COGNITIVE NETWORK OPTIMIZATION VIA NETWORK VIRTUALIZATION

Master of Science Thesis

Tobie Yefferson BIYIHA AFOUNG

Eskişehir, 2019

COGNITIVE NETWORK OPTIMIZATION VIA NETWORK VIRTUALIZATION

Tobie Yefferson BIYIHA AFOUNG

MASTER OF SCIENCE THESIS

Program in Telecommunications

Supervisor: Assoc. Prof. Dr. Nuray AT

Eskişehir

Anadolu University

Graduate School of Sciences

July 2019

FINAL APPROVAL FOR THESIS

This thesis titled “Cognitive Network Optimization via Network Virtualization” has been prepared and submitted by Tobie Yefferson BIYIHA AFOUNG in partial fulfillment of the requirements in “Anadolu University Directive on Graduate Education and Examination” for the Degree of Master of Science (M.Sc.) Electrical and Electronics Engineering Department has been examined and approved on 16/07/2019**.**

Committee Members		**Signature**
Member (Supervisor)	: Assoc. Prof. Dr. Nuray AT	...……..
Member	: Assoc. Prof. Dr. Hakan ŞENEL	……..
Member	: Dr. Ögr. Üyesi Gökhan DINDIŞ	……..

Prof. Dr. Murat TANIŞLI
Director
Graduate School of Science

ABSTRACT

COGNITIVE NETWORK OPTIMIZATION VIA NETWORK VIRTUALIZATION

Tobie Yefferson BIYIHA AFOUNG

Department of Electrical and Electronics Engineering
Program in Telecommunications
Anadolu University, Graduate School of Sciences, July 2019

Supervisor: Assoc. Prof. Dr. Nuray AT

Cognitive networks are designed to sense, monitor and extract valuable data from their physical environment, and adapt quickly to support complex network applications in order to satisfy fast changing service demands. Virtualization technologies such as Software-Defined Network (SDN) and Network Function Virtualization (NFV) can be combined to create new frameworks offering the advantages of both SDN and NFV. These include dynamic resource reservation and flexible virtual network creation via NFV, and programmability of these resources and easy network management via SDN. These new combined frameworks could be leveraged to optimize and manage cognitive network architectures. However, optimizing cognitive networks using combined SDN/NFV frameworks requires new network management techniques and fast virtual network provisioning algorithms to replace the legacy manual algorithms. In this study, a new system called AUTOVNET introducing automation in the management and provisioning of virtualized software-defined networks is designed and implemented. AUTOVNET simplifies the manual configurations from the network administrator and increases the flexibility and adaptability of virtual networks. In addition, AUTOVNET performs pre-virtualization fault detection and deep packet analysis to determine the best healthy routing option between the source and destination hosts in the network. This approach allows the network administrator to easily and rapidly create, configure and manage virtual networks in larger complex network topologies.

Keywords: Cognitive network, Network virtualization, Software-defined network, Network function virtualization, Network management, OpenVirteX

ÖZET

AĞ SANALLAŞTIRMASI ÜZERINDEN BILIŞSEL AĞ OPTIMIZASYONU

Tobie Yefferson BIYIHA AFOUNG

Elektrik Elektronik Mühendisliği Bölümü
Telekomünikasyon Programı
Anadolu Üniversitesi, Fen Bilimleri Enstitüsü, Temmuz 2019

Danışman: Doç. Prof. Dr. Nuray AT

Bilişsel ağlar, bulundukları fiziksel ortamı algılamak, izlemek ve ortama ait değerli verileri elde etmek; karmaşık ağ uygulamalarını ve hızla değişen servis taleplerini zamanında kendini uyarlayarak desteklemek üzere tasarlanmışlardır. Yazılım tanımlı ağ (SDN) ve ağ fonksiyon sanallaştırması (NFV) gibi sanallaştırma teknolojileri, her iki teknolojinin avantajlarını sunan yeni çerçeveler oluşturmak için birleştirilebilir. Bu avantajlar NFV ile dinamik kaynak ayırma, esnek sanal ağ oluşturma işlemlerini ve SDN ile bu kaynakların programlanması ve ağ yönetiminin kolaylıkla yapılabilmesini içerir. Dolayısıyla, bu yeni çerçeveler bilişsel ağ mimarilerinin iyileştirilmesi ve yönetiminde kullanılabilirler. Bununla beraber, bilişsel ağların SDN/NFV çerçeveleri kullanılarak iyileştirilmesi, geleneksel manuel algoritmaları değiştirecek yeni ağ yönetim teknikleri ve hızlı sanal ağ sağlama algoritmalarını gerektirir. Bu çalışmada, AUTOVNET olarak adlandırılan, sanallaştırılmış yazılım tanımlı ağ sağlama ve yönetiminde otomasyonu tanıtan yeni bir sistem tasarlanmış ve gerçeklenmiştir. AUTOVNET ağ yöneticisinin manuel olarak yapması gereken yapılandırmaları basitleştirir ve sanal ağların esneklik ve uyarlanabilirliğini arttırır. Bunun yanı sıra, AUTOVNET ağdaki kaynak ve hedef terminaller arasındaki en sağlıklı yönlendirme seçeneğini belirlemek üzere sanallaştırma öncesi hata tespit ve derin paket analizini gerçekleştirir. Bu yaklaşım, ağ yöneticisinin daha büyük ve karmaşık ağ topolojilerinde hızlı ve kolay bir şekilde sanal ağ oluşturması, yapılandırması ve yönetmesini sağlar.

Anahtar Kelimeler: Bilişsel ağ, Ağ sanallaştırılması, Yazılım tanımlı ağ (SDN), Ağ fonksiyon sanallaştırılması (NFV), Ağ yönetimi, OpenVirteX

ACKNOWLEDGEMENTS

First and foremost, I am truly indebted and wish to express my gratitude to my supervisor Assoc. Prof. Dr. Nuray AT for her inspiration, excellent guidance, continuing encouragement, and unwavering confidence and support during every stage of this endeavor without which, it would not have been possible for me to complete this undertaking successfully. I also thank her for the insightful comments and suggestions which continually helped me to improve my understanding. I could not have imagined having a better advisor and mentor for my study.

I express my deepest gratitude to the management and staff members of the Department of Electrical and Electronics Engineering for their kind co-operation and encouragement, which help me in the completion of my postgraduate program.

Had I forgotten to appreciate the Turkish government and her scholarship board (YTB) for providing this opportunity, I would have been an ingrate. I thank my fellow lab mate, Ibrahim Wonge Lisheshar for the stimulating discussions and for all the fun we have had in the last two years.

I would also like to express my heartfelt gratitude to my friends Obby Nawa Likando and Razak Musa Mohammed for their steadfastness and support.

Fimally, my wholehearted gratitude to my Mother, Afoung Solange, brothers, sisters and family relatives for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Tobie Yefferson BIYIHA AFOUNG

16/07/2019

STATEMENT OF COMPLIANCE WITH ETHICAL PRINCIPLES AND RULES

I hereby truthfully declare that this thesis is an original work prepared by me; that I have behaved in accordance with the scientific ethical principles and rules throughout the stages of preparation, data collection, analysis, and presentation of my work; that I have cited the sources of all the data and information that could be obtained within the scope of this study, and included these sources in the references section; and that this study has been scanned for plagiarism with "scientific plagiarism detection program" used by Anadolu University, and that "it does not have any plagiarism" whatsoever. I also declare that, if a case contrary to my declaration is detected in my work at any time, I hereby express my consent to all the ethical and legal consequences that are involved.

Tobie Yefferson BIYIHA AFOUNG

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

INDEX OF ABBREVIATIONS AND SYMBOLS

CN	: Cognitive Network
QoS	: Quality of Service
FCAPS	: Fault, Configuration, Accounting, Performance and Security
NV	: Network Virtualization
VN	: Virtual Network
SDN	: Software-Defined Network
OF	: OpenFlow
vSDN	: virtualized Software-Defined Network
API	: Application Programming Interface
ONF	: Open Networking Foundation
ITU	: International Telecommunications Union
ETSI	: European Telecommunications Standards Institute
NFV	: Network Function Virtualization
VNF	: Virtualized Network Function
NFVI	: Network Function Virtualization Infrastructure
NFV MANO	: Network Function Virtualization Management and Orchestration
VIM	: Virtualized Infrastructure Manager
VNFM	: Virtualized Network Function Manager
NFVO	: Network Function Virtualization Orchestrator
ICMP	: Internet Control Message Protocol
ARP	: Address Resolution Protocol
LLDP	: Link Layer Discovery Protocol
OVX	: OpenVirteX
BSR	: Best Switch Route
NID	: Network Information Database
SRD	: Switch Route Database
SID	: Switch Interface Database
ARD	: Active Route Database

1. INTRODUCTION

Over the last decades, Internet technology has undergone very few changes even though data communications and Internet traffic have surged increasingly at a very fast pace. This lack of adaptation to current network conditions has led to suboptimal network performances. Current network frameworks, behaviors, policies, and protocols do not have adequate response mechanisms to make intelligent adaptations. These networks heavily rely on routing algorithms to forward packets over the network, and external operators to perform manual configurations and policy decisions. Furthermore, limited information is shared between network elements and network failures can only be detected once packets are lost. For example, switches and routers follow strict rules for transferring packets and have very little knowledge about the activity (network status) on the other end of the network. This current network framework was originally designed to simplify the design of network devices and reduce the overall complexity of the network. As a result, these networks became rigid and network problems could only be detected and resolved, with little means of predicting them.

Cognitive networking embraces complexity to provide end-to-end optimal network performance. In (Thomas, DaSilva, & MacKenzie, 2005), R. Thomas and al. defined for the first time the term Cognitive Network (CN) as a network that uses cognitive processes (sensing, learning, and adaptation) to perceive the state of the entire network in real-time, then plans, decides, and acts, i.e., learns from collected network data and adapts accordingly to meet end-to-end network goals.

The cognitive approach aims to develop proactive networks capable of using intelligent network devices to collect network data, make network predictions, easily adjust their activities, and modify their configurations to adapt to current and future network conditions. CNs can detect and manage network problems before they occur and use each network scenario to improve their problem-solving abilities. As a result, CNs increase the performance and efficiency of the network and optimize end-to-end network traffic between the source and destination for the entire network (Fortuna & Mohorcic, 2009).

A network is said to be cognitive when self-aware and self-adjusting components replace all statically configured portions of the network. Cognitive networking ushers a

new era of communication with promising network architectures offering the best end-to-end performance by adjusting automatically network parameters for optimal data transfer between devices.

1.1. The Problem Statement

There are multiple research directions on how to optimize cognitive networks. One uses artificial intelligence models and game theory to predict future network behaviors and network problems beforehand. Another direction focuses on how to utilize new networking frameworks like virtualization technologies such as Software-Defined Networks (SDNs) and Network Function Virtualization (NFV) and their combinations to optimize the management of cognitive networks since both paradigms offer adaptability, flexibility and programmability.

Some challenges and open questions regarding research on the combination SDN and NFV for better network performance and the challenges of modeling, managing and optimizing their combined architectures are summarized below,

Network information extraction

A proper management system for cognitive networks must be able to fetch accurate network information from the physical network. This information can then be updated at regular time intervals and used by a central management system to coordinate network operations.

Virtualization layer design

Combining SDN and NFV paradigms requires a virtualization layer or hypervisor. There are different hypervisors architectures, from hardware-enhanced to pure software-based, distributed and centralized ones. However, a general understanding of how these hypervisors operate is needed in order to design, create, configure, and manage the network. Moreover, additional research is needed to properly select the hypervisor for simulations since the architecture and operation of this hypervisor will affect the performance of the network.

Fast virtual network configuration and provisioning

Combining SDN and NFV to create virtual networks requires mapping of physical network resources to virtual ones. The challenge is to rapidly map these resources. Therefore, a sophisticated mapping scheme should be designed to rapidly configure and provide virtual networks according to users' demands.

Automatic management

CNs consist of intelligent and programmable network nodes (Thomas, Friend, Dasilva, & Mackenzie, 2006). For this reason, SDN and NFV technologies can be used to efficiently manage these programmable networks. However, this management must be automatic and/or autonomous to fit into the cognitive nature of the system. Therefore, management systems must be designed to automatically detect and predict network scenarios such as link failures, to ensure end-to-end performance with limited human intervention, as promised by CNs.

1.2. The Objective and Relevance of The Study

This study addresses the challenges of manual configuration in SDN virtualization and provides a solution called AUTOVNET, a self-configurable module that replaces the network embedder module in the traditional OpenVirteX (OVX) system architecture (Ali Al-Shabibi et al., 2014) to enable automatic virtual SDN configuration for a variety of network topologies and dynamic scenarios. We begin by developing an algorithm to extract network information from the underlying network topology and later use this information to create a network repository (database). The management module AUTOVNET uses this database to minimize manual configurations and to rapidly create and configure virtual SDN networks.

The second major contribution of the study is fault detection and fault prevention. AUTOVNET can analyze network data from Wireshark (Ndatinya, Xiao, Rao Manepalli, Meng, & Xiao, 2015) and carry out pre-virtualization faulty link/switch detection. In other words, AUTOVNET is a proactive fault management system that detects and isolate faulty network resources to enable the creation of healthier virtual networks.

The third major contribution of the study is on performance management. AUTOVNET is a pure software implementation written in Python programming. AUTOVNET can perform performance computations to determine congested routes as well as packet delay analysis to automatically determine optimal routes for data transfer between devices and avoid congested routes to maximize network resource utilization and create faster virtual networks.

1.3. Thesis Organization

The remainder of the thesis is structured as follows:

Chapter 2 gives detailed background information on cognitive networks. It also describes the general capabilities of cognitive networks, the cognitive cycle, and aspects of cognitive network management. Furthermore, it provides a brief summary of the different types of cognitive networks in the literature.

Chapter 3 addresses the optimization of cognitive networks via SDN, NFV, and combined SDN/NFV. It first provides a detailed background on SDN and NFV and describes their respective architectural frameworks. It explores the optimization benefits of both SDN and NFV and offers comparisons between SDN and traditional networks as well as between SDN and NFV technologies. This chapter finally discusses the different types of hypervisor architectures and reviews the limitation of available virtualization solutions using OpenFlow in the literature.

Chapter 4 is mainly concerned with the design and implementation of AUTOVNET. It describes how the system architecture of AUTOVNET compares to tradition OVX (an SDN hypervisor) system architecture, and explains the management capabilities (configuration, fault and performance) of AUTOVNET as well. Mininet, Floodlight and Wireshark tools are also presented in this chapter as they are used for the simulations. Furthermore, the two AUTOVNET operations (random and smart virtualization) are explained and tested in Spyder IDE. The results show that AUTOVNET improves on the legacy OVX architecture by providing rapid virtual network configurations and fault detection with limited manual configurations (inputs) from the network administrator.

Chapter 5 concludes the study and provides recommendations and thoughts on future work.

2. BACKGROUND ON COGNITIVE NETWORKS

In this chapter, we first introduce the concepts and capabilities of Cognitive Networks (CNs) (Sec. 2.1). Afterwards, Sec. 2.2 gives a detailed view and stages in the cognitive cycle. In Sec. 2.3, we review the various cognitive network management schemes in the literature. Finally, Sec. 2.4 explores the two types of cognitive networks, namely the cognitive wireless network and the cognitive Internet network.

2.1. General Capabilities of Cognitive Networks

Every CN must be able to sense and monitor the physical environment (sensing capabilities), analyze and extract useful information from sensed data (learning capabilities) and reconfigure its parameters according to current physical environmental conditions (adaptive capabilities) (Nazmul Siddique, Syed Faraz Hasan, & Salahuddin Muhammad Salim Zabir, 2017).

2.1.1. Sensing capability

CNs have the ability to sense or capture real-time information from the physical environment. In wireless networks, sensing is very important. For example, this capability can be used for channel monitoring, interference avoidance, and wireless device power monitoring. Likewise, in wired networks, sensing plays a vital role in decision making and routing. Sophisticated algorithms like autonomous learning and network protocols can be used to capture temporal variations in the physical network environment. Additional, sensing capabilities involve:

- **Network monitoring:** A CN can monitor network activities and detect congested as well as under-utilized network resources. This may be used to increase network resource utilization since service running on congested network resources can be redirected to idle network resources. A CN with resource sharing mechanisms monitors the network and allow the utilization of network resources by secondary users when primary users are inactive.
- **Location identification:** CNs are able to identify and determine the location of network entities. In recent years, location technologies have led to location-based services (FCC, 2012) and ultrafast routing.
- **Path discovery:** A CN can determine the best path to establish communications between its terminals. Some terminals might be reachable after one or multiple network entities (router, switches, and servers) and multiple routes might lead to the

same network device. That's why, the ability to select the appropriate communication path between CN terminals is very important.

- **Service discovery:** A CN analyzes services subscribed by the network users and recommends other appropriate services that the user might be interested in. Machine learning algorithms like "*recommender systems"* are based on this concept. Service discovery is also important for creating clusters or group devices utilizing the same applications and services together.

2.1.2. Learning capability

The sensing capability provides resource awareness, whereas learning capability enables the network to be programmed dynamically according to the physical network environment. CNs can be programmed to use a variety of transmission, access and network topologies and technologies supported by their hardware resources. Reconfigurations supported by CNs can be given as follows:

- **Resource agility:** This is the ability of a CN to change its network topology. This ability dynamically selects the appropriate network resources and routing paths for communications between network terminals.
- **Flexible network:** A CN is an opportunistic network (Nazmul Siddique et al., 2017) which adapts constantly to variations in network scenario. The real-time data acquired during sensing enables the network to adequately select a new network topology and communication paths.
- **Dynamic system access:** A CN may contain multiple heterogeneous systems providing a variety of services running on different communication protocols. Thus, system access reconfiguration is necessary in order to be compatible and fully support every system.

2.1.3. Adaptive capability

The adaptive capability provides the ability to fine-tune network parameters and functions based on collected network data and meet Quality of Service (QoS) goals. Based on the monitoring and learning capabilities discussed before, CNs are able to self-organize their network operations and provide faster network connectivity and greater performance. The adaptive capabilities of CNs involves the following:

- **Resource management:** An improved resource management scheme is needed to efficiently organize information on network resources and allocate these resources

to minimize network failures and maximize network efficiency. A resource pooling technique can be utilized to manage network resources as in cloud computing.

- **Connection management:** The large heterogeneous nature of CNs has increased the complexity of routing. Connection management techniques can help in network discovery and provide useful information. This information can be used to select the best routes for data transfer between network devices. Connection management can equally be useful in capacity measurement to minimize network congestion and under-utilization of network resources.

2.2. Cognitive Cycle

Based on the information collected during sensing and monitoring, CNs uses the cognitive cycle to methodologically decide on the best course of action (Vishram Mishra , Jimson Mathew, & Chiew-Tong Lau, 2017). The cognitive cycle is made up of various stages or states through which the CN continuously monitors the physical environment and acts accordingly. The cognitive cycle is shown in Figure 2.1.

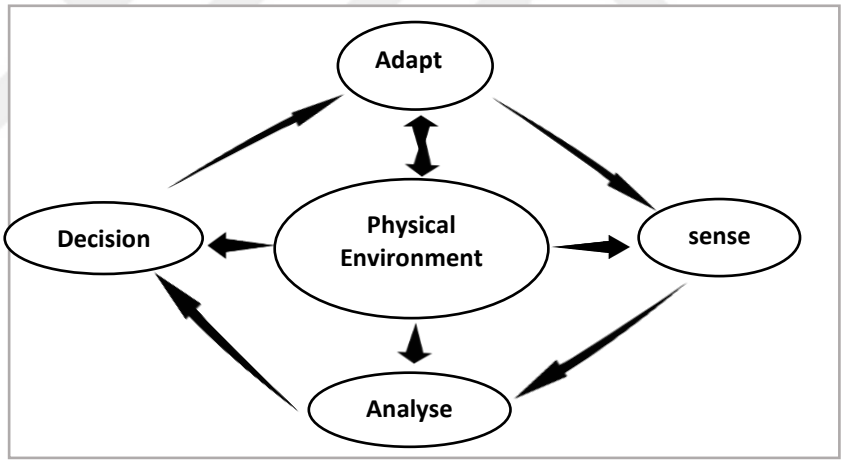


Figure 2.1. *Cognitive cycle*

The cognitive cycle starts with sensing the environmental state. During this state, physical network parameters are recorded. Next, these parameters are analyzed to obtain more physical and logical figures. Based on these analyses and the operational requirements (bandwidth, speed, latency, service capacity, etc.) of network services, the cognitive network responses accordingly and makes a decision in the next phase. The decision involves the adaptation or reconfiguration of the network system parameters to avoid performance degradation and ensure optimal network settings in different environmental scenarios. However, when the existing network parameters are already perfectly adapted to the current network state, the adaptation state will not be

applicable. In this case, there is no adaptation and the cycle restarts with the next cognitive sensing phase.

2.3. Cognitive Network Management

Cognitive network management (Ayoubi et al., 2018) refers to all activities associated with running a cognitive network. Network management can be divided into the following areas; Fault, Configuration, Accounting, Performance, and Security (FCAPS) (Alexander Clemm, 2006) management.

2.3.1. Fault management

Fault management deals with failures (hardware or software failures) that occur in the network. Fault management, therefore, relies on network monitoring to ensure that all system functions are running smoothly, and failure mechanisms are properly activated when faults occur. Fault management is critical to ensure optimal user experience, i.e., users do not experience service disruptions and that when they do, these disruptions have minimal impact on the overall service experience. Fault management functions include network monitoring, alarm management, fault detection, fault diagnosis, fault prevention, fault prediction, troubleshooting, and proactive fault management.

2.3.2. Configuration management

All the devices within a network need to be configured for the network to function properly as a homogenous unit. Configuration management performs the modification of the configuration settings of network equipment. This includes initial configuration (i.e., startup configurations required to properly connect all network entities), and ongoing configuration (i.e., the operational configuration that continuously adjusts or updates core network settings to provide novel network services and functionalities). Configuration management is very critical since other network management functions depend on it to provide an acute diagnosis. With a defective configuration management system, the network provider will be unable to fine-tune network services. Configuration management functions include network resource configuration, network auditing, network configuration backup, and device synchronization.

2.3.3. Accounting management

It is essential for any organization to properly evaluate the cost/benefit ratio of their services and generate revenues for the services they provide. Accounting management provides the functions that allow businesses to generate revenue for the services they provide and keep track of their usage. From accounting data, service providers might

terminate an unprofitable service to invest in a more lucrative one. Accounting management functions include fraud detection, service consumption data collection, and business model forecast.

2.3.4. Performance management

The performance of a network is characterized by a number of performance measurements known as performance metrics. These metrics provide a measurement of the behavior of the network under a variety of physical conditions. Some performance metrics include:

- *Throughput:* This measures the number of communication units performed per unit time. The communication units depend on the type of the network layer. For example, bytes per seconds for link layer transmission and service per second for web service throughput measured at the application layer.
- *Delay:* This is measured per unit time. Network services experience different types of delay which are measured differently. For example; packet delay at the network layer and octet delay during transmission at the link layer.
- *Quality of Service:* This parameter shows how well a service performs under certain conditions. For example, the percentage of packets dropped in a communication system can determine whether the output is of acceptable quality or otherwise there is a need for retransmission.

Understanding these performance metrics and providing real-time optimized network settings suitable for higher throughput, lesser delays and greater service quality in a variety of network scenarios are key aspects to enhance the overall user experience.

2.3.5. Security management

The last letter “S” in FCAPS stands for security. Security management deals with all aspects related to securing the network from the spread of worms and viruses, threats from hacker attacks and malicious intrusion attempts. Two aspects of security management exist, namely, security of management and management of security. Security of management deals with the security of all management operations like password request before data access and the request for access privileges to view sensitive corporate data. Whereas, management of security deals with the security of the network system itself. Some of the threats management of security deals with include hacker attacks, Denial-of-Service (DoS) attacks, viruses and worms, and spam. Some

management of security strategies includes Intrusion Detection System (IDS), firewalls, and blacklists.

2.4. Types of Cognitive Networks

In this section, we briefly describe the two types of CNs, that is, cognitive wireless (cognitive radio network) and cognitive Internet (cognitive core network). As a reminder, this thesis will focus solely on Cognitive Internet. Nevertheless, a review of the cognitive Internet, as well as cognitive wireless, is aimed at painting a broader view of the cognitive network world.

2.4.1. Cognitive wireless

Currently, the static spectrum allocation policy is used to assign spectrum channels to license holders by governmental agencies on a long-term basis for large geographical coverage. This policy is very inefficient because it turns out that these license spectrum holders use the spectrum, sporadically leading to underutilization of wireless resources. In a world with increasing demand for wireless channels, a new allocation policy is needed for efficient spectrum usage.

Cognitive Radio Network (CRN) proposed to solve these spectrum problems by implementing dynamic spectrum allocation techniques (Ian F. Akyildiz, Lee, Vuran, & Mohanty, 2006). A CRN uses Cognitive Radios (CRs) which has the capacity to access and share the wireless channel with licensed holders in an opportunistic manner (Khozeimeh & Haykin, 2010). CRN has as goals to guarantee seamless communication, reliable QoS and minimize interference (Haykin). These goals can be realized through efficient and dynamic management (I. F. Akyildiz, Lee, Vuran, & Mohanty, 2008) of the wireless spectrum.

CRs operates as follows:

- Monitor and identify idle and available channels
- Select the best available spectrum
- Collaborate with other CRs to avoid channel access monopoly
- Vacate the channel seamlessly when a licensed holder is detected.

2.4.2. Cognitive Internet

Our modern society is centered around communication networks. Sophisticated network applications and advanced wireless technologies have changed the way we share information and communicate with one another. The Internet represents the

communication medium of the modern era as well as a platform delivering services like social networks, e-commerce, and video-on-demand. However, as the Internet represents a global interconnection of complex heterogeneous networks, several issues about its management and performance are rising increasingly. The legacy Transmission Control Protocol/Internet Protocol (TCP/IP) architecture created in 1983 was not designed to support the Quality of Service (QoS) requirements of modern multimedia applications due to lack of adaptability and cross-layer mechanisms (R. Jain, 2006). For this reason, both academic and industrial researchers are focusing on adaptable network protocols to optimize core network performance in a decentralized manner.

The future Internet (Stuckmann & Zimmermann, 2009) is said to be a self-manageable system called the autonomic computing paradigm. This paradigm is based on the principles of self-optimization, self-healing, self-configuration, self-protection, and content awareness. In this context, the communication system would be capable of autonomous management, limiting human intervention. Autonomous network devices should be able to reconfigure themselves and constantly adapt to changing network conditions in order to avoid performance degradations with limited manual configurations. Thus, the behavior of these autonomous systems must be guided by high-level rules defined by administrative and business policies.

There are key enabling technologies to support self-adaptation within the TCP/IP stack and pave the path to the evolution and deployment of cognitive Internet solutions. These technologies include cross-layer design, distributed and agent-based solutions, Artificial Intelligence/ Machine Learning (AI/ML) based algorithms and autonomic network architectures.

To sum up, the cognitive core network is an evolution of the concept of cognitive wireless network (Di Benedetto, Cattoni, Fiorina, Bader, & De Nardis, 2015). While cognitive radio technologies focus on tuning the parameters of the link and physical layers to provide efficient spectrum management, cognitive core technologies, on the other hand, expands the dynamic tuning of network parameters to improve overall network performance at a system-wide scale.

This thesis focuses on the optimization of the cognitive core network architecture and proposes an autonomic network management system which can be leveraged to dynamically manage large complex heterogeneous network topologies.

3. OPTIMIZATION OF COGNITIVE NETWORKS

CNs can monitor and collect useful data from the physical environment, and after analyzing this data, react in a timely manner to satisfy all related requirements in order to maximize network capacity and quality of service (QoS) at all times and in all scenarios. This is known as cognitive network optimization. It should be noted that effective optimization should provide:

- *Load balancing:* Load balancing plays a critical role in full optimization activities, especially in cases where there is the need to efficiently distribute the incoming workload (network traffic) across multiple network resources to avoid traffic congestion and decrease latency. User demands and quality of service requirements of an application can trigger the obligatory reconfiguration activity of core network elements.
- *Network resource management optimization:* To cope with traffic growth, investing in additional network infrastructure is often not an option. Service providers and enterprises are always under pressure from investors to attain the highest possible output with little investment in order to maximize the profit. Therefore, improving and efficiently utilizing the available network infrastructure by implementing strong network resource optimization solutions is the key for all business models.

Several optimization techniques (Z. S. Zhang, Long, & Wang, 2013) are currently designed and tested by both industrial and academic researchers with the most promising being the ***Network Virtualization (NV)***.

The International Telecommunication Union Sector (ITU-T) (ITU-T, 2012) defines the concept of NV as the creation of network partitions, logically isolated on a shared physical network so that heterogeneous clusters of multiple virtual networks can coexist simultaneously over the same network resources. The idea of network virtualization that allows several virtual networks to coexist within a single physical network is not novel. Conventional technologies such as Virtual Private Network (VPN), Voice over IP (VoIP) and Virtual Local Area Network (VLAN) are commonly used to build isolated networks over shared physical infrastructures.

Technologies like VPN and VoIP services are examples of overlay networks. An overlay network is a logical network which runs independently and does not cause any

changes to the underlying physical network. Figure 3.1 shows an overlay network architecture.

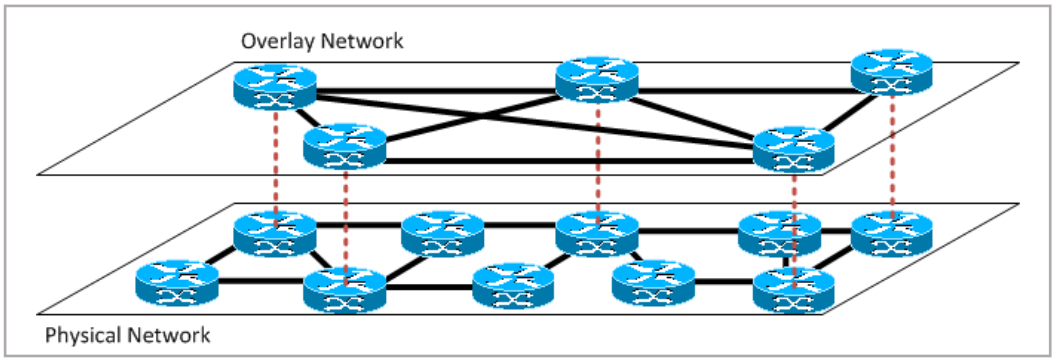


Figure 3.1. *Overlay network (Elsen, 2013)*

The following design goals must be achieved in the realization of a network virtualization solution:

- *Isolation:* Since multiple independent virtual networks coexist over the same physical infrastructure, their operations may cause interference resulting in instability in the entire network ecosystem. In order to mitigate these interferences, the virtualization solution must provide secure isolations, such as performance, security, control plane, and data plane isolation among the virtual networks.
- *Network abstraction:* This involves hiding the overall underlying characteristics of physical network resources from the virtual network tenants and provides simplified interfaces for resources access and control. The network virtualization solution has the ability to customize network operations and manage all virtual networks independently.
- *Topology awareness and rapid reconfigurability:* The network virtualization solution must provide effective use of the virtual resources during the creation of virtual networks and allow the dynamic reconfiguration of these networks during optimization processes.
- *Performance:* Network virtualization increases the complexity of the overall network ecosystem leading to significant performance degradation of the network. Thus, the network virtualization solution must guarantee virtual network performances that are as good as non-virtualized networks, keeping any performance degradation to the minimum.

- *Programmability:* The network virtualization solution should support the programmability of both control and data plane in order to provide evolvability and flexibility of the virtual networks using customized protocols, new control schemes, and packet forwarding or routing functions. Furthermore, programmability allows each virtual network to support the rapid deployment of new network architectures and control schemes independent of other virtual network architectures.
- *Management:* The network virtualization solution should provide an effective and integrated management system capable of accessing both physical and virtual resource information. Due to rapid changes of the virtualized network ecosystem, network management is essential to monitor all network operations in order to troubleshoot network failures.
- *Mobility:* The network virtualization solution should support the mobility of virtual resources including computing resources, applications, and services across virtual networks in order to meet their QoS requirements, respond effectively to users' demands, and increase the efficiency of the entire network.

Network virtualization is still in its early stages and there are many open research opportunities to develop new virtualized architectures, applications, and systems. Currently, there are three key network virtualization solutions (R. Jain & Paul, 2013): Software-Defined Networking (SDN), Network Function Virtualization (NFV), and Cloud Computing as illustrated in Figure 3.2.

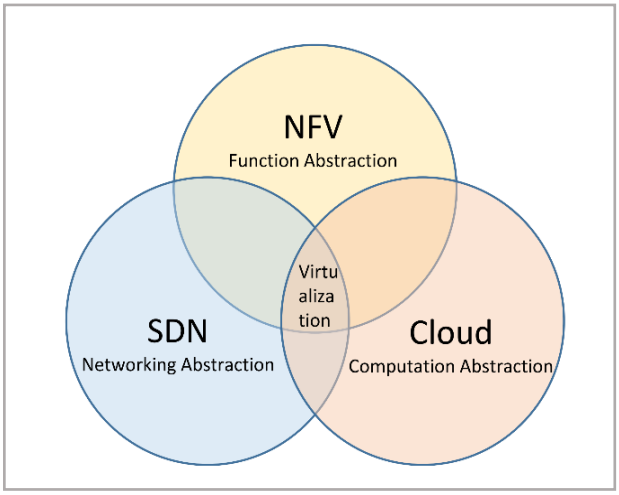


Figure 3.2. *Network virtualization technologies*

This thesis exploits SDN and NFV to optimize the cognitive network architecture. Cloud computing, on the other hand, is not covered, but rather recommended as future research direction.

The remaining of this chapter can be outlined as follows; Sec. 3.1 introduces cognitive network optimization via SDN, presents a background on SDN, SDN architecture and compares between SDN and traditional networks. Sec. 3.2 explores cognitive network optimization via NFV, presents a background on NFV, NFV architecture, and compares the concepts SDN and NFV. Finally, Sec. 3.3 concludes this chapter by presenting the combined NFV/SDN optimization and related works.

3.1. Optimization via Software Defined Networking (SDN)

This section first introduces background on SDN and SDN architecture and reports on the optimization benefits of using SDN. Finally, this section concludes with a comparison between SDN and traditional networking.

3.1.1. Background on Software-Defined Networks (SDN)

Currently, SDN is significantly attracting attention from both industry and academia as an important architectural solution for the management of large-scale networks, which may require dynamic re-configuration and re-policing from time to time. The goals of SDN include the ability to accelerate innovation, network business cycles, adapt to customer demands and customize network resources to include service-aware networking.

The ITU (ITU-T, 2014) defines SDN as a set of techniques that allow network administrators to directly program, manage, control, and orchestrate network resources to facilitate the operation, design, and delivery of network services in a scalable and dynamic manner via open interfaces such as OpenFlow (OF) protocol (ONF, 2013).

Prior to the advent of SDN (Tourrilhes, Sharma, Banerjee, & Pettit, 2014), networking architectures were hardware-driven and proprietary. These conventional architectures were unable to dynamically and adequately respond to the needs of modern data centers, carrier environments, and college campuses. As a result, the need to design a software-driven networking architecture, where configuration and policing could be done in a dynamic, centralized, and logical manner without the need to configure every device separately was inevitable.

3.1.2. SDN framework and architecture

According to the Open Network Foundation (ONF) (ONF, 2015), the SDN paradigm has the potential to dramatically simplify network deployment, operations, management, and innovation.

In traditional networks, the data plane is tightly coupled into the same network device as the control plane. SDN (Hu, Hao, & Bao, 2014) decouples the forwarding functions and the network control, allowing the underlying infrastructures in the data plane to become simple programmable packet forwarding and routing network devices. Similarly, it allows network control and automation in the control plane to be directly programmable via OF. Figure 3.3 shows the SDN architecture.

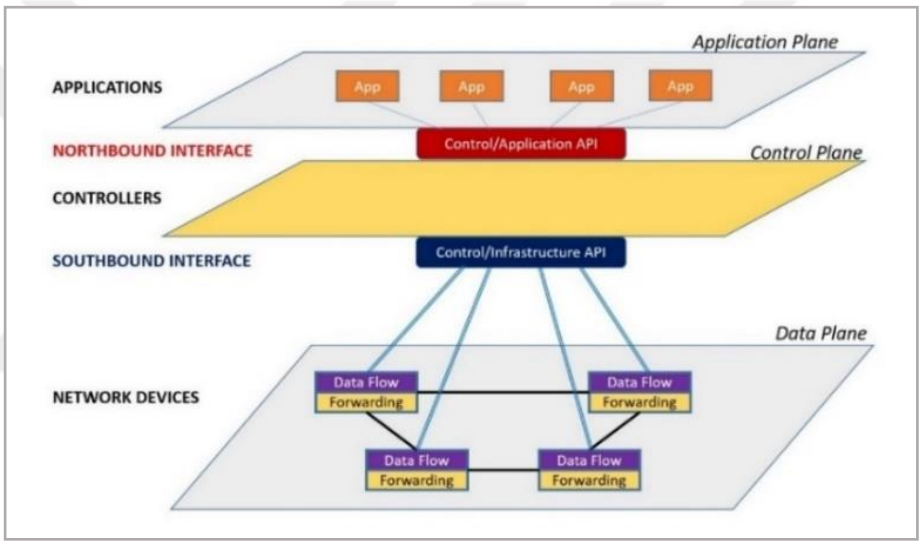


Figure 3.3. *Software-Defined Network (SDN) framework and architecture (Hoang, 2015)*

A programmable SDN controller logically centralized in the control plane manages and directs all packet transfer and routing policies in the network via the OF protocol. This protocol sets the rules of communication between the data plane and the control plane through the southbound interface. Whereas, the northbound interface is used for communications between the application layer and the control plane using the application programming interface called REST API (REST-API).

The SDN architecture is divided into three planes or layers: data (infrastructure), control, and application planes (ONF, 2014).

3.1.2.1. Data plane (infrastructure layer)

The data plane is sometimes referred to as the forwarding plane or infrastructure layer. This plane is mainly composed of basic traffic processing or traffic forwarding resources like routers and switches. The data plane is responsible for the forwarding and routing of data traffic (Braun & Menth, 2014). In addition to routing and packet steering, the data plane collects datagram which is used by the controller to monitor, update, and optimize the performance of the network according to QoS requirements.

In SDN, all control functions associated with the data plane of traditional networks have been abstracted and transferred to the centralized SDN controller in the control plane. As such, all forwarding, and routing actions performed by elements in the data plane are predetermined in the flow tables of the controller. The data plane contains the underlying network resources which can be virtualized, managed and controlled via the control plane or other implementation planes.

The OF-witches in the data plane communicates with the SDN controller in the control plane through the southbound interface using the OpenFlow (OF) protocol, which determines the operation and management messages.

OF was initially designed to decouple the simple forwarding hardware elements from the routing software intelligence to allow testbed and academic research networks to rapidly deploy and evaluate new algorithms and control methods. Later, OF was adopted by industries because it guaranteed business benefits for hardware manufacturers and provided an open control interface to the operating systems of network devices.

There are three different types of OpenFlow messages defined by the OF protocol, namely, Asynchronous messages (initiated by OF-switch to the controller), Controller-to-Switch messages (initialed by the SDN controllers to the OF-switch), and Symmetric Messages (initiated by either the OF-switch or the SSDN controller) (Azodolmolky, 2013). Figure 3.4 shows the three OpenFlow messaging protocols and their functions. Under OF rules, each incoming packet is matched with a specific header, if a packet marches multiple header in the flow entries, the selected entry is the one with the highest priority. The basic OF actions include deny forwarding, forward as default, forward to the controller, forward out through, and modify various field headers in the packet. The counter numbers the packet processes by the protocol rule and flow entries are deleted if their processing time exceeds a specific time frame.

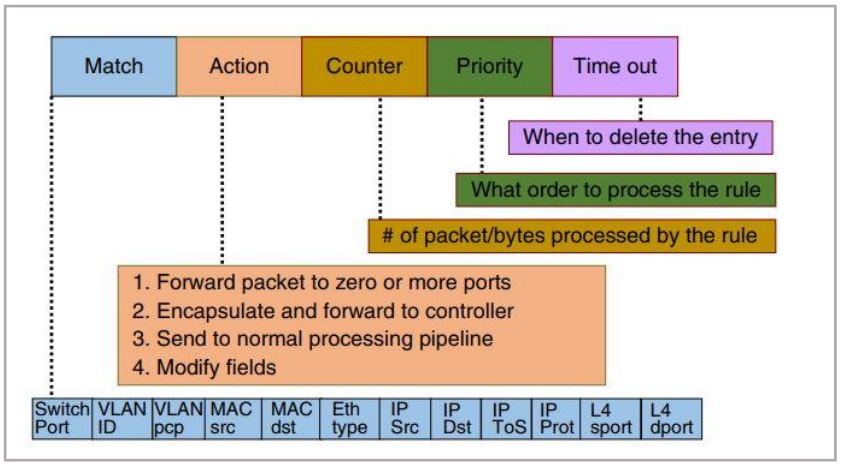


Figure 3.4. *OpenFlow messaging protocol (Azodolmolky, 2013)*

3.1.2.2. SDN control layer (control plane)

The control layer is considered as a virtual overlay network that is logically found on top of the underlying data plane. The forwarding and routing intelligence (the SDN controller) abstracted from the routing devices in legacy networking resides in this plane. The control plane contains one or more software controllers that are programmable and stands as an intermediate layer connecting both the application layer through a northbound interface like REST API and the data plane through a southbound interface like OpenFlow. The SDN controller in the control plane provides a centralized and uniform programmatic interface to the entire network. Figure 3.5 shows an overview of an SDN controller.

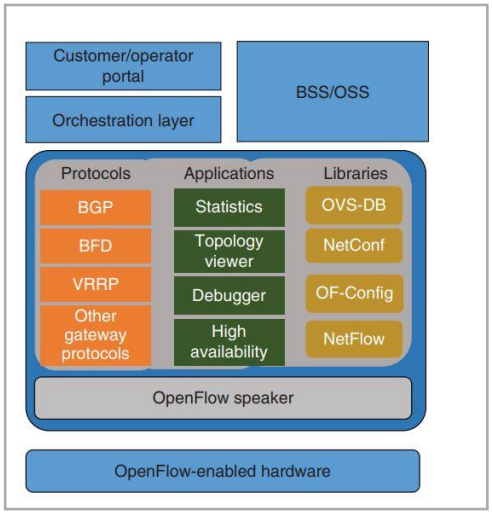


Figure 3.5. *Overview of SDN controller (Y. Zhang, 2018)*

The SDN controller consists of multiple subprocesses and can be installed as a single consistent unit on commodity servers. The SDN controller contains three fundamental components: the protocol handler (deals with legacy network protocols), the applications (use network information), and libraries (to support various southbound interfaces). Using these components, the SDN controller can perform functions such as management of network states, implementation of firewall rules, routing, switching, update flow entries, network device, topology and service discovery.

SDN controller architectures can either be distributed or centralized. A centralized SDN controller has a single control plane, which performs all control and management tasks in the entire network, whereas a distributed SDN controller (Jiménez, Cervelló-Pastor, & García, 2014) has several control planes shared by all the devices in the data plane. Such distributed architectures are very beneficial for large, complex networks like data centers since each control plane may handle a specific task such as traffic engineering and virtual network management. Table 3.1 provides a comparison of the most popular open source SDN controllers.

Table 3.1. *A comparative feature-based analysis of open source SDN controllers* (Stancu et al., 2015)

	Programming Language	GUI	Documentation	Modularity	Distributed/ Centralized	Platform Support	Southbound APIs	Northbound APIs	Partner	Multithreading Support	OpenStack Support	Application Domain
ONOS	Java	Web Based	Good	High	D	Linux, MAC OS, And Windows	OF1.0, 1.3, NETCONF	REST API	ON.LAB, AT&T, Ciena, Cisco, Ericsson, Fujitsu, Huawei, Intel, Nec, Nsf.Ntt Comunnication, Sk Telecom	Y	N	Datacenter, WAN and Transport
Open-Day-Light	Java	Web Based	Very Good	High	D	Linux, MAC OS, And Windows	OF1.0, 1.3, 1.4, NET-CONF/ YANG, OVSDB, PCEP, BGP/LS, LISP, SNMP	REST API	Linux Foundation With Memberships Covering Over 40 Companies, Such As Cisco, IBM, NEC	Y	Y	Datacenter
NOX	C++	Python + QT4	Poor	Low	C	Most Supported On Linux	OF 1.0	REST API	Nicira	NOX _MT	N	Campus
POX	Pyth on	Python + QT4	Poor	Low	C	Linux, MAC OS, And	OF 1.0	REST API	Nicira	N	N	Campus
RYU	Pyth on	Yes	Fair	Fair	C	Most Supported On Linux	OF 1.0, 1.2, 1.3, 1.4, NETCONF, OF-CONFIG	REST For Southb ound	Nippo Telegraph And Telephone Corporation	Y	Y	Campus
Beacon	Java	Web Based	Fair	Fair	C	Linux, MAC OS, And Windows	OF 1.0	REST API	Stanford University	Y	N	Research
Maestro	Java	-	Poor	Fair	C	Linux, MAC OS, And Windows	OF 1.0	REST API	RICE, NSF	Y	N	Research
Flood-Light	Java	Web/ Java Based	Good	Fair	C	Linux, MAC OS, And Windows	OF 1.0 , 1.3	REST API	Big Switch Networks	Y	N	Campus

SDN controllers are software, written in several programming languages like Java, Python, and C++, and are capable of controlling hardware from various vendors operating in the same physical architecture. Some SDN controllers (Salman, Elhajj, Kayssi, & Chehab, 2016) include Floodlight, ONOS, RYU, POX, NOX, OpenDayLight, etc. Several studies and surveys have been carried out to compare these SDN controllers in terms of their supported platform, southbound and northbound APIs compatibility, multithreading support, application domain, etc.

3.1.2.3. Application layer

The application layer is logically located above the control layer. It is the topmost layer in an SDN architecture. The northbound interface between this layer and the control layer is used to communicate the policy requirements of applications to the SDN controller. For example; a QoS application may require Voice over IP (VoIP) traffic to be delivered within a specific time frame, and a security application may need to redirect all the traffic from an infected host to a remediation server. The SDN controller implements these policies by writing flow table rules that are used for traffic engineering by switches at the infrastructure layer.

Some fundamental problems may arise during the deployment of a new application on the network. Since each application has its own objectives, conflicts may occur between applications over certain changes to the shared network resources. These conflicts include powering on/off network resources, competition over limited flow table entries of switches, or network bandwidth scheduling and allocation. As a result, optimal resource allocation techniques, policy management schemes, and conflict resolution (AuYoung et al., 2014) are hot research topics in SDN. Vendors such as Cisco, HP, Brocade, etc., provide out-of-the-box SDN applications to small and large enterprises, network operators, and data centers to perform tasks like network monitoring, network security, network management and configuration, network troubleshooting, etc.

3.1.3. Optimization benefits of SDN

Advantages of SDN implementation include:

- *Programmability:* Network control in SDN is directly programmable since the control plane is decoupled from the routing devices in the data plane. With programmability, network automation can be introduced to rapidly reconfigure and optimize the Quality of Experience (QoE) of network services like VoIP calls.

- *Agility and flexibility:* Abstracting the forwarding functions from control intelligence enables network administrators to dynamically adjust traffic flow in the entire network to meet changing needs. This increases the agility and flexibility of the network since the network depends on software installed on commodity servers to dictate network policies.
- *Centralized management:* Logically centralized software-based SDN controllers in the control plane provide the network intelligence and global oversight of the network. Forwarding devices in the data plane are simply referred to as "dumb" switches. The SDN controller may represent the entire network as a single, logical switch in order to reduce the management complexity of large complex networks.
- *Vendor-neutral and open standardization:* SDN architectural solutions simplifies network design, deployment, and operations since policing is implemented through open standard SDN controllers instead of a collection of vendor-specific protocols and devices.

Other benefits of SDN include support for virtualization and big data, effective QoS delivery, and centralized security.

3.1.4. Comparison between SDN and legacy networks

In traditional networks, both the data plane and the control plane were encapsulated together in the same network device as shown in figure 3.6. In traditional networks, each switch updates its own MAC address tables independently, there is no centralized network control and visibility. Furthermore, network management difficulties increase with the growth of independent distributed devices.

On the other hand, SDN is highly advantageous compared to traditional networking because decoupling the networking intelligence from the forwarding devices centralizes network control and visibility.

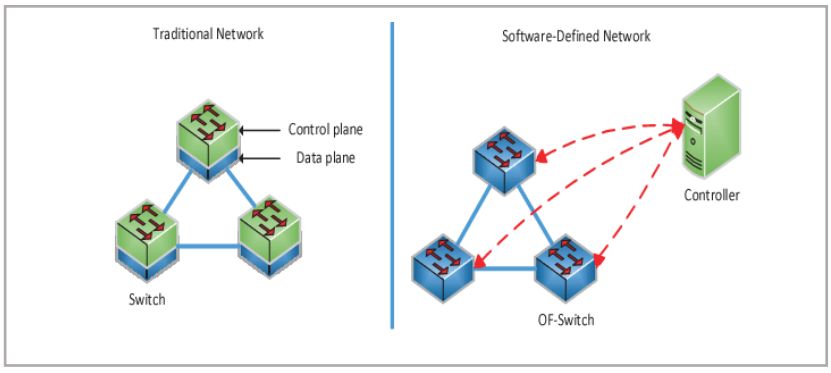


Figure 3.6. *Traditional network vs SDN (Maleki, Hossain, Georges, Rondeau, & Divoux, 2017)*

In addition, the programmability of SDN simplifies network security, management, automation, reconfiguration, innovation, and optimization of large-scale complex networks, like data centers and the cognitive core networks.

3.2. Optimization via Network Function Virtualization (NFV)

This section presents a background on NFV, NFV architecture, and explains the optimization benefits of NFV. Finally, this section will conclude with a comparison between SDN and NFV optimization solutions.

3.2.1. Background on Network Function Virtualization (NFV)

Currently, NFV is significantly attracting attention from both academia and industry as an important shift in network service provisioning. Fundamentally, NFV decouples network Functions (NFs) from the hardware devices on which they run (ETSI, 2014a). By so doing, NFV has the potential to significantly reduce the Operation Expenses (OPEX) and Capital Expenses (CAPEX) of the service providers and facilitates innovation and deployment of new networking services with increased flexibility and agility.

The European Telecommunications Standards Institute Group Specification (ETSI GS) (ETSI, 2018) defines NFV as a set of principles to separate network functions (i.e., functional blocks within the network infrastructure with well-defined external interfaces and functional behaviors) from the hardware resources on which they run by applying virtual hardware abstraction.

Figure 3.7 shows a traditional deployment of large dedicated vendor-proprietary devices by service providers to provide typical network functions like firewalling, server load balancing, and network security tools.

The traditional deployment approach in Figure 3.7 creates a large, complex network environment which is difficult to manage, operate, expensive to maintain, and typically contains underutilized hardware resources. Furthermore, network automation, orchestration, and evolvability is very difficult to implement on proprietary-based equipment with vendor-specific Application Programming interfaces (APIs).

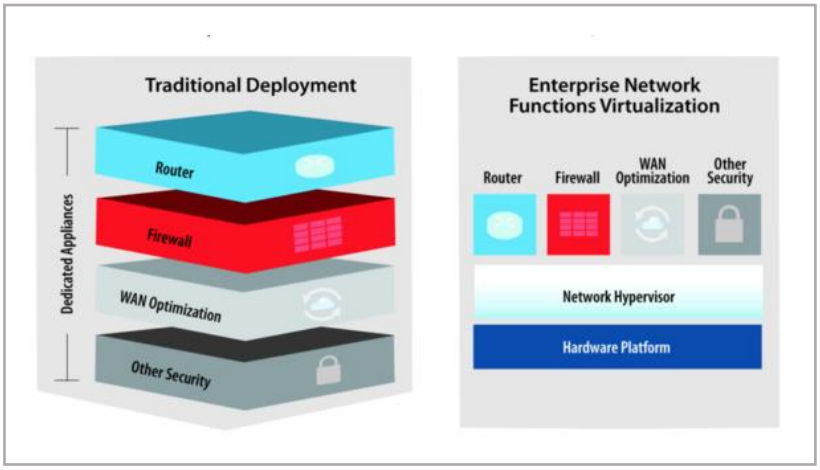


Figure 3.7. *Traditional vs Network Function Virtualization (NFV) deployments*

In order to maximize resource utilization and reduce network complexity, NFV was proposed to break down complex network functions running on specialized vendor hardware into small functional and manageable units which can be dynamically orchestrated to form a homogenous virtualized network ecosystem, (see Figure 3.7).

The concept and initial work on NFV was developed in October 2012 by a number of world's leading telecommunication service providers during the collaborative authorship of a white paper (NFV_White_Paper, 2012) calling for research action in NFV technology. Since November 2012, ETSI was selected by seven of these providers (namely, BT, Orange, Verizon, AT&T, Deutsche Telekom, Telefonica, and Telecom Italia) to be the host of the industry group specification for NFV called the ETSI GS NFV.

Nevertheless, research (Han, Gopalakrishnan, Ji, & Lee, 2015) has shown that several challenges need to be addressed to implement NFV. These challenges include portability, integration and interoperability of NFV technological solutions with legacy networking, and seamless migration of legacy networks to modern NFV platforms. Similarly, NFV reduces the performance of traditional networks since it increases latencies, and processing overheads. Moreover, the management, automation, stability, and security of NFV platforms are very complex and difficult.

The architectural framework of NFV is described below:

3.2.2. NFV architectural framework

The NFV paradigm has the potential to lead rapid service innovation and deployment through software-based services, improve operational efficiency with automation, improve capital efficiency by using general purpose hardware to provide specific

network functions, and improve network flexibility by assigning virtual network functions to hardware according to users' demands and QoS requirements. These network virtualization techniques are also leveraged in cloud computing services (Rao Battula, 2014).

The NFV architectural framework introduces a new way service provisioning is realized in the network. NFV decouples software from hardware, introduces dynamic network operations and network function deployment flexibility.

According to ESTI specifications, (ETSI, 2014c) the NFV architectural framework shown in Figure 3.8 is made up of three key components: the Network Function Virtualization Infrastructure (NFVI), Virtual Network Functions (VNFs), and NFV Management and Orchestration (NFV MANO). Figure 3.8 shows the NFV architecture.

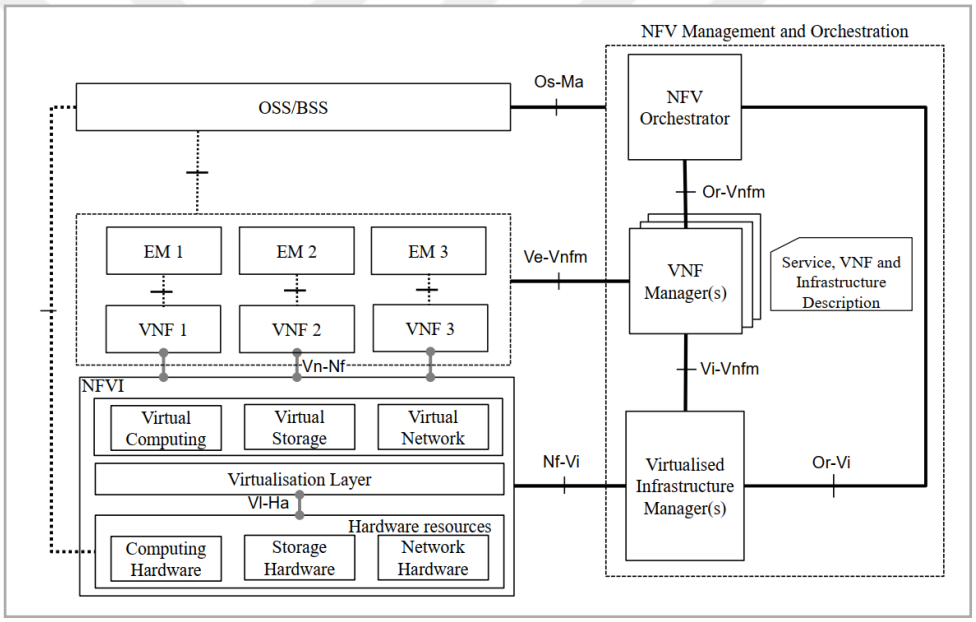


Figure 3.8. *Network Function Virtualization (NFV) architecture (ETSI, 2014b)*

3.2.2.1. Network Function Virtualization Infrastructure (NFVI)

The NFVI is composed of all hardware and software elements which make up the network environment in which VNFs are deployed, executed and managed. The NFVI can be distributed across several geographical locations. The network links connecting these locations are also considered to be part of the NFVI. The physical hardware resources (storage, computing, and network) in NFVI provide storage, processing, and connectivity. These physical resources can be abstracted to form dynamic virtual resources like virtual storage, virtual network, and virtual computing. Resource

allocation and resource release are controlled and performed by the NFV MANO dynamically to accommodate the consumption of those resources by other network functions or services.

3.2.2.2. Virtualized Network Function (VNF)

In a legacy non-virtualized network, the VNF enables the virtualization of all Network Functions (NFs). Types of NFs that can be virtualized include edge devices, gateway functions, and performance improvement functions (load balancers). Similarly, application optimization functions (cache and video transcoder) and security functions (abnormality detection, firewalls, and intrusion detection systems) can also be virtualized. In NFV, these NFs run as software and can be deployed on one or more virtual machines to provide network services. The NF composition and type of VNF are determined by the functionalities and specifications of the provided services (ETSI, 2017).

3.2.2.3. NFV Management and Orchestration (NFV MANO)

NFV requires new management and orchestration functions to perform network maintenance, administration, operation, and NF provisioning. NFV insulates the network resources from NFs through abstraction and decoupling.

The NFV MANO is made up of three functional blocks (ETSI, 2014c); the NFV Orchestrator (NFVO), VNF Manager (VNFM), and Virtualized Infrastructure Manager (VIM).

The NFV Orchestrator (NFVO) performs two major functions; resource orchestration functions by managing NFVI resources across multiple VIMs and network service orchestration functions by managing the lifecycle of network services.

The VNF Manager (VNFM) manages the lifecycle of VNF instances. Each VNF instance is associated with a VNF Manager. A VNF manager can be assigned to manage a single or multiple VNF instances of the same or different types. VNF instances include software updates and upgrade, VNF configuration and termination, performance and fault measurements, etc.

Finally, the Virtualized Infrastructure Manager (VIM) is responsible for managing and controlling NFVI storage, compute and network resources. Specialized VIM may exist to handle certain types of NFVI resources (e.g., storage-only, compute-only, network-only resources). Other functions of VIM include the allocation, update, and

release NFVI resources, as well as the management and provisioning of virtualized resources.

3.2.3. Optimization benefits of NFV

The benefits of Network Function Virtualization revolve around its three major technological breakthroughs; decoupling software from hardware, flexible network function deployment and dynamic scaling (Mijumbi et al., 2016). Other advantages of NFV include:

- Optimization of network configuration and topology can be performed in near real-time based on actual mobility/traffic patterns and service demands.
- Reduces equipment cost and power consumption. Consolidating hardware equipment increases economies of scale in the IT industry and eliminates the need for dedicated hardware for network applications and services.
- Reduces development cost by providing the ability to simultaneously run test, reference and production traffic on the same physical infrastructure.
- Enables a wide variety of network eco-systems and encourages lower risk innovation from non-corporate organizations like small players and academia.
- Orchestration mechanisms provide automated configuration, installation, scaling-up and scaling-down of the network capacity.
- Enables service providers to deliver customized services (based on geographical locations) and provides network isolation for multiple applications, users, and systems operating on the same physical hardware infrastructure.

3.2.4. Comparison of SDN and NFV concepts

SDN and NFV have many similarities, which makes them more compatible, as they both advocate for network evolution towards open software. Moreover, both automation and virtualization play a vital role in achieving SDN and NFV goals. Therefore, due to the complementarity of SDN and NFV, combining both lead to greater efficiency. Recent research (Duan, Ansari, & Toy, 2016) has shown that SDN can accelerate the deployment of NFV by providing automation and flexibility in configuration setup, connectivity, security operations, and policy control.

However, SDN and NFV offer two different virtualization concepts (A. Jain, Sadagopan, Lohani, & Vutukuru, 2016). In SDN, virtualization is achieved by abstracting resources to particular tenants, whereas in NFV, virtualization aims to abstract NFs from dedicated hardware devices. Because of these differences, SDN

requires the construction of a new network with OpenFlow supported hardware where the control and data planes are separated. Whereas NFV can operate on existing network infrastructures since it can be installed standard servers. Table 3.2 compares SDN and NFV.

Table 3.2. *Comparison of NFV and SDN (Mijumbi et al., 2016)*

Issue	**Software-Defined Networking**	**NFV (telecom networks)**
Concept	Network Intelligence Abstraction	Network Function Abstraction
Promises	Open interface and programmable control	Flexibility, agility and cost reduction
Protocol	OpenFlow	Multiple control protocols (e.g. NETCONF, SNMP)
Leaders	Mostly networking hardware and software vendors	Mostly Telecom network operators
Runs on	Control plane on commodity hardware and data plane on specialized hardware	Commodity switches and servers

3.3. Optimization via Combining SDN and NFV

Although the goals of NFV can be achieved using non-SDN approaches relying on other techniques currently implemented in many data centers, it is more advantageous to implement NFV with SDN techniques. This is because SDN separates the control plane from the data forwarding plane leading to greater efficiency and network simplification. In addition, SDN architectural solutions facilitate network operation, programmability, and maintenance procedures.

Combining NFV and SDN to create virtualized SDN (vSDN) offers the benefits of both SDN and NFV paradigms, i.e., flexible and dynamic resource allocation and acquisition by network tenants through NFV and a standardized method to program and manage those resources through SDN.

Some SDN controllers such as Floodlight, ONOS, and OpenDayLight provide a form of network virtualization by allowing certain network applications to utilize isolated virtual network resources. This is however not considered as full virtualization because the operation of these virtual networks can only be orchestrated by the parent SDN controller which does not allow any other SDN controllers to manage these virtual

networks. Therefore, for Full virtualization, we use a hypervisor to create virtual SDN networks.

In the following section, we explain how hypervisors are used to create vSDN networks. We present the two major hypervisors: Flowvisor and OpenVirteX and their limitations.

3.3.1. Virtualizing SDN networks using hypervisors

Initially, hypervisors were developed for virtual computing to monitor and allocate physical resources (RAMs, CPUs, storage, etc.) to several virtual machines running on the shared computing infrastructure (Sonam Srivastava & S.P Singh, February 2016). In a virtualized SDN architecture, the hypervisor adds an additional abstraction layer called the virtualization layer. Typically, the virtualization layer or hypervisor sits between the networking hardware and multiple virtual SDN controllers (A. A. Blenk, 2018). This layer creates and manages logically isolated network partitions or virtual networks. The virtualization layer reduces the performance of the overall network since it increases overheads and CPU usage. Thus, optimization is needed to minimize performance degradation. Figure 3.9 shows a virtualized SDN architecture.

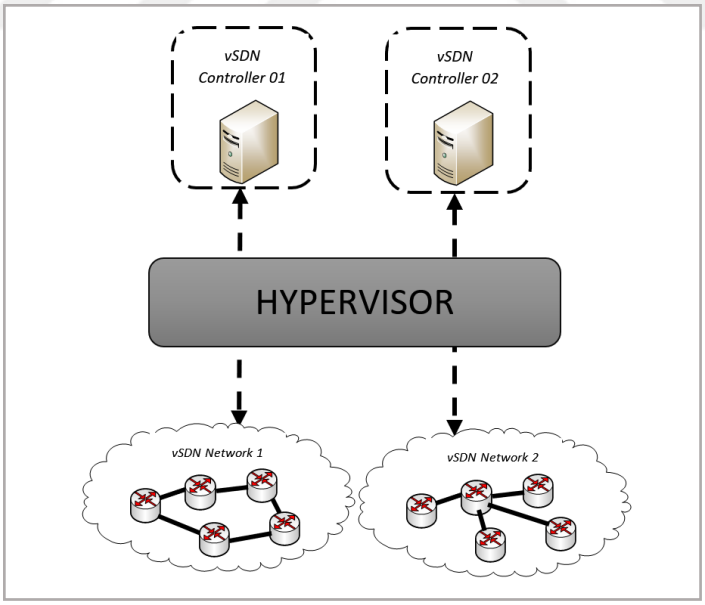


Figure 3.9. *SDN virtualization with hypervisor*

The creation of virtual SDN networks using hypervisors involves 3 steps (A. Blenk, Basta, Reisslein, & Kellerer, 2016): management of the physical SDN network, virtualization of network attributes, and isolation of network attributes.

3.3.1.1. Management of the physical SDN network

The programmability of SDN can be employed to facilitate the implementation of virtual SDN networks. In vSDN architectures, the hypervisor lies between the physical SDN network and the vSDN controller. Due to its intermediate position, the hypervisor interacts and monitors the entire physical SDN infrastructure.

The hypervisor creates isolated virtual SDN networks by abstracting the physical SDN network resources. Packet forwarding and network policing in these virtual SDN network are controlled by virtual SDN controllers. Figure 3.9 illustrates how the underlying physical SDN network can be abstracted to produce the two separate vSDN networks. Non-transparent hypervisor acts as a proxy by intercepting control messages between the physical SDN network and the vSDN controllers.

3.3.1.2. Virtualization of network attributes

The Hypervisor communicates an abstraction (simplified representation) of the physical SDN network to the virtual SDN controllers. There are three types of SDN network abstraction, namely topology, physical node resources, and physical link resources abstractions.

Topology abstraction involves the abstraction of topology, virtual nodes, and virtual links information. This information (nodes location and links interconnection) is transmitted by the hypervisor to the tenants (vSDN controller). The entire network topology is always hidden from the tenants. As such, multiple physical switches can be represented by a single virtual switch. This is known as a big switch.

Physical node resource abstraction involves the abstraction of CPU and memory resource information. CPU resource information can be represented by the percentage utilization of the CPU or the number of CPU cores available. Consequently, 150% CPU availability equals to one and a half available cores in a dual core physical CPU. Similarly, Memory resource information like flow tables may be abstracted using memory partition representations or the number of hardware or software flow table. Based on the degree of abstraction, the hypervisor abstracts this information from the tenants' view reducing the network complexity.

Physical link resource abstraction involves the abstraction of link buffers, number of queues, queuing priorities as well as bandwidth information (Chowdhury & Boutaba, 2010). The hypervisor abstracts this information to create virtual networks with specific loss or delay bounds to guarantee effective service delivery.

3.3.1.3. Isolation of network attributes

The hypervisor must provide tenants with isolated virtual networks that efficiently share the same physical infrastructure. Isolated physical resources include data plane (nodes, links), control plane (instances), and vSDN addressing isolation.

Data Plane Isolation involves the isolation and reservation of data plane physical resources like link data rates, link queues, flow table spaces, and CPU nodes. These resources can be isolated to enhance data plane traffic processing between tenants. Performance variations must be considered when allocating physical resources since these variations constantly change due to different network workloads and scenarios. Optimizing the isolation process can prevent a single vSDN network from starving other virtual networks of physical resources.

Control Plane Isolation involves the isolation of control plane elements from the tenants. High CPU and memory usage by the vSDN controllers in the control plane can significantly degrade the performance of data plane processes. As a result, the switches at the data plane might experience forwarding delays. Therefore, only one vSDN controller controls a single vSDN network to prevent vSDN controller interferences. Furthermore, CPU, storage and network resources used by the hypervisor must also be isolated to provide efficient operation of the hypervisor.

Finally, vSDN addressing isolation involves isolating the different tenants' forwarding decisions to avoid conflicts. Tenants should have the freedom to direct flows according to their service demands and configurations. Generally, the attributes of the physical infrastructure limits flow space addressing in non-vSDN networks. Whereas in virtualized SDN networks, several addressing approaches can be used. One addressing technique called the flow space splitting technique suggests the provision of non-overlapping flow spaces to tenants. Another addressing technique uses the fields not required by the OF protocol to provide unique vSDN identification and addressing. This technique allows the tenants to have the entire flow space.

3.3.2. Types of hypervisor architecture

Hypervisor architectures can be classified into centralized and distributed hypervisors (A. Blenk et al., 2016). A centralized hypervisor architecture has a single central entity which controls multiple network elements in the underlying physical network infrastructure. Data centers employing this type of hypervisors have them installed on a virtual machine on a standard server. On the other hand, a distributed hypervisor

architecture logically separates its virtualization functions across multiple network elements.

3.3.2.1. Flowvisor (FV)

FV (Rob Sherwood et al., 2009) was the first OF-based hypervisor used to virtualize, slice and share SDN resources between multiple vSDN controllers. FV was developed to isolate production network traffic from experimental network traffic allowing both experimental and production networks to share the same physical SDN infrastructure.

The architecture and operation of FV are described below:

- *Architecture:* FV is a centralized network hypervisor. It is a pure software which can run in a virtual machine or on any standard server. FV occupies the virtualization layer and controls networking traffic between the physical SDN hardware and the vSDN controllers. FV abstracts the SDN hardware resources from SDN controllers by controlling their respective network resource views. FV supports OF 1.0 (ONF, December 31, 2009).
- *Flowspace:* FV uses the term *flowspace* to describe a non-contiguous sub-space in the header field space. FV allocates a unique flowspace to every vSDN tenant to ensure that the tenants are isolated from one another with non-overlapping flowspaces. FV intercepts traffic between the tenants and the physical SDN hardware to rewrite packet headers or generate OF error messages every time tenants uses the same flowspaces for flow addressing.
- *Topology Isolation:* FV creates network slices by isolating the network topology and allows each SDN controller to view only the switches and ports under their control i.e., their slices, (see Figure 3.11). For this, OF messages from the tenants are rewritten and forwarded by the FV to their respective slices and vice versa.
- *Bandwidth Isolation:* Even though a QoS mechanism for data pane isolation is not specified in the OF 1.0 version, FV uses priority bits in the data packet to provide data plane isolation in VLAN. These priority bits (3-bits) are used to set and map data packets to eight distinct priority queues.
- *Flow Entries Isolation:* Flow entries are limited resources. Therefore, FV monitors the use of flowspaces, isolates flow entries from different network slices and prevents vSDN controllers from using the flowspace of other vSDN controllers when their flowspaces are fully utilized.

- *Control Channel Isolation:* FV uses unique transaction identifiers for each tenant in order to distinguish between network slices. FV equally modifies and rewrites message identifiers whenever vSDN controllers use identical OF identifiers. Figure 3.10 shows network slicing with FV.

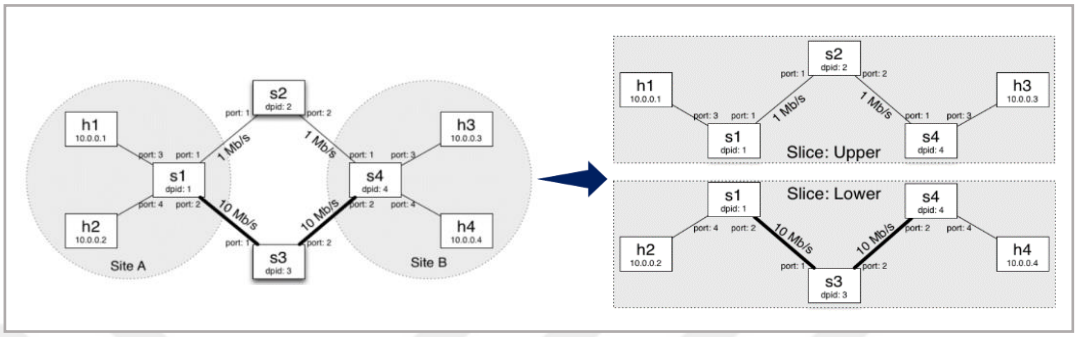


Figure 3.10. *Network slicing with Flowvisor (Rob Sherwood et al., 2009)*

Nevertheless, FV experiences a flowspace problem because the same address space is shared (sliced) between all tenant vSDN controllers. As a result, the flowspace of tenants is very short and frequently overlaps.

3.3.2.2. OpenVirteX (OVX)

OVX (Ali Al-Shabibi et al., 2014) is a decentralized hypervisor built on the concept of FV. OVX uses the physical SDN hardware to create virtualized SDN functionalities. OVX provides topology virtualization and address virtualization. The Address virtualization is used by OVX to tackle the flowspace problem in FV by providing full header fields spaces to individual virtual SDN controllers.

The architecture and operation of OVX are described below:

- *Architecture:* OVX is a pure software-based hypervisor that supports OF 1.0 (ONF, December 31, 2009) and Oracle Java Version 7 (James Gosling, Bill Joy, Guy Steele, Gilad Bracha, & Alex Buckley, 2013). OVX can be installed on any computing platform or standard server in a data center. OVX creates virtual networks that are more resilient to node and link failures by rewriting the MAC addresses of physical SDN links and switches. This allows the same network infrastructure to be used simultaneously by multiple tenants. Figure 3.11 shows the OVX system architecture.
- *Address Isolation:* Virtualization in OVX is better than that in FV because OVX rewrites the tenants' IP addresses and the MAC addresses of physical SDN switches

to identify vSDN tenants instead of using packet header fields as FV does. This process increases the available flowspace for tenants.

- *Topology Abstraction:* OVX is not a transparent hypervisor because it intercepts and answers to the link layer discovery protocol messages and other topology discovery processes. This prevents the tenant vSDN controllers from viewing the entire physical underlying network. OVX can create "big switches" by merging multiple physical switches together.

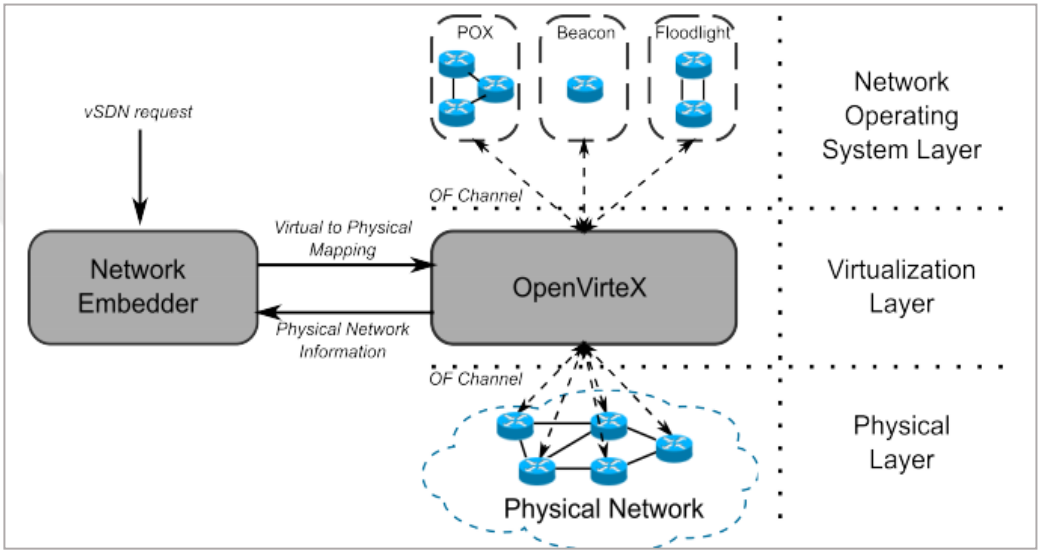


Figure 3.11. *OpenVirteX system architecture (Al-Shabibi et al., 2014)*

Comparing OVX and FV (Al-Shabibi et al., 2014) shows that OVX performs better and delivers a more robust SDN virtualization. However, virtual network creation and configuration with OVX is very difficult and requires manual configurations from the network administrator. For this reason, we proposed automatic virtual network configuration module called AUTOVNET to rapidly create and configure virtual networks using OVX.

Section 4 provides more details on the drawbacks of OVX and explains how AUTOVNET is used to automate the creation and configuration of virtual SDN networks using OVX.

3.4. Review of available vSDN solutions with OpenFlow

Several network virtualization solutions have been implemented with OF (Ahmed Abdelaziz et al., 2016). In this section, we review some of these implementations and their limitations.

VeRTIGO proposed by R. Doriguzzi et al. (Corin, Gerola, Riggio, Pellegrini, & Salvadori, 2012) applied proxy virtualization to facilitate the design of NV networking platform with OpenFlow. VeRTIGO improved on Flowvisor to provide customized virtual topologies where different network resources could be allocated to tenants depending on their service level requirements (packet loss or maximum latency). VeRTIGO also allowed customers to either select between topology customization or routing policing. However, VeRTIGO has the same limitation as Flowvisor, and could not provide a full isolated address space to each tenant.

In (Jin et al., 2017), the authors used OVX to virtualize optical networks based on bandwidth availability. They proposed an Optical-OpenVirteX (O-OVX) architecture consisting of two main modules, Topology Discovery (TD) and Bandwidth on Demand (BoD) modules. The TD module collects information (optical devices, ports and links) from the physical network, whereas the BoD module maps the available network resources and performs virtual network resource allocation. However, this design is difficult to implement since it uses the Network Embedder module of OVX to receive vSDN requests from tenants.

Similarly, W. Jeong and al. in (Jeong, Yang, Kim, & Yoo, 2017) used OVX to design an efficient Big Link Allocation Scheme (BAS) for vSDN. A Big link in vSDN is a single virtual link created by mapping several switches and links. BAS reduces unnecessary reallocation of resources, provides big links with a greater throughput and mitigates vSDN performance degradation. However, BAS lacked proper fault detection capabilities.

The configuration of virtual networks using OVX is very complex and tedious due to lack of automation. This is because the network administrator has to configure the virtual networks manually. Motivated by the limitations of the existing research in the literature, we propose AUTOVNET which replaces the Network Embedder of OVX (Al-Shabibi et al., 2014) and automatically creates and configures virtual networks from a simplified vSDN request. In addition, AUTOVNET has fault-detection features to ensure that virtual networks are created and configured using faultless physical resources.

4. AUTOVNET - AUTONOMOUS VIRTUAL NETWORK SYSTEM

In this chapter, we propose an autonomous virtual network management system called AUTOVNET, describe its implementation and specify its advantages over the legacy OVX system. AUTOVNET uses the OVX hypervisor to configure virtual SDN networks. OVX is pure software implemented in Python and has two main advantages in NV, namely, address virtualization and topology virtualization (see Sec. 3.3.2.2).

Traditional OVX requires manual inputs from the user or network administrator to configure virtual networks on a given underlying network topology. This is the main reason setting up even the smallest virtual networks using OVX takes a lot of efforts. As a result, OVX developers designed the network embedder module (see Figure 3.11) to map the virtual topology to physical resources and rapidly create virtual networks based on the user requests.

Nevertheless, the network embedder module did not solve the problem of speeding up virtual network configurations for larger network topologies, since it equally required manual configurations from the network administrator. Therefore, larger network architectures required more complex manual configurations (vSDN request). This configuration instruction is essentially a script that specifies the MAC addresses and port information of switches and hosts and how these resources interconnect (link or node mapping information). Consequently, configuring virtual networks on large complex networks using OVX required large complex vSDN request scripts which are time-consuming and challenging to code.

AUTOVNET introduces a novel approach to solve these problems by automating the virtual network configuration process and minimizes manual instructions. The vSDN request required by AUTOVNET is extremely simplified, and regardless of the topology of the underlying network, the network administrator must only specify the MAC addresses of the source host and the destination host in the script. With AUTOVNET, the network administrator does not need to know or indicate the MAC addresses of the edge switches (switches to which the source and destination hosts are directly connected), intermediate switches (switches connecting the source and destination hosts), and port information. AUTOVNET automatically fetches this information.

4.1. AUTOVNET Management Functions

AUTOVNET is an automatic virtual network management and orchestration system that is attached to the OVX architecture (Figure 3.11) to replace the network embedder module of OVX.

Amongst the five management areas of cognitive networks described in Sec. 2.3, AUTOVNET can perform three of them, namely configuration management, fault management, and performance management.

4.1.1. AUTOVNET: configuration management

AUTOVNET automatically monitors the underlying physical network and collects information like MAC addresses of switches and hosts, port numbers, and link information of the entire network. AUTOVNET uses the vSDN request from the network administrator to create and configure an OVX "Big Switch". A "Big Switch" is a single virtual switch that is created from multiple physical switches. In other words, the tenant views a single switch representing the underlying physical network. This technique reduces the configuration complexity of virtual networks and creates redundant network links within the virtual switch. Moreover, AUTOVNET assigns a controller to manage and control the virtual network.

4.1.2. AUTOVNET: fault management

AUTOVNET differs from other virtualization modules described in Sec. 3.4 because it has a built-in fault detection mechanism. Traditionally, OVX abstracts physical network resources to create virtual networks without checking whether they might be working properly or not. As a result, if we abstract a broken physical link or damaged switch to create our virtual node, OVX will create a faulty virtual node. Consequently, this virtual network will not be able to transmit data traffic between the hosts. To avoid this type of situations, AUTOVNET performs a pre-virtualization fault analysis to ensure that all switches and links are functioning properly and lists all potential faulty resources to avoid using them to create virtual networks.

4.1.3. AUTOVNET: performance management

AUTOVNET uses real-time network data (packet delay, number of packets on links) to perform congestion analysis. This analysis is used to identify congested routes in the underlying physical network. After this analysis, AUTOVNET uses resources from the least congested route to create the virtual network. It should be noted that a *"route"* is made up the switches connecting the source host to the destination host. Congestion

analysis is particularly useful in large complex networks where multiple routes connect the source and destination hosts. Having the ability to determine which route is the best will provide a great advantage and AUTOVNET accomplishes this task.

4.2. AUTOVNET Architecture

AUTOVNET simplifies the configuration of virtual networks using OpenVirteX (OVX). Figure 4.1 shows the architecture of AUTOVNET. AUTOVNET is an automatic virtual network management and orchestration system that is attached to the OVX architecture in Figure 3.12 to replace the network embedder module of OVX.

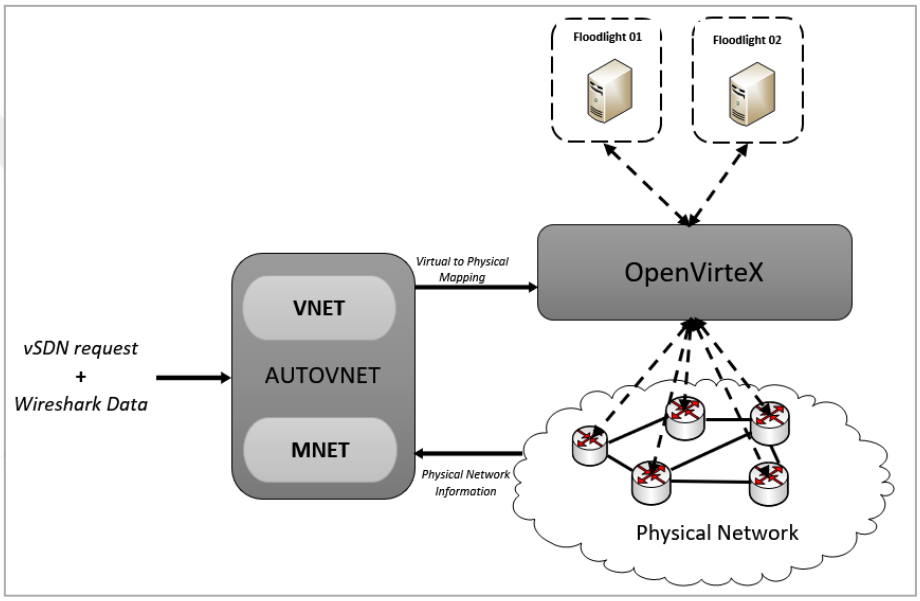


Figure 4.1. *AUTOVNET system architecture*

AUTOVNET is pure software implemented in Python programming. It has two main components: MNET and VNET.

4.2.1. MNET module

The MNET module is directly connected to the physical network. It monitors the underlying hardware, collects, updates and stores network information in the Network Information Database (NID). This database is a repository containing information about all hardware resources and their features, i.e., MAC addresses and port attachment numbers of all hosts and their corresponding edge switches, switch-to-switch connection information and bandwidth capacity of the links. The MNET module can receive real-time Wireshark data to create other databases and performs other analyzes

such as link budget calculation and deep packet inspection to determine potential congested links and failing hardware resources.

4.2.2. VNET module

The VNET receives and validates vSDN requests from the network administrator (source and destination host MAC addresses) and uses the NID and other MNET-generated databases to make resource allocation and path selection decisions. It automatically generates a configuration command to instruct OVX to create and configure the appropriate virtual network for the source and destination hosts. The VNET isolates physical resources for the creation of virtual networks, manages the lifecycle and topology of virtual networks, and assigns a specific SDN controller to manage and control routing and forwarding in each virtual network.

Thus, the MNET and VNET components provide AUTOVNET with cognitive capabilities like sensing, adapting, and learning capabilities making AUTOVNET a complete cognitive system.

4.3. AUTOVNET Implementation

An MSI PL627RC Computer with Intel ® Core ™ i7 CPU @ 2.80 GHz, RAM: 8:00 GB, Ubuntu 18.04 as Host Operating System is used as the computing platform in this research. Mininet V2.2 is used to design and test the SDN network topologies. Floodlight (V2.1.2) is used as the SDN Controller. Both AUTOVNET modules (MNET and VNET) are programmed in Python 3.7.

The three major tools used to design and test AUTOVNET, Mininet (to create and test the SDN networks), Floodlight (the SDN controller), and Wireshark (to collect data from the SDN network) are described in the following subsections.

4.3.1. Mininet and network topology design

Mininet is an SDN emulator software. Emulators (Wang, 2014) are commonly used to recreate or replicate the behavior of real networks. This is very useful for research, testing, learning, development, debugging, and prototyping SDN networks on a single computer. Mininet supports OpenFlow and uses a process-based virtualization method to create a network of virtual controllers, switches (Open vSwitch), hosts and links. It runs on standard Linux network software and allows network engineers to use Python scripts to create, design, and test custom topologies. Figure 4.2 shows the Fat Tree topology used for our analysis in this research.

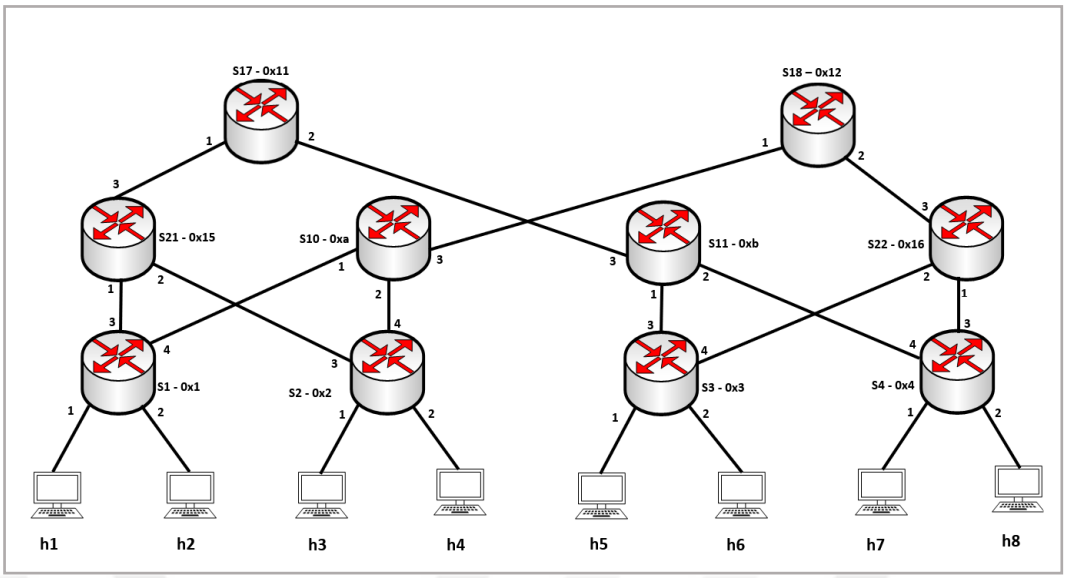


Figure 4.2: *Fat tree topology*

Table 4.1. *MAC addresses of the switches and hosts*

Network Element	***Mac Address***	***Description***
S17	00:00:00:00:00:00:00:11	Core Switch
S18	00:00:00:00:00:00:00:12	Core Switch
S1	00:00:00:00:00:00:00:01	Edge Switch
S2	00:00:00:00:00:00:00:02	Edge Switch
S3	00:00:00:00:00:00:00:03	Edge Switch
S4	00:00:00:00:00:00:00:04	Edge Switch
S21	00:00:00:00:00:00:00:15	Switch
S10	00:00:00:00:00:00:00:0a	Switch
S11	00:00:00:00:00:00:00:0b	Switch
S22	00:00:00:00:00:00:00:16	Switch
h1	00:00:00:00:00:01	Host
h2	00:00:00:00:00:02	Host
h3	00:00:00:00:00:03	Host
h4	00:00:00:00:00:04	Host
h5	00:00:00:00:00:05	Host
h6	00:00:00:00:00:06	Host
h7	00:00:00:00:00:07	Host
h8	00:00:00:00:00:08	Host

This topology contains 8 hosts, 10 switches, and 20 links. Table 4.1 shows the MAC addresses of the hosts and switches in the network. Each host is directly connected to a single switch called the edge switch.

This network topology provides two routes through which the hosts can communicate with one another. This double routing option is very important because if one of the routes is broken, the other can be used as backup. For this reason, this type of network topology is often used to create fast, fault-tolerant data centers (Lebiednik, Mangal, & Tiwari, 2016)

Similarly, this topology is used in this analysis to test AUTOVNET in a faulty network scenario, where one route is faulty (contains broken links or inactive switches). In this case, AUTOVNET should detect the faulty route and use the other backup route to create the virtual network.

4.3.2. Floodlight

The Floodlight SDN controller was used in our analysis. Table 3.1 shows how Floodlight compares to other SDN controllers like OpenDayLight, ONOS, NOX, etc.

Floodlight is an open source OpenFlow controller developed by Big Switch Networks. An active developer community, very good documentation, and minimal dependencies make this controller easy to build, update, run and use. Floodlight is written in the Java programming language and licensed by Apache. Floodlight can manage both OpenFlow and non-OpenFlow networks and supports cloud orchestration platforms.

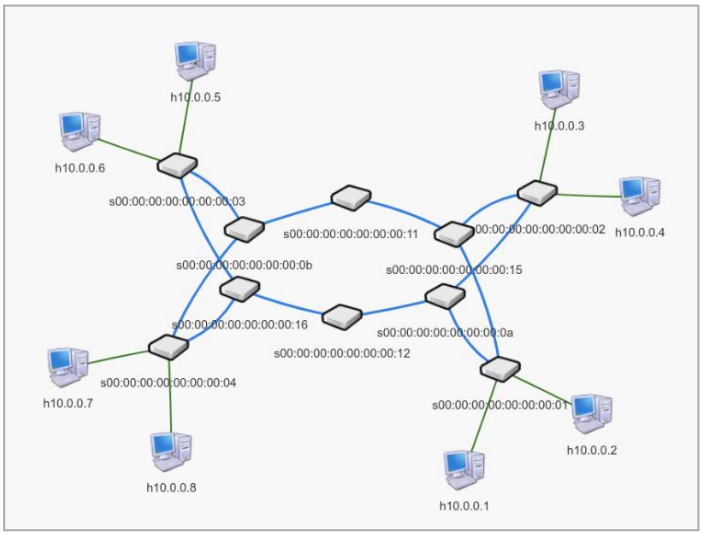


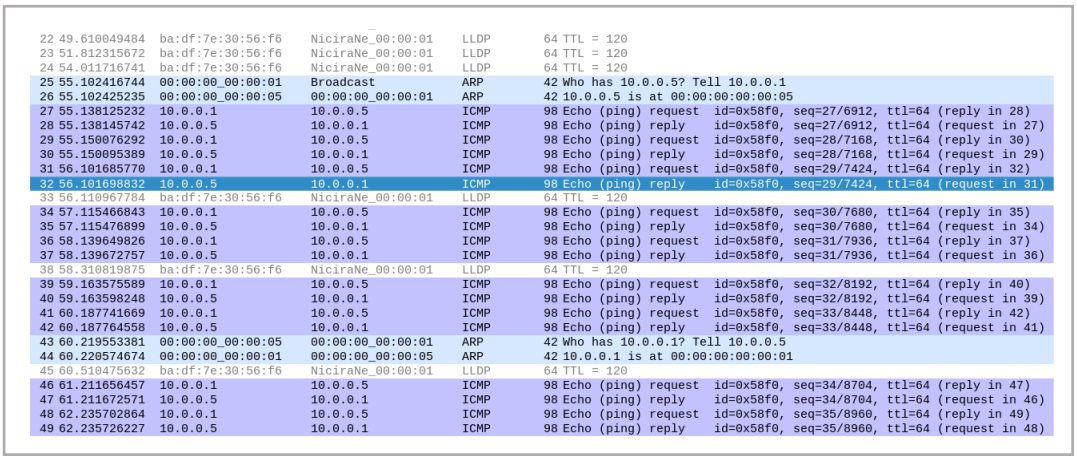
Figure 4.3. *Network topology (fat tree) viewed by Floodlight*

The Floodlight has a Web GUI which allows network engineers to view the overall network topology, all network resources (switches, hosts), switch flow tables, inter-switch links, and controller state information. The Web GUI also provides some OpenFlow statistics in an easy to read tabular format that can be reached from *http://<controller-ip>:8080/ui/index.html*. Figure 4.3 shows the fat tree topology in Figure 4.2 represented in the Floodlight Web GUI.

4.3.3. Wireshark

AUTOVNET can analyze data collected by Wireshark to carry out congestion analysis and deep packet inspections. These analyses are used for load balancing, effective resource utilization, and best route selection for virtualization.

Wireshark is the most popular open source network packet analyzer (Banerjee, Ashutosh, & Mukul, 2010). It is used by network professionals as a packet sniffing and logging tool. Wireshark can capture hundreds of communication protocols and as a result, it is mostly used for network troubleshooting, communication software, and protocol developments. A GUI is used to view, browse, and analyze network data. Wireshark can be installed on most computing platforms like Linux, Windows, and UNIX. Figure 4.4 shows captured network data in the Wireshark GUI.



22	49.610049484	ba:df:7e:30:56:f6	NiciraNe_00:00:01	LLDP	64 TTL = 120
23	51.812315672	ba:df:7e:30:56:f6	NiciraNe_00:00:01	LLDP	64 TTL = 120
24	54.011716741	ba:df:7e:30:56:f6	NiciraNe_00:00:01	LLDP	64 TTL = 120
25	55.102416744	00:00:00_00:00:01	Broadcast	ARP	42 Who has 10.0.0.5? Tell 10.0.0.1
26	55.102425235	00:00:00_00:00:05	00:00:00_00:00:01	ARP	42 10.0.0.5 is at 00:00:00:00:00:05
27	55.138125232	10.0.0.1	10.0.0.5	ICMP	98 Echo (ping) request id=0x58f0, seq=27/6912, ttl=64 (reply in 28)
28	55.138145742	10.0.0.5	10.0.0.1	ICMP	98 Echo (ping) reply id=0x58f0, seq=27/6912, ttl=64 (request in 27)
29	55.150076292	10.0.0.1	10.0.0.5	ICMP	98 Echo (ping) request id=0x58f0, seq=28/7168, ttl=64 (reply in 30)
30	55.150095389	10.0.0.5	10.0.0.1	ICMP	98 Echo (ping) reply id=0x58f0, seq=28/7168, ttl=64 (request in 29)
31	56.101685770	10.0.0.1	10.0.0.5	ICMP	98 Echo (ping) request id=0x58f0, seq=29/7424, ttl=64 (reply in 32)
32	56.101698832	10.0.0.5	10.0.0.1	ICMP	98 Echo (ping) reply id=0x58f0, seq=29/7424, ttl=64 (request in 31)
33	56.110967784	ba:df:7e:30:56:f6	NiciraNe_00:00:01	LLDP	64 TTL = 120
34	57.115466843	10.0.0.1	10.0.0.5	ICMP	98 Echo (ping) request id=0x58f0, seq=30/7680, ttl=64 (reply in 35)
35	57.115476899	10.0.0.5	10.0.0.1	ICMP	98 Echo (ping) reply id=0x58f0, seq=30/7680, ttl=64 (request in 34)
36	58.139649826	10.0.0.1	10.0.0.5	ICMP	98 Echo (ping) request id=0x58f0, seq=31/7936, ttl=64 (reply in 37)
37	58.139672757	10.0.0.5	10.0.0.1	ICMP	98 Echo (ping) reply id=0x58f0, seq=31/7936, ttl=64 (request in 36)
38	58.310819875	ba:df:7e:30:56:f6	NiciraNe_00:00:01	LLDP	64 TTL = 120
39	59.163575589	10.0.0.1	10.0.0.5	ICMP	98 Echo (ping) request id=0x58f0, seq=32/8192, ttl=64 (reply in 40)
40	59.163598248	10.0.0.5	10.0.0.1	ICMP	98 Echo (ping) reply id=0x58f0, seq=32/8192, ttl=64 (request in 39)
41	60.187741669	10.0.0.1	10.0.0.5	ICMP	98 Echo (ping) request id=0x58f0, seq=33/8448, ttl=64 (reply in 42)
42	60.187764558	10.0.0.5	10.0.0.1	ICMP	98 Echo (ping) reply id=0x58f0, seq=33/8448, ttl=64 (request in 41)
43	60.219553381	00:00:00_00:00:05	00:00:00_00:00:01	ARP	42 Who has 10.0.0.1? Tell 10.0.0.5
44	60.220574674	00:00:00_00:00:01	00:00:00_00:00:05	ARP	42 10.0.0.1 is at 00:00:00:00:00:01
45	60.510475632	ba:df:7e:30:56:f6	NiciraNe_00:00:01	LLDP	64 TTL = 120
46	61.211656457	10.0.0.1	10.0.0.5	ICMP	98 Echo (ping) request id=0x58f0, seq=34/8704, ttl=64 (reply in 47)
47	61.211672571	10.0.0.5	10.0.0.1	ICMP	98 Echo (ping) reply id=0x58f0, seq=34/8704, ttl=64 (request in 46)
48	62.235702864	10.0.0.1	10.0.0.5	ICMP	98 Echo (ping) request id=0x58f0, seq=35/8960, ttl=64 (reply in 49)
49	62.235726227	10.0.0.5	10.0.0.1	ICMP	98 Echo (ping) reply id=0x58f0, seq=35/8960, ttl=64 (request in 48)

Figure 4.4. *Wireshark GUI with captured network data*

4.4. AUTOVNET Operation

Figure 4.5 shows a flowchart of the AUTOVNET operation.

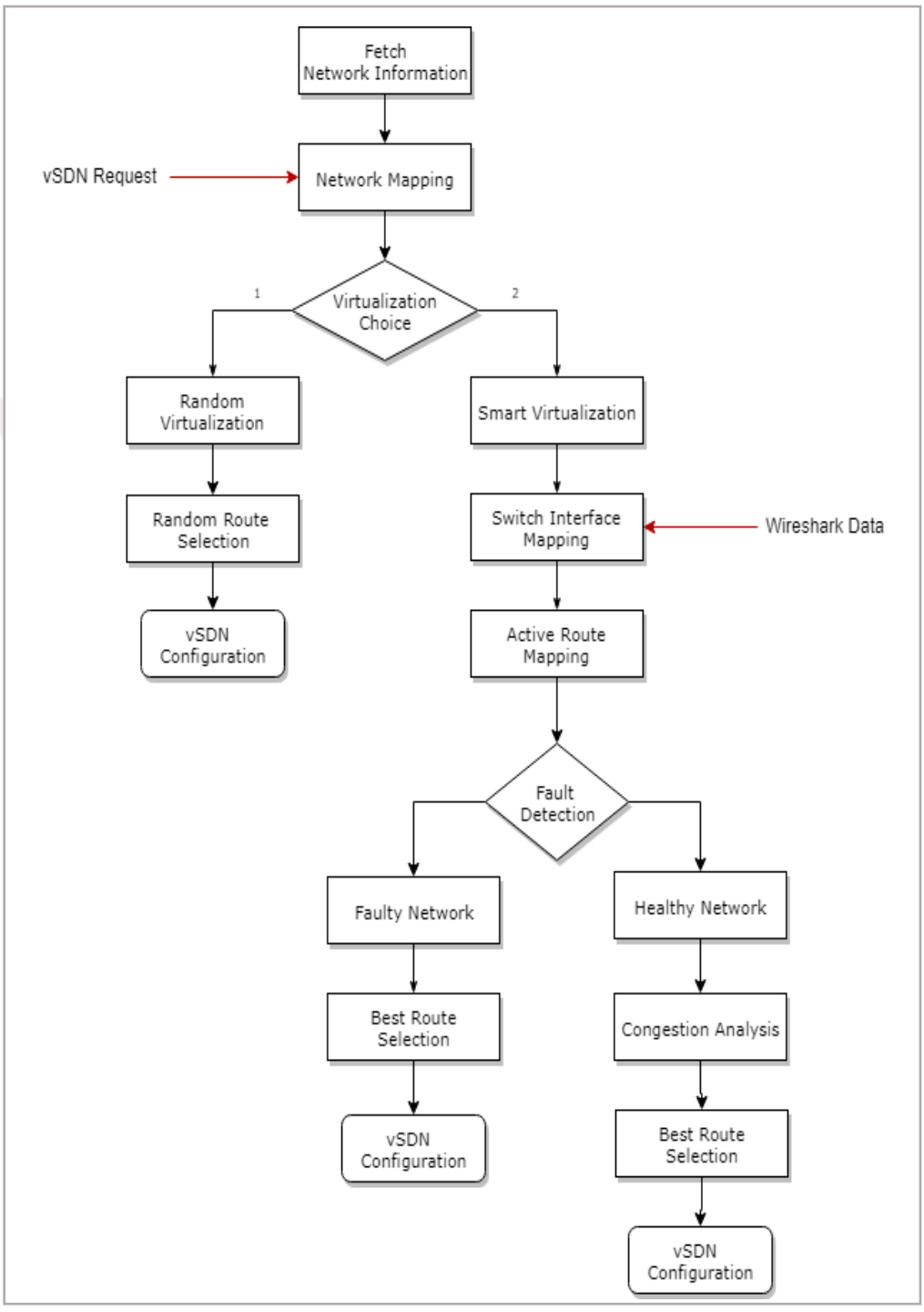
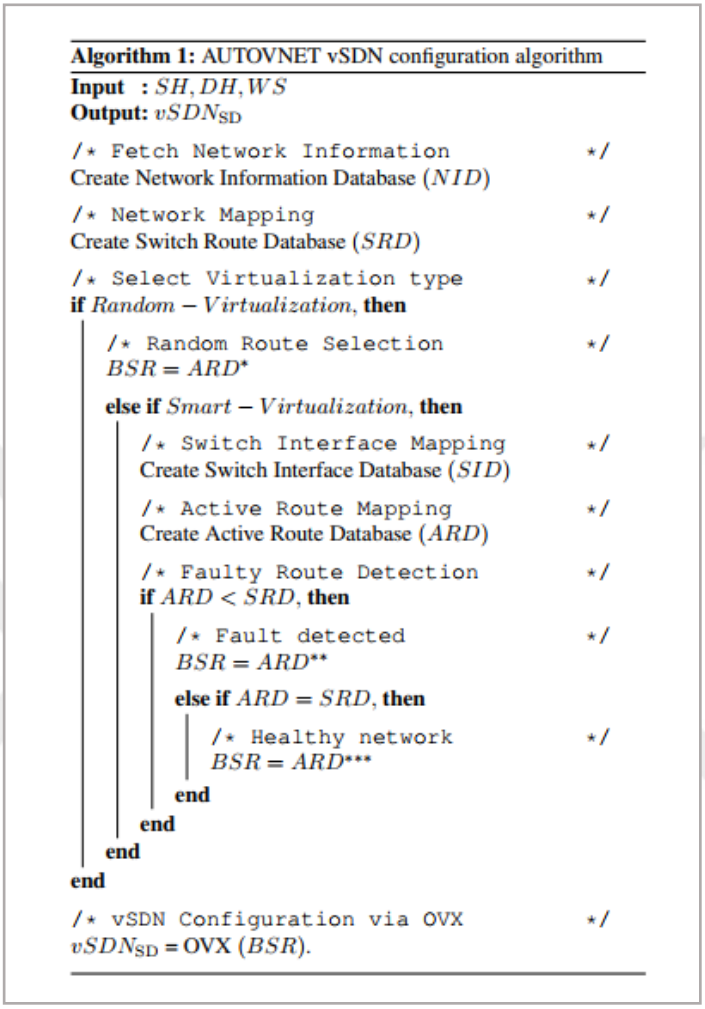


Figure 4.5. *Flowchart of AUTOVNET operation*

Figure 4.6 shows the AUTOVNET vSDN configuration algorithm



Algorithm 1: AUTOVNET vSDN configuration algorithm

Input : SH, DH, WS
Output: $vSDN_{SD}$

/* Fetch Network Information */
Create Network Information Database (NID)

/* Network Mapping */
Create Switch Route Database (SRD)

/* Select Virtualization type */
if $Random - Virtualization$, **then**

  /* Random Route Selection */
  $BSR = ARD^{*}$

  else if $Smart - Virtualization$, **then**

    /* Switch Interface Mapping */
    Create Switch Interface Database (SID)

    /* Active Route Mapping */
    Create Active Route Database (ARD)

    /* Faulty Route Detection */
    if $ARD < SRD$, **then**

      /* Fault detected */
      $BSR = ARD^{**}$

      else if $ARD = SRD$, **then**

        /* Healthy network */
        $BSR = ARD^{***}$

      end

    end

  end

end

/* vSDN Configuration via OVX */
$vSDN_{SD}$ = OVX (BSR).

Figure 4.6. *AUTOVNET vSDN configuration algorithm*

In the following, we discuss the various operations or steps shown in Figure 4.5. These steps are carried out by AUTOVNET to automatically detect faults in the network (e.g., broken links), isolate healthy network resources and create virtual networks.

Since AUTOVNET and OVX are both written in Python, AUTOVNET uses OVX as a Python library to automatically configure and create a virtual network between two hosts in the network,

The AUTOVNET algorithm in Figure 4.6 requires three inputs denoted by SH (source host MAC address), DH (destination host MAC address), and WS (Wireshark data). The vSDN request only indicates SH and DH.

The goal of AUTOVNET is to select the Best Switch Route (BSR) which is the shortest, healthiest, fastest, and least congested route in the network. The BSR contains switches which are used to create a virtual network between the SH and DH using OVX. The output of AUTOVNET is denoted by $vSDN_{SD}$, which represents the virtual network created by OVX for SH and DH communications.

Mathematically, this output is given by

$$vSDN_{SD} = OVX(BSR). \tag{4.1}$$

The following steps describe the operation of AUTOVNET.

- **Fetch network information**

During the process known as sensing, AUTOVNET automatically monitors and fetches network data from the underlying physical resources via the link layer discovery protocol to create the Network Information Database (NID).

The fat tree topology in Figure 4.2 is used as the underlying network topology in this study. Figure 4.7 shows a screenshot the NID when AUTOVNET is executed in Spyder IDE. The NID contains information on the number of switches and hosts in the network, as well as the MAC addresses of the hosts and their corresponding edge switches, switch-to-switch connections and port attachment numbers.

Note that in this section, the operation of AUTOVNET is illustrated using screenshots from Spyder IDE.

```
number of switches:  10
number of Hosts:  8


[['host mac_address' '  Connected Switch ' '  Port']
 ['00:00:00:00:00:02' '00:00:00:00:00:00:00:01' '2']
 ['00:00:00:00:00:04' '00:00:00:00:00:00:00:02' '2']
 ['00:00:00:00:00:07' '00:00:00:00:00:00:00:04' '1']
 ['00:00:00:00:00:01' '00:00:00:00:00:00:00:01' '1']
 ['00:00:00:00:00:06' '00:00:00:00:00:00:00:03' '2']
 ['00:00:00:00:00:05' '00:00:00:00:00:00:00:03' '1']
 ['00:00:00:00:00:08' '00:00:00:00:00:00:00:04' '2']
 ['00:00:00:00:00:03' '00:00:00:00:00:00:00:02' '1']]


[['Switch Source mac', '   S_Port', ' Destination mac ', 'D_Port'],
['00:00:00:00:00:00:00:04', 4, '00:00:00:00:00:00:00:0b', 2],
['00:00:00:00:00:00:00:02', 3, '00:00:00:00:00:00:00:15', 2],
['00:00:00:00:00:00:00:04', 3, '00:00:00:00:00:00:00:16', 1],
['00:00:00:00:00:00:00:11', 1, '00:00:00:00:00:00:00:15', 3],
['00:00:00:00:00:00:00:03', 4, '00:00:00:00:00:00:00:16', 2],
['00:00:00:00:00:00:00:01', 4, '00:00:00:00:00:00:00:0a', 1],
['00:00:00:00:00:00:00:01', 3, '00:00:00:00:00:00:00:15', 1],
['00:00:00:00:00:00:00:12', 2, '00:00:00:00:00:00:00:16', 3],
['00:00:00:00:00:00:00:0a', 3, '00:00:00:00:00:00:00:12', 1],
['00:00:00:00:00:00:00:02', 4, '00:00:00:00:00:00:00:0a', 2],
['00:00:00:00:00:00:00:0b', 3, '00:00:00:00:00:00:00:11', 2],
['00:00:00:00:00:00:00:03', 3, '00:00:00:00:00:00:00:0b', 1]]
```

Figure 4.7. *Network Information Database (NID) viewed in Spyder IDE*

- **Network mapping**

After the NID is created, AUTOVNET is ready to receive the vSDN request (SH and DH) from the network administrator. In AUTOVNET, the vSDN request is simplified compared to the one used in traditional OVX hypervisor. Furthermore, the vSDN request in AUTOVNET remains unchanged regardless of the underlying network topology.

The inputs SH and DH, together with the NID are used by AUTOVNET to analyze the various connections in the physical network topology and create a model representing the connections in the underlying network. This model shows the connections between the physical switches in the network.

Given a vSDN request containing the MAC addresses of h1: 00:00:00:00:00:01 as SH and h5: 00:00:00:00:00:05 as DH, the network model generated by AUTOVNET to present the connections between the physical switches in the network is shown in Figure 4.8. In this model, the "green" dots represent the physical switches in the network. Note how Figure 4.8 is very similar to Figure 4.3 (Floodlight's representation of the network topology).

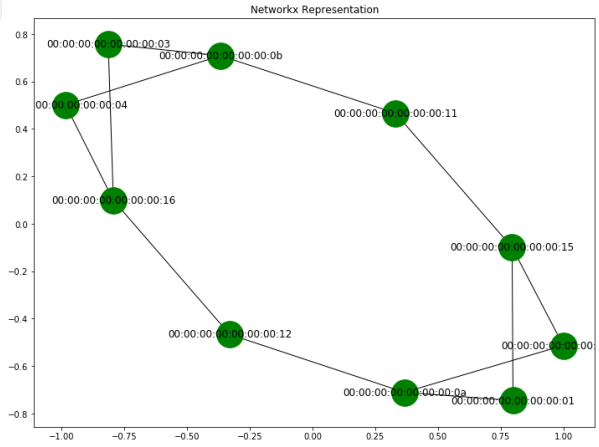


Figure 4.8. *Network representation of the physical network*

Using this model, AUTOVNET computes the shortest routing options between SH and DH. The topology used in this analysis offers two routing options between any given SH and DH.

The MAC addresses of the switches in each route are used to generates the Switch Route Database (SRD). Table 4.2 shows the SRD generated for a vSDN request containing h1 as SH and h5 as DH.

Table 4.2. *Shortest Route Database (SRD) for h1 and h5*

Route 1	Route 2
00:00:00:00:00:00:00:01	00:00:00:00:00:00:00:01
00:00:00:00:00:00:00:0a	00:00:00:00:00:00:00:15
00:00:00:00:00:00:00:12	00:00:00:00:00:00:00:11
00:00:00:00:00:00:00:16	00:00:00:00:00:00:00:0b
00:00:00:00:00:00:00:03	00:00:00:00:00:00:00:03

The SRD in Table 4.2 shows the two routing options (Route 1 and Route 2) between h1 and h5 in this network topology. For a different network topology offering, for instance, five shortest routing options between the source and destination hosts, the corresponding SRD generated by AUTOVNET will contain five routes.

From Table 4.1, the MAC addresses in Route 1 correspond to switches S1, S21, S17, S11, and S3. Similarly, the MAC addresses in Route 2 correspond to switches S1, S10, S18, S22, and S3. Switches S1 and S3 are called edge switches and appear in both routes because they are directly connected to h1 and h5 (see Figure 4.2).

Therefore, the network mapping process uses the MAC addresses of the source and destination hosts, and the Network Information Database (NID) to create Switch Route Database (SRD).

Once the SRD is created, AUTOVNET allows the network administrator to select between two virtualization choices, i.e., either random or smart virtualization.
These two choices are explained in the following subsections.

4.4.1. Random virtualization

Random virtualization triggers a process known as random path selection.

- **Random route selection**

In this process, AUTOVNET randomly selects one routing option from the SRD. The randomly selected route becomes the Best Switch Route (BSR).

If, for example, Route 1 in SRD is selected randomly, then, Route 1 is considered the best routing option (BSR) between h1 and h5. The switches of Route 1 are isolated and dedicated to the virtual network. AUTOVNET automatically uses this BSR to configure OVX to create a vSDN connection between h1 and h5 as in E.4.1.

After the execution of AUTOVNET in Spyder IDE, Figure 4.9 shows the random network virtualization creation steps.

```
********* vSDN Request Phase ***********

Enter MAC address of Source

00:00:00:00:00:01

Enter MAC address of Destination

00:00:00:00:00:05
****************************************

Create Switch Route Database

Route * 1 *

 ['00:00:00:00:00:00:00:01', '00:00:00:00:00:00:00:0a', '00:00:00:00:00:00:00:12', '00:00:00:00:00:00:00:16', '00:00:00:00:00:00:00:03']

Route * 2 *

 ['00:00:00:00:00:00:00:01', '00:00:00:00:00:00:00:15', '00:00:00:00:00:00:00:11', '00:00:00:00:00:00:00:0b', '00:00:00:00:00:00:00:03']

************ DECISION PHASE *************
CHOICE [1] for RANDOM VIRTUALIZATION (Works on any Topology)
CHOICE [2] for SMART VIRTUALIZATION (Specific to Fat-Tree Topology; Wireshark Data Needed)

****************************************
SELECT 1 or 2

1

RANDOM VIRTUALIZATION SELECTED

***********NETWORK VIRTUALIZATION STARTS*************

CONNNECTING TO OPENVIRTEX

Setting up virtual network between 00:00:00:00:00:01 and 00:00:00:00:00:05 .....

Virtual network has been created (network_id {u'mask': 16, u'networkAddress': 167772160, u'controllerUrls': [u'tcp:localhost:10000'],
u'tenantId': 1}).
Network (tenant_id 1) has been booted

***********NETWORK VIRTUALIZATION COMPLETED*************
```

Figure 4.9. *AUTOVNET random virtualization*

In Figure 4.9, the vSDN request contains SH and DH as 00:00:00:00:00:01 and 00:00:00:00:00:05, respectively. Using this information, AUTOVNET generates the SRD containing Route 1 and Route 2. Selecting random virtualization allows AUTOVNET to choose from either one of these routes to create and configure a vSDN connection for SH and DH using OVX.

To test whether a vSDN network is created, one can use the ping command in Mininet. Figure 4.10 shows this step for the case studied; specifically, *"h1 ping -c3 h5"* command is executed in Mininet.

```
Hosts configured with IPs, switches pointing to OpenVirteX at 127.0.0.1:6633
mininet> h1 ping -c3 h5
PING 10.0.0.5 (10.0.0.5) 56(84) bytes of data.
64 bytes from 10.0.0.5: icmp_seq=1 ttl=64 time=26.4 ms
64 bytes from 10.0.0.5: icmp_seq=2 ttl=64 time=0.827 ms
64 bytes from 10.0.0.5: icmp_seq=3 ttl=64 time=0.099 ms

--- 10.0.0.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2012ms
rtt min/avg/max/mdev = 0.099/9.116/26.424/12.242 ms
mininet>
```

Figure 4.10. *Testing AUTOVNET random virtualization in Mininet*

From Figure 4.10, it is seen that AUTOVNET has successfully created a vSDN connection between h1 and h5 since all pings between h1 and h5 are successfully transmitted and received with 0% packet loss.

Random virtualization is very fast and works with any given network topology. On the other hand, random virtualization does not check for faults in the underlying network and simply creates virtual networks between two hosts using the switches from a randomly chosen SRD route.

Therefore, if the chosen route has the broken link or an inactive switch, the virtual network created by AUTOVNET will be faulty by default. To avoid this situation, AUTOVNET offers a second virtualization option called smart virtualization which can detect faulty links and inactive switches in the underlying network and avoid using these resources when creating virtual networks. This virtualization is described as follows.

4.4.2. Smart virtualization

Smart virtualization requires another input from the network administrator which is Wireshark data (WS).

When the network is created in Mininet, executing the “pingall” command allows Wireshark to collect datagram from the network interfaces called WS. WS is analyzed by AUTOVNET to detect faults, congested and slow routes in the network.

Smart virtualization allows one to test AUTOVNET in two different scenarios: (a) a healthy network scenario in which all shortest routing options in the network are operational, in which case AUTOVNET determines and selects the least congested and fastest route as the best shortest route, and (b) a faulty network scenario in which one or more links in the network are broken, in which case AUTOVNET detects the faulty route and uses the other backup route for virtualization.

Using the network topology in Figure 4.2, the Wireshark data collected from a healthy network and a faulty network are plotted in Figure 4.11 and Figure 4.12.

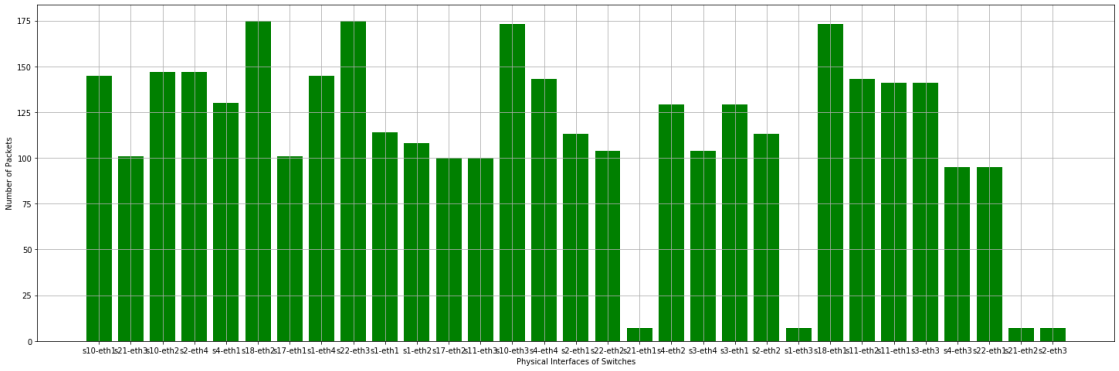


Figure 4.11. *Wireshark data plot from a healthy network*

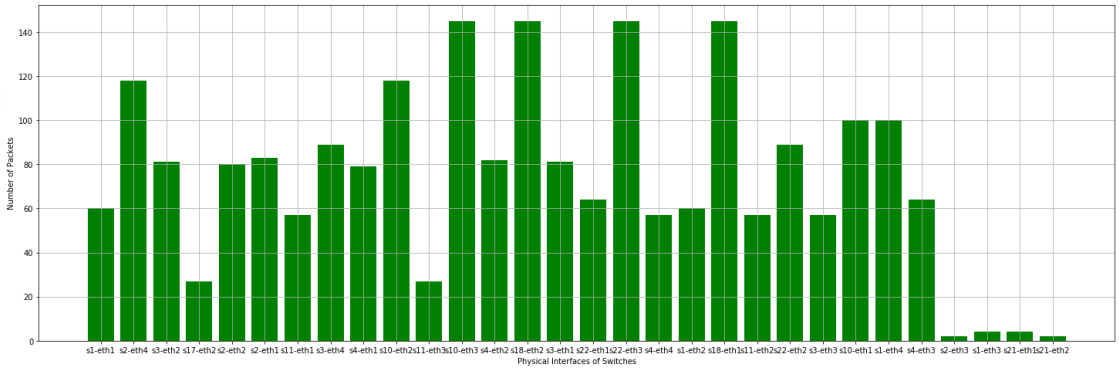


Figure 4.12. *Wireshark data plot from a faulty network*

The leftmost bar in Figure 4.11 shows the activity on interface s10-eth1 which is the number of packets at switch S10 at Port 1. Since every connected port on a switch is powered by an interface, every Switch-to-Switch link has two interfaces (see Figure 4.2). For example, link S21 S17 consists of interfaces s21-eth3 and s17-eth1, where eth3 and eth1 indicate their respective ports.

Notice that Figure 4.12 has fewer bars compared to Figure 4.11. This is because faulty links have inactive interfaces.

Since we use the network topology in Figure 4.2, the NID remains unchanged. A vSDN request containing the MAC addresses of h2: 00:00:00:00:00:02 as SH and h8: 00:00:00:00:00:08 as DH, generates the SRD in Table 4.3 during network mapping.

Table 4.3. *Shortest Route Database (SRD) for h2 and h8*

Route 1	Route 2
00:00:00:00:00:00:00:01	00:00:00:00:00:00:00:01
00:00:00:00:00:00:00:0a	00:00:00:00:00:00:00:15
00:00:00:00:00:00:00:12	00:00:00:00:00:00:00:11
00:00:00:00:00:00:00:16	00:00:00:00:00:00:00:0b
00:00:00:00:00:00:00:04	00:00:00:00:00:00:00:04

Under smart virtualization, after network mapping, AUTOVNET moves to the switch interface mapping stage.

- **Switch interface mapping**

Using only Wireshark data (WS), AUTOVNET cannot determine if a switch is healthy or defective. This is because WS only indicates the name of interfaces (e.g. s1-eth3) without referring to the MAC addresses of the switches. Similarly, NID indicates the MAC addresses of the switches without referring to the interfaces. To resolve this problem, AUTOVNET carries out a switch interface mapping process. During this process, AUTOVNET maps all the interfaces in WS to switches in NID. Consequently, AUTOVNET identifies all the switches with active interfaces and stores the MAC addresses of these switches in a database called the Switch Interface Database (SID).

This mapping allows AUTOVNET to use the NID and WS to determine whether a switch or link is healthy or faulty since a missing interface in WS indicates that the corresponding link or port on the switch is defective.

- **Active route mapping**

At this point, the SRD displays the switches in the routing options between SH and DH, and the SID shows all the switches with healthy interfaces.

To determine whether the switches in SRD are active or healthy, AUTOVNET checks to see if these switches are also present in SID during a process called active route mapping. During this process, routes in SRD containing switches also found in SID, are stored in the Active Route Database (ARD).

The ARD contains routes with both SRD and SID switches. Routes in SRD with non-SID switches are rejected. This is how AUTOVNET detects faulty routes in the underlying network topology. Mathematically, the active route mapping compares SRD and SID which is an intersection operation, that is,

$$ARD = SRD \cap SID \tag{4.2}$$

From E.4.2, we deduce that the size of ARD should be less than or equal to the size of SRD. That is,

$$s(ARD) \leq s(SRD) \tag{4.3}$$

where s(SRD) is the size of SRD. For example, the SRD in Table 4.3 has a size of two.

E.4.3 leads to two possibilities. Either s(ARD) is less than s(SRD), in which case fault is detected and one or more routing options in SRD are eliminated, or s(ARD) is identical to s(SRD), in which case no fault is detected and all the routing options in SRD are healthy.

We describe the behavior of AUTOVNET under these two possibilities in the following subsections.

4.4.2.1. Case 1: faulty network

In a faulty network scenario, s(ARD) is less than s(SRD). That is,

$$s(ARD) < s(SRD) \tag{4.4}$$

E.4.4 indicates that one or more routes in SRD are not in ARD because they contain inactive or faulty switches.

After the execution of AUTOVNET in Spyder IDE, Figure 4.13 shows the smart network virtualization creation steps for a faulty network scenario.

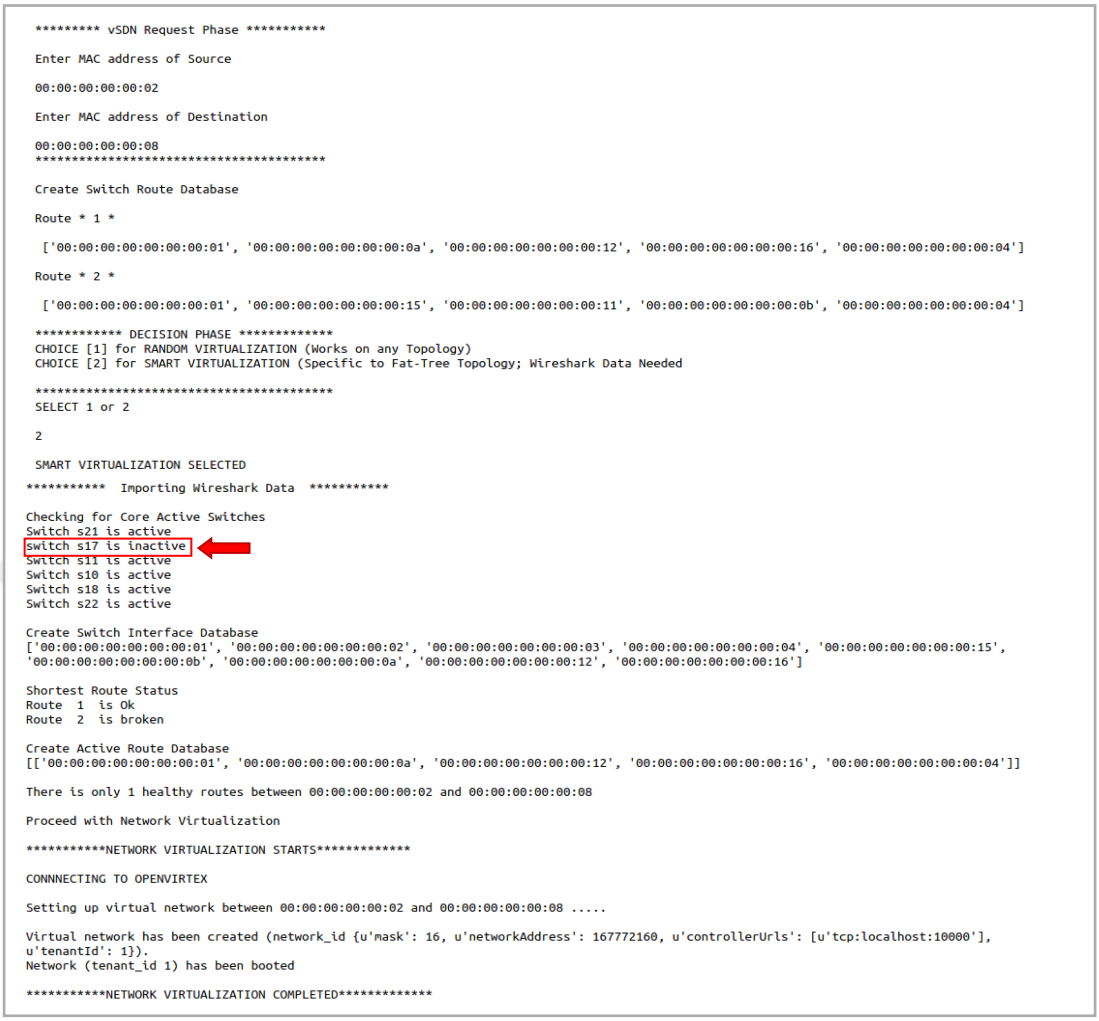


Figure 4.13. *AUTOVNET smart virtualization with a faulty network*

To create a faulty network version of the network topology shown in Figure 4.2, the link between the switches S21 and S17 is disabled using the *"link S21 S17 down"* command in Mininet. Figure 4.12 shows the Wireshark data plot of the faulty network.

Using this data and a vSDN request containing the MAC addresses 00:00:00:00:00:02 as SH and 00:00:00:00:00:08 as DH, which are the MAC addresses of hosts h2 and h8, respectively, AUTOVNET generates the SRD in Table 4.3 which shows the routing options (Route 1 and Route 2). Figure 4.13 shows that AUTOVNET correctly detects network failures by indicating that switch s17 is inactive and Route 2 is broken. The ARD created by AUTOVNET is shown in Table 4.4.

Table 4.4. *Active Route Database (ASD) for h2 and h8 with switch s17 down*

Route 1
00:00:00:00:00:00:00:01
00:00:00:00:00:00:00:0a
00:00:00:00:00:00:00:12
00:00:00:00:00:00:00:16
00:00:00:00:00:00:00:04

As predicted in E.4.4, the ARD in this case contains only Route 1. Route 2 was eliminated since it contained the faulty link and the MAC address of the inactive switch, S17, 00:00:00:00:00:11 (see Table 4.3).

Therefore, Route 1 is considered as the best routing option (BSR) between h2 and h8. The switches of Route 1 are isolated and dedicated to the virtual network and AUTOVNET automatically uses this BSR to configure OVX to create a vSDN connection between h2 and h8 as in E.4.1.

To test whether a vSDN network is created, one can use the *ping* command in Mininet. Figure 4.14 shows this step for the case studied specifically *"h2 ping -c3 h8"* command is executed in Mininet.

```
biyiha@biyihatobie-Pc:~/mininet/mylabo/VN$ sudo python topologyV.py
Hosts configured with IPs, switches pointing to OpenVirteX at 127.0.0.1:6633
mininet> h2 ping -c3 h8
PING 10.0.0.8 (10.0.0.8) 56(84) bytes of data.
64 bytes from 10.0.0.8: icmp_seq=1 ttl=64 time=25.5 ms
64 bytes from 10.0.0.8: icmp_seq=2 ttl=64 time=0.914 ms
64 bytes from 10.0.0.8: icmp_seq=3 ttl=64 time=0.101 ms

--- 10.0.0.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2032ms
rtt min/avg/max/mdev = 0.101/8.844/25.519/11.795 ms
mininet>
```

Figure 4.14. *Testing AUTOVNET smart virtualization with faulty network*

From Figure 4.14, it is seen that AUTOVNET has successfully created a vSDN connection between h2 and h8 since all pings between h2 and h8 are successfully transmitted and received with 0% packet loss.

4.4.2.2. Case 2: healthy network

In a healthy network scenario, s(ARD) is identical to s(SRD). That is,

$$s(ARD) = s(SRD) \tag{4.5}$$

E.4.5 indicates that there are several healthy routes in ARD. Therefore, AUTOVNET performs congestion analysis to determine which route in ARD is the least congested.

- **Congestion analysis**

Using the network topology in Figure 4.2 and a vSDN request containing the MAC addresses of h2: 00:00:00:00:00:02 as SH and h8: 00:00:00:00:00:08 as DH, AUTOVNET generates the SRD in Table 4.3 which shows the two routing options (Route 1 and Route 2). The data plotted in Figure 4.11 is collected from this healthy network by Wireshark. AUTOVNET analyzes these data to determine the fastest and least congested route. Table 4.5 shows the number of packets on the interfaces of SRD routes.

Table 4.5. *Number of packets on interfaces of Route 1 and Route 2*

Interfaces	**I1**	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
Route 1	145	145	175	175	175	175	104	104
Route 2	7	7	101	101	100	100	141	141

Figure 4.15 shows a plot of the total number of packets on the interfaces associated with Route 1 and Route 2.

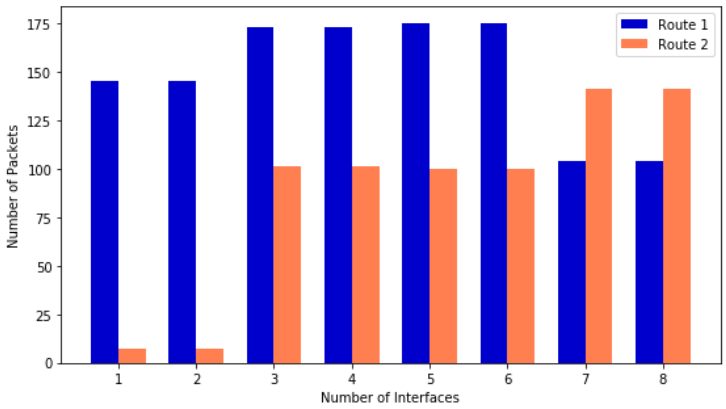


Figure 4.15. *Congestion plots of SRD routes*

There are eight interfaces on both routes because each switch-to-switch connection consists of two interfaces and the two routes in SRD (see Table 4.3) consist of four switch-to-switch connections (five switches connected in series).

Figure 4.15 shows that Route 2 is the least congested routing option because it contains fewer packets than Route 1.

For deep packet inspection analysis, we consider Internet Control Message Protocol (ICMP) packets (echo request) time delays because ICMP packets are only generated by the "*ping"* command. Other protocols' packets (see Figure 4.4) include the Address Resolution Protocol (ARP) and the Link Layer Discovery protocol (LLDP) which are arbitrarily broadcasted across the network. Figure 4.16 shows the distribution of ICMP packet time delay on Route 1 and Route 2.

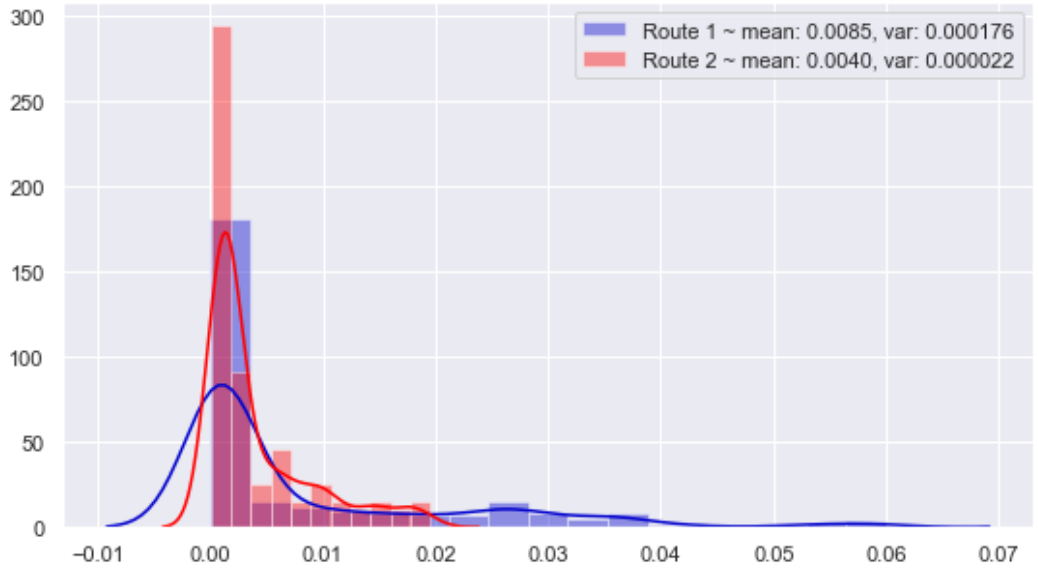


Figure 4.16. *Distribution of ICMP packet time delays on healthy SRD routes*

From Figure 4.16, the ICMP packet time delay distribution has the mean, 0.0040 and variance, 0.000022 on Route 2, and the mean, 0.0085 and variance, 0.000176 on Route 1. These statistics indicate that the packets on Route 2 experience lesser delays than those on Route 1.

Both Congestion and deep packet inspection analyses indicate that Route 2 is the least congested and the fastest routing option compared to Route 1.

After the execution of AUTOVNET in Spyder IDE, Figure 4.17 shows the smart network virtualization creation steps for a healthy network scenario.

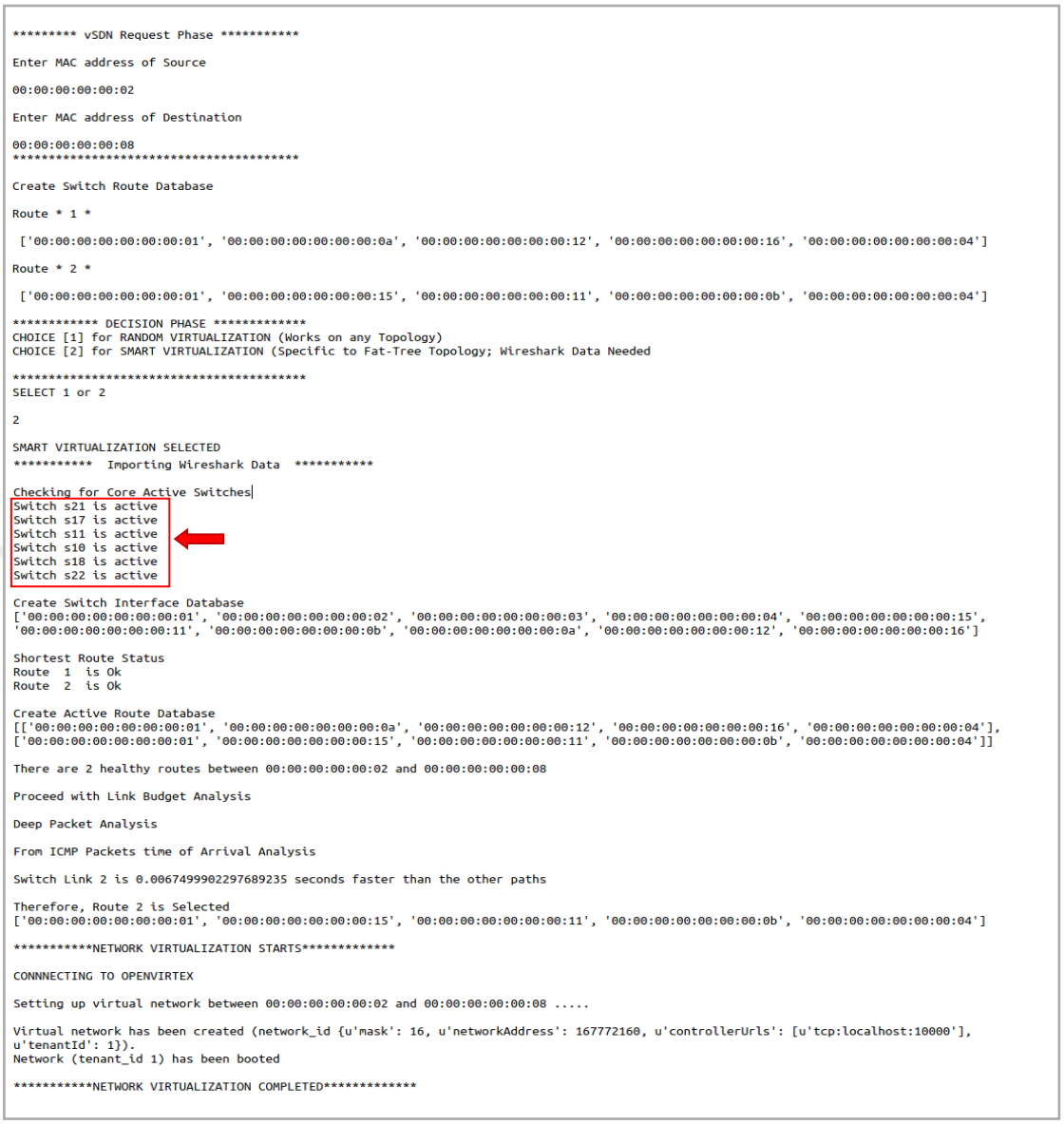


Figure 4.17. *AUTOVNET smart virtualization for a healthy network*

Figure 4.17 shows that in a healthy network scenario, all the switches are active and the SRD generated by AUTOVNET is identical to the ARD. AUTOVNET uses these databases to declare that the physical network is healthy. That is, all the switches are active and the routing options (Route 1 and Route 2) are both operational.

With two active routing options, AUTOVNET must determine which one is the fastest and least congested route. Following the congestion and deep packet inspection analyzes described earlier, AUTOVNET computes that Route 2 is the least congested and the fastest routing option (approximately 4.5ms faster).

As a result, Route 2 is selected as the BSR and its switches are dedicated and used to create and configure the virtual network using OVX.

The "*ping"* command is used in Mininet to test the created vSDN network. Figure 4.18 shows this step for the case studied specifically *"h2 ping -c3 h8"* command is executed in Mininet.

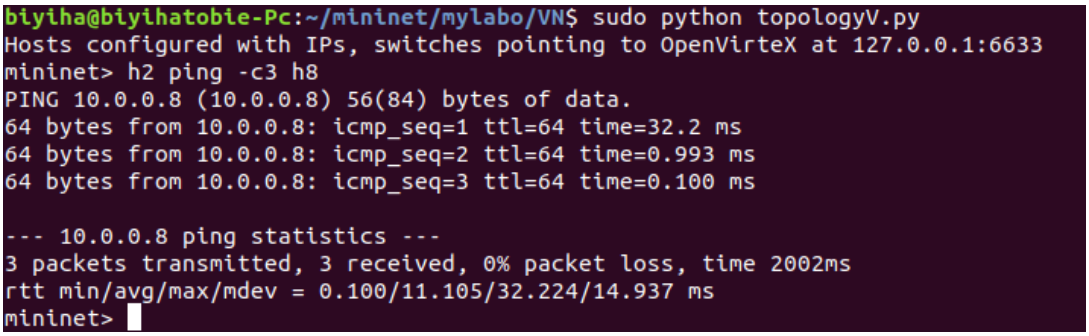


Figure 4.18. *Testing AUTOVNET smart virtualization with a healthy network*

Figure 4.18 shows that AUTOVNET has successfully created a vSDN connection between h2 and h8 since all pings between h2 and h8 are successfully transmitted and received with a 0% packet loss.

Smart virtualization in AUTOVNET offers two key advantages:

(a) Fault detection: In a network with faulty physical resources, smart virtualization guarantees that the virtual networks created are operational. This is because smart virtualization detects and disregards all faulty network resources (switches and links) and uses only the healthy (active) network resources to create the virtual networks.

(b) Load balancing and greater network efficiency: In a network with multiple healthy routing options, smart virtualization provides load balancing by using the least-congested routing option (containing idle and underutilized network resources) to create the virtual network. This method distributes the network load across multiple physical resources and increases the efficiency of the entire network.

5. CONCLUSION AND RECOMMENDATION

Future networks such as Cognitive Networks must be able to sense, monitor and extract valuable data from the physical environment or resources and automatically reconfigure themselves accordingly to meet growing service demands, limited physical resources, and increased connectivity for users. As a result, there is a significant increase in virtualization technologies and optimization research in both the academia and industry to allow various heterogenous complex networks to coexist and share the same physical resources without significant deterioration in network performance. Virtualization concepts like Software-Defined Network (SDN), the abstraction of the data plane from the control plane, and Network Function Virtualization (NFV), the abstraction of network platforms from network services are very promising optimization tools.

In addition, the combination of SDN and NFV to create Virtualized Software-Defined Networking (vSDN) offers the advantages of both paradigms: dynamic resource reservation and flexible virtual network creation through NV and programmability of those resources and easy network management through SDN. However, vSDN solutions increase the complexity of the network by introducing an additional layer called the virtualization layer. This layer typically hosts an SDN hypervisor that handles all virtualization-related tasks.

Virtualized software-defined network is very much at the infancy stage and research is needed to finally declare this technology as the ultimate virtualization solution offering the best value-added performance. This is because the existing shortcomings of virtualized software-defined network architectures are numerous and efficient implementation methods are required in the design and orchestration of physical and virtualized network resources. Furthermore, management issues such as security, configuration, accounting, performance, and fault management become NP-hard for large complex heterogeneous networks. Automated algorithms designed to reduce human inputs and improve network efficiency and responsiveness might open a new chapter to improve the management of these networks.

5.1. Conclusion

In this thesis, a new approach AUTOVNET was designed and implemented to automate and address three important management challenges in virtualized SDN architectures. Network automation in AUTOVNET simplifies manual configurations from the network administrator and increases the flexibility and adaptability of the network. The first management issue addressed by AUTOVNET is the rapid configuration and provisioning of vSDN networks. OpenVirteX was used as the hypervisor in the simulations because it offers network topology and address abstraction. The second management issue addressed was fault detection. AUTOVNET can detect broken links and defective switches in the underlying physical network during the pre-virtualization fault detection analysis. These mechanisms ensure that only healthy network resources are used to create the virtual networks. The third management issue addressed concerns performance. AUTOVNET analyzes and compares routing options to select the best resources and as such increases the efficiency and performance of the entire network by distributing the network load.

AUTOVNET has the following advantages over legacy OVX implementations:

Pure Software Implementation: AUTOVNET is a pure software implemented in Python language. AUTOVNET can run on any python environment like Spyder IDE installed on any computer platform (Linux, Windows, etc.). Because OVX is a Python-based hypervisor, AUTOVNET uses OVX as a Python library to create virtual networks using Mininet SDN emulator and the Floodlight SDN controller. The programmability of AUTOVNET facilitates automation, flexibility, and agility.

Fast deployment of virtual networks: AUTOVNET simplifies the vSDN request for configuring virtual networks with OpenVirteX (OVX). Traditionally, OVX requires a script specifying the MAC addresses and port numbers of network devices to create a simple virtual connection between the hosts in the network. This script is very complex to code and varies for different network topologies. AUTOVNET allows the network administrator to know very little about the underlying physical network topology. The script (vSDN request) required by AUTOVNET always remains the same regardless of the network topology. This is because the MAC addresses of the source and destination hosts indicated by the network administrator are enough for AUTOVNET to map the underlying network topology.

These rapid deployment, provisioning and configuration techniques increase the flexibility, agility, and programmability of OVX and allow network administrators to easily configure virtual networks in larger complex network topologies.

Automated pre-virtualization fault detection: Physical network resources are abstracted by network virtualization technologies to create virtual communications. By default, if the physical resource is defective, the resulting virtual connection will also be faulty. Traditional OVX creates virtual networks using physical resources without checking network faults. AUTOVNET, on the other hand, has a smart virtualization option that automatically checks and detects network faults such as broken links and inactive (defective) switches in the physical network. Ultimately, defective network resources are disregarded, and only healthy resources are shortlisted and used to guarantee the creation of healthy networks. Using the cognitive lifecycle, frequent network sensing and monitoring can be implemented at regular time intervals to further update the network status, produce maintenance reports and keep track of both healthy and defective physical resources.

Optimal resource selection: In large complex networks, selecting the best possible routing option is often challenging. In addition, congestion and over-utilization of physical resources can significantly reduce the performance of the network. AUTOVNET uses network data from Wireshark to carry out the congestion and deep packet inspection analysis in order to identify congested routes and uses underutilized healthy network resources to create virtual networks. Consequently, AUTOVNET distributes the load in the network across multiple network resources and the increases network efficiency and performance.

In conclusion, AUTOVNET is a cognitive network management and orchestration system which monitors and uses network data to automatically create fast, healthy virtual networks in an SDN-supported physical network.

5.2. Future Work

AUTOVNET answers an open question on fast configuration and rapid virtual network provision using OVX. Although several designs and techniques have been implemented over the years to speed up virtual network provisioning in virtualized software-defined network architectures, the algorithm and coding proposed in this thesis

(AUTOVNET) have not been implemented before and the results are extremely promising.

AUTOVNET is very efficient and offers fast virtual network configurations for large, complex network topologies using OVX, but much remains to be done to design a scalable management and orchestration system with complete cognitive capabilities. The followings may be of particular interest for future work.

Greater host capacity: Most real-life data centers and computer communications involve large, complex heterogeneous networks with multiple users and multi-tenants (SDN controllers). So far, AUTOVNET can only support two hosts (a source and a destination) in each virtual network. Therefore, improving the network mapping algorithm of AUTOVNET will improve the host capacity of the proposed system.

Adaptive switch interface mapping: Random virtualization in AUTOVNET can use OVX to quickly create and configure a virtual network with two hosts, regardless of the topology of the network. However, the smart virtualization option only works well for the fat tree network topology. This is because the switch interface mapping algorithm is very limited. Developing an adaptive algorithm to map switch data and interface data will enable the AUTOVNET smart network virtualization option to support larger network topologies.

Security management: AUTOVNET focuses on fault detection, configuration, and performance management. Nevertheless, security management is also a very important aspect of cognitive networks and all autonomous systems must have security mechanisms against network attacks like Distributed Denial of Service (DDoS) attacks.

Integrating machine learning techniques: Future AUTOVNET versions might incorporate machine learning models such as neural networks, K-means and support vector machines to improve the routing option selection mechanisms. These machine learning models could also be leveraged to dynamically map larger network topologies and enable virtual network reconfigurations and virtual network host mobility.

REFERENCES

Ahmed Abdelaziz, Tan Fong Ang, Mehdi Sookhak, Suleman Khan, Athanasios Vasilakos, Chee Sun Liew, & Adnan Akhunzada. (2016). Survey on Network Virtualization Using OpenFlow: Taxonomy, Opportunities, and Open Issues. *KSII Transactions On Internet And Information Systems, 10* (10), 234-245.

Akyildiz, I. F., Lee, W.-Y., Vuran, M. C., & Mohanty, S. (2006). Next generation dynamic spectrum access/cognitive radio wireless networks: A survey. *Computer Networks, 50* (13), 2127-2159. doi:10.1016/j.comnet.2006.05.001.

Akyildiz, I. F., Lee, W. Y., & Mohanty, S. (2008). A survey on spectrum management in cognitive radio networks. *IEEE Communications Magazine, 46* (4), 40-48. doi:Doi 10.1109/Mcom.2008.4481339.

Al-Shabibi, A., De Leenheer, M., Gerola, M., Koshibe, A., Parulkar, G., Salvadori, E., & Snow, B. (2014). *OpenVirteX: Make Your Virtual SDNs programmable*. Paper presented at the Proceedings of the third workshop on Hot topics in software defined networking - HotSDN '14.

Alexander Clemm. (2006). Network Management Fundamentals [Press release].

Ali Al-Shabibi, Marc De Leenheer, Matteo Gerolay, Ayaka Koshibe, William Snow, & Parulkar, G. (2014). OpenVirteX: A Network Hypervisor. *Open Networking Laboratory, Menlo Park, CA 94025, USA*, 1-2.

AuYoung, A., Ma, Y., Banerjee, S., Lee, J., Sharma, P., Turner, Y., Mogul, J. (2014). *Democratic Resolution of Resource Conflicts Between SDN Control Programs*.

Ayoubi, S., Limam, N., Salahuddin, M. A., Shahriar, N., Boutaba, R., Estrada-Solano (2018). Machine Learning for Cognitive Network Management. *IEEE Communications Magazine, 56* (1), 158-165. doi:10.1109/mcom.2018.1700560.

Azodolmolky, S. (2013). *Software defined Network with OpenFlow*: Packt Publishing.

Banerjee, U., Ashutosh, V., & Mukul, S. (2010). *Evaluation of the Capabilities of WireShark as a tool for Intrusion Detection*, 6 (4), 34-43.

Blenk, A., Basta, A., Reisslein, M. (2016). Survey on Network Virtualization Hypervisors for Software Defined Networking. *IEEE Communications Surveys & Tutorials, 18* (1), 655-685. doi:10.1109/comst.2015.2489183.

Blenk, A. A. (2018). *Towards Virtualization of Software-Defined Networks: Analysis, Modeling, and Optimization.*

Braun, W., & Menth, M. (2014). *Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices* 6 (9), 16-23.

Chowdhury, N. M. M. K., & Boutaba, R. (2010). A survey of network virtualization *Computer Networks, 54* (5), 862-876. doi:10.1016/j.comnet.2009.10.017.

Corin, R. D., Gerola, M., Riggio, R., Pellegrini, F. D., & Salvadori, E. (2012). *VeRTIGO: Network Virtualization and Beyond,* European Workshop on Software Defined Networking, 9 (3), 71-84.

Di Benedetto, M.-G., Cattoni, A. F., Fiorina, J., Bader, F., & De Nardis, L. (2015). *Cognitive Radio and Networking for Heterogeneous Wireless Networks*.

Duan, Q., Ansari, N. (2016). Software-defined network virtualization: an architectural framework for integrating SDN and NFV for service provisioning in future networks. *IEEE Network, 30*(5), 10-16. doi:10.1109/MNET.2016.7579021.

Elsen, C. (2013). Physical networks for VMware NSX. *On edge of cloud computing.*

ETSI. (2014). Network Function Virtualization (NFV); Infrastructure; Network domain. *ETSI GS NFV-INF 005, 1.1.1*.

ETSI. (2014). Network Function Virtualization Architectural framework. *ETSI GS NFV 002, 1.2.1*.

ETSI. (2014). Network Function Virtualization Management and Orchestration. *ETSI GS NFV-MAN 001, 1.1.1*.

ETSI. (2017). Network Function Virtulization (NFV); Use Cases. *ETSI GR NFV 001, 1.2.1*.

ETSI. (2018). Network Function Virtualization (NFV); Terminology for Main Concepts in NFV. *ETSI GS NFV 003, 1.3.1*.

FCC. (2012). Location-Based Services - An Overview Of opportunitiea and Other Considerations. *Federal Communications Commission Wireless Telecommunications Bureau*.

Floodlight Controller. (2002). *http://www.projectfloodlight.org/floodlight/*. Retrieved from http://www.projectfloodlight.org/floodlight/. (Access Date: 20/01/2019)

Fortuna, C., & Mohorcic, M. (2009). *Trends in the development of communication networks: Cognitive networks*, 53 (12), 13-17.

Han, B., Gopalakrishnan, V., Ji & Lee, S. (2015). Network Function Virtualization: Challenges and Opportunities for Innovations. *IEEE Communications Magazine, 53*(2), 90-97. doi:Doi 10.1109/Mcom.2015.7045396.

Haykin, S. (2007). Cognitive Radio, Fundamental Issues and Research Challenges. *IEEE, Hamilton Chapter*.

Hoang, D. (2015). Software Defined Networking ? Shaping up for the next disruptive step? *Journal of Telecommunications and the Digital Economy*(AJTDE), 3 (4), 45-55.

Hu, F., Hao, Q., & Bao, K. (2014). A Survey on Software-Defined Network and OpenFlow: From Concept to Implementation. *IEEE Communications Surveys & Tutorials, 16* (4), 2181-2206. doi:10.1109/COMST.2014.2326417.

ITU-T. (2012). *Framework of network virtualization for future networks* (ITU-T Y.3011).

ITU-T. (2014). Framework of software-defined networking. *Series Y: Global Information Infrastructure, Internet Protocol Aspects And Next-Generation Networks Future networks* (Telecommunication Standardization Sector of ITU).

Jain, A., Sadagopan, N. S., Lohani, S. K., & Vutukuru, M. (2016). *A comparison of SDN and NFV for re-designing the LTE Packet Core.* Paper presented at the 2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN).

Jain, R. (2006). Internet 3.0: Ten problems with current Internet architecture and solutions for the next generation. *MILCOM, Military Communications Conference, 1 (7)*, 2309-2317. doi:10.1109/MILCOM.2006.301995.

Jain, R., & Paul, S. (2013). Network Virtualization and Software Defined Networking for Cloud Computing: A Survey. *IEEE Communications Magazine, 51* (11), 24-31. doi:Doi 10.1109/Mcom.2013.6658648.

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, & Alex Buckley. (2013). The Java® Language Specification Java SE 7 Edition. *Oracle and Java Final Release*.

Jeong, W., Yang, G., Kim, S. M., & Yoo, C. (2017). Efficient Big Link Allocation Scheme in Virtualized Software-defined Networking. *2017 13th International Conference on Network and Service Management (Cnsm)*.

Jiménez, Y., Cervelló-Pastor, C., & García, A. J. (2014). *On the controller placement for designing a distributed SDN control layer.* Paper presented at the 2014 IFIP Networking Conference.

Jin, B., Guo, B., Huang, H., Li, S., Shang, Y., & Huang, S. (2017). *An implementation of optical network virtualization based on OpenVirteX*. Paper presented at the 2017 16th International Conference on Optical Communications and Networks (ICOCN). <Go to ISI>://WOS:000425859500191.

Khozeimeh, F., & Haykin, S. (2010). *Self-Organizing Dynamic Spectrum Management for Cognitive Radio Networks*. Paper presented at the 2010 8th Annual Communication Networks and Services Research Conference.

Lebiednik, B., Mangal, A., & Tiwari, N. (2016). *A Survey and Evaluation of Data Center Network Topologies*.

Maleki, A., Hossain, M., Georges, J.-P., Rondeau, E., & Divoux, T. (2017). *An SDN Perspective to Mitigate the Energy Consumption of Core Networks – GEANT2*.

Mijumbi, R., De Turck, F., & Boutaba, R. (2016). Network Function Virtualization: State-of-the-Art and Research Challenges. *IEEE Communications Surveys & Tutorials, 18* (1), 236-262. doi:10.1109/comst.2015.2477041.

Mininet. (2004). Network Emulator. *http://mininet.org/download/*. Retrieved from http://mininet.org/download/ (Access Date: 23/02/2019).

Nazmul Siddique, Syed Faraz Hasan, & Salahuddin Muhammad Salim Zabir. (2017). *Opportunistic Networking Vehicular, D2D and Cognitive Radio Networks* CRC Press Taylor & Francis.

Ndatinya, V., Xiao, Z., Rao Manepalli, V., Meng, K., & Xiao, Y. (2015). *Network forensics analysis using Wireshark* (Vol. 10).

NFV_White_Paper. (2012). Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action. *ETSI*.

ONF. (2013). *OpenFlow Switch Specification*. Retrieved from http://onf/tutorial/.

ONF. (2014). SDN Architecture. *Open Networking Foundation (ONF) TR-502, Issue 1*.

ONF. (2015). Framework for SDN: Scope and Requirements. *Open Networking Foundation (ONF) TR-516, Version 1.0*.

ONF. (2009). OpenFlow Switch Specification. *Open Networking Foundation Version 1.0.0 (Wire Protocol 0x01)* (ONF TS-001).

OpenVirteX. (2010). Tutorial *http://ovx.onlab.us/getting-started/tutorial/*. Retrieved from http://ovx.onlab.us/getting-started/tutorial/ (Access Date: 15/03/2019).

Rao Battula, L. (2014). *Network Security Function Virtualization (NSFV) towards Cloud computing with NFV Over Openflow infrastructure: Challenges and novel approaches*.

REST-API. (2002). Tutorial. *http://www.restapitutorial.com*. Retrieved from http://www.restapitutorial.com (Access Date: 17/02/2019).

Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, & Parulkar, G. (2009). *FlowVisor: A Network Virtualization Layer*.

Salman, O., Elhajj, I. H., Kayssi, A., & Chehab, A. (2016). *SDN Controllers: A Comparative Study*. Paper presented at the Proceedings of the 18th Mediterranean Electrotechnical Conference Melecon 2016. <Go to ISI>://WOS:000390719500129.

Sonam Srivastava, & S.P Singh. (2016). A survey on Virtualization and Hypervisor-based Technology is Cloud Computing Environement. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET), 5*(2).

Stancu, A. L., Halunga, S., Vulpe, A., Suciu, G., Fratu, O., & Popovici, E. C. (2015). *A comparison between several Software Defined Networking controllers.* 12th International Conference on Telecommunication in Modern Satellite, Cable and Broadcasting Services.

Stuckmann, P., & Zimmermann, R. (2009). European Research on Future Internet Design. *IEEE Wireless Communications, 16* (5), 2721-2734, doi:Doi 10.1109/Mwc.2009.5300298.

Thomas, R. W., DaSilva, L. A., & MacKenzie, A. B. (2005). *Cognitive networks.* Paper presented at the First IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005.

Thomas, R. W., Friend, D. H., and Dasilva, L. A. (2006). Cognitive networks: adaptation and learning to achieve end-to-end performance objectives. *IEEE Communications Magazine, 44* (12), 51-57. doi:10.1109/MCOM.2006.273099.

Tourrilhes, J., Sharma, P., Banerjee, S., & Pettit, J. (2014). *SDN and OpenFlow Evolution: A Standards Perspective*, 47 (12), 32-45.

Vishram Mishra , Jimson Mathew, & Chiew-Tong Lau. (2017). *QoS and Energy Management in Cognitive Radio Network*: Springer International Publishing.

Wang, S. (2014). *Comparison of SDN OpenFlow network simulator and emulators: EstiNet vs. Mininet.* Paper presented at the 2014 IEEE Symposium on Computers and Communications (ISCC).

Zhang, Y. (2018). *Network Function Virtualization, Concepts and Applicability in 5G Networks*: John Wiley & Sons, Inc.

Zhang, Z. S., Long, K. P., & Wang, J. P. (2013). Self-Organization Paradigms and Optimization Approaches for Cognitive Radio Technologies: A Survey. *Ieee Wireless Communications, 20* (2), 36-42. doi:Doi 10.1109/Mwc.2013.6507392.

Appendix A

Customized Fat Tree Topology

```
#!/usr/bin/python
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch
from mininet.topo import Topo

class fatTreeTopo(Topo):

   def __init__(self):

      Topo.__init__(self)

      #Add hosts
      h7 = self.addHost('h7', cls=Host, ip='10.0.0.7', defaultRoute=None)
      h8 = self.addHost('h8', cls=Host, ip='10.0.0.8', defaultRoute=None)
      h1 = self.addHost('h1', cls=Host, ip='10.0.0.1', defaultRoute=None)
      h2 = self.addHost('h2', cls=Host, ip='10.0.0.2', defaultRoute=None)
      h4 = self.addHost('h4', cls=Host, ip='10.0.0.4', defaultRoute=None)
      h3 = self.addHost('h3', cls=Host, ip='10.0.0.3', defaultRoute=None)
      h5 = self.addHost('h5', cls=Host, ip='10.0.0.5', defaultRoute=None)
      h6 = self.addHost('h6', cls=Host, ip='10.0.0.6', defaultRoute=None)

      #Add switches
      s10 = self.addSwitch('s10', cls=OVSKernelSwitch)
      s3 = self.addSwitch('s3', cls=OVSKernelSwitch)
      s17 = self.addSwitch('s17', cls=OVSKernelSwitch)
      s4 = self.addSwitch('s4', cls=OVSKernelSwitch)
      s18 = self.addSwitch('s18', cls=OVSKernelSwitch)
      s1 = self.addSwitch('s1', cls=OVSKernelSwitch)
      s11 = self.addSwitch('s11', cls=OVSKernelSwitch)
      s21 = self.addSwitch('s21', cls=OVSKernelSwitch)
      s22 = self.addSwitch('s22', cls=OVSKernelSwitch)
      s2 = self.addSwitch('s2', cls=OVSKernelSwitch)

      #Add links
      self.addLink(h1, s1)
      self.addLink(h2, s1)
      self.addLink(h3, s2)
      self.addLink(h4, s2)
      self.addLink(h5, s3)
      self.addLink(h6, s3)
      self.addLink(h7, s4)
      self.addLink(h8, s4)
      self.addLink(s1, s21)
      self.addLink(s21, s2)
      self.addLink(s1, s10)
      self.addLink(s2, s10)
      self.addLink(s3, s11)
      self.addLink(s4, s22)
      self.addLink(s11, s4)
      self.addLink(s3, s22)
      self.addLink(s21, s17)
      self.addLink(s11, s17)
      self.addLink(s10, s18)
      self.addLink(s22, s18)

topos = { 'mytopo': (lambda: fatTreeTopo() ) }
```

Appendix B

OpenVirteX Script to create a Virtual network between h1, 00:00:00:00:00:01 and h8, 00:00:00:00:00:08

```
# Create the Virtual Network
$ python ovxctl.py -n createNetwork tcp:localhost:10000 10.0.0.0 16
# Create the Virtual Switches
$ python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:00:01
$ python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:00:15
$ python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:00:11
$ python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:00:0b
$ python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:00:04
# Create the Virtual Ports
$ python ovxctl.py -n createPort 1 00:00:00:00:00:00:00:01 1
$ python ovxctl.py -n createPort 1 00:00:00:00:00:00:00:01 3
$ python ovxctl.py -n createPort 1 00:00:00:00:00:00:00:15 1
$ python ovxctl.py -n createPort 1 00:00:00:00:00:00:00:15 3
$ python ovxctl.py -n createPort 1 00:00:00:00:00:00:00:11 1
$ python ovxctl.py -n createPort 1 00:00:00:00:00:00:00:11 2
$ python ovxctl.py -n createPort 1 00:00:00:00:00:00:00:0b 3
$ python ovxctl.py -n createPort 1 00:00:00:00:00:00:00:0b 2
$ python ovxctl.py -n createPort 1 00:00:00:00:00:00:00:04 4
$ python ovxctl.py -n createPort 1 00:00:00:00:00:00:00:04 2
# Create Virtual Links
$ python ovxctl.py -n connectLink 1 00:a4:23:05:00:00:00:01 2 00:a4:23:05:00:00:00:02 1 spf
$ python ovxctl.py -n connectLink 1 00:a4:23:05:00:00:00:02 2 00:a4:23:05:00:00:00:03 1 spf
$ python ovxctl.py -n connectLink 1 00:a4:23:05:00:00:00:03 2 00:a4:23:05:00:00:00:04 1 spf
$ python ovxctl.py -n connectLink 1 00:a4:23:05:00:00:00:04 2 00:a4:23:05:00:00:00:05 1 spf
# Create Virtual Hosts
$ python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:01 1 00:00:00:00:00:01
$ python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:05 2 00:00:00:00:00:08
# Create Start Virtual Network
$ python ovxctl.py -n startNetwork 1
```

Appendix C

OpenVirteX Script to create Big Switch for vSDN connecting h1, 00:00:00:00:00:01 and h8, 00:00:00:00:00:08

```
# Create the Virtual Network
$ python ovxctl.py -n createNetwork tcp:localhost:20000 10.0.0.0 16
$ python ovxctl.py -n createSwitch 2
00:00:00:00:00:00:00:01,00:00:00:00:00:00:00:15,00:00:00:00:00:00:00:11,00:00:00:00:00:00:00:0b,00:00:00:00:00:00:00:04
$ python ovxctl.py -n createPort 2 00:00:00:00:00:00:00:01 1
$ python ovxctl.py -n createPort 2 00:00:00:00:00:00:00:04 2
$ python ovxctl.py -n connectHost 2 00:a4:23:05:00:00:00:01 1 00:00:00:00:00:01
$ python ovxctl.py -n connectHost 2 00:a4:23:05:00:00:00:01 2 00:00:00:00:00:08
# Start the Virtual Network
$ python ovxctl.py -n startNetwork 2
```

Appendix D

AUTOVNET Script to create Big Switch for vSDN connecting h1, 00:00:00:00:00:01 and h8, 00:00:00:00:00:08 using OpenVirteX

```
# Create the virtual SDN network
$ 00:00:00:00:00:01
$ 00:00:00:00:00:08
```

RESUME

Name-Surname	: Tobie Yefferson BIYIHA AFOUNG
Language	: English, French, and Turkish
Birthplace and Year	: Yaounde, Cameroon / 1995
Email	: biyihatobie@hotmail.com

EDUCATION

2017 – 2019 Anadolu University, Graduate School of Science, Electrical and Electronics Engineering, Telecommunications.

2010 – 2014 University of Buea, Faculty of Engineering and Technology Electrical and Electronics Engineering Department, Telecommunications Systems Engineering.

WORK EXPERIENCE

2016 Cameroon Oncology Center, Douala, Cameroon, Computer Applications Engineer.

2013 – 2014 Cameroon Railway Cooperation, Douala, Cameroon, Information Systems Engineer (Internship).

CONFERENCE PAPER

- Nuray AT, Tobie Yefferson BIYIHA AFOUNG (2019), Making Virtual and Augmented Reality Real via Network Virtualization, 1st International Conference on Virtual Reality, Turkey.