Openflow Virtual Networking: A Flow-Based Network Virtualization Architecture

Georgia Kontesidou
Kyriakos Zarifis

Master of Science Thesis
Stockholm, Sweden 2009

TRITA-ICT-EX-2009:205

ROYAL INSTITUTE
OF TECHNOLOGY

Openflow Virtual Networking: A Flow-Based Network Virtualization Architecture

Master Thesis Report

November 2009

Students

Kyriakos Zarifis Georgia Kontesidou

Examiner

Markus Hidell

Supervisor

Peter Sjödin

Telecommunication Systems Laboratory (TSLab)

School of Information and Communication Technology (ICT)

Royal Institute of Technology

Stockholm, Sweden

Abstract

Network virtualization is becoming increasingly significant as other forms of virtualization constantly evolve. The cost of deploying experimental network topologies, the strict enterprise traffic isolation requirements as well as the increasing processing power requirements for virtualized servers make virtualization a key factor in both the research sector as well as the industry, the enterprise network and the datacenter.

The definition of network virtualization as well as its manifestations vary widely and depend on the requirements of the environment in which it is deployed. This works sets the foundation towards a network virtualization framework based on a flow-based controlled network protocol like Openflow.

Abstract

Så småningom, har nätverk virtualization blivit signifikant. Hög kostnaden för att utveckla experimentella nätverk topologier, noggranna kraven för en effektiv trafik isolering samt ökande centralenhets krav för virtuella servrar har gjort nätverk virtualization en viktik faktor i båda forskning och företag.

Definitionen av nätverk virtualization samt dess manifestationer beror på miljön som den utvecklas. Den här arbeten försöker att ställa grundvalarna för ett nätverk virtualization framework baserat på ett flow-baserat protokoll som Openflow. Vi beskriver föreslagen arkitekturen och komponenterna som den består av. Sedan beskriver vår proof-of-concept implementation och presenterar en utvärdering av den.

Acknowledgements

We would like to express our sincere gratitude towards Markus Hidell and Peter Sjödin, for proposing the initial idea of the thesis work and for their profound interest and remarkably constructive guidance throughout the duration of the project.

We would also like to express our appreciation to Voravit Tanyingyong, for his valuable support and feedback, for assisting with our equipment needs, but also for his general interest in our work and in moving forward with it, with which we wish him all the best.

Finally, we would like to thank the NOX and Openflow development communities - a special mention goes to Martin Casado - for being greatly responsive and providing us with updated information and instructions when it was required.

Table of Contents

Section C: Implementation

Section D: Epilogue

Chapter 1

An overview of this thesis

1. Introduction

1.1 Goals of this thesis

This goal of this work is threefold: First, we aim to provide an overview of network virtualization techniques, compare them and point out where each of them fails. Second, we suggest a virtualization architecture that, due to its generic and abstract nature, ameliorates most of current techniques' restrictions. This is a result of looking at virtual networks not as slices of physical network infrastructures, but as subsets of network traffic. This traffic-oriented definition of virtual networks is made possible by the use of Openflow, a flow-based controlled protocol which we also describe. Finally, we give a description of a proof-of-concept implementation of this architecture.

1.2 Contents of this thesis

Following on from this introduction, the thesis is organized in 4 sections:

Section A (Chapters 2-4) provides the necessary background on topics involved in this thesis:

Chapter 2 gives a general description of the term virtualization, and provides definitions and examples of several forms of virtualization in computer science.

Chapter 3 narrows the definition of virtualization describing how it is manifest on network infrastructures. An overview of the most popular network virtualization techniques is provided.

Chapter 4 introduces the Openflow protocol, which is the foundation of our proposed network virtualization architecture.

Section B (Chapters 5-6) introduces our own contribution and the proposed schema towards an Openflow-based network virtualization architecture:

Chapter 5 describes our design steps and challenges, and concludes with a description of the final proposed architecture.

Chapter 6 gives a description of the high level API and protocols that we defined in order to build

the architecture proposed in Chapter 5.

Section C (Chapters 7-8) presents our own implementation, experiments and evaluation of the architecture proposed in Section B:

Chapter 7 delves deeper into implementation details regarding our own approach towards the architecture described in Chapter 5, using the high-level API of Chapter 6.

Chapter 8 describes the test environment that we established in order to evaluate the design and our implementation, as well as some conclusions based on the experiments.

Section D (Chapter 9) concludes with some general discussion and possibilities for future work.

SECTION A

Background

Chapter 2 Virtualization

Chapter 3 Network Virtualization

Chapter 4 Openflow

Chapter 2

Virtualization in Computer Science

2. Virtualization

The term virtualization has been around for many years in computer science. Although it is a very broad term, and can be implemented in various layers of a computer system or network, virtualization always refers to the abstraction between physical resources and their logical representation. This definition of virtualization will become clearer as we go through some of the most basic forms of virtualization and as we examine the need for their development.

2.1 Storage Virtualization

Storage virtualization [2] refers to the separation of physical disk space from the logical assignment of that space. Storage available on multiple physical locations can be combined into a single logical resource. The virtualization program or device is responsible for maintaining a consistent mapping between physical and logical space, called **meta-data**.

The main need for the adoption of storage virtualization was inefficient storage utilization. The initial advantage of aggregating disk space into one logical resource is that there is a single point of management for administrators, leading to more efficient storage utilization. However, it also provides flexibility, as it makes it possible to assign storage where it is needed at any time, and also allows for non-disruptive data migration (i.e. the process of copying the virtual disk to another location is transparent to the host that is using it).

Since users are presented with a logical view of the storage space, I/O operations are refering to this virtual space and therefore need to be translated into I/O operations on the physical disk. The virtualization program can perform this translation by using the mapping information on the meta-data. This process of translating «virtual» I/O requests to real I/O requests on the physical storage space is called **I/O redirection** and is one of the most interesting key concepts of storage virtualization.

There are three main techniques that implement storage virtualization. These can be summarized in the following [53], [54]:

Host-based virtualization, is the simplest and most intuitive way to provide storage virtualization. The existing device drivers are responsible for handling the physical storage space. A virtualization program on top of these drivers is responsible for intercepting I/O requests, and based on the the logical to physical space mapping found on the meta-data, it redirects them accordingly.

Storage device-based virtualization, can be described as virtualization on the hardware level. It leverages the capability of RAID controllers (i.e devices that handle multiple disk drives) to create a logical storage space by using resources of a pool of physical device drivers (pooling) and to handle meta-data. Additionally, advanced RAID controllers allow further storage devices to be attached as well as features such as cloning and remote replication.

Network-based virtualization, operates on a network device, typically a server or a **smart switch**. This device sits between the host and storage providing the virtualization functionality (I/O redirection, virtualizing I/O requests, mapping of physical to logical space). The various storage devices appear as physically attached to the Operating System. This network of hosts, storage and virtualization device is called a **Storage Area Network** (SAN). The latter is currently the most commonly deployed type of storage virtualization.

Analyzing the pros and cons of each category of storage virtualization is beyond the scope of this work. However it is worth mentioning a few challenges and hazards that storage virtualization is tackling with today.

First, all storage virtualization technologies offered today are closed, vendor-specific solutions. Therefore, interoperability between different hosts, storage devices and virtualization software at low overhead and performance cost is a key enabler to storage-virtualization. Performance and scalability issues must also be taken under consideration. The lookup performed to determine the mapping between physical and logical storage space can become a time and resource consuming process even more so as the system scales. Furthemore, keeping a consistent meta-data table can become tricky especially in the case of a dynamically changeable system where physical storage units are frequently attached/detached thus meta-data tables need to be constantly updated. Backing up the meta-data table by creating replicas imposes an additional performance burden since all copies of the meta-data table

should be kept up-to-date. Finally, like any other form of virtualization, storage virtualization introduces an additional level of complexity to the system. This affects the maintainace of the system (i.e, troubleshooting becomes a more complex procedure), as well as the performance of the storage device with regards to the I/O operations.

2.2 Server Virtualization

Server virtualization [3] refers to the partitioning of the resources of a single physical machine into multiple execution environments each of which can host a different server. This type of virtualization which is also known as ***server consolidation***, is a common practice in enterprise datacenters in an effort to improve resource utilization and centralize maintainance. Server virtualization decouples the software running on a server from the physical server hardware. A single host is logically divided into a number of Virtual Machines (VMs) or Virtual Environments (VEs) each of which can run its own operating system and its own server applications, making use of an allocated portion of the physical host's resources.

Server virtualization is recommended for small or medium-scale usage applications. The advantages here also include flexibility in terms of non-disruptive migration of the virtual server, ease of management by a single administrative console, and efficiency, as several low-utilized systems are replaced by a single physical entity.

There are three main server virtualization techniques that are deployed today in enterprise datacenters--Virtual Machines, paravirtualization and OS-level virtualization. We will provide a brief overview of these techniques and mention some of the benefits and drawbacks of using each technique.

Virtual Machines (VMs), are software implementations of real or fictional hardware that run on the same physical host. Each VM runs an Operating System which is known as ***guest OS*** requiring physical resources from the host. VMs run on user-space thus on top of the hosting operating system without however being aware that they are not running on real hardware. The **Virtual Machine Monitor VMM** is an intermediate software layer between the OS and the VMs which provides virtualization . The VMM presents each VM with a virtualized view of the real hardware. It is therefore responsible for managing VMs' I/O access requests and passing them to the host OS in order to be executed. This operation introduces overhead which has a noteworthy impact on performance.

However this virtualization technique which is commonly referred to as **full virtualization** offers the best isolation and security for virtual machines and can support any number of different OS versions and distributions. Popular solutions that follow this method of virtualization include Vmware [4], QEMU [5], and Microsoft Virtual Server [6].

Paravirtualization is a technique where each VM is provided with an API to the normal system calls of the underlying hardware. By using this technique operations that require additional privileges and cannot run in user-space are relocated to the physical domain instead of the virtual domain. This way the VMM is freed from performing the complicated and time-consuming task of managing I/O access requests and passing them to the host OS. This technique is also known as ***transparent virtualization*** since it requires of the VMs to be aware of the fact that they are running on a virtualized environment and to communicate with the **VMM**. Paravirtualization introduces less overhead compared to full virtualization but can suppport fewer OS versions since the VMs need to be tailored according to the VMM they are running on top of. Products such as Xen [7] and UML [8] use this technique.

OS-level virtualization, is based on the concept of **containers.** The containers are multiple isolated instances of the same OS running in user-space. The kernel is responsible for creating containers and for providing resource management mechanisms to enable seamless, concurrent operation of multiple containers. OS-level virtualization does not provide the same degree of flexibility as other virtualization methods in terms of supporting different OS versions and distributions. Nevertheless, it introduces much less overhead since containers can use the normal system call interface instead of a virtualized version of it. OpenVZ [9] and Linux-Vservers [10] are examples of OS-level virtualization.

2.3 Application Virtualization

Application virtualization [39] refers to the process of isolating a certain application from the underlying Operating System it runs on. An application has certain dependencies on the OS it is installed over. Except for OS services such as memory allocation and device drivers an application has dependencies such as registry entries, use of environmental variables, access to the database servers, modification of existing files and much more. Through use of virtualization, an application is under the illusion of interfacing directly with the operating system in order to request these services.

In order to virtualize an application an intermediate virtualization layer must be inserted between the underlying OS and the application. This layer is responsible for intercepting the application's requests for resources, logging them separately in a file without commiting them to the OS. In this way, an application is no longer dependent either on the system or on the OS it runs on. This kind of virtualization is very similar to the concept of OS virtualization, except for the fact that in this case it is not the entire OS but rather an application that is being encapsulated from the system.

Application virtualization has many benefits. Sandboxing applications allows them to run independently on different platforms. This independence of the application from the OS shields the OS and provides improved system security since buggy or malicious applications do no longer interfere with the system. Furthermore, since every application runs in its own virtualized space, incompatibilities between different applications can be eliminated through the use of virtualization. Microsoft’s SoftGrid [51] is an example of application virtualization.

This chapter aimed at providing a brief overview of how virtualization of resources (such as storage, servers or applications) can be achieved today. Some of the approaches for network virtualization that will be described in the next chapter leverage resource virtualization techniques extensively. In fact, their approach of network virtualization relies on virtualizing physical devices along the network. The following chapter will present a selection of network virtualization techniques and projects that exist today, attempting to contribute towards resolving the confusion that is often caused by the wide and sometimes abstract use of the term network virtualization.

Chapter 3

Network Virtualization

3. Network Virtualization

Although resource virtualization can be instantiated in many more layers, describing all forms of it is outside the scope of this document. However, it has probably been made clear by now that network virtualization is much more efficient when it is backed by other forms of virtualization. Just like other forms of virtualization, network virtualization refers to the decoupling of the hardware that form a physical network, from the logical networks operating over it.

Network virtualization allows for different groups of users to access virtual resources on remote systems over a virtual private network, a logical network that seems to have been cut out for the specific needs of these users. The combination of network virtualization with remote virtual resources provides researchers with very powerful and highly customizable experimental environments. It makes sense to adopt this technology when there are available virtual resources spread out over a physical network.

Of course the use of network virtualization, useful as it might be in the research field, is not limited to this scope. ISPs, companies or even households often adopt network virtualization techniques to achieve network isolation, security and traffic anonymity. Many technologies have been developed during the few last decades for that purpose. In the following paragraph we will briefly go through the development of network virtualization, noting some of its most important implementations.

3.1 VLANs

VLAN (Virtual Local Area Network) [46] is a widely used technology that enables the existance of a virtual network abstraction on top of a physical packet-switched network. A VLAN is essentially a broadcast domain for a specified set of switches. These switches are required to be aware of the existance of VLANs and configured accordingly, in order to perform switching of packets between devices belonging to the same VLAN. VLAN membership can be defined by roughly three ways.

- **Port membership: :** A VLAN can be defined as a set of ports in one or more switches. For

example VLAN 1 can be described as port #1 of switch #1 and port #2 of switch #2 and so on. This type of VLAN is commonly refered to as a static VLAN because the assignment of ports in various switches to VLANs is done manually by the network administrator.

- **MAC address membership:** A VLAN can be defined as a set of specific MAC addresses. This type of VLAN is often refered to as a **dynamic VLAN**, since the assignment between ports and VLANs is done automatically based on the source MAC address of the device connected to a port. The VLAN membership is determined by a querey to a database containing the information on the mapping between MAC addresses and VLAN memberhips.

- **Layer 3 membership:** A VLAN can be defined as a set of IP addresses or an IP subnet or a set of protocols/services. In this case, the assignment between ports and VLANs is done automatically based on the source IP address, IP subnet or the services/protocols running at the device connected to the port. This is also a case of a dynamic VLAN and like the MAC address membership method, a database is queried as a device enters the network, to determine its VLAN membership.

VLANs perform LAN segmentation, by limiting broadcast domains within the borders of a specific VLAN, hence improving bandwidth usage across the network. Implementation of VLANs also provides isolation, higher security and more efficient network management. VLANs require from network elements to have a complete knowledge of the VLAN mapping, regardless of which of the three ways mentioned above this is performed and regardless of whether this knowledge is injected statically or acquired automatically. This very fact impairs network performance by creating additional workload for all network elements. Another drawback is that this solution does not allow generality in the definition of a virtual network. There are three ways to define VLAN membership, and the administrators or network designers are expected to evaluate the tradeoffs according to the type of network they wish to deploy and chose one of the possible approaches. If one wishes to have a more flexible definition of a VLAN or even a custom definition (for example use a combination of IP addresses and ports), this is not possible, especially considering the fact that VLAN implementations on network devices are closed and proprietary.

3.2 VPNs

Virtual Private Networks is another network virtualization concept. A VPN [33] is a private data network that uses a public physical network infrastructure, while maintaining privacy for its users. This privacy is achieved through the use of tunneling protocols and security functions. Physically remote sites that are connected to the same backbone network, can have IP connectivity over this common backbone (that could be the public Internet) by being associated to a common VPN.

A VPN consists of the following components (fig. 1):

- **CE (customer edge) routers:** They reside in the customer side and can be managed and configured either by the customer or the provider.

- **PE (provider edge) routers:** Provide entry and exit points for the VPN to the customer by peering with CE routers and are managed and configured by the provider. They are responsible for most of the VPN/MPLS functionality in the network (paragraph 3.3).

- **P (provider) routers:** All routers that form the provider's core network and are not attached to any CE routers. They are responsible for forwarding VPN traffic along the provider core and between PE routers.

There are various different ways to implement VPNs, which have different requirements in network equipment and configuration. In the following sections, some of the most characteristic VPN implementations will be briefly presented.

3.3 VPN over MPLS

MPLS (Multi-Protocol Label Switching) [53] is a method of forwarding packets in a fast, protocol-agnostic manner. In MPLS, packets are encapsulated by inserting one or more MPLS headers between a packet's layer-2 and layer-3 headers. MPLS is thus referred to as a layer 2,5 protocol. MPLS headers have a simple format which includes a 20-bit *Label* field, a 3-bit *Type of Class* field, a 1-bit *Bottom of Stack* field and an 8-bit *Time-to-Live* field. Forwarding decisions in MPLS are simply made based on the *Label* field of the layer-2,5 header. This simple functionality (along with many extensions that are out of our scope, but an interested reader can refer to [52]) is what makes MPLS fast, reliable and the most popular choice for Service Provider networks.

Network virtualization over MPLS is a popular technique [54]. In VPN over MPLS the concept of sites belonging to VPNs is maintained. Different sites can be connected to one another over the common backbone if they belong to the same VPN. MPLS tunnels are used to forward packets along the backbone network. The main course of events for a data packet arriving at a CE is the following:

- Data arrives from CE (Customer Edge) via access network
- Data is encapsulated by PE (Provider Edge) and sent over tunnel
- Data is decapsulated by receiving PE and sent over access network to CE

Each PE router needs to maintain a number of separate forwarding tables, one for each site that the PE is connected to. When a packet is received from a particular site, the forwarding table associated with that site only is being consulted in order to decide how the packet will be routed. A particular site's forwarding table contains only the routes to other sites that have at least one VPN in common with this particular site. This ensures VPN isolation and allows for different VPNs to use the same or overlapping address space. PE routers use BGP to distribute VPN routes to each other. P routers do not contain any VPN routing information but they simply forward packets according to their MPLS label through the core network and to the appropriate PE router. A basic setup of such a topology can be seen in figure 1.

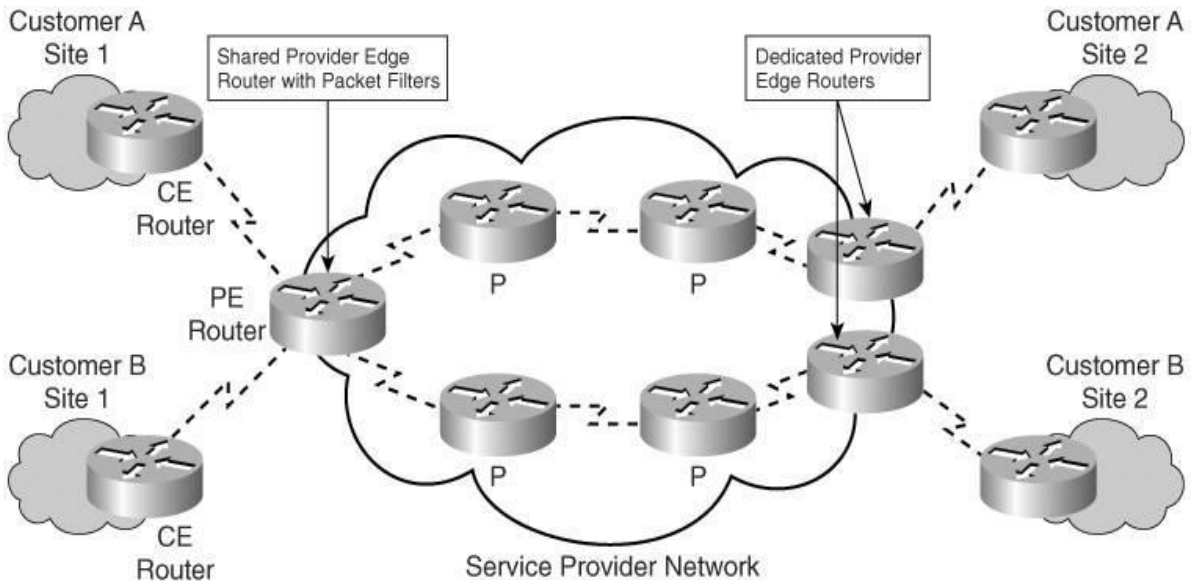


Figure 1. A WAN VPN topology

3.4 Layer 3 VPN

In this type of VPN [54], providers use the IP backbone to provide VPN services to their clients. Thus the set of sites that compose the Layer 3 VPN are interconnected through the public, shared Internet infrastructure. These sites share common routing information. Layer 3 VPNs make use of the BGP protocol to distribute VPN routing information along the provider's network. Additionaly, MPLS is used to forward VPN traffic from and to the remote sites through the provider's backbone.

In order to isolate VPN traffic from the production traffic flowing through the provider's backbone, a ***VPN identifier prefix*** is added to a VPN site address. In addition, each VPN has its own VPN-specific routing table that contains the routing information for that VPN only. There are two approaches for implementing a Layer 3 VPN:

- **CE-based,** where the provider network is not aware of the existence of the VPNs. Therefore the creation and management of tunnels is left to the CE devices.

- **PE-based,** where the provider network is responsible for VPN configuration and management. A connected CE device may behave as if it were connected to a private network.

3.5 Layer 1 VPN

A Layer 1 VPN (L1VPN) [36] is a service offered by a core layer 1 network to provide layer 1 connectivity between two or more customer sites and where the customer has some control over the establishment and type of connectivity. An alternative definition is simply to say that an L1VPN is a VPN whose data plane operates at layer 1. Layer 1 VPNs are an extention to L2/L3 VPNs to provide virtually dedicated links between two or more customer sites. The routing between these two sites however relies entirely on the customer, therefore the data plane does not guarantee control plane connectivity.

3.6 Recent Network Virtualization Frameworks

While the technologies described above are designed to create logical partitions of the same physical substrate they do not aim in providing a complete framework for performing network virtualization. They are commercial, hands-on solutions for logically partitioning the physical network.

Several initiatives to address this objective have spawned as a result of the need of the research community for realistic, programmable and controlled environments on which they can deploy and test novel services, protocols and architectures. Some are more reserved and others more radical, trying to follow a "clean-slate" approach, addressing the Internet community's reluctance to adopt new, innovative ideas and technologies.

3.6.1 PlanetLab

PlanetLab [20] is a network of nodes owned by different research organizations in various locations around the world, interconnected through the Internet creating a testbed for researchers to deploy and test experimental services at real-scale. PlanetLab currently consists of more than 800 nodes in 400 different parts of the world, belonging to the Universities, organizations and companies that participate in the project. The objective of PlanetLab is to provide an overlay network on which researchers can deploy and test their services at real-scale in a way that new innovative architectures, protocols and services can emerge and dominate the underlying technologies. Basically, PlanetLab addresses the problem of experimentation of novel services over the real Internet, at large-scale.

By virtualizing its nodes PlanetLab allows multiple experimental services to run simultaneously, using different topologies over the same physical network, being however isolated from one another.

In the context of PlanetLab a node is a machine that is able to host one or more virtual machines [5] while a collection of Virtual Machines on a set of nodes is referred to as a ***slice***. Each different service should acquire and run in a slice of the overlay. Therefore in PlanetLab, virtualization is a synonym for "slice-ability", referring to the ability to slice up the node's processor, link and state resources.

The main *components* of the PlanetLab architecture can be summarized in the following:

- **Node:** A machine that is able to host one or more virtual machines.
- **Slice:** A collection of Virtual Machines on a set of nodes.
- **Virtual Machine (VM):** A slice of the node seen as an independent machine with its own kernel and resources.
- **Node Manager (NM):** A process that runs on each node, establishing and controlling the virtual machines on the node.
- **PlanetLab Central (PLC):** A centralized front-end that controls and manages a set of nodes on

behalf of their respective owners. Additionally, the PLC creates slices on behalf of the user (namely the researchers). The PLC acts as:

  - **A Slice Authority,** which maintains state of all the slices currently active on the PlanetLab. This includes information on the users that have access to the slice as well as the state of each registered slice.
  - **Management Authority,** the PLC maintains a server responsible for the installation and update of software running on the nodes it manages. The management authority is also responsible for monitoring the nodes' behavior, detecting and correcting possible problems and failures.

- **Virtual Machine Manager (VMM**): A process that runs on each node ensuring allocation and scheduling of the node's resources.

In addition to these components, the architecture includes two *infrastructure services* that, like the experimental services, run on a separate slice on the overlay network, but are associated with core administrative functions of the PlanetLab:

- **Slice Creation Service:** A service running on each node on behalf of the PLC by contacting the Node Manager on each node which in its turn creates a local VM and gives access to the users associated with this particular VM.

- **Auditing Service:** Runs on each node. Logs traffic coming from the node and is responsible for associating network activity to the slice that is generating it.

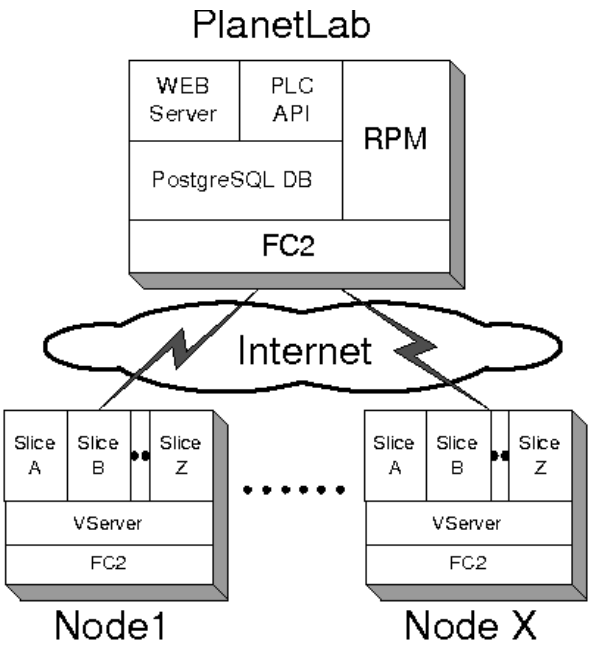


Figure 2. PlanetLab Architecture

source: http://www.usenix.org/event/lisa07/tech/full_papers/jaffe/jaffe_html/index.html

3.6.2 VINI

VINI [21] is a virtual network infrastructure on top of the PlanetLab shared infrastructure. VINI allows network researchers to evaluate their protocols and services in the wide area in a controlled and realistic manner. VINI provides researchers with **control** thus enabling them to induce external events (e.g. link failures, traffic congestion) to their experiments. Additionally it provides a realistic platform for deploying and testing new services or protocols, since its architecture, topology and functionality resembles real networks. To provide researchers flexibility in designing their experiments, VINI supports simultaneous experiments with arbitrary network topologies on a shared physical infrastructure.

VINI currently consists of nodes at sites connected to routers in the National Lambda Rail, Internet2, and CESNET (Czech Republic). The nodes have their own global IP address blocks and participate in BGP peering with neighboring domains.

A prototype implementation of VINI on the PlanetLab nodes resulted in **PL-VINI**. The choice of using PlanetLab as the physical infrastructure on which PL-VINI is deployed seems natural. PlanetLab already provides node virtualization with each Virtual Machine running a different service as described

above. Additionally it provides good control over resources and network isolation while giving each slice the impression of root-level access to a network device. PL-VINI extends the capabilities of PlanetLab by providing researchers with better level of configuration of routing on their slices.

PL-VINI enables arbitrary virtual networks, consisting of software routers connected by tunnels, to be configured within a PlanetLab slice. A PL-VINI virtual network can carry traffic on behalf of real clients and can exchange traffic with servers on the real Internet. Nearly all aspects of the virtual network are under the control of the experimenter, including topology, routing protocols, network events, and ultimately even the network architecture and protocol stacks.

PL-VINI uses a set of widely known Open Source networking tools in order to provide on-demand overlay topologies on top of the PlanetLab nodes. More specifically PL-VINI uses XORP [27] for routing, Click [28] for packet forwarding, OpenVPN [29] for creating server to end-user connections and for interconnection, and rcc **[**30] for parsing router configuration data from operational networks to use in the experiments.

In order to efficiently deploy and run networking experiments, the virtual nodes must be under the impression of having access to real network devices. This can be achieved by running software which simulates the functionality of real network devices (such as XORP in the case of routing) in user-space mode, using UML (User-Mode Linux), a full-featured kernel running on user-space on the same nodes. Additionally in order for network services to be able to send and receive packets on the overlay, a modified version of the Linux TUN/TAP driver is used. Applications running in user-space can send and receive packets by writing/reading the TUN interface while being able to only see packets sent from the slice they belong.

3.6.3 FEDERICA

FEDERICA [22][23] is a European project of the 7th Framework, aiming to provide a Europe-wide technology-agnostic infrastructure destined for research experimentation. The main concept behind FEDERICA is no different than that of PlanetLab and PL-VINI. FEDERICA uses the resources provided by National Research and Education Networks (NRENs), namely links, virtualization-capable nodes and network equipment in order to provide an open platform for the deployment and trialling of new Internet architectures and protocols.

The FEDERICA physical network consists of 13 Points of Presence (PoPs) hosted by different NRENs. These PoPs host nodes (known as Physical Nodes in the context of FEDERICA) which are connected by dedicated links and which are able to host multiple **Virtual Nodes** (VNs). Virtual nodes can either be virtual machines based on various operating systems or emulated L2/L3 network devices. VNs are connected through **Virtual Paths** (VPs) created on top of the physical topology using technologies such as tunneling.

Following the same concept as PlanetLab and PL-VINI, FEDERICA allocates slices to researchers who request them for their experiments and who view slices as physical infrastructures. What is new and interesting about the FEDERICA project is that it enables researchers to access lower network layers, something that it is not feasible in the PL-VINI and PlanetLab implementations. Moreover, compared to PL-VINI, FEDERICA focuses more into creating **virtual infrastructures** as opposed to **virtual network topologies** on top of a common shared infrastructure. This difference is crucial since it implies that FEDERICA allows experimentation on the L2 layer while PL-VINI is limited to experimentation on the L3 layer.

3.6.4 GENI (Global Environment for Network Innovations)

GENI [47] takes PlanetLab and PL-VINI one significant step forward. While PlanetLab and PL-VINI are trying to propose a framework for creating slices over the traditional Internet (basically assuming IP connectivity), GENI follows a "clean-slate" approach, by incorporating different types of networks such as sensor, wireless and mobile networks. A virtual network as envinsioned by GENI can spread over a collection of heterogeneous underlying network technologies. In this sense, PlanetLab and PL-VINI can be seen as small-scale prototypes of GENI.

3.7 A summary of network virtualization techniques and concepts

The plethora of different technologies and research projects in the field of network virtualization might create confusion as to how all these different concepts differ both in their definition of network virtualization as well as in their approach towards implementing it.

Chowdhury et al [25] have identified four main characteristics for categorizing network

virtualization projects:

- **Layer of virtualization:** The layer of the network stack (physical to application) that the virtualization is performed.
- **Networking technology:** The underlying network technology on top of which virtualization is implemented.
- **Level of virtualization:** The component of the network infrastructure on which virtualization is attempted.
- **Architectural Domain:** Particular architectural concepts that virtualization techniques have focused on.

Examining the architectural domain focus of each project is out of the scope of this paper, however the three remaining characteristics will be used hereafter in order to evaluate and categorize the various network virtualization techniques and projects.

VLANs: VLANs operate on level 2 (link-layer) and layer 3 in the case of IP address mapping. VLANs create the illusion that two (or more) machines belong to the same broadcasting domain. Hence they provide virtualization on the level of links. Layer 2 VLANS are technology agnostic. Layer 3 VLANS assume IP connectivity since they use IP addressing to map machines to VLANS.

VPNs: As analyzed above VPNs are capable of providing virtualization on Layer 1 (physical), Layer 2 and Layer 3 (network layer). Hence they provide virtualization on the level of links, switches and routers.

PlanetLab: PlanetLab creates virtual networks or slices, by virtualizing nodes by the use of several specialized programs running on these nodes. It therefore provides application-layer virtualization PlanetLab is implemented on top of traditional Internet and implies IP connectivity.

VINI: Implements virtualization on Layer 1. It virtualizes routers and nodes. It is implemented on PlanetLab assuming IP connectivity.

FEDERICA: Performs virtualization on the link, network and application layer. It virtualizes all network elements- nodes, links and network devices.

GENI: Performs virtualization on the link, network and application layer. It virtualizes all network elements, nodes, links, network devices. GENI is technology agnostic, therefore it can provide virtualization across the span of a collection of heterogeneous networks.

Chapter 4

The Openflow Standard

4. Openflow

Looking back at how networks were implemented 20 years ago and looking at the image of networking today it is obvious that networks have come a long way and evolved tremendously in terms of speed, reliability, security and ubiquity. And while physical layer technologies have evolved providing high-capacity links, network devices have improved in computational power and a vast amount of exciting network applications has emerged, the network in its structure has not seen much change from its early days. In the existing infrastructure, complex tasks that make up the overall functionality of the network such as routing or network access decisions are delegated to network devices from various different vendors all running different firmware. This well-established proprietary base of equipment making up the entire network infrastructure does not give much space for novel research ideas such as new routing protocols to be tested in wide-scale, real networks.

Moreover the penetration of networking in various crucial sectors of our every-day lives discourages any attempt of experimentation with critical-importance production traffic. This is essentially one of the main reasons why network infrastructure has remained static and inflexible and no major breakthroughs have been achieved towards this direction.

Openflow [13] has emerged from the need to address these critical deficiencies in networking today in order to help research bloom. Openflow exploits the existence of lookup tables in modern Ethernet switches and routers. These flow-tables run at line-rate to implement firewalls, NAT, QoS or to collect statistics, and vary between different vendors. However the Openflow team has identified a common set of functions that are supported by most switches and routers. By identifying this common set of functions, a standard way of manipulating flow-tables can be deployed for all network devices regardless of their vendor-specific implementation. Openflow provides this standard way of manipulating flow-tables, allowing a flow-based network traffic partition. This way, network traffic can be organized into various different flows which can be grouped and isolated in order to be routed, processed or controlled in any way desired.

Openflow can find great use in campus networks where isolating research and production traffic is a crucial operation. Flows can be created and maintained by a centralized entity called the **controller.** The controller can be extended in order to perform additional tasks such as routing and network access decisions. By removing network functionality from devices scattered along the network and centralizing it completely or locally, one can more easily control and therefore change it. The only requirements in order to implement this modification are switches that can support Openflow and a centralized controller process which contains the network logic. This way, the control and data plane are no longer colocated in one single network device, but separated and dynamically linked to one another. The separation of control and data plane functions and the adoption of a centrally controlled network model are concepts that have been discussed and approached by researchers before. Efforts like ForCES [43] and SoftRouter [44] have proposed architectures for enabling the decoupling of the control and data plane functionality of network devices, aiming in providing more efficient packet forwarding and greater flexibility in control functions. Openflow shares much common ground with these architectures, however inserting the concept of flows and leveraging the existence of flow tables in commercial switches today. In the following sections the components of an Openflow-based network will be briefly presented.

4.1 The Openflow network

The main components of a controller-based Openflow network are:

- Openflow enabled switches
- Server(s) running the controller process
- Database containing the network view, a «map» of the entire topology of the network.

The Openflow switch, consists of a **flow table** containing flow entries, used to perform packet lookup and forwarding and a **secure channel** to the controller, through which Openflow messages are exchanged between the switch and the controller.

By maintaining a flow table the switch is able to make forwarding decisions for incoming packets by a simple lookup on its flow-table entries. Openflow switches perform an exact match check on specific fields of the incoming packets. For every incoming packet, the switch goes through its flow-table to find a matching entry. If such entry exists, the switch then forwards the packet based on the action

associated with this particular flow entry.

Every flow entry in the flow-table contains [13]:

1. **header fields** to match against packets : These fields are a ten-tuple that identifies the flow.

Ingress Port	Ether Source	Ether Dst	Ether Type	VLAN Id	IP Src	IP Dst	IP Proto	Src Port	Dst Port

Figure 3: Fields used by OpenFlow to match packets against flow entries

2. **counters** to update for matching packet : These counters are used for statistics purposes, in order to keep track of the number of packets and bytes for each flow and the time that has elapsed since the flow initiation.

3. **actions** to apply to matching packets : The action specifies the way in which the packets of a flow will be processed. An action can be one of the following: 1) forward the packet to a given port or ports, after optionally rewriting some header fields, 2) drop the packet 3) forward the packet to the controller.

The **controller** is a core, central entity, gathering the control plane functionality of the Openflow network. Currently there are several controller implementations available. However the most widely used and deployed is the NOX controller [14] an open-source Openflow controller, therefore this report will be mostly focused around its implementation. The controller provides an interface for creating, modifying and controlling the switche's flow-tables. It runs typically on a network-attached server and could either be one for the entire set of Openflow switches on the network, one for each switch or one for each set of switches. Therefore the control functionality of the network can be completely or locally centralized according to how the delegation of switch management to controllers is performed. The requirement, however, is that if there are more than one controller processes, they should have the same view of the network topology at any given time. The network view includes the switch-level topology; the locations of users, hosts, middleboxes, and other network elements and services. Moreover it includes all bindings between names and addresses. In NOX the controller creates the network view by observing traffic related to services such as LLDP, DNS and DHCP.

It should be clear by now that, although the term ‘forwarding’ is used, this does not refer to L2 forwarding. This is because the examined fields include L3 information. Similarly, an Openflow switch does not perform L3 routing. There is no longest-prefix-match, or any other complicated calculation that takes place on the switch. In fact, the protocol does not define how the forwarding decisions for specific header fields (i.e. the actions) are made. The decisions are made by the programmable controller and are simply installed in the switches’ flow tables. Openflow switches address the flow tables and match the incoming packets’ header fields to pre-calculated forwarding decisions, and they simply follow these decisions.

The **secure channe**l is the interface that connects each Openflow switch to a controller. Through this interface the controller exchanges messages with the switches in order to configure and manage them.

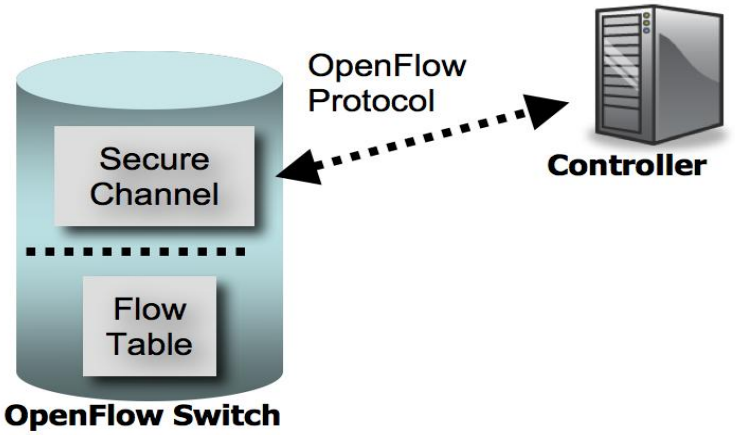


Figure 4: Idealized Openflow Switch. The FlowTable is controlled by a remote controller via the Secure Channel.

Openflow provides a protocol for communication between the controller process and the Openflow switches. There are three types of messages supported by the Openflow protocol. The **controller-to-switch,** the **asynchronous** and the **symmetric** messages. We will briefly describe these three types of messages. For further study, the Openflow specification [1] provides an excellent source of information.

The **controller-to-switch messages** are initiated by the controller and may not always require a response from the switch. Through these messages the controller configures the switch, manages the

switch's flow table and acquires information about the flow table state or the capabilities supported by the switch at any given time.

The **asynchronous messages** are sent without solicitation from the switch to the controller and denote a change in the switch or network state. This change is also called an event. One of the most significant events is the packet-in event which occurs whenever a packet that does not have a matching

flow entry reaches a switch. When this happens, a packet-in message is sent to the controller, containing the packet or a fraction of the packet, in order for the controller to examine it and determine which kind of flow should be established for it. Other events include flow entry expiration, port status change or other error events.

Finally, the third category of Openflow messages are the **symmetriconous messages** which are sent without solicitation in either direction. Those can be used to assist or diagnose problems in the controller-switch connection.

4.2 Openflow use for network virtualization

The controller-based, Openflow-enabled network architecture described previously, aims to gather the network logic in a centralized entity, the controller. The controller is responsible for all forwarding and routing decisions by managing and manipulating the flow-tables on the Openflow switches. Once traffic is logically organized in flows, its manipulation becomes easier and much more straight-forward.

This unique feature of Openflow can be used towards achieving network traffic isolation. By grouping together flows with different characteristics we create logical partitions of the common physical network infrastructure. If we map these groups of flows to different logical partitions and store this mapping in a centralized entity that has a complete image of the physical network, we will have created a flow-based virtual network abstraction on top of the physical overlay.

SECTION B

Design

Chapter 5

Openflow Network Virtualization

5. Openflow Network Virtualization

A number of representative virtualization techniques and architectures have been presented in previous chapters. An interesting conclusion deriving from this short survey is that most of the available network virtualization solutions today do not intend to provide a straight-forward definition of a virtual network. The definition is certainly implied but is not the starting point for proposing a certain technique or architecture. It could be said that all existing solutions accede to the loose concept of network virtualization being the partitioning of the physical network infrastructures to logical networks and procede in implementing it.

In the case of VLANs and VPNs the virtual network boils down to a set of endpoints identified through their MAC/IP addresses or services (essentially TCP ports) operating on them. In the context of PlanetLab and VINI virtual networks are overlays on top of the existing Internet infrastructure. GENI and FEDERICA have a more abstract and holistic approach, being technology agnostic and providing full virtualization of nodes, network devices and links.

Our approach is somewhat different. The definition of a virtual network is a crucial step that will enable us to further propose an Openflow-based architecture. The distinguising factor between our approach and the existing network virtualization approaches lies in the fact that our toolset for achieving network virtualization was very specific. Hence we were able to be very specific with our virtual network definitions. Having a definition as a starting point we can then move on to designing an Openflow-based architecture.

5.1 Towards the definition of a virtual network

The attempt to come up with a straight-forward, concrete definition of a virtual network that would allow us to further develop an Openflow-based architecture invoked a number of interesting issues. Many possible approaches for defining a virtual network were examined. We will briefly mention the alternative solutions that were considered along the way and the reasons why they lacked to provide an adequate means of defining an Openflow virtual network.

An approach that comes naturally to mind is to think of a virtual network in terms of the endpoints, links and network devices it comprises of. Similarly to PlanetLab's or FEDERICA's approach, a virtual network can be defined as a slice which consists of a set of virtual links, endpoints and network devices. End-users can opt in, requesting a set of resources which they can utilize in their preferred way, while being assured both high levels of isolation and interconnection with other existing virtual or real networks. This definition of a virtual network is resource-based, meaning that a virtual network can be adequately defined by the set of resources it consists of. Although this kind of approach is rather straightforward and focused on the hands-on usage of virtual networks, it is very high-level and does not make use of Openflow's capabilities. As described in the previous chapter, Openflow enables the logical partitioning of network traffic in flows. Therefore, in order to leverage this unique capability, we should no longer think in terms of virtual links, end points and devices but rather in terms of traffic that goes through this set of resources. This way we can take advantage of the Openflow capability of describing traffic in terms of flows.

The next question that comes to mind is how many and which of Openflow’s ten-tuple fields can be used to describe a virtual network. Following the same approach as VLANs for example, a set of MAC/IP addresses or services (TCP ports) can mapped to one virtual network. However this approach is not flexible enough, since it should be possible for an endpoint with a certain MAC address to belong to more than one virtual networks. And VLANs fail to cover this case. Additionally, the IP address approach assumes IP connectivity.

In order to allow generality and produce a more abstract, hence more flexible, definition of a virtual network, it is essential to impose as few restrictions as possible. Instead of using MAC/IP addresses or services to define a virtual network membership, we can assume that everything that can be found on a packet header, i.e. IP, MAC, port info or any possible combination of this information, can be used to define virtual network membership.

Instead of defining a virtual network through all possible information found on an individual packet, we can group packets in flows while preserving a certain level of generality. For example, a flow could be uniquely identified by an IP source and an IP destination address, but this statement makes the assumption that the packet is an IP packet. Therefore we need to use more fields from the packet's

header to be able to avoid assumptions while adding flexibility to our definition. MAC source/destination addresses are possible candidates. Following this concept, we conclude in a set of fields that could provide the building blocks for creating customized virtual network memberships.

Following this idea, virtual networks can be defined not by the purpose that they serve, but by the kind of traffic that they carry. While these two concepts sound similar, they are fundamentally disparate: Mapping a virtual network to human-recognizable ideas, such as the service it provides or the hosts that use it, is bound to provide a constrained definition. The goal of this work is to provide a realization of virtual networks that are defined solely by the traffic that flows within them. It is a way of looking at virtualization from a network point of view, instead of the users' point of view. In this sense, *a virtual network can be defined as a sum of traffic flows*. This definition will become clearer as we revisit it in paragraph 5.3.

Openflow is a protocol that provides this kind of flow-based network abstraction. As mentioned in the previous chapter, a flow can be identified by a 10-tuple which is part of the Openflow header [1]. By using flows, we can achieve the following:

- One endpoint can belong to one or more virtual networks.
- The definition of a virtual network becomes general hence virtual network membership can be customized according to the special requirements of the network.

In a production network there needs to be a way to map a large amount of flows to virtual networks. This should not significantly affect the network's performance or create an increased workload for the network devices.

This chapter will go through the steps followed during the design phase, highlighting these decisions that were of critical importance to the architecture while presenting the alternative options that were available at any time. Further on, a set of terms and definitions that will be used in the context of the architecture will be presented. Finally, an overview of the final architecture will be briefly described and the high-level functionality of the various components comprising it will be discussed in detail.

5.2 Design Steps

This section describes the evolution of our design. Each paragraph describes an extension to the approach of the previous paragraph, in an attempt to deal with shortcomings of every approach. These are considerations that had to be taken into account, and are provided as an indication of the challenges involved with an abstract design. They are not necessarily included in the actual architecture proposal, which is described in section 5.3.

The first steps in our design stage were made with network virtualization for university campus networks in mind. In this context, a use case scenario of the proposed architecture would be to provide several virtual slices of the physical campus topology, each of them allocated to groups of researchers for experimental purposes. The target would be to allow research and experimentation on the available infrastructure, without affecting the operation of the campus production network. The benefits of deploying network virtualization in these environments are several, the most considerable of them being the ability to utilize large experimental topologies at no extra cost for equipment, or the need to establish distinct experimentation environments.

5.2.1 Flow establishment

5.2.1.1 Preconfigured flows

With the above scenario in mind, our initial thoughts involved the configuration of virtual slices on a centralized supervising entity, and the establishment of matching flows that instantiate the configured slices. In this first approach, the administrator configures the virtual networks by providing a description for each network to the controller. The description is a topological description of the virtual network, which involves the definition of the network devices and their interfaces that constitute the slice.

Depending on the capabilities of the controlled protocol, the slice configuration can optionally include additional information, used to customize the level of abstraction of the slice description. For example, in the case of OpenFlow, the administrators can provide fine-grained descriptions by explicitly specifying host identifiers (MAC/IP addresses), type of traffic (port), etc. However, if no additional information is provided, the slice will still be abstractly defined as a subset of the physical topology, with no host, traffic, or even protocol restrictions.

Upon the configuration of a slice, flow entries that match the given configuration are automatically set up on the appropriate network devices, forming the slice. This way, the flows are pre-configured, and are ready to be used whenever a host tries to use the slice from a designated location. This could refer to any host connected to a designated location or specific hosts connected on designated locations, depending on the granularity of the configuration as described in the previous paragraph.

While this simple approach provides some level of abstraction, this is far from the desirable generality. Pre-configured flow entries might be useful for virtual networks with very specific requirements, hosts and general usages (e.g. applications or type of traffic), but this kind of configuration is too static and does not favor scalability. Moreover, in order to abide by the principle of abstraction, the definition of a virtual network would ideally be independent of the network devices that comprise it, and a virtual network will not bound to a specific overlay of the physical topology, but the used paths can shift dynamically.

5.2.1.2 Dynamic flows with host identification

An alternative solution that is a step closer to the desired abstraction involved the dynamic establishment of flows upon user request. In this case, when a user would like to use a virtual slice, they would first contact this supervising entity, which, after authentication, would provide the user with a list of configured virtual slices of the network. The virtual slices listed there, would depend on the user's credentials, and/or the physical point of connection to the network. The user would then choose to join one of the available virtual networks, and after authentication the host would be a part of the selected slice. After that point, both the underlying production network and the other available virtual slices would be transparent to this host.

A simple analogy of the aforementioned scenario is the process of authentication to wireless LANs. In WiFi, the physical medium is the air. In a given environment, several groups of users utilize this medium by configuring and operating separate networks on top of it. A WiFi user retrieves a list of advertised beacons and chooses to use one of them. After authentication, the host's network stack is unaware of other networks on the physical medium. Of course this is irrelevant to virtualization, since network isolation is achieved on the physical layer. However this describes the desirable functionality. In our case, the list of available networks is provided by a supervising entity that maintains their

configurations, and just like WLANs are isolated and used by different groups, the virtual slices are ignorant to each other's existence although they share the same medium; the campus network.

This approach still raises several limitations that we wanted to avoid. One significant restriction is that each host can only operate on a single OVN at a time. This is something that contradicts the desirable generality of our objective. Virtualization becomes most efficient when it is realized in multiple forms. Network virtualization that forces a host to operate on a single slice automatically negates the benefits of server virtualization on that host. In other words, one machine would be unable to host several virtual servers that would operate on separate network slices. This kind of loss of generality should be avoided.

Moreover, in the above scenario virtual network membership identification essentially takes place on the application layer, and is based on user credentials. In this way users are aware of the slices, making virtualization an opaque function. Thus, while this design introduces a centralized virtual network management entity, it does not seem to abolish several undesirable, restricting characteristics.

The objective of our work has been to reach a definition of virtual networks which would be as abstract - and thus as customizable - as possible. A nice feature enforcing this kind of abstraction would be to completely conceal the concept of virtualization from the end users as well as the network devices. We wanted the notion of virtual slices to be manifested only on the control plane, which is in this case represented by our supervising entity, the controller.

5.2.2 Path Calculation

The above considerations also raised the problem of defining a method of selecting a physical path in order to connect two points of a slice. Even for this simple approach, the connection locations to a virtual network are only half of the required information that defines the network. In order to describe it in its entirety, we actually need to describe how these locations are connected, by setting up the flow entries along available physical paths.

It is thus obvious that there is need for an entity that, given the network topology and a set of routing requirements, can calculate a path which connects two points of the network. So the question that rises is where should this path calculating entity fit into our architecture? The selection of a

physical path is a separate and complex process. Moreover, considering the variable purposes of network slices, the path selection might be required to be based on different algorithms for different networks. These considerations led us to separate the path finding functionality from the rest of the architectural mechanisms.

The notion of an offline path-finding entity has been discussed a lot in the research community during the last decade. This trend has been followed by researchers working on routing protocols in general, and is not strictly related to virtualization. RFC 4655 [40] discusses a network architecture based on an offline Path Computation Entity (PCE). While the purpose of the PCE in that document is to run constrained based routing algorithms for MPLS networks, the benefits provided by a path finder as a separate finding entity are the same.

5.3 Terms and definitions

Having discussed several initial alternative approaches, we will proceed to the proposed architecture. However, first, certain terms that will be used throughout this paper should be defined:

- We will refer to any machine that sends or receives data through an OVN as an **Endpoint**.

- An **OF switch** is a switch that supports the Openflow protocol ('type 0' switch as defined in the Openflow whitepaper [1])

- The term **Access Point** will frequently be used for describing an ingress/egress port for the network. An access point is represented by a [switchID, switchPort] pair.

- A **path** is a physical route between two access points.

- The term **OF 10-tuple** will refer to the 10 header fields found on every packet that enters the OF network, as described in the previous chapter.

- **Flow entries** are forwarding rules stored in an OF switch's flow table. These rules provide a forwarding decision for an incoming packet based on its OF 10-tuple.

- The term **flow** refers to traffic that matches a specific OF 10-tuple. This definition is traffic-oriented in a sense that it only reveals information about the values of the header-fields of the packets that constitute the flow and not the path that they follow through the network. In this sense, a combination of flow entries on several OF switches instantiates a flow, binding it to a specific path.

- A sum of flows can thus adequately define a subset of the total traffic that goes through the physical network. Our approach to a virtual network is to define it in an abstract way using the traffic that goes through it. Therefore an **Openflow Virtual Network (OVN)**, can be defined as a sum of flows. This provides the desirable abstract definition, which is based solely on the type of traffic that goes through the virtual network. Inductively, the definition of the OVN does not imply anything about the paths that the OVN traffic follows along the network, or the endpoints that use the OVN. These could and should be able to change dynamically in order to adapt to network changes or failures without affecting the OVN definition.

5.4 Additional components for the proposed architecture

The proposed architecture comprises of all the components needed for an Openflow-based network as described in the previous section, as well as some additional entities which provide OVN abstraction. These entities are the following:

- **Administrator:** The administator is the entity that provides OVN abstraction to the network. It maintains a database of OVN entries called the **OVN database.** Furthermore it provides an interface for adding, deleting and modifying OVNs on that database. Being the only entity which has a global view of existing OVNs, the administrator is also responsible for OVN identification of a packet that is introduced to the network. The administrator could be seen as a process that runs on the same machine as the controller or a separate machine in the network. There could be one or more administrator processes as long as they have access to the same or to exact copies of the OVN database.

- **Path-Finder:** An entity that is responsible for making the routing decisions within an OVN. This process takes 2 access points (an ingress port, and a destination port) as input and, knowing the network view at a particular time, calculates the path that a packet should follow

through an OVN. This entity could also run separately or on the same machine as the controller, and there can also be one or more path-finder entities in the network.

The figure below describes the basic architecture of an OVN enabled Openflow network

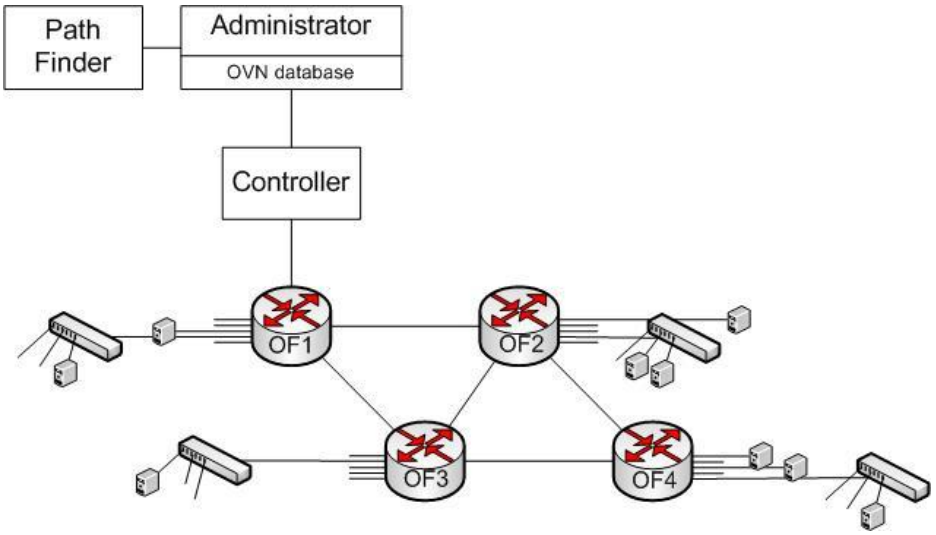


Figure 5: An OVN enabled Openflow network

5.5 Components in detail

5.5.1 Administrator

As mentioned in the previous paragraph, the administrator is the entity that provides OVN abstraction within the network. It is the only entity that can understand and interpret this abstraction. The administrator maintains and has access to the OVN database which is the core information tank of our architecture. The OVN database contains one OVN entry for each of the existing OVNs. The format of the OVN entires is shown in the following figure:

Name	Id	Properties	Services	Datapaths	Auth
Professors	1	[...]	Http 10.10.0.100 nl:1 port:2	All	Yes
MyRouting	2	[...]	Dns 192.168.1.10 nl:2 port:4	All	Yes
BobSlice	3	[…]		dp2, dp4, dp5, dp9, dp10	No

Figure 6: The OVN database

The various fields contained in an OVN entry are explained below:

- **Name**: Name of the OVN, chosen by the OVN owner.
- **Id**: Unique identifier for the OVN, incremented for every added ovn.
- **Properties**: An array of properties of OVN traffic according to the OF 10-tuple definitions, based on which the administrator can decide if a packet belongs to this OVN. Each property can be assigned more than one value, as well as wildcard values. These properties are described in the next paragraph.
- **Services:** List of services running on the OVN, host of the service, switchID and port where the service resides.
- **Auth**: indicates whether access to the ovn requires authentication.
- **Datapaths:** Indicates which OF switches are available for this OVN. This parameter is optional. An empty field indicates that the whole physical topology is at the path-finder's disposal, and the OVN can consequently expand anywhere on the network. A non-empty value requires a path-finder that supports route calculation based on given topology restrictions. Although this field introduces path restrictions for an OVN, the fact that it is optional does not affect our OVN definition which does not associate an OVN to the paths it uses. An implementation that supports a restriction-based path finder and makes use of this field, binds an OVN to a subset of the openflow network topology. The datapath restrictions can be either loose or strict. In the case of loose restrictions, when the path-finder cannot provide a path using the preferable set of datapaths, it looks for an alternative using the whole topology. If the datapath restrictions are strict and no path can be calculated using the designated datapaths, the path-finder notifies the administrator that there is no path available.

The properties array is the most significant element of an OVN entry. OVN identification for a new packet is based on this array. When a packet that is received by an OF switch matches an existing flow in its table (be it either a production network flow or an OVN specific), it will be forwarded

accordingly. If however the packet cannot be matched to any flow, it will be forwarded to the controller. The controller will in turn consult the administrator process which will have to deduce which OVN the packet is intended for. In order to do so, it will need to match information found on the packet's OF 10-tuple to the properties that the owner of the OVN has defined for the OVN. These OVN properties are held in the OVN database and they correspond to 10 fields defined in an OF 10-tuple. The only difference is that the original ingress port field is now replaced by an access point. Each field can be assigned multiple values, for example many source MAC addresses, or many possible destination ports. The sum of all different values for every field provides a description of all the traffic that belongs to this OVN. It is the responsibility of the OVN owner to define the OVN traffic as strictly as possible. The more information the administrator is provided with, the finer the granularity of the flow entries corresponding to the OVN.

Connected Ports [switch, port]	**Ether Src**	**Ether Dst**	**Ether Type**	**Vlan ID**	**IP Src**	**IP Dst**	**IP Proto**	**Src Port**	**Dst Port**
[1,3] [1,4] [3,2] [4,1]	Any	Any	0x0800	-	Any	Any	-	-	4444

Figure 7: The Properties array of an OVN entry.

The elements of the properties array are explained in detail below:

- **Connected ports**: A list of access points (pairs of <switchID, port>), indicating which ports of the physical topology have been enabled for the OVN.

- **Ether type**: The hexadecimal code for the ethernet protocol used in this OVN. If multiple OVNs are matched based on this, the tiebraker is one of the next properties.

- **Ether src/dst**: Ethernet addresses registered in this OVN. If a packet is originated/destined from/to one of these addresses, it can belong to this OVN. If multiple OVNs are matched based on this, the tiebraker is one of the next properties.

- **VLAN id:** reserved – could be used as ultimate tiebraker.

- **IP src/dst**: IP addresses registered in this OVN. If a packet is originated/ destined from/to one of these addresses, it belongs to this OVN. If multiple OVNs are matched based on this, the

tiebraker is one of the next properties.

- **IP proto:** The hexadecimal code for the ethernet protocol used in this OVN.

- **Src/dst port :** Ports on which OVN-specific applications operate.

5.5.2 Path-finder

The path-finder is the entity responsible for making routing decisions within OVNs. The path-finder provides a path in the form of a list of consecutive access points. The administrator process acquires this list, translates every two consecutive access points from the list into a flow entry for a specific switch and instructs the controller to invoke the flow entry creation in the respective switches. The process through which a path is calculated is left at the discretion of the implementation of the path-finder. The routing decision can be based on one or more of the following criteria:

- **Conflicts** (mandatory): Any newly created flow entry should not conflict with the pre-existing flow entries on a switch.

- **Hops** (optional): Select the path with the minimum number of hops

- **Link costs** (optional): Select the path with the minimum total cost

- **Flow table sizes** (optional): Select the path in a way that the average flow table size is maintained at a minimum.

- **Datapaths** (optional): Calculate a path using a subset of the available OF switches, which is provided by the administrator upon a path request

- **User defined factors (optional):** The user can define their own metrics for example request that a specific switch is not included in the path.

5.6 Related Work

Rob Sherwood et al, have proposed a special purpose Openflow application called Flowvisor [32], which enables experiments by different researchers to run simultaneously on the same Openflow network. FlowVisor creates slices (OVN equivalents) of the same physical network which are

independent and isolated from one another. Flowvisor uses the same virtual network definition, where a set of flows define a virtual network. However the architecture differs. Each slice is controlled by a different controller and Flowvisor acts as a proxy, siting between the experimenters' controller and the OF switches in the network. Flowvisor acts as a transparent multiplexer/demultiplexer of controller-switch communications in such a way that controllers talking to Flowvisor believe that they communicate with their own dedicated switch, although another controller might be using another part of the same switch through the Flowvisor. Similarly, in the other direction, an Openflow switch thinks that it is communicating with a single controller, although the Flowvisor might be aggregating commands from several controllers, and dispatching them to different parts of the switch. Flowvisor makes sure that packets are forwarded according to the virtual network they belong to by modifying the Openflow messages received from the OF switches and controllers accordingly. Flowvisor allows for recursive delegation of virtual networks. This means that a slice can be further delegated to one or more experiments.

Based on cluster computing architectures where the resources of several distributed servers are combined to form one virtual server, Fang Hao et al., [45] propose an interesting approach for implementing network virtualization. The main purpose of this architecture is to assist the migration of services in data centers. Service migration refers to services moving between different Virtual Machines distributed along the network, in order to minimize user delays and improve performance. The work proposes a centralized architecture where different FEs (Forwarding Elements) distributed along the network, form a a virtual router, called VICTOR (Virtually Clustered Open Router). The sum of different FEs can be seen as a set of input and output ports on the virtual router. A VICTOR is controlled by a centralized entity called the CC (Centralized Controller) which gathers the control plane functionality of many VICTORS. Each VICTOR is allocated for a different virtual network. The advantage of this architecture is that there is a large pool of available ports on the virtual router in different locations since the FEs are distributed along the network. This way traffic can be routed in different ways according to parameters such as the location of the end hosts.

Having presented the main components that comprise our architecture and go through their functionality in detail, the following chapter will focus on the high-level implementation of these components covering issues such as the communication between them.

Chapter 6

Openflow Virtualization API

6. Openflow Virtualization API

This chapter describes in further detail the architecture proposed in Chapter 5. Specifically, we define a) the communication protocol used between our entities, b) the functions performed by each entitiy and c) the interface between the controller and our entities.

6.1 Communication between entities

Communication between the OF switches and the controller is taken care of by the Openflow protocol and has been briefly described in previous sections. The introduction of the administrator and path-finder entities which provide the OVN abstraction in the existing architecture mandates a need for communication between these entities. It also requires slight modifications in the behaviour of the controller as described previously. Before the OVN abstraction was introduced, it was the controller that was responsible for invoking the addition, deletion and modification of flows in the flow tables of the OF switches. This task is now delegated to the administator entity which, based on the OVN information found in its OVN database, will decide which flows will be added, deleted or modified and will hence instruct the controller to invoke these changes on the respective OF switches. It becomes obvious that an additional interface should be introduced, providing communication between the controller and the administrator process and enabling the support of the OVN abstraction from the controller side.

6.1.1 Communication between Controller and Administrator

We have defined 3 different types of messages for the communication between the controller and the administrator. The documentation as well as the source code refers to these messages as CA messages. These messages constist of the CA header, and possibly CA data. The CA header contains 3 fields:

- Packet type : Indicates type of CA message (PUSH_PACKET, OVN_HIT, OVN_MISS)
- Length : The length of the CA data segment
- Packet ID : Used to identify the transaction between controller - administrator

6.1.1.1 Controller-to-Administrator

These messages are sent from the controller to the administrator in an asynchronous manner, everytime a packet-in event (with reason:'no match') occurs at the controller.

- **CA_PUSH_PACKET:** As mentioned previously, for all packets that do not match any flow entry, a packet-in event along with the full packet or a fraction of it, is sent from the OF switch to the controller. The controller at this point needs to push the packet or the fraction of the packet to the administrator process in order for the administrator to perform the OVN lookup and determine which OVN the packet belongs to. This is done by sending a Push-Packet message to the administrator. The data segment in this case is the OF 10-tuple of the packet that triggered the packet-in event. (If the switch that raised the packet-in event supports buffering, it will only send the 10-tuple to the controller. Other wise the swtich will push the whole packet to the controller and it will be the resbonsibility of the controllers ovn interface to strip the unnecessary fields and only pudh the OF 10-tuple to the administrator)

Type	Length	PacketID		OF 10-Tuple

PUSH_PACKET message

6.1.1.2 Administrator-to-Controller

- **CA_OVN_HIT:** As mentioned previously, Modify-State messages are sent from the controller to the switch in order to add/delete and modify flows in the flow tables. In the context of the OVN architecture, it is the administrator that decides which flows will be added, deleted or modified at a particular time. Therefore the administrator sends a Modify-Flow message to the controller containing all the necessary information in order for the controller to construct the Modify-State message that will be sent to the respective switch or set of switches. The data segment of this message is a list of switched and flow-mod commands that should be issued on each switch.

Type	Length	PacketID		Flow-mod commands

OVN_HIT message

- **CA_OVN_MISS:** The administrator sends this packet to the controller in order to notify it that the

requested packet was not matched to an existing OVN. The packet has no data field, as the CA header is enough to let the controller know that the specified packet is production network traffic and should be passed to the underlying controller functions.

Type	Length	PacketID	

OVN_MISS message

6.1.2 Communication between Administrator and Path-Finder

There are only 2 messages involved in our current definition for this communication: one message for each direction. Similarly to the CA messages, the message used here are called PA messages and have header containing the same fields (Packet Type, Length, Packet ID).

6.1.2.1 Administrator-to-Path-finder

These messages are sent from the administrator to the path-finder entity

PA_PATH_REQUEST: Through this message the administrator requests a path from the path-finding entity. The Request-path message sent by the administrator contains the two access points (an ingress port, and a destination port) that the path-finder needs to know in order to calculate the path. The Packet ID is again used to tag the request. The request message may optionally include datapath restrictions, asking the path-finder to calculate a path that uses the designated OP switches. This occurs when the 'Datapaths' field of the corresponding OVN entry in the OVN database is not empty. This optional parameter requires a path-finder that supports this functionality. The parameter is provided as a list of datapath Ids.

- ✓ **Note:** There have been discussions about adding a 'Capabilities' message in order to advertise support for the optional 'datapaths' parameter. Although this could be useful in future work, at this point we only mention it as a possibility and do not standardize it or implement it, as its format would depend on forthcoming extensions.

Type	Length	PacketID		AccessPoint1	AccessPoint2	[datapaths]

PATH_REQUEST message

6.1.2.2 Path-finder-to-Administrator

- **PA_PATH_REPLY:** A message containing a list of consecutive access points that indicate a path through a specific OVN. The administrator will use this list to invoke the corresponding flow-mod commands through the controller. Again, Packet ID is used to map the path to the corresponding request.

Type	Length	PacketID		path

PATH_REPLY message

6.2 Entity Functions

6.2.1 Administrator API

As we mentioned earlier, the administator is the only entity which is aware of the existence of OVNs. As such, there needs to be an interface between it and the OVN owners, in order to provide the administator with OVN descriptions. The following set of OVN functions has been defined for this purpose:

- **createOvn (*ovnName*):** This function creates a new OVN with the name *ovnName.* When a new OVN is created, a new entry is added in the OVN database of the administrator process. Such new entries are initially simple frameholders for the newly created OVN. The OVN is not functional until some switches/ports have been attached to it.

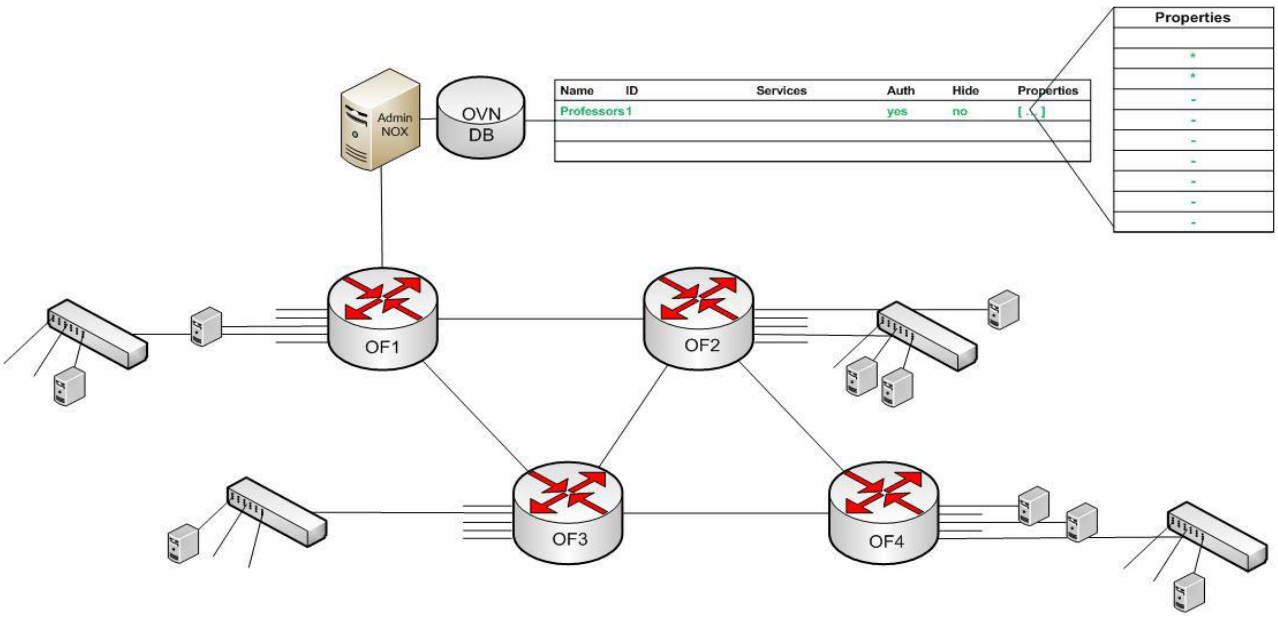
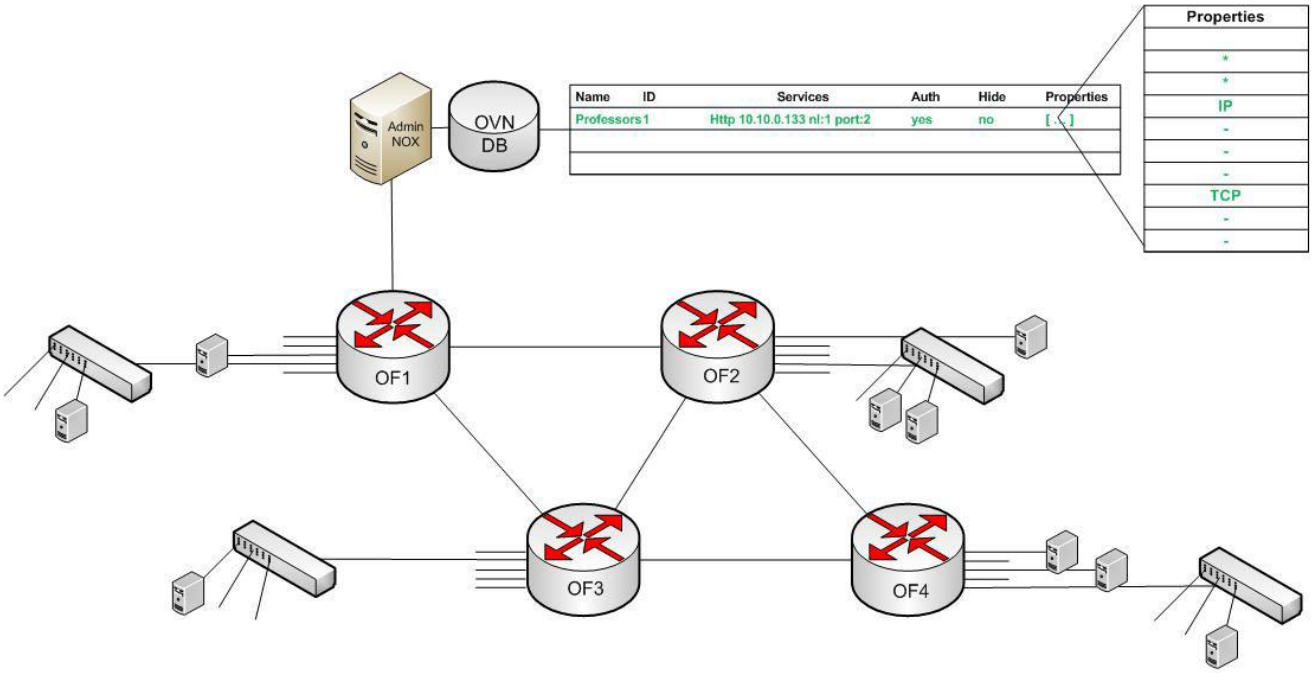


Figure 8: CreateOVN(Professors) adds a new OVN entry in the database.

- **addProperty (*ovnName, propertyName, value*):** This function adds a new rule in the property list field of the *ovnName*'s entry in the OVN database. Before a property is added, the administrator process must check for conflicts with properties of other OVNs. A conflict occurs if for two OVNs who have the same port enabled on any switch, a property with the same value is added (that goes also for wildcard value overlapping). In this case the addProperty function will return an error.

Figure 9: The commands addProperty(Professors, Ether proto, IP) and addProperty(Professors, IPproto, TCP)



add two properties for the Professors OVN.

- **removeProperty (*ovnName, propertyName, value*):** This function removes a property from the property list field of the *ovnName*'s entry in the OVN database.

- **addPort (*ovnName, switchID, port*):** This function attaches a port of an OF switch to *ovnName*. After this function is called, an endpoint connected to *port* can be a member of *ovnName.* The function does not set up any flows, but merely indicates that *port* is now enabled for *ovnName*. When *port* is successfully attached to *ovnName*, the 'Connected Ports' field in the 'Properties' list of the *ovnName* entry in the OVN database needs to be updated with the pair *<switchID,port>*.

- **addPorts (*ovnName, [(switchID, port), (switchID, port),...]*):** Same as above, but can be used to add multiple ports on multiple switches to *ovnName*.

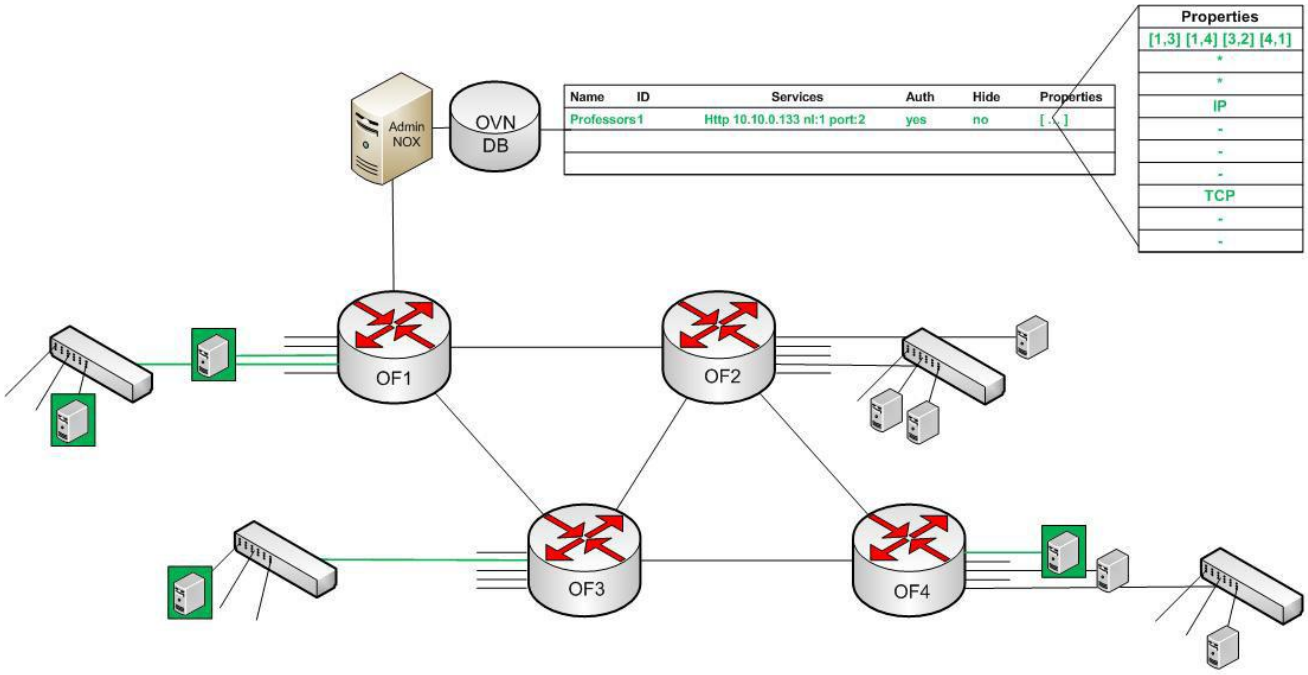


Figure 10: addPorts(Professors, [1,3], [1,4], [3,2], [4,1]) enables the Professors OVN on 4 access points.

- **removePort (ovnName, switchID, port):** This function detaches *port* from *ovnName*. The pair *switchID,port* is removed from the 'connected ports' field of the *ovnName* entry in the OVN database. Endpoints connected to *port* can no longer be part of *ovnName*.

- **removePorts *(ovnName, [(switchID, port), (switchID, port),...]):*** As above, but used to detach multiple ports on multiple switches from *ovnName*.

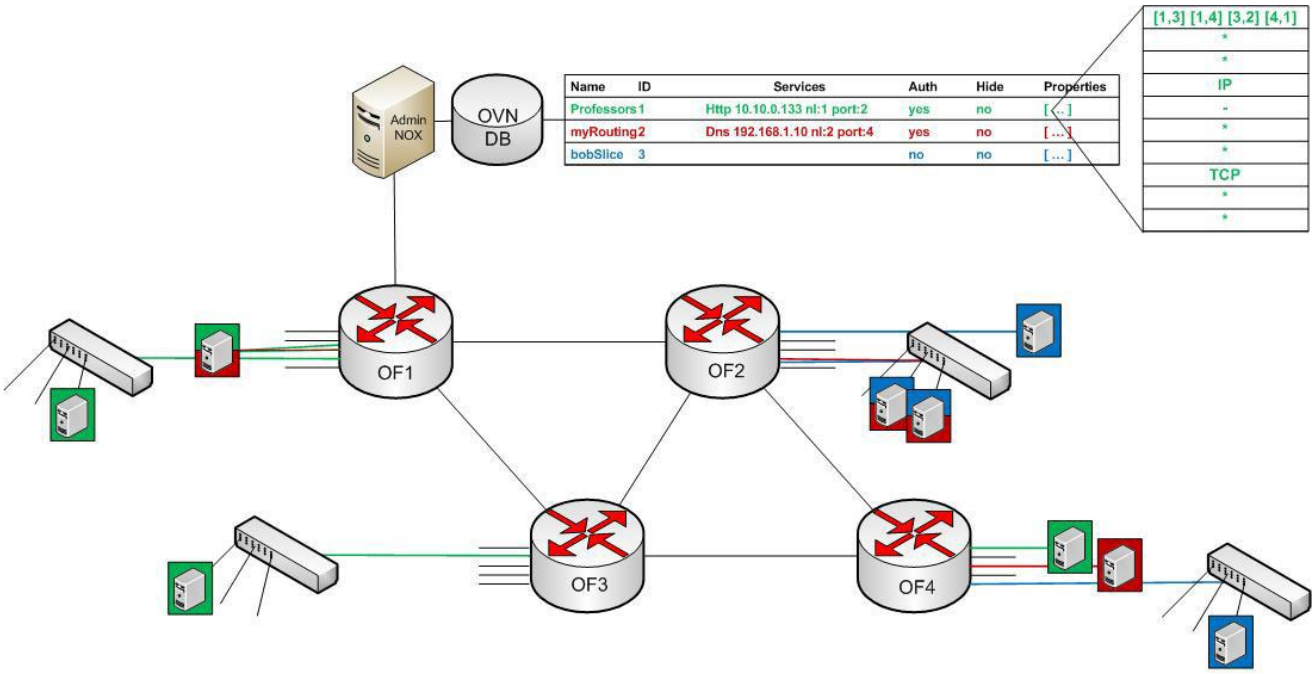


Figure 11: An example view of an OVN enabled Openflow network after the definition of several OVNs

Apart from the operations on the OVN database, the administrator needs some functions that implement the connectivity to the path-finder as well as the controller.

- **SendPathRequest** ***(<switchID1, port1>, <switchID2, port2> , [datapaths]):*** This function is used to send a Request-Path message to the path-finder

- **SendOvnHit** ***(packetID, commands):*** This function sends an OVN_HIT message to the controller, asking it to invoke flow-mod commands on OF switches.

- **SendOvnMiss** ***(packetID):*** This function sends an OVN_MISS message to the controller, indicating production network traffic.

6.2.2 Path-finder API

The path finder communicates with the administrator process each time a route needs to be calculated. This paragraph describes that functions that are performed on the path-finder when invoked by the administator.

- **findPath** ***(pathID, [(switch1, port1), (switch2, port2)):*** This function calculates a path from

the first access point to the second access point of the parameters. The function returns a path in the form of cosecutive access points, e.g. [***(switch1, port1), (switch1, portA), (switchX, portB), (switchX, portC), ... (switchY, portD), (switchY, portE), (switch2, portF) (switch2, port2)]***

- **sendPath** ***(pathID, path):*** This function sends a PA_PATH_REPLY message to the administator. The *path* parameter is a list that was returned by the *findPath* function. The administator uses *pathID* to map the newly learnt path to the correct request.

6.2.3 NOX OVN interface

The following describes a NOX application which is used as an interface between the NOX controller and the OVN administrator process.

The interface should intercept all packets arriving to the controller from the network, and forward them to the administrator so that it performs the OVN membership lookup. At this point, the administrator will either identify the packet as traffic belonging to an OVN, or conclude that it is production network traffic. In the latter case, it will simply inform the OVN interface that the packet can be treated as production traffic, and the OVN interface will push it the underlying NOX apps. If, however, the packet is matched to and OVN, the administrator will consult the path finder, get a valid path to the destination, and send the appropriate flow-mod instructions to the OVN interface. The OVN interface will in turn invoke flow-mod commands on the corresponding datapaths, and send the packet back to the network, in order to follow the newly establish flow.

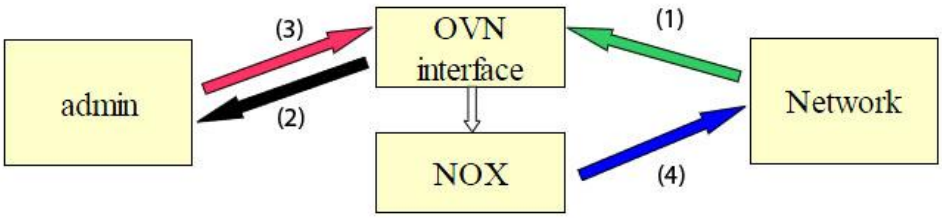


Figure 12: The OVN interface for NOX

Figure 12 illustrates the following OVN interface callback functions:

- **packet_in_event (1)** → Send **PUSH_PACKET (2)** message to administrator. Buffer the packet

until a response from the administrator is received.

- **OVN_MISS (3)** → Post a **packet_in_event (1)**, as originally received from the network, to other nox apps. (Occurs when the administrator fails to match a packet to an existing OVN. Packet is treated as production network traffic. This has the same impact as if the OVN interface didn't exist.)

- **OVN_HIT (3)** → Instruct NOX to send **Modify-State (4)** messages to corresponding datapaths, as instructed by administrator. Send the buffered packet back to the network (i.e. instruct the datapath that raised the original packet_in_event to send it out a specific port according to the new flow)

SECTION C

Implementation

Chapter 7 Implementation

Chapter 8 Test Environment

Chapter 7

Low level Implementation

7. Implementation

This chapter delves deeper into the lowest level of the implementation. While chapter 6 decribes higher level functionalities and APIs, this chapter will go through our own implementation of the entities and the communication between them.

7.1 Implementation of the Entities

7.1.1 Controller OVN interface

The Controller OVN interface is an interface attached to the controller (in our case NOX) for OVN support. This interface is a python script and it is implemented as any other NOX application. The main requirement for this interface was to be totally transparent to the controller itself, in case of production network traffic. This means that it should intercept packet-in events, and if a packet is not matched to an existing OVN, it should be passed to the controller which will handle it as if the OVN interface never existed.

The OVN interface, just like any NOX application, attaches a behaviour to the controller, instructing it to perform specific procedures upon particular openflow events. The NOX controller can prioritize the order with which the running applications will handle events. Putting the OVN interface on top of that list lets us intercept packet-in events, and, if needed, pass them on to the underlying applications. This accomplishes independent handling of OVN and production network traffic.

> ✓ **Note:** The description provided here will be kept at a high level. For a more detailed description as well as instructions, the interested reader can refer to our project's wiki for documentation and source code [42]

Our implementation was written in Python. The OVN application, defined in *ovn.py*, is in constant communication with the administrator. The communication specifics are covered in the next paragraph. When the application receives a packet-in event, it buffers the packet, and creates a CA_PUSH_PACKET message which is forwarded to the administrator. The forwarded packet

essentially contains the original packet's OF 10-tuple along with a datapath id (this is what the administrator requires in order to perform an OVN lookup) and a buffer id.

A reply from the administrator can either be a CA_OVN_HIT, or a CA_OVN_MISS message. In case of the latter, the event is simply passed to the underlying NOX applications as production network traffic. In case of an CA_OVN_HIT, the reply from the administrator will also carry the path through which flow entries must be established, as well as a buffer ID. The OVN application retrieves the buffered packet using this ID, extracts the packet's flow, and sets up corresponding flow entries on switches along the path provided by the administrator.

This functionality will setup a flow from the source to the destination, but not vice versa. In order for the communication to be 2-way, the OVN application currently sets up a returning flow along the same path.

7.1.2 Administrator

7.1.2.1 The OVN database

The OVN database is a standard MySQL database that contains two tables: the *OVNEntry* and the *Properties* table. The tables format is shown below:

OVN id	OVN name

Figure 13: OVNEntry Table

OVN id	Connected Points	Ether Src	Ether Dst	Ether Type	VLAN id	IP Src	IP Dst	IP Proto	Src Port	Dst Port

Figure 14: Properties Table

The *OVNEntry* table consists of the fields *OVNname* and *OVNid*. The *OVNname* field serves as a user-friendly identifier for the OVN. The *OVNid* field is a unique identifier for the OVN and is used as an index field, that is to provide a mapping between OVN entries in the *OVNEntry* table and their respective properties in the *Properties* table. The *Properties* table consists of the *OVNid***,** the *ConnectedPoints* field which specifies which access points have been enabled for a specific OVN, as well as the 10-tuple fields which are used to describe the kind of traffic that belongs to a specific OVN.

The free software phpMyAdmin written in PHP has been used to handle the administration of the OVN database. PhpMyAdmin supports the majority of operations with MySQL and provides an easy to use web interface to manage and administer a MySQL database.

7.1.2.2 Database Handlers

As mentioned in previous chapters the administrator entity is responsible for maintaining, editing and accessing the OVN database. Our administrator entity is written in Python. In order to extend its capabilities to support database operations, the DB-API Python module *MySQLdb* [37] was used, which provides a database application programming interface (API).

There are two classes for handling the OVN database, the *DBHandler* and the *OVNdbHandler* class. The primitive class *DBHandler* provides basic database operations, including generic methods such as connect, disconnect, issue a query and access the results of a query. The *OVNdbHandler* extends the class DBHandler with the custom methods that make up the administrator API, as described in section 5.2.1, for managing the OVN database. The administrator creates a reference to an OVNdbHandler object in order to be able to access its functions both for editing and accessing the OVN database.

7.1.2.3 OVN lookup function

This is one of the most essential functions of the administrator entity. Through this function, the administrator matches a packet based on its ten-tuple values against all possible OVN entries in the OVN database. When a packet enters the administrtor entity the administrator checks the Access Point that the packet was received from. It first queries the OVN database to get the OVN entries for which this Access Point has been enabled. If no such OVN exists the match is unsuccessful as there is no OVN entry corresponding to this Access Point. The administrator then sends an OVN_MISS message to the OVN interface entity. In the case that there is one or more OVN entries with this Access Point enabled, these entries are further checked in order to determine which of those have a description that matches the incoming packet.

The fields of the packet are checked one by one against all the fields that have been set for each OVN entry. If all fields whose values have been set for a specific OVN entry match the respective field values of the packet, a successful match has been made. The next step is to check which Access Points

have been enabled for the matching OVN. Unless a unique destination Access Point can be determined through some other way, the packet will have to be sent to all Access Points enabled for the matching OVN except for the ingress access point. The lookup function returns all these possible Access Points to the administrator which then issues path finding requests for each of these access points to the path-finder entity.

7.1.2.3 Conflict detection in OVN definitions

Since two or more OVNs can have common Access Points enabled, OVN definition conflicts might occur which will result in false handling of packets. The table below shows an example of such a conflict. OVN 1 and OVN 2 have a common Access Point enabled ([3 4]). The description of OVN 1 in terms of flows would be «all traffic coming from IP 10.0.0.1» while the description of OVN 2 would be «all traffic coming from 10.0.0.1 at port 80 with destination 10.0.0.2». It becomes obvious that the definition of the first OVN is a superset of the definition of the second OVN. In this way packets that perfectly match definition of OVN 2, can also match the definition of OVN 1.

OVN ID	Connected Ports	Ether Src	Ether Dst	Ether Type	Vlan ID	IP Src	IP Dst	IP Proto	Src Port	Dst Port
1	[2,4] [3,4]	-	-	-	-	10.0.0.1	Any	-	-	-
2	[3,4]	-	-	-	-	10.0.0.1	10.0.0.2	-	80	-

Figure 15: OVN definition conflict example

OVN definition conflicts should be avoided or else there will be no way for the administrator entity to determine which packet streams belong to which OVN. Therefore a conflict detection function should run everytime a new value is to be inserted or an Access Point is to be enabled for a specific OVN.

The conflict detection function allows the insertion of new values at the OVN entry only if they don't cause any definition conflicts. Therefore everytime an addProperty() or addPort() function is called, a conflict detection function checks if the value to be inserted causes possible definition conflicts and allows or bans the insertion of the value.

7.1.3 Path Finder

The path finding functionality was intentionally seperated from the rest of the entities in order for its implementation to be flexible for any given network. This way the respective path finding algorithm can encompass any of the factors defined in pragraph 5.5.2, based on the network requirements.

For our purpose we re-used the routing functionality provided by the NOX controller, in a python application called *samplerouting*. This provided routing application depends on a couple of other NOX applications, namely the topology and the discovery application. These dependencies provide the global network view for the routing application. Thus, when the controller runs with this application, the OF switches adapt the behaviour of routers. The provided application takes care of calculating paths from a source access point to a destination access point, based on the minimum number of hops, and then sets up the flow entries that correspond to this route on the involved switches. For our proof of concept we wanted to strip down this functionality, and simply have the application calculate paths. The paths are then returned to the administrator, and it is the administrator's responsibility to setup the flows on the corresponding switches.

The edited version of the *samplerouting* application instantiates a *PathfinderPAchannel* upon initiation, which takes care of the communication with the administrator. The communication is covered in the next paragraph. When a PA_PATH_REQUEST arrives from the administrator, the path finder creates a list of <dpid, inport, outport> triplets, and it sends it back to the administrator encapsulated in a PA_PATH_REPLY message.

7.2 Implementation of communication between Entities

Communication between the controller, the administrator and the path finder is event driven, and it is implemented using Twisted, an event-driven networking engine written in python [50]. For this purpose, two simple python modules have been defined:

7.2.1 Controller – Administrator communication

CAchannel.py describes the communication between the controller's OVN interface and the administrator. This file defines two Twisted protocols, one for each end of the CA channel. The controller end acts as the server and the administrator as the client for this connection. The class *ControllerCAprotocol* describes the callback functions that are triggered on the controller when events

occur on this channel. The controller simply instantiates an object of the *ControllerCAchannel* class, which takes care of the CA channel communication from that point on. Each time a CA message is sent out to the CA channel from the controller, the channel interface buffers the packet and assigns it a packetID, which will be used to identify replies (from the administrator) to specific CA messages. Similarly, the class *AdminCAprotocol* describes the behaviour of the administrator in case of CA channel events. The administrator process instantiates an *AdminCAchannel*, which takes takes care of the communication with the controller. Apart from buffering messages, the *CAchannel* module also takes care of the encapsulation of CA messages. This means that the end entities (administrator, controller) do not need to worry about creating CA headers as this is done by the *CAchannel* objects that they use.

7.2.2 PathFinder – Administrator communication

PAchannel.py describes the communication between the administrator and the path finder. The relationship between the administrator and the path finder regarding their communication is similar to the relationship between the administrator and the controller for the CA channel. Just like in the CA channel, the administrator acts as the client in this connection. The class *PathfinderPAprotocol* describes the callback functions that are triggered on the controller when events occur on this channel. The path finder only needs to instantiate an object of of the *PathfinderPAchannel.* Whenever a PA_PATH_REQUEST arrives from the administrator, it calculates a path and replies with a PA_PATH_REPLY message. PA message encapsulation is taken care of by the module. The class *AdminPAProtocol* describes the behaviour of the administrator in case of PA channel events. When a PA_PATH_REPLY message is received, the administrator matches it to an originating CA_PUSH_PACKET and pushes a CA_OVN_HIT to the controller, encapsulated accordingly.

Chapter 8

Testing Environment

8. Testing Environment

A significant portion of the project involved the establishment of an experimental environment that we could use in order to deploy the proposed architecture and experiment with it. While neither the timeframe nor the available resources allowed for a realistic scale deployment, it was considered vital to provide at least a proof of concept implementation over a small, controllable topology.

The establishment of the testbed was a challenging procedure in itself, mostly due the primitive and experimental stage of the involved tools and protocols at the time of our research. Both Openflow and NOX continue to be under heavy development at the time of writing of this document. Due to the volatile state of the code of these projects, it has been understandably difficult to maintain updated documentation, thus making the process of setting up a testbed even more challenging.

8.1 Choice of controller

As previously stated, NOX and OpenFlow do not necessarily need to operate together. NOX is a centralized programming environment which provides the ability to specify flow entries in the forwarding table of controlled switches. OpenFlow enabled switches are examples of controllable switches, but NOX can utilize any similar controlled protocol in order to manage a network. Similarly, OpenFlow provides an interface for controlling network devices, and defines a protocol that describes how devices can be manipulated by a centralized entity. However, this interface is not bound to a specific controller. Any simple programmatic entity with functionality that utilizes the switch API provided by OpenFlow can essentially act as a controller.

As a result, we were faced with three alternatives regarding the implementation of our testbed: The first option was to use a simple controller application, provided by the OpenFlow community for experimental purposes. A second alternative would be to use NOX. Finally, there was the alternative of implementing our own custom version of a controller with stripped functionality, enough to expose the features of the tested architecture.

Given that the purpose of our testbed environment was not to provide a complete, functional,

production quality implementation, but merely to demonstrate a proof-of-concept example for our proposed application, it seemed reasonable to opt for the alternative that would help reach the goal more efficiently. This ruled out the option of implementing our own controller. Although a custom version of a controller would be simple and easily manageable, the trade-off between having a simple, to-the-point controller and investing time in order to implement it seemed unfavorable.

NOX appealed as the best option for several reasons. Although its configuration would prove a little more complex than a simplified controller, and it probably provided more functionality than what we required, using a mature program obviated the need to take care of low level implementation. Moreover, the active community alleviated the problem of complexity of configuration and operation. Finally, it made sense to provide proof of concept using the most widely used controller. At the time of writing of this document, NOX is still the predominant controller for OpenFlow, still undergoing heavy development and maturing rapidly.

8.2 Choice of Path Finder

We were faced with a similar dilemma to that of the choice of controller, regarding the implementation of the path finding entity. The purpose of the path finder being a separate entity is to provide the ability to apply custom routing algorithms to the architecture. For our testing purposes, a trivial routing protocol would suffice. One alternative was to implement our own path finding entity. However, NOX provides a default routing application which works in cooperation with the global network view information held by NOX, in order to provide basic routing functionality. Thus instead of implementing a path finding entity from scratch, we decided to use a customized version of the provided NOX routing application. Some tweaking was required, in order to attach the *PAchannel* interface that would enable the communication between the path finder and the administrator, as described in chapter 7.

8.3 Test topology

Our test topology consists of two linux-based openflow switches, *ofswitch1* and *ofswitch2*, the NOX controller which runs on the same machine as *ofswitch1* and finally the administrator process which runs on the same machine as *ofswitch2*.

The two linux boxes have been configured to operate as openflow switches based on the examples provided by the NOX/Openflow community [41]. The specific instructions for our environent are documented on the project wiki [42]. Both Openflow switches are connected to the controller through a secure channel, while the controller is also connected to the administrator through our CA channel.

A laptop is connected to each openflow switch, acting as the end-host. The final test environment is depicted below:

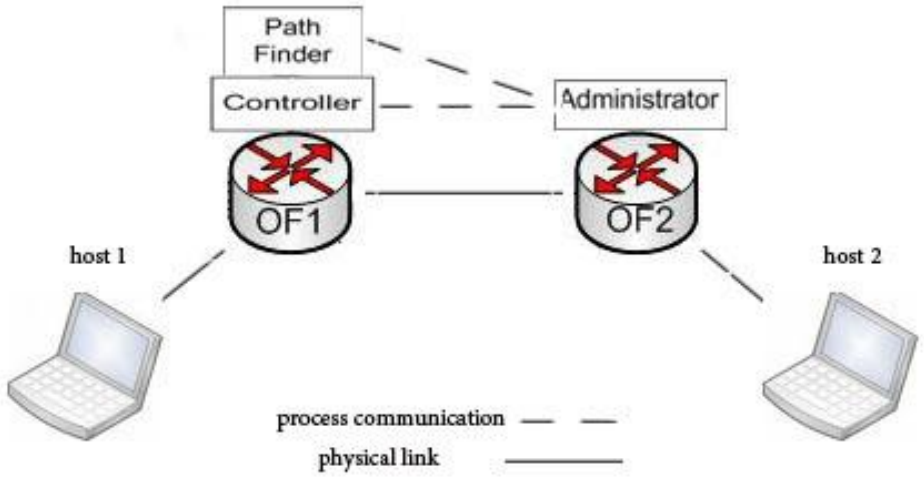


Figure 16: The testing topology. The image illustrates our 2 hosts and our 2 linux boxes. The switch icons specifically represent the openflow datapaths.

8.4 Tests

For our basic proof-of-concept experiments, we initially simply needed to check for connectivity within the defined OVNs, and isolation from other OVNs. For these purposes we started our experimentations with simple ICMP connectivity tests. The end hosts were manually configured. ICMP connectivity was tested starting with an abstract description of the while OVN, while a description of finer granularity was gradually provided.

The OVN database is initially empty. In this case a ping request from host 1 to host 2 was regarded as production network packet. Lacking underlying routing functionality, the packet was dropped after being forwarded to NOX from OF1. We then defined a basic OVN on the database, which was enabled on the 2 corresponding ports of the switches. The only detail provided at this point was that the OVN serves ICMP packets. After the creation of this abstract OVN the ICMP request was matched by the

administrator, and invoked the creation of flow entries forwarding the request to the only other port of the topology were the OVN was enabled. The request was delivered to host 2 and the reversed process was initiated in order to route the reply back to host 1. We closely monitored every small step of this process, including the bouncing of the first packet from OF to NOX, from NOX to the administrator (CA packet) where it was matched to our OVN, resulting to a request from the administrator to the path finder (PA packet), and then back to the administrator which commanded NOX to invoke the corresponding flow entries on OF1 and OF2. We confirmed that the fine-granularity entries were installed on the switched, and left to expire after a given period.

Moving a step forward towards a more strictly defined OVN, we provided the OVN database with specific information regarding the hosts. On top of ICMP traffic, we demanded that the OVN traffic originate from the specific hosts' addresses. While the exact same functionality was expected, this made the definition of our OVN much narrower. Again, the same ping request was sent by host 1 towards host 2's address. This time the administrator had to explicitly match the source/destination IP addresses before the packet was identified as OVN traffic. Following the same process, connectivity was granted and the ping went through. Finally we experimented with modifying host 2's address and other OVN properties and verified that a ping to the new address was not successful, since it was not covered by the OVN definition.

After restoring the host 2's address, a second OVN was defined, enabled on the same ports, this time serving TCP traffic. More specifically, this OVN was used for FTP traffic, and identification was done by matching the destination port. On host 1, we initiated an FTP connection to the FTP server listening on host 2. An initial problem in this scenario was that the source port for the FTP sessions is arbitrary, so, while the client-to-server flow will be matched to the OVN, the server-to-client flow cannot be matched to the OVN entry, if port restrictions have been defined. The workaround for this was to have the administrator pre-establish the reversed flow along the same path that was chosen for the client-to-server flow. Thus when the server replies to the client, the flow entries for the returning flow has been pre-established.

Similar experimentations were also made for www traffic.

8.5 Results and Conclusions

The experiments we conducted confirmed the expected functionality for the specific scenarios that we tested. While simple, these tests provided proof of concept that can be generalized to more complex topologies and OVN configurations.

As the goal was only to provide a prototype deployment of the architecture, the experiments did not involve measurements regarding performance. However, it is expected that the overall network performance is not affected by the transparent interception of packets by the administrator process. This interception is of course excepted to add some latency for every first packet of a stream, since it is not only pushed to the controller but it is in turn handled by the administrator. However this is the only performance handicap that we expect to notice. Further performance implications can be evaluated in future work.

SECTION D

Epilogue

Chapter 9 Discussion and Future Work

Chapter 9

Discussion and Future Work

9. Discussion and Future Work

This chapter provides a general discussion and conclusions of our work, and addresses matters worth investigating when moving forward with the project.

9.1 Future work

Due to the limited timeframe within which this work was completed, there are several aspects in our design that have not been investigated. However, it is essential to identify these aspects of our design that require further study and could constitute topics for future research.

9.1.1 Flow aggregation

As previously mentioned, an OVN can be viewed as a sum of flows which can be expressed as a set of flow entries in different OF switches along the physical network. This definition of an OVN implies that as the system scales and more OVNs are created on top of the physical infrastructure, the respective flow-tables of the various OF switches become larger in size preventing fast switching decisions on the data plane. Additionally maintaining and querying the OVN database becomes a costly procedure as the OVN database increases in size to keep track of the mappings between flow entries and OVNs. A flow aggregation mechanism should exist, detecting these flow entries that can be grouped together and expressed by one, granular flow entry and committing the changes to the OVN database and the respective switches' flow tables. However flow aggregation is not a trivial process and should cautiously be examined because of the potential conflicts that it entails

9.1.2 Conflict Resolution

A conflict occurs if for two OVNs who have the same port enabled on any switch, a property with the same value is added. The proposed architecture introduces a simple conflict detection mechanism which ensures that no conflicting flow entries are inserted into the OVN database. However this mechanism need not be limited to conflict detection but it could be further expanded to provide some

type of conflict resolution. This mechanism will provide the administrator with a set of alternative options for expressing a flow entry without causing a conflict. The functionality of the conflict resolution mechanism can be seen in the example of a flow entry that causes conflicts on a certain OF switch. In this case, the conflict resolution mechanism will introduce the administrator with a set of alternative OF switches that the flow entry could be moved at, in order to avoid conflicting OVN definitions.

9.1.3 OVN Database Optimization

Another interesting field for further study concerns the storing of OVN entries in the OVN database. The current design keeps two tables namely the *OVNEntry* and the *Properties* table. By using a relational key between these two we can map OVN entries to their respective properties. During an OVN lookup the database is queried in order to determine which OVN a certain packet belongs to. The properties (essentially the set of flow entries) that correspond to an OVN are fetched from the database and checked against the packet's flow header. Database optimization techniques could be examined in order to accelerate the OVN database querying process and for the query to produce more easily parseable data for the administrator.

9.1.4 Security

The Openflow network architecture raises some security issues which are inherited by the proposed architecture due to its dependence on the Openflow protocol. The existence of an additional virtualization layer and the introduction of new architectural components in our architecture add some security issues of their own.

First off, security issues are raised in terms of the communication between the various architectural components. The path finder, administrator and controller entities exchange information that is critical for the uninterrupted function of the network. The use of secure channels is one way to increase security. Authentication between the entities is also possible in order for the entities they are communicating with a trusted party.

Another aspect of security regards the access rights for the administering of the OVN database.

This include the creation, deletion or modification of an existing OVN. The OVN database contains a global (or locally global) view of all the available OVNs in a network. However accessing and altering all or part of the information in the OVN database should only be allowed to a specific, restricted group of people. User groups and permissions along with authentication should be examined for accessing the OVN database.

Additionally, some OVNs might need to be hidden or accessed by a specific group of people. Optional security parameters could be added as OVN properties, specifying security features such as blocking of IP addresses, authentication for users opting-in, firewall rules and more.

9.1.5 Performance

Similarly to security issues, our Openflow-based architecture inherits all performance issues that the Openflow network faces. The network logic in an Openflow network is gathered on a central entity, the controller. The flow requests are sent from the network switches to the controller which, in its turn, commands the flow creation on the respective switches' flow tables. This process of moving the data plane on a centralized entity raises reasonable questions for the network's performance and reliability since this type of architecture implies a single point of failure. The Openflow research team claims that the controller can handle 10000 flow requests per second. They additionally make the reasonable arrogation that computational power nowadays has reached a level that allows for complicated and time-consuming processes like packet lookup and forwarding to become less distributed between the network elements, thus gathered and executed by one or more entities on behalf of a group of network elements. However, these assumptions vary depending on the type and size of the network and the intensity of traffic.

Our architecture inserts additional entities in the Openflow network. The flow requests now are further forwarded to the administrator which has to perform a packet lookup in order to determine which OVN the packet belongs to. This additional, centralized task increases performance considerations. On top of that, the path request and its calculation by the path-finder entity create an additional burden on the overall performance. In a real-scale network facilitating hundreds of research experiments and handling production traffic, having one entity responsible for handling the OVN lookup, flow creation and path calculation and establishment might seem impossible.

One possible approach would be to try out different topologies depending on the size and needs of the network. Instead of completely centralize the network, a locally centralized approach could be followed. For example we could have one admin/controller per set of switches. Another way to look at the performance and reliability problem of this centralized architecture is to think of the controller/administrator as a regular server on the Internet, burdened with the task of serving millions of requests per minute. Server reliability and scalability techniques such as clustering, which are widely used and deployed in large data centers can thus be applied to address the single point of failure and performance problem that the controller/administrator entity introduces.

The issues of performance and scalability remain very critical enablers for the adoption of these type of centralized network architectures and ought to be thoroughly examined and further discussed since tackling these problems will deeply affect centralized architectures in general.

9.1.6 QoS

Due to the ongoing development of Openflow, the possible extensions to this work are proliferating. At the time of writing of this document there are active discussions in the community pending proposals regarding support for QoS mechanisms. For example, it will be possible for the controller to manage queues on the Openflow switch, and define queue characteristics such as minimum rate. Then the controller will be able to define, within the action of a flow entry, the queue that a flow will join. Once supported, these extensions will allow the deployment of different QoS schemes on each OVN. This would be extremely useful in scenarios where quality guarantees are required for some types of traffic, aggregated in an OVN, and would allow prioritization of OVN with regards to resource reservation.

9.2 Conclusions

This work intended to examine possible ways in which the Openflow architecture can be used to implement network virtualization. The work started off by defining an Openflow-based virtual network and went further in designing and prototyping an architectural framework for providing network virtualization on an Openflow network. However, there is a much more profound contribution of this work than the proposed architecture as described in previous chapters.

First off, this work provides a brief and consistent overview of the Openflow protocol while highlighting the interesting and unique features of this technology. Understanding Openflow is the first step towards realizing the potential of this technology and its possible uses that are not necessarily limited to network virtualization.

Moreover, it aims to provide insight to the various existing definitions and implementations of network virtualization, a valuable contribution, considering that the term has been widely used and put in different context, hence often causes confusion. The documentation of this overview of the currently existing approaches to network virtualization accelerates the pace at which research towards this direction is moving, providing a solid source of information for researchers who wish to engage in research activity within this field.

Proposing and designing a bulletproof architecture has not been the intention of this work. What is more important is the attempt to investigate the possibilities of Openflow as a tool for network virtualization and to highlight the potential of this technology towards this direction to whomever wishes to further deal with the subject. The architecture is designed in a component-based way hence can be easily subject to alterations, corrections or improvements. In fact, this is highly encouraged and desirable for an architecture to be viable. What is most important is that the reader gains good grasp of the process of designing an architecture and the several steps that have to be traversed, revisited and skipped in order to reach to a final design.

Finally, one of the main goals of this work was to highlight the points of the Openflow network architecture that need re-evaluation and re-thinking. There is no panacea in technology and every architecture always leaves room for improvement. Sometimes radical and complete reevaluation is needed in order to come up with innovative solutions. For such newly introduced concepts as the one described in this work it is more crucial to start investigating and putting pieces together, combining the tools available in order to come up with creative ideas rather than reach the perfect, faultless architecture. The viability and further expansion of this concept relies on consistently documented ideas that will raise the confidence needed within the research community to take up on this subject and to shed light on this promising field.

References

1. *«Openflow: Enabling Innovation in Campus Networks»*, Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, Jonathan Turner, ACM SIGCOMM Computer Communication Review, v.38 n.2, April 2008.

2. *«Storage Virtualization:Technologies for simplifying data storage and management»*, Tom Clark published by Addison-Wesley Professional, Mar 14, 2005, Edition:1st, ISBN-10: 0-321-2251-4.

3. *«Performance Evaluation of Virtualization Technologies for Server Consolidation»,* Pradeep Padala, Kang G. Shin, University of Michigan, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, Hewlett Packard Laboratories, 30 September 2008.

4. *«VMware's Virtual Platform: A virtual machine monitor for commodity Pcs»*, M. Rosenblum. In Hot Chips 11: Stanford University, Stanford, CA, August 15-17, 1999.

5. QEMU, http://www.nongnu.org/qemu/

6. Microsoft Virtual Server, http://www.microsoft.com/windowsserversystem/virtualserver/

7. Xen Hypervisor, http://www.xen.org/

8. User-Mode Linux, http://user-mode-linux.sourceforge.net/

9. OpenVZ, http://wiki.openvz.org/Main_Page

10. Linux-Vserver, http://linux-vserver.org/Welcome_to_Linux-VServer.org

11. «*Overview of Routing between Virtual LANs*» , cisco whitepaper

12. *«Characterizing VLAN usage in an Operational Network »*, Prashant Garimella, Yu-Wei Sung, Nan Zhang, Sanjay Rao , Purdue University 2007.

13. *«Openflow Switch Specification»* Version 0.8.9 (Wire Protocol 0x97) , December 2, 2008

14. *«NOX: Towards an Operating System for Networks»,* Natasha Gude, Teemu Koponen, Justin Pettit , Ben Pfaff, Martín Casado, Nick McKeown , Scott Shenker .

15. *«A Blueprint for Introducing Disruptive Technology into the Internet*.» , L. Peterson, T. Anderson, D. Culler, and T. Roscoe. *(HotNets-I '02),* October 2002.

16. *«Operating System Support for Planetary-Scale Network Services.»* A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. (*NSDI '04),* May 2004

17. «*Experiences Building PlanetLab»,* Larry Peterson, Andy Bavier, Marc E. Fiuczynski, Steve Muir, Department of Computer Science, Princeton University 2006.

18. «*Overcoming the Internet Impasse through Virtualization»***,** Anderson, T., Peterson, L., Shenker, S., Turner, J., Dept. of Computer Science & Eng., Washington Univ., Seattle, WA, USA; IEEE Computer, Publication Date: April 2005, Volume: 38, Issue: 4

19. «*PlanetLab Core Specification 4.0»*, Aaron Klingaman, Mark Huang, Steve Muir, Larry Peterson, Princeton University,PDN–06–032, June 2006

20. «*PlanetLab Architecture: An Overview»,* Larry Peterson, Steve Muir, Aaron Klingama, Princeton University, Timothy Roscoe,Intel Research – Berkeley, May 2006

21. «*In VINI Veritas: Realistic and Controlled Network Experimentation.»,* Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, Jennifer Rexford. (*SIGCOMM '06*), September 2006

22. FEDERICA Project, «*DJRA1.1: Evaluation of current network control and management planes for multi-domain network infrastructure»,* Cristina Cervelló-Pastor – UPC, Robert Machado – UPC, Peter Kauffman, Monika Roesler – DFN, 31 July 2008

23. FEDERICA Project, *«DJRA2.1: Architectures for virtual infrastructures, new Internet paradigms and business models»*

24. *«Network Virtualization: State of the Art and Research Challenges»*, N.M. Mosharaf Kabir Chowdhury and Raouf Boutaba, University of Waterloo

25. «*A survey of Network Virtualization»* , N.M. Mosharaf Kabir Chowdhury and Raouf Boutaba, David R. Cheriton, School of Computer Science University of Waterloo, October 15, 2008

26. «Virtualized Routing: Design and Experimental Evaluation of a Linux-Based Virtual Router Platform», Eneas Hunguana, Master thesis, September 18, 2008.

27. «XORP - Open Source IP Router», http://www.xorp.org

28. «Click- Modular Software Router», http://read.cs.ucla.edu/click/

29. «OpenVPN: An Open Source SSL VPN Solution», http://www.openvpn.net/

30. «Detecting BGP configuration faults with static analysis», Nick Feamster, Hari Balakrishnan, in Proc. Networked Systems Design and Implementation, pp. 49-56, May 2005.

31. «Linux Universal TUN/TAP driver», http://vtun.sourceforge.net/tun/

32. «Carving Research Slices Out of Your Production Network with OpenFlow», Rob Sherwood, Michael Chan,Glen Gibb, Nikhil Handigol,Te-Yuan Huang,Peyman Kazemian,Masayoshi Kobayashi, David Underhill, Kok-Kiong Yap, Guido Appenzeller, and Nick McKeown, Deutsche Telekom Inc. R&D Lab, Stanford University, NEC System Platforms Research Labs

33. « A framework for IP based Virtual Networks (RFC 2764)», B. Gleeson, A. Lin, Nortel Networks, J. Heinanen, Telia Finland, G. Armitage, A. Malis, Lucent Technologies, February 2000

34. *«Services Requirements for Layer 2 Provider Provisioned Virtual Private Networks (RFC 4665)»,* W. Augustyn, Ed., Y. Serbest, Ed. AT&T, September 2006

35. *«Generic Requirements for Provider Provisioned Virtual Private Networks PPVPN (RFC 3809)»*, Juniper Networks, June 2004

36. *«Layer 1 VPN Basic Mode (RFC 5251)»,* D. Fedyk Ed., Nortel, Y. Rekhter Ed., Juniper Networks, D. Papadimitriou, Alcatel-Lucent, R.Rabbat, Google, L. Berger, LabN, July 2008

37. «MySQLdb- MySQL for Python», http://sourceforge.net/projects/mysql-python/

38. *«How to build an application virtualization framework»*, Amir Husain, VDI Works, July 1 2008

39. *«Facilitating Microsoft Windows Vista Migration Through Application Virtualization»,* Coby Gurr, Dell Power Solutions, February 2008 http://www.dell.com/downloads/global/power/ps1q08-20080154-LANDesk.pdf

40. *"A Path Computation Element (PCE)-Based Architecture",* RFC 4655

41. Setting up a virtual testing environment: http://noxrepo.org/manual/vm_environment.html

42. OVN project wiki: http://www.tslab.ssvl.kth.se/mediawiki/ovn/index.php/Main_Page

43. *"Forwarding and Control Element Separation (ForCES) Framework"*, I. Yang et al., RFC 3746, 2004

44. *"The SoftRouter Architecture"*, T.V. Lakshman, T. Nandagopal, R. Ramjee, K. Sabnani, Bell Laboratories, Lucent Technologies

45. *"Enhancing Dynamic Cloud-based Services using Network Virtualization"*, Fang Hao, T.V. Lakshman, Sarit Mukherjee, Haoyu Song, Bell Labs, Alcatel-Lucent

46. VLAN http://standards.ieee.org/getieee802/download/802.1Q-2005.pdf

47. GENI http://www.geni.net/

48. RFC3031, “MultiProtocol Label Switching Architecture”, E. Rosen, A. Viswanathan, R. Callon, Jan 2001

49. RFC2547, “BGP/MPLS VPNs”, E Rosen, Y. Rekthter, Cisco Systems, Inc. March 1999

50. Twisted Python http://twistedmatrix.com/

51. "Microsoft SoftGrid Application Virtualization". http://www.microsoft.com/windows/products/windowsvista/enterprise/softgrid.mspx.

52. “MPLS: Next Steps”, B. Davie, A. Farell

53. “HP StorageWorks virtualization”, White paper available at: http://h20195.www2.hp.com/v2/GetPDF.aspx/4AA0-7358ENW.pdf

54. Introduction to Storage Area Networks, Jon Tate, Fabiano Lucchese, Richard Moore, IBM Redbooks, http://www.redbooks.ibm.com/abstracts/sg245470.html?Open