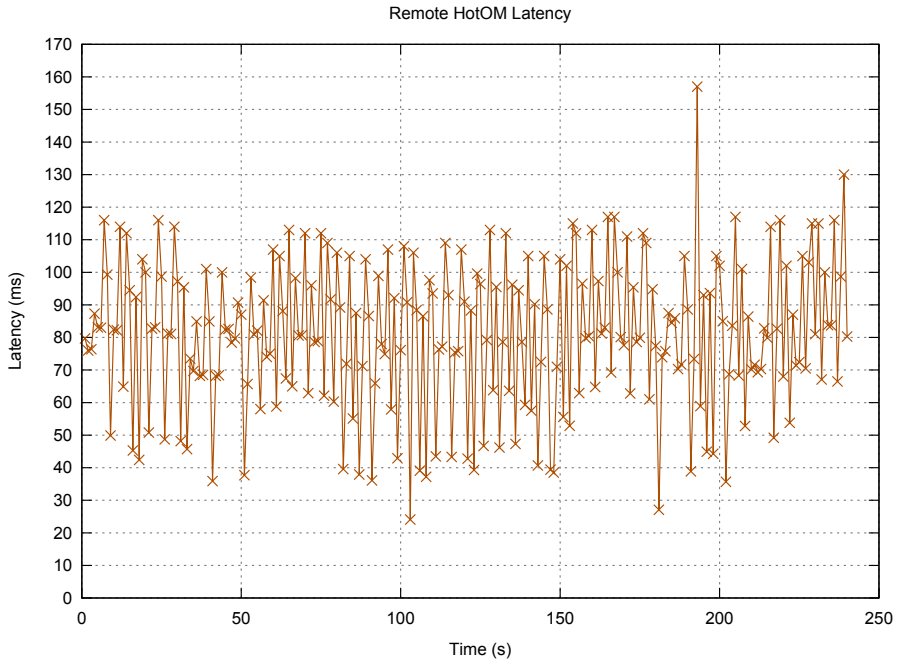


Figure 5.11: Remote HotOM RTT



also capable of share equally the bandwidth among VNs, in addition to hypervisor and AVS.

Figure 5.12: Min/max and average RTTs

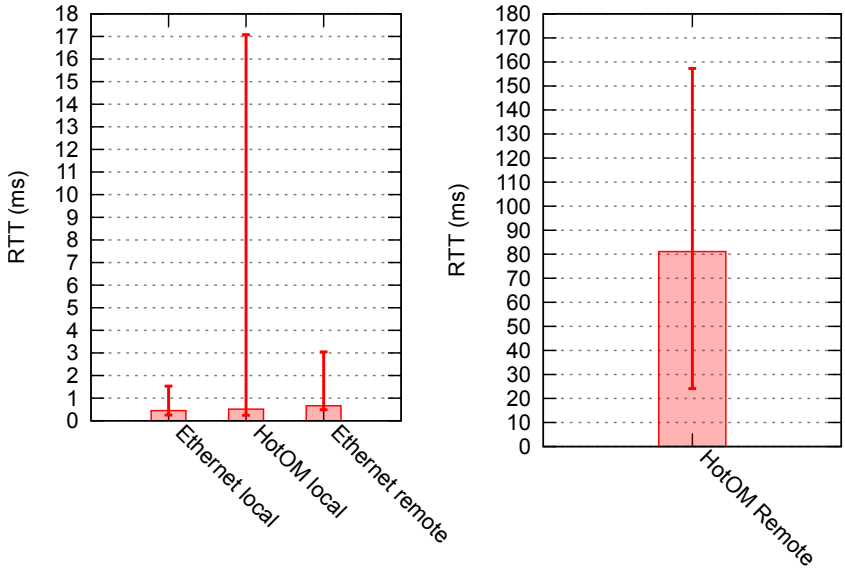


Figure 5.13: Multiple local VNs throughput

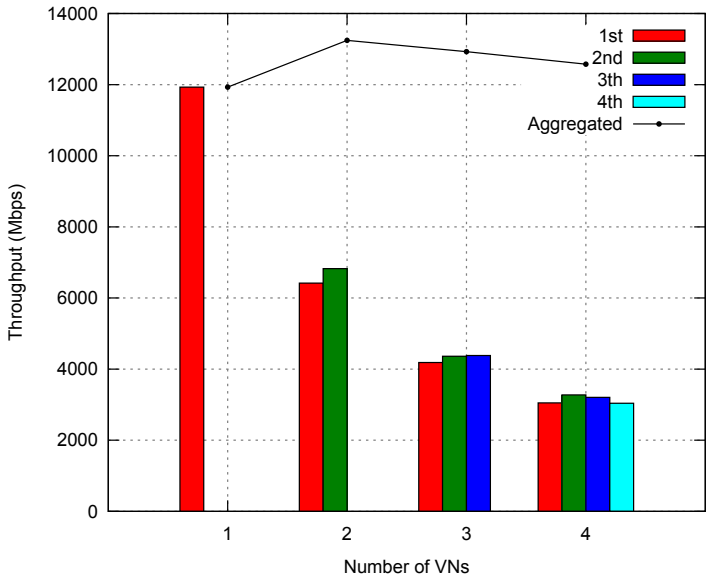


Figure 5.14: Multiple remote VNs throughput

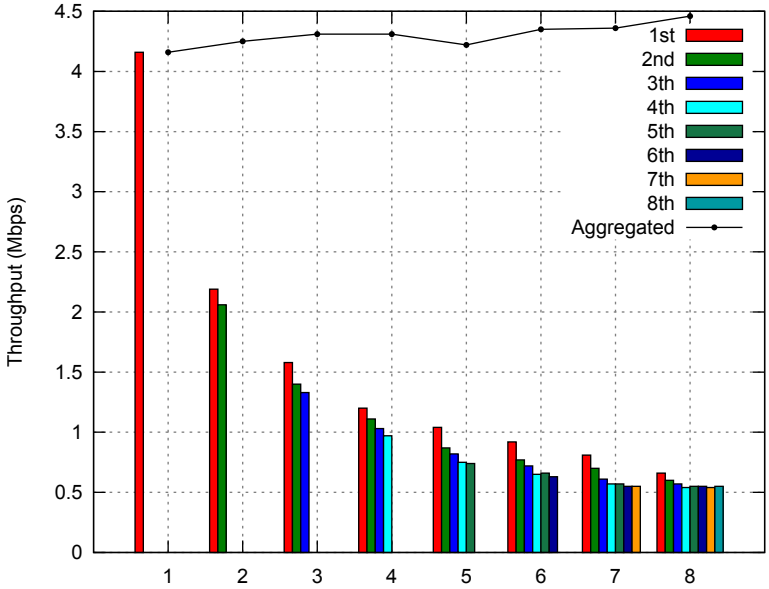


Figure 5.15: Ping and ARP commands for isolation tests

```
# ping -c2 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=11.87 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.338 ms
--- 10.0.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 2013ms
rtt min/avg/max/mdev = 0.339/6.10/11.87/5.76 ms

# arp -n
Address   HWtype  HWaddress           Flags Mask Iface
10.0.0.2  ether   00:00:00:00:00:02   C          eth0
```

5.3.1.3 Isolation

Isolation is a "must-have" feature in a multi-tenant datacenter. This property assures that a data traffic will not be accessible or "sniffed" from an unauthorized entity (tenant). HotOM builds its isolation towards upper layers from L2.5. Different tenants can use any IP address they want to, so the same IP could be assigned to many VMs. Tenants also have freedom on choosing any HotOM addresses to their VMs. Ethernet addresses will then be accordingly assigned by the VM managing system, as discussed in Section 4.3.4.

In this experiment, two virtual networks were instantiated. The first VN was VN_1, net_id `0xAABBCC`, that connected two VMs VM_{11} and VM_{12}. The second VN, VN_2, had its net_id `0XCCBBAA` and connected VMs VM_{21} and VM_{22}. To VMs were assigned, respectively, HotOM addresses `00:00:01` and `00:00:02`, and IP addresses 10.0.0.1 and 10.0.0.2.

At the same time, a ping was initiated from VM_{11} to VM_{12} and from VM_{21} to VM_{22}. The results of VM_{11} and VM_{21} are similar. Figure 5.15 demonstrates the ping output and the populated ARP table, showing that there is connectivity.

After the initial connectivity test, a packet sniffing test were performed. `tcpdump` was executed in VM_{21} and VM_{22} trying to capture frames from VN_1. No frames were captured. Then, the symmetric test was done, pinging VM_{12} from VM_{11} and trying to capture these frames in VN_2. Again, no frames were captured, demonstrating the complete isolation between both VNs.

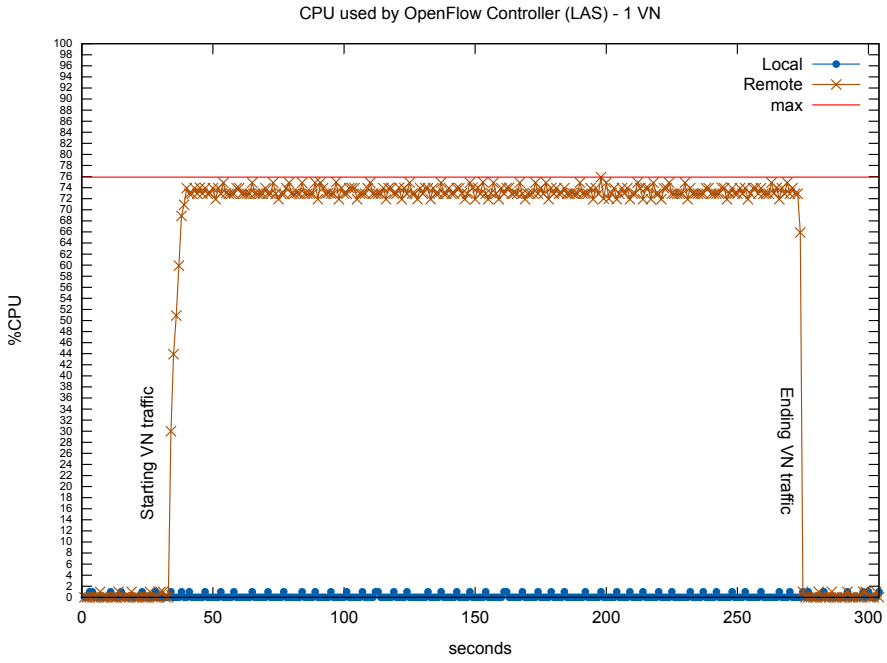
5.3.2 CPU usage evaluation

During the experiments, the controller CPU usage was measured to investigate how is the impact of Pox running on the same server host as hypervisor and VMs. If controller allocate too much CPU time, it diminish VM's performance and surely introduces delays on networking.

The first row of collecting CPU usage was done when the system was performing a one VN local communication and one VN remote communication (Figures 5.8 and 5.10). Figure 5.16 plots the results. The blue-with-dots line is the controller's CPU usage during a 300 seconds local communication. It shows that controllers operates lightly, using from 0% to 1% of CPU time. On the other hand, when a remote communication takes place, the controller demands a high CPU usage. At about 35 seconds the traffic between VMs on a remote communication starts and immediately the CPU usage ramps up to about 73% with a maximum of 76%. After 240 seconds, the VN traffic ceases and the CPU usage drops to nearly 0%.

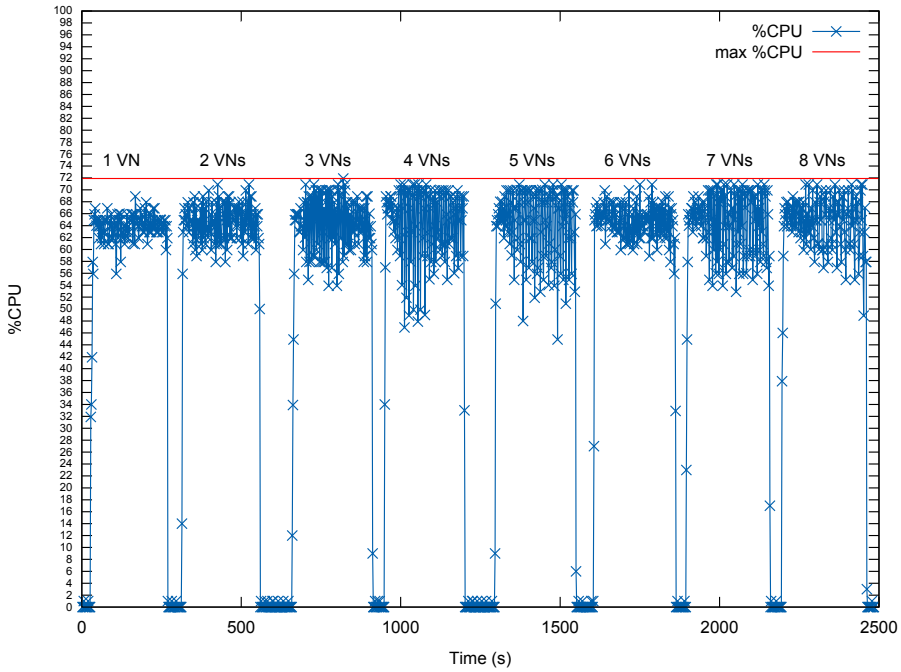
The second collected CPU usage was done during the experiments on scaling VN's number in a remote communication, which throughput results were shown on Figure 5.14. The rationale behind this

Figure 5.16: Controller's CPU usage - local vs remote



evaluation is to investigate how controller allocates CPU when multiple VNs are running. The CPU usage results, presented in Figure 5.17, show that no matter how many VNs are in place, the controller's CPU usage is barely the same, with an average about 64%. This average is a little bit lower than the one VN test (Figure 5.16) because not just 1, but 8 VMs were running and thus allocating more CPU resources, what left a little less to controller's usage.

Figure 5.17: Controller's CPU usage - multiple VN

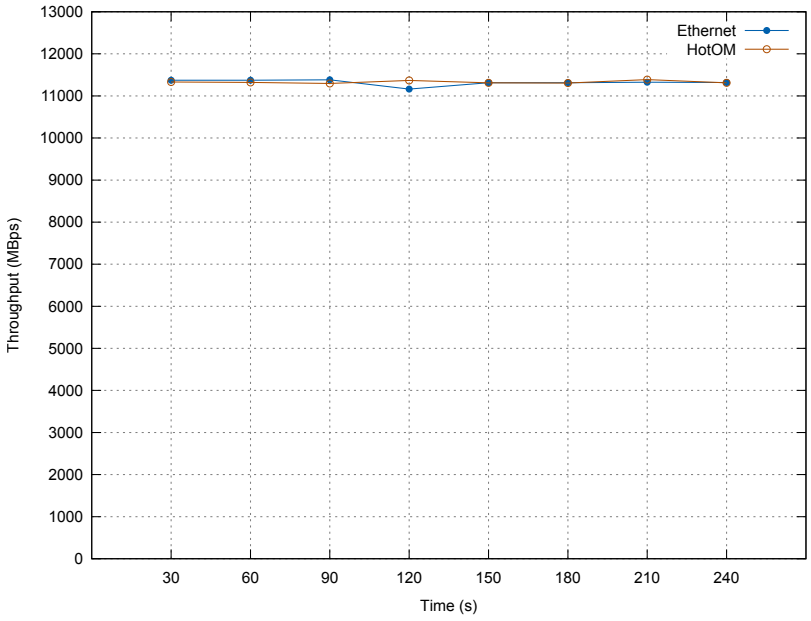


5.4 Discussion

Discussions worth to take place for better understanding the meaningfulness of results. Some numbers that were discovered when measuring metrics in some conditions seems, in a first moment, to be odd. However, with a deeper analysis its possible to realize on which areas HotOM must give more attention to relinquish the development stage and reach a mature, "ready-to-go" deployment status.

The first set of experiments evaluated the throughput and RTT during local communication. One VN was created interconnecting two VMs through both traditional Ethernet (using VLAN) *and* HotOM. Figure 5.4 and Figure 5.5 depicted the Ethernet results, while Figure 5.8 and Figure 5.9 the HotOM results. By comparing both throughput results, as can be seen in Figure 5.18, both results are close to each other.

Figure 5.18: Local Ethernet vs HotOM throughput

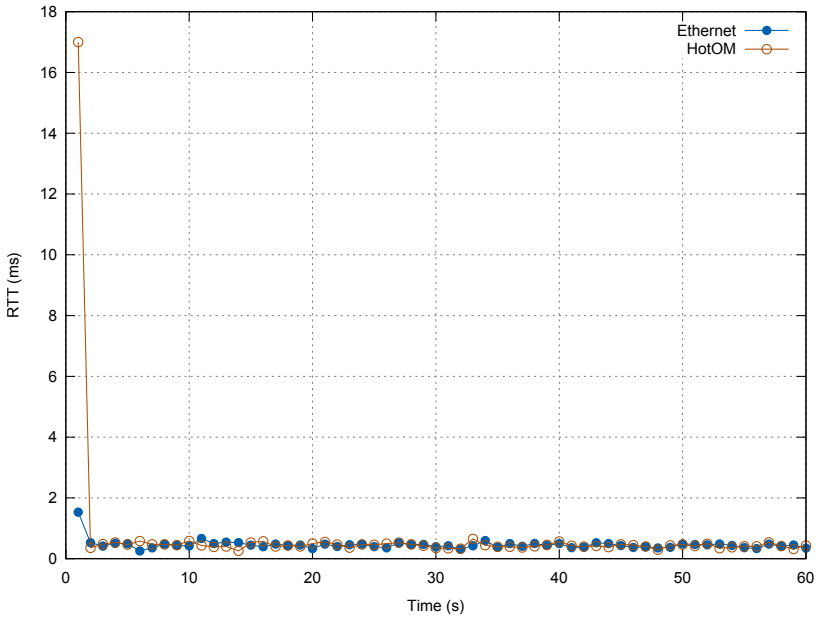


RTT results are also the same as shown in Figure 5.19, where this graph depicts the first 60 out of 240 seconds of the experiment for better detailing. The only difference is the first RTT that is higher in HotOM, due the ARP resolution being processed by the controller. This first high RTT is amortized in seconds, not imposing any penalty to communicating VMs. This inference can also be seen on the first and second bargraphs in left size of Figure 5.12, where the average RTT (bar level) are the same.

During the HotOM local communication, the controller does not use a considerable amount of CPU resources. Indeed, just the first packet of a flow is diverted to the controller and then it installs an OpenFlow rule in AVS' flow table connecting directly both VMs. In the employed AVS (Open vSwitch), the flow table component is part of the dataplane, which in implemented as a kernelspace code, i.e., a Linux kernel module(PFAFF et al., 2009). This approach ensures a huge shift in throughput performance with a low CPU usage, especially in data movement code and packet switching between network interfaces, no matter if they are logical or physical interfaces. Figure 5.16 shows with a "blue-and-dot" line that the controller uses from 0 to 1% of CPU time.

The second set of experiments also evaluated the throughput and RTT, but during remote communication. Again, one VN was created connecting two VMs using both Ethernet and HotOM. Ethernet

Figure 5.19: Local Ethernet vs HotOM RTT



remote communication throughput and RTT were already shown in Figures 5.6 and 5.7. HotOM results were also shown in Figures 5.10 and 5.11.

Also, the real throughput from "bare-metal" (i.e. outside the hypervisor, on the Linux kernel level) transversing the physical network had to be measured. With this results it would be able to determine if the hypervisor and virtual switch (AVS) are attenuating performance. The throughput measured was 939Mbps, but iperf computes the TCP *payload* bandwidth. Since the layering protocols Ethernet/IP/TCP imposes 3.7% of overhead, the real achieved bandwidth was about 974Mbps, very close to the theoretical 1000Mbps.

A direct comparison between Ethernet and HotOM remote communication's throughput can be seen in Figure 5.20. Due to its difference in order of magnitude, Remote Ethernet throughput is plotted using the left side y-axis, while HotOM results are using the right one. Indeed, Remote Ethernet throughput is about 232 times higher than HotOM.

In turn, Figure 5.21 shows a direct comparison between Ethernet and HotOM remote communication's RTT. As depicted in both plots, HotOM suffer a throughput performance degradation, and a higher RTT variance. These problems are introduced into HotOM due the inability of OpenFlow protocol and virtual switch's datapath to insert and/or remove arbitrary network headers to or from a frame. In other words, the entire encapsulate/decapsulate service is done by the controller, not in kernelspace datapath.

Using the controller for providing the encapsulate/decapsulate service is not performance-prone due a sequence of factors. When the traffic is originated in a VM to be sent throughout the network, a number of steps are performed. The following list enumerates the steps, where for the sake of simplicity controller's local cache is considered to be populated with origin and destination's VM information (port, HotOM addresses, destination vs-tag, etc):

1. The AVS receives the frame from the VM interface;

2. The AVS encapsulates the frame in an OpenFlow message (PacketIn) and sends it to the

Figure 5.20: Remote Ethernet vs HotOM throughput

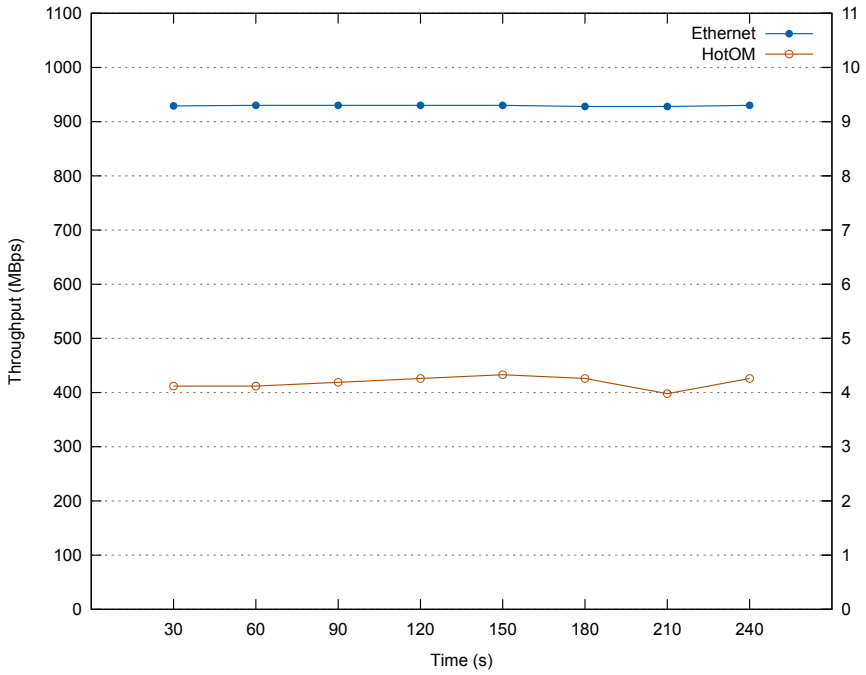
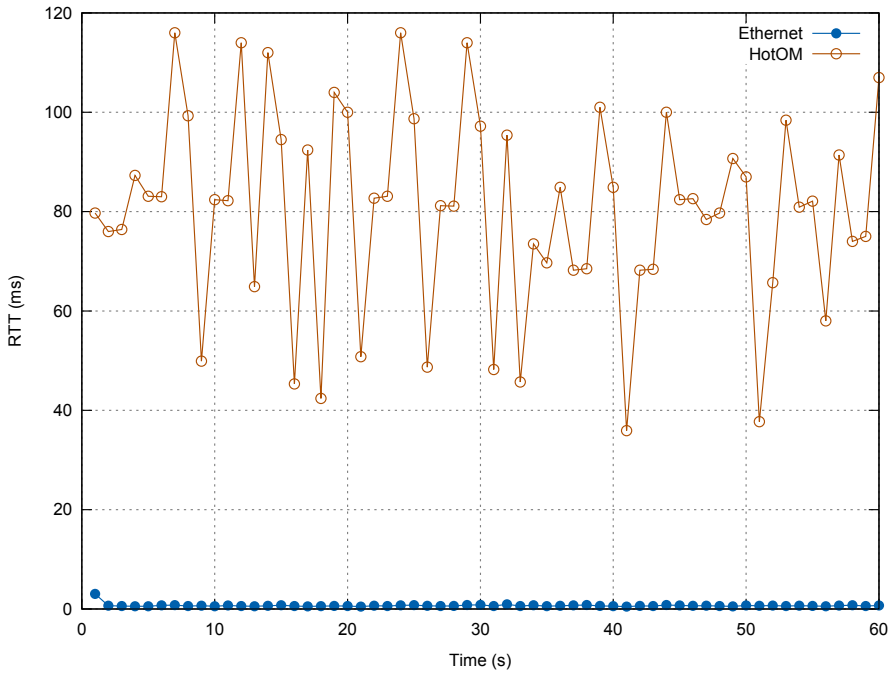


Figure 5.21: Remote Ethernet vs HotOM RTT



controller;

3. The controller parses the message extracting the input port to identify the origin VM and the L2 origin and destination addresses;

4. Controller searches in its local cache which AVS the destination VM is connected to, querying for destination vs-tag;

5. The HotOM header is created using origin and destination L2 addresses from VMs, also adding the type field accordingly to original EtherType;

6. A new L2 header is created by the controller using AVS' origin and destination Ethernet addresses, as well as placing the destination vs-tag in VLAN ID field;

7. The controller encapsulates the original frame payload into the HotOM header, then encapsulates it into the new L2 header;

8. Finally, the controller encapsulates the new frame in an OpenFlow message (PacketOut) and send it back to AVS, demanding it to switch the new frame out by the uplink port.

The decapsulate process is similar, basically in a reverse order, but there is an additional step when parsing a frame received from the network: extracting HotOM header together with L2 header and creating Ethernet addresses from HotOM address.

It worth to note the OpenFlow controller is a userspace process managed by Linux kernel. During the encapsulation/decapsulation service, many data (frames, headers fields, payload, etc) are copied back and forth between kernelspace buffers, used by virtual switch datapath, and userspace buffers, used by controller (as a Linux process). Also, many copies and computations are done within the controller itself. Moreover, Python interpreter do not take advantage of multithreading neither scale well in multiprocess environment due its Global Interpreter Lock[7], imposing a Python code to be virtually single-threaded. This fact can be confirmed in Figure 5.17, where no matter how many VNs exists, almost the same controller's CPU usage is measured. So, the bottleneck is the number of frames per second that the controller can manipulate. Other important factor that imposes controller performance limitations and higher and unpredictable latency is that being a userspace process, the Linux kernel does many context switches between userspace and kernelspace, also putting the controller in sleeping status and back in execution when needed. The rightmost bargraph from Figure 5.12 depicts the high RTT values, average and variance. Section 6.3 discuss possible researches and implementations that can overcome this performance limitation.

5.5 Protocol overhead

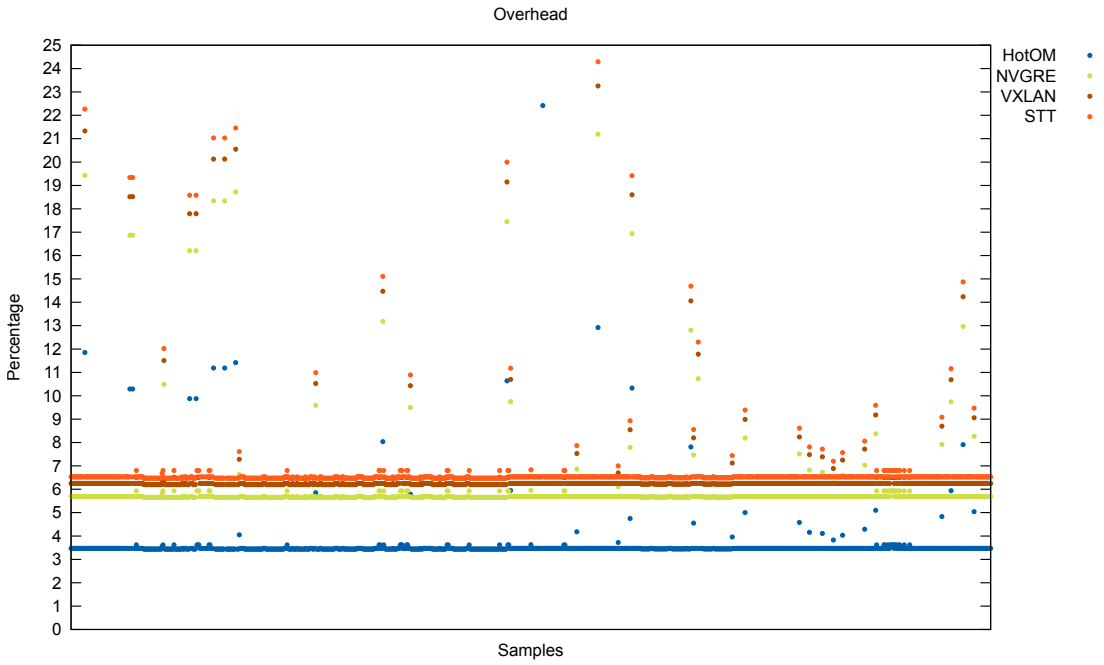
Protocol overhead is a traffic characteristic that might be as minimum as possible. Network protocol designers struggle to diminish overhead in order to allow more useful data (application layer information) to be carried in the same, upper bounded size, Ethernet frame. The lower overhead, the lower protocol fragmentation and higher data payload size. As discussed in Section 2.3.4, efforts are being done to create virtual networks with reduced, or even none, overhead.

A simulation were done in order to evaluate protocol overhead imposed by HotOM facing others "bleeding-edge" virtualized datacenter network protocols - STT, VXLAN and NVGRE. A tracing of HTTP and HTTPS bidirectional packets was captured for the entire day using the `tcpdump` tools. The target website was the author's employer one[8]. The application data payload was extracted from each packet

[7] <http://www.dabeaz.com/python/GIL.pdf>
[8] <http://www.prt6.mpt.mp.br>

sample in tracing, then the appropriated headers, from L2 to above, were added to it respecting each protocol definition. The output was a new trace just like if these packets were captured from a HotOM, STT, VXLAN or NVGRE network. Finally, the overhead was calculated and plotted in Figure 5.22.

Figure 5.22: HotOM vs VXLAN/NVGRE/STT overhead



The better case, HotOM, applies in most of the cases about 3.47% of protocol overhead, while the worst case, STT, 6.53%. In turn, VXLAN imposes 6.25% and NVGRE imposes 5.69% overhead. Indeed, these result shows that HotOM's overhead is about 47% *less* than STT, 44% *less* than VXLAN and 41% *less* than NVGRE. This fact signals that more useful payload is moved within the same bandwidth, as well as lower power consumption is required for the same payload, since a reduced energy quantity is used for protocol itself.

5.6 Comparison between proposals

No proposal deals with every network requirements neither fits in all user cases. So, each one exposed in Chapter 3 has its particular strengths and weakness. Some characteristic are similar or even based on each other. Others are completely different. This section discusses a comparison between the visited proposals to HotOM.

5.6.1 Features

Within a vast myriad of network virtualization characteristics, a number of features were part listed to be primary targets for comparison. Besides *performance*, which was designated as guidance to validate real deployability on actual state of proposal's development, these features were chosen for better facing HotOM because they are its main challenges.

Those features are:

- **Legacy L2 switches usage:** This feature demonstrates the proposal's ability on employment of legacy plain L2 switches, i.e, not L3 and above, whose are the simpler thus less expensive network devices available.
- **CAM pressure:** This feature disclose if the proposal exposes legacy physical switches to a huge number of VM assigned MAC addresses. An exposure to a high number of MAC addresses demands switches to frequently flood unknown MAC address, remove older entries from its forwarding table and insert new ones, causing scalability issues.
- **L2 abstraction:** A total L2 abstraction provides a proper way of virtual network's migration from two totally different physical datacenter's network. Moreover, this abstraction also gives tenants an opportunity of define their own data link address schemes.
- **Protocol overhead:** The more a protocol overhead in a network, the more waste of bandwidth to transport useful data placed in payload. For this reason the protocol overhead should be as less as possible.
- **Scalability:** This is a very important feature and nowadays is considered as a "must-have". I realizes the likeliness of a proposal to be deployed within a real virtualized datacenter. Newer virtualized datacenter's requirements demands that a high number of VNs (hundred of thousands or more) might be instantiated seamlessly.
- **Programmability:** Having a potentiality of applying dynamic actions on flows within a VN creates an opportunity to tenants to use unpreceded network services with a remarkable control over virtual network behavior. By exposing an API, tenants could explore this programmability on their VN.
- **Performance:** Data movement with high throughput and low latency is extremely desired in real network deployment. However, newer proposals can be in a state that are not already mature in this aspect. So, this feature can be used as a metric if a proposal is ready to be used in a real-world virtualized datacenter.

5.6.2 Analysis

This section discuss the compliance of each proposal to the target features considered in Section 5.6.1.

- **Legacy L2 switches usage:** Desired result: *yes*
  - **HotOM**: By using VLAN ID tags for routing and placing AVS' MAC addresses in L2 header, HotOM demands only L2 switches. Result: *yes*;
  - **Trellis**: It creates GRE tunnels between endpoints that lays over L3 protocol, but Trellis authors does not explore inter-subnet routing, so it uses L2 switches. Result: *yes*;

- **VL2**: By encapsulating IP-in-IP, VL2 do not support L2-only switches. Result: *no*;
- **PortLand**: It uses OpenFlow-enabled switches throughout the entire topology. Result: *no*;
- **NetLord**: By encoding tenant information in L3 header, NetLord can't be deployed using plain L2 switches. Result: *no*;
- **CrossRoads**: Just like PortLand, it uses only OpenFlow-enabled switches. Result: *no*;
- **NVP**: By employing overlay networks, NVP needs L3 switches. Result: *no*;

- **CAM pressure:** Desired result: *low*
  - **HotOM**: By exposing only AVS' MAC addresses to switches, HotOM imposes a low CAM pressure. Result: *low*;
  - **Trellis**: By exposing endpoint's MAC addresses to switches, Trellis performs a low CAM pressure. Result: *low*;
  - **VL2**: Network core forwards based on LA addresses, so imposing a low CAM pressure. Result: *low*;
  - **PortLand**: Each VM has a PMAC address, so switches must deal with them. Result: *high*;
  - **NetLord**: L2 headers have the origin and destination switches' MAC address, so a low CAM pressure is done. Result: *low*;
  - **CrossRoads**: Similar to Portland, each VM has a PMAC address. Result: *high*;
  - **NVP**: By using tunneling, network core switches deals only with endpoint's MAC addresses. Result: *low*;

- **L2 abstraction:** Desired result: *yes*
  - **HotOM**: By using the HotOM Protocol Header, HotOM deliver a L2 abstraction to tenants. Result: *yes*;
  - **Trellis**: It encapsulates the entire VM's frame in a GRE tunnel, so Trellis abstracts the L2 network. Result: *yes*;
  - **VL2**: It assigns a AA IP address to each server host, so VL2 do no abstracts L2 addresses. Result: *no*;
  - **PortLand**: By translating back and forth PMACs to AMACs addresses, PortLand allows tenants to use their own L2 address space. Result: *yes*;
  - **NetLord**: Similar to Trellis, NetLord encapsulates the entire VM's frame in a L3 packet. Result: *yes*;
  - **CrossRoads**: Similar to PortLand, CrossRoads translates PMACs to AMACs, allowing L2 abstraction. Result: *yes*;

        - **NVP**: By tunneling the VM's frame, NVP abstracts L2 network to tenants. Result: *yes*;

- **Protocol overhead:** Desired result: *low*

    - **HotOM**: The HotOM Protocol Header applies low overhead. Result: *low*;
    - **Trellis**: It encapsulates the VM's frame in a GRE packet, consequently in a L3 and L2 frames. Result: *high*;
    - **VL2**: Employs IP-in-IP encapsulation. Result: *high*;
    - **PortLand**: By just translating L2 addresses, PortLand does a low overhead. Result: *low*;
    - **NetLord**: It encapsulates the VM's frame in a L3 and L2 frames. Result: *high*;
    - **CrossRoads**: Similar to PortLand, it does low overhead. Result: *low*;
    - **NVP**: By the use of overlay networks, the entire VM's frame is encapsulated in L3 and L2 frames. Result: *high*;

- **Scalability:** Desired result: *high*

    - **HotOM**: It addresses 2^{24} VNs. Result: *high*;
    - **Trellis**: It scales up to 60 VNs. Result: *low*;
    - **VL2**: By addressing services with IP (AA), and employing L3 switches, it can deal with a high number of services. Result: *high*;
    - **PortLand**: It allows expanding the network by adding more PoDs. Result: *high*;
    - **NetLord**: It addresses 2^{24} tenants, each one with multiple L2 network. Result: *high*;
    - **CrossRoads**: Similar to PortLand, it scales the network by adding PoDs. Result: *high*;
    - **NVP**: Its overlay protocols (VXLAN, NVGRE, STT) can address at least 2^{24} VNs. Result: *high*;

- **Programmability:** Desired result: *yes*

    - **HotOM**: By using OpenFlow to define flow behavior, HotOM allows network programmability. Result: *yes*;
    - **Trellis**: Trellis implementation just creates a mesh of virtual links (tunnels), without enabling programmability. Result: *no*;
    - **VL2**: It just encapsulates IP-in-IP, not conceding any dynamic behavior. Result: *no*;
    - **PortLand**: By using OpenFlow, PortLand allows programmability. Result: *yes*;

- **NetLord**: Its implementation is "ossified", not allowing changes on how it works. Result: *no*;
- **CrossRoads**: Similar to PortLand, by using OpenFlow it allows programmability. Result: *yes*;
- **NVP**: By employing a SDN cluster, as well as defining a network programming language (nlog), NVP allows programmability. Result: *yes*;

- **Performance:** Desired result: *high*
  - **HotOM**: By doing L2.5 shim header encapsulation/decapsulation in OpenFlow controller, HotOM deliver a low performance. Result: *low*;
  - **Trellis**: It uses a slow bridge implementation that imposes a low performance. Result: *low*;
  - **VL2**: By spreading traffic using VLB, VL2 achieves a high performance. Result: *high*;
  - **PortLand**: It creates a huge number of flow entries rules in OpenFlow-enabled switches, what direct impacts in performance. Result: *low*;
  - **NetLord**: By using NLA and spreading traffic in network's core using SPAIN, NetLord achieves a high performance. Result: *high*;
  - **CrossRoads**: Similar to PortLand, it creates a huge number of OpenFlow rules. Result: *low*;
  - **NVP**: By precomputing all virtual network topology, creating overlay tunnels and using ECMP to scatter traffic, NVP achieves a high performance. Moreover, when using STT there is the TSO/LRO advantage. Result: *yes*;

With all described above, Table 5.1 summarizes the comparison among discussed proposals.

Table 5.1: Proposals comparison

	HotOM	Trellis	VL2	PortLand	NetLord	CrossRoads	NVP
L2 switches	**yes**	**yes**	no	no	no	no	no
CAM pressure	**low**	**low**	**low**	high	**low**	high	**low**
L2 abstraction	**yes**	**yes**	no	**yes**	**yes**	**yes**	**yes**
Protocol overhead	**low**	high	high	**low**	high	**low**	high
Scalability	**high**	low	**high**	**high**	**high**	**high**	**high**
Programmability	**yes**	no	no	**yes**	no	**yes**	**yes**
Performance	low	low	**high**	low	**high**	low	**high**

Note: Desirable results are in **green bold face** and in red otherwise.

5.7 Chapter summary

This chapter presents the measured results of a HotOM initial implementation in an available testbed. Two metrics were used as major target of the experiments, namely throughput and RTT, and another one as support for explaining some results - CPU usage.

The evaluation was performed by a comparison between a traditional VLAN and HotOM networks. Within all possible network organization that can be created, two can are considered as boundaries in terms of throughput and RTT: local and remote communications. So those were chosen to be evaluated.

Results were shown that there is no difference between VLAN and HotOM regarding local traffic. In the other hand, the actual implementation of HotOM has a performance penalty in handling remote traffic because the virtual switch, on behalf of the controller, cannot insert or remove arbitrary protocol headers. So, it is necessary to divert *every* frame in a flow to the controller, what is a low performance operation. Discussions were done about the results, as well as deeper analysis on how virtual switch datapath and controller operates.

Moreover, a proof about tenant's isolation was done. Although the test of trying to capture frames from other VN are fairly simple, it is effective. In addition, an evaluation about how HotOM performs in terms of overhead was done. It shows that HotOM has a great advantage over others virtualized datacenter network proposal in such arena.

Finally, a direct comparison between HotOM and "bleeding-edge" proposals over many features has proven that it is well positioned and its real-world deployment depends on resolving just one feature: performance.

6
Conclusion

This chapter concludes this dissertation by summarizing the relevant points that were addressed. Section 6.1 enlightens the most significant parts of this dissertation. Section 6.2 highlights the contributions done by this work. Finally, Section 6.3 mentions future paths that can be taken to push HotOM further.

6.1 Summary

This dissertation presents HotOM, a novel approach to allow virtualized datacenter's networks to be able of hosting a massive number of VNs and tenants, while using legacy plain L2 switches. This facts eases the acceptance, deployment or migration from former network's technologies to HotOM.

By using the VLAN ID tag (vs-tags) for routing decision within network's core, HotOM takes advantage on the high performance CAM lookups procedures, reaching a low latency packet switching. Furthermore, the introduction of the L2.5 shim header - HotOM Protocol - provides tenants a complete L2 abstraction, while alleviates CAM pressure. The overall network performance benefits from the low CAM pressure, since switches are less prone to flood frames throughout the entire network when the forwarding table is full.

Going further on HotOM Protocol Header, it was designed to bring scalability to the datacenter network. It allows a datacenter to host up to 16.8 million VNs, while each VN in turn can interconnect up to 16.8 million of VMs. A tenant can own one or more VNs. This shim header also contributes to lower overhead. As discussed in Section 5.5, it imposes about a half overhead than STT, for instance.

Chapter 3 visited the most prominent datacenter network proposals that came up on early years, and they were deeply discussed. HotOM was directly compared to them in Section 5.6, along with an analysis about their strengths and weaknesses.

Moreover, most of the Chapter 5 was dedicated to the real, current implementation of HotOM. An extensive experimentation, data collecting, plotting and comparison with "pure" VLAN Ethernet were done. Finally, Section 5.4 discussed all collected results.

6.2 Contributions

This work contributed in several ways to network community.

First, a extensive review about “bleeding-edge” virtualized datacenter’s network was done, allowing the reader to quickly learn about them.

Second, by proposing HotOM, a myriad of contributions was done. Using the VLAN ID tag for routing purposes opens an opportunity for development of a even simpler, thus faster and cheaper, network switch. Also, by demanding only L2 switches, legacy datacenters can step further and start to offer IaaS to their clients, surely increasing profits.

Third, HotOM attracts attention to the fact that network engineers must be aware about high protocol overheads and their drawbacks. Indeed, HotOM employs something like “stackable” networks, where virtual L2 domains sit over a physical L2 infrastructure, instead of overlaying L2 over L3 (and then L2) networks. Thus, much less overhead takes place.

Finally, OpenFlow was designed to deal with fixed size, predefined protocol headers. But HotOM shows that it is possible to introduce an arbitrary header in any desired position, by diverting the flow to controller and then manipulate it.

6.3 Future works

HotOM establishes a ground that many researches can go further and improve it. As discussed in Section 5.4, the apparently only drawback is performance. So, one possible future work can be tackling with this aspect by adding HotOM support directly to virtual switches (Open vSwitch), on its kernelspace datapath just like VXLAN and NVGRE authors did. This development might vanish HotOM’s performance limitations. Even more, a researcher can investigate a newer HotOM implementation porting it directly to a hardware platform, off-loading from operational system kernel the task of encapsulate/decapsulate packets in L2.5 shim header. NetFPGA(NAOUS et al., 2008) would be a natural choice for such investigation.

Finally, another direction that can be taken is pushing HotOM’s programmability to it limits, exploring every single opportunity. A researcher, for instance, can investigate an implementation of logical datapaths over HotOM VNs, delivering virtual network services like switching, routing, load-balancing and firewalling to tenants. So, a tenant could instantiate any service it needs and program LAS to perform these services. In turn, LAS might arbitrate over service requests from many tenants.

References

GOOGLE. D.C. Abts, P.M. Klausler, H. Liu, M. Marty e P. Wells. **Systems and methods for energy proportional multiprocessor networks**. 2013. US Patent 8,601,297, 18 jun. 2010, 3 dez. 2013.

BARI, M. F. et al. Data Center Network Virtualization: A Survey. **IEEE Communications Surveys & Tutorials**, v. 15, n. 2, p. 909–928, maio/ago. 2013.

BHATIA, S. et al. Hosting Virtual Networks on Commodity Hardware. **Georgia Tech. University., Tech. Rep. GT-CS-07-10**, jul. 2008.

BI, J. et al. Dynamic Provisioning Modeling for Virtualized Multi-tier Applications in Cloud Data Center. In: 3RD INTERNATIONAL CONFERENCE ON CLOUD COMPUTING, 3., 2010, Miami,FL. **Anais...** Miami,FL: IEEE, 2010. p. 370–377.

BIEDERMAN, E. W. **An Introduction a Path for Merging Network Namespace Work**. Linux Weekly News, 2007. Disponível em: <http://lwn.net/Articles/219597/>. Acesso em: 15 dez. 2014.

CASADO, M. et al. Ethane: Taking Control of the Enterprise. **ACM SIGCOMM Computer Communication Review**, v. 37, n. 4, p. 1–12, ago. 2007.

CASADO, M. et al. Virtualizing the network forwarding plane. In: WORKSHOP ON PROGRAMMABLE ROUTERS FOR EXTENSIBLE SERVICES OF TOMORROW, 2010, Philadelphia. **Anais...** New York, NY: ACM, 2010. p. 8:1–8:6.

Cisco Systems, Inc. **Data Center: Load Balancing Data Center Services SRND**. 2004. Disponível em: <https://learningnetwork.cisco.com/servlet/JiveServlet/downloadBody/3438-102-1-9467/cdccont_0900aecd800eb95a.pdf>. Acesso em: 1 maio 2015.

Cisco Systems, Inc. **Cisco Datacenter Infrastructure 2.5 Design Guide**. 2007. Disponível em: <http://www.cisco.com/application/pdf/en/us/guest/netsol/ns107/c649/ccmigration_09186a008073377d.pdf>. Acesso em: 1 maio 2015.

CPqD; FERNANDES, E. L. **OpenFlow 1.3 Software Switch**. 2015. Disponível em: <http://cpqd.github.io/ofsoftswitch13/>. Acesso em: 16 fev. 2015.

DALLY, W.; TOWLES, B. **Principles and Practices of Interconnection Networks**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN 0122007514.

EBERT, R. J.; GRIFFIN, R. W. **Business Essentials**. 10th. ed. New Jersey: Prentice Hall, 2014. ISBN 0133454428.

FARINACCI, D. et al. **Generic Routing Encapsulation (GRE)**. RFC Editor, 2000. Disponível em: <https://tools.ietf.org/html/rfc2784>. Acesso em: 15 dez. 2014.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The Road to SDN. **Queue**, ACM, v. 11, n. 12, p. 20, 2013.

GREENBERG, A. et al. The Cost of a Cloud: Research Problems in Data Center Networks. **SIGCOMM Computer Communication Review**, ACM, New York, NY, v. 39, n. 1, p. 68–73, 2008.

GREENBERG, A. et al. VL2: A Scalable and Flexible Data Center Network. In: SIGCOMM, 2009, Barcelona. **Anais...** New York, NY: ACM, 2009. p. 51–62.

GUENENDER, S. et al. NoEncap: Overlay Network Virtualization with no Encapsulation Overheads. In: SIGCOMM SYMPOSIUM ON SOFTWARE DEFINED NETWORKING RESEARCH, 1., 2015, Santa Clara, CA. **Anais...** New York, NY: ACM, 2015. p. 9:1–9:7.

HANDLEY, M.; HODSON, O.; KOHLER, E. XORP: An Open Platform for Network Research. **SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 33, n. 1, p. 53–57, jan. 2003.

HSU, C.-H.; KREMER, U. Iperf: A framework for automatic construction of performance prediction models. In: WORKSHOP ON PROFILE AND FEEDBACK-DIRECTED COMPILATION, 1998, Paris. **Anais...** Pennsylvania: Citeseer, 1998.

IEEE Computer Society. **IEEE Bridges and Virtual Bridged Networks**. 2014. Disponível em: <https://standards.ieee.org/findstds/standard/802.1Q-2014.html>. Acesso em: 15 dez. 2014.

IEEE Computer Society. **IEEE Link Aggregation**. 2014. Disponível em: <https://standards.ieee.org/findstds/standard/802.1AX-2014.html>. Acesso em: 15 dez. 2014.

IEEE Standards Association. **EtherType Public List**. 2015. Disponível em: <http://standards.ieee.org/develop/regauth/ethertype/eth.txt>. Acesso em: 1 jun. 2015.

KERRISK, M. **Namespaces in operation**. Linux Weekly News, 2013. Disponível em: <https://lwn.net/Articles/531114/#series_index>. Acesso em: 15 dez. 2014.

KODIALAM, M.; LAKSHMAN, T. V.; SENGUPTA, S. Efficient and Robust Routing of Highly Variable Traffic. In: WORKSHOP ON HOT TOPICS IN NETWORKS, 3., 2004, San Diego, CA. **Anais...** New York, NY: ACM, 2004.

KOPONEN, T. et al. Network Virtualization in Multi-tenant Datacenters. In: NETWORKED SYSTEM DESIGN AND IMPLEMENTATION, 11., 2014, Seattle, WA. **Anais...** Berkeley, CA: USENIX, 2014. p. 203–216.

KOPONEN, T. et al. Onix: A Distributed Control Platform for Large-scale Production Networks. In: OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, 9., 2010, Vancouver. **Anais...** Berkeley, CA: USENIX, 2010. v. 10, p. 1–6.

KREUTZ, D. et al. Software-Defined Networking: A Comprehensive Survey. **Proceedings of the IEEE**, New York, v. 103, n. 1, p. 14–76, jan. 2015.

LEISERSON, C. E. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. **IEEE Transactions on Computer**, New York, v. 100, n. 10, p. 892–901, out. 1985.

LEVIN, D. et al. Toward Transitional SDN Deployment in Enterprise Networks. **OpenNetwork Summit**, 2013.

LVM Project. **Linux Kernel Logical Volume Manager**. 2015. Disponível em: <http://sourceware.org/lvm2/>. Acesso em: 16 fev. 2015.

MANN, V. et al. CrossRoads: Seamless VM mobility across Data Centers through Software Defined Networking. In: NETWORK OPERATIONS AND MANAGEMENT SYMPOSIUM, 2012, Maui, HI. **Anais...** New York, NY: IEEE, 2012. p. 88–96.

MCKEOWN, N. et al. Openflow: enabling innovation in campus networks. **SIGCOMM Computer Communication Review**, ACM, v. 38, n. 2, p. 69–74, 2008.

MUDIGONDA, J. et al. Spain: COTS Data-Center Ethernet for Multipathing over Arbitrary Topologies. In: NETWORKED SYSTEM DESIGN AND IMPLEMENTATION, 7., 2010, San Jose, CA. **Anais...** Berkeley, CA: Usenix, 2010. p. 265–280.

MUDIGONDA, J. et al. NetLord: A Scalable Multi-tenant Network Architecture for Virtualized Datacenters. **SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 41, n. 4, p. 62–73, ago. 2011. ISSN 0146-4833.

MYSORE, R. N. et al. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In: SIGCOMM, 2009, Barcelona. **Anais...** New York, NY: ACM, 2009. p. 39–50.

NAOUS, J. et al. NetFPGA: Reusable Router Architecture for Experimental Research. In: WORKSHOP ON PROGRAMMABLE ROUTERS FOR EXTENSIBLE SERVICES OF TOMORROW, 2008, Seattle, WA. **Anais...** New York, NY: ACM, 2008. p. 1–7.

OpenNetworking Foundation. **OpenFlow Switch Specification 1.3.4**. 2014. Disponível em: <https://goo.gl/RWyV1R>. Acesso em: 14 dez. 2014.

PFAFF, B.; DAVIE, B. **The Open vSwitch Database Management Protocol**. RFC Editor, 2013. Disponível em: <https://tools.ietf.org/html/rfc7047>. Acesso em: 15 mar. 2015.

PFAFF, B. et al. Extending Networking into the Virtualization Layer. In: WORKSHOP ON HOT TOPICS IN NETWORKS, 8., 2009, New York. **Anais...** New York, NY: ACM, 2009.

Project Floolight. **Indigo Virtual Switch**. 2015. Disponível em: <http://www.projectfloodlight.org/indigo-virtual-switch/>. Acesso em: 16 fev. 2015.

Quagga Project. **Quagga Routing Suite**. 2015. Disponível em: <http://www.quagga.net>. Acesso em: 16 fev. 2015.

REKHTER, Y. et al. **Address Allocation for Private Internets**. RFC Editor, 1996. Disponível em: <https://tools.ietf.org/html/rfc1918>. Acesso em: 15 mar. 2015.

ROSEN, E.; VISWANATHAN, A.; CALLON, R. **Multiprotocol Label Switching Architecture**. RFC Editor, 2001. Disponível em: <https://tools.ietf.org/html/rfc3031303>. Acesso em: 15 mar. 2015.

SAHOO, J.; MOHAPATRA, S.; LATH, R. Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues. In: INTERNATIONAL CONFERENCE ON COMPUTER AND NETWORK TECHNOLOGY, 2., 2010, Bangkok, Thailand. **Anais...** New York, NY: IEEE, 2010. p. 222–226.

SOUNDARARAJAN, V.; GOVIL, K. Challenges in building scalable virtualized datacenter management. **SIGOPS Operating System Review**, ACM, New York, NY, v. 44, n. 4, p. 95–102, dez. 2010.

TCPDump Project. **TCPDump a powerful command-line packet analyzer**. 2015. Disponível em: <http://www.tcpdump.org>.

VAHDAT, A. et al. Scale-out networking in the data center. **IEEE Micro**, IEEE Computer Society, Los Alamitos, CA, USA, v. 30, n. 4, p. 29–41, 2010.

VMWare Inc. **VMWare ESXi hypervisor**. 2015. Disponível em: <http://www.vmware.com/products/esxi-and-esx/>. Acesso em: 16 fev. 2015.

VMWare Inc. **VMWare Network Virtualization Platform**. 2015. Disponível em: <http://www.vmware.com/products/nsx/>. Acesso em: 16 fev. 2015.

VServer Project. **Linux VServer Project**. 2014. Disponível em: <http://www.linux-vserver.org/>. Acesso em: 15 dez. 2014.

Xen Project. **Xen Virtualization Project**. 2015. Disponível em: <http://www.xenproject.org/>. Acesso em: 16 fev. 2015.

ZHANG, Y.; ANSARI, N. Green Data Centers. **Handbook of Green Information and Communication Systems**, Academic Press, Cambridge, MA, p. 331, nov. 2012.