ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

THESIS PRESENTED TO
ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
A MASTER'S DEGREE WITH THESIS IN INFORMATION TECHNOLOGY
M.Sc.A.

BY
Mohamed FEKIH AHMED

TOWARDS FLEXIBLE, SCALABLE AND AUTONOMIC VIRTUAL TENANT SLICES

MONTREAL, JANUARY 28, 2014

This thesis is dedicated to my late father,

I hope he appreciates this humble gesture as evidence of recognition by a son who has always prayed for the salvation of his soul. May God, the Almighty, have mercy in his holy!

To my dear mother,

It is through this modest work the expression of my deep respect and all the love I have for you. "May God protect you."

To my wife,

Thank you for your love and support during this first chapter of our lives.

To my family . . .

ACKNOWLEDGEMENTS

I am deeply grateful to my advisors, Professors Mohamed Cheriet and Chamssedine Talhi, for their advice and feedback on conference papers, journals, patents, demonstrations, slides, practice talks, and countless other things. Their encouragement and guidance always push me to improve and steer my focus towards the right piece of the puzzle, even when I have not realized its importance yet. Thanks to them, I worked on an inspiring research projects, close to passionate researchers.

I would also like to express my gratitude to Doctor Makan Pourzandi for demonstrating excellence and passion for cloud security research. He has inspired me to pursue, and helped me to obtain, amazing results on two different projects: Multi-tenant Isolation and Cloud Defense Orchestration. His guidance was crucial in the realization of my master project.

Without the help of an innumerable number of people, this thesis would not have been possible. In this section, I will try to thank many of them. First, I would like to thank my committee. The unwavering support and direction provided by Ericsson Research Canada has been invaluable. Each of diverse perspectives that they provide have been necessary for the completion of this work in particular for Yves Lemieux, Ali Kanzo, Benoit Tremblay and Denis Monette. Second, I would like to thank Mathieu Lemay, as well as everybody else at the Inocybe Technologies, for his continuous help and support while setting up OpenStack and SDN testing environment on the Ericsson Blade Systems. Last, I want to thank my friends and fellow students from Synchromedia labotory for celebrating my successes, taking the sting out of paper rejections, and providing moral support along this journey. Its friendly and relaxed atmosphere as well as the kindness and helpfulness of its members has put me in the best conditions possible.

I would like to thank NSERC, Ericsson, Inocybe Technologies and MITACS for supporting the Green Sustainable Telco Cloud Project. This work is partly funded by Natural Sciences and Engineering Research Council of Canada Research Chair on Sustainable Smart Eco-Cloud, NSERC-950-229052 and by the NSER CCRDPJ 424371-11: ECOLOTIC Sustainable and Green Telco-Cloud.

Finally, I would like to thank thesis committee members: Professors Abdelouahed Gherbi, Mohamed Cheriet and Chamssedine Talhi, and Mr. Yves Lemieux.

TOWARDS FLEXIBLE, SCALABLE AND AUTONOMIC VIRTUAL TENANT SLICES

Mohamed FEKIH AHMED

ABSTRACT

Multi-tenant flexible, scalable and autonomic virtual networks isolation has long been a goal of the network research and industrial community. With Software-Defined Networking (SDN) and Overlay Virtualization technologies (OVT), multiple and independent virtual networks each with potentially different and heterogeneous addressing, forwarding and tunneling mechanisms can coexist above the same underlay infrastructure. For today's cloud platforms, providing tenants requirements for scalability, elasticity, and transparency is far from straightforward. SDN addresses isolation and manageability through network slices, but suffers from scalability limitations. SDN programmers typically enforce strict, inflexible, and complex traffic isolation resorting to low-level encapsulations mechanisms which help and facilitate network programmer reasoning about their complex slices behavior. Overlay protocols have successfully overcome scalability issues of isolation mechanisms, but remain limited to single slice. However, the opportunity cost of the successful implementation of transparent and flexible slices is to find an alternative isolation design to satisfy multiple and different tenant' requirements such as slice scalability and enabling the deployment of arbitrary virtual network services and boundaries.

In this thesis, we propose Open Network Management and Security (OpenNMS), a novel software-defined architecture overcoming SDN and OVT limitations. OpenNMS lifts several network virtualization roadblocks by combining these two separate approaches into an unified design. It enables to reap the benefits of network slice while preserving scalability. Our design leverages the benefits of SDN to provide Layer 2 isolation coupled with network overlay protocols. It offers multi-tenants isolation with simple and flexible Virtual Tenant Slices (VTSs) abstractions. This yields a network virtualization architecture that is both flexible, scalable and secure on one side, and self-manageable on the other. At the core of these challenges, we extend our research to outline the SDN control plane scalability bottleneck and demonstrate the benefits of OpenNMS to limit the load on the controller for supporting larger number of tenants. OpenNMS exploits the high flexibility of software-defined switches and controllers to break the scalability bottleneck and scale the network to several thousands of isolated tenants networks on top of shared network infrastructures. It requires only a small amount of line as extended application to OpenFlow controller without any modifications on SDN dataplane which makes it suitable for legacy systems. Furthermore, this work takes a step towards reducing the complex network management operations. We designed OpenNMS as an autonomic communication based architecture, to provide self-configured and self-awareness VTSs network for cloud tenants. The result is recursive, layered isolation architecture, with control and management planes both at tenant and overall network levels. Moreover, we describe novel capabilities added for the isolation model: Split, Merge and Migrate (SMM) that can be well suited for cloud requirements. We implemented our approach on a real cloud testbed, and demonstrated our isolation model's flexibility and scalability, while achieving order of magnitude improvements over previous isolation approaches investigated in this work. The experiment results showed that the proposed design offers negligible overhead and guarantees the network performance while achieving the desired isolation goals.

Keywords: Isolation, Flexibility, Scalability, Autonomic Management, Transparency, Software-Defined Networking

TOWARDS FLEXIBLE, SCALABLE AND AUTONOMIC VIRTUAL TENANT SLICES

Mohamed FEKIH AHMED

RÉSUMÉ

L'isolation flexible, évolutive et autonome des réseaux multi-locataire a été pour longtemps un objectif de la recherche et de la communauté industrielle. Avec Software-Defined Networking (SDN) et Overlay Virtualization Technologies (OVT), plusieurs réseaux virtuels indépendants, chacun avec mécanismes d'adressage, expédition, et tunnels potentiellement différents et hétérogènes, peuvent coexister au-dessus de la même infrastructure. Pour les plateformes de Cloud Computing d'aujourd'hui, les exigences des locataires en termes d'évolutivité, l'élasticité et la transparence sont loin d'être simple. SDN aborde l'isolation et la gestion des réseaux multi-locataires par les tranches virtuelles, mais souffre de limitations d'évolutivité. Les programmeurs de SDN appliquent généralement une isolation stricte, rigide et complexe du trafic et ont recourt à des mécanismes d'encapsulations de bas niveau afin d'aider et faciliter le raisonnement du locataire pour le comportement de leurs tranches complexe. Les protocoles d'OVT ont réussit à surmonter les problèmes d'évolutivité des mécanismes d'isolation, mais restent limités pour une seule tranche du réseau. Cependant, l'opportunité de réussir à mettre en œuvre une tranche transparente et flexible est de trouver un autre design de l'isolation afin de satisfaire les exigences de multiples et différents locataires telles que l'évolutivité du tranches et le déploiement arbitraire des services et les zones de limitations des réseaux virtuels.

Dans ce mémoire, nous proposons une nouvelle architecture intitulée, Open Network Management and Security (OpenNMS), qui nous permet de surmonter les limites de SDN et OVT. OpenNMS soulève plusieurs barrages de virtualisation de réseau. En combinant ces deux approches distinctes en une conception unifiée, notre design nous permet de profiter des avantages du tranchement du réseau tout en préservant l'évolutivité. Notre conception s'appuie sur SDN afin de fournir une isolation au niveau du couche 2 couplée avec les protocoles de OVT. Elle offre une isolation pour les multi-locataires avec des abstractions simples et flexibles des tranches virtuelles d'un locataire (Virtual Tenant Slices (VTSs)). On obtient ainsi une architecture de virtualisation des réseaux qui est à la fois souple, évolutive, et sécurisée d'un côté, et l'auto-gérable sur l'autre. Au cœur de ces défis, nous étendons nos recherches pour pointer sur le problème major du plan de contrôle du SDN plan. On démontre que le design d'OpenNMS nous a permis de limiter la charge sur le contrôleur afin de soutenir plus grand nombre de locataires. OpenNMS exploite la grande flexibilité des commutateurs et contrôleurs de SDN afin de briser le goulot d'étranglement de l'évolutivité et d'étendre le réseau à plusieurs milliers de locataires isolés sur la même infrastructure de réseau partagés. Notre design nécessite seulement une centaine de lignes comme application prolongée au contrôleur OpenFlow sans aucune modification sur le plan de données de SDN, qui le rend approprié pour les systèmes existants. En outre, ce travail fait un pas vers la réduction des opérations de gestion de réseau complexes. Nous avons conçu OpenNMS comme une architecture à base de communication autonome afin de fournir un réseau SDN auto-configuré et l'indépendance aux locataires de nuages. Le résultat est récursive, une architecture en couches d'isolation, avec des plans de contrôle et de gestion accessible au même temps par le fournisseur de l'infrastructure et les locataires. De plus, nous décrivons des fonctionnalités supplémentaires pour notre modèle d'isolation: Split, Merge et Migration (SMM) qui peut être bien adapté pour la nature du cloud computing. Nous avons implémenté notre approche sur un nuage réel. On a démontré la flexibilité et l'évolutivité de notre modèle d'isolation, tout en réalisant des améliorations de grandeur par rapport aux approches d'isolation précédentes étudiées dans le cadre de ce travail. Les résultats de l'expérience ont montré que le modèle proposé offre des coûts négligeables et garantit les performances du réseau tout en atteignant les objectifs d'isolation souhaités.

Mots-clés: Isolation, Flexibilité, Évolutivité, Gestion autonome, Transparence, Software Defined Networking

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABREVIATIONS

API	Application Program Interface
CE	Control Element
CIA	Confidentiality, Integrity and Availability
CLI	Command Line Interface
CRUD	Create, Read, Update and Delete
DCNs	Data Center Networks
DDoS	Distributed Denial-of-Service
Dp	DataPath
DPI	Deep Packet Inspection
FE	Forwarding Element
ForCES	Forwarding and Control Element Separation
FW	Firewall
GENI	Global Environment for Network Innovations
GRE	Generic Routing Encapsulation
IaaS	Infrastructure as a Service
IDS	Intrusion Detection System
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPS	Intrusion Prevention System

IT	Information Technology
KVM	Kernel-based Virtual Machine
L2	Layer 2
L3	Layer 3
LB	Load Balancer
LLDP	Link Layer Discovery Protocol
MAC	Media Access Control
MAPE	Monitoring, Analyzing, Plan, and Execute
NaaS	Network as a Service
NFV	Network Function Virtualization
NP-hard	Non-deterministic Polynomial-time hard
NVGRE	Network Virtualization using Generic Routing Encapsulation
NVP	Network Virtual Platform
OA	OpenNMS Agent
OAM	OpenNMS Autonomic Manager
OF	OpenFlow
OF-Config	OpenFlow Configuration protocol
ODL	OpenDayLight
OpenNMS	Open Network Management and Security
ONF	Open Networking Foundation

OVS	Open vSwitch
OVSDB	Open vSwitch DataBase
OVT	Overlay Virtualization Technologies
OVN	Overlay Virtualized Network
PaaS	Platform as a Service
PF	Permitted Flow
QoS	Quality of Service
SaaS	Software as a Service
SDN	Software Defined Networking
SLA	Service Level Agreement
SMM	Split, Merge and Migrate
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
TOR	Top Of Rack
TNI	Tenant Network Identifier
TVD	Trusted Virtual Domain
UCLP	User Controlled LightPaths
UDP	User Datagram Protocol
vCDNI	vCloud Director Network Isolation
VLAN	Virtual Local Area Network

VNF	Virtuak Network Function
VNI	VXLAN Network Identifier.
VNI	Virtual Network Interface.
VM	Virtual Machine
VTN	Virtual Tenant Network
VTS	Virtual Tenant Slice
VXLAN	Virtual Extensible Local Area Network

CHAPTER 1

INTRODUCTION

1.1 Cloud Computing and Network Virtualization

Cloud computing has emerged as a new paradigm for sharing computing and networking resources among multi-tenant. By moving the applications and services to the cloud, the organisations no longer have to build and maintain expensive Data Center. Cloud Computing has become very popular because it reduces the cost of IT capital. Additionally, cloud providers offer a "pay-as-you-go" service, where tenants can scale their applications and services on demand. It allows them to keep pace with demand spikes, and save money when the demand is low.

The cloud providers can be classified in four main categories, depending on the model of service they are offering. The first category is the Infrastructure as a service (IaaS), which is the most basic categorie. IaaS consists in providing computer resources (can be physical or virtual machines), servers, storage, and network devices (switches, routers, ...). IaaS clients have to install and configure their operating systems on allocated computing resources as well as networking topology. For example, Amazon EC2, Rackspace and OpSource are cloud IaaS providers. These providers allow clients to dynamically scale up and down their applications, and provision resources by adjusting them to the current demand. Secondly, there is the Platform as a service (PaaS) model where providers deliver only a computing platform. Tenant can develop and run their private cloud platform without the cost and complexity of building a Data Center. Examples of PaaS include Amazon AWS and Microsoft Azure. The third category is the Software as a service (SaaS), which is the most widely known cloud service. SaaS basically refers to the use of the software through internet, which could be an online service (e.g., Google Apps). In this model, service provider handle all the management and configuration operations of the infrastructure and platform where the application is installed. This means that tenant are no longer need to go on the infrastructure details. Finally, the last category is the

Network as a service (NaaS), which refers to the use of network and transport services (e.g., VPN, Bandwidth on demand, and Virtual Nerworks). It is a business model for delivering online connectivity services on a "pay-per-use" manner. Moreover, with the introduction of new paradigms in server and network virtualization, other new categories are proposed for the Cloud Computing.

Cloud Computing has completely changed the IT scenery. Along with the spread of Cloud Computing, network virtualization is highly used in Data Center Networks (DCNs). The popularity of network virtualization results from flexible and efficient management. It makes the infrastructure of network providers more profitable on one side, and helps tenants to decrease their IT capital expense on the other. It allows also new features and dynamic usage models while eliminating the costs and overhead of underlay networks management. In DCNs, network resources such as switches and routers are shared among multiple tenants to reduce physical resources and power costs.

However, despite all the advances in server virtualization, traditional network security practices are still common. Network virtualization benefits have created new opportunities (e.g., scaling application) as well as new challenges (e.g., node migration) for the network security tasks such as security services and applications scalability, migration and provisioning. Thus, the opportunity cost of the virtualization spread and Cloud Computing success is to rethink about security enforcement in the cloud applications. Two major challenges for network security are relevant to be discussed in next sections: Data Center fast and unplanned growth, and the cloud tenant requirements for flexible, scalable and elastic isolation.

1.2 Paradigm Shift for Network Security in Cloud Computing

With rate and scale unforeseen, large companies are building progressively more enormous data centers. For example, Ericsson as the world's largest maker of wireless networks, is building three data centers to be up to 120,000 square meters, approximately the size of 14 football fields. It will open the third center with massive 40,000 square meters in Montreal Vaudreuil-

Dorion. Imagine Ericsson's data centers area filled with nothing but servers and networking equipment, and supporting tens and hundreds of thousands of tenants. Several tenants require substantial number of virtual networks for separating their different types of applications. As tenant numbers and applications continue to scale, scaling the capacity of the network fabric for isolated virtual networks becomes a major challenge. Such cloud-computing platform has grown so large that it has become not possible to separate such number of tenants and applications with the traditional isolation techniques (e.g., Overlay protocols, Local Hypervisor, Middleboxes). In addition, only the infrastructure provider has all the privileges to configure and isolate each virtual network using the same old tools. These traditional isolation mechanisms are no more suitable for the cloud environment. It is very difficult to apply a consistent access control and implement a network-wide isolation.

Furthermore, this exciting environment presents also powerful management challenges not seen before on the "open" cloud. A data center exists always under a single administrative domain. The overly complicated control of traditional routers makes network management more complex. This is principally resulted by the tight coupling of control plane and data plane in the same box. Due to this high complexity, telecommunication vendors and network operators are seeking for new protocol and technology to facilitate the management operations. Providing transparent control according to high-level security policies specifications for cloud tenants is the most challenging objective.

From multi-tenant perspective, the single and privileged control is no more adequate for the current cloud applications requirements. Security concerns are raised and it is becoming increasingly difficult to follow the elasticity of cloud computing as soon as one begins to run applications beyond the designated security defense line (e.g., Firewall, Load-Balancer, Virtual Private Network, and Intrusion Detection System) and move them to unprotected perimeter. In addition to adopting network virtualization, the dynamic and elastic nature of cloud computing is challenging many aspects of network virtualization including isolation tools and technologies which are no longer practical, efficient, or flexible enough for today's tenant requirements. Network architectures limitations are attributed to big-data workload setting, traffic changing and resources sharing among between virtual tenant networks. They suffer of the following drawbacks: (i) expensive scaling of the network to the sizes needed by service providers, (ii) limited support for multi-tenancy as they do not give the possibility to the tenant for designing his virtual networks and defining its own Layer 2 and 3 spaces, and (iii) complex and manual management operations of large set of network nodes including switches, routers and firewalls.

With all these challenging objectives, it is a very exciting time for networking research. We are on the cusp of a major shift in network virtualization, and the related technologies of network security and isolation. It is evolving toward open standards of control. It is argued that the clear separation and open interfaces between the control and data planes is a major factor for the rapid innovation and growth in network virtualization. Thereby, network research and industrial community are leading of innovative and complementary networking technologies and protocols designed for the internet at large. One interesting evolution to the growth story of data centers is the design of new overlay protocols to solve the scalability bottleneck of existing low-level encapsulation protocols (e.g., VLAN, GRE). An industrial competition was started to replace these traditional isolation mechanisms. New encapsulations approaches are proposed as successor (e.g., Cisco's VXLAN (Mahalingam *et al.* (2012)), HP's NVGRE (Sridharan *et al.* (2013))). At the core of this challenge is how to provide a large number of segmentation that can solve the VLAN limits.

The major recent development in network virtualization is the notion of Software-Defined Networking (SDN) (McKeown (2009)). It was the results of the research community frustration by the difficulty faced with large-scale networks experimentation. Networking researchers were calling for the same thing to solve the previous challenges: the decoupling of network control from the underlying physical devices. These calls successfully reached their goals by the design and implementation of new protocols like OpenFlow (McKeown *et al.* (2008) and ForCES (Doria *et al.* (2007)).

Finally, another approach related to the introduction of SDN is the virtualization of network security middleboxes. Network-based security boxes were traditionally implemented and placed at network choke points as one classes of functions in physical middleware boxes across data centers with significant configuration overhead and assuming the underlay network would remain static with occasionally update. Therefore, there has been efforts in research community aiming to integrate these security middleboxes into SDN for exploiting the benefit of programmability to redirect selected network traffic through small, chained and distributed middleboxes.

In section 2.2, we give more details about the previous technologies and protocols which are important to our work.

1.3 Multi-Tenant Network Isolation: What is missing ?

While the emerging areas of network virtualization and SDN are teeming with exciting unsolved problems and research opportunities, this work attempts to tackle three specific challenges that arise from the cloud elastic nature and today's tenant urgent needs; namely, i) Virtual Networks Isolation Scalability and Flexibility, ii) Multi-Tenant Network Autonomic Management, and iii) Both SDN Controller and Network Scalability. We discuss each of these challenges in turn.

1.3.1 Multi-Tenant Flexible and Scalable Isolation

The multi-tenancy nature of the cloud is challenging many aspects of traditional and current virtual network isolation tools and technologies. The research and industrial network communities have tackled the basic isolation functionality that guarantees the separation between tenants' traffic as one block of security application and assuming the tenant security defense would remain static with occasionally update. They have proposed three different approaches for the isolation presented above: 1) Overlay Encapsulation Protocols, 2) Middleboxes and, 3) SDN Slicing Technique (See section 2.2). They are relying on low-level encapsulation mechanisms (e.g., VLANs, GRE, VXLAN, NVGRE), burden special-purpose security middleboxes (e.g., L3, L4-L7 Firewalls, Intrusion Detection and Prevention Systems, Deep Packet Inspection) or complicated SDN hypervisor installed on data plane which are no longer practical, efficient, or flexible enough for today tenants' requirements.

With the OVT, they provide traffic isolation by compiling and installing packet-processing rules with diverse sets of tenant-specific requirements into a single configuration space to avoid multi-tenant traffic interference and information leaks. Furthermore, They make blurred boundaries between multi-tenants' virtual networks and add extra complexity to the already difficult task of writing a single security network configurations sharing the underlay infrastructure. Additional time for the consistency of the enforced, bugs and conflicts are often faced.

For both overlay and underlay levels, the origin of security consciousness in cloud computing comes from the complexity of management operations of security middleboxes. For example, firewalls boxes or data plane processing rules require complex management operations and they are difficult to manipulate, modify and update. Middleboxes approaches limitations are attributed also to big-data workload setting, traffic changing, resources sharing between virtual tenant networks, and tenant VMs migration. With one complex block of security configuration including a mix of tenants' policy rules, they do not address how to create and build a single virtual tenant network out of multiple, independent, reusable network policies that can be suitable for the nature of the cloud. Tenant shields must escort the protected cloud applications, so they can be merged, separated and moved depending on applications status, requirements and locations. We must release the network programmers from having to reason, analyze and verify the buggy complex block of security configuration and labored architecture issues such as tenant's VMs and related security boxes placements (Qazi *et al.* (2013)). In addition to unsatisfactory and complex middleboxes, they offer little ability for cloud's users to leverage their benefits. Two complex and static manners cannot be avoided: first, the overlay level where tenants can allocate security VMs dedicated for necessary traffic redirection and protection (e.g., allocating NAT VM in Amazon Virtual Private Cloud (VPC)), and second the underlay level where cloud provider implements specific hypervisor (e.g., Amazon VPC Route Tables) and lets tenants actively create and manage their networks.

The common drawback of research approaches is focusing on exploiting SDN capabilities to provide only strict and inflexible isolation without considering both cloud providers and tenants requirements. They suffer from the following drawbacks that will be addressed in this work:

- Scaling the network to the size needed of isolated virtual tenant networks by service providers is very expensive and requires huge number of nodes (e.g., hosts, switches, routers) which demand complex management steps to enforce one of the traditional isolation mechanisms listed above.
- Limited support for multi-tenancy with different security requirements as they do not give the possibility to the tenant for designing his virtual networks and defining his own layer L2 and L3 spaces.
- No support for tenant scalability and flexibility needs such as adding more computing and networking resources, splitting tenant virtual network into two or many or merging it back into one.
- Cannot act in case of tenant VMs migration resulting security domain changing. Tenant security defense must be migrated with the target VM. The defense on the source location must be no longer active and enforced on the destination host. It requires also doing some configuration adjustments in the new host. IaaS providers are seeking for an alternative design for the virtual tenant networks boundaries that support larger number of tenants with arbitrary topologies, scalability support, ease of management operation at low cost, and satisfy the tenant's needs by enabling to take full control of his virtual programmable networks.

1.3.2 Autonomic Management

Network virtualization approaches are facing increasing difficulty to meet the requirements of current cloud applications, network services, multi-tenants and infrastructure providers. Network researchers were led to think more about new network architecture as well as management technologies. Network management poses challenges that need to be addressed in order to fully achieve an effective and reliable networking environment.

Early network management systems were designed with limited access and provided only to network administrators who were responsible for responding, monitoring, querying, and manually configuring every virtual tenant network. Involving tenant in management plane is a new goal for network research and industrial community. It can reduce the management complexity for the fast growing network and automate the control. Providing transparency and self-management for multi-tenants represent today an urgent need which must be built on top of efficient isolation. It is widely accepted that next-generation networks will require a greater degree of service awareness and optimal use of network resources. It is becoming primordial that future networks should be self-controlled and self-manageable. Furthermore, the new emerging SDN paradigm seems ignoring or not addressing properly network management problem. The first time that the management plane has been introduced with SDN and OpenFlow, was with OpenFlow-Config protocol (OF-Config; Pfaff and al.). However, small network management functions are proposed with OF-Config. Few network management approaches for Software-Defined Networks (e.g., Devlic *et al.* (2012a), Devlic *et al.* (2012b)) were proposed using a combination between OpenFlow protocol, OF-config protocol, and management protocols like SNMP (Feit (1993)) or its successor NETCONF (Enns *et al.* (2011)).

In particular, some researchers have taken a different and interesting track with the Autonomic Communication (Sestinim (2006)) or IBM's Autonomic Computing (Murch (2004)):

- Autonomic communication is one of the emerging research fields in future network control and management. Self-Awareness, Self-Configuration, Self-Protection, Self-Healing, Self-Optimization and Self-Organization represent the autonomic communication attributes. The objective of introducing these autonomic self-* attributes into network management plane is to decrease the complexity of network control and management operations, and to automate the network configuration.

- Autonomic computing is an initiative started by IBM in 2001. It aims to create self-manageable networks and enable their further growth. It frequently refers to the same self-* attributes. IBM's vision defines a general management framework in which the autonomic manager is responsible for four main tasks: Monitoring, Analyzing, Plan and Execute, termed MAPE loop. This loop is a knowledge-base that maintains the necessary information for the managed systems.

We believe that by introducing the Autonomic self-* attributes into SDN-based architecture and involving the tenant in management plane, the programmable capability of the SDN could be enhanced to an environment aware programmable capability. Autonomic SDN-based architecture could build a specified network application to provide self-configuration in tenant level and self-organization in network provider level. It can also improve its performance by supporting self-optimization through the self-awareness attribute. SDN, which decouples control and forwarding capabilities, has improved its intelligence in network control so that the operation and management functions of SDN could be more efficient.

1.3.3 SDN Controller Scalability

Despite all the discussed SDN advantages, there have always concerns about the OpenFlow controller scalability in DCNs which must be considered to avoid its negative effects on SDN based solutions. SDN-based architectures rely completely on centralized or distributed controllers to manage all network switches and neglect the management plane, which has major drawbacks including lack of either network or controller scalability or both.

Unfortunately, as the network scales up, both the number of physical and virtual switches increases to guarantee the minimal QoS to the end host. Regardless of the controller capacity, a controller likes NOX does not scale as the network grows. The SDN controller becomes a key bottleneck for the network scalability. Specially, SDN community (Tavakoli *et al.* (2009)) estimates that large DCNs consisting of 2 million VMs may generate 20 million flow per second (fps) and current OpenFlow controllers can handle 10^3 fps. Another study (Tootoonchian *et al.* (2012b)) demonstrates that NOX-MT (Tootoonchian *et al.* (2012a)), Beacon (Erickson *et al.* (2013)), and Maestro (Ng (2012) process 1.6 million fps with an average time of 2 milliseconds when controlling 64 emulated switches using up to 8 core machine with 2 GHZ CPUs. Later measurements (Voellmy and Wang (2012)) demonstrated that NOX-MT can scale up to 5 million fps using 10 CPU cores machine and Beacon scales to 13 million fps with 10 more CPU cores.

The relying on a centralized network control plane introduced scaling bottleneck with the fast and unplanned growth of the network. Some approaches have attempted to reduce the burden on the centralized controller by using distributed controllers or by including some changes on SDN paradigm and delegating a part of control functionalities to network switches. For example, DevoFlow (Curtis *et al.* (2011)) focuses on improving the performance of OpenFlow controller tasks such as installing flow entries and network monitoring. It reduces the load on OpenFlow controller by refactoring the OpenFlow API and extending the network switch with control engine. However, DevoFlow reduces the decoupling between control and data planes and loses the centralized visibility. With a similar approach, DIFANE (Yu *et al.* (2010)) treats the controller scalability by keeping all traffic in data plane. It considers the switch rule memory like a large distributed cache and releases the controller from handling the traffic processing and checking.

Towards a similar goal but with different method, NOX-MT, Beacon, and Maestro scale network controllers using multi-threaded controller. In particular, Maestro is based on a 8 cores server machine. It distributes the controller workload among available cores, so that will balance the load between the 8 cores. Actually 7 cores are used for worker threads and one core is used for management functionalities.

Other systems Onix (Koponen *et al.* (2010)) and HyperFlow (Tootoonchian and Ganjali (2010)) provide multiple distributed controllers in order to load balance the incoming events load across between them. They partition network state across these multiple network control systems and alleviate scalability and fault-tolerance concerns. They take advantages from the OpenFlow 1.3

version that allows multiple controllers connection with OpenFlow enabled switches in aim to resolve the controller issue. Thus allowing distributed multiple controllers in managing DCNs is an appropriate solution, however there is no mechanism for cross virtual tenant networks packets processing and no approaches for multi-tenant scalable isolation. Using cluster of synchronized controllers (Master/Slaves) to offload the network charges would be sufficient to resist and recover from control failures but would leave parts of the network brainless and loose the key advantage of SDN: centralized control and global network view.

1.4 Purpose of the Research

We have seen that multi-tenant network isolation is a major hurdle to the cloud adoption, and that the complex and manual network configuration makes it hard to manage and enable its further growth. Due to the elastic nature of multi-tenant network, it is not possible to keep the old and manual management approaches in a similar way to what is done in the private networks. It would not be flexible and reactive enough to meet with the cloud applications requirements for dynamic network boundaries changes, policy rules adaptation and transformation on-demand, and allocated resources migration between isolated networks.

As results, we have identified the multi-tenant isolation as the first requirement of our model. First of all, we have seen the Overlay Virtualization Technologies (OVT), where low encapsulation protocols (e.g., VLAN, GRE, VXLAN, NVGRE) are used to separate tenant virtually from one another. These techniques provide a guaranteed scalability but remain limited to a single virtual tenant network or slice. The last isolation technology evoked was the SDN slicing approach. It allows multiple researchers to share the same underlay network by creating separate and independent virtual slices. However, this approach faces a critical scalability bottleneck.

In order to build our solution that considers all the discussed multi-tenant network requirements, we believe the more suitable technology would be the last one exposed (See section 2.2.2.4 for more details). Indeed, the decoupling between SDN planes leads to interesting properties and brings numerous advantages such as flexibility. Furthermore, OpenFlow protocol is becoming essential for network control. It can manipulate each incoming packet finely such that deciding the forwarding port on-the-fly based on dynamically set flow rules and altering destination address or path. These properties give us a great opportunity to define elastic and flexible virtual tenant networks. Therefore, our solution is built as a SDN-based architecture. It will allow us to provide multi-tenant isolation by defining a high level abstraction at the application layer before being mapped on the underlay layer. This abstraction can summarize all the tedious details related to the security configuration implementing the desired tenant isolation. We believe that our abstraction should be based on modularity which is the key of managing complexity in any software system, and SDN is no exception.

In this work, we present Open Network Management & Security (OpenNMS), a novel modular software-defined architecture enabling multi-tenant scalable, flexible and autonomic isolation for virtual networks. OpenNMS takes a step towards providing L2 isolation with simple and flexible Virtual Tenants Slices (VTSs) abstractions in an automatic and dynamic way. To overcome SDN scalability bottleneck and overlays protocols limitation to single slice, OpenNMS lifts several roadblocks by combining these two separate technical approaches into an unified design. This yields a network virtualization architecture that is both flexible, and secure on one side, and scalable on the other. OpenNMS exploits the high flexibility of software-defined components and the scalability of overlay protocols to create several thousands of VTSs on top of shared network infrastructures. It gathers tenant's distributed VMs across physical networks into distributed VTSs which can be split, merged or migrated. It requires only a small amount of line as extended application to OpenFlow controller.

Using hierarchical OpenFlow controllers, we succeed to enforce our flexible isolation model and provide in the same manner an efficient and scalable offloading of control functions without losing the SDN centralized advantage. By delegating VTSs frequent and local packets to tenant controller, we limit the overhead on centralized controller that will process only global and rare events to maintain network-wide view. One master application on the control plane will orchestrate, build and map small tenant distinct portions or VTSs into the virtual and underlay networks. VTS space on data plane will be allocated for each tenant for building his own small security boxes out of multiple, independent, reusable network policies included in the VTS abstraction. Thus, as results of the simple and flexible VTS definitions model, we describe novel features and capabilities added for the isolation: Split, Merge and Migrate that can be well suited for the tenants' requirements in such dynamic nature of cloud computing.

In compliance with the latest trends in network management, we adopt the autonomic computing approach to add the autonomic aspect for our isolation model. Autonomic computing can be a complementary approach to SDN for evolving the neglected management plane and self-aware network configuration. It can allow an embedded management of all VTSs and gradual implementation of management functions providing code life cycle management for multi-tenant applications as well as the ability to on-the-fly configuration update. The Self-* capabilities in a SDN network can accomplish the centralized controller functions by recommending an appropriate action based on the overall network policies and tenant requirements. These capabilities are included in OpenNMS through control loop in the cloud provider's management plane.

1.5 Organisation

The remainder of this thesis presents a detailed description of our L2 multi-tenancy isolation approach, Virtual Tenant Slices abstraction model, OpenNMS framework design and evaluations.

The rest of this thesis is organized as follows:

Chapter 2 begins with all definitions and concepts required for this work, introducing the current multi-tenant data center practices and presenting the existing OpenFlow enabled components implementations. The rest of Chapter 2 lays out the required background for the three aforementioned network isolation techniques of this thesis and investigates their related issues. We follow with an overview of the currently available autonomic computing architectures.

In Chapter 3, we present the flexible L2 isolation model proposed for OpenNMS and detail how it can be exploited to provide VTSs scalability. We describe the concept, motivation behind it, and detail the used syntax. Finally, we show how OpenNMS reaches the isolation goals as well as the scalability of SDN controller.

Next, in Chapter 4, we describe the OpenNMS design and detailing the composition of OpenNMS planes including the autonomic management plane. We present OpenNMS evaluation along with experiments and results in chapter 5.

Finally, we draw conclusions, discuss OpenNMS limitations and present avenues of future research.

CHAPTER 2

BACKGROUND AND RELATED WORK

The current chapter provides a literature review of the current data center practices for the cloud's tenants. We introduce also all definitions and concepts about the previous technologies and protocols which are important to our work. As evoked in the previous chapter, the actual isolation approaches are facing scalability roadblocks and raises many challenges regarding to the cloud applications elasticity.

Next, we will present the existing Software-Defined Networking (SDN) implementations. Across our analysis, we will provide a detailed description of the industrial and research isolation using the SDN slicing or the Overlay Virtualization Technologies. We will analyse the different ways existing in order to separate the traffics in a multi-tenant infrastructure. We will expose their issues and advantages, before giving our conclusions on their utility for the current multi-tenant network requirements.

In a multi-tenant data center, any network isolation approach must provide enough scalability and flexibility in order to cope with the tenant requirements for network scalability and must tolerate the cloud applications live migration as a particular characteristic of the cloud elastic environment.

Our second goal is to provide an autonomic isolation solution. The isolation process has to be automatic in order to provide self-manageable virtual networks. In order to do so, we want establish a combination between SDN-based architecture and Autonomic Computing; this is why the literature review of the available Autonomic Computing is articulated as follow.

Finally, the conclusion will summarize and close the chapter, recapitulating the main drawbacks in actual isolation solutions while enlightening the key improvement that need to be achieved in order to provide flexible, scalable and autonomic virtual tenant networks using a high-level abstraction while providing secure and independent slices to each tenant.

2.1 Current Multi-tenant Data Center Practices

In cloud architectures, the provider's resources are pooled to serve multi-tenants, with different physical and virtual resources (e.g., compute, network, and storage) dynamically assigned and reassigned on demand. Any individual or business can get a portion or slice of the available resources for hosting their applications and deploying the desired topology in order to gain price and performance advantages compared to the cost of private data center. The cloud service providers are offering "a pay-as-you-go" model with little costs versus buying all of the hardware and software for building an individual data center. Each tenant subscribed to compute, network, and storage resources in a cloud is entitled to a given Service Level Agreement (SLA).

The notion of a tenant in the context of cloud computing is not as simple as it might first appear. The first security concerns in server virtualization relate to the co-existence of virtual machines (VMs) owned by different tenants. For this purpose, a very specialized and optimized software system, termed Hypervisor, was included to separate tenants' VMs. It is a virtualized layer located between VMs and physical hosts resources providing resources isolation allocated for different tenants. Operating systems, such as KVM (Habib (2008)), and Xen (Menon *et al.* (2006)), are developed for providing the means of turning a single piece of hardware into many separated VMs. These technologies can provide performance isolation between collocated VMs and prevent information leaks.

The missing piece for shared platform was the network isolation. A next generation for modular and virtualized platform for multi-tenant data center was proposed with basic concepts for network virtualization. It involves new low-level encapsulations protocols (e.g., VLAN, GRE, ...) for creating network tunnels between hypervisors. These tunnels allow the infrastructure to create virtual networks between different groups of VMs that are allocated for multi-tenants.

With the rapid development of management platforms and the emergence of different open-source laaS frameworks (e.g., OpenStack and OpenNebula), they have been developed in order to facilitate the deployment of cloud environments. They have been widely used due to their scalability with increasing number of resources, some of them can substitute commercial cloud management platforms. In particular, OpenStack is a collection of open source software projects which delivers a massively scalable cloud operating system and controls large pools of compute (Nova), storage (Swift), networking (Neutron) and Orchestration (Heat). However, these exciting environment present powerful management challenges not seen before on the open cloud. A data center exists under a single administrative domain. The overly complicated control of traditional routers makes network management more complex.

Despite all progress in virtualization and cloud management platforms (See Figure 2.1), the decision to choose the most suitable framework that meets the customers' needs becomes a difficult task, because every platform has its specific characteristics. The tenant generally has no control or knowledge over the virtual and physical configurations and even the deployed network topology. He is not able to pull any change on the allocated resources or specify the modification needed at a higher level of abstraction.

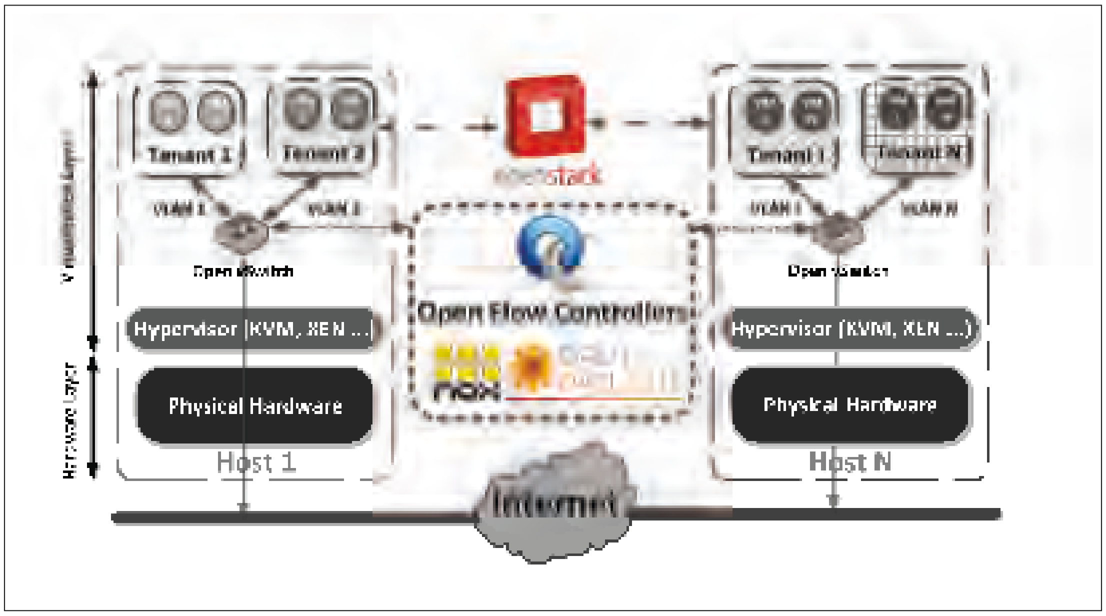

Figure 2.1 Current Multi-tenant Data Center Architecture

The situation has improved with the emergence of new network virtualization paradigm: SDN and related technologies like OpenFlow switches (e.g., Nicira's Open vSwitch) and Controllers (e.g., NOX, OpenDayLight), which create an open and transparent approach for multi-tenant network and automate the network control and management operations. SDN has great potential and flexibility to create a new generation for multi-tenant data center. SDN is still evolving but it is not ready for the data center business plan.

2.2 Definitions and Concepts

In this section, we will discuss the use of previous and novels technologies and protocols in order to provide secure virtual networks isolation. First, we review the use of classic encapsulation protocols, where Virtual Local Area Network (VLAN) and Generic Routing Encapsulation (GRE) are the most popular. Second, we focus on Virtual eXtensible LAN (VXLAN) and Network Virtualisation using GRE (NVGRE), which extend the VLAN and GRE protocols. Then, we move our focus on SDN paradigm and present the critical role of the OpenFlow protocol in SDN. In addition, we present and discuss the SDN slicing technique as emergent network isolation approach. Finally, we introduce the evolution of middleboxes approaches with SDN.

2.2.1 Virtual Network Overlay Protocols: Virtual Isolation

The concept of multiple and isolated virtual networks that can coexist in the same underlay infrastructure appeared in the networking literature with different segmentation capacities. In this subsection, we discuss four protocols: VLAN, GRE, VXLAN and NVGRE.

2.2.1.1 Virtual Local Area Network

Initially, traditional multi-tenant data centers have employed VLANs (Clear *et al.* (2002)) to isolate the machines of different tenants on a single Layer 2 (L2) network. They have deployed Ethernet networks based on IEEE 802.1Q to provide for VLAN bridging and isolation among tenants. The VLAN tag is a 12-bit field in the VLAN header, limiting this to at most 4K tenants. This imposes a growth limitation on the number of tenants that can be supported, depending on services and topology. Additionally, the VLAN widespread utilization makes an inescapable complexity in network management nowadays. It cannot be used as it represents a limitation to the scalability and management of the cloud model. It is rare to find VLANs in recently proposed cloud architecture.

2.2.1.2 Generic Routing Encapsulation

Similar to VLAN, Generic Routing Encapsulation (GRE) (Farinacci *et al.* (1994)) has been developed by Cisco Systems and allows tunneling to encapsulate a wide range of types of packets from the Internet Protocol (IP) layer. GRE tunnels encapsulate isolated L2 traffic in IP packets. The originality of GRE tunnels is that they are designed to eliminate the need of maintaining the connection state, which means that each tunnel does not keep any state information of the remote connection. Using GRE tunnels as tenant networks in Cloud solution avoids the need for a network interface connected to a switch configured to trunk a range of VLANs. For example, connection with servers that need to encrypt the data, can use GRE tunnel inside virtual point-to-point links over the internet for secure Virtual Private Networks. Another key benefits of GRE is the IP address and MAC address table scalability.

2.2.1.3 Virtual eXtensible Local Area Network

An industrial competition was started to replace the previous traditional isolation mechanisms. New encapsulations approaches were proposed as successor. At the core of this challenge, Virtual eXtensible Local Area Network (VXLAN by Mahalingam *et al.* (2012)) was proposed and implemented by Cisco to provide a large number of segmentation that can solve the VLAN limits. VXLAN addresses the scalability requirements of the L2 and L3 data center network in a multi-tenant environment. It extends the VLAN segmentation capacity and allows up to 16 million VXLAN segments to coexist within the multi-tenant network. The VXLAN segment is a 24-bit field, hereafter termed the VXLAN Network Identifier (VNI). The VNI scopes the inner MAC frame originated by the Virtual Machine (VM). Thus, it allows to overlap MAC addresses across VXLAN segments but never have cross over traffic since the traffic is isolated using the VNI qualifier which is an outer header envelope over the inner MAC frame originated by the VM. We can define shortly VXLAN with a L2 overlay scheme over a L3 network. Each overlay network is termed a VXLAN tag. Only VMs within the same overlay network can exchange packets. VXLAN was the first MAC-over-IP overlay virtual networking technology that could be implemented for large-scale L2 multi-tenant DCNs (e.g., VMware's vSphere).

2.2.1.4 Network virtualization using GRE

Another concurrent network virtualization technology to VXLAN with similar objectives, Network Virtualization using Generic Routing Encapsulation (NVGRE), was also developed by Hewlett-Packard (Sridharan *et al.* (2013)). However, it is not widely adopted as VXLAN. The companies supporting the development of NVGRE are Microsoft, F5 Networks, Arista Networks, Mellanox, Broadcom, Dell, Emulex, and Intel. NVGRE attempts to alleviate the scalability problems associated with large multi-tenant DCNs. It extends the GRE encapsulation to tunnel L2 packets over L3 networks and create large numbers of VLANs. Every virtual L2 network is associated with a 24 bit Tenant Network Identifier (TNI). This TNI represents a virtual L2 broadcast domain and allows up to 16 million virtual networks in the same multi-tenant DCNs in contrast to only 4K achievable with VLANs.

2.2.2 Software-Defined Networking, OpenFlow and Virtual Isolation

This thesis certainly would not have been possible without the emerging Software-Defined Networking (SDN, McKeown (2009)) paradigm. Next, we present SDN architecture and key aspects of the OpenFlow (McKeown *et al.* (2008)) protocol, which enables controller to switch interaction through a standardized protocol. We also describe SDN alternative standards such as ForCES (Doria *et al.* (2007)). Finally, we introduce the slicing technique proposed and deployed by SDN researchers.

2.2.2.1 Software-Defined Networking

SDN has gained significant traction and momentum in the network research and industrial communities in recent year. It is one promising technology and suitable to substitute traditional network architectures. Traditional architectures follows an unified design where the control and data planes are tightly coupled in the same network box. It usually results in overly complicated control plane and complex network management. Due to this high complexity, research community is convinced that networking must also start to shift and argue that the clear separation and open interfaces between control and forwarding planes is the key major factor for the rapid innovation and growth in multi-tenant networks. This was the heart of what they term SDN. Some researchers consider that Ethane project (Casado *et al.* (2007)) was the key turning point of the SDN paradigm. Ethane architecture allows fine-grained admission and routing policies for tenant applications in enterprise networks using a centralized networking control. Similar to Ethane, SDN uses an external centralized controller to manage the behavior of all network switches. Switches are abstracted as simple forwarding engines, and they export incoming packets to SDN controller which defines a set of actions (e.g., forward, drop) for each packet. This global network view and control allows fine-grained security and high performance required from a datacenter network. SDN allows also a global management abstractions. A network manager would collect the network view and topology graph discovered by the centralized controller. It would orchestrate and present possible management actions and event triggers (e.g. node failures). The SDN architecture has drawn the attention of the whole network community. It has been used and implemented with a variety of researchers and industrials, including management and control solutions for the SDN switch implementations (Naous *et al.* (2008), Koponen *et al.* (2010), Sherwood *et al.* (2009), Mattos *et al.* (2011)), network virtualization (Drutskoy *et al.* (2013), Rotsos *et al.* (2012), Monsanto *et al.* (2013)), and network security enforcement (Schlesinger *et al.* (2012), Porras *et al.* (2012), Qazi *et al.* (2013), Shin *et al.* (2013a)).

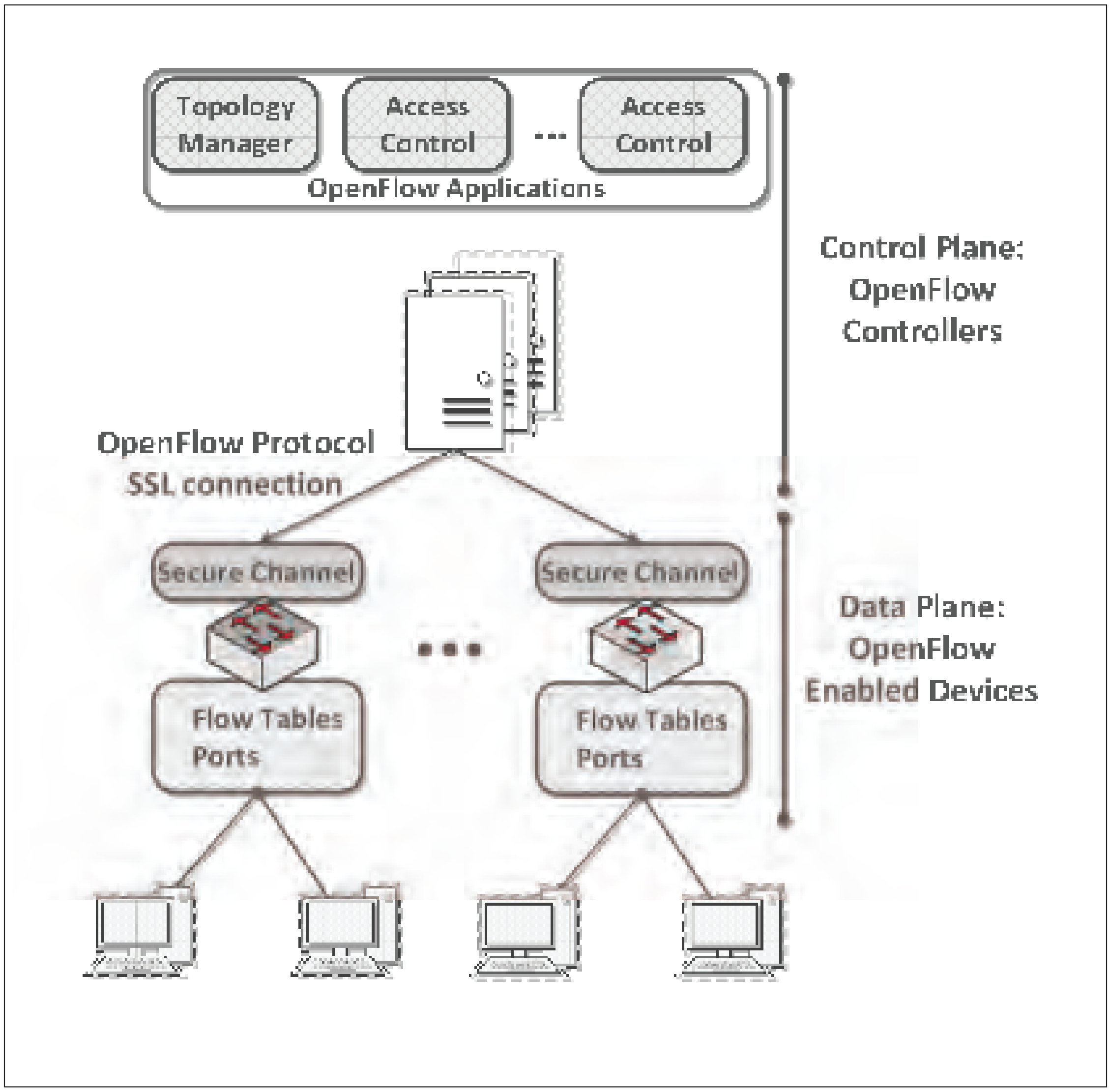

Figure 2.2 SDN/OpenFlow Network Architecture

2.2.2.2 OpenFlow

The missing puzzle to reach the desired decoupling between control and data planes was the SDN OpenFlow standard (McKeown *et al.* (2008)). OpenFlow has emerged as powerful, programming API for managing and controlling large-scale distributed network systems. It allows the network programmer to customize network switches behaviours through OpenFlow applications extending the centralized SDN controller (See Figure 2.2). This open design exposes the capabilities of network nodes and provides for both infrastructure providers and tenants an increased flexibility in managing and deploying network services. This uniform forwarding abstraction allows to support various and multiple network processing functions, such as L2 isolation, IP multicast forwarding, traffic policing, monitoring, load balancing, multi-path routing, access control and quality-of-service (QoS) forwarding.

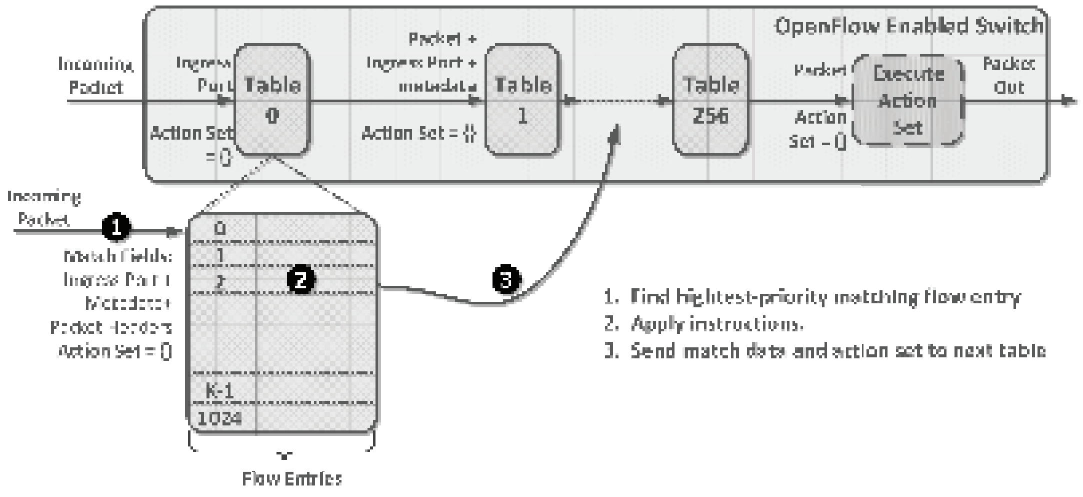

Figure 2.3 Packet Processing Through Multiples OpenFlow Tables

OpenFlow standard is rapidly evolving and is becoming more mature, from the 1.0 specification version (Pfaff *et al.* (2009a)) to the 1.3 version (Pfaff *et al.* (2012)), and the current 1.4 version (Pfaff *et al.* (2013)). OpenFlow protocol has made significant progress by: (i) enabling forwarding plane abstraction for distributed switches using multiple flow tables (See Figure 2.3), (ii) notifying the centralized controller with incoming flows at data plane and allowing it to install forwarding rules on forwarding table with matching flows and query the state of packets, (iii) extending OpenFlow features such as ability for multiple controllers connection to underlay data plane, and (iv) offering a simplified interface for caching packet forwarding rules in OpenFlow tables, and querying traffic statistics and notifications for topology changes. The OpenFlow table allows packets to be matched based on their Layer 2 (dl_{src}, dl_{dst}, dl_{type}), Layer 3 (nw_{src}, nw_{dst}, nw_{proto}), and Layer 4 (tp_{src}, tp_{dst}) headers [1], in addition to the switch port ID where the packet arrived on (See Figure 2.4). When a packet enters an OpenFlow enabled switch, it will be checked with the already installed flow entries. If there is no matching rule, the packet will be forwarded to the connected OpenFlow controller via the secured channel. Next, the controller will decide how to handle this type of packet based on the existing network applications and install the required flow entry on the corresponding switch. Once the new flow entry is installed on the switch, flows of this match are handled at the data plane without been sent to the controller.

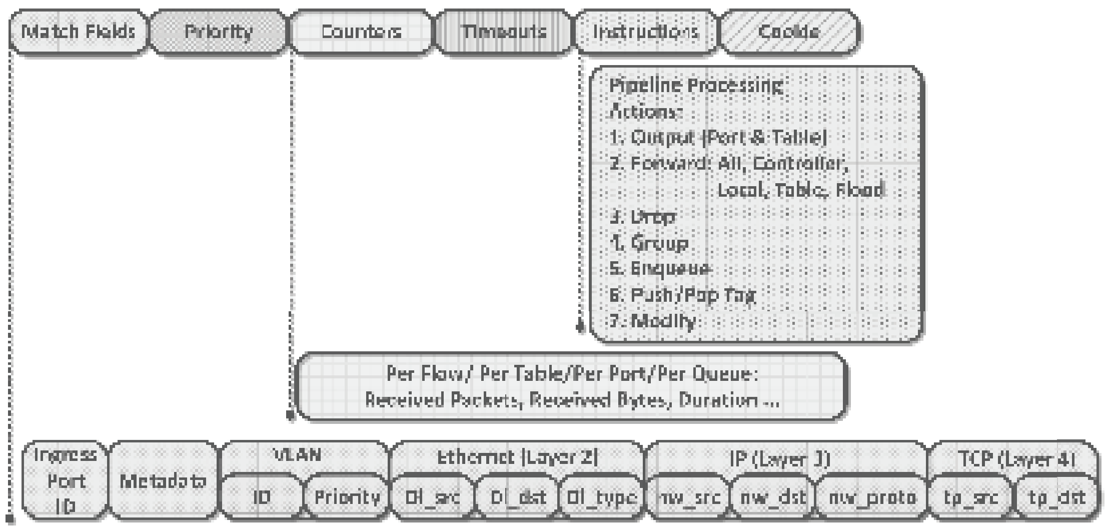

Figure 2.4 OpenFlow Flow Entry

Diverse open source deployments have been proposed to achieve the full potential for this standards. For example, between the top ranked OpenFlow controller platforms, we can cite NOX (Gude *et al.* (2008)), POX (Mccauley *et al.*), Floodlight (Erickson *et al.* (2012)), Trema (Shimonishi *et al.* (2011)), Beacon (Erickson *et al.* (2013)), Ryu (Ryu SDN Framework (2013)), and OpenDaylight (Linux Foundation (2013)). Altogether, OpenFlow and the SDN has un-

[1]dl, nw, and tp denote data link layer (MAC), network layer (IP), and transport (TCP/UDP) port layer respectively. *dst* refers to destination and *src* to source

deniably sparked a new generation of network architectures, enabling exciting projects like FlowVisor (Sherwood *et al.* (2009)), Onix (Koponen *et al.* (2010)), MiniNet (Lantz *et al.* (2010)), Maestro (Ng (2012)), Hedera (Al-Fares *et al.* (2010)) among others too numerous to list. In parallel, OpenFlow has quickly become a standard with broad industry support. It has being incorporated into switches made by virtually all major telecommunication vendors like Big Switch, Cisco, Ericsson, IBM Blade, Juniper, and others.

2.2.2.3 Forces

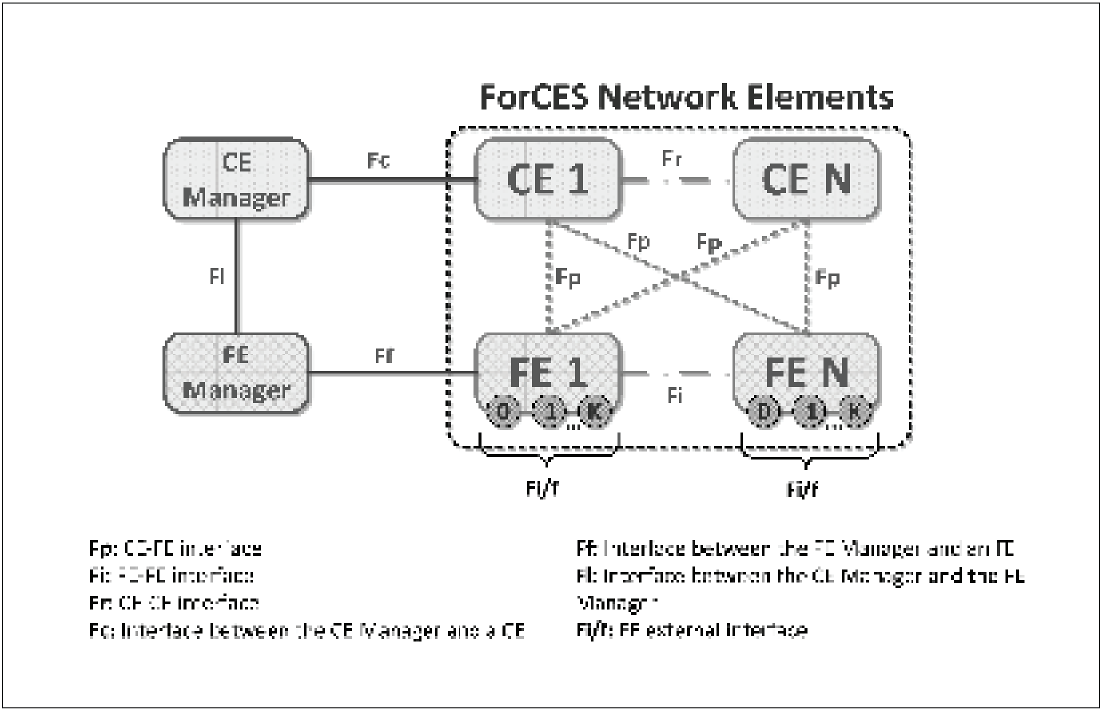
ForCES Network Elements
CE Manager
FE Manager
Fc
Ff
Fl
CE 1
CE N
Fr
Fp
Fp
Fp
Fp
FE 1
FE N
Fi
0
1
K
0
1
K
Fi/f
Fi/f
Fp: CE-FE interface
Fi: FE-FE interface
Fr: CE-CE interface
Fc: Interface between the CE Manager and a CE
Ff: Interface between the FE Manager and an FE
Fl: Interface between the CE Manager and the FE Manager
Fi/f: FE external interface

Figure 2.5 ForCES Architectural Diagram

The IETF working group defined the Forwarding and Control Element Separation architecture and protocol (ForCES, Doria *et al.* (2007)). ForCES came before OpenFlow with several years. Several works provide detailed comparisons of OpenFlow and ForCES (e.g., Tsou *et al.* (2012), Lara *et al.* (2013)). Similar to OpenFlow, it attempts to separate the control and forwarding data planes of multiple network devices (e.g., L2/L3 router, switches and security middleboxes).

Therefore, the ForCES model shares many similarities with OpenFlow protocol but they are technically different. ForCES specification allows more flexible definition of the data plane (See Figure 2.5). ForCES data plane contains directly connected Forwarding Elements (FEs) and Control Elements (CEs). FEs are flow tables with OpenFlow. Inside the same network box, each FE can be connected to one or more local CE. All local CEs are connected to distant CE Manager. FE can be managed also by distant FE Manager. However, ForCES appears to be less widely deployed than OpenFlow. In this thesis, we focus entirely on programming flow tables, which can also be implemented on ForCES using FEs.

2.2.2.4 SDN Slicing Technique

Novel approaches for multi-tenant isolation that are not based on traditional encapsulations mechanisms have became a real possibility since the appearance of SDN. SDN has allowed network researchers to try different isolation ideas via slicing. Providing multi-tenant isolation by creating virtual network slices (e.g, FlowVisor with Sherwood *et al.* (2009), HyperFlow with Tootoonchian and Ganjali (2010), Onix with Koponen *et al.* (2010)) is a very challenging issue that has not been completely solved before the introduction of SDN mechanisms like OpenFlow and ForCES. SDN has gained a lot of attention due to its flexibility for creating separate and independent virtual slices on top of underlay network infrastructures. SDN and OpenFlow enable dynamic configuration of an entire network using a control brain in an open approach thus decoupling control and data planes which represents the fundamental difference to traditional architectures. This decoupling leads to interesting properties. Both SDN planes can evolve independently by adding extension components to each plane. The scalability of SDN planes brings numerous advantages such as high flexibility and cloud resiliency. This propriety gives us the opportunity to define elastic and flexible virtual tenant slices. OpenFlow is becoming essential for network flows control because it can manipulate each incoming packet finely such that deciding the forwarding port on-the-fly based on dynamically set flow rules and altering destination address or path. In particular, the last OpenFlow version (1.3) offers a standard interface for caching packet forwarding rules in the flow table, querying traffic statistics and notifications for topology changes. To accomplish isolation requirements, Software-Defined switches like Open vSwitch (OVS) (Nicira, Pfaff *et al.* (2009b)) and controllers like NOX (Gude *et al.* (2008)) can be used to separate tenants' flows. Such components capabilities offer high flexible network control and management.

2.2.3 Security Middleboxes

Network-based security middleboxes were traditionally implemented and placed at network choke points as one class of network functions in monolithic physical boxes across data centers with significant configuration overhead and assuming the underlay network would remain static with occasionally update (e.g. Cisco ACE Web Application Firewall (Cisco (2008a))). Network providers deploy middleboxes to supplement the network with additional functionalities such as firewalling, intrusion detection and prevention, and load balancing. They rely on pre-built middleboxes that require significant manual management operations and high expertise to ensure that the desired security functions through the placement sequential are provided. Generally, the size, the number, and the places for these boxes are planned in advance and are most of the times over-estimated or under-estimated.

With virtualization, these solutions become so difficult and unfeasible due to the dynamicity of the overlay network. The underlay network is subject to the rapid change of the overlay network due to orchestration requirements between virtualized resources. With the advances in virtualization technology and the introduction of new paradigms like SDN and OpenFlow, network security providers start offering virtual middleboxes, but still in the same bloated boxes. Such complicated and overloaded virtual security appliances are not suitable for the cloud elastic nature and even more for the "pay-as-you-go" model. They are usually designed to protect against all possible attacks, while they are consuming huge memory and processing resources and generating overgrown traffic.

Recently, there has been efforts for the integration of middle-boxes into SDN exploiting the benefit of programmability for traffic steering and services chaining in aim to redirect tenant network traffic through small boxes. For example, the Slick architecture (Anwer *et al.* (2013)) proposes a centralized controller for network middleboxes, which is responsible for installing, removing and migrating security functions throughout the network. It is similar protocol to OpenFlow's, in that the controller installs the necessary action/match policy rules based on tenant security requirement. Applications can then direct the Slick controller to install the necessary functions for routing particular flows based on security requirements. Slick supports multiple concurrent group of policies or security code running on a the same middlebox. Another architecture termed FlowTags (Fayazbakhsh *et al.* (2013)) proposes an API enabling the interaction between SDN controller and middleboxes. The authors proposes FlowTags which consist of traffic flow information embedded in packet headers. FlowTags provide flow tracking and enable controlled routing of tagged packets. A clear disadvantage of FlowTags is the fact that it supports only pre-defined policies and does not handle dynamic actions. In contrast to Slick and FlowTags, the SIMPLE-fying middlebox policy enforcement (Qazi *et al.* (2013)) is an approach for using SDN to manage middleboxes deployments without any modifications to SDN capabilities or middlebox functionality, which makes it appropriate for legacy systems.

In particular, middleboxes approaches using SDN, such as CoMb (Sekar *et al.* (2012)), SideCar (Shieh *et al.* (2010b)) and Flow-Stream (Greenhalgh *et al.* (2009)) introduced novel mechanisms for providing scalable programmability in data plane and efficiently redirecting flows to network security applications using processing nodes. They can share the processing resources with middleboxes in order to increase resource utilization and decrease the number of network nodes. Kandoo (Hassas Yeganeh and Ganjali (2012)) is orthogonal to these approaches in the sense that it operates the flow redirection in the control plane, but it provides a similar distribution for control applications.

In the context of network security, Frenetic (Foster *et al.* (2011)) and Fresco (Shin *et al.* (2013a)), and CloudNaaS (Benson *et al.* (2011)) provide a language for managing middleboxes and SDN installed flow entries. Frenetic and Fresco propose a framework for composing security modules and resolving security policy rules conflicts. CloudNaaS leverages SDN by providing a flexible language for tenants to request middlebox interposition and custom end-point addressing in virtual networks. It considers the problem of traffic processing to specific nodes and middlebox placement in conjunction with cloud applications. However, it does not consider middlebox migration or dynamic network behaviour transformations.

2.3 Software Defined Components: Switches, Controllers and Applications

SDN has lead out of massive projects working on the design and deployment of OpenFlow products: hardware and virtual OpenFlow switches and OpenFlow controllers. The majority of these projects are an open source SDN initiative aiming to evolve, mature, and accelerate a common robust SDN platform. The current SDN products can be classified into three layers: OpenFlow switches, OpenFlow Controllers, and OpenFlow Applications (See Fig 2.6). In the following, we review the most popular OpenFlow-enabled implementations.

2.3.1 Software Defined Switches

2.3.2 OpenFlow Hardware Switches

The SDN paradigm has attracted a lot of attention in the networking community. Several OpenFlow-enabled Ethernet switches have been commercialized by networking equipment vendors (See Fig 2.6). Network Industries are competing to achieve high performed OpenFlow switch on one hand, and flexible configuration on the other hand. The idea of OpenFlow-enabled switch originates from the fact that most Ethernet switches use a flow table for implementing network and security services (e.g., L2/L3 switching, firewalls, NAT, QoS, . . .). Most OpenFlow switches use Ternary Content Addressable Memory (TCAM) (Pagiamtzis and Sheikholeslami (2006)) to implement multiple flow tables. TCAM was chosen as memory chip for OpenFlow hardware implementations because it supports fast lookup and efficient matching of packet with flow entries installed on flow tables comparing to Static Random Access Memory (SRAM) and others memories. However, TCAM is very expensive and has limited amount of space. It is important to use flow tables capacity efficiently otherwise it will lead to network

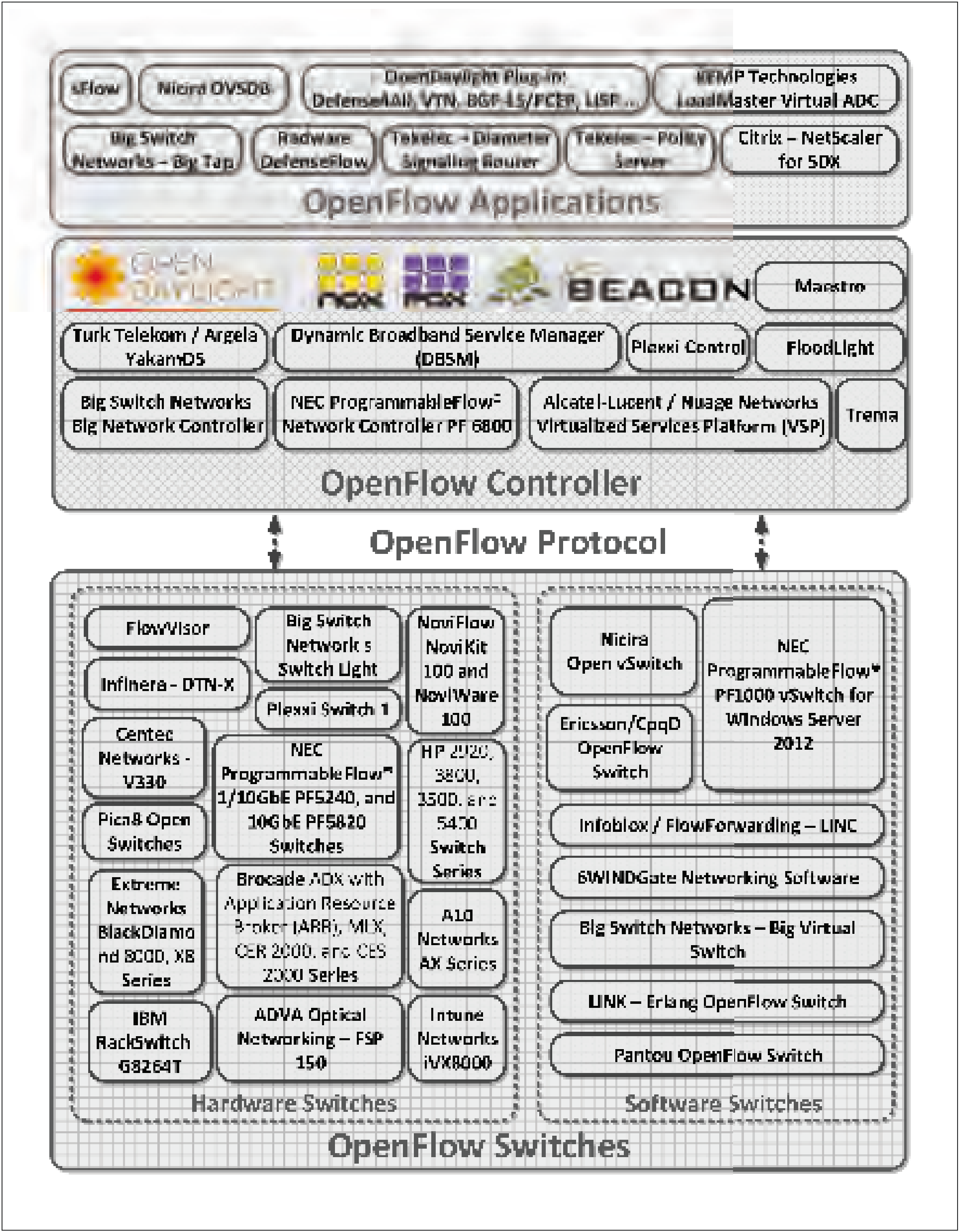
sFlow
Nicira OVSDB
OpenDaylight Plug-in: DefenseAll, VTN, BGP-LS/PCEP, LISP ...
KEMP Technologies LoadMaster Virtual ADC
Big Switch Networks – Big Tap
Radware DefenseFlow
Tekelec – Diameter Signaling Router
Tekelec – Policy Server
Citrix – NetScaler for SDX
OpenFlow Applications
OPEN DAYLIGHT
NOX
POX
BEACON
Maestro
Turk Telekom / Argela YakamOS
Dynamic Broadband Service Manager (DBSM)
Plexxi Control
FloodLight
Big Switch Networks Big Network Controller
NEC ProgrammableFlow® Network Controller PF 6800
Alcatel-Lucent / Nuage Networks Virtualized Services Platform (VSP)
Trema
OpenFlow Controller
OpenFlow Protocol
FlowVisor
Big Switch Networks Switch Light
NoviFlow NoviKit 100 and NoviWare 100
Infinera - DTN-X
Plexxi Switch 1
Centec Networks - V330
NEC ProgrammableFlow® 1/10GbE PF5240, and 10GbE PF5820 Switches
HP 2920, 3800, 3500, and 5400 Switch Series
Pica8 Open Switches
Extreme Networks BlackDiamond 8000, X8 Series
Brocade ADX with Application Resource Broker (ARB), MLX, CER 2000, and CES 2000 Series
A10 Networks AX Series
IBM RackSwitch G8264T
ADVA Optical Networking – FSP 150
Intune Networks iVX8000
Hardware Switches
Nicira Open vSwitch
NEC ProgrammableFlow® PF1000 vSwitch for Windows Server 2012
Ericsson/CpqD OpenFlow Switch
Infoblox / FlowForwarding – LINC
6WINDGate Networking Software
Big Switch Networks – Big Virtual Switch
LINK – Erlang OpenFlow Switch
Pantou OpenFlow Switch
Software Switches
OpenFlow Switches

Figure 2.6 Software-Defined Networking Products

performance degradation and over workload on the OpenFlow controller handling the packets processing.

2.3.3 OpenFlow Virtual Switches

Several OpenFlow software switches were proposed and implemented in networking community (e.g., Open vSwitch, Link, Pantou, ...). The most popular is Nicira's Open vSwitch (OVS). OVS is an open source OpenFlow software switch designed to be used as a virtual switch in virtualized server environment. It supports popular Linux-based virtualization platforms using KVM, Xen, and other hypervisors. It is used to manage virtual networking sharing the same physical server and also forward traffic between VMs and the underlay network. OVS can be extended programmatically and controlled using OpenFlow controller and OVSDB management protocol. It supports standard management interfaces such as sFlow (Wang *et al.* (2004)), NetFlow (Cisco (2008b)), and CLI.

2.3.4 Software Defined Controllers

The attraction to SDN and OpenFlow has led also to various implementations of OpenFlow controllers. SDN controllers typically consist of three distinct layers, as illustrated in Figure 2.7. The lowest layer in the control plane is the South-bound interface. It manages the connections with OpenFlow-enabled switches (hardware and software) and implements the basics of the OpenFlow protocol and other network control protocols (e.g., Netconf, OF-Config, ...). The middle layer consists of network service functions and others OpenFlow applications developed as plug-ins (e.g., monitoring, DDoS protection, ...). The upper layer, called North-bound controller API, enables the network administrator to control and monitor networks behaviour. It consists of business and network logic applications that use the last two layers to gather network intelligence, deploy the required topology and orchestrate the network's policy rules. Many SDN controllers exist as seen in section 2.2.2.2, primarily distinguished by their programming languages. Some examples are presented in Table 2.1. In particular, the project OpenDaylight (ODL), is an open source SDN initiative aiming to create, accelerate and ad-

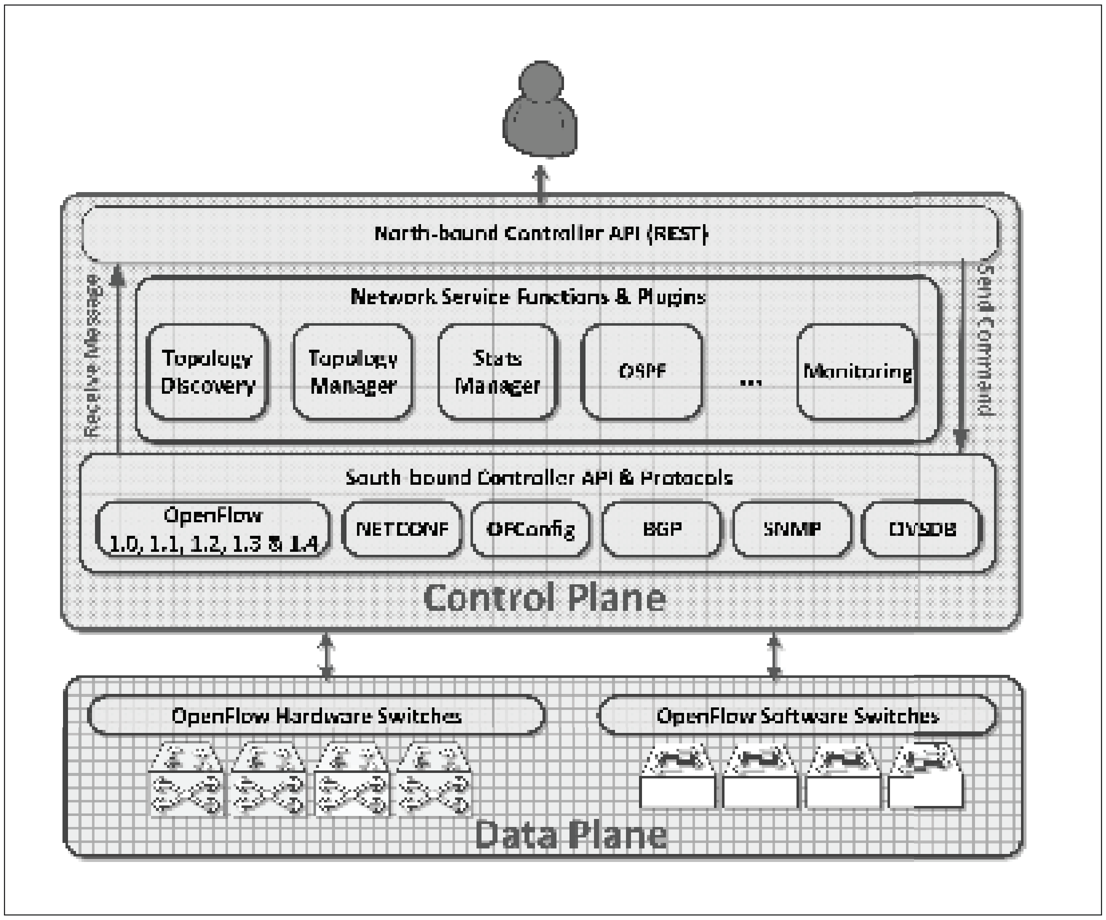
North-bound Controller API (REST)
Receive Message
Send Command
Network Service Functions & Plugins
Topology Discovery
Topology Manager
Stats Manager
OSPF
...
Monitoring
South bound Controller API & Protocols
OpenFlow 1.0, 1.1, 1.2, 1.3 & 1.4
NETCONF
OFConfig
BGP
SNMP
OVSDB
Control Plane
OpenFlow Hardware Switches
OpenFlow Software Switches
Data Plane

Figure 2.7   Typical Architecture for OpenFlow Control Plane

vance a common centralized robust SDN platform. This project has the support of many major technology companies (e.g., Ericsson, Big Switch, Brocade, Cisco, Citrix, . . .).

2.3.5   Software Defined Applications

SDN did not bring only on the decoupling between control and data planes. In addition to this interesting propriety, it delivers also the ability to externally program network nodes in real time through emerging protocol like OpenFlow. OpenFlow has emerged as powerful programming API for managing and controlling large-scale distributed network systems. It allows the network programmer to customize network switches behaviours through OpenFlow applications extending the centralized SDN controller. This open design exposes the capabilities

Table 2.1 OpenFlow Controllers and Programming Languages

Programming Language	**Controllers**
C	Trema
C++	NOX
Python	POX Ryu
Java	Beacon Maestro Floodlight OpenDayLight

of network nodes and provides for both infrastructure providers and tenants with increased flexibility in managing and deploying network services. This uniform forwarding abstraction allows to support network processing functions, such as L2 isolation, IP multicast forwarding, traffic policing, monitoring, load balancing, shortest path routing, access control and QoS forwarding.

The concept of a programmatic network has extended the basic networking services with more personalized and sophisticated network applications that were not realisable before the introduction of OpenFlow-enabled applications. This includes Onix as distributed control platform (Koponen *et al.* (2010)), Veriflow as monitoring and packets verification application (Khurshid *et al.* (2012)), FortNox, FRESCO and Avant-Guard as security enforcement applications (Porras *et al.* (2012); Shin *et al.* (2013a,b)), flow scheduling and multipathing with Hedera (Al-Fares *et al.* (2010)), VM Migration (Arora and Perez-Botero) and service chaining and network steering (Gember *et al.* (2013); Qazi *et al.* (2013)).

Recently with the introduction of Network Function Virtualization (NFV) (Brief, 2014), the notion of a composed OpenFlow applications has recently coalesced into the more general notion of SDN-enabled applications, in which the network is treated as another programmable resource for delivering multiple and independent Virtual Network Functions (VNF). This is provided and facilitated by abstraction layers like OpenDaylight in which an open north-bound APIs are provided. These APIs allow to include pluggable modules to perform needed network tasks such as dynamic bandwidth reservation, virtual networks isolation, chaining and traffic steering, etc.

2.4 Multi-tenant Network Isolation

The advancement of network virtualization with the Overlay Virtualization Technologies (OVT) and Software-Defined Networking (SDN), outlined in chapter 1 and previous sections of this chapter, has led to interesting approaches for achieving multi-tenant network isolation. In the following, we are resuming the most important research and industrial approaches for reaching the isolation goal.

2.4.1 Research Isolation Approaches

Some approaches concerning the concept of multi-tenancy isolation have been proposed for network virtualization before the introduction of SDN. For example, Cabuk et al. (Cabuk *et al.* (2010)) presented prototype of automated security policy enforcement for multi-tenancy based on the concept of Trusted Virtual Domains (TVDs). Their approach allows to group VMs belonging to a specific tenant dispersed across multiple Xen Hypervisor into a TVD zone. Tenant's requirements for isolation are automatically enforced by a privileged domain (DOM-0). Such solution offers tenants separation using VLAN, EtherIp and VPN tagging. Their solution presents a significant step towards: (i) tenant transparency by automating the deployment and mapping of tenants desired network topology, and (ii) isolation elasticity by orchestrating TVDs through a management framework that automatically enforces isolation necessary changes (e.g., load balancing, migration) among different hosts' hypervisors.

Based on network virtualization paradigms, SDN comes with better alternative than using the TVD privileged domain, which is not recommended for the security. By decoupling control and data planes, SDN offers a new way to create transparent and isolated virtual networks by dividing, or slicing, the network resources.

In the SDN space, there are several temptations to achieve a complete isolation solution suitable for multi-tenant data center. We can classify these software-defined techniques depending on the emplacement of their isolation engine as the following:

2.4.1.1 SDN Hypervisor

FlowVisor (Sherwood *et al.* (2009)) is a one of popular project working on developing virtual network slicing in hardware programmable router. The FlowVisor hypervisor is implemented as OpenFlow proxy to allow multiple researchers use the same network resources using multiple controllers and enforcing strict traffic isolation between tenants' controllers. It enables filtering events to controllers and masking messages to switches. Such mechanism aims to separate researchers' slices called "Flowspaces" and let each slice managed by a single controller. In addition, FlowVisor manages provisioning of shared resources (e.g., bandwidth, controller) using heuristics to estimate the amount of each processing resources needed by each slice. Nevertheless, these slices are completely independent and FlowVisor does not consider the virtual tenant networks scalability and inter-domains collaboration requirements.

HyperFlow (Tootoonchian and Ganjali (2010)) has a complementary approach. It introduces the idea of enabling the interconnection between separated slices. HyperFlow uses multiple controllers to manage each tenant slice following the same concept as FlowVisor. The connection between slices is provided by a shared publish/subscribe system because controllers use to update the network state and send commands to the other controllers. This mechanism does not support routing over slices and either the network scalability.

Similar to HyperFlow, Onix (Koponen *et al.* (2010)), is also one of the most prominent examples of systems that provide isolation using OpenFlow/SDN. It represents a distributed control platform that facilitates implementation of distributed control planes in aim to enable tenant to create virtual topologies that will be mapped to physical elements. It provides control applications with a set of general APIs to facilitate access to network state, which is distributed over Onix instances.

Previous SDN researches' works have successfully achieved the transparent isolation in virtual data center networks which has been a long goal of the infrastructure providers. However, it lacks the ability to use OpenFlow slices within individual SDN programs. They have made use of centralized or distributed controllers to achieve strict isolation between different tenant's slices without addressing tenant requirements of today (e.g., slice scalability, inter/intra slices communication and collaboration, load-balancing, migration, . . .) and arriving to a consistent solution that consider both networks and SDN control plane scalability bottleneck (See 1.3.3 for more details).

2.4.1.2 SDN Middle-boxes

Before the spread of network virtualization, isolation was being enforced through special devices or middle-boxes located protecting tenant applications. Recently, there have been efforts to bring this practice to virtualized environments which still valid to provide the necessary network security functions that can guarantees the defense for multi-tenant data center. For example, Anwer et al. propose (Anwer *et al.* (2013)) sLICK architecture exploiting the benefit of SDN to redirect tenant traffic through middle-boxes. sLICK is based on a centralized OpenFlow controller which is responsible for installing and migrating security functions onto tenant allocated middle-boxes. Another similar approach is FlowTags (Fayazbakhsh *et al.* (2013)), proposing the interaction with the controller through FlowTags application programming interface using an embedded traffic flow information in the packet headers in aim to track the flow and enable controller traffic redirection based on tagged packets. Unlike sLICK and Flowtags, requiring modifications to SDN capabilities, SIMPLE architecture (Qazi *et al.* (2013)) is congruent to standards. Simple authors present SDN-based policy enforcement layer for efficient middlebox-specific traffic steering.

A clear disadvantage of this architecture is the fact that it works with only pre-defined policies and currently does not handle dynamic actions. Based on these approaches, it would appear that tenant isolation requirements can be achieved using a simple and traditional practice by redirecting traffic to appropriate security devices. However, satisfying tenant requirement is not straightforward as that. This practice brings other questions to be discussed before talking about traffic steering between middle-boxes: (i) which the appropriate placement of these middle-boxes? (ii) What is the penalty that can be tolerated when traffic is redirected through additional nodes? and (iii) finally how these middle-boxes can follow the movement of protected VMs?. Such questions have not yet been answered.

2.4.1.3 SDN traffic Isolation through OpenFlow Controller

A variety of SDN controller platforms have been developed as seen in section 2.2.2.2, including Beacon (Erickson *et al.* (2013)), NOX (Gude *et al.* (2008)) and POX (Mccauley *et al.*), Maestro (Ng (2012)), Nettle (Voellmy and Hudak (2011)), Ethane (Dixit *et al.* (2013)) and OpenDaylight (Linux Foundation (2013)), but none of them come with traffic isolation for network applications running on the shared infrastructure. In reason to overcome this challenge, a number of systems were proposed to provide traffic isolation in multi-tenant data center through OpenFlow controller:

- Splendid Isolation (Gutz *et al.* (2012); Schlesinger *et al.* (2012)): is one notable contribution. Gutz et al. sustain that old isolation practice increase the complexity of the network configuration. They propose building network slices to isolate traffic using high-level language-based security called “Pyretic” enabling flow-based policy enforcement implemented as NOX application to forward and add automatically OpenFlow rules to the virtual switches. More recently, they proposed also (Monsanto *et al.* (2013)) an extension to their work that supports the idea of parallel and sequential composition of tenant slices’ rules to support multiple concurrent and parallel tenant network tasks execution in SDN platform.

- Fresco (Shin *et al.* (2013a)): introduced a new OpenFlow security application development framework in SDN extending FortNox (Porras *et al.* (2012)), a security enforcement kernel which handles and solves possible conflicts when adding new rules on OpenFlow switch. The idea behind Fresco is to solve several key issues during combining security modules on demand, which can be incorporated as an OpenFlow application. They presented a library of scripting language for reusable modules which can be used for the detection and mitigation against attacks.

- NetLord (Mudigonda *et al.* (2011)) and VL2 (Greenberg *et al.* (2009)): provide isolation through OpenFlow controller using VLAN encapsulation. NetLord provides tenants network abstractions, by fully and efficiently virtualizing the address space at both L2 and L3. VL2 supply tenant security functions through a directory service that makes address assignment independent of the underlying topology.

- CloudPolice (Popa *et al.* (2010)): is different from the previous approaches depending on centralized OpenFlow controller. CloudPolice designed a distributed access control mechanism providing tenant isolation for Cloud Computing using a more general end-host based architecture for enforcing security properties implemented as hypervisor on virtual switches interacting with the controller.

- Seawall (Shieh *et al.* (2010a)): goes beyond basic traffic isolation objective and address performance isolation as well by proposing weight abstraction that provides dynamic fair sharing of data center networks' resources between tenants. Authors presented algorithms that allocate and manage bandwidth at run-time that guarantee that each tenant receives a fair proportion of the available capacity.

2.4.2 Industrial Isolation Approaches

Industrial isolation approaches are ideally based to provide the transport by the physical network and the VM service by the hypervisors. The traditional slicing technique has employed VLAN to isolate tenants' machines on a single L2 virtual network. However, this simple isolation approach depends heavily on routing and forwarding protocols and is not easily configured. VLAN management complexity imposes limitations on cloud nature and services. More importantly, VLAN lacks scalability resulting to a segmentation capacity limit to 4K tenants.

Table 2.2 summarizes the competing multiple overlay virtualization approaches as alternative technologies to substitute VLAN. These technologies have been proposed within industrials that are in contrast with the open standards used in OpenFlow solutions. They use Open vSwitch (OVS) plus typical L2/L3 physical switch to provide virtual networking isolation and tunneling unlike VLAN which ignores and dumbs OVS (e.g., VMware's vCloud Director Networking Infrastructure (vCDNI) (Krieger *et al.* (2010)), HP's NVGRE (Sridharan *et al.* (2013)), Nicira's Network Virtual Platform (NVP) (Rosenblum (1999))). More recently, Cisco's Virtual eXtensible LAN (VXLAN) (Mahalingam *et al.* (2012)) has been adopted within several network vendor for scalable LAN segmentation and automated provisioning of logical networks between data centers across L3 networks.

Table 2.2 Comparison of industrial network virtualization solutions

OVT	**Encapsulation**	**Control Plane**	**VLAN limit**	**Bridging**	**VM MAC visible**	**virtual Network Flooding**	**Multicast Flooding**	**Network State**
VXLAN	L2-over-UDP	✗	✗	✗	✗	✓	Some	IP Multicast
NVGRE	MAC-over-IP	✗	✗	✗	✗	✓	✓	✓
vCDNI	MAC-in-MAC	✗	✗	✓	✗	✓	✓	Hypervisors' MAC
Nicira NVP	MAC-over-IP	✓	✗	✗	✗	Some	✗	✗
ODL VTN	–	✓	✓	✓	✗	✗	✗	✗

The major inconvenience of overlay technologies (e.g., VXLAN, NVGRE, vCDNI) is the missing of control plane. They can support only one Overlay Virtualized Network (OVN) or slice due to the lack of scalability. They are competing encapsulations with minor technological difference (e.g., TCP offload, load balancing, Security features) and no one supported by legacy systems. The key mechanism of scalable architecture is the control plane which maps remote VMs' MAC address into a transport network IP address. These technologies face a crucial problem to determine the VM destination's IP address.

Nicira's NVP seems to be solving the problem with better scalable solution than VXLAN using an OpenFlow controller to install MAC-to-tunnel forwarding rules on OVS. Nicira's NVP extends the SDN/OpenFlow standards and introduces OVS database (OVSDB) protocol to configure OVSs. Through user-space OVSDB, it keeps track of the topology existing tunnels. This approach scales better than VXLAN but causes network overall performance degradation.

The OpenDayLight project has a complementary approach to Nicira's NVP. Among OpenDayLight proposals, we find Virtual Tenant Network (VTN) which is recently proposed to fill the gap of existing OVN technologies by building an OpenDayLight application to support multi-tenancy. OpenDayLight VTN used OpenFlow ports, VM's MAC and VLAN mapping to isolate tenants' flows and enabling to configure virtual tenant networks across multiple SDN controllers. VTN design relies on OpenFlow 1.0 specification and do not take benefit from 1.3 extended features such as ability for multiple controllers connection to OVS. It uses a VTN manager extended to OpenDayLight controller and an extern VTN coordinator to manage and synchronize distributed OpenDayLight controllers. Despite the efforts of the OpenDayLight project, the bridge between the SDN paradigm and OVN is no sufficient to provide today cloud's tenants requirements.

2.5 Network Management Systems

As presented in last chapter in section 1.3.2, network management systems are facing several roadblocks even after the introduction of SDN paradigm. The explosion of cloud applications and services is driving network community to rethink of management technologies. Network management is posing new challenges that need to be addressed in order to fully achieve an flexible and reliable networking environment suitable for the elastic nature of cloud computing. It is becoming primordial that future networks should be transparent, self-controlled and self-manageable. It is widely accepted that next-generation of multi-tenant network will require a greater degree of service awareness and optimal use of network resources.

Next, we will discuss the management challenges for the emerging SDN paradigm. We present Autonomic management as the last trend in network management and overview some approaches enforcing this technology.

2.5.1 Management of Software-Defined Networks

SDN appears to be a major evolution towards network programmability. It has become extremely popular as a means to program and customize network behavior. However, SDN relies only on the virtualization layer, which decouples the control and data planes. The management layer in SDN networks seems to be ignored or not addressed properly. The major issue of SDN is the huge load in the centralized controller, which has to perform all control and management tasks in real time. Moreover, in case of using a cluster of OpenFlow controllers, each controller has to handle the monitoring of internal resources such as policy rules database and sharing the status of the controlled nodes. For example in FlowVisor, the virtualization layer was used to slice the network and provide isolated “Flowspaces” to different researchers sharing the same infrastructure. One of the main issues in which FlowVisor must deal is managing among multiple Flowspaces. These slice are managed by a centralized controller used to separate bandwitdh, CPU and Openflow tables’s memory allocated for each group of researcher through a series of checking loops. Also, the centralized controller is used to provision the shared resources using heuristics and estimate the amount of each processing resources needed by each space. In addition to basic controller tasks (e.g., create, remove, update, and delete flow entries), other isolation mechanisms were added in control plane including the verification of flow entries limits in flow tables allocated for each “Flowspace” and the rewriting of control messages originated at a particular portion to prevent conflicts with other tenants.

Commonly, it is admitted that SDN/OpenFlow could be complementary to classical management technologies, but they have limited management capabilities and completely ignore the lasted advances in network management such as autonomic computing and communication concepts. The Autonomic management represent an opportunity for evolving the management capabilities in SDN networks. It allows for embedded management of all network nodes and gradual implementation of management functions providing their code life cycle management as well as the ability to on-the-fly code update. So far, only few papers on this subject were published: Wendong *et al.* (2012); Kim (2013); Cannistra *et al.* (2014); Li *et al.* (2013)

2.5.2 Autonomic Management Background

Autonomic management is one of the ascending research fields in future network control and management. Self-Awareness, Self-Configuration, Self-Manageable, Self-Optimization and Self-Organization are a series of autonomic management attributes. The objective of introducing these self-* autonomic attributes into SDN networks is to decrease the load on the centralized OpenFlow controller and move the management functionalities to an external management entity. The ultimate objective is to create self-manageable virtual networks for multiple tenants, overcome the rapidly growing management complexity and enable both tenant and network scalability.

Very often, any management architecture uses a local agent in network device connected to external manager. Autonomic management approaches vary in how managers and agents are organized. Some approaches rely on a centralized manager responsible for all management tasks, while others uses a cluster of distributed managers to share the task of managing the infrastructure. Next, we have selected one recent project in network management to be representative of autonomic management model.

4WARD: Distributed Autonomic Management Approach

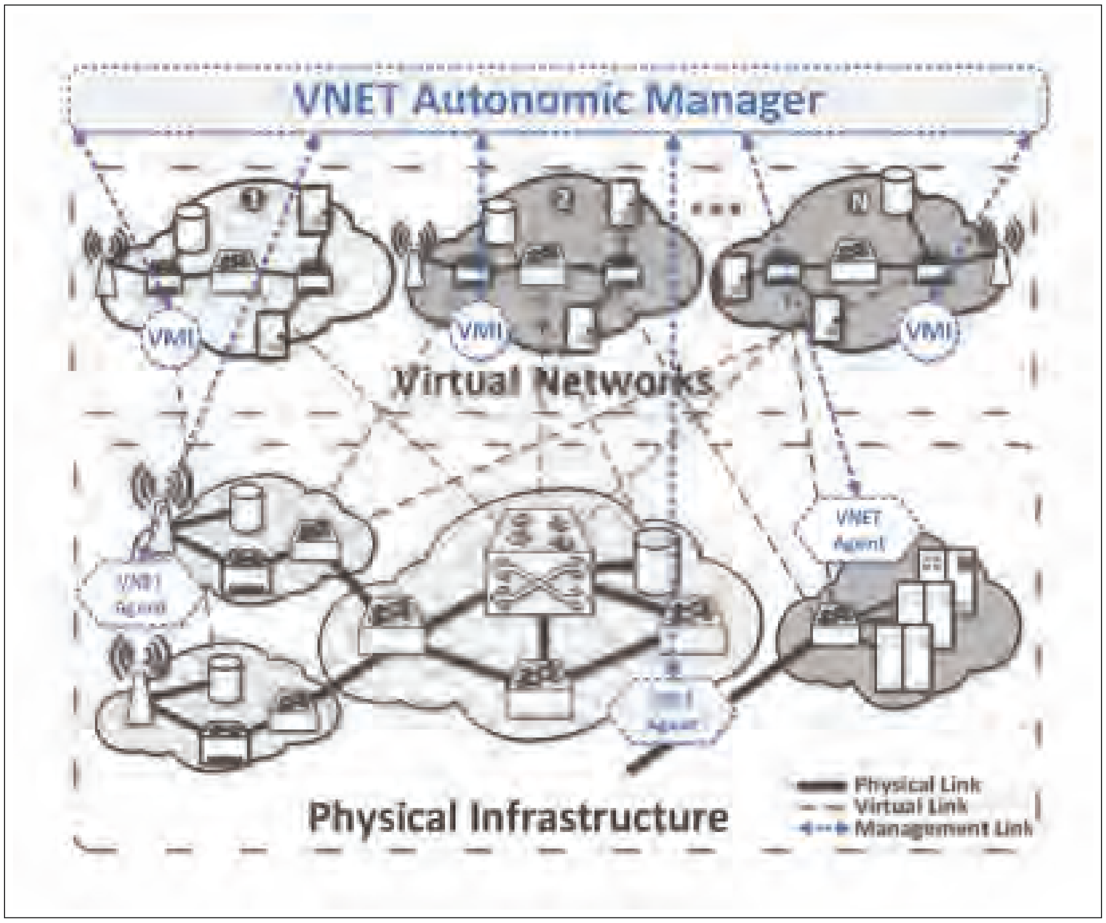

Figure 2.8 4WARD VNET Architecture

The 4WARD (FP7-4WARD (2008); Niebert *et al.* (2008); Correia *et al.* (2011)) is one of the attractive network management project. 4WARD aims to create dependable and interoperable networks providing direct, transparent and faster access for multi-tenancy. Its goal is to design future networks adaptable to the current and future needs with acceptable cost. Among 4WARD proposals, we can find VNET framework for network virtualization. VNET is designed to manage multiple virtual networks sharing the same infrastructure. They have developed a systematic and general approach for managing and controlling the shared virtual resources using standardised interfaces. Based on high level abstraction and using the control and management interfaces, VNET is developing a systematic approach to enable dynamic instantiation of virtual networks and enabling the on-demand network deployment and scalability. This framework includes also the discovery of available resources as well as dynamic provisioning of resources needed for virtual networks. Once virtual networks have been instantiated, management mechanisms are required to control their resources and support dynamic update during the lifetime of the virtual networks. Figure 2.8 depicts the 4WARD management framework along with the relationships among VNET components. VNET agents are placed at the physical infrastructure and linked to the physical nodes. It is responsible for providing updated information about physical and virtual resources to the centralized manager. VNET offers a Virtualization Management Interface (VMI) based on XML-RPC and placed at virtual networks. These interfaces orchestrate virtual networks and update their configurations on-demand.

2.6 Summary and Conclusions

In this chapter, we studied the literature in order to realize what is the current practices in multi-tenant network. Particularly, we focused on the criteria of scalability, flexibility and automatism in multi-tenant network isolation approaches.

We first have introduced the used architecture for supporting multi-tenancy, where the server and network virtualization and SDN are the key technologies to provide multiple and independent virtual networks. We then presented all required definitions and concepts for previous multi-tenant isolation approaches. We have introducted also the typical organization of SDN planes and related components: OpenFlow hardware and software switches, OpenFlow controller, and OpenFlow applications.

Furthermore, we have studied the existing research and industrial solutions in order to differentiate and separate the traffic between the different tenants. These solutions are resorting to SDN slicing technique, security middleboxes or overlay protocols. They pave the way towards the security of the cloud networks, but none of them was able to achieve our isolation goals, as we consider the elastic nature of the cloud and the urgent need to support network scalability. The common drawback between research and industrial multi-tenant isolation approaches is focusing on exploiting SDN capabilities to provide only strict isolation without considering the extended requirements added for the security such as node migration and virtual tenant networks scalability. They have tackled the basic isolation functionality that guarantees the separation between tenants' traffic as one block of security application and assuming the tenant security defense would remain static with occasionally update. However, with one complex block of security configuration including a mix of tenants' policy rules, they do not address how to create and build a single virtual tenant network out of multiple, independent, reusable network policies that can be suitable for the nature of cloud computing that can be merged, separated and moved with cloud applications. We believe that multi-tenant isolation should be defined with high level abstraction at the application layer before being mapped on the underlay layer. This abstraction can summarize all the tedious details related to the security configuration implementing the desired tenant isolation and releasing the network programmers from having to reason, analyze and verify the buggy complex block of security configuration and labored architecture issues such as tenant's VMs and related security boxes placements (Qazi *et al.* (2013)). We believe that our abstraction should be based on modularity which is the key of managing complexity in any software system, and SDN is no exception. We believe that we can lift several roadblocks by combining SDN network virtualization approaches and overlay techniques. More precisely, SDN addresses flexibility, isolation, and manageability through network slices, but suffers from scalability limitations. Network overlay techniques overcome scalability issues of cloud isolation mechanisms, but remain limited to single slice. In addition, both solutions do not offer a complete transparent isolation and lack the ability to use multiple network slices within individual SDN programs. We can overcome their respective limitations into a unified design and arrive to a consistent solution that consider both networks and OpenFlow control plane scalability bottleneck. This combination enables to reap the benefits of SDN slices while preserving scalability. This yields a network virtualization architecture that is both flexible and secure on one side, and scalable on the other.

Finally, we described one autonomic management approach (4WARD), and present the principal components and related relationships to achieve a self-manageable, self-configurable and self-ware multi-tenant environment. By introducing the self-* autonomic attributes into SDN-based architecture, we believe that programmable capability of the control plane could be enhanced to an environment aware programmable capability. It will be efficient to release the centralized controller from the extra management and control tasks (e.g., monitoring, provisioning, slice boundaries checking . . .) and keep just the basic control functionalities (create, read, update and delete flow entries on OpenFlow tables). This can also improve the controller performance, decrease the overhead added for packet processing and improve the control intelligence in multi-tenant network.

In this thesis, we develop a network management and security system for SDN architecture, called OpenNMS. It takes a step towards providing elastic, scalable and autonomic isolation for multi-tenancy by presenting L2 isolation with high level abstraction for Virtual Tenant Slice (VTS) that guaranties the tenant security as the basic objective. One master application on the control plane will orchestrate, build and map small tenant distinct portions representing the required VTSs into the virtual and underlay networks. VTS space on data plane will be allocated for each tenant for building his own small security boxes out of multiple, independent, reusable network policies included in the VTS abstraction. Thus, as results of the simple and flexible VTS definitions model, we describe novel features added for the isolation: Split, Merge and Migrate (SMM) that can be well suited for the tenants' requirements in such dynamic environment.

CHAPTER 3

VIRTUAL TENANT SLICES ABSTRACTION MODEL

In the previous chapters, we studied the literature on the research and industrial mechanisms used in order to provide multi-tenant network isolation. Although these techniques are tackling a basic, inflexible and strict isolation objective, we believe they are no longer suitable for today's cloud applications. Indeed, these solutions are not remaining consistent through tenant networks boundaries transformation and network nodes migration. The current SDN slicing solution shows scalability issues and overlay protocols remain limited to single virtual network.

OpenFlow Slice Benefits– Among the studied approaches, the SDN slicing technique is becoming very popular due to its high flexibility. SDN-based virtual slice has introduced several advantages for cloud applications. First, it allows multi-tenants to manage the shared network resources using multiple OpenFlow controllers. Second, it simplifies packet forwarding mechanism in the network without adding any extension to the packet header. OpenFlow is becoming essential for controlling network flows. OpenFlow can manipulate each incoming packet finely such that deciding the forwarding port on-the-fly based on the set of flow rules, and altering destination address or path. In particular the last OpenFlow "1.3" version (ONF (2012)) offers a standard interface for caching packet forwarding rules in OpenFlow tables, and querying traffic statistics and notifications for topology changes. Moreover, OpenFlow controller enables variety of actions including the basic ones (e.g., push, pop, copy, drop, forward, and header modification). Finally, OpenFlow enables both proactive and reactive behaviors. OpenFlow rules can be set at the initial slices deployment or can dynamically created on flows arrivals. However, the remain challenge to arrange isolation between multi-tenants virtual networks is the correct configuration of all OpenFlow switches and controllers.

Next, we build on what we have learned in the previous chapters in order to develop our own isolation solution by combining SDN slicing technique and overlay technologies into unified design. We want to enhance the SDN slicing technique flexibility and performance while conserving its advantages for creating secure virtual networks. Furthermore, we intend to give the cloud provider the possibility to dynamically enforce the desired isolation and boundaries in a multi-tenant network as well as providing transparent and self-manageable virtual networks for cloud's tenants.

In order to do so, we will first define the isolation goals which we want to reach with our solution. It will help us to define the Virtual Tenant Slices (VTSs) abstraction model. Then, we will analyze and refine our objectives regarding the isolation solution that we plan to develop based on both tenants and cloud provider current requirements. Next, we will present the L2 isolation model, complete definitions and proprieties of the proposed model, which will include and summarize the proposed solution. We will also go through the use cases once again in order to see the behavior of our model. Lastly, we will precise how we did to meet the planned objectives and which technologies we plan on using. An early version of this work appeared in IC2E conference (Fekih Ahmed *et al.* (2014)).

3.1 Our Isolation Goals: Flexible, Transparent, and Scalable

Software-Defined slicing technique typically enforces strict and inflexible traffic isolation using a hypervisor sitting between SDN planes as extension to OpenFlow switches. SDN slices enforce basic isolation properties by translating, inspecting, rewriting, and policing OpenFlow entries received from tenant controllers (e.g., no flows originating from tenant$_1$ slice can reach another tenant$_2$'s slices). This isolation layer added between SDN planes provide a mean for infrastructure provider to limit the scope of tenant, restricting the network resources that may be affected by the tenant configuration. However, SDN slices suffer from scalability limitations. To facilitate the tenant to reason about his virtual networks limits and the complex isolation mechanism, overlay network virtualization approaches resort to low-level encapsulation mechanisms (e.g., VLAN, GRE) that help to understand his virtual network compositions and boundaries. This basic technique limits the tenant access to the allocated network resources, remains limited to a single overlay network and increases the complexity of management.

Achieving multi-tenants flexible isolation suitable for cloud tenant requirements is far from straightforward. The opportunity cost of the successful implementation of SDN slices for cloud computing is to find an alternative design supporting transparent, flexible and scalable isolation on one side, and reducing the complexity of management on the other side. We take a step towards providing rich and extensible SDN slices by presenting high level Layer 2 isolation abstraction which divides the shared network into VTSs. We lift SDN and overlay technologies limitations by combining them into an unified design. With such combination, we enable to reap the benefits of SDN slices while preserving scalability. The results are recursive, solving overlay approach by adding the missing control plane, and overcoming both virtual networks and SDN control plane scalability bottleneck by coupling SDN slices with overlay protocols. Furthermore, providing a high level abstraction for multi-tenant network will help us make the isolation suitable for the cloud elastic environment and applicable for general tenant requirements. This approach will enable tenant to: (i) easily update and scale (in/out) his allocated virtual networks by adding new virtual resources, (ii) share allocated switches ports, tables, and links between his slices, and (iii) easily transport his VMs to new locations while maintaining the isolation proprieties.

3.2 Our Layer 2 Isolation Model

Strange as it may seem, but despite the huge number of research papers adopting the SDN slicing technique (e.g., Sherwood *et al.* (2009); Koponen *et al.* (2010); Tootoonchian and Ganjali (2010); Shin *et al.* (2013a); Schlesinger *et al.* (2012); Monsanto *et al.* (2013), . . .), none of them has attempted to define and apply formal model for SDN slices. These researchers have succeeded to provide only formal techniques to verification and compilation of SDN slices behaviour. We believe the formal model for SDN slices and topology can substantially enhance the portability of cloud applications and tenants' networks running on complex software and hardware infrastructure. It can also enable the association of the higher-level operational behavior with cloud infrastructure management.

3.2.1 Layer 2 Isolation Model's Definitions and Syntax

Model Notations

p	OpenFlow Port
t	OpenFlow Table
gt	OpenFlow Group Table
c	OpenFlow Controller
fe	Flow entry
pf	Permitted flow
sw	Physical OpenFlow switch
h	Open Virtual Switch installed on host
dp	Datapath
map	Virtual-To-Underlay Resource Mapping

In the following, we present our novel L2 isolation model of Virtual Tenant Slices (VTSs). Figure 3.1 depicts our L2 isolation model in which we provide a high level abstraction for virtual tenant networks. Today's SDN slicing techniques miss such abstraction for L2 isolation in multi-tenant network. This model will be further formalized and extended in our future work.

The first layer, Data Plane, contains all OpenFlow switches that can exist in multi-tenant data center. It can be physical OpenFlow switch (*sw*) or virtual one (*h*) installed on hosts. Each

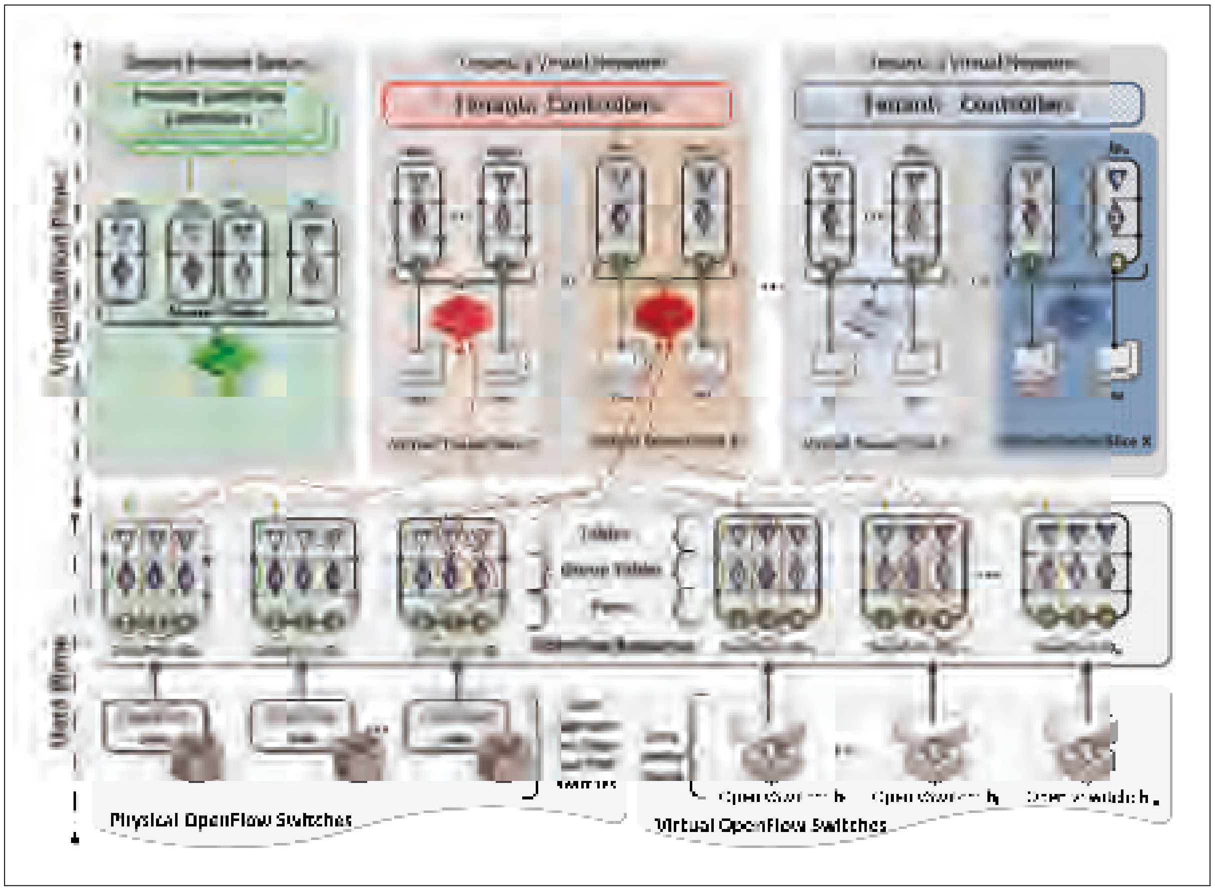

Figure 3.1 Open Network Management and Security's L2 Isolation Model

of these OpenFlow-enabled switch represents a "datapath" (dp). The set of all datapaths (dp) is denoted as D=$\{dp_0, dp_1, \ldots, dp_u\}$; u = $|D| \geq 1$. Each data path ($dp_{i;i=1..u}$) contains a set of OpenFlow resources including Ports, Tables and Group Tables. It presents a high level abstraction for OpenFlow switches's resources. Each flow table contains a set of flow entry which define packets's fields to match, and an action (such as send out-port, modify packet's field, or drop packet). When an OpenFlow switch receives a packet that has never seen before and it has no matching flow entries, it sends this packet to the controller. The controller then makes a decision on how to handle this packet. It can drop the packet, or it can add a flow entry directing the switch on how to forward similar packets in the future.

In the second layer, Virtualization Plane, we allocated for the service provider a set of the OpenFlow resources on each data path and we share the remaining resources between multi-tenant network. A set of OpenFlow controllers is provided to isolate and manage tenants networks and slices. Each tenant slice be composed of multiple and distributed OpenFlow resources. They can be physical or/and virtual OpenFlow resources. Virtual tenant networks is a set of Virtual Tenant Slices (VTSs) distributed over the data center. Inter and Intra VTSs communication is controlled by tenant dedicated OpenFlow controller.

Given a set of OpenFlow physical switches SW and a set of virtual switches H where:

$$SW = \{sw_0, sw_1, \ldots, sw_n\};\ n = |SW| \geq 1\ ,\ H = \{h_0, h_1, \ldots, h_m\};\ m = |H| \geq 1 \text{ and } SW \cup H = D$$

Let's f a function specifying the category of data path ($dp_{i;i=1..u}$) that may either physical or virtual.

$$f : D \mapsto \{0,\ 1\}, \text{ where :}$$

$$f(dp_i) = \begin{cases} 1 & \text{if } dp_i \text{ is an OpenFlow hardware switch} \\ 0 & \text{if } dp_i \text{ is an OpenFlow virtual switch} \end{cases}$$

Each physical or virtual OpenFlow switch represents one datapath ($dp_{i;i=1..u}$). Each dp_i is composed of subset of OpenFlow ports, tables and group tables, denoted respectively $P_+^{(i)}$, $T_+^{(i)}$,and $GT_+^{(i)}$.

$$dp_i = \{P_+^{(i)}, T_+^{(i)}, GT_+^{(i)}\}\ /\ P_+^{(i)} \subset P,\ T_+^{(i)} \subset T, \text{ and } GT_+^{(i)} \subset GT$$

where:

- P=$\{p_0, p_1, \ldots, p_k\}$; $k = |P| \geq 1$: the set of all OpenFlow switches' ports in the multi-tenant data center. These ports can be classified into two types. The first class (P_{ext}) is composed of the external ports which link between switches ($p_{i;i=1..k} \in P_{ext}$). It must be shared between multi-tenant network to forward packets to external networks or neighbours. The second class (P_{in}) regroups the edge or internal port which links OpenFlow port to virtual machine's virtual network interface (VNI) ($p_{i;i=1..k} \in P_{in}$). Therefore, P can be defined as $P = P_{in} \cup P_{ext}$.

A port of an OpenFlow switch is considered as boolean vector: $p_i = (status,\ type,\ dedicated) \in \{0,\ 1\}^3$. The first class *status* indicates the activation status of port ($p_i(0) = 0$ for inactivated port and $p_i(0) = 1$ for activated one). The arriving packets to this port will be automatically dropped. For the *type* value, If $p_i(1) = 0$ then $p_i \in P_{in}$, otherwise $p_i \in P_{ext}$. Finally, the *dedicated* class indicates if the port is dedicated only for one tenant ($p_i(2) = 1$) or shared between multi-tenant network ($p_i(2) = 0$).

- T=$\{t_0, t_1, \ldots, t_h\}$; $h = |T| \geq 1$: the set of OpenFlow tables where:
  $t_{i;i=1..h}$ is boolean vector: $t_i = (status,\ dedicated) \in \{0,\ 1\}^2$. $t_i(0) = 0$ points out inactivated table and $t_i(0) = 1$ for activation. The *dedicated* class indicates if the table is dedicated only for one slice ($t_i(1) = 1$) or shared between multiple tenant slices ($t_i(1) = 0$).

- GT=$\{gt_1, gt_2, \ldots, gt_l\}$; $l = |GT| \geq 1$: the set of OpenFlow group tables [1]. Each $gt_{i;i=1..l}$ is handling similar or common actions. Similar to OpenFlow table, gt_i is boolean vector: $gt_i = (status,\ dedicated) \in \{0,\ 1\}^2$.

We define a set of controllers C=$\{c_1, c_2, \ldots, c_r\}$; $r = |C| \geq 1$ that can be connected to one or more OpenFlow switches. Each controller, $c_{i;i=1..r}$ can manage one or more VTS.

Multi-tenant Network is a set of virtual networks dedicated for multi-tenant, denoted as $VTM = \{VTN_1, VTN_2, \ldots, VTN_w\}$; $w = |VTM| \geq 1$.

Virtual Tenant Network is a set of virtual tenant slices, denoted as VTN=$\{VTS_1, VTS_2, \ldots, VTS_q\}$; $q = |VTN| \geq 1$. We allocate for each VTN a dedicated and shared resources from D.

[1] A group table consists of group entries. The ability for a flow entry to point to a group enables OpenFlow protocol to present additional methods of forwarding (e.g. select and all). It can be used for grouping common actions of different flows or handling specific forwarding like load-balancing.

$VTN_{i;i=1..w}$ can be composed of one or more **Virtual Tenant Slice (**VTS**)** where given SDN resources allocated for tenant topology are dedicated for each slice including:

- a subset of OpenFlow switches ports (P), denoted as : $P_{-}^{(i)}$ / $P_{-}^{(i)} \subset P$,
- a subset of processing tables (T), denoted as : $T_{-}^{(i)}$ / $T_{-}^{(i)} \subset T$,
- a subset of groups tables (GT), denoted as : $GT_{-}^{(i)}$ / $GT_{-}^{(i)} \subset GT$.
- dedicated OpenFlow controllers, denoted as : $C_{-}^{(i)}$ / $C_{-}^{(i)} \subset C$,
- and finally the list of the installed OpenFlow entries ($FE_{-}^{(i)} \subset FE$) in the allocated processing space $T_{-}^{(i)}$ and $GT_{-}^{(i)}$ where:

  FE=$\{fe_1, fe_2, \ldots, fe_v\}$; v = $|FE| \geq 0$. $fe_{i;i=1..v}$ can be installed on OpenFlow table t_i or group table gt_i and is dedicated for specific port p_i. By default, we deny all flow entering the tenant slice's port similar to basic firewall configuration. fe_i groups the list of all permitted flows, denoted as PF=$\{pf_1, pf_2, \ldots, pf_z\}$; z = $|PF| \geq 0$. Each $pf_{i;i=1..z}$ is detailing the permitted packet header attributes (e.g., IP source and destination, Port source and destination, VXLAN tag ...).

3.2.2 Layer 2 Isolation Abstraction Properties

Ou L2 isolation model ensure that no intersection can be found between different tenants' networks ($VTNs$) and tenant's slices ($VTSs$):

$$VTN_i \cap VTN_j = \varnothing;\ \forall\ \mathrm{i} \neq \mathrm{j}\ \text{and}\ VTS_i \cap VTS_j = \varnothing;\ \forall\ \mathrm{i} \neq \mathrm{j}$$

The following properties guarantee the separation between $VTNs$ and $VTSs$:

Property 1 (OpenFlow port connection)

Each in-port ($p_{i;i=1..k} \in P_{in}$) can be linked only to one virtual machine ($vm_{j;j=1..p}$) where: VM=$\{vm_1, vm_2, \ldots, vm_p\}$; p = $|VM| \geq 0$, is the set of virtual machines running on the multi-tenant data center.

$$(vm_j,\ p_i) - (vm_o,\ p_i) \neq (\alpha,\ 0);\ \alpha \neq 0, \forall\ \text{i} \in \text{k and j, o} \in \text{p}.$$

Each out-port ($p_i \in P_{ext}$) can be linked only to one out-port p_j from another datapath.

Property 2 (Processing Space Allocation)

Each table, $t_{i;i=1..h}$, is the unique responsible for a distinct group of *vms*' traffic processing.

Similar to OpenFlow table, each group table, $gt_{i;i=1..l}$, is the unique responsible for a distinct group of *vms* with similar traffic forwarding behaviour.

Property 3 (OpenFlow Controller Connections)

Each OpenFlow controller, $c_{i;i=1..r}$, can manage only one VTN but can control more than one VTS.

3.3 Multi-tenant Network Isolation using OpenNMS: Example

As our previous definitions and properties states, VTS is a subset of virtual tenant topology including :

- OpenFlow switches' ports linked to allocated tenant's VMs,
- OpenFlow allocated processing spaces which are responsible for the tenant slice's VMs traffic forwarding,
- a list of flow entries installed on processing spaces specifying VTS behaviors,
- and a cluster of OpenFlow controllers managing the incoming flows into the tenant slice.

The only remaining puzzle for our VTS definition is the mapping from tenant logical topology to the real one. Our high-level abstraction specifies only the logical network resources contained in VTS. The mapping will indicate how tenant allocated network elements will be translated to the corresponding resources in the underlay network.

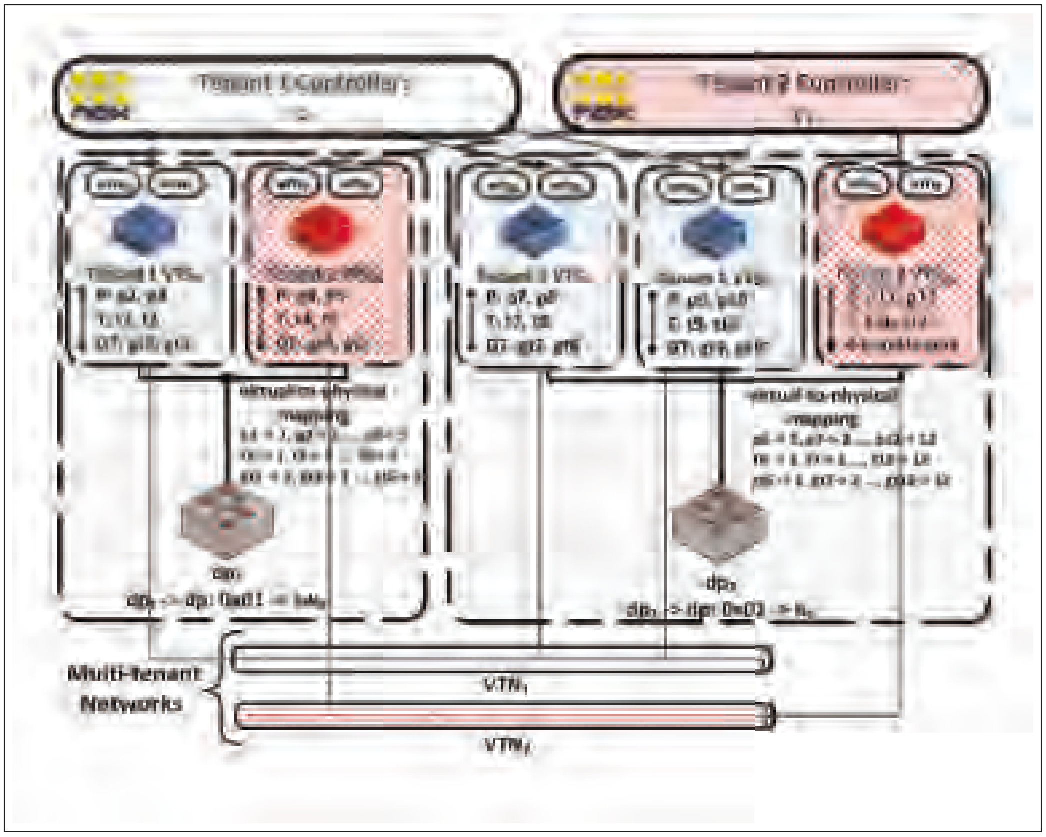

Figure 3.2 OpenNMS Multi-tenancy Support: example VTNs and related VTSs

Listing 3.1: Example: $Tenant_1$'s VTN_1 Definition

```
{"OpenNMS-VTN": { "ID": "1",
"VTSs": {
{"VTS_ID": "11", "P": [...], "T": [...], "GT": [...], "FE": [...], "C": [...]},
{"VTS_ID": "12", "P": [...], "T": [...], "GT": [...], "FE": [...], "C": [...]}
}}}
```

To illustrate the use of VTS abstraction in practical use case the following JSON structures in Listings 1, 2, 3 and 4 are used to define and deploy the $tenant_1$ virtual network (VTN_1) in Figure 3.2 which is composed of two VTSs: $VTS_{1,1}$ and $VTS_{1,2}$. The first listing depicts VTN_1's general elements. We can notice that the first VTS is distributed into two pieces. The first portion of $VTS_{1,1}$ is allocated on dp_1 and the second on dp_2. We allocate only OpenFlow resources for $VTS_{1,2}$ in the second datapath dp_2. Next, $VTS_{1,1}$ is taken as an example to encompass all cloud's tenant requirements. The tenant can have distributed VMs across hosts.

Listing 3.2: Example: $Tenant_1$'s $VTS_{1,1}$ Definition (Part 1)

```
{"VTS_ID": "11",
"P":[
{"p_ID":"p1","dp_ID":"dp1","dp_map":"0x01","p_map":"1","Dedicated":"false"},
{"p_ID":"p2","dp_ID":"dp1","dp_map":"0x01","p_map":"2","Dedicated":"true"},
{"p_ID":"p3","dp_ID":"dp1","dp_map":"0x01","p_map":"3","Dedicated":"true"},
{"p_ID":"p6","dp_ID":"dp2","dp_map":"0x02","p_map":"1","Dedicated":"false"},
{"p_ID":"p7","dp_ID":"dp2","dp_map":"0x02","p_map":"2","Dedicated":"true"},
{"p_ID":"p8","dp_ID":"dp2","dp_map":"0x02","p_map":"3","Dedicated":"true"},
],
...}
```

The first block in Listing 2 defines the $tenant_1$'s logical ports allocated for $VTS_{1,1}$, represented as P. These ports are allocated on two data paths identified as dp1 and dp2. We use p_{map} function to identify them on underlay spaces as well as their status indicated in *Dedicated* field. *Dedicated* field with value "*true*" (equal to $p_{i;i=1..k}(2) = 1$) means that OpenFlow port is not shared with other VTSs, and vice-versa. This simple condition simply states that no p can be both dedicated and shared. dp_{map} field defines the OpenFlow switch mapping from VTS down to the underlay network: dp1 maps to 0×01 (the physical switch sw_0) and dp2 maps to 0×02 (the virtual switch installed on h_0). The intersection between VTS_X's dedicated P and the others VTSs' dedicated P, for the same or different tenants, must be empty: $VTS_{1,1}\{p_i(2) = 1\}$ $\cap\, VTS_{j\in w,o\in q;\ o\neq 1}\ \{p_i(2) = 1\} == \varnothing\ \forall\ i\ in\ 1..k$.

In this example, $VTS_{1,1}$ has four dedicated ports ($p2$, $p3$, $p7$ and $p8$) and two shared ports ($p1$ and $p6$). Usually, we share all external OpenFlow switches ports linked to their hosting servers' network interface cards (NICs) between different tenants' VTSs ($p1$ and $p6$ in our case) which models the behaviour of real network program supporting multi-tenancy. As results, incoming packets on shared p_i must be processed by shared tables and group tables, and managed by neutral OpenFlow controller. For VTSs collaboration and inter/intra communications, p_i can also be shared between VTSs (See details in 3.4). These conditions are automatically enforced either by tenants or service providers controllers.

Listing 3.3: Example: $Tenant_1$'s $VTS_{1,1}$ Definition (Part 2)

```
{"VTS_ID": "11",
...
"T":[
{"t_ID":"t1","dp_ID":"dp1","dp_map":"0x01","t_map":"54","Dedicated":"false"},
{"t_ID":"t2","dp_ID":"dp1","dp_map":"0x01","t_map":"1","Dedicated":"true"},
{"t_ID":"t3","dp_ID":"dp1","dp_map":"0x01","t_map":"2","Dedicated":"true"},
{"t_ID":"t7","dp_ID":"dp2","dp_map":"0x02","t_map":"54","Dedicated":"false"},
{"t_ID":"t8","dp_ID":"dp2","dp_map":"0x02","t_map":"1","Dedicated":"true"},
{"t_ID":"t9","dp_ID":"dp2","dp_map":"0x02","t_map":"2","Dedicated":"true"}
],
"GT":[
{"gt_ID":"gt1","dp_ID":"dp1","dp_map":"0x01","t_map":"54","Dedicated":"false"},
{"gt_ID":"gt2","dp_ID":"dp1","dp_map":"0x01","t_map":"1","Dedicated":"true"},
{"gt_ID":"gt3","dp_ID":"dp1","dp_map":"0x01","t_map":"2","Dedicated":"true"},
{"gt_ID":"gt7","dp_ID":"dp2","dp_map":"0x02","gt_map":"54","Dedicated":"false"},
{"gt_ID":"gt8","dp_ID":"dp2","dp_map":"0x02","gt_map":"1","Dedicated":"true"},
{"gt_ID":"gt9","dp_ID":"dp2","dp_map":"0x02","gt_map":"2","Dedicated":"true"}
],
...}
```

Similar to P block, the subsequent elements T and GT define the tenant processing space, specified as shared or dedicated OpenFlow tables and Group tables, and mapped from the tenant slice down to the underlay network (See Table 3.1). In this example, shared tables (t_1 and t_7) and group tables (gt_1 and gt_7) are used to process all incoming multi-tenant flows from an external source.

Table 3.1 $VTS_{1,1}$ dedicated/shared OpenFlow tables and group tables

Tables and Group Tables / **DataPath identifier**	**Dedicated**	**Shared**
dp_1	T: $t2$, $t3$ GT: $gt2$, $gt3$	T: $t1$ GT: $gt1$
dp_2	T: $t8$, $t9$ GT: $gt8$, $gt9$	T: $t7$ GT: $gt7$

Listing 3.4: Example: $Tenant_1$'s $VTS_{1,1}$ Definition (Part 3)

```
{"VTS_ID": "11",
...
"FE":[
{"fe_ID":"fe1", "p_ID":"p1", "t_ID":"t1", "OF_action":"GoTo(SP_controller)", "OF_priority":"0"
    , "PF":[]},
{"fe_ID":"fe2", "p_ID":"p2", "t_ID":"t2", "VXLAN_ID":"100", "OF_action":"GoTo(c1)", "
    OF_priority":"0",
"PF":[
{"pf_ID":"pf1", "port_src":"8443"},
{"pf_ID":"pf2", "ip_src":"10.0.0.12", "eth_src":"00:11:11:11:11:02", "port_dst":"5984"}
]
},
{"fe_ID":"fe3", "p_ID":"p3", "t_ID":"t3", "VXLAN_ID":"101", "OF_action":"GoTo(c1)", "
    OF_priority":"0",
"PF":[
{"pf_ID":"pf1", "port_src":"5984"},
{"pf_ID":"pf2", "ip_src":"10.0.0.13", "eth_src":"00:11:11:11:11:03", "port_dst":"8443"}
]
}
,
{"..."}
],
"C":[{"c_ID":"c1","c_adr":"10.21.100.2","c_port":"6633"}]}
} //End VTS
```

The last block of $VTS_{1,1}$ definition contains:

- the list of all FE associated with each tenant slice's p_i, and detailing the permitted packet header's attributes. The first fe with identifier $fe1$ is installed on the shared $t1$ table and will forward any incoming packet on port $p1$ to the provider controller ($SP_{controller}$). The second flow entry $fe2$ will accept incoming traffic into tenant's virtual machine linked to VTS' $p2$ port and forward it to tenant controller $c1$. We exclude all traffic with $VXLAN$ tag different than 100.

- The list of all tenant dedicated OpenFlow controllers (C) with their attributes such IP address and connection port to OpenFlow switches.

Our L2 isolation model simply states that no network element ($p_{i;i=1..k}$, $t_{j;j=1..h}$ or $gt_{o;o=1..l}$) can be simultaneously dedicated and shared. A tenant slice could allocate entire datapath' ports and processing spaces. Furthermore, it may consist of a mix of physical and virtual ports, and multiple processing spaces (t_j and gt_o) on different and distributed OpenFlow switches. The mapping process is required to ensure VTS isolation and compatibility on underlay network. Every logical identifier must map to unique real one. This condition should be satisfied for all OpenFlow switches and their related ports, tables and groups tables. Moreover, FE and their respective PF specifies the list of permitted flows entering the VTS at P linked to tenant's VMs. Finally, VTS can be programmed independently by tenant allocated controllers (C) while ensuring that VTS functions are compatible with pushed FE, and that packets do not get interfering or sticking during processing, except possibly by processing incoming flows on shared P which are checked on shared T and GT. Even this form of interference is automatically ruled out through service providers' neutral controllers. These controllers manage the shared network resources by detecting and eliminating any possible conflicts between tenants' FE before pushing it down to the underlay network. This last feature is already covered by previous researches (e.g., Porras *et al.* (2012), Shin *et al.* (2013a)).

From these definitions, a VTS extends the network with new self-ruling virtual networks that can be programmed in a standalone mode just like an ordinary OpenFlow switch, without worrying about other OpenFlow applications running on the top of any other VTSs. OpenFlow applications managing tenant ports will be written exclusively for a specific VTS, and will not affect any other VTSs. We believe that our VTS definition model will serve a strong specification for the infrastructure providers to translate and map it to the underlay network while ensuring isolation properties. Even if tenant makes changes and updates on his VTS program, the other slices belonging to the same or different tenants will stay intact. Our model holds even if VTSs are overlapped over the same OpenFlow switches. If a flow arrives at a shared p_i between multiple VTSs, it will be checked first by service providers controllers and a copy will be forwarded to the VTS whose FE's attributes are matched with the packet header. If there is no match with any VTS, the packet will be dropped.

3.4 How Our L2 Isolation Model Meets the Goals ?

Cloud multi-tenants will have different, multiple and arbitrary virtual network services to deploy. A tenant may need to scale his VTN to an arbitrary size or modify their boundaries without getting forced into a complex and static reconfigurations. For example, new VM may be needed to join tenant's slice for arbitrary reasons such as scaling tenant applications capacity or for load-balancing purpose. For that, we facilitate such operation by simply adding new P, T and GT to the VTS allocated resources without getting forced to rebuilding steps. Also, a tenant wishing to share a VM or processing space between two VTSs might simply need to set a port, table or group table as shared network elements between them. A tenant might also want to move some VMs or entire VTS's VMs from one slice to another. Our flexible isolation supports also this need by enabling tenant to replicate the same configuration of moving VMs on the joined VTS. We simply link migrated VMs to the new OpenFlow ports added on the target VTS and then copy and adapt all their related flow entries. For each p and related FE, the following information must be updated:

a. $p_{ID_{old}} \leftarrow p_{ID_{new}}$;

b. $dp_{ID_{old}} \leftarrow dp_{ID_{new}}$;

c. $dp_{map_{old}} \leftarrow dp_{map_{new}}$;

d. $p_{map_{old}} \leftarrow p_{map_{old}}$;

e. For each fe in tenant's FE do:
   $fe_{ID_{old}} \leftarrow fe_{ID_{new}}$;
   $p_{ID_{old}} \leftarrow p_{ID_{new}}$;
   $t_{ID_{old}} \leftarrow t_{ID_{new}}$;
   $gt_{ID_{old}} \leftarrow gt_{ID_{new}}$;
   and $OF_{action_{old}} \leftarrow OF_{action_{new}}$;

Most existing SDN isolation solutions cannot meet all these needs simultaneously. OpenNMS's L2 isolation model gives each tenant a straightforward abstraction view of his allocated VTN topology and related VTSs: each VM is linked to a single and unique p_i, and one dedicated processing space (t_j and gt_o) is used to forward all flows entering and leaving the slice. Our flexible and self-manageable model (See Chapter 4 for more details) handles all tedious details related to enforcing the isolation, freeing cloud tenant from having to reason about tricky issues and all complex management operations. This model allows a tenant to design his VTSs as if it is the sole occupant of the shared infrastructure. That is, a tenant can be able to define his own slices while he can also automatically: (i) scale them to desired size by adding new instances, (ii) join migrated VMs from one to another, and (iii) apply optimizations that make efficient use of the allocated network resources by sharing them between allocated slices. The first two features are straightforward and automatically adapted by our high-level VTS abstraction. However, the last one needs to be more developed and discussed considering that several questions related to security enforcement and resources optimization can be raised. If tenant wants to share a virtual application between two VTSs (VTS_X and VTS_Y), all what he needs is to change the "dedicated" field of VM's linked p_i from "true" to "false" in the first VTS_X and automatically add this OpenFlow port to the second VTS_Y with a shared state. The questions that can be raised after these actions in order to accomplish the tenant requirement are:

- Which table (t_j) and group table (gt_o) will handle the processing of entering and leaving packets from/to this shared port (p_i)?
    a. It will be the dedicated processing space for VTS_X or VTS_Y or both by simply adding the shared application attributes (MAC/IP Addresses and TCP port) into the corresponding PF?
    b. Or, it will be redirected to a shared t_j or gt_o grouping the same FE and PF?
- Are both VTSs controlled by the same tenant controllers (C)? IF not, the last possible solution (b.) will be not valid for shared t_j, unless the tenant will add also VTS_X' controller to VTS_Y in aim to handle all the shared p_i's traffic.

Considering any possible tenant choice from the above solutions, we combine all possibilities of sharing VTS's resources that can have place, not only to predict any tenant needs but also to have a rich and optimal isolation solution. Tenant's VTSs can scale to huge sizes with larger number of virtual applications or ports and accordingly the number of allocated tables, group tables and controllers. To fully realize the benefits of network resources sharing, we consider all multiplexing choices that can be used to achieve the best resource efficiency and cost savings. Increasing the scale alone or resorting to sharing mechanisms like the first sharing solution (a.), however, cannot fully minimize the total cost, on an attractive "*pay-as-you-go*" model for cloud computing. Therefore, we propose different VTS types denoted by $St_{i \in \{0..3\}}$ in Table 3.2. Each St considers that one or more network resources types (P, T, GT, C) will be dedicated only for the deployed VTS. Thus, this constraint eliminates the possibility of sharing these dedicated resources with other VTSs.

Table 3.2 OpenNMS's Virtual Tenant Slice Types

Dedicated Resource / **VTS Type**	***P***	***T***	***GT***	***C***
St_0	✓	✓	✓	✓
St_1	✗	✓	✓	✓
St_2	✗	✓	✗	✓
St_3	✗	✗	✗	✗

The core idea of these types is to provide different security levels that ensure the desired isolation proprieties between tenant's VTSs while managing shared resources under mutual agreement enforcing unified security policies. Each St will enforce the tenant sharing constraints but it will have advantages and drawbacks at the same time. The following table 3.3 compares between VTS's types basing on the following criteria:

- **Scalability**: will be always guaranteed for all St. Our isolation model supports VTS scalability. It allows adding more network resources to VTS.

Table 3.3 VTS Types Advantages

VTS Type \ Advantages	***Scalability***	***Optimization***	***Overhead***	***CIA***
St_0	✓	local	highest	high
St_1	✓	partial	$> St_2$	medium
St_2	✓	medium	$> St_3$	low
St_3	✓	overall	negligible	very low

- **Optimization**: A restriction mechanism is enabled by preventing the share of one or more resource types. St_0 restricts sharing all allocated resources, and thus network and compute resources optimization can be done just locally in VTS. In St_1, it is permitted to share only P, so tenant will resort to replicate the same resources configurations (T, GT and C) on VTSs sharing these P. This kind of optimization is indicated as "partial" because it can minimize the total cost. The difference between St_2 and St_1 is that sharing OpenFlow tables and controllers is permitted between VTS. Therefore, this type can offer better optimization than the previous one. Finally, the last type St_3 allows the most efficient usage of tenant's allocated resources and overall optimization suitable for the cloud "*pay-as-you-go*" model. This isolation type provides the required optimization, smoothness and performance for cloud application while ensuring the security between tenant's VTSs. The tenant can adapt the trust level and share any allocated network resources between his VTSs for any reasons and any desired combination.

- **Overhead**: Obviously, any extended mechanism to flow processing will add more overhead. Thus, enforcing each kind of St will require more additional time to check and verify restrictions on VTS. Logically, St_3 do not add any restriction on VTS definition. All allocated resources may be shared. Therefore, it will generate negligible overhead. We can conclude that more overhead will be added as much we add restrictions: $St0$ overhead $> St1 > St2 > St3$, as shown in the evaluation section 5.3.2.

- **Confidentiality, Integrity and Availability (CIA)**: Tenant must be aware about the risks of each *St* on CIA before choosing. If a shared resource will be compromised, it can affect the rest of tenant network. Thus, different security levels are attributed to VTS types, from the highest (attributed to St_0) to the lowest one (attributed to St_3).

3.5 Summary and Conclusions

In this chapter, we have first introduced the benefits of SDN slicing technique and the actual opportunities with such approach, where the multi-tenant flexible, scalable isolation can be achieved by providing a solid Layer 2 model and combining it with the overlay technologies.

Second, we defined our Layer 2 isolation model providing high level abstraction for multi-tenant network by combining SDN slicing technique and overlay virtual protocols (e.g., VXLAN, NVGRE). Our model enhances the SDN slicing technique flexibility while preserves its advantages for providing the basic isolation objectives. We take benefits of the scalability of overlay protocols and overcome their limits to one overlay network by enabling to support several thousands of isolated tenants networks and slices on top of shared network infrastructure. Using these extended encapsulations mechanisms, the number of segmentation is increased and as result it offers the possibility of creating more number of VTSs. Our isolation model formalizes the simultaneous use of multiple resources (ports, tables, group tables and controllers) allocation for multi-tenant network. This allows service providers to define multiple processing spaces dedicated for individualized network services. From our definitions and proprieties, the VTS model extends the network virtualization with new self-ruling virtual networks that can be programmed in a standalone mode just like an ordinary OpenFlow switch, without worrying about other OpenFlow applications running on the top of any other VTSs. It will serve a strong specification for the infrastructure providers to translate and map it to the underlay network while ensuring isolation properties.

Further, our VTS types show that we can leverage almost the full optimization of allocated resources to meet differing tenant application objectives. Even if tenant switches between VTS type for higher sharing or restriction level, the other slices belonging to the same or different tenants will stay intact. However, some penalties are analysed with the tenant choice such as the added overhead and the impact on CIA.

The next chapters introduce our autonomic SDN-based architecture design for achieving the self-manageable multi-tenant network. It will bring this model in the real world and define hierarchical controllers and managers enabling tenant to take full control of their virtual networks and slices. It will give a straightforward interface for network providers to dynamically enforce the L2 isolation with high level abstraction while supporting the scalability and flexibility for the tenant. Flowcharts are provided for better understanding of our isolation model and the interaction between network components and layers. We will evaluate a OpenNMS prototype system including overhead, capacity in term of VTNs and VTSs, flexibility and the performance of our slice types. Finally, we will extend our objectives to demonstrate the recursive results of OpenNMS model on the OpenFlow controller centralized scalability.

CHAPTER 4

OPEN NETWORK MANAGEMENT & SECURITY ARCHITECTURE DESIGN

Open Network Management & Security (OpenNMS) architecture is designed with the following principles that will be respected in all our components: (i) adopting the SDN/OpenFlow standards without introducing any new layer, (ii) enforcing multi-tenant isolation in an automatic and dynamic way, (iii) reaching both tenant and provider networks scalability without losing the centralized controller's global view.

4.1 Open Network Management & Security Design

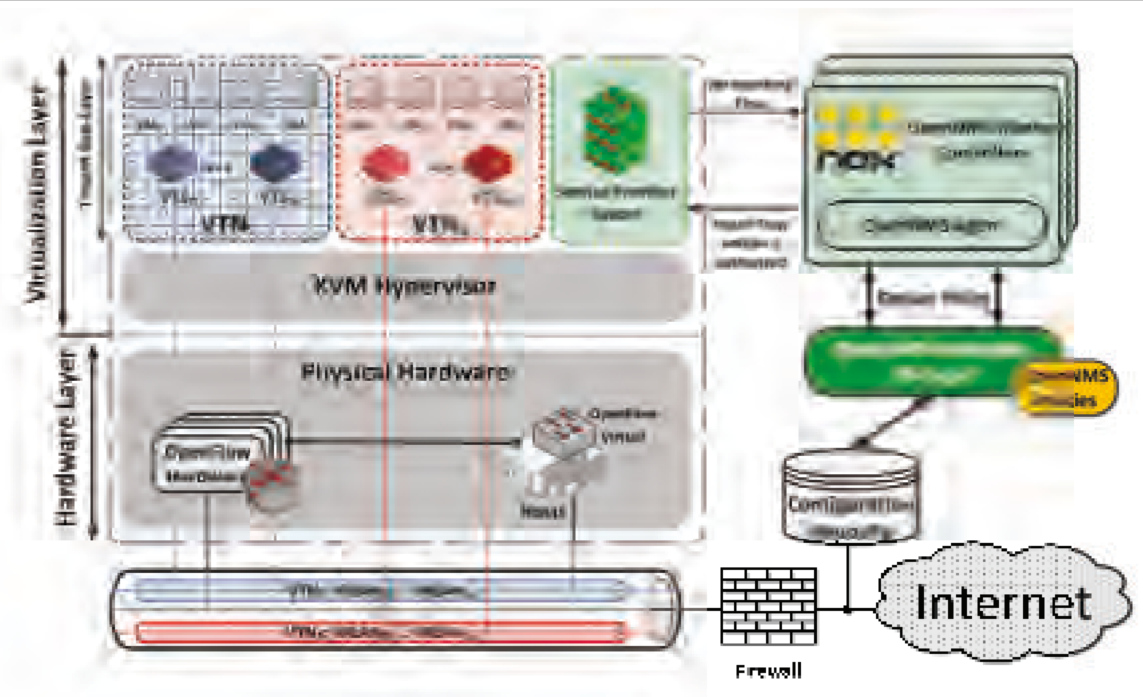

Figure 4.1 OpenNMS's High Level Components

The proposed OpenNMS architecture is depicted in Figure 4.1. The main two components of the architecture are the following:

- OpenNMS Agent (OA): an OpenFlow application extending the SDN Controller. It belongs to control plane layer and it is responsible for providing isolated tenant slices by verifying incoming packets.
- OpenNMS Autonomic Manager (OAM): enables an advanced and enriched self-manageability of the SDN network which is realized through a number of control loops into the control plane. It is a planning, analyzing, orchestrating and provisioning entity. It also represents the third party component which verifies the identity of the network nodes requesting to join a specific customer zone or sharing specific resources.

4.2 OpenNMS Detailed Planes and Components

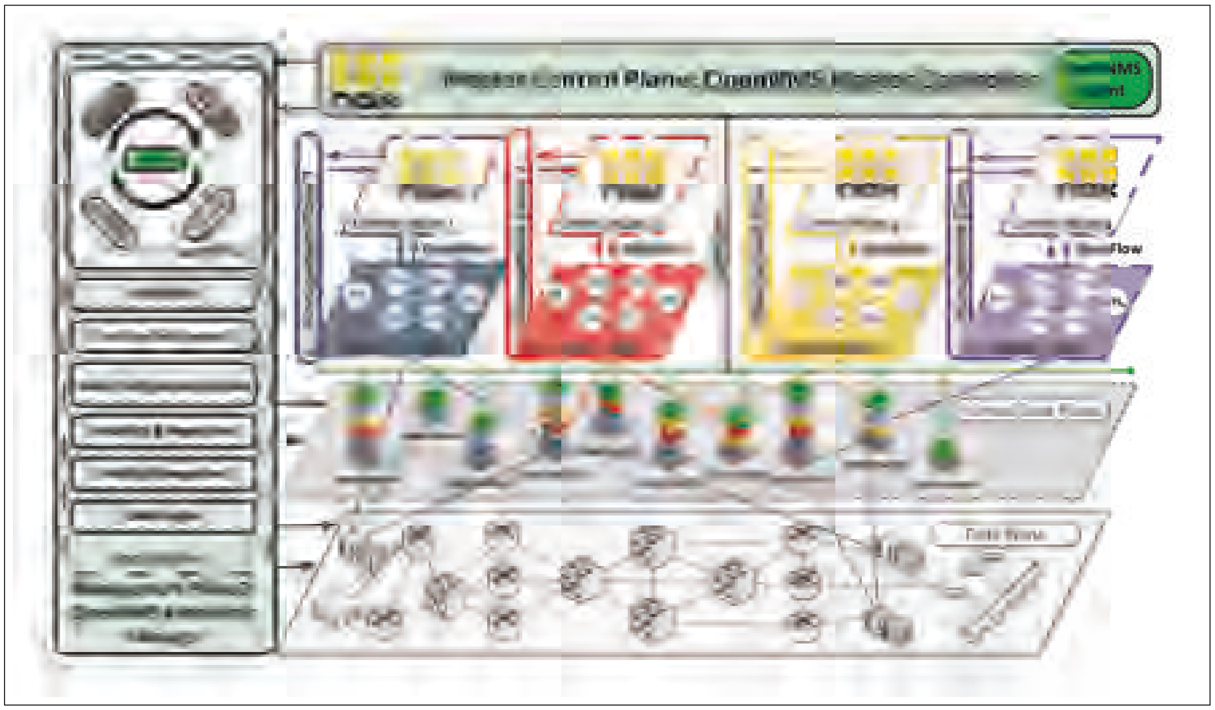

Figure 4.2 Autonomic OpenNMS Architecture Detailed Planes

The OpenNMS detailed planes are depicted in Figure 4.2. The design is composed mainly of the following four planes:

- **Data Plane :** It is the physical layer, it includes physical servers hosting OpenFlow virtual switches and tenants' VMs, OpenFlow physical switches and routers (Top of Rack (TOR), Edge, Aggregation and Core) and other network elements. The physical network infrastructure achieves the basic connectivity of the networks. It also provides some secure channel to connect SDN controllers to virtual data plane.

- **Virtual Data Plane :** It consists of all virtual applications hosting in servers and OpenFlow-enabled switches (e.g., KVM, Open vSwitches, and VMs). It represents virtual forwarding layer resources such as OpenFlow virtual switches, Virtual Tenant Network (VTNs) and related Virtual Tenant Slices (VTSs), and the provider network spaces (small switches linked to the Master Control Plane). Tenant's VTN is defined by a set of OpenFlow resources composing VTSs, and relevant control and management functions running on the northbound planes. The tenant control and management planes are created to provide transparency and enable the self-management and self-controlling of the allocated virtual layer's resources. Our design can support different network architectures and services, and even tenant's VTSs types could also be different from one VTS to another, which is customized according to the tenant's requirement.

- **Control Plane :** our isolation approach gives to network virtualization a specific description of abstraction to the control layer. Tenant's controller is responsible for specific VTS behavior determination such as path creation, data forwarding rules, traffic engineering, etc. The Master controller is the network brain. It is responsible for global network functionalities and requirements such as multi-tenancy isolation and cross-domains communication. OpenNMS Agent (OA) is an OpenFlow application extending the OpenFlow Controller. It belongs to control plane layer and it is responsible for providing the L2 isolation by verifying incoming packets on datapath.

- **Management Plane :** there are two types of management plane in our design:

  - OAM (Management $Plane_0$): OpenNMS Autonomic Manager (OAM) enables an advanced and enriched self-manageability of the SDN network which is realized through number of control loops into the control plane. It is a planning, analyzing, orchestrating and provisioning entity. It also represents the third party component which verifies the identity of the network nodes requesting to join a specific customer slice or sharing specific resources. Control loop entity is responsible for collecting and analyzing the network changes and the tenant's needs from Master Controller. The analyzing entity would analyze the context and produce the required sets of control policies. Then, OAM will plan and execute the necessary management steps. The control loops are typically policy based. Once a certain condition is satisfied on the tenant's modifications and management requirements, an action will be running. Working with the Master Controller, OAM could update its global knowledge of the network and implement several functionalities such as self-provisioning, self-configuration, self-organization and self-optimization. These new features in management plane improve the network flexibility and automatism. This design supports the tenant networking services and policies changes with guaranteed security for provider and the other tenants. OAM and Master Controller are the key elements of an elastic, flexible and transparent isolation. They give to tenants the possibility of controlling and managing their networks. This design supports the tenant networking services and policies changes with guaranteed security for service providers and the other tenants.

- Tenant Management Planes (Management $Plane_{1toS}$): These planes give to the tenants full access to control and manage their VTSs and VTNs. The desired tenant's configuration is uploaded on OpenFlow switches through the allocated controller.

A key advantage of the autonomic OpenNMS architecture is that it provides a flexible and adaptive isolation and sharing of network resources for multi-tenants network. This autonomic design enables the following network services to the tenants under its control:

- **Scalability & Negotiation:** each tenant is responsible for its own set of virtual resources and managing services that it governs. Scalability enables a set of slices in data center networks to be combined into a larger network, where allocated resources of each constituent slice contribute to the overall resources of the larger network. For the priority definition to access the shared resources between these domains, our design offers the possibility of negotiation and priorities setting.

- **Cloud Federation:** Connectivity between VTSs to provide the vision of "one cloud" still a challenge. There are situations where tenants need to be able to link between their allocated resources from different cloud providers. Our approach reduces the multi-layer architecture complexities such as L2 VTS definition and L3 forwarding networks and policies. It provides a transparent workload orchestration between the data center networks on behalf of the tenant.

4.2.1 OpenNMS Agent (OA) Design

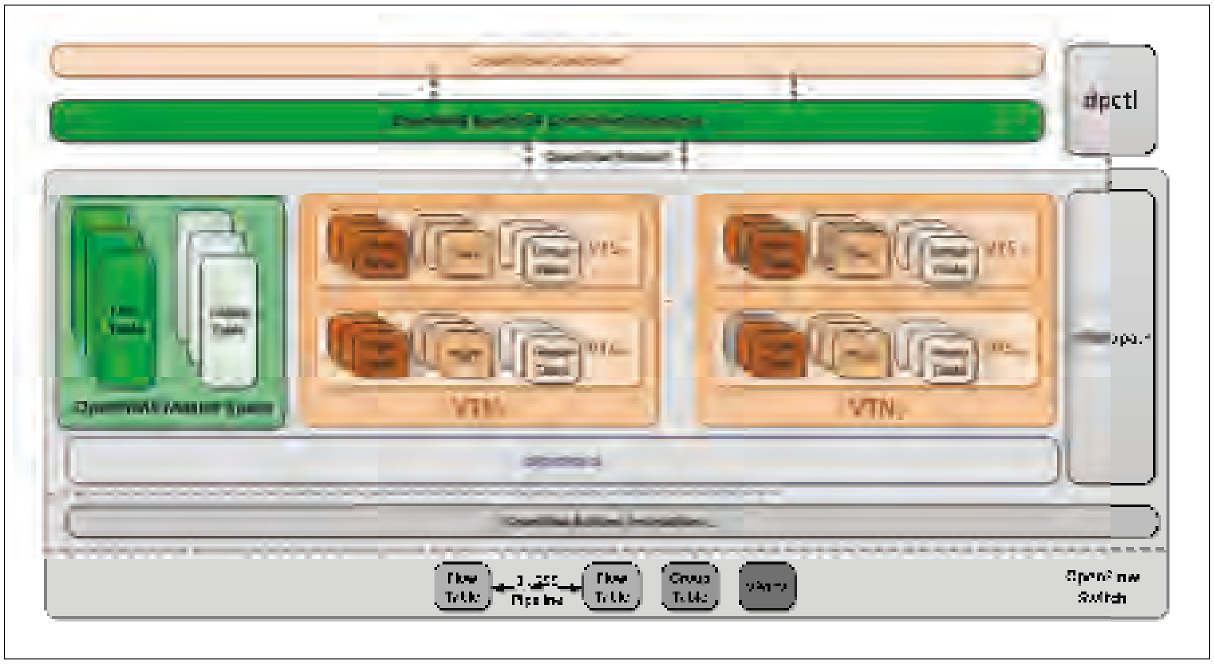

Figure 4.3 OpenNMS L2 Isolation Concept

A full virtualization mechanism is needed in order to support multi-tenancy in data center network. We designed OpenNMS Agent (OA) as OpenFlow 1.3 application for OpenFlow switch (See Figure 4.3) based on extensible packet matching and pipeline processing. It belongs to control plane layer and it is responsible for providing isolated network slices by verifying incoming packets. According to the specifications of this OpenFlow version, OpenFlow pipeline is divided into multiple sub-pipelines (T and GT). The pipeline of OpenFlow 1.3 software switch contains up to 256 flow tables, each contains up to 1024 flow entries. The pipeline processing always starts at the first flow table: the incoming packet is first matched against flow entries of table 0 (t_0). For that, we designed OpenNMS Agent to occupy the flow table 0 (OpenNMS Master Table) on each OpenFlow switch in data center network. We took this design decision based on the VTS definition which is translated into flow entries in Master Table. The rest of flows tables will be allocated or shared between OpenNMS slices. All VTSs' ports are mapped to separated OpenFlow tables. Also, the processing table can be only dedicated or shared depending on tenant's VTS type choice. Thus, the Master Table is used as a de-multiplexer which dispatches flows to different and distributed VTSs and the extensible packet matching and the pipeline features are used by OA to provide our flexible L2 isolation model.

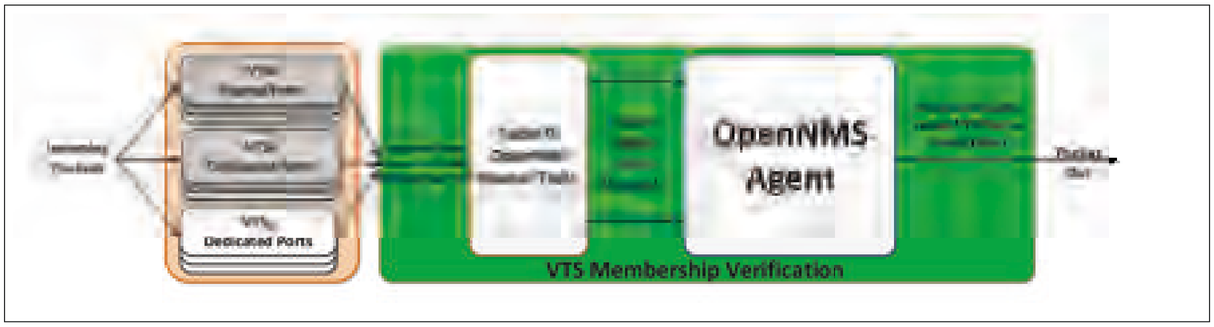

Figure 4.4 L2 Isolation using OpenNMS Packets Verification Mechanism

To control the OpenFlow port access (See Figure 4.4), the OA validates the required virtual machine identification in OpenFlow matchfields. After each incoming packet in OpenNMS Master Table, the OA adds a new flow entry to the VTS's table. If it finds that the traffic belongs to a specific VTS, the OA can go to the next step to check out-port and in-port in the new flow entry's matchfields and actions. In the VTS table, OA will deny any entry using not allocated port. Therefore, the OA can fully or partially isolate OpenFlow ports and control the access. In each OpenFlow port, there is a waiting table for incoming packet. An easy mapping of the table's ID to the corresponding VTS's ID and allocated port's ID is used to identify the incoming packet and forward it to the VTS's table. This process is controlled by the OA. The OA can change ports configuration on-the-fly without any modification on the flow entries pushed to the VTS. After validating the OpenNMS VTS identification and authorizing interfaces in OpenFlow matchfields, the OA verifies and validates the parameters of the "GoTo" instruction in OpenNMS Master Table of the corresponding OpenFlow switch and finally forwards incoming flow to the corresponding VTS's table. It checks if VTS's allocated tables are all mapped to a separated datapath tables. Thus, these tables can be totally isolated or shared depending on tenant's choice, and flow entries are not modified when pushed.

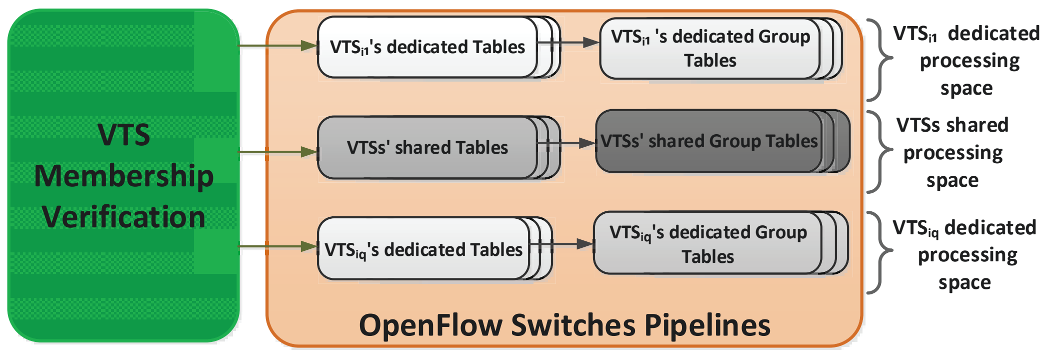
VTS Membership Verification
VTSi1's dedicated Tables
VTSi1 's dedicated Group Tables
VTSs' shared Tables
VTSs' shared Group Tables
VTSiq's dedicated Tables
VTSiq's dedicated Group Tables
OpenFlow Switches Pipelines
VTSi1 dedicated processing space
VTSs shared processing space
VTSiq dedicated processing space

Figure 4.5 VTS Shared and Dedicated OVSs' Tables/Group Tables

For each VTS (See Figure 4.5), dedicated ports are used for intra-VTS forwarding and shared one for inter-VTSs communication. OA redirects incoming packets to VTS tables. Figure 4.6 represents a detailed example for the packets processing using OA for two different tenants (1 and 2). When there is an incoming packet in Master Table (Table 0), the OA decides to redirect the packet to a specific OpenFlow table or group table depending on its multi-tenant database and VTS identification. To control the interfaces access, the OA validates the required VM identification in OpenFlow match-fields. In this example, $tenant_1$'s incoming packet originated from a local VM hosted in the same server is forwarded to his allocated Table 1 and then to

Group Table 2 with out-port 3. For the same tenant with different remote destination, the flow is forwarded to Groupe Table 3 with out-port 0 different than the previous one. A tunnel tag is added for this type of flow.

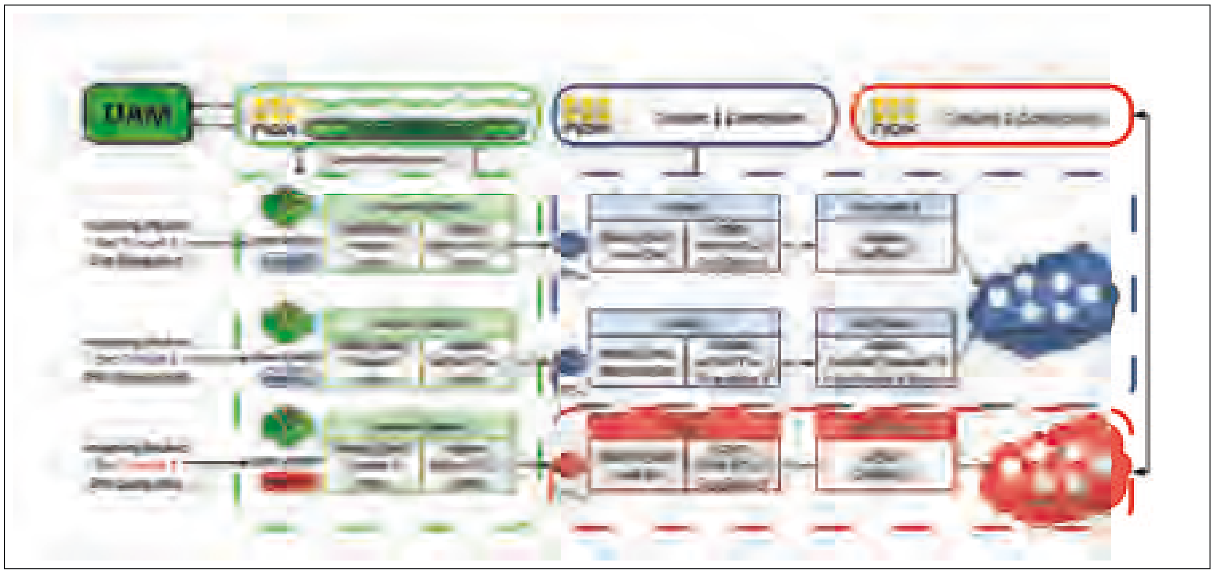

Figure 4.6 Example: Packets Processing using OA

4.2.2 OpenNMS Autonomic Manager (OAM)

Both centralized and distributed SDN-based architectures are facing increased difficulty to meet the requirement of current virtualization users and the explosion of the cloud applications. These trends are driving the research community to think more about new network architecture as well as management technologies. Until now, the key elements in the concept of the network virtualization's switches are SDN and OpenFlow protocol. This concept is mainly based on the separation of data plane and control plane. However, we need to improve the management plane that can block and slow the progression of SDN ecosystem and limit its benefits. It is necessary to use new technologies to improve protocols and network management.

In order to evolve management plane, we combine Autonomicity with SDN and design our OpenNMS Autonomic Manager (OAM) which is the component responsible for managing all

OpenNMS slices and verifying the identity of the network nodes requesting to join a specific OpenNMS slice. It requires membership management which is defined by high level user policies. Several parameters can be applied to decide whether an entity is allowed to join an OpenNMS slice, for example: VM identity provided by KVM Hypervisor (UUID) (Hammel (2011)) and encapsulation tag (e.g., VXLAN, NVGRE, VLAN, GRE ...).

In general, a network node needs to show its identities to prove its membership before it is allowed to join an OpenNMS slice. We choose a Layer 2 isolation based on VM's MAC-address and VM's allocated OpenFlow switch port for simplifying the isolation model and supporting heterogeneous encapsulations mechanisms. OAM is responsible for the global topology definition, OpenFlow resources (tables, group Tables and ports) management and associated services. This component decreases the complexity of the rapid growth of the network control and management, automate and minimize manual configuration. It provides to the network self-awareness, self-configuration, self-organization and self-optimization. OAM objective is to automate the instantiation of a virtual infrastructure while automatically deploying the corresponding security mechanisms to enforce the network isolation between different tenant networks. This deployment is driven by tenant' global isolation policy, and thus covers all resources. The instant communication between the OAM and OA guarantees a global vision of all the data center networks. This function can fullfill the demands from different VTNs and VTSs and realize an optimal context of configuration and provisioning. This centralized controller with the OA as extension solves the SDN scalability bottleneck and the cross-domains (VTNs or VTSs) communications by managing inter-domain sharing resources and access priority in case of load-balancing for example.

Figure 4.7 illustrates an architecture example with multi-tenancy support in either the same (e.g. Host A) or different (e.g. Hosts A, B, D and C) physical platform with different OpenNMS slices memberships (two different VTN). Our slicing model is based on Layer 2 isolation, VTN's VMs can communicate in the same or different platforms with need for tunneling encapsulations (e.g., VLAN, VXLAN, NVGRE and GRE) to support remote communication. For supporting heterogeneous protection and segmentation, our OAM provides to OA the required

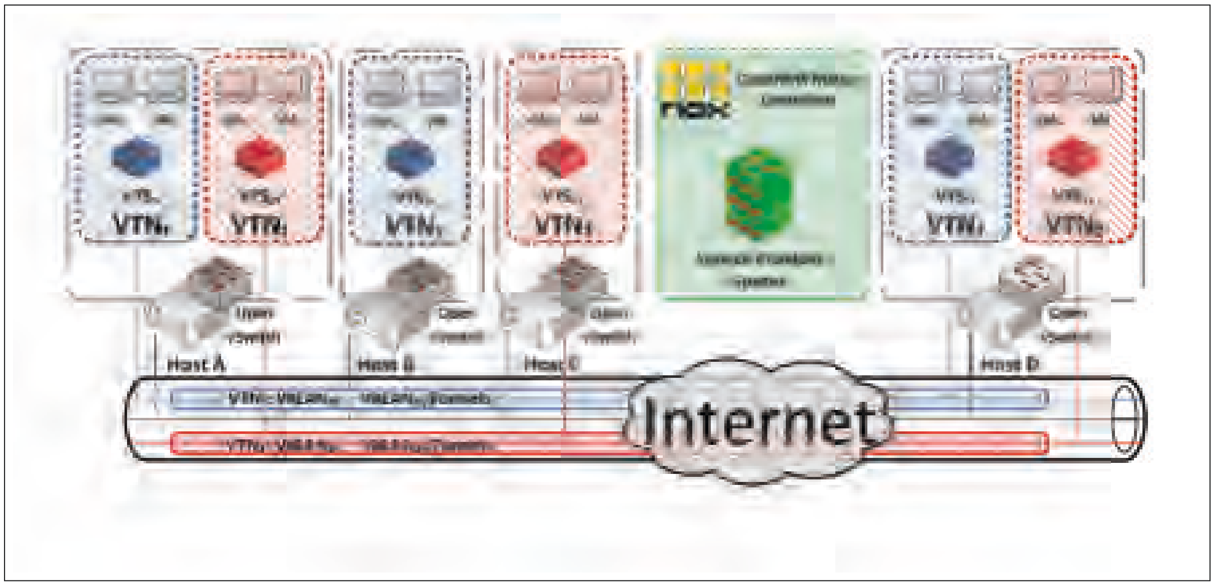

Figure 4.7 OpenNMS Multi-tenancy support

configuration with the ability of translation features for cross-domains communications. For example in host D, two tenants' VMs are running on the same physical host, and each VM is connected to a same OpenFlow enabled virtual switch. In case of intra-DCN communication, data packets coming from VMs are identified in OpenFlow switch using OpenNMS Master Controller from their L2 matching fields. For inter-DCN communication, before packet processing in the first switch, the specific tenant controller adds some ID based like Virtual Network Identifier (VNI) on the tunneling protocol. The tagged frame is transmitted to its remote destination based on flow entries which are decided by the Master Controller at each hop.

4.3 OpenNMS Security Objectives and Policy Enforcement

We are following the more promising approach in SDN slicing technique: FlowVisor (Sherwood *et al.*, 2009) which provides multi-tenant isolation using a proxy sitting between OpenFlow switches and tenants' controllers. This proxy enables only the filtering of incoming events to controllers and masking messages to switches and ensures the tenants' rules verification before getting installed on network switches. However, no sharing and inter/intra communica-

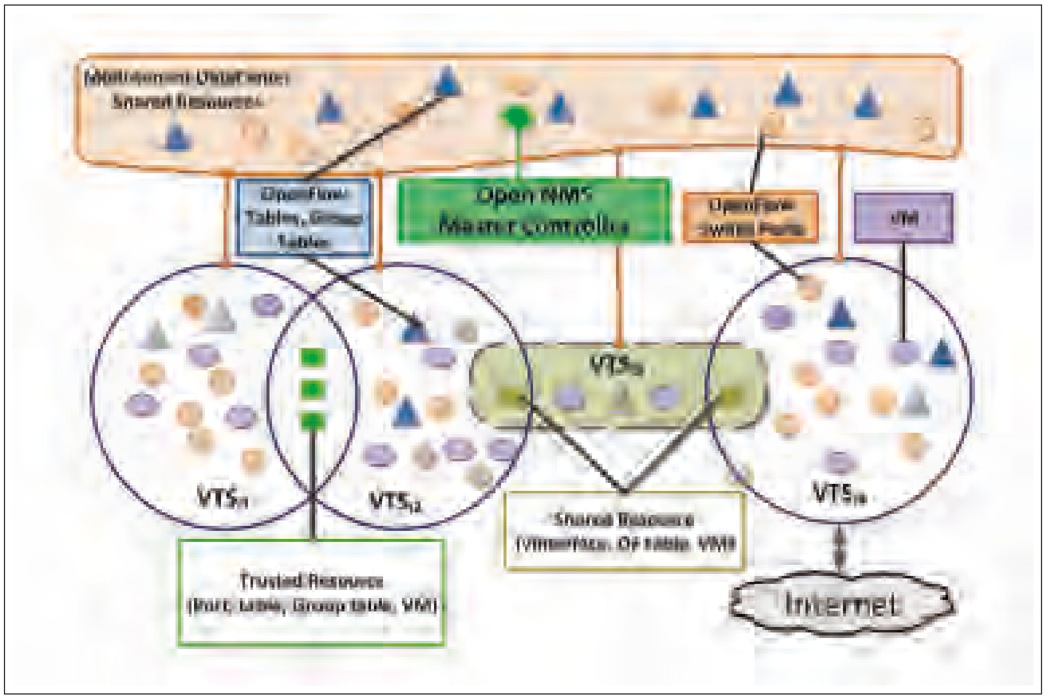
Multitenant DataCenter Shared Resources
OpenFlow Tables, Group Tables
Open NMS Master controller
OpenFlow Switch Ports
VM
VTS
VTS
VTS
VTS
Trusted Resource (Port, table, Group table, VM)
Shared Resource (VInterface, OF table, VM)
Internet

Figure 4.8 OpenNMS Usage Control for Shared/Trusted resources

tions are allowed between tenant's allocated resources. The tenant do not have a full access to the underlay resources to apply the required configurations for his applications.

With OpenNMS, we are providing a high level abstraction for multi-tenancy. OpenNMS's policy rules are applied to all shared infrastructure resources (VM, OpenFlow Switch Ports, Tables and Group Tables) (See Figure 4.8). In the first step of network deployment, we define for each slice a set of network nodes and associated resources to enforce the slicing type St_0. If a flow between two slices is allowed, resources can be shared between them. For example, if a network customer wants to share a VM hosting a database with other slices, he can allow network traffic of other slices. Each slice can define rules regarding incoming and outgoing flows for restricting external communications (e.g. Flow denied). The OpenNMS policy-enforcement guarantees that only resources trusted by a set of model slices can be shared. The Datacenter Shared Resources in Figure 4.8 represents all resources that can be shared among slices, each OpenFlow switch in a data center networks contains an OpenNMS Master Table on each data path, VMs trusted by all network nodes (e.g. OpenNMS Master Controller's VM) and trusted mutually ports.

Depending on the trust between the slices, there are two methods of enforcing rules. The first method requires trusted resources in order to share resources that are accessed from two or more slices while allowing controlling flows. This mechanism is used between slice 1 (VTS_{i1}) and 2 (VTS_{i2}) in Figure 4.8. The second method is used to share two resources between slice 2 and 4 (VTS_{i4}) connected by an intermediate slice 3 (VTS_{i3}). In this method, each slice enforces its flow control rules by means of its own shared resources. This method is used when the trust level between slice 2 and 4 is low, and the two slices cannot agree on a shared resource that is mutually trusted. The solution of this situation is to share resources through a "neutral" slice 3 with its own set of membership and identification requirements.

In an OpenFlow table, there are multiple priorities (rank) ranging from 0 to max. In each VTS's table, the permanent flow entries with the lowest priority (0) are used to send no matching packets to the OpenFlow controller which decides the actions to be taken with the flow. We give all permitted flows ($Flow_{pf}$) and denied flow ($Flow_{df}$) a higher priority than the controller flows ($Flow_{c}$). The denied flows have higher rank than the permitted flows in the matching table: rank($Flow_{c}$) = rank($Flow_{pf}$) + 1 and rank($Flow_{pf}$) = rank($Flow_{df}$) + 1.

4.4 Virtual Tenant Networks Scalability and Extended Tenant Capabilities

4.4.1 Inter Virtual Tenant Networks Sharing Methods

Depending on the desired trust between VTSs as explained in previous section, we propose two methods of enforcing rules:

- **Trusted :** requires shared resources between two or more slices. This mechanism is used in host 2 between VTN_2's VTS_{22} and VTN_3's VTS_{32} in Figure 4.9 when the desired trust level between tenant's slices is high. Tenant VMs in VTN_2 can only communicate with the shared resources circled in green (e.g., VM_{n-1}).

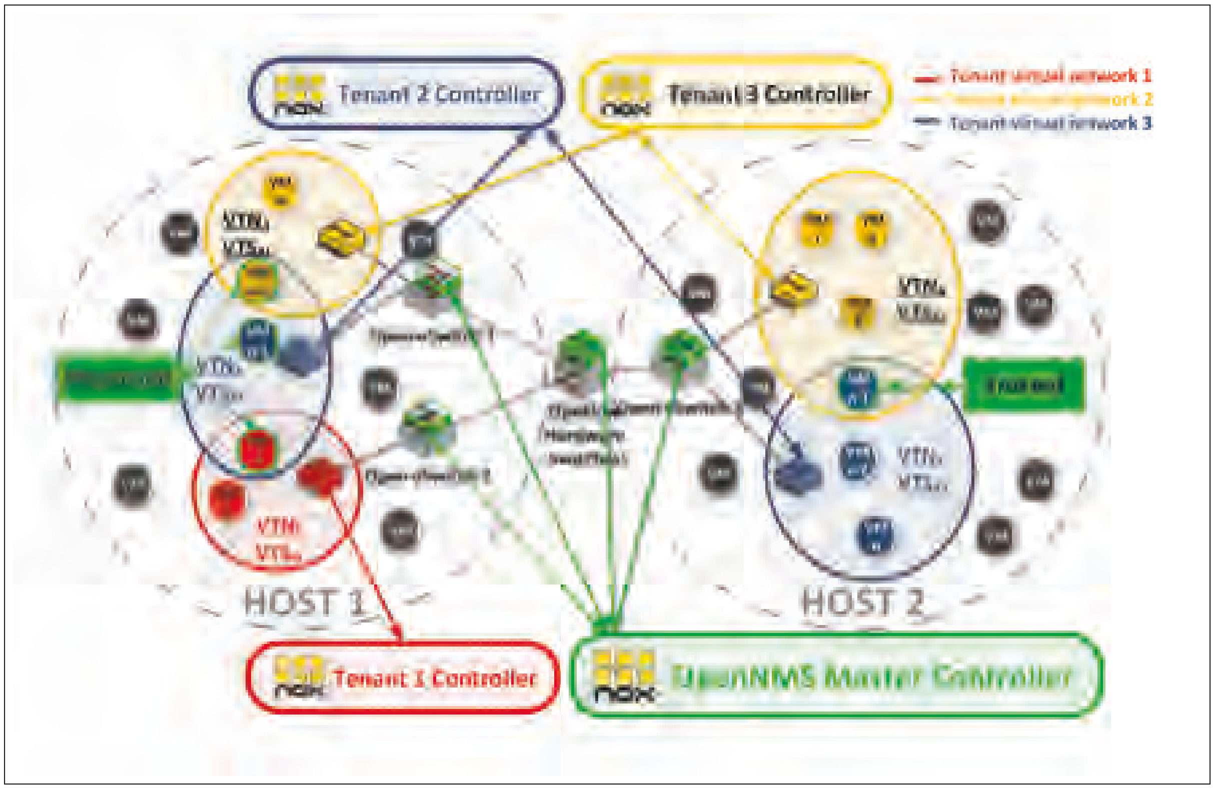

Figure 4.9 OpenNMS Trusted/Restricted Resources

- **Restricted :** is used when the trust level is low and requires sharing resources with restriction. The tenant shares slices' resources using an intermediate slice "neutral" to connect others slices. In this method, each slice enforces its flow control rules by means of its own shared resources. This method is represented in host 1 using the 3 slices from different VTNs.

4.4.2 Extended Tenant Capabilities: Split, Merge and Migrate

The Split, Merge and Migrate (SMM) capabilities enable transparent and balanced elasticity for virtual tenants networks and their related slices and applications. Using SMM, virtual tenant slices can continue to be deployed and configured individually and oblivious of the future connection and re-scaling requirements that may be needed. SMM capabilities support tenant topology and boundaries transformation on-demand. Deployed VTSs can be composed

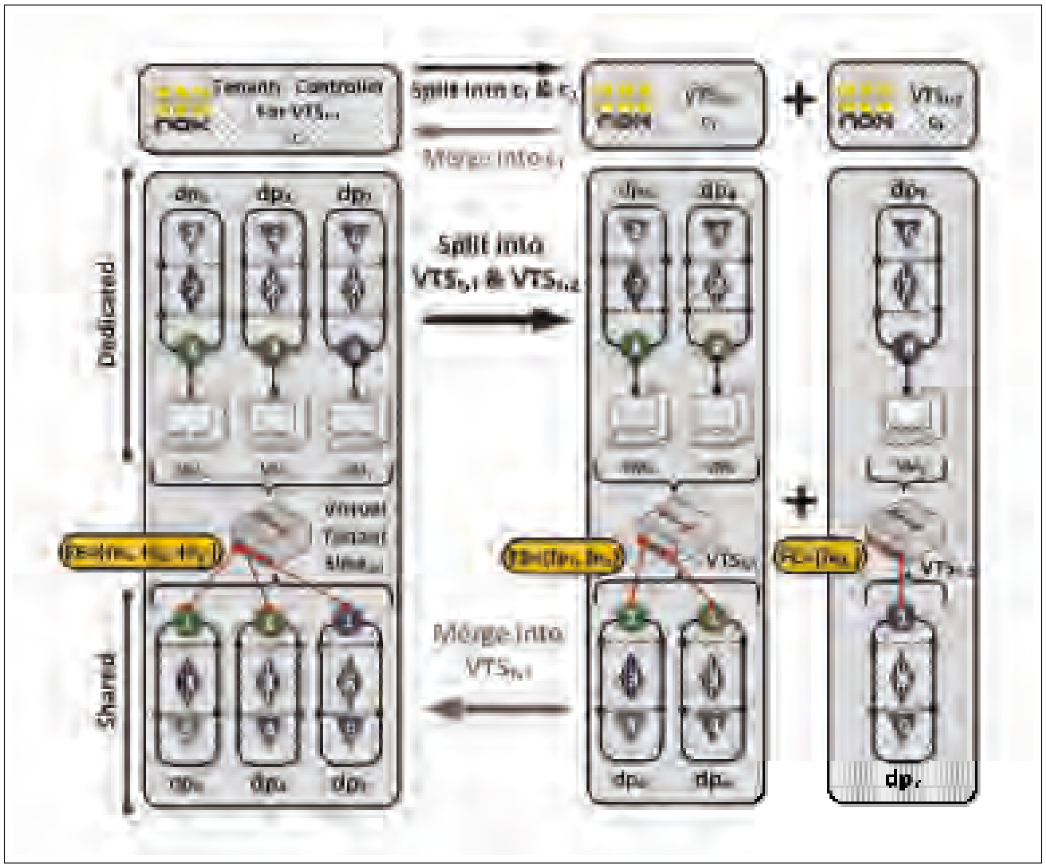

Figure 4.10 Split/Merge Vitual Tenant Slices

and decomposed as required by the tenant. The migration capability will permit to the provider to move a portion or all tenant slices from data center to another and to the tenant to displace his VTSs from one VTN to another. To realize the previous Trusted and Restricted scalability methods, SMM will allow cloud tenant to flexibly compose and dynamically apply the desired trust levels between allocated slices.

Supporting these capabilities with traditional isolation mechanisms (e.g., firewalls, middle-boxes) was very difficult. Composing, decomposing and moving flow policies from one box to another are very challenging for three as-yet-unresolved reasons. First, cloud tenant may have to re-provision and re-scale (up or down) his virtual network dynamically to account for application elasticity and potentially to mesh with the “pay-as-you-go” model of cloud computing. Second, tenants may experience poor performance and insufficient flexibility due to the lack of transparency and control on his allocated resources, which may negate some of the benefits of deploying VTN and VTSs. Finally, these operations were very complex and even if it can be realisable, an efficient checking and compiling engines must be developed. In addition, network policies typically require packets to go through a chain of services. If we change or move a node from the chain, an interruption of network services could occur.

Using our tenant dedicated space or slice, we simplified the complexity of moving, composing and decomposing flow rules. With SDN, the manual plan of flow path is eliminated. The centralized controller can manage the services chaining and configure routes to enforce such changes. By taking a network-wide view, OpenNMS Master controller eliminates errors from these tedious process and has clear visibility to set up forwarding rules that account for such transformations. All these capabilities will allow to load balance between VTSs when and where it is needed. We introduce new dimensions for SDN that fall outside the purview of the current multi-tenant isolation approaches functions that SDN tackles today.

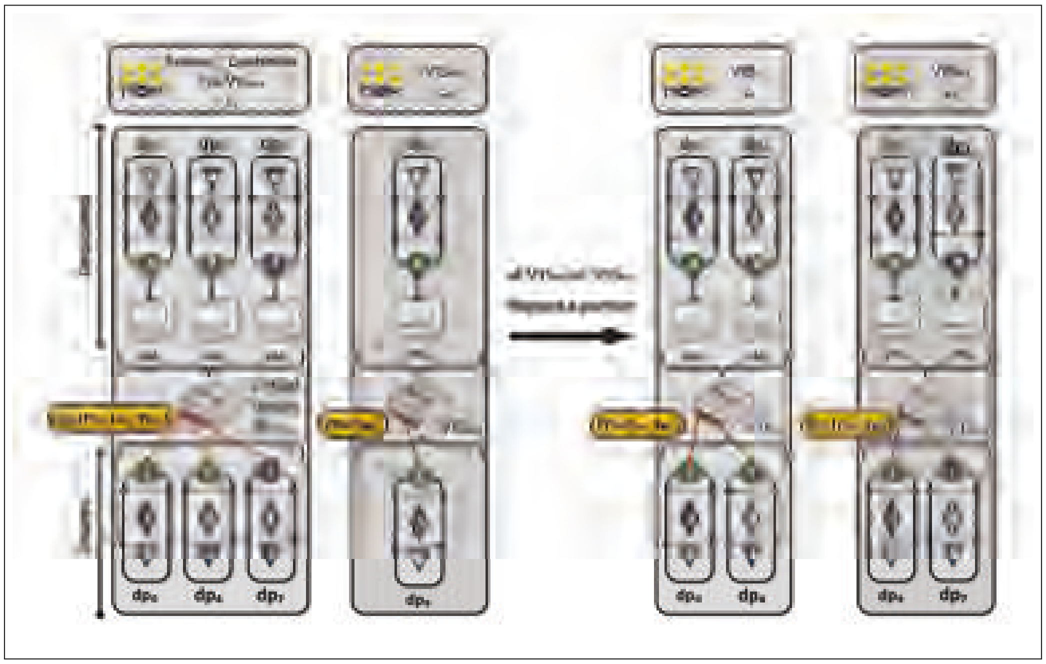

Figure 4.11 Migrate a portion of VTS to another

Figures 4.10, 4.11 and 4.12 depict how tenant slices can be split, merged and migrated when re-scaling and load balancing tenant network components.

Split/Merge Capabilities: A VTS that has appropriately defined and deployed using our high level abstraction can be dynamically split and merged. Tenant slice may be transparently split between many VTSs or merged back into one, while ensuring the tenant flows are forwarded to the correct VM. As illustrated in Figure 4.10, on split, the same configuration and definition related to dp_7 of $VTS_{1,1}$ internal and external ports, shared and dedicated tables and groups is replicated to the $VTS_{1,2}$. Each VTS is managed by independent controllers (c_1 and c_2). This allows each slice to work in parallel with its own allocated and shared resources. Coherent installed flow entries are also moved with the replicated resources and remains consistent. In our case, fe_3 is installed on the allocated new slice space. On merging, one of VTSs is selected to be destroyed. All resources belonging the deleted VTS are joining the remain slice. All flow entries are merged into one VTS processing space. Based on this, we ensure that we have sufficient degree of placement awareness that the target VTS have sufficient available OpenFlow resources that will not violate the VTS capacity constraints. In other words, there is sufficient number ports, tables and group tables to replicate the same configuration of the deleted VTS.

Migration Capability: We support the migration capability by decomposing the migration challenge into two main use cases: (i) a portion of the VTS migrating to another slice (e.g., VM live or offline migration) where both VTSs definition and policies need transformation and adaption, and (ii) the migration of all the VTS to new location that only deals with datapaths identifier translation. All the slice definition and flow entries will remain intact (See Figures 4.11 and 4.12).

We have implemented a SMM capabilities using the SMM engine on the OAM (See Figure 4.2). This engine implements the most fundamental primitive for enabling these capabilities for tenant networks and slices. The SMM-aware engine is ensuring the appropriate transformation and translation needed for VTN and VTSs re-provisioning and re-scaling. To do this, Master

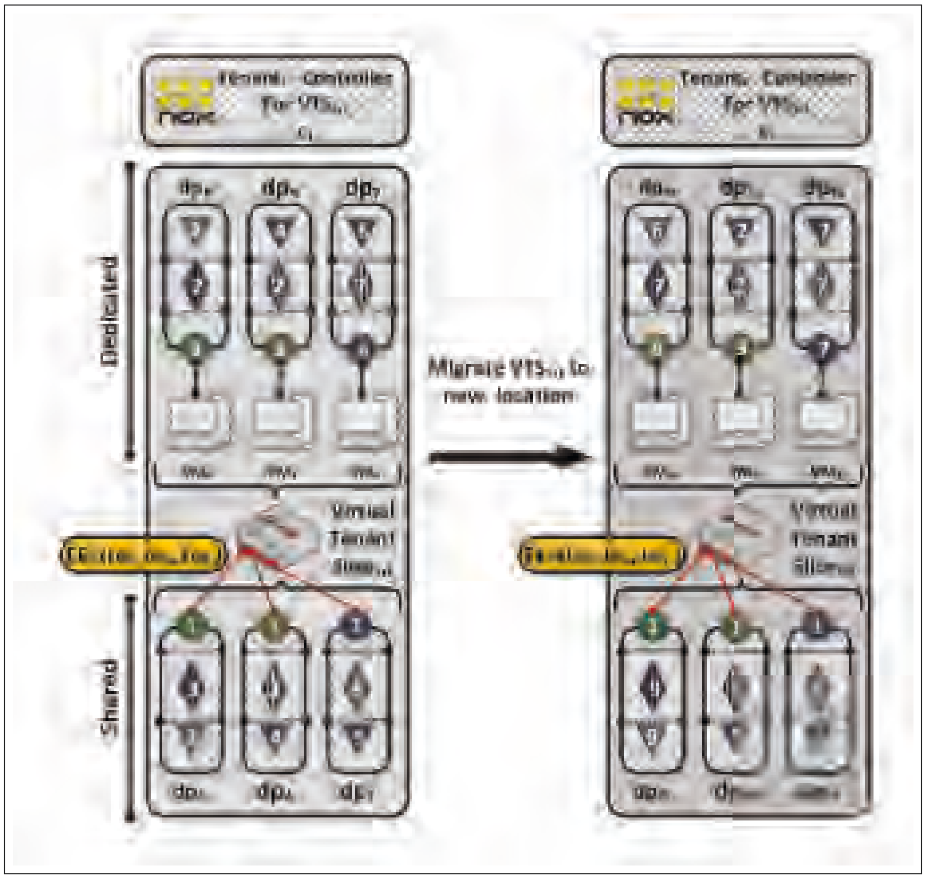

Figure 4.12 Migrate VTS to new Location

controllers is customized to communicate with OAM for enforcing and verifying all actions needed for the tenant elasticity requirements. Packets belonging to new VTSs are forwarded to the corresponding OpenFlow controller by default.

4.5 SDN Controller Scalability: Enabling To Provide A Global Large Scale Network View

As introduced before in section 1.3.3, SDN controller scalability bottleneck affects directly the scalability of the data center networks.

Some approaches have attempted to reduce the burden on the centralized controller by using distributed controllers or by including some changes on SDN paradigm and delegating a part of control functionalities to network switches. For example, Devoflow (Curtis *et al.* (2011))

focuses on improving the performance of OpenFlow controller tasks such as installing flow entries and network monitoring. It reduces the load on OpenFlow controller by refactoring the OpenFlow API and extending the network switch with control engine. However, DevoFlow reduces the decoupling between control and data planes and loses the uses centralized visibility. With a similar approach, DIFANE treats the controller scalability by keeping all traffic in data plane. It considers the switch rule memory like a large distributed cache and releases the controller from handling the traffic processing and checking. Other systems Onix (Koponen *et al.* (2010)) and HyperFlow (Tootoonchian and Ganjali (2010)) provide multiple centralized controllers but physically distributed in order to load balance the incoming events load across between them. They partition network state across these multiple network control systems and alleviate scalability and fault-tolerance concerns. They take advantages from the OpenFlow 1.3 version that allows multiple controllers connection with OpenFlow enabled switches in aim to resolve the controller issue. Thus allowing distributed multiple controllers in managing data center networks is an appropriate solution. However, current distributed controller architectures fail to adequately address the scalability concern and raise two important concerns, (i) increased overhead and inability to adapt to shifts in traffic load, (ii) there is no mechanism supported for tenant scalability and no approaches for inter-domains communications and resources sharing.

Towards a similar goal but with different method, NOX-MT (Tootoonchian *et al.* (2012a)), Beacon (Erickson *et al.* (2013)), and Maestro (Ng (2012)) scale network controllers using multi-threaded controller. In particular, Maestro is based on a 8 cores server machine. It distributes the controller workload among available cores, so that will balance the load between the 8 cores. Actually 7 cores are used for worker threads and one core is used for management functionalities. Although, many efforts have been spent to improve the controller's capacity and scalability), they are not expected to satisfy the requirements of the fast and unplanned growth of multi-tenant network.

Different to previous approaches, Pratyaastha (Krishnamurthy *et al.* (2014)) takes bottom-up approach instead of assigning a controller to switch. For Pratyaastha, the switch will seek the optimal controller to connect. It is an Non-deterministic Polynomial-time hard (NP-hard) problem for finding the optimal controller to assign to the switch. The assignment constraints is based on CPU load, available memory and processing load (number of flow per second is currently under processing). In addition, Pratyaastha enables multiple SDN applications deployment using Master centralized controller or distributed controllers. SDN applications requiring coordination are assigned and composed in sequence or parallel way in the master controller. Applications with no coordination are deployed in distributed controllers where incoming events will be sent in parallel to all controllers (Master and distributed controllers) . This approach is preserving the centralized view advantage of SDN and decrease the load on applications' controller. But, the scalability problem is remaining with the master controller which is facing increased amount of mirrored incoming packets from all data center switches. Using cluster of synchronized controllers (Master/Slaves) to offload the network charges would be sufficient to resist and recover from control failures but would leave parts of the network brainless and loose the key advantage of SDN: centralized control and global network view.

We are following the more promising approach consisting on offloading the centralised controller. This is enabled in our approach OpenNMS using hierarchical controllers (See Figure 4.13). In fact, benchmarks on OpenFlow switch shows that it regroups 255 tables with 1024 flow entries capacity per table. We believe that we can benefit from the OpenFlow switch capacity and the current OpenFlow version "1.3" that allows multiples controllers connection with OpenFow switches in aim to solve the controller issue by evolving both SDN planes. With this advantage, our OpenNMS slicing model enables the network scalability by introducing the centralized Master Controller which provides a global large scale network view and solves SDN bottleneck.

As depicted in Figure 4.14, We succeed to overcome the scalability bottleneck and provide in the manner an efficient offloading of control functionalities without losing the SDN centralized advantage. By delegating tenant's slices frequent and local packets to tenant controller, we limit the overhead on centralized controller that processes only global and rare events to maintain network-wide view. These events are the global network events including flows processing

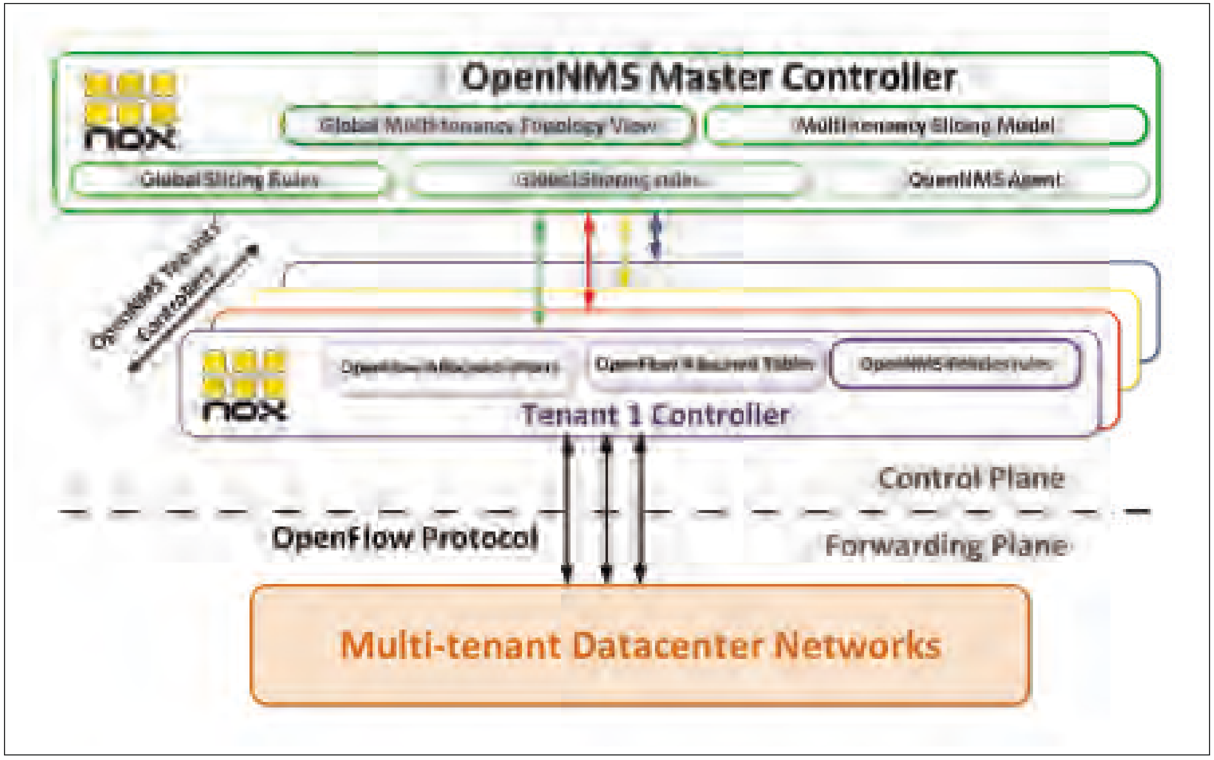
OpenNMS Master Controller
nox
Global Multitenancy Topology View
Multitenancy Slicing Model
Global Slicing Rules
OpenNMS Agent
OpenNMS Tenants Controllers
nox
Tenant 1 Controller
Control Plane
OpenFlow Protocol
Forwarding Plane
Multi-tenant Datacenter Networks

Figure 4.13 Hierarchical Control Plane

to the target VTN, CRUD VTN configuration (create, replace, update and delete OpenNMS slices), inter VTNs and VTSs communication and unrecognized flows (remote communication or unauthorized flows). We use a hierarchical design of the control plane (Master Controller – Tenants Controller) to solve the bottleneck.

From the tenant viewpoint, the inflexibility of previous approaches is limiting the scalability of its virtual networks (e.g., split, merge, update, migrate were not possible). The scalability is supported for tenants with our approach. The results are recursive, solving overlay approach by adding the missing control plane, and overcoming both virtual networks and SDN control plane scalability bottleneck by coupling SDN slices with overlay protocols.

Our concept reduces the centralized controller workload, increases its capacity to scale as the network grows and provides a satisfactory L2 isolation solution that removes the extra complexity to already difficult task of writing network configurations and placing firewalls boxes into topology. It automates and facilitates the slices definition and deployment by replacing

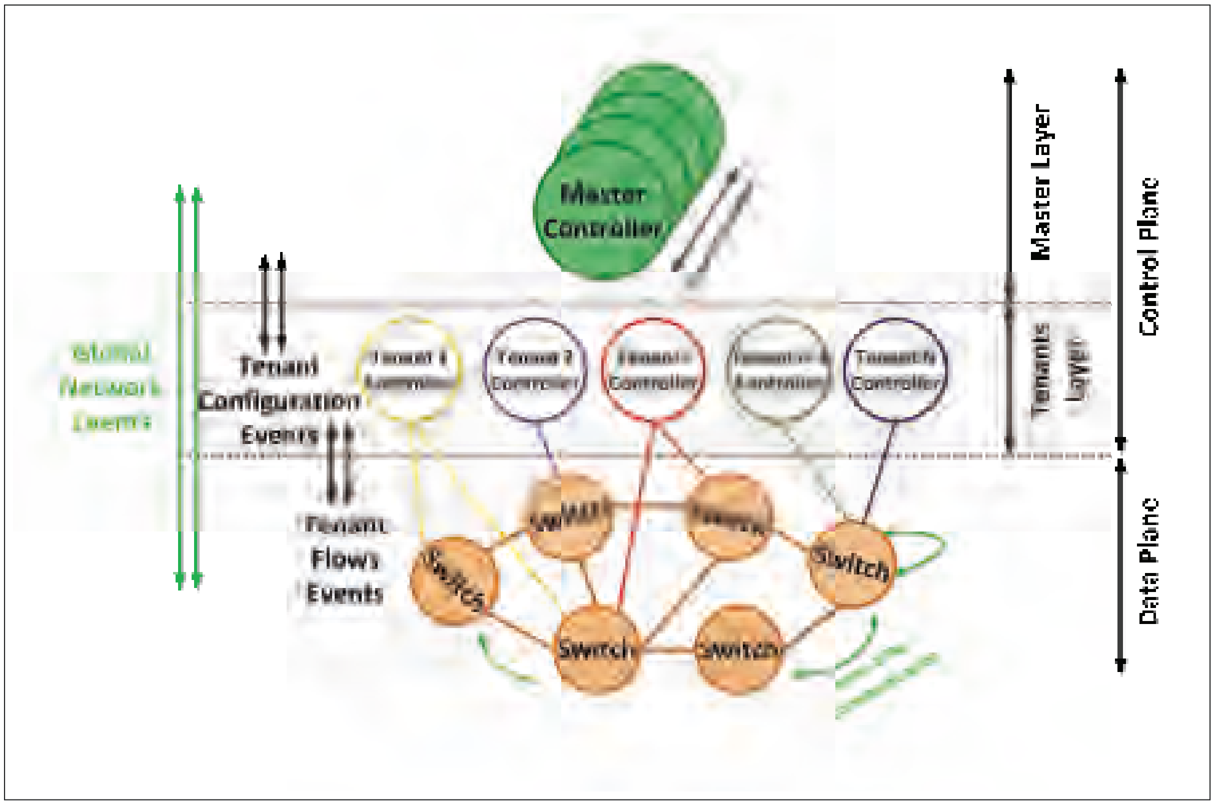

Figure 4.14 Offloading Centralized Controller's Functionalities

complicated security policies and VMs identification with a few lines which represent slice's allocated and shared data plane resources. The following representation of OpenNMS slice decreases the latency introduced by parsing operation of the controller back-end database which has a compressed size.

The centralized OpenNMS Master controller (OMC) solves the SDN scalability bottleneck and the cross-VTNs communications by managing inter-tenant slices sharing resources and access priority (e.g., load balancing). Once the tenant's slice is designed on OA, it will automatically be mapped into underlying physical network, and then configured on the individual switch leveraging SDN control protocol. The definition of logical plane makes it possible not only to hide the complexity of the underlying network but also to better manage network resources. It achieves reducing reconfiguration time of network services and minimizing network configuration errors.

In general, a network node needs to show its identities to prove its membership before it is allowed to join an OpenNMS slice. For that, we choose VM's Mac-address and VM's allocated OpenFlow switch port to identify tenant's VM. VTNs' VMs can communicate in the same or different platforms with need for tunnelling encapsulations to support remote communication. Our OA provides to OMC the required configuration with the ability of translation features to support heterogeneous protection and segmentation. In case of intra-slices communication, incoming flows are identified in OpenFlow switch using OMC from their L2 matching fields based on specific parameter as shown in Table 4.1. For inter-tenant's slices communication, before flow processing in the first OpenFlow switch, the specific tenant controller adds virtual tunnel ID for remote communication. The tagged frame is transmitted to its remote destination based on flow entries which are decided by the OMC at each hop.

Table 4.1 OpenNMS Master Controller Packets Verification Functions

OpenNMS Master Controller	**dp-src ID**	**port-dst ID**	**port-src ID**	**Mac-src**	**port Mapping**	**Mac Mapping**	**Tunnel Mapping**
Intra-VTSs	✓	✓	✓	✓	✗	✗	✗
Inter-Tenant's slices	✓	✗	✗	✗	✓	✓	✓

Notations :

- port-dst ID : Only for intra-VTSs communication, OA will search for port destination ID.

- port Mapping : For remote communication, untagged frame mapping is supported but also our design provides mapping between physical network interface (NIC) and OpenFlow switch interface using the datapath ID, destination MAC address and encapsulation Tag ID of the incoming L2 frame.

- Mac Mapping : Maps NIC to an interface of OpenFlow switch using MAC address of the incoming L2 frame.

- **Tunnel Mapping :** Maps NIC to a port using encapsulation tag ID of the incoming L2 frame.

4.6 L2 Isolation Model Flowcharts

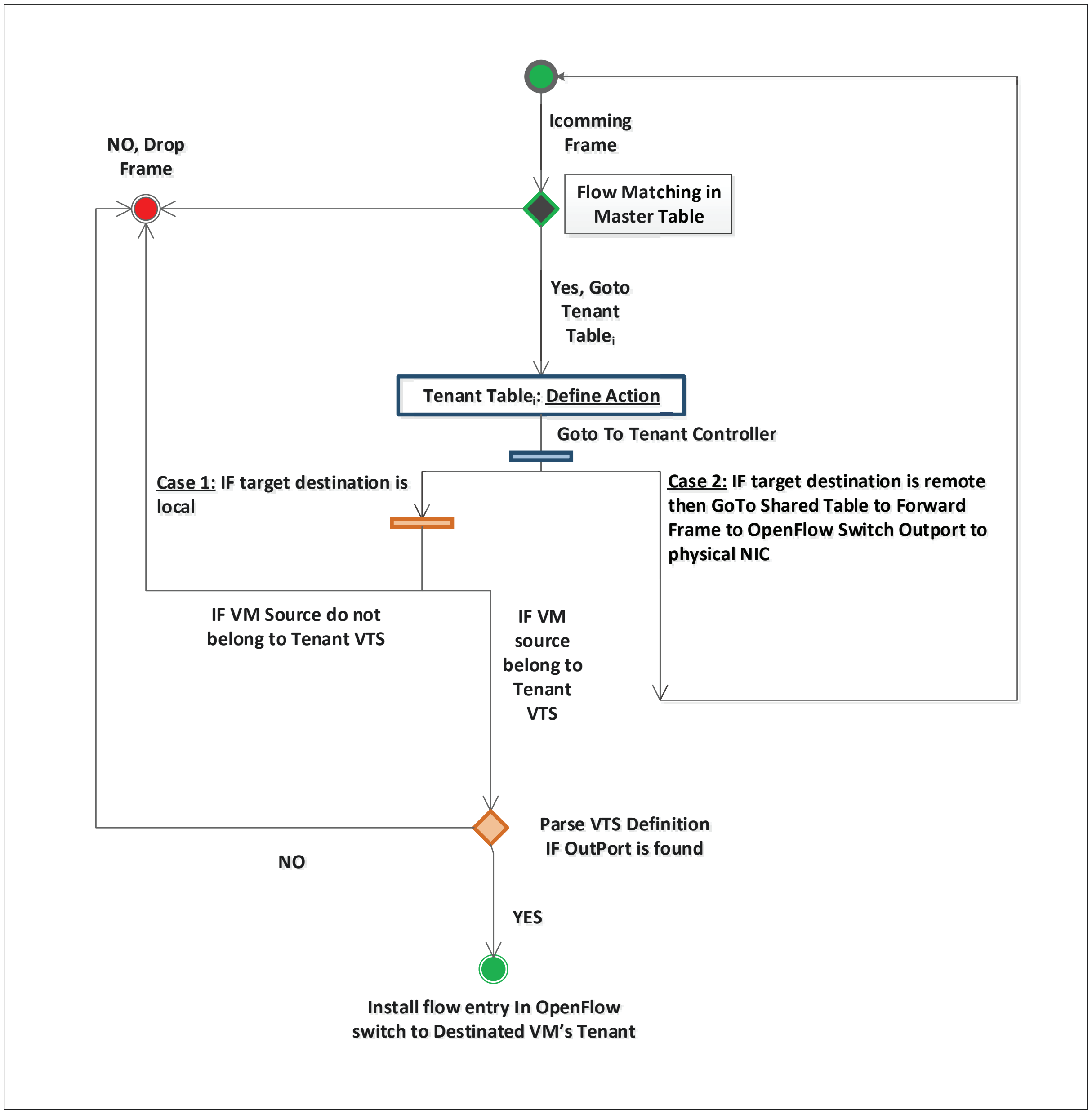
Icomming Frame
NO, Drop Frame
Flow Matching in Master Table
Yes, Goto Tenant Table$_i$
Tenant Table$_i$: Define Action
Goto To Tenant Controller
Case 1: IF target destination is local
Case 2: IF target destination is remote then GoTo Shared Table to Forward Frame to OpenFlow Switch Outport to physical NIC
IF VM Source do not belong to Tenant VTS
IF VM source belong to Tenant VTS
Parse VTS Definition IF OutPort is found
NO
YES
Install flow entry In OpenFlow switch to Destinated VM's Tenant

Figure 4.15 Packet Processing Diagram in OpenNMS Solution

After the development of the L2 isolation model and the OpenNMS architecture design, we now have a clear view of the concepts and components needed to achieve our goals. However, in order to have a better understanding of it, the following flowcharts will help us in the deployment and development phase in order to efficiently build a proof of concept of our scalable, flexible and autonomic multi-tenant network.

The first flowchart presented on Figure 4.15 summarizes the journey of OpenNMS packet. As follow, we present the detailed explication of the packet processing flowchart in OpenNMS network.

The initial situation is an incoming packet originated from any VM running on the same or different host in data center network. As an application installed on VM starts generating packets, the related hypervisor running on the same server as the VM intercepts these packets and forwards them to the OpenFlow switch. The packet reaches the first switch, where it is recognized as belonging to none of the existing flows, because it does not match any flow entry inside the OpenFlow switch's first flow table or OpenNMS Master Table. The pipeline processing always starts at the OpenFlow table 0. If the packet will be recognized as denied flow, it will be dropped, otherwise it will be forwarded to the Master controller. The Master controller have the capability to analyze the packet and decide what flow entries to push down the data path for the incoming packet. For all unrecognised packets in Master table, the first step is to determine destination of the flow. The Master controller parses his database to find any correspondence between the incoming packet and tenants slices and networks. The packet's destination can be identified using the destination IP address, destination MAC and encapsulation tag identifier (e.g., VXLAN). Once the Master controller holds a VTS matching with the arriving flows, it defines the necessary action and installs new flow entry in Master tables to forward the packet to the tenant's allocated tables for the matching VTS. Finally, the tenant controller must determine the destination port for the incoming flow, it can be forwarded to in-port or out-port depending on the target destination and source.

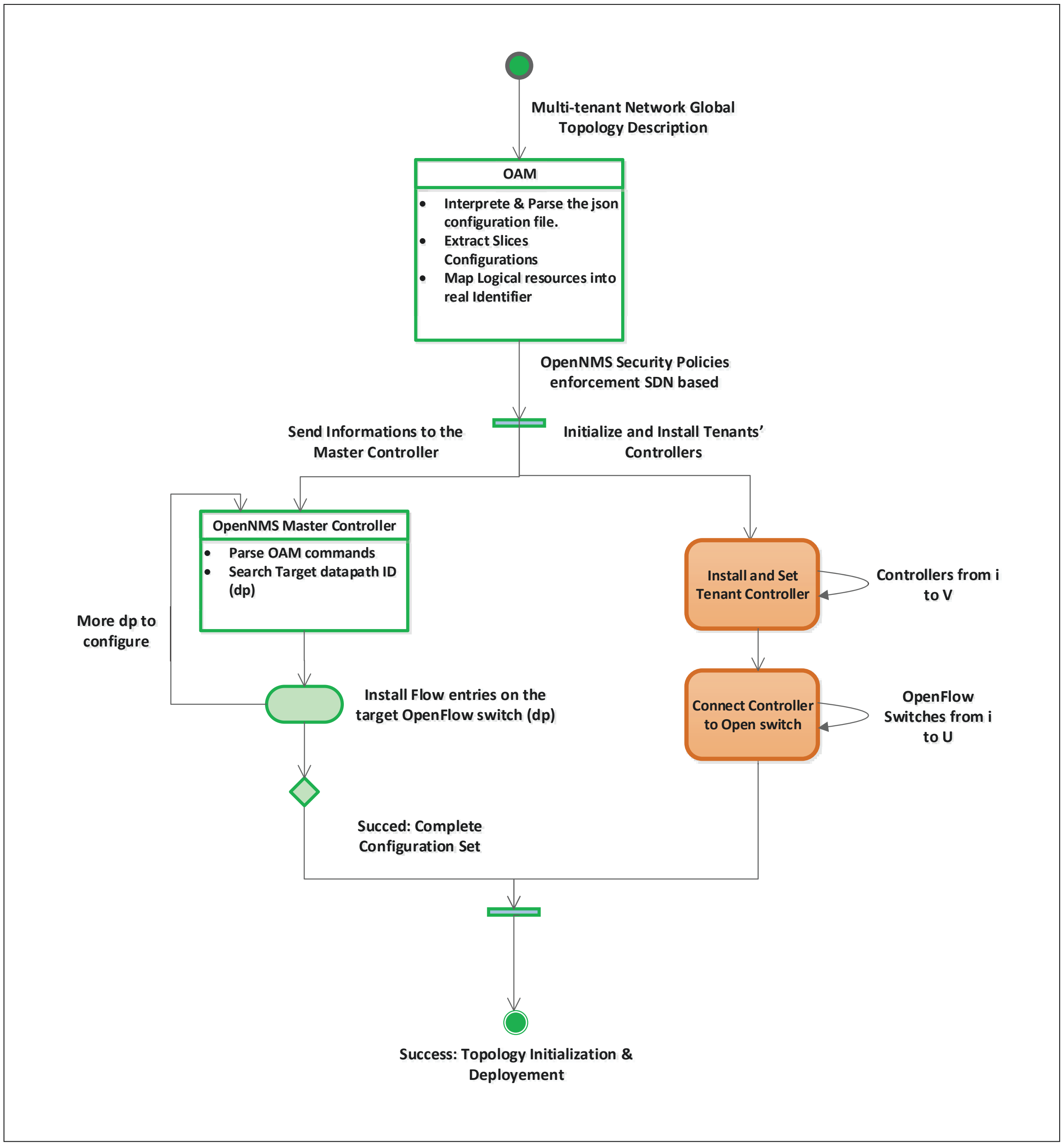
Multi-tenant Network Global Topology Description
OAM
Interprete & Parse the json configuration file.
Extract Slices Configurations
Map Logical resources into real Identifier
OpenNMS Security Policies enforcement SDN based
Send Informations to the Master Controller
Initialize and Install Tenants' Controllers
OpenNMS Master Controller
Parse OAM commands
Search Target datapath ID (dp)
More dp to configure
Install and Set Tenant Controller
Controllers from i to V
Install Flow entries on the target OpenFlow switch (dp)
Connect Controller to Open switch
OpenFlow Switches from i to U
Succed: Complete Configuration Set
Success: Topology Initialization & Deployement

Figure 4.16 Multi-tenant Network Deployment using OpenNMS

The second flowchart presented on Figure 4.16 summarize the initialization and deployment of Multi-tenant using our OpenNMS approach. To illustrate how our model works, we describe in this flowchart the processes needed to deploy a multi-tenant topology.

a. The first step is the build of the JSON file including all virtual tenants networks and related virtual slices.

b. After, receiving the multi-tenant topology JSON structure, the OpenNMS Autonomic Manager (OAM) will interpret and parse all contained elements. Next, OAM extracts the VTNs and VTSs configurations and finally maps their logical identifier into real one.

c. Based on our L2 isolation model, OAM will checks the configuration of all elements. Then, if it matches with the security strategy selected, in parallel, the manager will initialize and install the required components for these networks (tenant controllers and secure connection to OpenFlow switches) and deploy the required flow entries in OpenFlow master tables.

Similar to the previous flowchart, the VTS configuration and update require similar processes and tasks on the OAM. The third flowchart presented on Figure 4.17 illustrates adding and update of tenant slice at run time by the service provider.

For self-controlling and self-management capabilities, our system supports also the update and re-scaling operations from the network tenant. The last flowchart presented on Figure 4.18 illustrates the tenant update operation life cycle for applying the required changes on the network.

4.7 Summary and Conclusions

In this chapter, we introduced Open virtual Network Management & Security (OpenNMS), an autonomic SDN architecture for supporting multi-tenancy and providing elastic isolation between tenants' slices. Our architecture aims to automate the instantiation of a virtual infrastructure while dynamically deploying the corresponding security mechanisms to enforce the network isolation between different tenant networks. This deployment is driven by tenants' global isolation policy, and thus covers all resources. Our contribution is a new method that manages virtual switches to explore, solve the scalability concerns in the SDN design infras-

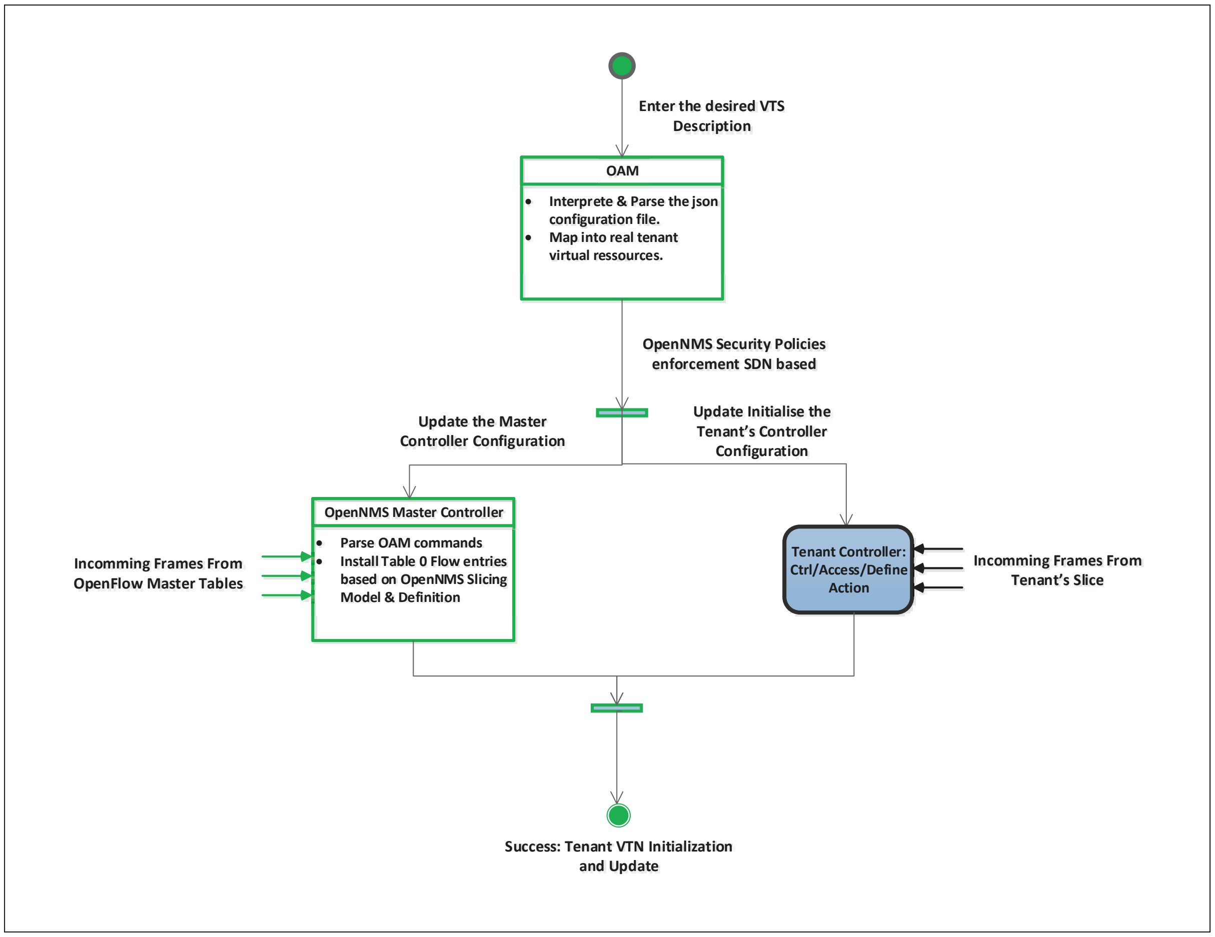
Enter the desired VTS Description
OAM
Interprete & Parse the json configuration file.
Map into real tenant virtual ressources.
OpenNMS Security Policies enforcement SDN based
Update the Master Controller Configuration
Update Initialise the Tenant's Controller Configuration
OpenNMS Master Controller
Parse OAM commands
Install Table 0 Flow entries based on OpenNMS Slicing Model & Definition
Incomming Frames From OpenFlow Master Tables
Tenant Controller: Ctrl/Access/Define Action
Incomming Frames From Tenant's Slice
Success: Tenant VTN Initialization and Update

Figure 4.17 VTS Configuration Initialization and Update using OAM at run time

tructure while providing high flexibility and performance for packet processing and keeping the benefits of network control centralization. The OpenNMS architecture gives each tenant the control of its own isolated space, topology and controller and the ability for cross-domain resources sharing.

We have described new capabilities for multi-tenant network, Split, Merge and migrate, that enable transparent, balanced elasticity, flexible re-provisioning and re-scaling. Using our L2 isolation model with the high level abstraction for tenant networks and slices, tenant distributed resources among slices which can be split into two or many VTS, merged together into a single VTS, or migrated to new location separately or in group . At the same time, we ensure tenant services chaining and traffic steering required for such requirements. As networks become

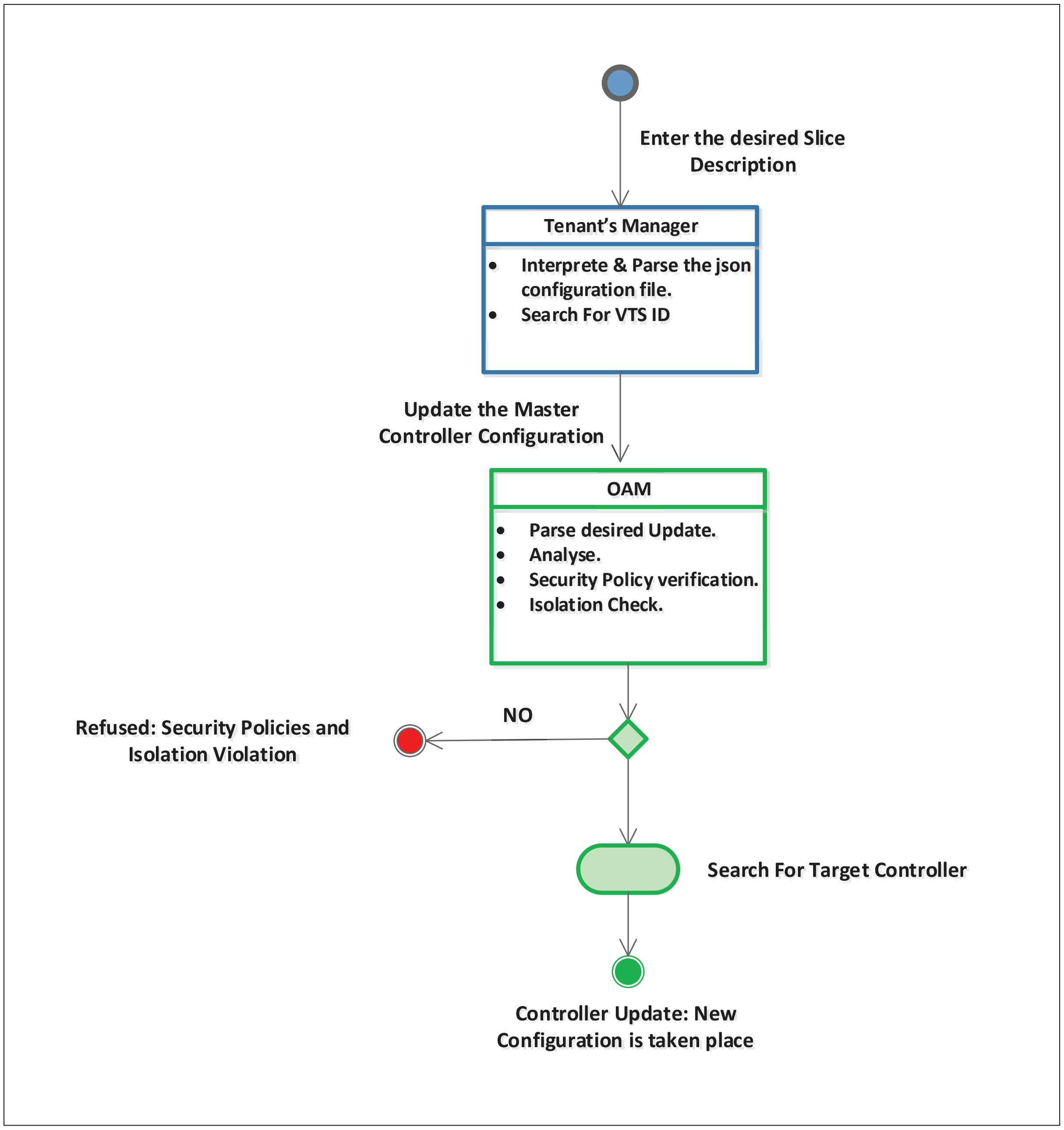
Enter the desired Slice Description
Tenant's Manager
Interprete & Parse the json configuration file.
Search For VTS ID
Update the Master Controller Configuration
OAM
Parse desired Update.
Analyse.
Security Policy verification.
Isolation Check.
NO
Refused: Security Policies and Isolation Violation
Search For Target Controller
Controller Update: New Configuration is taken place

Figure 4.18 VTS Update and Re-Scaling by Tenant

increasingly virtualized, the cloud addresses a need for elasticity in tenant networks and the ability to split, merge and migrate tenant allocated resources is becoming even more important.

Our approach demonstrates the simplicity and the feasibility of Software-Defined security mechanisms. The flexibility gained through this approach helps to adapt the network dynamically to both unforeseen and predictable changes in the network. It offers the possibility to run multiple slices within the same logical and physical switches while adapting the trust level to the desired performance and optimized usage of the allocated resources.

CHAPTER 5

OPEN NETWORK MANAGEMENT & SECURITY ARCHITECTURE IMPLEMENTATION & EVALUATIONS

In this chapter, we present the deployment model of Open Network Management & Security (OpenNMS) architecture, implementation, testbed and evaluation results.

This chapter will cover the experiments and tests with which we want our implementation to go through for each objective defined in this work. We will describe the scenarios that we made in order to gather the proof we want. We will explain our testbed as well as the materials, software and tools used for our prototype.

After the deployment and implementation of our prototype is done and the testbed is created, it was time to start testing our OpenNMS architecture. For each goal that we have succeeded to achieve, we planned different experiments to demonstrate its feasibility with our architecture and then we made observations on the behavior of our multi-tenant network. Thereafter, we analyzed the obtained results and compare it with the actual approaches. For each scenario, we justified the adopted choices as well as the tools that used for each experiment.

Our experiments will start by overhead evaluation as important criteria for the adoption of our design and L2 isolation model in multi-tenant network. Next, we will test all types of our slicing model and compare it with unmodified NOX controller. We evaluate the flexibility of OpenNMS in aim to reach and demonstrate the feasibility of Trusted and Restricted sharing methods as well as the elasticity aspect using Split, Merge and Migrate (SMM) capabilities. We will end our experiments by demonstrating the scalability of SDN controller and how we overcome the blocked bottleneck of SDN architectures. Finally, we will expose our design limitations at each experiment as well as the potential solutions and improvements which can be brought to our approach.

5.1 Deploying OpenNMS OAM and OA

For improving management plane in SDN architecture, we use in the implementation OF-Config (Pfaff and al.) and NetConf (Enns *et al.* (2011)) protocol for OpenFlow which are emerging technologies for improving network management.

OAM is implemented as a plug-in for IaaS Framework (Figuerola *et al.* (2009)), it is developed using Scala (Cremet and Odersky) language and CouchDB (Anderson *et al.* (2010)) as Datastore which saves slicing definitions and policies. OAM is responsible for checking slice membership and identification in messages received in NetConf XML format. It answers OA identity confirmation requests. OA is implemented as a Native C++ extension to NOX 1.3 OpenFlow controller (See Appendix I).

5.2 OpenNMS Lab Setup and Testbed Scenario

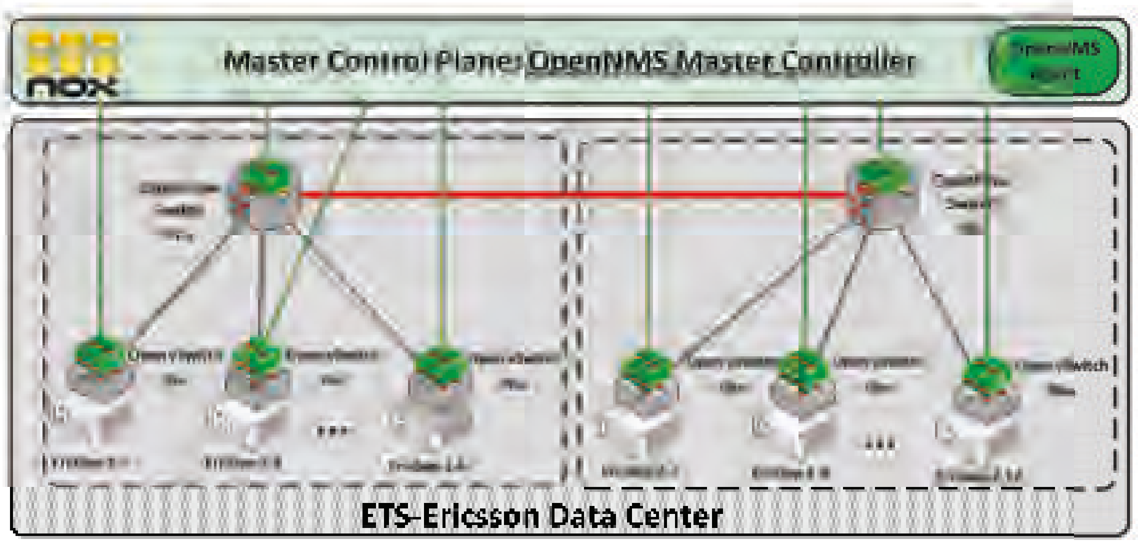

Figure 5.1 OpenNMS Lab Setup

As illustrated in Figure 5.1, We used 12 servers on Ericsson Blade System (EBS) interconnected with two 100 Gbps Ethernet OpenFlow-based switch (dp_0 and dp_1). Each server is an x86 based general processing board within EBS and utilizes a 64-bit, 6 cores, XEON L5638 running at 2.00 GHz and RAID support chipset with 24GB memory with several variants (HDD, SSD, E1/T1).

Using KVM virtualization and Open vSwitch Software (Kis (2012)), we have created 8 virtual machines per host ($VM_1 ... VM_{96}$) linked to OpenFlow virtual switch running on each host ($dp_2 ... dp_{12}$). All switches supports OpenFlow 1.3 version. We run a NOX OpenFlow 1.3 controller (Fernandes *et al.* (2012)) with our OA as extension which links the OpenFlow switches and presents the centralized OpenNMS Master controller. The switches are based on the Ericsson software implementation (Kis (2012)), with a modification in the forwarding plane to support OpenFlow 1.3. This switch is based on Nicira's Open vSwitch, in which OpenFlow processing model is replaced with "Oflib" from the OpenFlow 1.3 Software Switch.

We start our evaluation by deploying a simple scenario similar to the architecture in Figure 3.2, with two tenant ($tenant_1$ and $tenant_2$) and each $VTN_{i,i=1,2}$ with four VTSs; each VTS has 1 VM running on each server. For each VTN, we use one NOX to control tenant allocated resources. Tables 5.1, 5.2 and 5.3 represent the shared and dedicated resources in this scenario.

Table 5.1 OpenNMS Tenants Dedicated Resources

VTN_{ID}	VTS_{ID}	**Dedicated resources**
VTN_1	$VTS_{1,1}$	$dp_{i.t_1}, dp_{i.gt_1}, dp_{i.p_2}$
	$VTS_{1,2}$	$dp_{i.t_2}, dp_{i.gt_2}, dp_{i.p_3}$
	$VTS_{1,3}$	$dp_{i.t_3}, dp_{i.gt_3}, dp_{i.p_3}$
	$VTS_{1,4}$	$dp_{i.t_4}, dp_{i.gt_4}, dp_{i.p_4}$
VTN_2	$VTS_{2,1}$	$dp_{i.t_6}, dp_{i.gt_6}, dp_{i.p_5}$
	$VTS_{2,2}$	$dp_{i.t_7}, dp_{i.gt_7}, dp_{i.p_6}$
	$VTS_{2,3}$	$dp_{i.t_8}, dp_{i.gt_8}, dp_{i.p_7}$
	$VTS_{2,4}$	$dp_{i.t_9}, dp_{i.gt_9}, dp_{i.p_8}$

Notations :

$dp_i \in Dp \ / \ \forall \ i \ in \ \{1..12\}$

$dp_{i.p_j}$: dp_i's p_j

$Dp_{i.t_j}$: dp_i's t_j

$S_{i.gt_j}$: dp_i's gt_j

Table 5.2 OpenNMS Tenants' Slices Shared Resources and Trust Level

VTN_{ID}	$VTSs_{ID}$	**Shared resources** $/ \ \forall \ i \ in \ \{1..12\}$	**Sec. Enforcement**
VTN_1	$VTS_{1,1}$ & $VTS_{1,2}$	$dp_{2.t_5}, dp_{2.gt_5}, dp_{i.p_1}, dp_{2.p_2}$	Restricted
	$VTS_{1,2}$ & $VTS_{1,3}$	$dp_{5.t_5}, dp_{5.gt_5}, dp_{i.p_1}, dp_{1.p_3}, dp_{1.p_4}$	Trusted
	$VTS_{1,3}$ & $VTS_{1,4}$	$dp_{i.p_1}$	Basic
VTN_2	$VTS_{2,1}$ & $VTS_{2,2}$	$dp_{12.t_{10}}, dp_{12.gt_{10}}, dp_{i.p_1}, dp_{12.p_5}$	Restricted
	$VTS_{2,2}$ & $VTS_{2,3}$	$dp_{8.t_{10}}, dp_{8.gt_{10}}, dp_{i.p_1}, dp_{8.p_6}, dp_{1.p_7}$	Trusted
	$VTS_{2,3}$ & $VTS_{2,4}$	$dp_{i.p_1}$	Basic

Table 5.3 OpenNMS Master Controller Tables

	OpenFlow Switches Tables
OpenNMS Master Tables	$dp_{i.t_0}$ and $dp_{i.gt_0} \ / \ \forall \ i \ in \ \{1..12\}$

5.3 Experimental Analysis

5.3.1 Performance Overhead

In the following, we evaluate the overhead generated by involving OA extending the NOX controller for enforcing our L2 isolation. Since the Master controller intercepts all tenants' first incoming and outgoing packets and installs the required flow entries on the corresponding datapaths, we measured its per-packets overheads by emulating random traffic using hping and Iperf traffic generators from the 96 VMs running on the top of our 12 servers. Our hping experiments used for latency measurement with 100 TCP packets per second. The Iperf traffic generator can be used for network bandwidth performance on one way (client $\rightarrow$ server) and bi-directional tests (client $\leftrightarrow$ server). We compare our OpenNMS results against an unmodified NOX.

The first row in Table 5.4 shows the results of latency evaluation. OA added few microseconds to the latency time compared to basic NOX. We measured both one-way and bi-directional TCP invocation throughput using Iperf. For each case, we ran over 100 iterations with default 1470 bytes packet size. The second and third rows in Table 5.4 show throughput results. OpenNMS causes a 0.556 % decrease in mean one-way throughput, and a 1.84 % drop in mean bi-directional test. Our L2 isolation model and design guarantees also negligible packets drops with the minimal MTU[1].

Table 5.4 OpenNMS Architecture Overheads

Tool	**Metric**	**Basic NOX**	**NOX with OpenNMS**
hping (in μs)	Avg	96	98
	Min-Max	90-112	94-118
Iperf One-Way (in Mbps)	Avg	981.91	976.45
	Min-Max	980.11-981.98	975.23-978.63
Iperf Bi-directional (in Mbps)	Avg	1920.11	1884.64
	Min-Max	1910.4-1925.88	1880.23-1917.46

5.3.2 OpenNMS L2 Slicing Model and Types Evaluation

Our Second experiment uses a the dedicated packet generator hping to evaluate the VTS types $St_{i,i=0..3}$. In Figure 5.2, we generate 100 UDP flows per second, then we change the rate

[1]Maximum Transmission Unit

from 100 to 1000 to compare the delay of different slicing types. The figure shows that the variation of delay comparing to NOX is negligible. Even for VTS types, there is no remarkable difference in delay between VTS types.

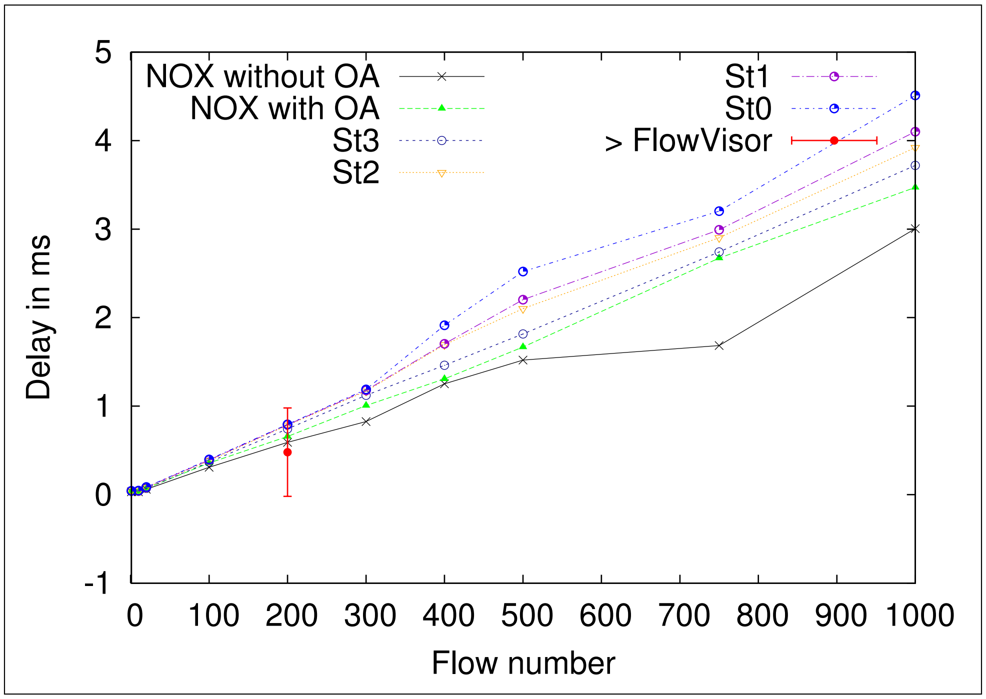

Figure 5.2 Delay imposed over UDP flows by OpenNMS with different VTS Types

While it is difficult to establish direct comparison with others SDN slicing approaches like FlowVisor (Sherwood *et al.*, 2009) since they have implemented their solutions using local hypervisor on physical OpenFlow switches. FlowVisor results show that including the additional isolation layer in physical switches causes an average overhead for responses of 0.48 milliseconds with 200 flow per seconds. With the same number of requests, OpenNMS has higher delay of 0.17 milliseconds (See Figure 5.2). However, this delay is acceptable seen we use distant controller handling the isolation tasks.

Figure 5.3 illustrates the pushing time of one flow entry by the NOX controller and the corresponding flow latency with the four slicing types. There is also no variance between our slicing types in pushing one flow entry time. As results, we add maximum 90 μs to the flow latency which does not affect the network performance.

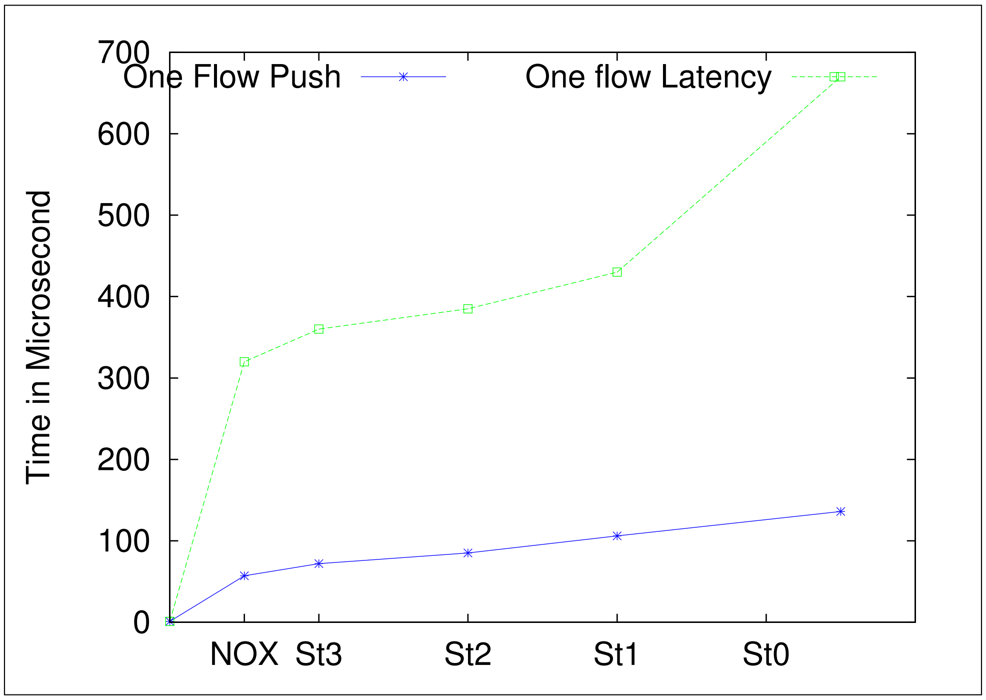

Figure 5.3 Pushing one flow entry with different slicing levels

5.3.3 Multi-tenant Network Scalability

We did more experimental analysis for the OpenNMS isolation model and architecture design to evaluate the scalability of multi-tenant network. To motivate the efficiency and robustness of our model, we evaluate the OpenNMS's scalability, performance, and capacity properties for supporting the fast and unplanned growth of the multi-tenant network. We have asserted before that OpenNMS can scale to large numbers of VMs and switches and can support several thousands of isolated VTNs and VTSs on the top of the shared infrastructure.

Our current testbed is limited with four VTSs in each OpenFlow-enabled switch and 96 VMs distributed over 12 hosts. So, to demonstrate the OpenNMS scalability, we emulated a much larger number of VMs in our testbed that could normally run on 12 hosts using Mininet

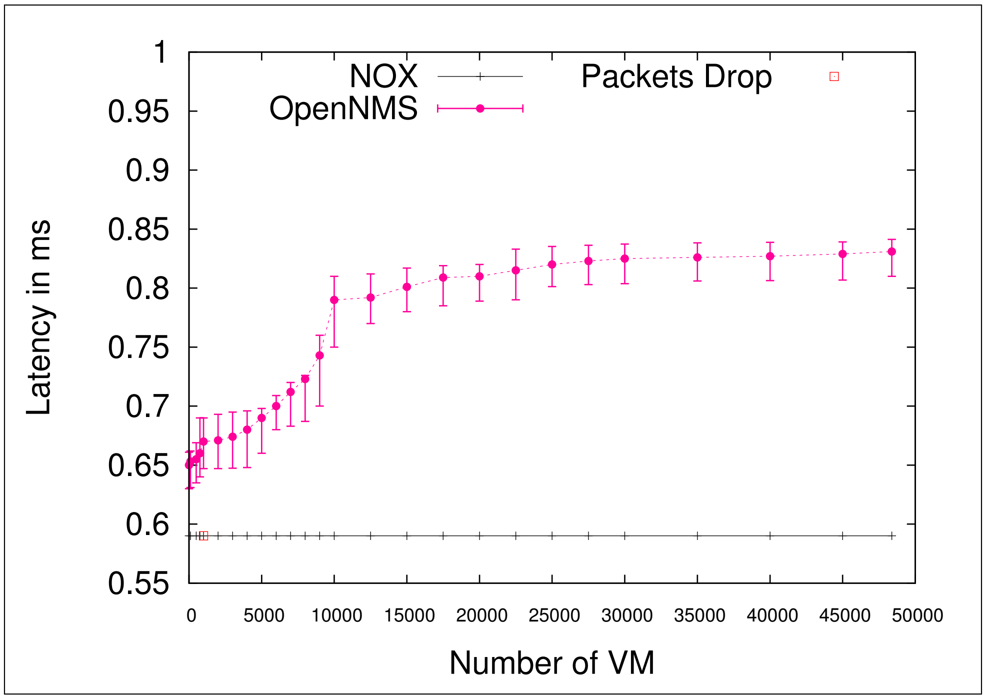

Figure 5.4   Latency vs. VM count

(Mininet; de Oliveira *et al.* (2014)). We implemented Mininet on each server to run four thousand more VMs linked to running Open vSwitch and assigning a capacity of 100 Mb/s to each link. Using this technique, we can emulate up to 48K of VMs in the multi-tenant data center. We have created thousand VTS more per host and we assign each pair of VTSs for different tenant and VTN. We allocated one Mininet's VM for each VTS per host. With our real testbed and this emulation, we have now up than 24.002 VTNs and VTSs. By generating random rate of TCP requests from random VTSs and VMs using hping, we compared unmodified NOX to OpenNMS. We ran several trials with varying numbers of tenants, related VTNs, VTSs and VMs.

Figures 5.4 and 5.5 show that:

a. NOX does not scale as anticipated. Aside from failing to support the higher number of networks. It drops incoming packets after the number of VMs exceeds one thousands.

b. OpenNMS scales well with negligible overhead even with large number VTS and VM. Figures show that the latency remains reasonable and flow processing is not affected by the increased number of VTSs and VMs. We observe that OpenNMS's latency increases slightly as the number of VMs exceeds one thousand. We suspect that this peak is caused by Open vSwitch's links capacity for supporting more than 10K of VMs.

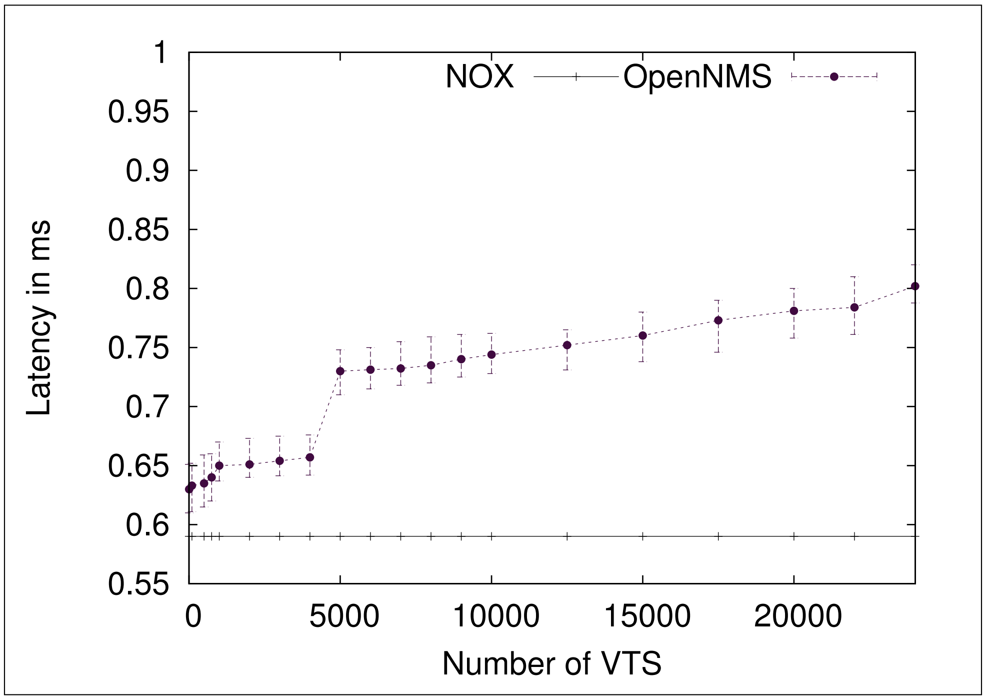

Figure 5.5 Latency vs. VTS count

5.3.4 Inter Tenant's Networks Sharing Methods: Trusted and Restricted Security Levels

For evaluating the flexibility of our design, trusted level was chosen as one of the services provided to the tenant with the first experiment. Figure 5.6 resumes testbed scenario for scaling tenant's VTSs by sharing VMs between them as trusted resources. For that, the tenant chooses scale-in option and checks the list of shared resources and the method of security enforcement (See section 4.4.1). This scenario evaluates the latency time of network response to the updated configuration. Table 5.5 shows the result of sharing two VMs from different VTSs ($VTS_{1,2}$ and

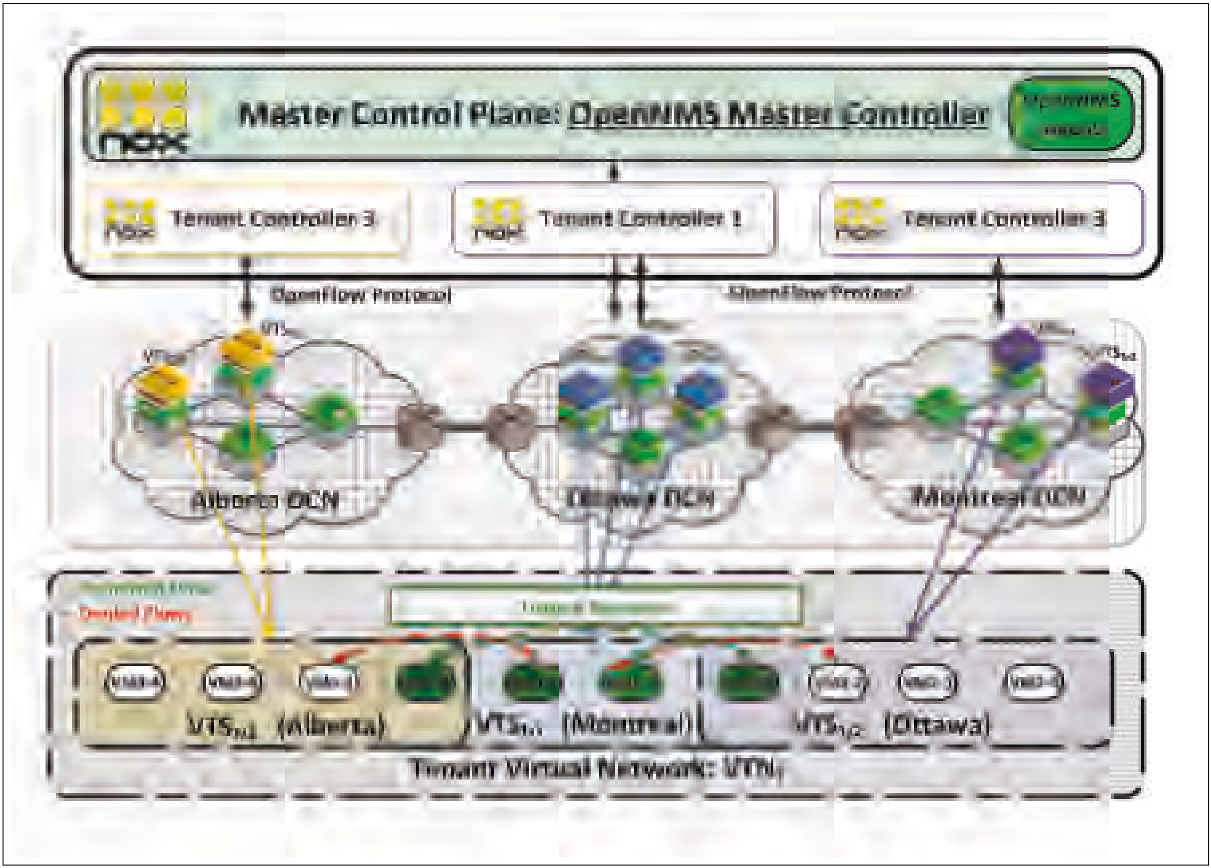

Figure 5.6 Inter Tenant Networks Scalability Scenario: Trusted/Restricted Methods

$VTS_{1,3}$) with the first $VTS_{1,1}$. Network configuration can be updated in 2 milliseconds without network packets processing interruption which is an acceptable response time for all OpenNMS architecture's involved components.

Table 5.5 OpenNMS Cross-Planes Response Time for Tenant Scalability

OpenNMS Action	**Response Time (ms)**
Upload New configuration to tenant controllers	0,35
Master Controller Notification	0.52
OAM Cost	0.62
Master Controller Network Update	0.72
Total	**2,21**

For evaluating the restricted level, we used the first testbed in Figure 5.1. The four tenant's VTSs have the same size in term of numbers of allocated virtual resources. This experiment consists of comparing the two level of security enforcement in aim to scale tenant network and slices. We have selected random two tenant's VTSs in each host for enforcing the trusted level and the remain VTSs for restricted level. As total, we will have 12 shared resources for each level to apply. We evaluate in table 5.6 the latency time of network response to the updated configuration in each component of OpenNMS architectures. Network configuration can be updated at maximum 3.5 milliseconds (See Table 5.6) which is also an acceptable response time. For the restricted configuration, we have observed some interruption and few packet drops with 20 μs due to the added processing and checking time for this level comparing to the trusted one.

Table 5.6 OpenNMS Scaling Methods Response Time: Trusted and Restricted Scenarios

OpenNMS Action	**Trusted (ms)**	**Restricted (ms)**
Upload Desired Configuration to Tenant Controllers	1,15	1.58
Master Controller Notification	0.22	0.37
OAM Cost	2.14	3.31
Master Controller Update	0.05	0.13
Total Time	**3.56**	**5.39**

For the inter-tenant and external communications, currently OpenNMS requires that the tenant enters some transformation within its own VTNs and VTSs, if a VTS's VM needs to communicate with external VM in the same of different data center. Our design is supporting such requirement by providing dynamic inter and intra VTSs policies adaptations.

Scalability and flexibility in such multi-tenant architecture as OpenNMS require much more extensive testing than the experiments reflected in this thesis. We mean, for instance, experiments within scenarios stressing specific planes or components of planes. This thesis reflects only the results of scalability tests in one particular scenario. Then we have to emphasize that our system scales well under the conditions of this particular scenario, and that by no means can these results be generalized to different scopes or situations. Additional testing is in fact part of challenging future work.

5.3.5 Autonomic Design Performance

Beside the total cost evaluation in last experiments, we have not yet measured the detailed costs for different tasks of the autonomic management plane, including OAM and tenant manager.

Table 5.7 OpenNMS Management Planes Detailed Costs

Task	**Autonomic Management Time**	**Tenant Manager Time**
Tenant Controller Setup	2,35 s	–
VTS Setup	1,45 s	2,12 s
VTN Setup	2.43 s	3.01 s
VTS Tranformation	1,24 ms	1.53 ms
VTN Tranformation	2 ms s	2,71 ms
Open vSwitch Setup	5,11 s	–
VM Setup	3,05 s	4.62 s
Port Setup	45 ms	61 ms
Flow entry Setup	32 ms	51 ms

We measured the setup time for SDN components including Open vSwitch and controller. We also evaluate the time needed for the instantiation of new VTS and VTN and the configuration of data plane that requires port and flow entry setup. We consider in our evaluation a VTS composed of four ports, tables and group tables on one OpenFlow switch. The VTN in our experience is grouping four VTSs distributed over the data center OpenFlow switches. The setup cost of OAM includes the time required for control loops and communications with the master controller. The VTS and VTN tranformation includes the necessary time for topology update and mapping and the flow entries setup and update. Table 5.7 shows the results of our evaluation. We can conclude that the overhead associated with the hierarchical managers is a fairly low (18.95 %).

5.3.6 Elasticity: Split, Merge and Migrate

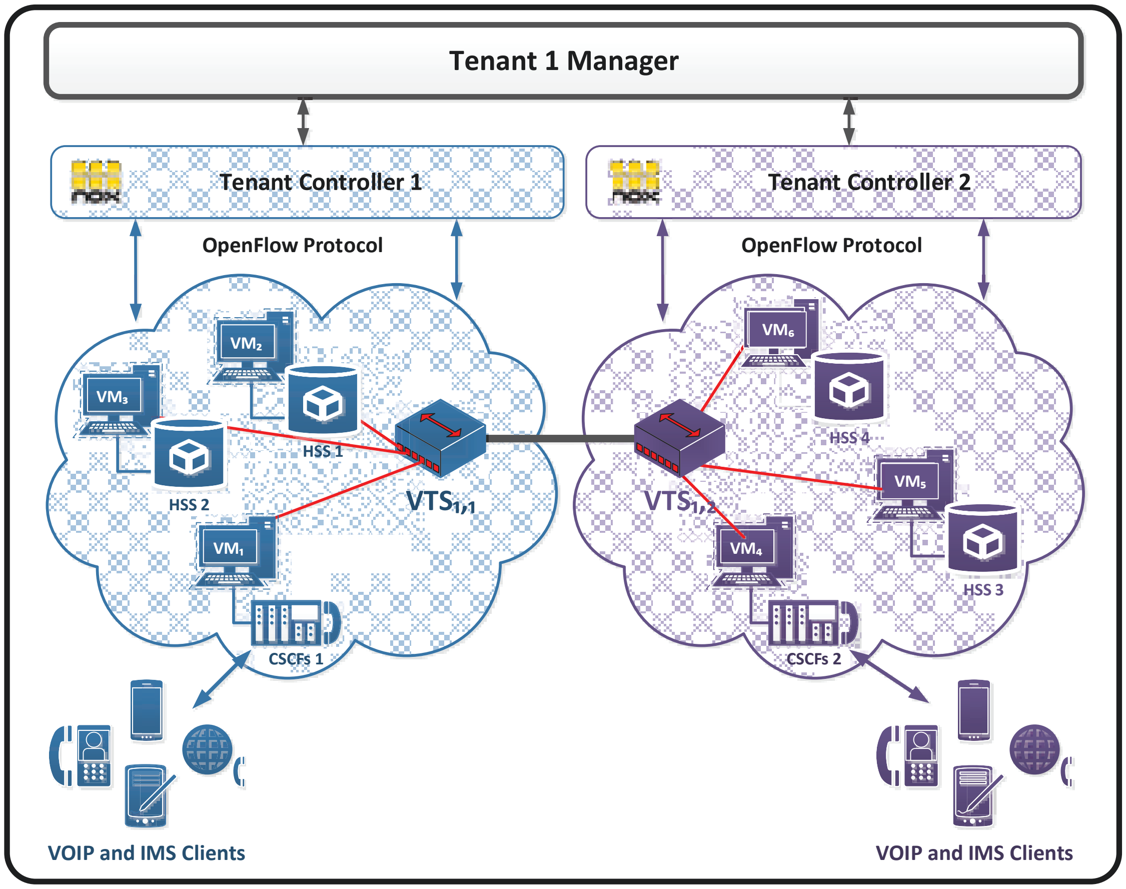

Figure 5.7 Elastic Multi-tenant Network Evaluation Scenario

We now evaluate how well our design and isolation model is suitable for the elastic nature of cloud computing. We demonstrate our first step towards elastic multi-tenant network by the feasibility and robustness of our Split, Merge and Migrate (SMM) capabilities (See section 4.4.2).

We use for the elasticity experiment two tenant VTSs ($VTS_{1,1}$ and $VTS_{1,2}$) instantiated on different sites. For each VTS, we deploy an OpenIMS (Vingarzan *et al.* (2005)) application, consisting of IMS Call Session Control Functions (CSCFs) and Home Subscriber Server (HSS) (See Figure 5.7). We use the configuration in table 5.8. Each VTS is responsible for a group of clients and controlled by independent tenant controller.

Table 5.8 Elastic Simulation Configuration

Simulation Parameter	**Description**
CSCFs CPU	8 vcpus
CSCFs memory	20 Gbytes
HSS CPU	2 vcpus
HSS memory	4 Gbytes
CSCFs/HSS bandwidth requirement	100 Mbps/flow

In the following, our objective of using SMM capabilities is enabling load balancing between the two VTSs' IMS components. We evaluate OpenNMS with the following goals:

- demonstrate OpenNMS's ability to provide dynamic elasticity for the "pay-as-you-go" cloud model in a real multi-tenant network,
- show OpenNMS's ability to merge tenant slices to support the burden load of calls on IMS's CSCFs,
- demonstrate the ability to scale down by split the merged VTSs when calls return to normal threshold,
- measure the gain in resource utilization when scaling in a deployment using OpenNMS's SMM capabilities, and
- quantify the performance overhead of migrating a portion of VTS under different IMS loads.

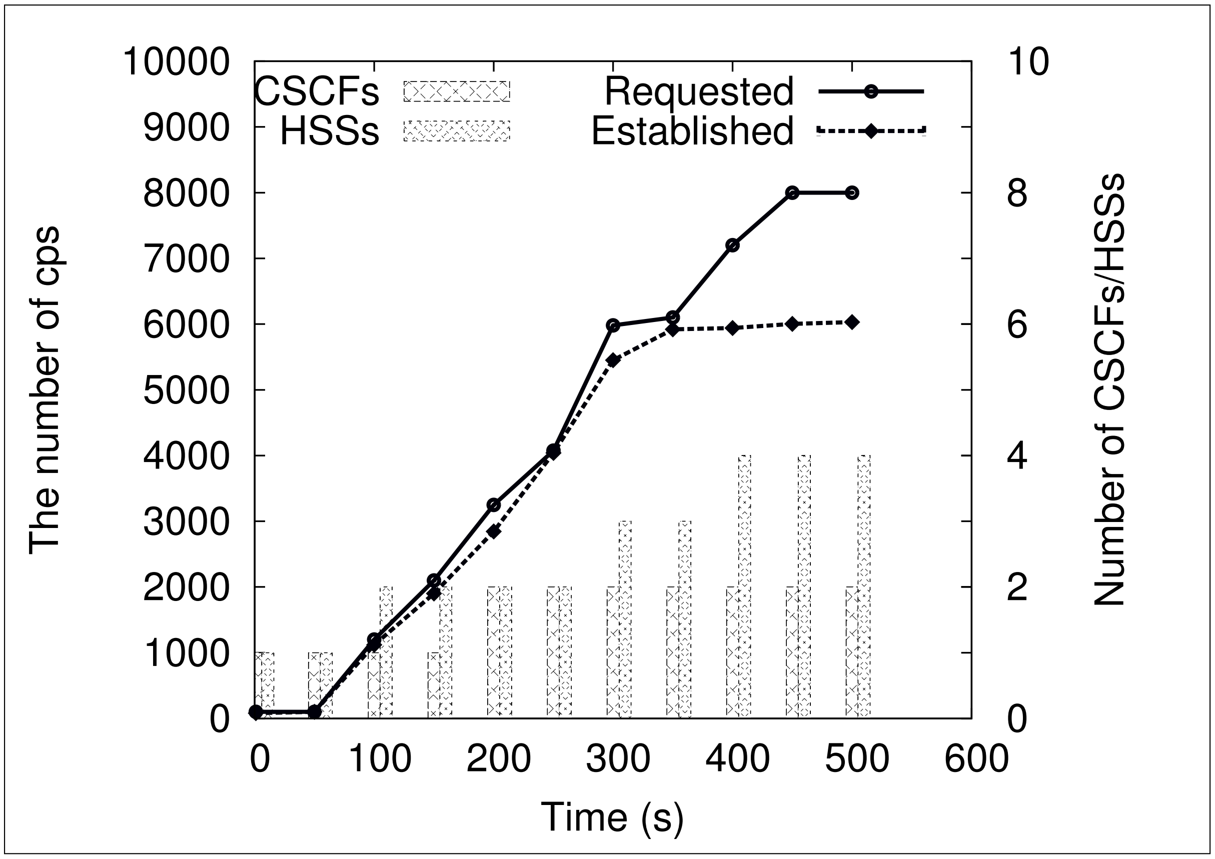

Figure 5.8 Scaling CSCFs by Merging Tenant VTSs

We set up a challenging performance scenario of IMS network to demonstrate the benefits of SMM capabilities to solve unplanned situation such as the sudden overload on IMS's CSCFs and HSSs. We have installed "SIPp" traffic generator tool (Gayraud *et al.* (2007)) on four IMS clients VMs.

We start generating traffic from the first VM client to $VTS_{1,1}$'s $CSCFs_1$ with a rate of 100 calls per second (cps). We induce $CSCFs_1$ scaling by increasing the request rate with 2000 cps each 100 seconds period; scale down is induced by decreasing the request rate to 100 cps. The increased number of cps will stress both IMS's CSCFs and HSS components.

Our first concern is the ability to detect the presence of a bottleneck with IMS components in order to initiate scaling. Therefore before starting our elasticity experiment, we have ran preliminary test for the CSCFs and HSSs capacity with the current configuration. We found that HSSs cannot support up to 2K cps and for CSCFs up to 4K. We have programmed the tenant $VTS_{1,1}$ to load balance the requests over 2K cps for HSS_1 using HSS_2. The merge with $VTS_{1,2}$ will start with more than 3K cps. $VTS_{1,1}$ scales up first by adding $CSCFs_2$, then HSS_3, and finally HSS_4. The results is a complete merge of $VTS_{1,1}$ and $VTS_{1,2}$ into one VTS managed by the tenant controller 1.

OpenNMS response time for the IMS components bottleneck is a simple and effective metric for the merging capability. When the request rate increases from 100 to 2000 cps, causing a bottleneck in HSS_1, the load balancing setup with HSS_2 has required just 5 milliseconds and the merging time with each $VTS_{1,2}$'s components ($CSCFs_2$, HSS_3, and HSS_4 in sequence) was only 11 milliseconds on average.

The behavior of our merged VTSs is shown Figure 5.8, which depicts the number of calls per second and the number of CSCFs and HSSs used over time while we enabled dynamic elasticity between tenant's slices.

With only one CSCFs and one HSS, the IMS network can support only 2K cps. The first scaling action increases calls capacity to 2500 cps by using two HSSs. The first merging action with $VTS_{1,2}$'s $CSCFs_2$ and HSS_3 has increased calls establishment to 6K cps. Finally, scaling to four HSSs was tried, but the 4^{th} HSS is not solving the scalability bottleneck due to the capacity of the two CSCFs which are not sufficient enough to support the burden load of calls.

As showed in Figure 5.9, the $VTS_{1,1}$ obtained from the merging with $VTS_{1,3}$ behaves in the same manner as a single slice, until the decrease of load burst is detected around t = 200s. Similar the merging scenario, we decrease the request rate with 2000 cps each 100 seconds period. While $VTS_{1,1}$ starts to deactivate load balancing between HSS_4 and HSS_3, the IMS network have enough resources for the incoming requests. At second 250, the tenant manager start splitting the two VTSs by eliminating HSS_3 from $VTS_{1,1}$ and finally cancel all merged resources at t = 300s. The total split tasks distributed over scenario time have taken only 21 milliseconds to turn back to the original VTSs setting.

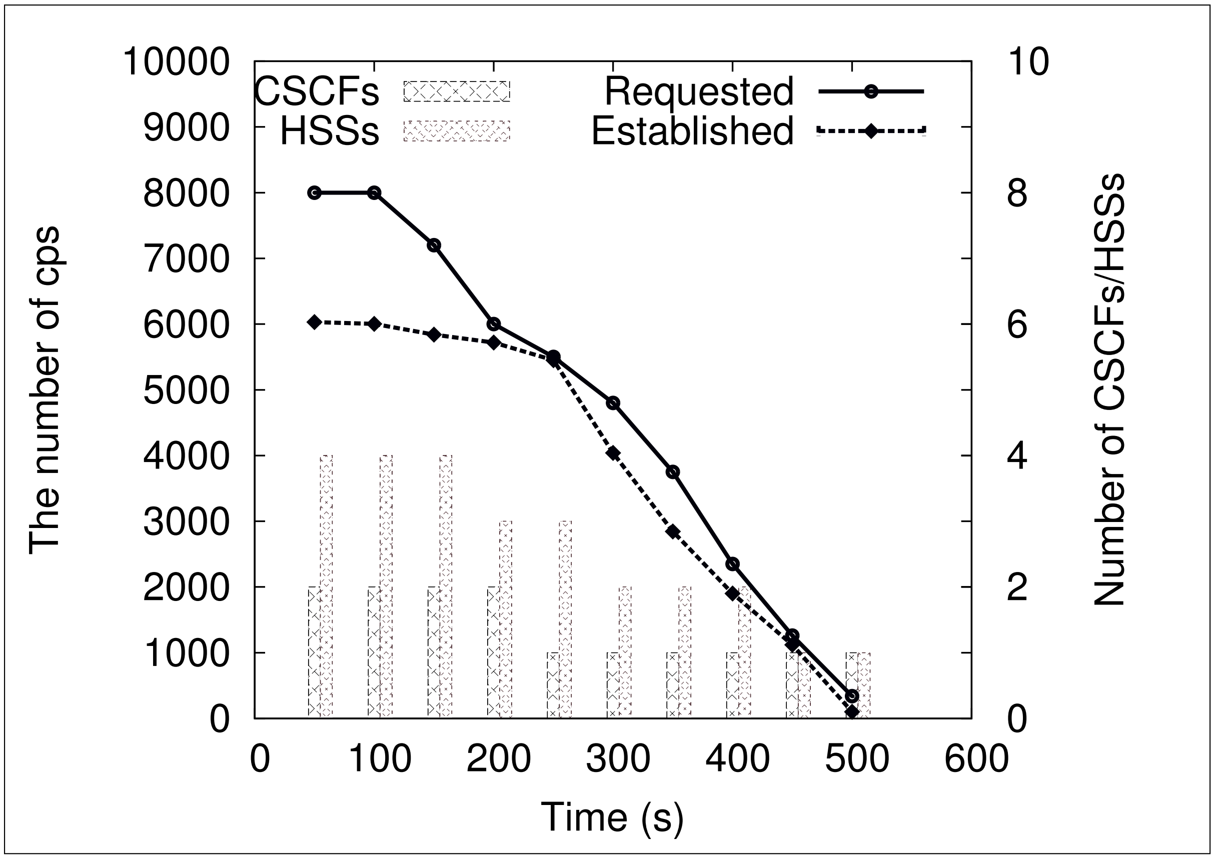

Figure 5.9 Split Tenant VTS

To solve the bottleneck of the merge scenario, we have decided to migrate a third CSCFs, with same configuration like the previous ones, from another tenant network which has only 10 % workload. This CSCFs must still serving originated network subscribers (VTN_2) and at the same time increase the capacity of the joined VTN_1.

To simulate live migration of CSCFs, we ran 15 trials of migration from both sides ($VTN_1 \Longleftrightarrow VTN_2$). We separate the two ways migration test with 10 seconds of break. We consider two scenarios for the migration: (i) the tenant can choose to keep the CSCFs' VM running on the same host and just VTNs' configuration update is required, or (ii) for better performance, tenant requires the migration of the VM to the same VTN's host.

As depicted in Figure 5.10, the average of our migration experiment with the first scenario is 17.08s for the first way and and 16.07s for the second way. We can notice that the bottleneck

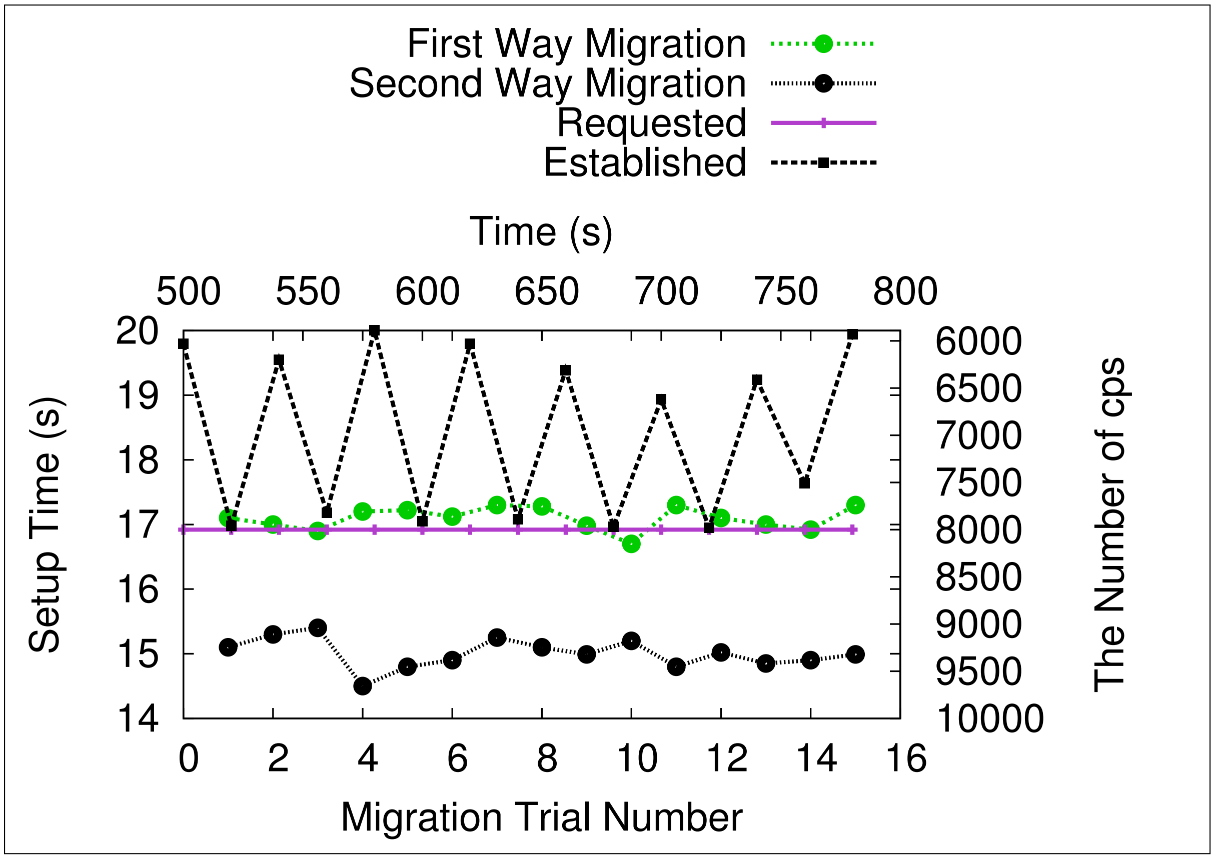

Figure 5.10 Scenario 1: CSCFs Live Migration

is eliminated at each time we migrate CSCFs to VTN_1 and almost subscribers' requests are served.

For the virtual migration scenario (See Figure 5.11), the average of joining the CSCFs to the VTN_1 and finishing all the necessary configuration is 1.45s for the first way and 1,65s for the second way. Using this kind of migration, we did not succeed to serve all calls and eliminate the bottleneck. The maximum established session is 7215 cps. We succeed to provide faster response for the bottleneck without achieving the maximum performance. We believe that this kind of migration can be the first solution for emergency and the live migration can be the next step to overcome the bottleneck and achieve the best performance.

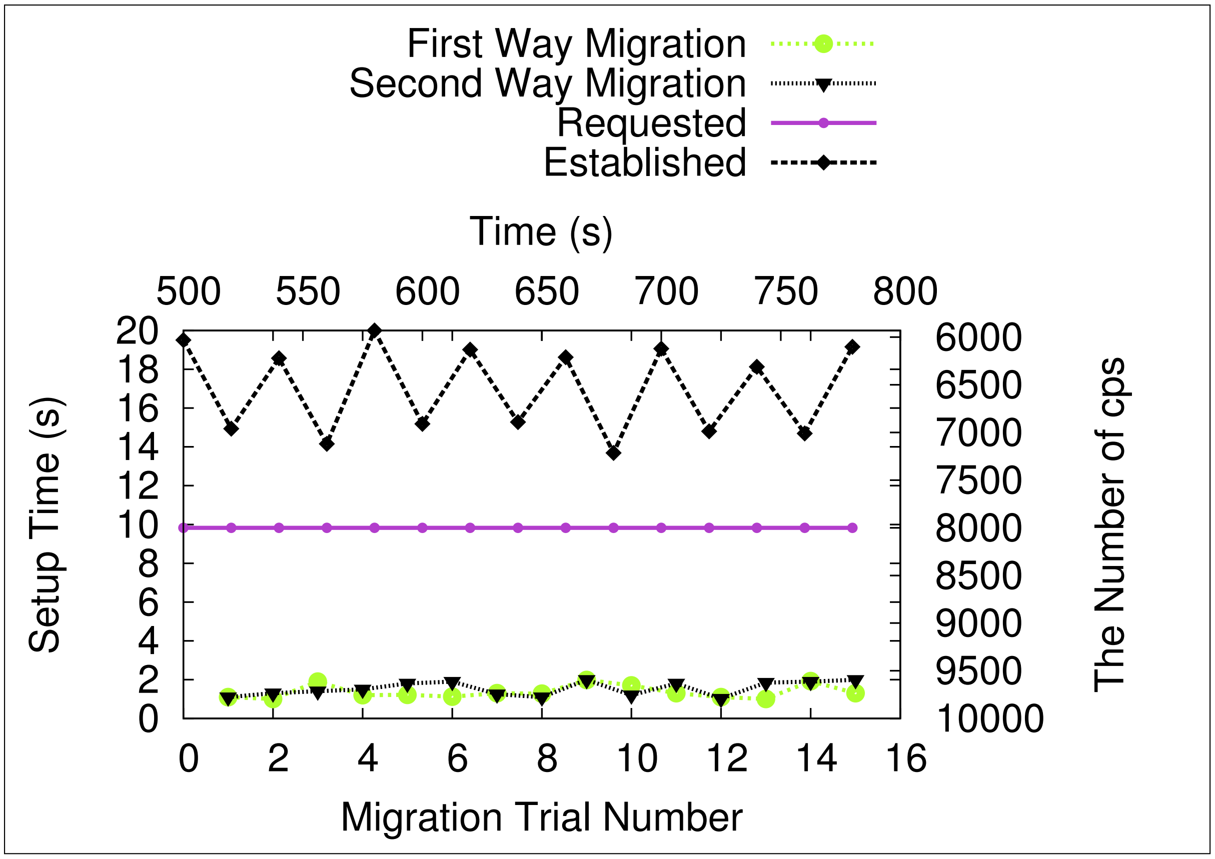

Figure 5.11 Scenario 2: CSCFs Virtual Migration

5.3.7 SDN Controller Scalability

We have already compared our extended OpenFlow controller with basic NOX implementation in the previous experiments. In paragraph 5.3.3, we demonstrated that NOX fails to support large multi-tenant network and drops incoming packets when the number of VMs, generating random rate of traffic, exceeds 1K. We proved that OpenNMS design scales well with fast growing network with negligible overhead and reasonable latency.

With random rate of packets, we do not have the exact capacity of flows handling for the OpenNMS Master centralized controller comparing to the traditional SDN architectures.

In current software-defined network, the centralized OpenFlow controller performs all tenants flow setup reactively. The load on this controller directly affects the flow completion times and the response time.

With OpenNMS design, we succeed to enforce our flexible isolation model and provide in the same manner an efficient and scalable offloading of control functions without losing the SDN centralized view advantage. By delegating VTSs frequent and local packets to tenant controller, we limit the overhead on centralized controller that processes only global and rare events to maintain network-wide view. One master application on the control plane will orchestrate, build and map small tenant distinct portions represented the required VTSs into the virtual and underlay networks.

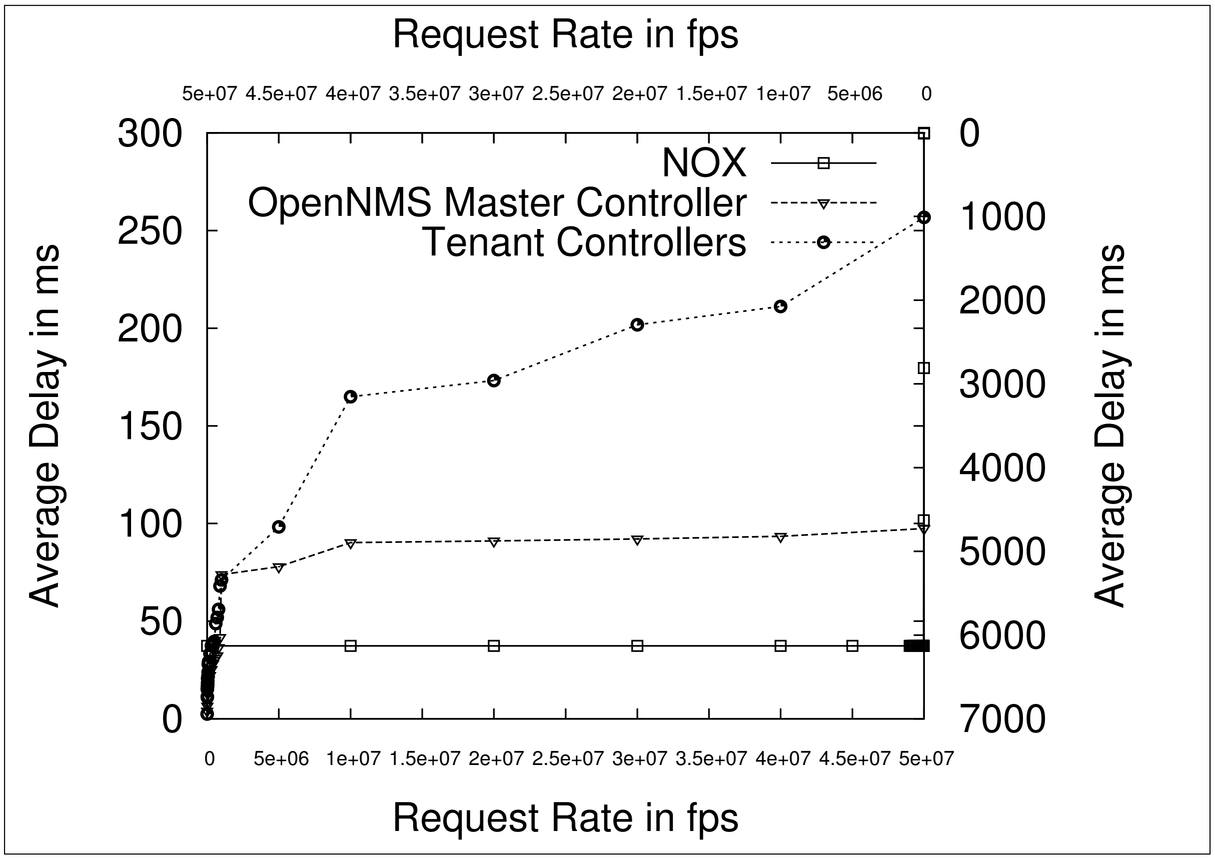

Figure 5.12 Average Delay Under Various Rate

In the following, we evaluate our design to demonstrate its benefits on the SDN controller scalability and overcome the current bottleneck. We ran our experiments to study the relation between the load level and response time.

We use the same testbed in figure 5.1 with one virtual network (VTN_1) to evaluate the performance of both NOX and OpenNMS Master Controller. We ran both NOX and OpenNMS Master Controller on VMs with 6 vcpus and 10 Gbytes memory. We included also the tenant controller in this experiment to demonstrate the repartition of network load between the provider controller and tenant controller. To avoid the bottleneck for tenant control plane, we offload the incoming traffic using a cluster of five controllers. For $tenant_1$'s traffic, we configured the OpenNMS master controller to install flow entries for all accepted packets as permanent so that their lifetime is properly synchronised with the flow entries in the Master tables. For each 100 seconds, we increase the requested rate and we generate different packets' source header for the new added flows. This setup will allow us to compare the Master controller behavior and response time each 10 seconds with higher traffic rate. The master controller will handle only the new packets not matched with the already installed flow entries. We have started our evaluation with 10^3 flow per second (fps) as the maximum load supported by NOX with 20 million of incoming flows, introduced in earlier benchmark (Tavakoli *et al.* (2009)).

Figure 5.12 shows the average delay comparison between NOX and OpenNMS Master controller as well as tenant's controllers. We find out that, under maximum load, NOX can achieve a maximum throughput of 31,000 fps, and the average delay is 6129ms. Because the difference in average delay for the OpenNMS controllers and NOX is very large, we use two X et Y axis. Since the Master controller handles only frequent and global events, the average delay is lower than NOX and tenant's controllers. Only unrecognised packets will be forwarded to the Master controller for being matched and then install the required flow entries. Using our design, we decrease the load on the Master controller and we reach better performance and scalability. For 5 million fps, the difference in average delay between OpenNMS Master controller and the cluster of tenant's controllers is 159.35 ms.

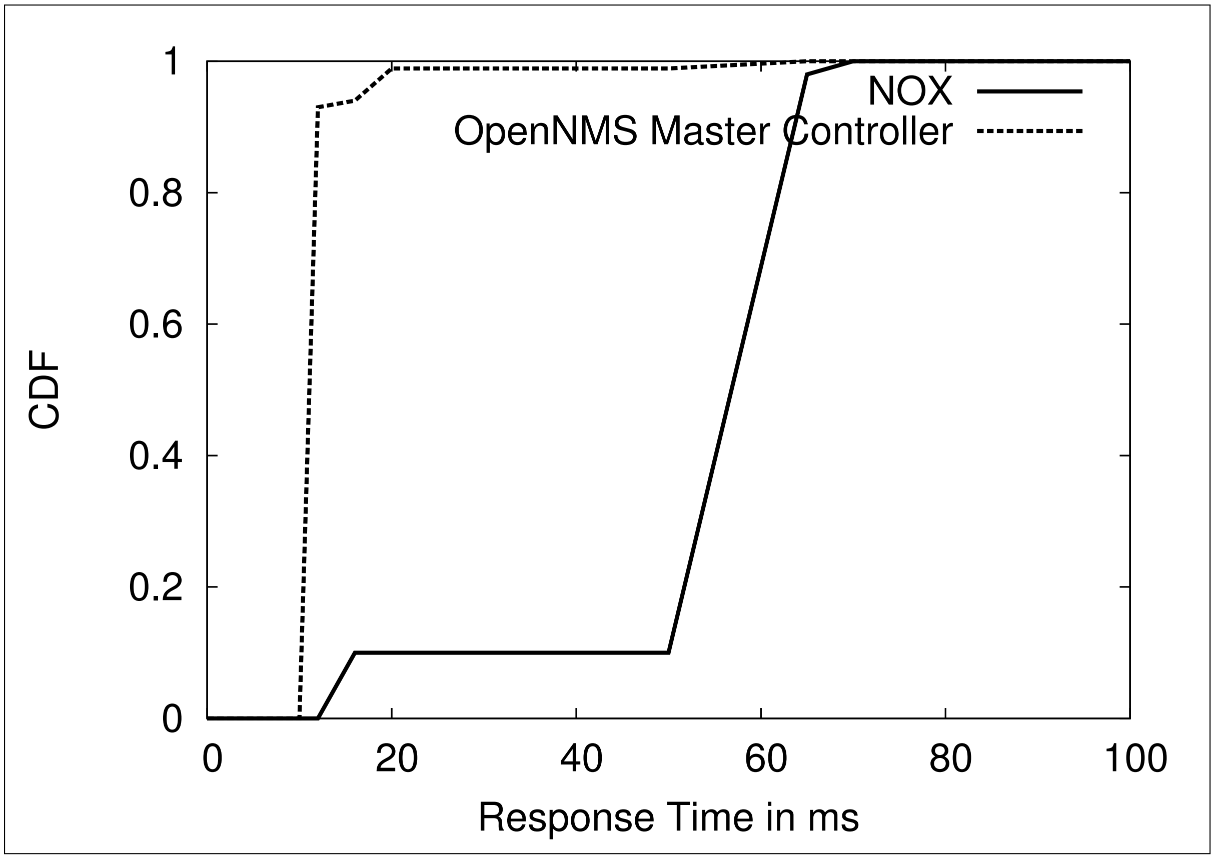

Figure 5.13 Response time CDF Comparaison

To compare and evaluate OpenNMS controller response time and for better understanding of the relation between controller load and response time, we plotted the response time Cumulative Distribution Function (CDF) fixing the load level to 31K fps. The response time is varying with the load level (See Figure 5.13). We find that for the same added workload, OpenNMS controller has lower response time ($\leq$ 20 ms) than NOX with increased load. Our OpenNMS Master controller performs significantly with increased load up to 31K fps better than the basic NOX controller handling all the incoming traffic of multi-tenant network.

5.4 Summary and Conclusions

In this chapter, we built a prototype of our OpenNMS architecture, in order to prove the viability of its autonomic design and the performance of our L2 isolation model. Our prototype improves the management plane in SDN architecture. We used diver and rich tools to realize our design and bring it to real multi-tenant network. Our design firstly prove that multi-tenant isolation is automatically enforced, without any human intervention. We succeed to run multiple and independent VTNs and VTSs while we support full transparency for cloud tenant and enable the self-configuration and self-management attributes of the autonomic computing model.

We ran exhaustive experiments and imagined several scenarios so as to cover all our objectives, our isolation features and situations that we believe are relevant to include in the evaluation. Our overhead evaluation demonstrate that our L2 isolation model using OpenNMS Master controller guarantees negligible packets drops with the minimal MTU comparing to unmodified NOX controller. OpenNMS Agent extending the NOX adds at most only microseconds to the end-to-end latency compared to NOX. The second evaluation for our slicing types shows that there is no difference between them with flow entries setup time. We add at maximum 90 μs to the flow latency which does not affect the network performance. Next, we did more experimental analysis for the OpenNMS isolation model and architecture design for evaluating its scalability and elasticity in multi-tenant network. The results proved that OpenNMS scales well, achieving negligible overhead, especially as the number of VTS and VMs increases.

We choose for the elasticity experiment the OpenIMS system as greedy cloud application which requires flexible and elastic environment to overcome the unplanned load. We ran different and rich experiment to present the benefits of our SMM capabilities with such system. OpenIMS networks and slices can split, merge and migrate easily to satisfy the subscribers demand as well as the tenant needs. OpenNMS response time for the IMS components bottleneck is a simple and effective metric for the merging capability. The total split tasks have taken only 21 milliseconds to turn back to the original VTSs setting. When a OpenIMS nodes migration, our third experiment demonstrates that the security is still applied after the migration. We succeed to provide faster response for the bottleneck without achieving a fair performance with the virtual migration approach. Indeed, the paths are reconfigured as the VM starts emitting packets again. The enforcement of the security policy can be realized either with live migration of virtual one.

The last tests we implemented for the OpenNMS master controller compared with current SDN architecture prove that our design overcome the SDN scalability bottleneck very well.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this chapter, we provide a summary of the contributions of this dissertation and a few concluding remarks, and finally give some important directions for future research beyond this work.

6.1 Conclusions

This work is accomplished by three conference papers (Fekih Ahmed *et al.*, 2013a, 2014, 2015) and one filed patent (Fekih Ahmed *et al.*, 2013b) with the collaboration of Ericsson Research Canada.

In this last chapter, we are reviewing all the contributions that has been done since the definition of the problematic of our project : "Towards Elastic and Scalable Multi-tenant Network".

Enforcing isolation in multi-tenant network is already tackled by previous industrial and research approaches, but the nature of cloud brings on new requirements that we must meet. We started our work with the review of the literature. Some researches have addressed the requirements for flexible and scalable isolation by developing several solutions using Software-Defined Networking (SDN) based slicing technique. This technique enforces basic and strict isolation proprieties by translating, inspecting, rewriting, and policing OpenFlow entries installed on shared data plane. SDN slices suffer from scalability limitations. Industrial isolation approaches are ideally based to provide the transport by the physical network and the virtual machine service by the hypervisors. The traditional slicing technique has employed VLAN to isolate tenants' machines on a single Layer 2 (L2) virtual network. However, this simple isolation approach depends heavily on routing and forwarding protocols and is not easily configured. VLAN management complexity imposes limitations on cloud nature and services. More importantly, VLAN lacks scalability resulting to a segmentation capacity limit to 4K tenants. Multiple overlay technologies (e.g., VXLAN, NVGRE) have been proposed within industrials as alternative approaches to substitute old encapsulation protocols that are in contrast with the open standards used in OpenFlow solutions. These techniques overcome scalability issues of cloud isolation mechanisms, but remain limited to single slice. In addition, both solutions do not offer a complete transparent isolation and lack the ability to use multiple network slices within individual SDN programs. However, no solution succeeds to provide the isolation goals motivating this project. We proved that we overcome their respective limitations into a unified design and arrive to a consistent solution that consider both networks and OpenFlow control plane scalability bottleneck. This combination enables to reap the benefits of SDN slices while preserving scalability. This yields a network virtualization architecture that is both flexible and secure on one side, and scalable on the other.

We here summarize how we tackled each of our isolation goals: Flexible, Scalable and Autonomic.

First, in chapter 3, we have introduced the benefits of SDN slicing technique and the actual opportunities with such approach, where the multi-tenant flexible, scalable isolation can be achieved by providing a solid L2 model and combining it with the overlay technologies. We defined our L2 isolation model providing high level abstraction for multi-tenant network by combining SDN slicing technique and overlay virtual protocols. Our model enhance the SDN slicing technique flexibility while preserve its advantages for providing the basic isolation objectives. We take benefits of the scalability of overlay protocols and overcome their limits to one overlay network by enabling to support several thousands of isolated tenants networks and slices on top of shared network infrastructure. Using these extended encapsulations mechanisms, the number of segmentation is increased and as result it offers the possibility of creating more number of Virtual Tenant Slices (VTSs). Our isolation model formalizes the simultaneous use of multiple resources (ports, tables, group tables and controllers) allocation for multi-tenant network. This allows service providers to define multiple processing spaces dedicated for individualized network services. From our definitions and proprieties, the VTS model extends the network virtualization with new self-ruling virtual networks that can be programmed in a standalone mode just like an ordinary OpenFlow switch, without worrying about other Open-

Flow applications running on the top of any other VTSs. It will serve a strong specification for the infrastructure providers to translate and map it to the underlay network while ensuring isolation properties. Further, our VTS types show that we can leverage almost the full optimization of allocated resources to meet differing tenant application objectives. Even if tenant switches between VTS types for higher sharing or restriction level, the other slices belonging to the same or different tenants will stay intact. However, some penalties are analysed with the tenant choice such as the added overhead and the impact on confidentiality, integrity and availability.

Building an elastic and scalable multi-tenant network for datacenters is very challenging. This thesis contributes to solve this challenge in three ways. First, we define a L2 isolation model with high-level abstraction providing the required flexibility for both tenant and provider for such elastic cloud nature. Secondly, we build a framework Open Network Management & Security based on the combination between SDN paradigm and autonomic communication. Autonomic computing is a complementary approach to SDN for evolving the neglected management plane and self-aware network configuration. It allows an embedded management of all VTSs and gradual implementation of management functions providing code life cycle management for multi-tenant applications as well as the ability to on-the-fly configuration update. The Self-* capabilities in a SDN network can accomplish the centralized controller functions by recommending an appropriate action based on the overall network policies and tenant requirement. These capabilities are included in OpenNMS through control loops in the management plane. Our design adds the necessary components to achieve our ultimate goals for scalability, flexibility and automaticity. The OpenNMS control and management planes are created in a way to provide transparency and enabling the self-management and self-controlling of the allocated OpenFlow resources (Ports, Tables and Group Tables). Lastly, we perform a general scalability study of the SDN architecture design space and we demonstrate the benefits of our isolation model and design for solving such controller scalability bottleneck. Using hierarchical OpenFlow controllers, we succeed to enforce our flexible isolation model and provide in the same manner an efficient and scalable offloading of control functions without losing the

SDN centralized advantage. By delegating Virtual Tenant Slices (VTSs) frequent and local packets to tenant controller, we limit the overhead on centralized controller that processes only global and rare events to maintain network-wide view. One master application on the control plane will orchestrate, build and map small tenant distinct portions represented the required VTSs into the virtual and underlay networks. VTS space on data plane will be allocated for each tenant for building his own small security boxes out of multiple, independent, reusable network policies included in the VTS abstraction. Thus, as results of the simple and flexible VTS definitions model, we describe novel features and capabilities added for the isolation: Split, Merge and Migrate that can be well suited for the tenants' requirements in such dynamic nature of cloud computing.

Finally, the OpenNMS design and evaluation chapters show that, as the network diameter increases, the performance penalty for fast growing network decreases. Our approach demonstrates the simplicity and the feasibility of Software-Defined security mechanisms. The flexibility gained through this approach helps to adapt the network dynamically to both unforeseen and predictable changes in the network. It offers the possibility to run multiple slices within the same logical switch without performance degradation.

6.2 Future work

The Cloud Computing field is in its infancy and rapidly evolving, from an operations engineering perspective in general, and a network interconnect design in particular. Even with the deluge of data center networking research and innovation by academia, small start-ups, and big industry players in the past decade, we feel that the community has barely scratched the surface, and that there is an almost infinite amount of work to be done. We here mention some exciting related on-going research and a few open problems.

In our model evaluation, with 12 servers, we have reached the limits to the size of an academic testbed, and the prototypes presented here would be benefit from further evaluation at a larger scale. Further, while we have attempted to measure our prototypes under reasonable assumptions, the research community desperately needs additional data and guidance from industry regarding tenants, workload, and network traffic characteristics. Network virtualization is an important problem in industry, and establishing a robust solution is a necessary step for the cloud computing model to meet its full potential. Several commercial systems are currently being developed, and though the final product may not look like OpenNMS, we believe the central ideas introduced in this thesis have strong advantages over other approaches and could be integrated into a commercial network virtualization solution. Future work will be an extension for our work to demonstrate the simplicity of VTSs scalability which requires more extensive testing that the experiment reflected in this thesis. Additional testing of tenant's VMs and VTSs migration is in fact part of challenging future work.

APPENDIX I

OPENNMS AGENT: OPENFLOW APPLICATION EXTENDING NOX 1.3 DEPLOYED IN C++

```
/*
 * Use OpenNMS Agent application as part of NOX 1.3 version to deploy
    Scalable, Flexible and Autonomic Isolation For Multi-Tenant Network.
 *
 * For More Information: https://github.com/CPqD/ofsoftswitch13/wiki/
    OpenFlow-1.3-Tutorial
 *
 * NOX is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published
 * by the Free Software Foundation, either version 3 of the License,
 * or (at your option) any later version.
 *
 * NOX is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with NOX.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <boost/bind.hpp>
#include <boost/foreach.hpp>
#include <boost/shared_array.hpp>
#include <sys/time.h>
#include <cstring>
#include <netinet/in.h>
#include <stdexcept>
#include <stdint.h>
#include "openflow-default.hh"
#include "assert.hh"
```

```
#include "component.hh"
#include "flow.hh"
#include "fnv_hash.hh"
#include "hash_set.hh"
#include "ofp-msg-event.hh"
#include "vlog.hh"
#include "flowmod.hh"
#include "datapath-join.hh"
#include <stdio.h>
#include <stdio.h>
#include <time.h>
#include <timeval.hh>
#include <iostream>
#include "netinet++/ethernetaddr.hh"
#include "netinet++/ipaddr.hh"
#include "netinet++/ethernet.hh"
#include "../../../oflib/ofl-actions.h"
#include "../../../oflib/ofl-messages.h"
using namespace vigil;
using namespace vigil::container;
using namespace std;
namespace {
struct Mac_source
{
    /* Key. */
    datapathid datapath_id;     /* Physical or OpenFlow Virtual Switch
        Identifier*/
    ethernetaddr mac;           /* Flow Source MAC. */
    ipaddr ip;                  /* Flow Source IP. */
    mutable int port;           /* Port where packets from 'mac' were seen.
         */
    Mac_source() : port(-1) { }
    Mac_source(datapathid datapath_id_, ethernetaddr mac_)
        : datapath_id(datapath_id_), mac(mac_), port(-1)
        { }
```

```
};

bool operator==(const Mac_source& a, const Mac_source& b)
{
    return a.datapath_id == b.datapath_id && a.mac == b.mac;
}

bool operator!=(const Mac_source& a, const Mac_source& b)
{
    return !(a == b);
}

struct Hash_mac_source
{
    std::size_t operator()(const Mac_source& val) const {
        uint32_t x;
        x = vigil::fnv_hash(&val.datapath_id, sizeof val.datapath_id);
        x = vigil::fnv_hash(val.mac.octet, sizeof val.mac.octet, x);
        return x;
    }
};

Vlog_module log("ONMS-Agent");

class ONMS
    : public Component
{
public:
    ONMS(const Context* c,
           const json_object*)
        : Component(c) { }

    void configure(const Configuration*);

    void install();

    Disposition handle(const Event&);
    Disposition handle_dp_join(const Event& e);
private:
    typedef hash_set<Mac_source, Hash_mac_source> Source_table;
    Source_table sources;

    /* Set up a flow when we know the destination of a packet?  This should
     * ordinarily be true; it is only usefully false for debugging purposes
         . */
    bool setup_flows;
};

void
ONMS::configure(const Configuration* conf) {
    setup_flows = true; // Boolean setting default value true
    BOOST_FOREACH (const std::string& arg, conf->get_arguments()) {
        if (arg == "noflow") {
            setup_flows = false;
        } else {
            VLOG_WARN(log, "argument \"%s\" not supported", arg.c_str());
        }
    }
    register_handler(Datapath_join_event::static_get_name(), boost::bind(&
       ONMS::handle_dp_join, this, _1));
    register_handler(Ofp_msg_event::get_name(OFPT_PACKET_IN), boost::bind(&
       ONMS::handle, this, _1));
}

void
ONMS::install() {
}

Disposition
ONMS::handle_dp_join(const Event& e){
```

```
const Datapath_join_event& dpj = assert_cast<const Datapath_join_event&>(
    e);
/*
* Install Flow entries in Master Table (Table 0) in each OpenFlow switch
    to route incoming packets basing on their identification to the
    related VTS.
*/
  if (dpj.dpid.as_host() == ? || dpj.dpid.as_host() == 5) {
  for (int tab=1; tab<N; tab++) { // N is the length of table containing
     VTS list.
  for (int slicenbrport=1; slicenbrport<Q; slicenbrport++) { // Q is the
     length of VTS set.
  Flow f;
  if (dpj.dpid.as_host() == ? && tab == ? && slicenbrport == ?) {f.
     Add_Field("in_port", ?);} // Repeat this for all VTS.
  Actions *acts = new Actions();
  Instruction *inst =  new Instruction();
  inst->CreateApply(acts);
  inst->CreateGoToTable(tab);
  FlowMod *mod = new FlowMod(0x00ULL,0x00ULL, 0,OFPFC_ADD,
     OFP_FLOW_PERMANENT, OFP_FLOW_PERMANENT, 0, 0,
                                OFPP_ANY, OFPG_ANY, ofd_flow_mod_flags());
  mod->AddMatch(&f.match);
  mod->AddInstructions(inst);
  send_openflow_msg(dpj.dpid, (struct ofl_msg_header *)&mod->fm_msg, 0/*
     xid*/, true/*block*/);
  VLOG_DBG(log,"Installing default flow with default priority  to send
     packets to the Dedicated ONMS Slices Tables on dpid= 0x%"PRIx64"\n"
     , dpj.dpid.as_host());
  }
  }
  }
  /* The behavior on a flow miss is to drop packets
     so we need to install a default flow */
  VLOG_DBG(log,"Installing default flow with priority 0 to send packets
     to the controller on dpid= 0x%"PRIx64"\n", dpj.dpid.as_host());
  if (dpj.dpid.as_host() == ? || dpj.dpid.as_host() == ? || dpj.dpid.
     as_host() == ?) {
     Flow  *f = new Flow();
  Actions *acts = new Actions();
  acts->CreateOutput(OFPP_CONTROLLER); //add output action to send flow
     to OFPP_Controller port: 6633
  Instruction *inst =  new Instruction();
  inst->CreateApply(acts);
  FlowMod *mod = new FlowMod(0x00ULL,0x00ULL, 0,OFPFC_ADD,
     OFP_FLOW_PERMANENT, OFP_FLOW_PERMANENT, 0, 0,
                                OFPP_ANY, OFPG_ANY, ofd_flow_mod_flags());
  mod->AddMatch(&f->match);
  mod->AddInstructions(inst);
  send_openflow_msg(dpj.dpid, (struct ofl_msg_header *)&mod->fm_msg, 0/*
     xid*/, true/*block*/);
  }
  else {
  for (int i=1; i<?; i++) {
  Flow  *f = new Flow();
  Actions *acts = new Actions();
  acts->CreateOutput(OFPP_CONTROLLER); // Add output action to send flow
     to OFPP_Controller port: 6633
  Instruction *inst =  new Instruction();
  inst->CreateApply(acts);
  FlowMod *mod = new FlowMod(0x00ULL,0x00ULL, i,OFPFC_ADD,
     OFP_FLOW_PERMANENT, OFP_FLOW_PERMANENT, 0, 0,
                                OFPP_ANY, OFPG_ANY, ofd_flow_mod_flags());
  mod->AddMatch(&f->match);
  mod->AddInstructions(inst);
  send_openflow_msg(dpj.dpid, (struct ofl_msg_header *)&mod->fm_msg, 0/*
     xid*/, true/*block*/);
 }}
  return CONTINUE;
```

```
}

Disposition
ONMS::handle(const Event& e)
{
    const Ofp_msg_event& pi = assert_cast<const Ofp_msg_event&>(e);
    struct ofl_msg_packet_in *in = (struct ofl_msg_packet_in *)**pi.msg;
    Flow *flow = new Flow((struct ofl_match*) in->match);

    /* drop all LLDP packets */
        uint16_t dl_type;
        flow->get_Field<uint16_t>("eth_type",&dl_type);
        if (dl_type == ethernet::LLDP){
            return CONTINUE;
        }

    uint32_t in_port;
    flow->get_Field<uint32_t>("in_port", &in_port);

    /* Learn the source. */
    uint8_t eth_src[6];
    flow->get_Field("eth_src", eth_src);
    ethernetaddr dl_src(eth_src);

    if (!dl_src.is_multicast()) {
        Mac_source src(pi.dpid, dl_src);
        Source_table::iterator i = sources.insert(src).first;
        if (i->port != in_port) {
            i->port = in_port;
            VLOG_DBG(log, "learned that @mac-src "EA_FMT" is on datapath %s
                port %d",
                         EA_ARGS(&dl_src), pi.dpid.string().c_str(),
                         (int) in_port);
        }
    } else {
        VLOG_DBG(log, "multicast packet source "EA_FMT, EA_ARGS(&dl_src));
    }

    /* Figure out the destination. */
    int out_port = -1;        /* Flood by default. */
    uint16_t vlan_vid = -1;
    uint32_t ip_src;
    uint32_t ip_dst;
    uint8_t eth_dst[6];
    flow->get_Field("eth_dst", eth_dst);
    ethernetaddr dl_dst(eth_dst);
    timeval tv;
    gettimeofday(&tv, NULL);

    timespec time;
    time.tv_sec = tv.tv_sec;
    time.tv_nsec = tv.tv_usec * 1000;
    double t1=(time.tv_sec * 1000000000.0) + time.tv_nsec;
    if (!dl_dst.is_multicast()) {
    Mac_source dst(pi.dpid, dl_dst);
        Source_table::iterator i(sources.find(dst));
    if (i != sources.end()) {
    out_port = i->port;
    flow->get_Field<uint32_t>("ipv4_src", &ip_src);
    flow->get_Field<uint32_t>("ipv4_dst", &ip_dst);
    flow->get_Field<uint16_t>("vlan_id", &vlan_vid);
    VLOG_DBG(log, "No matching Flow sent to Controller: @mac-src "EA_FMT" -
         @ipsrc "IP_FMT" - @ipdst "IP_FMT" - datapath ID %s - inport %d -
       outport %d - vlan %d -eth_type 0x\"%x\"",
                    EA_ARGS(&dl_src), IP_ARGS(&ip_src), IP_ARGS(&ip_dst),
                       pi.dpid.string().c_str(),
                    (int) in_port, (int) out_port , (int) vlan_vid,
                       dl_type);
        }
    }
    /* Set up a flow if the output port is known. */
    if (setup_flows && out_port != -1) {
        Flow  f;
        f.Add_Field("in_port", in_port);
        f.Add_Field("eth_src", eth_src);
        f.Add_Field("eth_dst",eth_dst);
        Actions *acts = new Actions();
    Instruction *inst =  new Instruction();
    acts->CreateOutput(out_port);
    inst->CreateApply(acts);
    FlowMod *mod = new FlowMod(0x00ULL,0x00ULL, 1,OFPFC_ADD,
       OFP_FLOW_PERMANENT, OFP_FLOW_PERMANENT, OFP_DEFAULT_PRIORITY,in->
       buffer_id,
                                     OFPP_ANY, OFPG_ANY, ofd_flow_mod_flags
                                         ());
    mod->AddMatch(&f.match);
        mod->AddInstructions(inst);
    send_openflow_msg(pi.dpid, (struct ofl_msg_header *)&mod->fm_msg, 0/*
       xid*/, true/*block*/);
    gettimeofday(&tv, NULL);
    time.tv_sec = tv.tv_sec;
    time.tv_nsec = tv.tv_usec * 1000;
    double t2=(time.tv_sec * 1000000000.0) + time.tv_nsec;
     VLOG_DBG(log, "Set Flow (Time to push: %.1lf nanoseconds ): @mac-src "
        EA_FMT" to @mac-dst "EA_FMT" is on datapath %s inport %d and
        outport %d",
                    t2-t1, EA_ARGS(&dl_src), EA_ARGS(&dl_dst), pi.dpid.
                       string().c_str(),
                     (int) in_port, (int) out_port);
    }
    /* Send out packet if necessary. */
    if (!setup_flows || out_port == -1 || in->buffer_id == UINT32_MAX) {
        if (in->buffer_id == UINT32_MAX) {
            if (in->total_len != in->data_length) {
                /* Control path didn't buffer the packet and didn't send us
```

```
                 * the whole thing--what gives? */
                VLOG_DBG(log, "total_len=%"PRIu16" data_len=%zu\n",
                        in->total_len, in->data_length);
                return CONTINUE;
            }
            send_openflow_pkt(pi.dpid, Nonowning_buffer(in->data, in->
               data_length), in_port, out_port == -1 ? OFPP_FLOOD :
               out_port, true/*block*/);
        } else {
            send_openflow_pkt(pi.dpid, in->buffer_id, in_port, out_port ==
               -1 ? OFPP_FLOOD : out_port, true/*block*/);
        }
    }
    return CONTINUE;
}
REGISTER_COMPONENT(container::Simple_component_factory<ONMS>, ONMS);
}
```

BIBLIOGRAPHY

Al-Fares, Mohammad, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. 2010. "Hedera: Dynamic Flow Scheduling for Data Center Networks.". In *NSDI*. p. 19–19.

Anderson, J Chris, Jan Lehnardt, and Noah Slater, 2010. *CouchDB: the definitive guide*.

Anwer, Bilal, Theophilus Benson, Nick Feamster, Dave Levin, and Jennifer Rexford. 2013. "A slick control plane for network middleboxes". In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. p. 147–148. ACM.

Arora, Dushyant and Diego Perez-Botero. "Live Migration of an Entire Software-Defined Network".

Benson, Theophilus, Aditya Akella, Anees Shaikh, and Sambit Sahu. 2011. "CloudNaaS: a cloud networking platform for enterprise applications". In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. p. 8. ACM.

Brief, ONF Solution. 2014. "OpenFlow-enabled SDN and Network Functions Virtualization".

Cabuk, Serdar, Chris I Dalton, Konrad Eriksson, Dirk Kuhlmann, Harigovind V Ramasamy, Gianluca Ramunno, Ahmad-Reza Sadeghi, Matthias Schunter, and Christian Stüble. 2010. "Towards automated security policy enforcement in multi-tenant virtual data centers". *Journal of Computer Security*.

Cannistra, Robert, Benjamin Carle, Matt Johnson, Junaid Kapadia, Zach Meath, Mary Miller, Devin Young, Casimer M DeCusatis, Todd Bundy, Gil Zussman, et al. 2014. "Enabling autonomic provisioning in SDN cloud networks with NFV service chaining". In *Optical Fiber Communication Conference*. p. Tu2I–4. Optical Society of America.

Casado, Martin, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. 2007. "Ethane: Taking control of the enterprise". *ACM SIGCOMM Computer Communication Review*, vol. 37, n° 4, p. 1–12.

Cisco, ACE. 2008a. "Web Application Firewall".

Cisco, IOS. 2008b. "NetFlow".

Clear, David, Sudhir Cheruathur, and Guy Erb. 1 2002. "Vlan tunneling protocol". US Patent 20,020,101,868.

Correia, Luis M, Henrik Abramowicz, and Martin Johnsson, 2011. *Architecture and design for the future internet: 4WARD project*.

Cremet, Vincent and Martin Odersky. *Foundations for scala*. Technical report.

Curtis, Andrew R, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. 2011. "Devoflow: scaling flow management for high-performance networks". In *ACM SIGCOMM Computer Communication Review*. p. 254–265. ACM.

de Oliveira, Rogerio Leao Santos, Christiane Marie Schweitzer, Ailton Akira Shinoda, and Ligia Rodrigues Prete. 2014. "Using mininet for emulation and prototyping software-defined networks". In *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on*. p. 1–6. IEEE.

Devlic, Alisa, Wolfgang John, and Pontus Sköldström. 2012a. "Carrier-grade Network Management Extensions to the SDN Framework". In *8th Swedish National Computer Networking Workshop SNCNW 2012 Stockholm*.

Devlic, Alisa, Wolfgang John, and Pontus Skoldstrom. 2012b. "A use-case based analysis of network management functions in the ONF SDN model". In *Software Defined Networking (EWSDN), 2012 European Workshop on*. p. 85–90. IEEE.

Dixit, Advait, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Kompella. 2013. "Towards an elastic distributed sdn controller". In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. p. 7–12. ACM.

Doria, Avri, R Haas, J Hadi Salim, H Khosravi, and WM Wang. 2007. "ForCES protocol specification". *EB/OL]. Dec.*

Drutskoy, Dmitry, Eric Keller, and Jennifer Rexford. 2013. "Scalable network virtualization in software-defined networks". *Internet Computing, IEEE*, vol. 17, n° 2, p. 20–27.

Enns, R, M Bjorklund, J Schoenwaelder, and A Bierman. 2011. "Network configuration protocol (NETCONF)". *Internet Engineering Task Force, RFC*, vol. 6241.

Erickson, D et al. 2012. "Floodlight Java based OpenFlow Controller". *Last accessed, Ago*.

Erickson, David et al. 2013. "The beacon openflow controller". In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. p. 13–18. ACM.

Farinacci, Dino, P Traina, Stan Hanks, and T Li. 1994. "Generic routing encapsulation (GRE)".

Fayazbakhsh, Seyed Kaveh, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. 2013. "FlowTags: enforcing network-wide policies in the presence of dynamic middlebox actions". In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. p. 19–24. ACM.

Feit, Sidnie M, 1993. *SNMP: A guide to network management*.

Fekih Ahmed, Mohamed, Chamssedine Talhi, Makan Pourzandi, and Mohamed Cheriet. 2013a. "Virtual Data Center Scalability using Open Flow Controller". In *The 2nd International Conference on Software Engineering and New Technologies (ICSENT 2013)*.

Fekih Ahmed, Mohamed, Chamssedine Talhi, Makan Pourzandi, and Mohamed Cheriet. 2013b. "Multi-tenant Isolation In A Cloud Environment Using Software Defined Networking".

Fekih Ahmed, Mohamed, Chamssedine Talhi, Makan Pourzandi, and Mohamed Cheriet. 2014. "A Software-Defined Scalable and Autonomous Architecture for Multi-tenancy". In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. p. 568–573. IEEE.

Fekih Ahmed, Mohamed, Chamssedine Talhi, and Mohamed Cheriet. 2015. "Towards Flexible, Scalable and Autonomic Virtual Tenant Slices". In *The 14th IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2015)*.

Fernandes, Eder Leao et al. 2012. "nox13:oflib". https://github.com/CPqD/nox13:oflib. Visited on 7 December 2013.

Figuerola, Sergi, Mathieu Lemay, Victor Reijs, Michel Savoie, and Bill St Arnaud. 2009. "Converged optical network infrastructures in support of future internet and grid services using IaaS to reduce GHG emissions". *Journal of Lightwave Technology*, vol. 27, n° 12, p. 1941–1946.

Foster, Nate, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. "Frenetic: A network programming language". In *ACM SIGPLAN Notices*. p. 279–291. ACM.

FP7-4WARD. 2008. "FP7 4WARD Project". Online. <http://www.4ward-project.eu/>.

Gayraud, R, O Jacques, and CP Wright. 2007. "SIPp: traffic generator for the SIP protocol".

Gember, Aaron, Anand Krishnamurthy, Saul St John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Vyas Sekar, and Aditya Akella. 2013. "Stratos: A Network-Aware Orchestration Layer for Virtual Middleboxes in Clouds". *arXiv preprint arXiv:1305.0209*.

Greenberg, Albert, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. 2009. "VL2: a scalable and flexible data center network". In *ACM SIGCOMM Computer Communication Review*. p. 51–62. ACM.

Greenhalgh, Adam, Felipe Huici, Mickael Hoerdt, Panagiotis Papadimitriou, Mark Handley, and Laurent Mathy. 2009. "Flow processing and the rise of commodity network hardware". *ACM SIGCOMM Computer Communication Review*, vol. 39, n° 2, p. 20–26.

Gude, Natasha, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. 2008. "NOX: towards an operating system for networks". *ACM SIGCOMM Computer Communication Review*, vol. 38, n° 3, p. 105–110.

Gutz, Stephen, Alec Story, Cole Schlesinger, and Nate Foster. 2012. "Splendid isolation: A slice abstraction for software-defined networks". In *Proceedings of the first workshop on Hot topics in software defined networks*. p. 79–84. ACM.

Habib, Irfan. 2008. "Virtualization with kvm". *Linux Journal*, vol. 2008, n° 166, p. 8.

Hammel, Michael J. 2011. "Managing KVM deployments with Virt-Manager". *Linux Journal*, vol. 2011, n° 201, p. 7.

Hassas Yeganeh, Soheil and Yashar Ganjali. 2012. "Kandoo: a framework for efficient and scalable offloading of control applications". In *Proceedings of the first workshop on Hot topics in software defined networks*. p. 19–24. ACM.

Khurshid, Ahmed, Wenxuan Zhou, Matthew Caesar, and P Godfrey. 2012. "Veriflow: Verifying network-wide invariants in real time". *ACM SIGCOMM Computer Communication Review*, vol. 42, n° 4, p. 467–472.

Kim, Sungsu. 2013. "Cognitive Model-Based Autonomic Fault Management in SDN". PhD thesis, Ph. D. thesis, Pohang University of Science and Technology.

Kis, Zoltan Lajos. 2012. "OpenFlow 1.1". https://github.com/TrafficLab. Visited on 20 December 2013.

Koponen, Teemu, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. 2010. "Onix: A Distributed Control Platform for Large-scale Production Networks.". In *OSDI*. p. 1–6.

Krieger, Orran, Phil McGachey, and Arkady Kanevsky. 2010. "Enabling a marketplace of clouds: VMware's vCloud director". *ACM SIGOPS Operating Systems Review*, vol. 44, n° 4, p. 103–114.

Krishnamurthy, Anand, Shoban P Chandrabose, and Aaron Gember-Jacobson. 2014. "Pratyaastha: an efficient elastic distributed SDN control plane". In *Proceedings of the third workshop on Hot topics in software defined networking*. p. 133–138. ACM.

Lantz, Bob, Brandon Heller, and Nick McKeown. 2010. "A network in a laptop: rapid prototyping for software-defined networks". In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. p. 19. ACM.

Lara, Adrian, Anisha Kolasani, and Byrav Ramamurthy. 2013. "Network innovation using openflow: A survey".

Li, Hongyun, Xirong Que, Yannan Hu, Gong Xiangyang, and Wang Wendong. 2013. "An autonomic management architecture for SDN-based multi-service network". In *Globecom Workshops (GC Wkshps), 2013 IEEE*. p. 830–835. IEEE.

Linux Foundation. 2013. "OpenDaylight project, Linux Foundation". http://www.opendaylight.org. Visited on 10 June 2013.

Mahalingam, Mallik, D Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreeger, T Sridhar, Mike Bursell, and Chris Wright. 2012. "VXLAN: A framework for overlaying virtualized layer 2 networks over layer 3 networks". *draftmahalingam-dutt-dcops-vxlan-01.txt*.

Mattos, Diogo MF, Natalia C Fernandes, Victor T Da Costa, Leonardo P Cardoso, Miguel Elias M Campista, Luıs Henrique MK Costa, and Otto Carlos MB Duarte. 2011. "Omni: Openflow management infrastructure". In *Network of the Future (NOF), 2011 International Conference on the*. p. 52–56. IEEE.

Mccauley, J et al. "Pox: A python-based openflow controller".

McKeown, Nick. 2009. "Software-defined networking". *INFOCOM keynote talk*.

McKeown, Nick, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. "OpenFlow: enabling innovation in campus networks". *ACM SIGCOMM Computer Communication Review*, vol. 38, n° 2, p. 69–74.

Menon, Aravind, Alan L Cox, and Willy Zwaenepoel. 2006. "Optimizing network virtualization in Xen". In *USENIX Annual Technical Conference*. p. 15–28.

Mininet. "An Instant Virtual Network on your Laptop (or other PC)". Online. <http://mininet.org/>. Visited on 17 January 2013.

Monsanto, Christopher, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. 2013. "Composing Software Defined Networks.". In *NSDI*. p. 1–13.

Mudigonda, Jayaram, Praveen Yalagandula, Jeff Mogul, Bryan Stiekes, and Yanick Pouffary. 2011. "Netlord: a scalable multi-tenant network architecture for virtualized datacenters". *ACM SIGCOMM Computer Communication Review*, vol. 41, n° 4, p. 62–73.

Murch, Richard, 2004. *Autonomic computing*.

Naous, Jad, David Erickson, G Adam Covington, Guido Appenzeller, and Nick McKeown. 2008. "Implementing an OpenFlow switch on the NetFPGA platform". In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. p. 1–9. ACM.

Ng, Eugene. 2012. "Maestro: A System for Scalable OpenFlow Control".

Nicira. "OVS". http://openvswitch.org. Visited on 7 June 2012.

Niebert, Norbert, Stephan Baucke, Ibtissam El-Khayat, Martin Johnsson, Börje Ohlman, Henrik Abramowicz, Klaus Wuenstel, Hagen Woesner, Jürgen Quittek, and Luis M Correia. 2008. "The way 4WARD to the creation of a future Internet". In *Personal, Indoor and Mobile Radio Communications, 2008. PIMRC 2008. IEEE 19th International Symposium on*. p. 1–5. IEEE.

ONF. 2012. "OpenFlow switch specification, version 1.3, Apr. 2012". https://www.opennetworking.org/images/stories/downloads/specification/openflow-spec-v1.3.0.pdf.

OpenNebula. "OpenNebula project". http://www.opennebula.org/. Visited on 2 June 2014.

OpenStack. "OpenStack project". http://www.openstack.org/. Visited on 10 December 2012.

Pagiamtzis, Kostas and Ali Sheikholeslami. 2006. "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey". *Solid-State Circuits, IEEE Journal of*, vol. 41, n° 3, p. 712–727.

Pfaff, B et al. 2009a. "Version 1.0. 0 (Wire Protocol 0x01)".

Pfaff, Ben and al. "OF-Config Protocol". http://www.openflow.org/wk/index.php/Config_Protocol. Visited on 10 July 2012.

Pfaff, Ben, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. 2009b. "Extending Networking into the Virtualization Layer.". In *Hotnets*.

Pfaff, Ben, B LANTZ, B HELLER, et al. 2012. "OpenFlow switch specification, version 1.3. 0". *Open Networking Foundation*.

Pfaff, Ben et al. 2013. "OpenFlow switch specification, version 1.4. 0".

Popa, Lucian, Minlan Yu, Steven Y Ko, Sylvia Ratnasamy, and Ion Stoica. 2010. "CloudPolice: taking access control out of the network". In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. p. 7. ACM.

Porras, Philip, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. 2012. "A security enforcement kernel for OpenFlow networks". In *Proceedings of the first workshop on Hot topics in software defined networks*. p. 121–126. ACM.

Qazi, Zafar Ayyub, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. "SIMPLE-fying middlebox policy enforcement using SDN". In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. p. 27–38. ACM.

Rosenblum, Mendel. 1999. "VMware's Virtual Platform^{TM}". In *Proceedings of Hot Chips*. p. 185–196.

Rotsos, Charalampos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W Moore. 2012. "Oflops: An open framework for openflow switch evaluation". In *Passive and Active Measurement*. p. 85–95. Springer.

Ryu SDN Framework. 2013. "Ryu SDN Framework". Online. <http://osrg.github.io/ryu/>. Visited on 19 January 2013.

Schlesinger, Cole, Alec Story, Stephen Gutz, Nate Foster, and David Walker. 2012. "Splendid Isolation: Language-Based Security for Software-Defined Networks". In *Proc. of Workshop on Hot Topics in Software Defined Networking*.

Sekar, Vyas, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. 2012. "Design and Implementation of a Consolidated Middlebox Architecture.". In *NSDI*. p. 323–336.

Sestinim, Fabrizio. 2006. "Situated and autonomic communication an EC FET European initiative". *ACM SIGCOMM Computer Communication Review*, vol. 36, n° 2, p. 17–20.

Sherwood, Rob, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. 2009. "Flowvisor: A network virtualization layer". *OpenFlow Switch Consortium, Tech. Rep*.

Shieh, Alan, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. 2010a. "Seawall: performance isolation for cloud datacenter networks". In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. p. 1–1. USENIX Association.

Shieh, Alan, Srikanth Kandula, and Emin Gun Sirer. 2010b. "SideCar: building programmable datacenter networks without programmable switches". In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. p. 21. ACM.

Shimonishi, Hideyuki, Yasunobu Chiba, Yasuhito Takamiya, and Kazushi Sugyo. 2011. "Trema: An Open Source OpenFlow Controller Platform". *GEC-11 Poster*.

Shin, Seungwon, Phillip A Porras, Vinod Yegneswaran, Martin W Fong, Guofei Gu, and Mabry Tyson. 2013a. "FRESCO: Modular Composable Security Services for Software-Defined Networks.". In *NDSS*.

Shin, Seungwon, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. 2013b. "AVANT-GUARD: scalable and vigilant switch flow management in software-defined networks". In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. p. 413–424. ACM.

Sridharan, Murari, Mark Pearson, Ilango Ganga, Geng Lin, Patricia Thaler, Chait Tumuluri, Albert Greenberg, Kenneth Duda, and Yu-Shun Wang. 2013. "NVGRE: Network virtualization using generic routing encapsulation".

Tavakoli, Arsalan, Martin Casado, Teemu Koponen, and Scott Shenker. 2009. "Applying NOX to the Datacenter.". In *HotNets*. Citeseer.

Tootoonchian, Amin and Yashar Ganjali. 2010. "HyperFlow: A distributed control plane for OpenFlow". In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*. p. 3–3. USENIX Association.

Tootoonchian, Amin, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. 2012a. "On controller performance in software-defined networks". In *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*.

Tootoonchian, Amin, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. 2012b. "On controller performance in software-defined networks". In *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*.

Tsou, Tina, Xingang Shi, Jing Huang, Zhiliang Wang, and Xia Yin. 2012. "Analysis of Comparisons between OpenFlow and ForCES". *Analysis*.

Vingarzan, Dragos, Peter Weik, and Thomas Magedanz. 2005. Design and implementation of an open ims core. *Mobility Aware Technologies and Applications*, p. 284–293. Springer.

Voellmy, Andreas and Paul Hudak. 2011. Nettle: Taking the sting out of programming network routers. *Practical Aspects of Declarative Languages*, p. 235–249. Springer.

Voellmy, Andreas and Junchang Wang. 2012. "Scalable software defined network controllers". In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. p. 289–290. ACM.

Wang, Mea, Baochun Li, and Zongpeng Li. 2004. "sFlow: Towards resource-efficient and agile service federation in service overlay networks". In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*. p. 628–635. IEEE.

Wendong, Wang, Yannan Hu, Xirong Que, and Gong Xiangyang. 2012. "Autonomicity design in Openflow based software defined networking". In *Globecom Workshops (GC Wkshps), 2012 IEEE*. p. 818–823. IEEE.

Yu, Minlan, Jennifer Rexford, Michael J Freedman, and Jia Wang. 2010. "Scalable flow-based networking with DIFANE". *ACM SIGCOMM Computer Communication Review*, vol. 40, n° 4, p. 351–362.

VITA

2014	M.Sc in Computer Science, École de Technologie Supérieure (ÉTS), Quebec University.
2013	M.Eng in Computer Science (Networks and Telecommunications), Institut National des Sciences Appliquées et de Technologie (INSAT Tunisia).
2007	High school Diploma in Mathematics, Tunisia.

PUBLICATIONS

Mohamed Fekih-Ahmed, Chamseddine Talhi, Mohamed Cheriet. “Towards Flexible, Scalable and Autonomic Virtual Tenant Slices”. Accepted in the 14th IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2015). Ottawa Canada. September 2014.

Alireza Shameli-Sendi, Yosr Jarraya, Mohamed Fekih-Ahmed, Makan Pourzandi, Chamseddine Talhi, Mohamed Cheriet. “Optimal Placement of Sequentially Ordered Virtual Security Appliances in the Cloud”. Accepted in the 14th IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2015). Ottawa Canada. September 2014.

Alireza Shameli-Sendi, Mohamed Fekih-Ahmed, Makan Pourzandi, Mohamed Cheriet. “Taxonomy of DDoS Mitigation Approaches for Cloud Computing”. Submitted for ACM Computing Surveys - CSUR-2014-0349. October 2014.

Mohamed Fekih Ahmed, Chamssedine Talhi, Makan Pourzandi and Mohamed Cheriet. “A Software-Defined Scalable and Autonomous Architecture for Multi-tenancy”. IEEE International Conference on Cloud Engineering (IC2E 2014), March 2014.

Mohamed Fekih Ahmed, Chamssedine Talhi, Makan Pourzandi and Mohamed Cheriet. “Virtual Data Center Scalability using Open Flow Controller”. The 2nd International Conference on Software Engineering and New Technologies (ICSENT 2013), December 2013.

PATENTS

Mohamed Fekih Ahmed, Chamssedine Talhi, Makan Pourzandi and Mohamed Cheriet. “Multi-tenant Isolation In A Cloud Environment Using Software Defined Networking”. Ericsson – ÉTS Patent (Filed). EAB/Q-03:000055. December 2013.

Yosr Jarraya, Alireza Shameli-Sendi, Mohamed Fekih Ahmed, Makan Pourzandi and Mohamed Cheriet. “Multi-stage Defense-aware Security Modules Placement in the Cloud”. Ericsson – ÉTS Patent (Submitted IVD) EAB|Q-04:000102. October 2014.