Horizon Project

ANR call for proposals number ANR-08-VERS-010
FINEP settlement number 1655/08

Horizon - A New Horizon for Internet

WP2 - TASK 2.2: Definition of Virtual Interfaces

Prototype and Report

(Annex E)

Institutions

Brazil
GTA-COPPE/UFRJ
PUC-Rio
UNICAMP
Netcenter Informática ltda.

France
LIP6 Université Pierre et Marie Curie
Telecom SudParis
Devoteam
Ginkgo Networks
VirtuOR

Contents

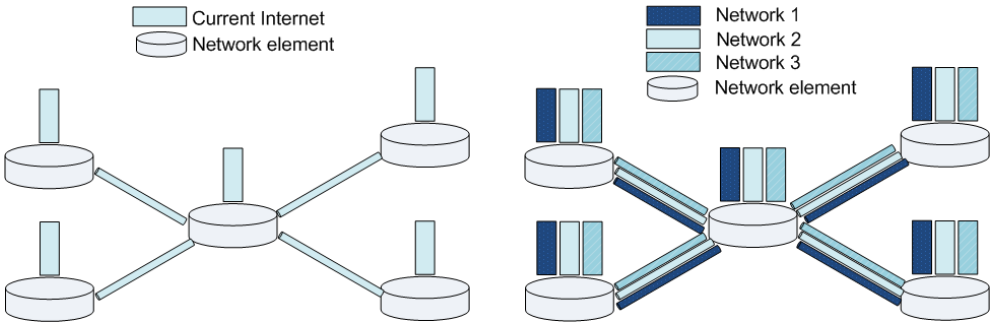
List of Figures

Chapter 1

Introduction

The Internet is a great success with more than one billion users spread over the world. Its model is based on two main pillars, the end-to-end data transfer service and the TCP/IP stack [1]. Indeed, those two pillars guarantee that the network core is simple and transparent, while all the intelligence is placed on the end systems. This architectural choice makes it easy to support new applications, because there is no need to change the network core. On the other hand, this model ossifies the Internet, making it difficult to solve structural problems like scalability, management, mobility, and security [2]. Presently, there is a rough consensus that upgrade patches are not enough to meet current and future requirements. Then, the Internet must be reformulated to provide a flexible infrastructure that supports innovation in the network, which is being called the Future Internet [2, 3, 4].

The Future Internet architecture proposals can be divided in two approaches: monist and pluralist. In the monist model, depicted in Fig. 1.1(a), the network has a monolithic architecture that must be flexible enough to provide support to the new applications. On the other hand, the pluralist approach, depicted in Fig. 1.1(b) [5], is based on the idea that the Internet must support multiple protocol stacks simultaneously. The pluralist model establishes different networks, according to the needs of network applications. Specialized networks provide specific services, such as security, mobility, or quality of service. New networks can be easily deployed to provide services required by new applications. We claim that multiple networks providing different services are simpler to implement than a unique network providing all different services at the same time. In addition to solving all of the known problems, a monist approach must be able to solve new problems which arise from novel applications. Furthermore, an important characteristic in favor of the pluralist model is that it intrinsically provides compatibility with the current Internet, which can be one of the supported protocol stacks.



(a) Current monist architecture model. Only one protocol stack running over the physical substrate.

(b) Pluralist architecture model. Different protocol stacks running, at the same time, over the same physical substrate.

Figure 1.1: Models for the monist and pluralist network architectures.

All pluralist proposals share the same idea that the virtual networks run over the same physical substrate [6, 7, 8], even though they differ in packet formats, addressing schemes, and protocols.

Based on the pluralist approach, the Horizon Project proposes the use of two different virtualization platforms, Xen and OpenFlow, to provide virtual networks. These platforms differ on how to share the physical medium between the virtual networks. These differences lead to a tradeoff between performance and flexibility. This report first presents these virtualization platforms and shows a performance analysis in order to exploit some of their characteristics.

Afterwards, this report describes all the interfaces developed in the Horizon project to provide system management. These interfaces must offer an easy way to manage resource sharing among virtual networks and, in addition, they must also facilitate actions such as reconfiguring the topology of each virtual network. In the Horizon Project, we designed and developed interfaces for both Xen na OpenFlow platforms, building a prototype for each platform as a proof of concept. Some of the developed interfaces can also be used by non-human elements, like a computer agent that controls the network. Consequently, these interfaces will be used by the Horizon Project Piloting System to control the network, i.e., the multi-agent system responsible for managing the network will use the defined interfaces to measure and act on the network.

The rest of this report is organized as follows. Chapter 2 presents Xen and Openflow virtualization platforms adopted by the Horizon Project, and provides a performance analysis of these platforms. Chapter 3 presents the

interfaces related to the Xen prototype developed by the Horizon Project team. Chapter 4 provides a similar view of the interfaces for the OpenFlow prototype. Chapter 5 presents the developed prototype of interfaces. Finally, Chapter 6 concludes this report.

Chapter 2

Virtual Networks: Isolation, Performance, and Trends

Part of this chapter was previously published on [9].

This chapter addresses the issue of sharing the network physical substrate among different virtual networks. We analyze two representative approaches for virtualizing the physical network, Xen [10] and OpenFlow [11], and discuss the use of these technologies for running virtual networks in parallel. The main objective of this chapter is to investigate the advantages and limitations of the network virtualization technologies for creating a virtual environment that could be used as the basis of a pluralist architecture for the Future Internet. To achieve this goal, we analyze Xen and OpenFlow performance as a software router and carried out experiments to evaluate these two virtualization tools using different packet forwarding schemes.

We conduct experiments to evaluate Xen and OpenFlow [12] performance acting as a virtualized software router. Based on our findings and on previous work, we conclude that there is a tradeoff between flexibility and performance that indicates that the use of shared data planes could be an important architectural choice when developing a virtual network architecture. Another key finding is that, using shared data planes, Xen and OpenFlow can multiplex several virtual networks without any measurable performance loss, comparing with a scenario where the same packet rate is handled by a single virtual network element.

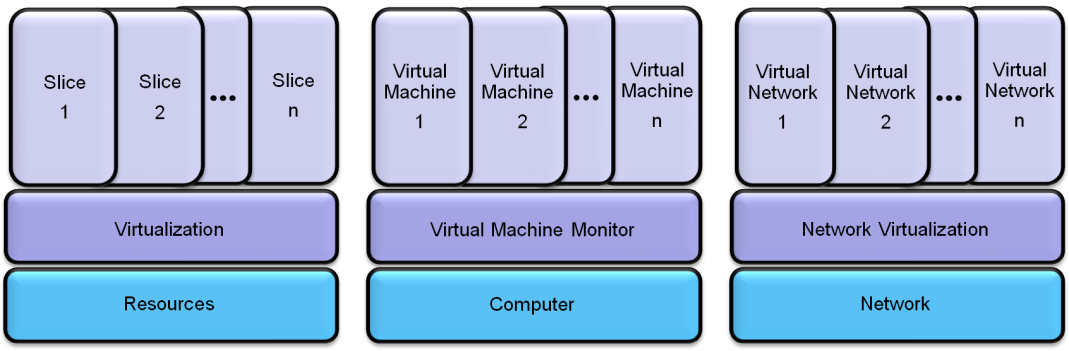
2.1 Network Virtualization

We consider virtualization as a resource abstraction that allows slicing a resource into several slices, as shown in Figure 2.1. This abstraction is often

implemented as a software layer that provides "virtual sliced interfaces" quite similar to the real resource interface. The coexistence of several virtual slices over the same resource is possible because the virtualization layer breaks the coupling between the real resource and the above layer. Fig 2.1 shows two examples of virtualization: computer virtualization and network virtualization. The computer virtualization abstraction is implemented by the so-called Virtual Machine Monitor (VMM), which provides to virtual machines (VMs) an interface (i.e., the hardware abstraction layer) quite similar to a computer hardware interface, which includes processor, memory, input/output devices, etc. Thus, each virtual machine (VM) has the impression of running directly over the physical hardware, but actually the physical hardware is shared among several VMs. We call slicing this kind of resource sharing, because the virtual machines are isolated: one VM cannot interfere with other VMs. Computer virtualization is widely used in datacenters to allow running several servers in a single physical machine. This technique saves energy and reduces maintenance costs, but flexibility is the most important virtualization feature, because each virtual machine can have its own operating system, application programs, configuration rules, and administration procedures. The flexibility of running whatever is desired into virtual slices, such as different and customized protocol stacks, is the main motivation of applying the virtualization idea to networks [5]. As shown in Fig 2.1, network virtualization is analogous to computer virtualization, but now the shared resource is the network. The concept of network virtualization is not new and it has been used in technologies such as virtual private networks (VPNs) and virtual local area networks (VLANs). Nowadays, there are new techniques that allow even the router to be virtualized. Accordingly, the slices are virtual routers, each one implementing a customized network protocol stack.

2.1.1 Network Virtualization Approaches

Approaches for realizing network virtualization differ on the level at which the virtualization layer is placed. Figure 2.2 compares two basic approaches for virtualizing a network element. Figure 2.2(a) shows the conventional network element architecture, with a single control and data plane. In a router, the control plane is responsible for running the network control software, such as routing algorithms (e.g., RIP, OSPF, and BGP) and network control protocols (e.g., ICMP), whereas the data plane is where forwarding tables and hardware data paths are implemented. To virtualize the routing procedure means that a virtualization layer is placed at some level of the network element architecture in order to allow the coexistence of multiple virtual network elements over a single physical network element. Assuming



(a) The concept of slicing resources by using virtualization.

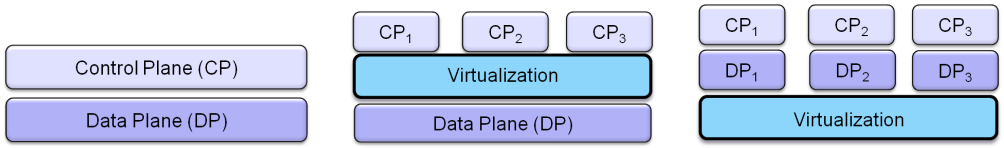
(b) Virtual slices on a computer harware.

(c) Virtual slices on a network.

Figure 2.1: Obtaining "sliced" resources.
[Obtaining "sliced" resources by means of virtualization for different shared resources.]

a virtualization layer placed between the control and data planes, then only the control plane is virtualized, as shown in Figure 2.2(b). In this case, the data plane is shared by all virtual networks and each virtual network runs its own control software. Compared to the conventional network architecture, this approach greatly improves the network programmability because now it is possible to run multiple and customized protocol stacks, instead of a single and fixed protocol stack. For instance, it is possible to program different protocol stacks for network 1, network 2 and network 3, as illustrated in the figure. In the second network virtualization approach, both control and data planes are virtualized (Figure 2.2(c)). In this case, each virtual network element implements its own data plane, besides the control plane, improving even more the network programmability. This approach allows customizing data planes at the cost of performance loss, because the data plane is no longer dedicated to a common task. This tradeoff between network programmability and performance is investigated in detail in Sections 2.3.1 and 2.3.2.

It is worth mentioning that the approach that virtualizes only the control plane, can be further divided into more subcategories depending on the isolation level in data plane sharing among virtual network elements. If a strong isolation is required, then each virtual control plane must only have access to its part of the data plane and cannot interfere with the other parts. On the other hand, if the whole data plane is completely shared among virtual control planes, then it is possible that a virtual control plane interferes with



(a) Conventional architecture. Monist model, only one network.

(b) Pluralist model virtualizing the control plane (CP).

(c) Pluralist model virtualizing the control plane (CP) and the data plane (DP).

Figure 2.2: Approaches for network virtualization differ on the level in which the virtualization layer is placed: a) no virtualization in the monist model; b) pluralist model with several virtual networks with the same data plane, but differing in control plane, and c) pluralist model with several virtual networks differing in control and data planes.

other virtual control planes. For instance, it is possible that a single virtual control plane fills the entire forwarding table with its own entries, which can lead to packet drops on the other virtual networks. The decision between strong isolation (slicing) and weak isolation (sharing) is analogous to the decision between circuit and packet switching.

2.2 Network Virtualization Technologies

In this section, we present in details two technologies that can be used to network virtualization: Xen and OpenFlow.

2.2.1 Xen

Xen is an open-source virtual machine monitor (VMM), also called hypervisor, that runs on commodity hardware platforms [10]. Xen architecture is composed of one virtual machine monitor (VMM) located above the physical hardware and several domains running simultaneously above the hypervisor, called virtual machines, as shown in Figure 2.3. Each virtual machine can have its own operating system and applications. The VMM controls the access of the multiple domains to the hardware and also manages the resources shared by these domains. Hence, virtual machines are isolated from each other, i.e., the execution of one virtual machine does not affect the performance of the others. In addition, all the device drivers are kept in an isolated driver domain, called *domain 0* (dom0), in order to provide reliable and efficient hardware support [10]. Domain 0 has special privileges compared

with the other domains, referred to as user domains (domUs), because it has total access to the hardware of the physical machine. On the other hand, user domains have virtual drivers that communicate with dom0 to access the physical hardware.

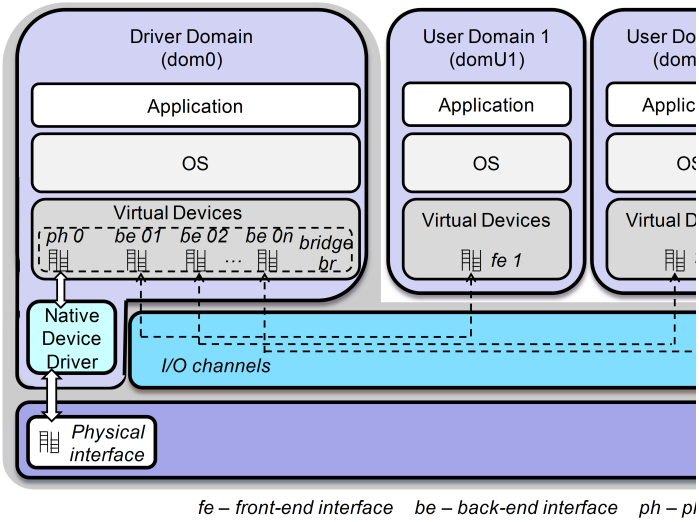
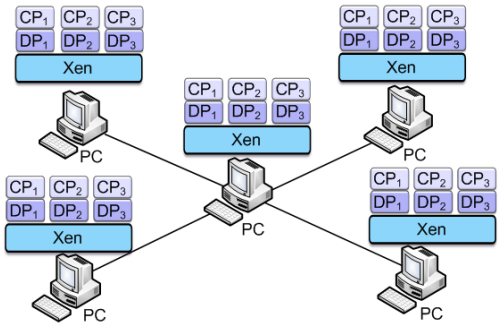


Figure 2.3: The Xen architecture.

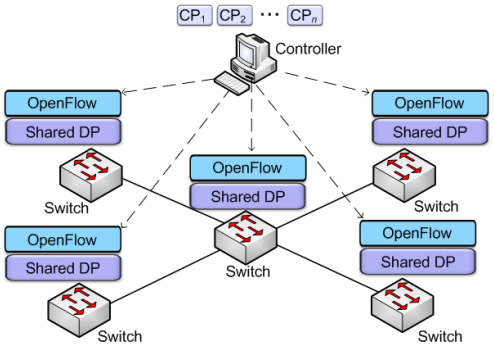
Xen virtualizes a single physical network interface by demultiplexing incoming packets from the physical interface to the user domains and, conversely, multiplexing outgoing packets generated by these user domains. This procedure, called network I/O virtualization, works as follows. Domain 0 directly access I/O devices by using its native device drivers and also performs I/O operations on behalf of domUs. On the other hand, user domains employ virtual I/O devices, controlled by virtual drivers, to request dom0 for device access [13], as illustrated in Figure 2.3. Each user domain has virtual network interfaces, called front-end interfaces, required by this domain for all its network communications. Back-end interfaces are created in domain 0 corresponding to each front-end interface in a user domain. The back-end interfaces act as the proxy for the virtual interfaces in dom0. The front-end and back-end interfaces are connected to each other through an I/O channel. In order to exchange packets between the back-end and the front-end interfaces, the I/O channel employs a zero-copy mechanism that remaps the physical page containing the packet into the target domain [13]. It is worth mentioning that as perceived by the operating systems running on the user domains, the front-end interfaces are the real ones. All the back-end inter-

faces in dom0 are connected to the physical interface and also to each other through a virtual network bridge. This is the default architecture used by Xen and it is called bridge mode. Thus, both the I/O channel and the network bridge establish a communication path between the virtual interfaces created in user domains and the physical interface.

Different virtual network elements can be implemented using Xen as it allows multiple virtual machines running simultaneously on the same hardware [10], as shown in Figure 2.4(a). In this case, each virtual machine runs a virtual router. Because the Xen virtualization layer is placed at a low level, each virtual router can have its own control and data planes.



(a) Xen: one data plane (DP) and one control plane (CP) per virtual router.



(b) OpenFlow: a shared data plane (DP) per node and all the control planes (CPs) on the Controller node.

Figure 2.4: Virtual networks with Xen and OpenFlow.

2.2.2 OpenFlow

OpenFlow [11] allows the use of the wiring closets on university campus not only for the production network, but also for experimental networks. The OpenFlow project, proposed by Standford University, aims at creating virtual environments for innovations in parallel with the production network using network elements such as switches, routers, access points, and personal computers.

OpenFlow presents a new architecture for providing virtual network environments. The key idea is the physical separation of control and data planes. Different network elements execute the packet forwarding function (data plane) and network control function (control plane). The virtualization of the forwarding elements is accomplished by a shared flow table, which represents the data plane and all control planes are centralized in a node called controller, which runs applications that control each virtual network. An example of network using OpenFlow is on Figure 2.4(b).

The OpenFlow protocol defines the communication between forwarding nodes and the network controller. It is based on the establishment of a secure channel between each forwarding node and the controller, which uses this channel to monitor and configure the forwarding nodes. Every time a new packet reaches a forwarding element and there is no previously configured flow, the first bits of the packet are forwarded to the controller, which sets a path for the packet in the chosen forwarding elements. The controller may also set the action of normal processing for a flow to be forwarded according to conventional layer-2 (L2) and layer-3 (L3) routing, as if OpenFlow did not exist. That is the reason why OpenFlow can be used in parallel to the production network without affecting production traffic.

The data plane in OpenFlow is a flow table described by header fields, counters, and actions. The header fields are a twelve-tuple structure that describes the packet header, as shown in Figure 2.5. These fields specify a flow by setting a value for each field or by using a wildcard to set only a subset of fields. The flow table also supports the use of subnet masks, if the hardware in use also supports this kind of match [14]. This twelve-tuple structure gives high flexibility for packet forwarding, because a flow can be forwarded based not only on the destination IP, as in the conventional TCP/IP network, but also on the TCP port, the MAC address, etc. Because the flows can be set based on layer-2 addresses, the forwarding elements of OpenFlow are also called OpenFlow switches. This, however, does not imply that forwarding in OpenFlow must be based on layer 2. Moreover, one of the future objectives of OpenFlow is that the header fields become user-described, which means that the packet header will not be described by fixed fields in a flow, but by

a combination of fields specified by the administrator of the virtual network. This will give OpenFlow the ability to forward packets belonging to networks with any kind of protocol stack.

After the header fields, the flow description is followed by the counters, which are used for node monitoring. Counters compute data such as the flow duration and the amount of bytes that were forwarded. The last fields in the flow description are the actions, which are a set of instructions that can be taken over each packet of a specific flow in the forwarding elements. These actions include not only forwarding a packet to a port, but also changing header fields such as VLAN data and source and destination addresses.

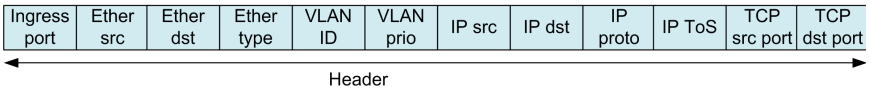


Figure 2.5: A flow entry in an OpenFlow forwarding element.

The controller node is a central element in the network, which communicates with all the nodes to configure the flow tables. The controller runs a network operating system, which provides the basic functions of network configuration to the applications that manage the virtual networks. Hence, the controller in OpenFlow works as an interface between the network applications and the forwarding elements, providing the basic functions for accessing the first packet in flows and for monitoring nodes. OpenFlow works with any controller that is compatible with the OpenFlow protocol, such as NOX [15]. In this case, each control plane is composed of a set of applications running over NOX. Hence, a virtual network in OpenFlow is defined by its control plane, which is a set of applications running over the controller, and by the flows that are being controlled by this control plane, as shown in Figure 2.6. Hence, the virtual network topology depends on the current flows in the network.

Using the single controller model, it is possible to create many virtual networks. It is important noticing, however, that different applications running over the same operating system are not isolated. As a consequence, if one application has some bug, it can stop the controller, harming all the other virtual networks. FlowVisor is a tool used with OpenFlow to allow different controllers working over the same physical network [16]. FlowVisor works as a proxy between the forwarding elements and the controllers, assuming, for instance, one controller per network. Using this model, it is possible to guarantee that failures in one virtual network will not influence the other virtual networks.

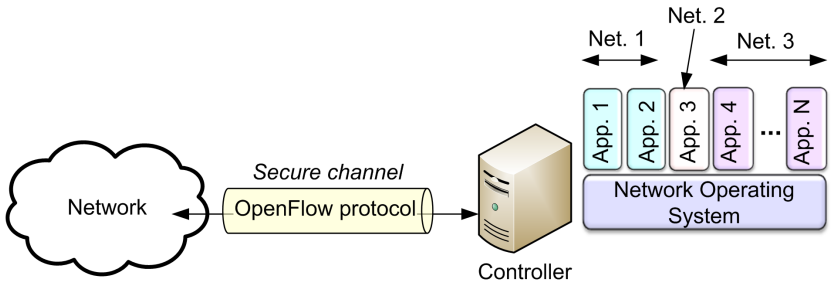


Figure 2.6: The OpenFlow controller model.

OpenFlow provides a flexible infrastructure based on the idea of distributed forwarding elements, which provide basic functions for operating a network, and centralized control planes. Using this infrastructure, it is possible to slice the physical network into multiple virtual networks. In OpenFlow, the instantiation of a network is just the creation of some set of applications in the controller. The new network flows will be created on demand, according to the packets that enter the network. OpenFlow also provides an flexible infrastructure for reallocating network resources. Re-allocating a network in OpenFlow means only to reprogram the flow table in each node that participates into the network. This is a simple operation for the controller, because it knows where the physical devices are and how they are connected.

2.3 Characteristics of Xen and OpenFlow Network Virtualization Technologies

Neither Xen nor OpenFlow were developed for supporting a pluralist architecture for Internet, but they are the best commodity alternatives for a virtual network substrate. We evaluate the main characteristics of each of these technologies, emphasizing the advantages and the disadvantages for supporting multiple networks and providing flexibility for innovations.

Xen and OpenFlow have different concepts of virtualization. Xen creates virtual networks by slicing physical network elements into different concurrent virtual routers. Consequently, a virtual network can be seen as a set of interconnected virtual routers distributed over the physical infrastructure. On the other hand, OpenFlow creates virtual networks by slicing the network control into many control planes, which create the forwarding tables in each switch. Hence, when using OpenFlow, a virtual network is a set of flows with common characteristics, which are controlled by the same set of applications

of the OpenFlow controller. The differences between Xen and OpenFlow virtualization models impact scalability, packet processing, packet forwarding, and the use of basic management tools, as we show next.

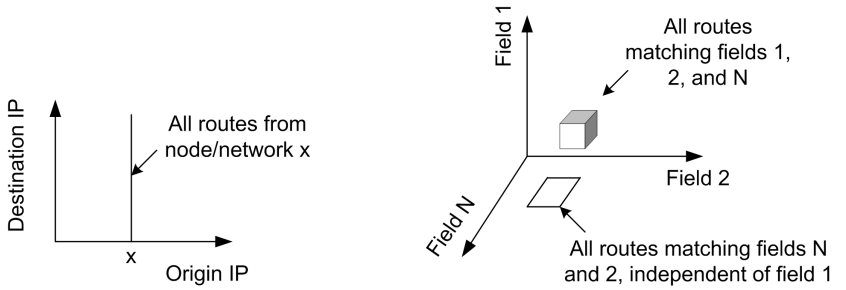
2.3.1 Programmability and Network Processing

One of the main advantages of the pluralist approach is to support innovation and, as consequence, the network must be flexible enough providing end-to-end paths over the available physical infrastructure, guaranteeing to the administrator the whole control of the network, which includes, the choice of the protocol stack, the forwarding rules, the packet processing, etc.

Because Xen virtualization layer is directly over the hardware, each virtual router has access to all computer components, such as memory, processor, and I/O devices. Therefore, the network administrator is free to choose everything that runs over Xen, the virtualization layer. Thus, different operating systems, forwarding tables, forwarding rules, and so on, can be defined for each individual virtual network. Furthermore, both data and control plane can be completely virtualized, as shown in Figure 2.2(c). Therefore, Xen actually provides a powerful and flexible platform for the network control and management, allowing hop-by-hop packet processing and forwarding. This way, virtual networks with new functionalities can be easily deployed. For instance, a virtual network with support for packet signature can be instantiated to guarantee authentication and access control. This functionality would solve security problems of the current Internet that cannot be implemented due to the network "ossification" [2]. Even disruptive network models can be implemented due to Xen flexibility for packet processing.

The OpenFlow virtualization model is different from Xen, because the virtual slice is a flow and, as a consequence, the actions concern flows, instead of packets. OpenFlow provides a simple packet forwarding scheme in which the network element looks for a packet entry on the flow table to forward the packet. If there is no entry, the packet is forwarded to the controller, so that the controller can set a forwarding rule in each node on the selected route to forward the packet. Hence, OpenFlow protocol version 1 specifies that the controller can set flow actions, which define that a header field can be modified before forwarding the packet. For instance, the forwarding element could change the destination address to forward the packet to a middle box before forwarding it to the next network element. While flow operations are easily handled by OpenFlow, packet-level features, on the other hand, such as packet signature verification, are not easily implemented in OpenFlow because such features must be executed by the controller or by a middle box, which causes a great loss in the network performance.

In terms of flexibility, the main disadvantage of the OpenFlow is that all virtual networks must base the packet forwarding on the same primitives (flow table lookup, wildcard matching, and actions), because there is a unique data plane shared by all the virtual networks in each network node. On the other hand, Xen provides independent data planes to different virtual networks. To increase the flexibility, OpenFlow provides a fine grained forwarding table, much more flexible than the current TCP/IP forwarding table, which is adopted by Xen. Currently, Xen provides a forwarding table that is based on IP routing, which means that the forwarding plane is only based on the source and destination IP addresses. In contrast, OpenFlow flow space definition is composed of N dimensions, where N is the number of fields in the header that could be used to specify a flow, as shown on Figure 2.7. Hence, we define a flow based on all the dimensions or based on a wildcard that defines which header fields are important for forwarding packets of that flow [11]. The consequence of this kind of forwarding table is that the packets can be forwarded based not only on the destination IP, but also on other parameters, such as the kind of application that is in use. This kind of forwarding table is also possible in Xen, but it is still not available.



(a) Flow space definition in the TCP/IP model.

(b) Flow space definition in the OpenFlow model.

Figure 2.7: Models of flow space to define the forwarding table in TCP/IP based networks and in OpenFlow based networks.

Another key difference between Xen and OpenFlow concerning programmability is the control plane model. In Xen, every virtual network node has both data and control plane and, consequently, the network control is decentralized. In OpenFlow, the network node has only the data plane. The control plane is centralized on the controller, which is a special node in the network. The use of a centralized control plane makes it easier to develop algorithms for network control, when compared with the use of a decentralized approach. A centralized control, however, creates the need for an extra server in the

network and also creates a single failure point in the network.

2.3.2 Performance on Forwarding

One important technology capability for providing a multiple virtual network environment for the Future Internet is a high performance on packet forwarding. The packet forwarding depends not only on the hardware being used, but also on the logic provided by each technology. In this section, we assume that both Xen and OpenFlow are running on the same hardware to evaluate which losses each technology imposes to the packet forwarding.

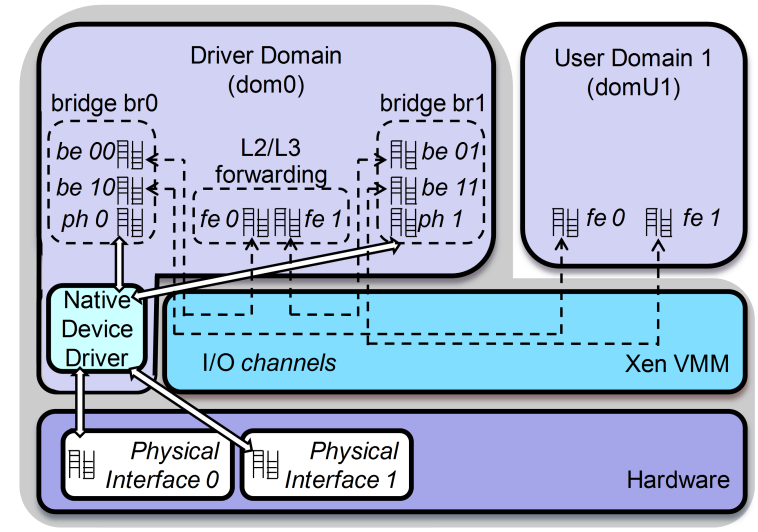
As we consider each virtual machine as a virtual router, packet forwarding becomes a key point in order to analyze the performance of Xen acting as a router. Basically, Xen performance depends on the location where packet forwarding is performed. For each virtual router, packet forwarding can be performed by the operating system running on the user domain corresponding to the virtual router or by domain 0. In the first case, the costs associated with moving packets between dom0 and domU to perform forwarding introduces control overhead and impact Xen performance. In the second case, packets for and from all virtual routers are forwarded by dom0, which deals with multiple forwarding tables simultaneously.

The performance of Xen packet forwarding also depends on the mode employed to move packets between network interfaces. Xen provides two modes to move packets: the bridge and the router modes [10]. The bridge mode is the default network architecture used by Xen, presented in Figure 2.3. Nevertheless, this architecture does not apply for a router, because we need more than one physical interface in each device. Figure 2.8(a) shows an example of the bridge mode with two physical interfaces. We have two bridges on dom0, one per physical interface, connecting the back-end interfaces and the physical ones. Packet forwarding, in this case, can be performed at dom0 by using layer-2 or layer-3 forwarding. Let p be a packet arriving at physical interface $ph0$ that must be forwarded to physical interface $ph1$. First, p is handled by the device driver running on dom0. At this time, p is in $ph0$, which is connected to bridge $br0$. This bridge demultiplexes the packet p and moves it to back-end interface $be00$ based on the MAC address of the frame destination. After that, p is moved from $be00$ to the front-end interface $fe0$ by using the I/O channel through the hypervisor. The packet p is then forwarded to the front-end interface $fe1$ and after that another I/O channel is used to move p to the back-end interface $be01$. This interface is in the same bridge $br1$ of the physical interface $ph1$. Thus, p reaches its outgoing interface. It is worth mentioning that the hypervisor is called twice to forward one packet.

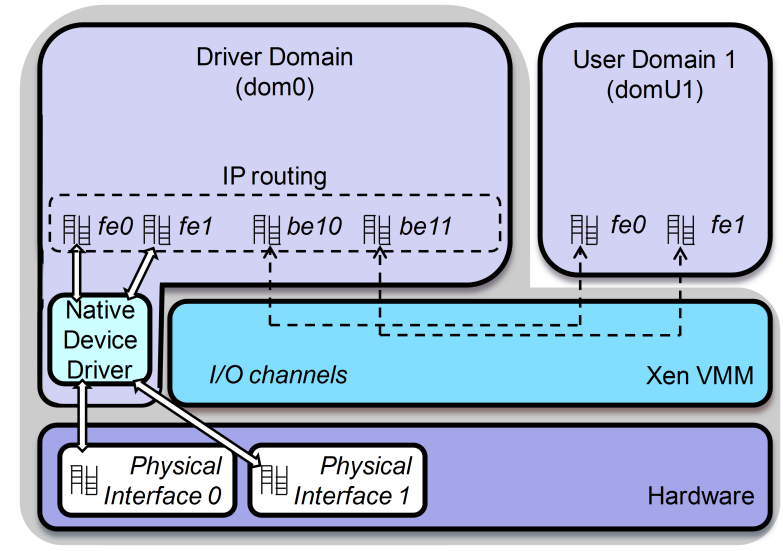
In the router mode, illustrated by Figure2.8(b), the domain 0 interfaces are the physical ones with an IP address associated to each one. As a consequence, the router mode does not require bridges connecting each physical interfaces and I/O channels, i.e., packet forwarding from a physical interface to another one at dom0 is performed as well as in native Linux. In this case, if Domain 0 is used as shared data plane (Figure 2.2(b)), there are no calls to the hypervisor. With the router mode, the hypervisor is called only when each virtual router implements its own data plane, as illustrated in Figure 2.2(c). In this case, packets are routed to the back-end interface associated to the destination domU and then are moved to the front-end interface by using the I/O channel through the hypervisor. Then, packets are moved to the back-end interface and finally routed to the outgoing physical interface. In order to allow user domains to send and receive packets, IP addresses are also assigned to back-end interfaces in contrast to the bridge mode.

The use of virtual machines in the packet forwarding gives to Xen-based virtual networks a high programmability, because each network can program and decide the actions for packet processing. This, however, implies in a low packet forwarding performance, when we compare Xen with a device without virtualization. To improve packet forwarding performance, this function can be entirely executed on Domain 0 and thus control plane runs on the virtual machine and data plane runs on Domain 0. When the packet forwarding function is accomplished on Domain 0, Xen has one control plane for each virtual router, but only one data plane shared by all the virtual networks, as illustrated in Figure 2.2(b), instead of the conventional model, shown on Figure 2.2(c). The performance of Domain 0 packet forwarding is close to the performance of the packet forwarding without the use of virtualization. The packet processing, however, looses flexibility, because Domain 0 does not process the packet hop-by-hop, as the virtual machines can do.

OpenFlow does not assume virtualized data planes on forwarding elements and, consequently, follows the model of one shared data plane for all the networks. Therefore, it is expected that OpenFlow performance on packet forwarding is just the same of the native packet forwarding. OpenFlow, however, also shows a disadvantage when the flow is not configured. As we explained before, when a packet reaches an OpenFlow switch, if the flow is not configured on the table, it is forwarded through the network to the controller. The controller, then, configures the OpenFlow switches to route the packet through the network. This mechanism introduces a greater delay when forwarding the first packet of each flow, due to the transmission and the controller processing delays. If the traffic is mostly formed of small flows, it can imply in a performance decrease in OpenFlow.



(a) The bridge mode.



(b) The router mode.

Figure 2.8: The Xen network architectures for packet forwarding.

2.3.3 Scalability

Scalability is another important issue for a virtualization technology that intends to provide multiple parallel networks for the Future Internet. Although OpenFlow natively performs better than Xen using the user domains for packet forwarding, Xen has a better support for the network scalability, assuming the number of nodes in the network as the parameter. First, OpenFlow assumes that all nodes run in the same level a layer-two protocol. This means that, if a node starts an ARP request, all nodes will listen this through a flood in the network. OpenFlow, up to now, does not provide support for creating domains in the network. As a consequence, the current OpenFlow solution is restricted to a local or metropolitan area network (layer 2) until new approaches are proposed. The Xen model is based on the idea that network nodes can run layer-three protocols. Hence, network nodes can operate as virtual routers and can establish network domains, which can be organized through a hierarchy. This structure is compatible with the current network model and is scalable.

Secondly, OpenFlow is based on a centralized controller, which configures the network elements. Since the control plane is centralized and the first packet of each flow must be forwarded and processed by the controller, the size of an OpenFlow network is restricted by the processing power and the link capacity of the controller. Up to now, OpenFlow has no native solution for providing support for different controllers for the same virtual network. Again, Xen model presents a better approach, because it is based on a decentralized control plane. Although decentralized algorithms may have a greater convergence time and a more complex logic, especially when all nodes share a network state, this kind of algorithm usually is more suitable when scalability is an issue.

Scalability is related to the number of virtual networks running over the same physical node. The new Internet requisites are still an open issue and the new architecture should not restrict the number of networks running over the available physical infrastructure. The Xen approach is less flexible in this sense, because the virtual network element is a virtual machine, which demands much more hardware resources, such as processing power and memory space, than a simple flow in an OpenFlow switch. Context switching and datapath in Xen are much more complex than in OpenFlow. The number of virtual networks in Xen is restricted by the hardware of the network element. Indeed, even if some network element has no traffic on a specific moment, it occupies a fixed amount of disc and memory in the physical network element, which may prevent the instantiation of another virtual network element on the same physical element. OpenFlow provides a more flexible infrastructure

for the instantiation of virtual network slices over a physical network element. Since the forwarding network element has only one shared data plane, its resources are not consumed by different virtual operating system or by the save of fixed amounts memory and disc for specific virtual networks. The concept of virtual networks in OpenFlow is given by a set of flows which corresponds to a specific set of characteristics that define that virtual network. For this reason, OpenFlow supports thousands of virtual networks running in parallel, while Xen is restricted to the number of virtual machines that can be multiplexed over the same hardware. It is worth mentioning that Xen scalability can be improved if Domain 0 is used as a shared data plane.

The main scalability disadvantage of Xen is that this technology is developed for personal computers and servers that do not support as many network interfaces as a router or a switch. OpenFlow can be used on commercial switches and routers, which solves the scalability issue with the number of available network interfaces per node.

2.3.4 Basic virtual network management primitives and tools

Virtual networks management primitives depends on specific tools for creating and deleting virtual networks, for virtual network reallocation over the physical infrastructure, for node resource reallocation, and for network monitoring. Since the structures of the Xen and OpenFlow models are different, as well as the definition of what is a virtual network in each model, they present different approaches for carrying out the above-mentioned primitives.

First, we must observe that managing virtual networks implies on the existence of a high hierarchical level entity that is above all the virtual networks managers. This is an important assumption, because if there was no arbiter, each network could try to consume all the available resources, damaging the other virtual networks. Indeed, a virtual network management framework also implies on the existence of isolation tools, which can be used by the arbiter to guarantee the minimum resources for each network. The idea of an arbiter which decides how to divide resources among parallel networks was previously presented by Feamster *et al.* [7]. They argue that Internet Service Providers (ISP) should be separated from Infrastructure Providers. According to this idea, the Infrastructure Provider would be responsible for arbitration of the resources among the virtual networks.

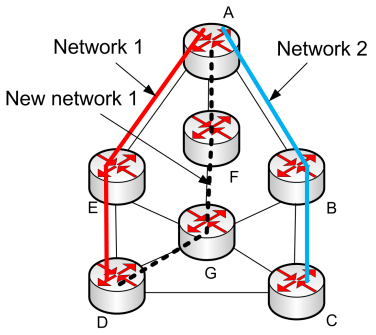
The existence of an arbiter raises an issue about security. Since we have an entity that has power over the whole network, the communication among this entity and the virtual/physical nodes must be completely secure. Moreover,

the arbiter cannot be influenced by malicious network nodes that want to divert resources from one network to other. In OpenFlow, the arbiter is naturally defined as the network controller. If the architecture is assumed to use a different controller for different networks, then this arbiter is given by the FlowVisor [16]. The secure channel among each network node and the controller/FlowVisor is defined in OpenFlow standard. The access control and the trust issues are not treated in the standard and must be implemented. Xen does not provide any kind of arbiter to manage virtual networks, because it was not developed for creating virtual networks.

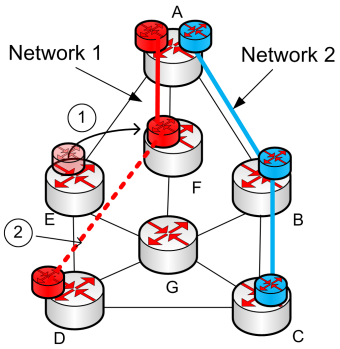
Assuming the existence of an arbiter entity, we can discuss how to implement the virtual networks basic operations for management in Xen and OpenFlow. In Xen model, we define a network as a set of virtual routers. Hence, to create or delete a network means to create or delete virtual routers over the same physical infrastructure. Xen provides mechanisms for locally instantiating virtual machines, assuming that the virtual machine image is already in the physical node memory. Hence, for instantiating a virtual network, the arbiter must first select the virtual infrastructure that will be used, transfer the virtual node image to each physical node, and then start the virtual machines. In OpenFlow, the instantiation of a network does not imply in changes on the forwarding node. Indeed, to create a new network is just to instantiate some set of applications in the controller or, in case of using FlowVisor, instantiate a new controller. The new network flows will be created on demand, according to the packets that reach the network. The selection of which physical resources to use for each network will be decided on the fly by the controller, or in the moment of network instantiation, in case of using FlowVisor. Nor Xen neither OpenFlow provides algorithms for selecting the best configuration of the virtual networks over a given physical infrastructure.

Another important operation is to re-allocate virtual networks on the fly [17, 18], which means, to re-allocate the virtual networks if a new virtual network is instantiated or if the traffic patterns of the virtual networks changed. To re-allocate a network in Xen, we can migrate, instantiate, and delete routers in the network. If we want to maintain the same virtual topology, which may be an important characteristic to not influence on virtual network functions, only migration can be used. This means that the virtual routers will be transferred among physical routers that can form the same virtual topology. Another restriction is that the new physical router needs to have at least the same number of network interfaces of the original physical router. This can imply in the construction of tunnels, to simulate a one-hop-neighborhood which does not exist in the physical infrastructure. Moreover, it can imply in the packet losses, unless some specific mechanisms are in

use [19]. Hence, instantiating and reallocating networks in Xen are challenging operations. OpenFlow, however, provides an easier infrastructure for reallocating network resources. Re-allocating a network in OpenFlow means only to reprogram the flow table in each node that participates into the network. This is a simple operation for the controller, because it knows where the physical devices are and how they are connected. Hence, to re-allocate networks in OpenFlow is easier not only because the network elements are not changed, but also because this kind of operation is much easily deployed when the control plane is centralized, which means that one server knows the whole topology and can act over all nodes. One example of network re-allocation in Xen and OpenFlow is shown on Figure 2.9.

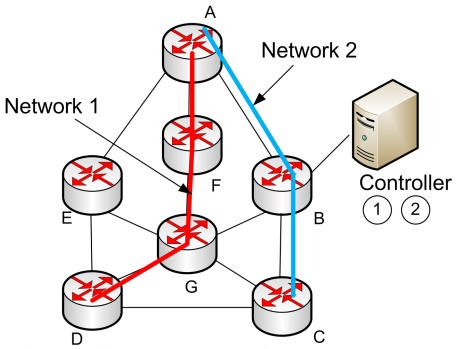


(a) Original virtual networks configuration.



1) Migrate virtual router from E to F
2) Create a tunnel from F to D

(b) Using Xen to migrate network 1.



1) Add a flow in nodes D, G, F and A.
2) Delete the old flows of nodes A, E, D.

(c) Using OpenFlow to migrate network 1.

Figure 2.9: Example of network re-allocation using Xen and OpenFlow.

Xen is based on the virtualization of the physical node. This means that the physical parameters of the node, such as memory, disc, I/O access and

CPU use must be shared among the virtual node. Indeed, Xen provides tools to manage this resource sharing, which can be used to distribute the resources with justice, privileging some networks or not. OpenFlow provides less control of the physical node, since the interface between the network node and the controller is rigid. Hence, the control of the physical resources of each network node in OpenFlow is restricted to control the frequency of monitoring messages among the node and the controller and the size of the flow table that each network is using. Also, since the controller/FlowVisor can measure the throughput of each flow, it is also possible to drop flows of specific network to control network bandwidth. These controls, however, do not provide the same precision in the control of each virtual network resource as in Xen.

2.4 Performance Evaluation

We evaluate the performance of Xen and OpenFlow in a testbed composed of three machines, as shown in Fig 2.10. The Traffic Generator machine (TG) sends packets to the Traffic Receiver machine (TR), through the Traffic Forwarder machine (TF), which simulates a virtual network element. The Traffic Forwarder machine (TF) is an HP Proliant DL380 G5 server equipped with two Intel Xeon E5440 2.83GHz processors and 10GB of RAM. Each processor has 4 cores, therefore TF machine can run 8 logical CPUs. When not mentioned, Traffic Forwarder machine, hereafter called Forwarder, is set up with 1 logical CPU. TF machine uses the two network interfaces of a PCI-Express x4 Intel Gigabit ET Dual Port Server Adapter. The Traffic Generator and Traffic Receiver, hereafter called Generator and Receiver respectively, are both general-purpose machines equipped with an Intel DP55KG motherboard and an Intel Core I7 860 2.80GHz processor. Traffic Generator (TG) and Traffic Receiver (TR) are directly connected to the Traffic Forwarder (TF) via their on-board Intel PRO/1000 PCI-Express network interface.

In the following experiments, we test packet forwarding using Native Linux, Xen, and OpenFlow.

In Native Linux experiments, the Forwarder runs a Debian Linux kernel version 2.6.26. This kernel is also used in OpenFlow experiments with an additional kernel module to enable OpenFlow. In Xen experiments, Domain 0 and User Domains run a Debian Linux system with a paravirtualized kernel version 2.6.26. For traffic generation, we use the Linux Kernel Packet Generator [20], which works as a kernel module and can generate packets at high rates. In the following, we explain the packet forwarding solutions evaluated

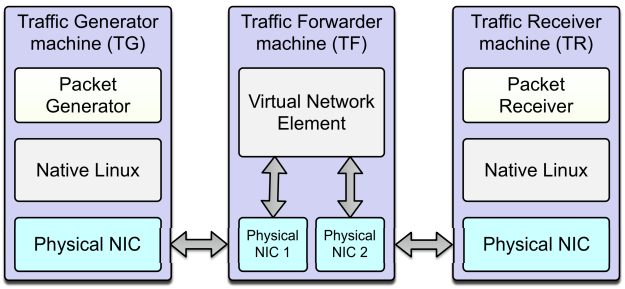


Figure 2.10: Testbed used in the evaluation. The Traffic Forwarder machine (TF) is set as Xen, OpenFlow, or Native Linux, according to each experiment.

in our experiments.

2.4.1 Xen, OpenFlow, and Native Linux Scenarios

In the Xen scenario, we test three different network configurations. In the two first ones, Xen works in the bridge mode, explained in Section 2.3.2. In the first configuration, called Xen-VM, virtual machines work as complete virtual routers, which means that both data and control plane are on the virtual machine. In the second configuration, called Xen-Bridge, we assume that virtual machines contain only the control plane. The data plane, running in Domain 0, is shared by all virtual routers. The Xen-Bridge configuration is expected to give a higher performance on packet forwarding, but it reduces the flexibility on packet processing when compared with the Xen-VM configuration. Finally, in the third configuration, Xen works in the router mode. In this case, we evaluate only the packet forwarding through Domain 0 and we call this configuration Xen-Router. We use the Xen hypervisor version 3.4.2 for all configurations.

In the OpenFlow scenario the Traffic Forwarder (TF) acts as an OpenFlow Switch. An OpenFlow Controller is connected to TF, using a third network interface. TF runs OpenFlow Reference System version 0.8.9. The controller is an IBM T42 Laptop that runs a Debian Linux system. We choose NOX version 0.6.0 [15] as the network controller. We use the `pyswitch` application, which is available in NOX to create flow rules in the OpenFlow switch.

In the Native Linux scenario, we test three different packet forwarding configurations. In the first one, Native-Router, the Forwarder (TF) works as a router. For this test we use the standard Linux kernel routing mechanism with static routes. The Native-Bridge configuration uses the Linux

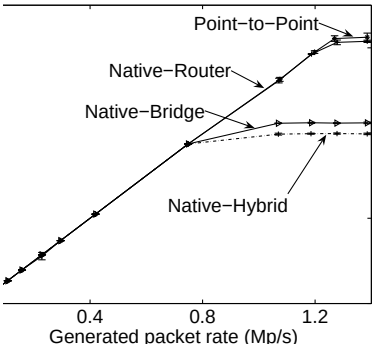
kernel bridge, which implements a software-based switch on the PC. Since we compare layer-2 and layer-3 solutions with OpenFlow and Xen, we need to compare their performance with both bridge and router modes of native Linux to evaluate the impact of virtualization on packet forwarding. Xen in the bridge mode, however, has a different configuration from the native Linux with bridge. This is because Linux bridge does L2 forwarding between two physical interfaces and Xen goes up to L3 forwarding. To perform a fair comparison between Xen in bridge mode and native Linux, we create an hybrid mode (bridge and router) for native Linux, which we call Native-Hybrid. In this hybrid mode, the Forwarder (TF) physical network interfaces are connected to different software bridges and kernel routing mechanism forwards packet between the two bridges. This configuration simulates in native Linux what is done on Xen bridge mode, illustrated in Figure 2.8(a).

2.4.2 Experimental Results

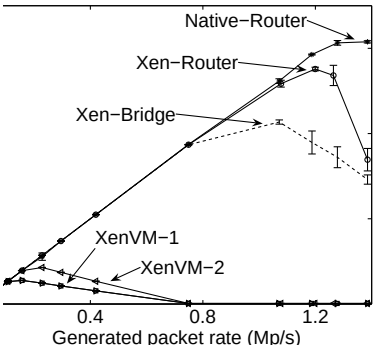
Our first experiments measure the forwarding rate achieved by the different packet forwarding solutions. The packet forwarding rate analysis is accomplished with minimum (64 bytes) and large (1512 bytes) frames. We use 64-byte frames to generate high packet rates and force high packet processing in the Forwarder (TF) and 1512-byte frames to saturate the 1 Gb/s physical link.

Figure 2.11(a) shows the forwarding rate obtained with Native Linux, which gives an upper bound for Xen and OpenFlow performances. We also plot the Point-To-Point packet rate, which is achieved when TG and TR are directly connected. Any rate achieved below the Point-To-Point packet rate is caused by loss between TG and TR. The results show that Native Linux in router mode performs as well as the Point-to-Point scenario. This is explained by the low complexity on kernel routing mechanism. In the bridge mode, however, Native Linux performs worse than in router mode. According to Mateo [12] this result may be due to the Linux bridge implementation, which is not optimized to support high packet rates. Finally, we observe that Native Linux in the hybrid mode has the worst forwarding performance. This is an expected result due to the previously mentioned limitations of bridge mode and the incremental cost required to forward packets from the bridge to IP layer in the Forwarder (TF).

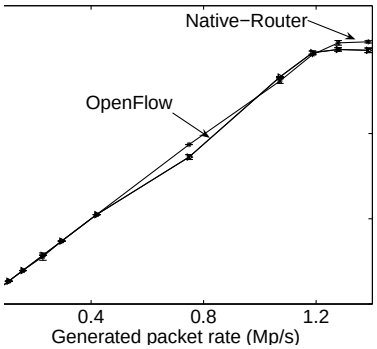
The forwarding rate results for Xen are shown in Figure 2.11(b). First, we analyze a scenario where Domain 0 forwards the packets. In this scenario no virtual machine is running, although the same results are expected when virtual machines are up and they do not forward packets [10]. In this experiment, we test the Xen bridge and router modes. Xen-Bridge uses



(a) Native Linux.



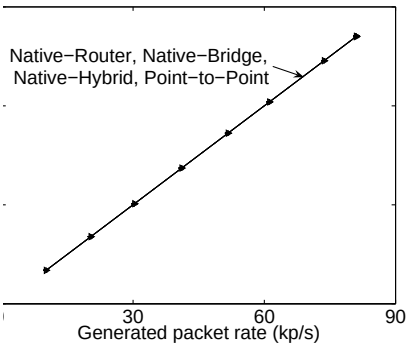
(b) Xen.



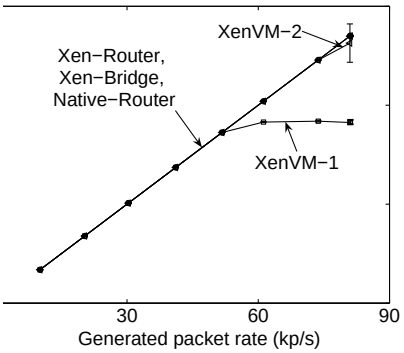
(c) OpenFlow.

Figure 2.11: Packet rate for different forwarding elements, using 64-byte frames.

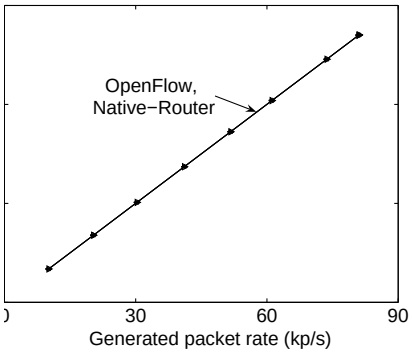
the Linux bridge to interconnect the virtual machines, as explained in Section 2.3.2. Xen-Bridge suffers the same limitations of Native Linux in bridge mode, since the bridge implementation is the same. In addition, Xen-Bridge forwards packets from the bridge to IP layer, as in hybrid mode, combined with hypervisor calls necessary in this mode. As expected, Xen-Bridge performs worse than all Native Linux forwarding schemes. On the other hand, Xen-Router performs better than Xen-Bridge, because the Linux bridge is not used and Xen hypervisor is not called when Domain 0 forwards packets. Nevertheless, Xen-Router is still worse than Native-Router. The forwarding rate rapidly decreases after about 1.2 Mp/s load. This behavior is also observed for Xen-Bridge and in the following experiments with virtual machine forwarding. This performance penalty is related to Xen interrupt handling



(a) Native Linux.



(b) Xen.



(c) OpenFlow.

Figure 2.12: Packet rate for different forwarding elements, using 1512-byte frames.

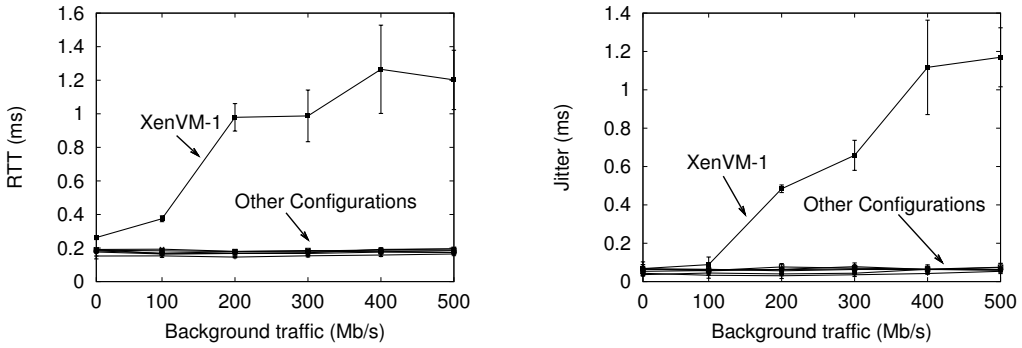
implementation and needs further investigation. Next, we analyze a scenario where a virtual machine forwards traffic using Xen bridge mode, the default Xen network configuration. In XenVM-1 configuration, both virtual machine and Domain 0 share the same CPU core. This result shows a drop in performance compared with previous results, in which Domain 0 was the forwarding element. At first glance, this poor performance could be caused by high contention for CPU resources due to the fact that a single CPU core is shared between the domains. To eliminate the contention for CPU resources we experiment with XenVM-2 configuration in Figure 2.11(b), where we give one exclusive core to each domain. The performance obtained with XenVM-2 experiment is better than with XenVM-1, but it is still lower than Domain 0 results. This can be explained due to the high complexity involving

virtual machine packet forwarding. When the traffic is forwarded through the virtual machines, it must undergo a more complex path before reaching TR. Upon packet receiving, it is transferred via DMA to Domain 0 memory. Domain 0 demultiplexes the packet to its destination, gets a free memory page associated with the receiving virtual machine, swaps the free page with the page containing the packet, and then notifies the virtual machine. To a virtual machine send a packet, it must put a transmission request along with a reference to the memory area where the packet is into Xen I/O ring. Domain 0 then polls the I/O ring and, when it receives the transmission request, it maps the reference into the physical page address, and then sends it to the network interface [21]. This increased complexity is partially responsible for the lower packet rate obtained in the two curves where virtual are used to forward packets.

Figure 2.11(c) shows that OpenFlow performs near Native Linux in router mode. In addition, the comparison between OpenFlow and XenVM results shows the tradeoff between flexibility and performance. On XenVM we have more flexibility, because the data and control planes are under total control of each virtual network administrator. In OpenFlow, however, the flexibility is lower because the data plane is shared by all virtual networks. On the other hand, due to lower processing overhead, OpenFlow performs better than XenVM in our scenario. Xen performance can be raised if the data plane is moved to Domain 0, as we can see in Xen-Router and Xen-Bridge results. In this case, however, the flexibility of customizing data planes is decreased.

We also carried out packet forwarding experiments with 1470-byte data packets, shown in Figure 2.12. With large packets, all forwarding solutions but XenVM-1 and XenVM-2 have the same behavior as in the Native-Router scenario. It means that there is no packet loss in TF and the bottleneck in this case is the 1 Gb/s link. Nevertheless, with XenVM-1, where a virtual machine shares the same core with Domain 0, the packet rate achieved is lower. In XenVM-2 experiments, where we give one exclusive CPU core for each domain, the behavior is similar to Native-Router. Thus, we conclude that, in this case, the performance decrease in XenVM-1 result is caused by high contention for CPU resources between domains and giving an exclusive CPU core to Domain 0 solves the problem.

Next, we analyze the impact of each type of virtual network element on the traffic latency. We create background traffic with different rates to be forwarded by the network element. For each of those rates, an ICMP (Internet Control Message Protocol) echo request is sent from the generator to the receiver, to evaluate the round trip time (RTT) and the jitter according to the generated background traffic. By measuring the jitter in the ICMP



(a) Round trip time (RTT) .

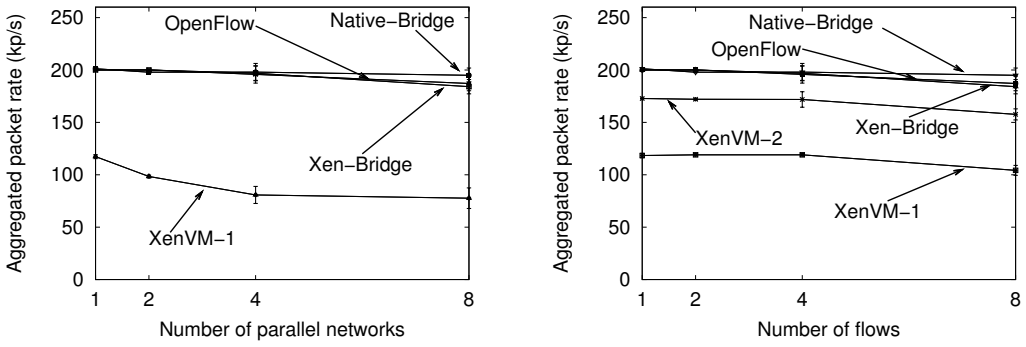
(b) Jitter.

Figure 2.13: Analysing network delays according to the network element which forwards the traffic, assuming 128-byte packets.

messages, we investigate if the network element inserts a fixed or a variable delay in the network, which could affect some real-time applications.

Figures 2.13(a) and 2.13(b) show the results for the RTT and the jitter, respectively. As the generated traffic increases, the RTT and jitter of the ICMP messages increase only for the configuration in which the traffic passes through the virtual machine, which we call XenVM-1 in the graph. The difference in the RTT between XenVM-1 and Native-Linux experiments is up to 1.5 ms in the worst scenario, with background traffic of 500 Mb/s. The RTT and the jitter of OpenFlow have the same order of magnitude as the RTT and jitter of Native-Linux. Despite of the delay difference between XenVM-1 and the other configurations, Xen virtual machines can handle network traffic without a significant impact on the latency. Because the RTT is always smaller than 1.7 ms, even in the worst case, virtual routers running over Xen do not significantly impact real-time applications such as voice over IP (VoIP), which tolerates up to 150-ms delay without disrupting the interactivity of the communication, even if one considers multiple hops [22].

We also analyze Xen and OpenFlow virtualization platform behavior for multiple networks and multiple flows per network. In this scenario, each network is represented as a flow of packets between the TG and TR for OpenFlow, and as a virtual machine for Xen. The packet size and the generated packet rate are fixed at 64 bytes and 200 kp/s, respectively. If there is more than one parallel flow, the aggregated generated traffic is still the same. For example, if the test is executed with four parallel flows, each flow



(a) Effect of the number of networks.

(b) Effect of the number of flows.

Figure 2.14: Aggregated packet rate according to the number of virtual networks.

corresponds to a packet rate of 50 kp/s, generating an aggregated rate of 200 kp/s.

Figure 2.14(a) shows the aggregated packet rate as a function of the number of virtual networks, with one flow per network. OpenFlow acts like a software switch despite the fact that the first packet of the flow must go to the OpenFlow controller. The performance obtained is very similar to a software bridge running over Native Linux, maintaining the received rate close to the generated rate of 200 kp/s. Although Xen's Domain 0 must have its interrupts first handled by the hypervisor, Xen-Bridge performs almost as well as native Linux in bridge mode. On the other hand, in the case where multiple virtual machines are simultaneously forwarding traffic (XenVM-1 configuration), the performance degrades as the number of parallel virtual machines increases. This degradation is mainly because of context switching due to the CPU scheduling, which must multiplex the processor among an increasing number of machines, each one requiring to forward its own flow.

Figure 2.14(b) shows the aggregated packet rate as a function of the number of flows, considering a single virtual network. As expected, OpenFlow and Xen-Bridge present the same behavior as in Figure 2.14(a), because both share the data plane and, consequently, there is no difference between a virtual network with multiple flows and multiple networks with one flow each. On the other hand, when the traffic is forwarded through the virtual machines (XenVM-1 configuration), the traffic must undergo a more complex path before reaching TR, as seen in previous results. In order to verify if

the complex path is the only bottleneck, the test was repeated in a configuration where the virtual machine does not share the same physical core with Domain 0, referred to as XenVM-2. In this configuration, the performance is increased by up to 50 kp/s, which indicates that the lack of processor availability is an important issue in network virtualization.

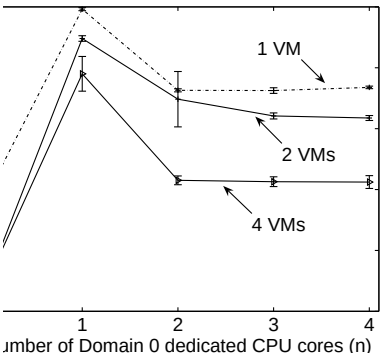


Figure 2.15: Received packet rate when varying the number of CPUs allocated to Domain 0.

To analyze the impact of CPU allocation on virtual machine forwarding, we have conducted a CPU variation test in which we send packets from TG to TR at a fixed rate of 200 kp/s through virtual machines and vary the number of dedicated CPU cores given to Domain 0. The 200 kp/s rate is used because near this rate we obtain the best performance in the 1-virtual machine scenario. According to previous results, the forwarding performance increases when both Domain 0 and virtual machine have a dedicated CPU core. This test aims to complement those results by analyzing the forwarding performance when the number of Domain 0 exclusive CPU cores increases and more virtual machines forward packets. When more than one virtual machine is used, the global sent rate of 200 kp/s is equally divided among virtual machines. Figure 2.15 shows the aggregated received rate in a scenario in which each virtual machine has one single core and the number n of CPU cores dedicated to Domain 0 is varied. According to Figure 2.15, the worst performance is obtained when all domains share the same CPU core (i.e., $n = 0$), due to a high contention for CPU resources. As expected, when $n = 1$ the performance increases, because each virtual machine has a dedicated CPU core and, consequently, has more time to execute its tasks. In addition, when Domain 0 receives more than one dedicated CPU core (i.e., $n \geq 2$), the performance is worse than when Domain 0 has a single dedicated CPU core, even when more virtual machines forward packets. These results show

that the network tasks that Domain 0 executes when each virtual router has 2 interfaces are single-threaded and these tasks are under-performing in a multi-core environment.

Chapter 3

Xen Prototype

A Xen prototype was developed on GTA laboratory in order to experiment the proposed interfaces. This chapter aims to demonstrate the interfaces related to the Xen system developed on the Horizon Project and its implementation on the GTA prototype.

The Xen prototype was developed according to the architecture described on Figure 3.1. The piloting plane requests services to the Virtual Machine Server (Section 3.1). These services can be related either to sense or to act on the infrasctructure. The Virtual Machine Server performs the action required and send the answer to the piloting plane. To ease the implementation of the piloting plane, the interfaces offered by the Virtual Machine Server must be well defined and independent of platform.

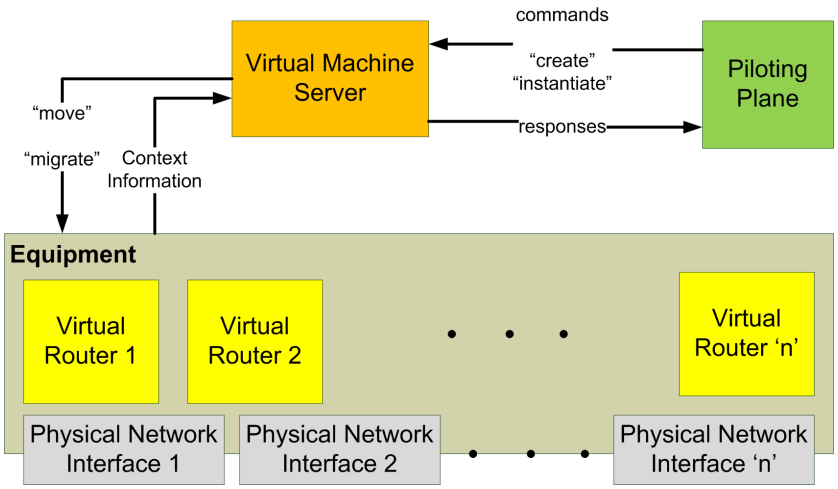


Figure 3.1: Horizon Xen Prototype Architecture.

To allow the operation of the network by an human agent, a Graphical User Interface was developed on the Xen prototype (Section 3.2). This interface takes the place of the piloting plane having access to all of the administrative tasks of the network.

3.1 Virtual Machine Server

The Virtual Machine Server [23] is a system that provide a set of services for managing virtual networks and virtual routers. This system has the capability of provide virtual routers on demand in order to match specific requirements of a protocol stack, i.e., from requests for new virtual networks creation, the server create the correct number of virtual machines and deploy them in specific nodes of the network. Moreover, this server can take part of the network administrative activities.

The server was implemented using the Web services [24] concept, and the protocol used for services requests was SOAP [25]. The web server used was the Apache Tomcat [26], and the services were developed using Java as programming language. This approach eases the creation of heterogeneous clients for the virtual machine server, besides decreases the complexity of adding new features. Each service is a public method of the class `VirtualMachineServer`.

The piloting plane proposed on the Horizon Project can decide to do some changes on the network. For example, an overloaded physical machine can have its load reduced by migrating one of its machines for another physical host. In this case, the piloting plane will send a command to Virtual Machine Server using the SOAP protocol requiring the migration of a virtual machine. The Virtual Machine Server will then use Libvirt [27], a virtualized systems management library, to perform the migration operation.

3.1.1 Services

3.1.1.1 `createVirtualMachine`

This service should be called whenever new virtual machines must be created on a node of the network.

3.1.1.2 `createVirtualNetwork`

This service creates a set of virtual machines on some physical nodes of the network. Moreover, the virtual machine server must make the mapping

between the created virtual network interface and the indicated physical network interface.

3.1.1.3 `destroyVirtualMachine`

This service destroys a virtual machine. A destroyed virtual machine cannot be reused in the future.

3.1.1.4 `getPhysicalServerStatus`

This service gets a list of basic information about the physical server. The current list contains the number of CPUS, the number of cores, the RAM memory size, the amount of free RAM memory, the name of the host, and the number and the name of the active virtual domains.

3.1.1.5 `getRegisteredNodes`

This service returns a list with the registered nodes on the virtual machine server (see service `registerNodes` to nodes registry).

3.1.1.6 `getVirtualMachineStatus`

This service returns a list of basic information about a virtual machine. The current list contains the name of the virtual machine, the current RAM memory size, the total RAM memory that can be used, the current number of VCPUs, the maximum number of VCPUs that the virtual machine can use, the CPU time used, and the current state of the virtual machine.

3.1.1.7 `migrateVirtualMachine`

This service migrates a virtual machine between two physical hosts of the network.

3.1.1.8 `registerNodes`

Initially, the virtual machine server knows nothing about the physical hosts on the network. This service can be used to register the existent nodes on the network, i.e., the name, the public key and the IP addresses of the virtual machines will be saved on the virtual machine server.

3.1.1.9 `sanityTest`

This service is a sanity test for the virtual machine server. It sends a string to the server and the server returns the same string to the client.

3.1.1.10 `shutdownVirtualMachine`

This service shut down the virtual machine. In this case, the virtual machine can be used again in the future.

3.1.1.11 `topologyDiscover`

This service creates a matrix with the adjacencies on the physical and virtual networks. There is one restriction to this service. To belong to the physical topology and have its virtual machines on the virtual topologies, a server must be registered on the server (using the service `registerNodes`).

3.1.1.12 `getVirtualMachineSchedulerParameters`

This service queries the hypervisor for the CPU scheduler parameters of a virtual machine. For the credit scheduler the parameters are weight and cap.

3.1.1.13 `setVirtualMachineSchedulerParameters`

This service sets the hypervisor for the CPU scheduler parameters of a virtual machine. For the credit scheduler the parameters are weight and cap.

3.1.2 Access to the Virtual Machine Server

In order to use the capabilities offered by the Virtual Machine Server one has to develop clients for it. A client for the Virtual Machine Server must create a SOAP message with the desired service and its parameters.

To ease the development of clients a class named `HorizonXenClient` was developed. For each service defined on the class `VirtualMachineServer` a method for the creation of message payloads is created on the `HorizonXen-Client` class. This is the API (Application Programing Interface) for the `HorizonXenClient`.

The possible message payloads are:

3.1.2.1 `public OMElement createVirtualMachinePayload(String phyServer, String vmName, String vmIP, String vmRAM);`

This method creates a payload for the request of the service `createVirtualMachine`, based on the name of the physical machine (`phyServer`) that will host the new virtual machine, on the name of this new virtual machine (`vmName`), on the desired IP address (`vmIP`), and on the desired RAM size (vmRAM).

The method returns an XML message, represented by an object of the class `OMElement`, with the operation result, either success or failure.

3.1.2.2 `public OMElement createVirtualNetworkPayload(Vector <String> phyServers, Vector<String> VMNames, Vector <String> IPs, Vector<String> RAMs, Vector<String> netInterface);`

This method creates a payload for the request of the service `createVirtualNetwork`, based on a list with the names of the physical machines (`phyServers`) that will host the new virtual machines, on a list with the names of the new virtual machines (`VMNames`), on a list with the desired IP addresses (`IPs`), on a list with the desired RAM memory sizes (`RAMs`), and on a list of the physical network interfaces (`netInterace`) that will be mapped to the new virtual network interfaces created on the virtual machines.

The method returns an XML message, represented by an object of the class `OMElement`, with the operation result, either success or failure.

3.1.2.3 `public OMElement destroyVirtualMachinePayload(String phyServer, String vmName);`

This method creates a payload for the request of the service `destroyVirtualMachine`, based on the name of the physical machine that hosts the virtual machine (`phyServer`), and on the name of the virtual machine (`vmName`).

The method returns an XML message, represented by an object of the class `OMElement`, with the operation result, either success or failure.

3.1.2.4 `public OMElement getPhysicalServerStatusPayload(String phyServer);`

This method creates a payload for the request of the service `getPhysicalServerStatus`, based on the name of the physical machine (`phyServer`).

The method returns an XML message, represented by an object of the class `OMElement`, with the operation result, either success or failure, the number of CPUS, the number of cores, the RAM memory size, the amount of free RAM memory, the name of the host, and the number and the name of the active virtual domains.

3.1.2.5 `public OMElement getRegisteredNodesPayload();`

This method creates a payload for the request of the service `getRegisteredNodes`, this method has no parameters.

The method returns an XML message, represented by an object of the class `OMElement`, with the operation result, either success or failure, and a list with the registered nodes.

3.1.2.6 `public OMElement getVirtualMachineStatusPayload(String phyServer, String vmName);`

This method creates a payload for the request of the service `getVirtualMachineStatus`, based on the name of the physical machine that hosts the virtual machine (`phyServer`), and on the name of the virtual machine (`vmName`).

The method returns an XML message, represented by an object of the class `OMElement`, with the operation result, either success or failure, the name of the virtual machine, the current RAM memory size, the total RAM memory that can be used, the current number of VCPUs, the maximum number of VCPUs that the virtual machine can use, the CPU time used, and the current state of the virtual machine.

3.1.2.7 `public OMElement migrateVirtualMachinePayload(String sourcePhyServer, String destPhyServer, String vmName, String live);`

This method creates a payload for the request of the service `migrateVirtualMachine`, based on the name of the source physical machine (`sourcePhyServer`), on the name of the destination physical machine (`destPhyServer`), on the name of the virtual machine (`vmName`), and a string indicating if the operation will be a live migration (`live`), i.e., if the migration will occur without the interruption of the programs running on the virtual machine.

The method returns an XML message, represented by an object of the class `OMElement`, with the operation result, either success or failure.

3.1.2.8 `public OMElement registerNodesPayload(Vector<PhysicalServer> phyServers);`

This method creates a payload for the request of the service `registerNodes`, based on a list with the physical servers to be registered (`phyServers`).

The method returns an XML message, represented by an object of the class `OMElement`, with the operation result, either success or failure.

3.1.2.9 `public OMElement sanityTestPayload(String testString);`

This method creates a payload for the request of the service `sanityTest`, based on a string that will send to the virtual machine server (`testString`).

The method returns an XML message, represented by an object of the class `OMElement`, with the string received by the virtual machine server, i.e., the test is considered passed if the received and the sent strings are the same.

3.1.2.10 `public OMElement shutdownVirtualMachinePayload(String phyServer, String vmName);`

This method creates a payload for the request of the service `shutdownVirtualMachine`, based on the name of the physical machine that hosts the virtual machine (`phyServer`), and on the name of the virtual machine (`vmName`).

The method returns an XML message, represented by an object of the class `OMElement`, with the operation result, either success or failure.

3.1.2.11 `public OMElement topologyDiscoverPayload();`

This method creates a payload for the request of the service `topologyDiscover`. The service has no parameters.

The method returns an XML message, represented by an object of the class `OMElement`, with the operation result, either success or failure, and the physical and virtual topologies.

3.1.2.12 `public OMElement getVirtualMachineSchedulerParametersPayload(String phyServer, String VMName);`

This method creates a payload for the request of the service `getVirtualMachineSchedulerParameters` based on the name of the physical machine that hosts the virtual machine (`phyServer`), and on the name of the virtual machine (`VMName`).

The method returns an XML message, represented by an object of the class `OMElement` with the operation result, either success or failure, and the values of the credit scheduler parameters (weight and cap).

3.1.2.13 `public OMElement setVirtualMachineSchedulerParameters-Payload(String phyServer, String VMName, String Weight, String Cap);`

This method creates a payload for the request of the service `setVirtual-MachineSchedulerParameters` based on the name of the physical machine that hosts the virtual machine (`phyServer`), the name of the virtual machine to be affected (`VMName`), the new value for the weight parameter (`Weight`), and the new value for the cap parameter (`Cap`).

The method returns an XML message, represented by an object of the class `OMElement` with the operation result, either success or failure.

3.2 Graphical User Interface

3.2.1 Introduction

The piloting system must act as a global intelligence system with the capability of understanding the knowledge shared on the network and feed the network with it in order to fix existing problems. The first step toward the development of this system is related to a framework with the capability of capturing the network information and offers it to network administrators in a understandable fashion.

3.2.2 Design choices

We believe that a graph representation is the bast way to visualize a network. In this manner, network elements, such as, physical routers must be represented as nodes while their interconnections are represented as edges. Those edges may carry information related to their availability, bandwidth, delay, and other characteristics. When we talk about network virtualization, we could extend this concept and draw graphs overlapping on the original graph of physical network, adding information about the virtual nodes that are running upon the physical network. To give the users the capability of immersion through the system, we decided to develop a tridimensional representation of networks, which allows users to navigate through the network elements and parameters.

3.2.3 Technologies adopted

The program is developed in python and, in order to provide the needed tridimensional functionalities with user immersion capabilities, we have utilized the OpenGL technologies. The graphical user interface (GUI) is produced in Qt.

3.2.4 Interface Sensors and Actuators

The Graphical User Interface (GUI) of the Xen prototype access the Virtual Machine Server, described in the section 3.1, using the command line executable client produced as a part of the Virtual Machine Server. All the services that the GUI prototype uses are accessed by this web services communication interface. The command line client receives the services and the parameters of the service that the GUI wants to request.

3.2.5 Interface functionalities

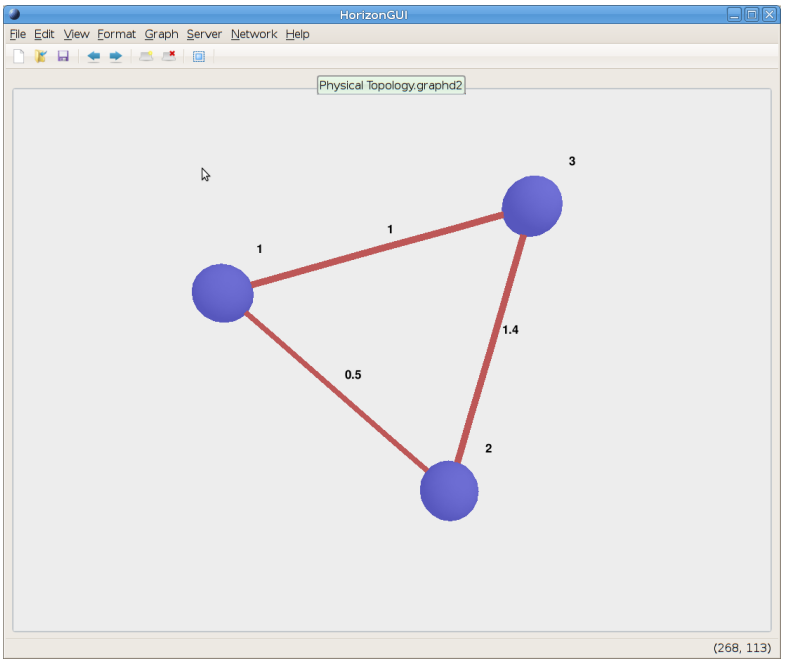


Figure 3.2: First view of the Horizon Graphic User Interface.

The prototype interface are user interactivity focused, so it has a lot of possibility to user interact with the program. As we can see in the Figure 3.2,

the focus are in the network representation. In the first GUI prototype, we represent the network nodes with spheres, but in the future, we aim to change this representation by different drawings depending of the network element type. The network visualization area is totally interactive because the user can move, rotate, and rescale the network in order to observe the way the user needs or wants.

The OpenGL draws act as a movie scene. The user has to imagine that his eyes are like cinematographic cameras, which allows it to freely move, rotate with defined focus and change the factor of the view frustum. In our prototype, all of these feelings can be executed using, respectively, the left, right, and middle mouse button.

Besides the network visualization frame, the GUI prototype has many sectors and menus that allows users to operate the network system in many different ways, as we can observe in the Figure3.3 The right proprieties panel is the network information centralizer. Information about the selected network element are shown in this panel. Another important widget of the GUI is the options panel, that helps the user interacting with the network. The first option panel implemented is the migration option panel that allows the user to migrate a virtual machine among the physical hosts. The main menu, positioned on the top of the program, reflects all the operations that could be executed in the GUI prototype and the toolbar, localized under the main menu, brings the basic and most common operations of the GUI. Another important element of the interface is the statusbar, which brings to the user information that helps him using the GUI prototype. The statusbar show the operation that has focus on that time.

3.2.5.1 Proprieties Panel

When the user selects one node or one edge in the network visualization frame, the proprieties panel (Figure 3.4) presents some information about the selected object. In this section, we present a brief description of the information shown. Out prototype user a graph representation for the network, and because of that, we represent the network elements as graph nodes and the network links as graph edges. The proprieties panel acts different depending on the selected object. In case of a node, the information is grouped as follows.

- General Information

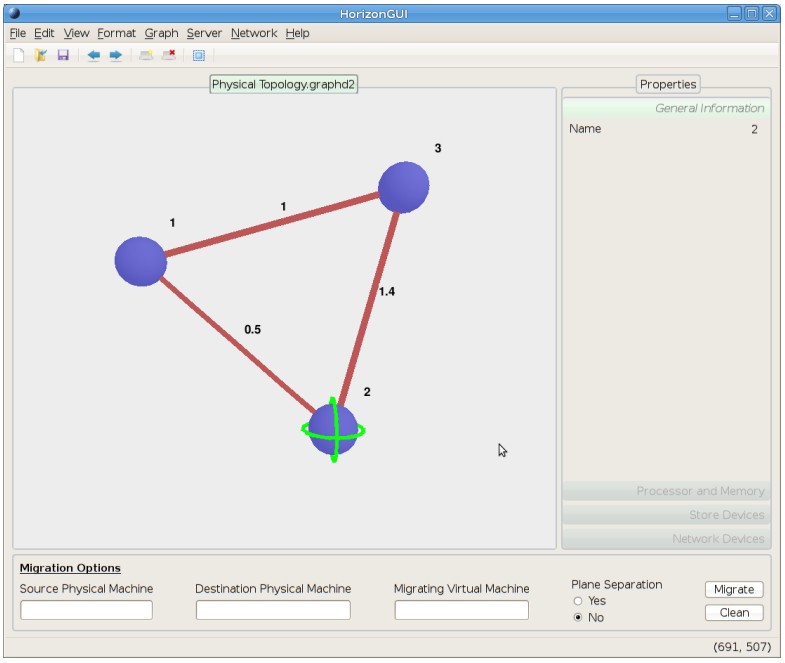


Figure 3.3: The Horizon Graphic User Interface with the proprieties panel and the migration options panel visible.

  - **Name:** The name of the network element.

- Processor and Memory
  - **CPU Usage:** The instantaneous CPU usage of the network node, presented in percentual, pointing the processing load of the node.
  - **Used Memory:** The instantaneous memory usage of the network node.
  - **Used Memory (%):** The instantaneous memory usage of the network node given in percentual of the total among of memory.
  - **Free Memory:** The instantaneous free memory available of the network node.
  - **Free Memory (%):** The instantaneous free memory available of the network node given in percentual of the total among of memory.
  - **Used Swap:** The instantaneous swap usage of the network node.
  - **Used Swap (%):** The instantaneous swap usage of the network node given in percentual of the total among of memory.

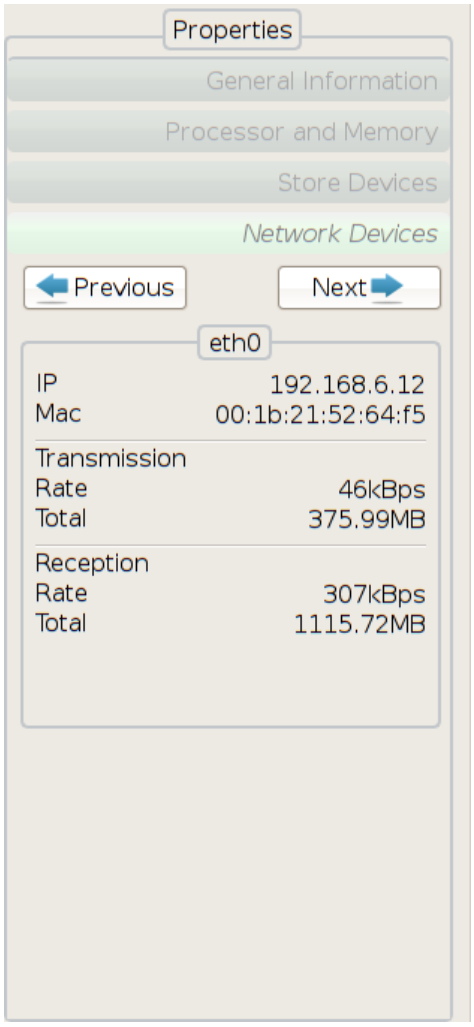


Figure 3.4: The proprieties panel. This panel is shows information about the selected element. This view represents the network devices view, and presents status of eth0 interface.

- **Free Swap:** The instantaneous free swap available of the network node.
- **Free Swap (%):** The instantaneous free swap available of the network node given in percentual of the total among of memory.
- **Virtual CPUs:** The number of virtual CPUs allocated for the network node. This parameter is used for both physical and virtual elements.

  - **Domain Count:** The number of the virtual domain in its actual physical node. This parameter is used for both physical and virtual elements.

- Store Devices
- Network Devices
  - **NetworkInterfaceName:** The name of network interface, e.g: eth0, br1, etc. The following parameters are related to this network interface. There are um panel for each network element (Figure 3.4).
  - **IPv4Addr:** The Internet Protocol, version 4, address of the network interface.
  - **LinkMAC:** The hardware address of the network interface.
  - **TransmissionRate:** The instantaneous rate that are being transmitted from the network element using this interface.
  - **TransmittedBytes:** The among of data that have been transferred from the network element since the network element starts.
  - **ReceptionRate:** The instantaneous rate that are being transmitted to the network element using this interface.
  - **ReceivedBytes:** The among of data that have been transferred to the network element since the network element starts.

If one edge is selected, the follow information is shown.

- General Information
  - **Name:** The name of the link is the concatenation of the nodes it connects.
  - **Latency:** The latency, given in milliseconds, is half of the link round trip time.
  - **Network:** The network address of the network which the link represents.
  - **Network Mask:** The network mask of the link network.

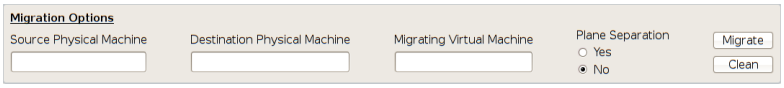


Figure 3.5: The migration options panel. This panel is responsible for virtual machine migration execution.

3.2.5.2 Migration Options Panel

The migration options panel is responsible for virtual machine migration execution. The migration services are provided by the virtual machine server and needs four parameters: source physical machine, destination physical machine, virtual machine, and type of migration 3.5. The migration process consists in the act of transmitting one virtual machine from its physical machine to another physical machine. When we talk migrate virtual machine, we are really talking about the live migrate of this virtual machine. Live migration means the virtual machine services keeps available during the migration process.

The source physical machine parameter is the physical machine the virtual machine runs before the process, and the destination physical machine is the machine it will run latter. The virtual machine is the name or id of the virtual machine in the source physical machine. This virtual machine will be live migrated sending its memory pages through the network, so the source physical machine have to have network connection with the destination physical machine.

The type of migration parameter decides whether the migration process will use the standard Xen migration tool or will use the Horizon live migration process[1]. The standard Xen migration tool has downtime in the forwarding mechanism, so it loses packets during the migration process, while the Horizon live migration process don't lose packets. Nevertheless, the standard Xen migration takes less time for complete the migration process. The Horizon live migration process are also called by plane separation process.

3.2.5.3 MenuBar

In this section, we have a brief description of each menu of the GUI prototype.

- File Menu
  - **New:**

[1]This process will be explained in the report 2.3.

  - **Open:** opens a topology file saved with the program format (keeps the position and spatial distribution of the network).
  - **Save:** saves the actual topology with the program format (keeps the position and the spatial distribution of the network).
  - **New Tab:** create a new tab where the user can load/analyze new topologies.
  - Import:
    * **Adjacency Matrix:** imports an Adjacency Matrix that represents a topology that were saved as a csv file.
  - **E**xport
    * **Adjacency Matrix:** exports an Adjacency Matrix that represents the actual topology in a csv(comma separated value) file.
  - **Close Tab:** close the active tab.
  - **Exit:** Exits the program.

- Edit Menu
  - **Undo Action:** If the topology was changed, allows the user to return to a previous topology state.
  - **Redo Action:** If the Undo Action was done, allows the user to forward to the next state.
  - **Select All:** Select all nodes in the topology.

- View Menu
  - **Fullscreen:** View the topology in full screen.
  - **Fullscreen 3D:** View the topology in 3D. Must use specific glasses and screen.
  - **All Objects**
  - **Restore Original View:** Restores the original view.
  - Reposition Nodes
    * **Random Algorithm:** Organizes the position of the topology nodes in a random fashion.
    * **Circle Algorithm:** Organizes the position of the topology nodes as a circle.

        * **Tree Algorithm:** Organizes the position of the topology nodes as a tree.
        * **Kamada Kawai Algorithm:** Organizes the position of the topology node with the advent of an entropy minimizing algorithm.
    - **Reorganize Graph Automatically:** At each topology change, recalculates the position of the nodes with the Kamada Kawai algorithm.
    - Proprieties Panel

- Format Menu
    - **Rename Node:** Rename a node in the topology.
    - **Change Weight:** Change the weight of a given link.
    - **Background Color:** Change the background color.
    - **Object Color:** Change the color of selected objects.
    - **Default Color:** Sets the default color that will be used.
    - **Object Size:** Set the size of the object.
    - **Default Size:** Set the default size of the objects

- Graph Menu
    - **Select Original Node:** Select the origin node.
    - **Select Destination Node:** Select the destination node.
    - **Clear Node Selection:** Deselect the selected origin and destination nodes.
    - Best Path
        * **Least Weight:** Calculates the best path with least weight.
        * **Least Number of Hops:** Calculates the best path with least hops.
        * **Least Product:** Calculates the best path with least product.

- Server Menu
    - **Connect:** Connects to a specific server.

- Network Menu
    - Discover Network Topology

    - **Physical Topology:** Retrieves the physical topology from the server.
    - **Logical Topology:** Retrieves the logical topology from the server

- Help Menu
  - **Content:** Explanations about the application.
  - **About:** Shows information about the developers and the program.

3.3 Integration with Prototype Sensors and Actuators

A number of applications related to Xen platform were developed inside the context of the Horizon project. These applications need to inter-operate in order to achieve the desired objective of a integrated environment for the Future Internet. This section aims to explain how these applications are integrated in the current state of the Horizon Xen testbed.

3.3.1 Virtual Machine Server and Prototype Sensors and Actuators

The Virtual Machine Server (Section 3.1) aims to offer an integrated interface to control the Xen machines of the network. Part of the administrative tasks were accomplished using the Libvirt library [27]. However, part of the tasks, for example topology discovery, can not be done only using the Libvirt library. For these tasks a set of applications were developed.

The Topology Discovery module was developed for obtaining the virtual and physical networks topologies. For acquiring CPU, memory and network related measures, we made the Measures Gatherer module. In order to obtain a virtual router migration with no packet loss, the Migration module was developed. For controlling the virtual router throughput, the Scheduler module was developed, acting on the virtual routers VCPUs CAPs. These modules have parts residing inside the virtual routers, the physical routers and the controller in order to accomplish their objectives. For communicating all the parts, we developed the Communication module, which has parts residing inside the physical router (Domain 0), the virtual routers (Domains U) and the controller. The interaction between all these modules is shown in Figure 3.6.

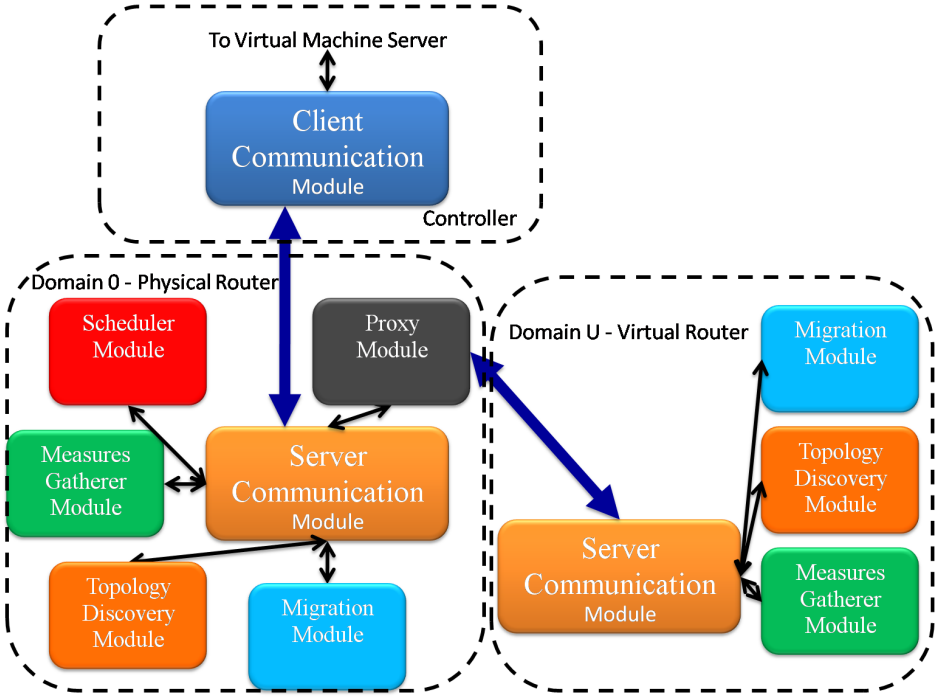


Figure 3.6: Controller, physical router and virtual routers modules interactions.

Whenever the Virtual Machine Server desires to use one of these applications, it calls the Client Communication module through command line passing the desired request. If the request is to a physical router, the physical router IP address is provided. If the request is to a virtual router, both the physical router IP address and the virtual router IP address are passed to the Client Communication module. Following, the Client Communication module builds an XML message with the request and sends it through a socket to the Server Communication module residing inside the physical router. If the request regards the physical router, the Server Communication module creates an instance of the correct application to handle the request, calls the application with the provided parameters and returns to the Client Communication module an XML containing the application response. If the request regards a virtual router, then the physical server acts as an intermediate between the controller and the virtual router. In this case, the Server Communication module desencapsulates the message sent by the Client Communication module and handles it to the Proxy module, which is responsible for sending the message to the Server Communication module of the right virtual router. When the message arrives to the Server Communication mod-

ule inside the virtual router, an instance of the appropriate application is created, it handles the request and the application response is sent back to the controller through the virtual router Server Communication module, the Proxy module and the physical router Server Communication module.

Chapter 4

OpenFlow Prototype

An OpenFlow prototype was developed on GTA laboratory to ease OpenFlow network administration [28]. This chapter aims at demonstrate the components related to the OpenFlow prototype developed on the Horizon Project.

The OpenFlow Prototype is based on a Web Service architecture. The communication between the core of the prototype and external applications uses the HTTP(Hyper Text Markup Language) protocol to exchange XML(eXtensible Markup Language) messages. These applications are similar to agents that controls the network. To measure the OpenFlow network performance we use network sensors. These sensors are basically counters installed on the switches, accessible by the OpenFlow protocol, or the OpenFlow table information, such as the number of flow entries in the tables and characteristics of those flows. NOX applications collect sensor information and make them available as a Web Service.

Figure 4.1 shows the OpenFlow Prototype architecture. The NOX Controller is the base of OpenFlow applications. The NOX Controller provides the OpenFlow Protocol and the secure channel implementation for the applications that run over it. The applications that run on NOX Controller in this prototype are described below.

- **Stats App:** Collects the statistics about the switches and convert them into a XML message.
- **Discovery App:** Discovers the network topology and describes it as a XML message.
- **SpanningTree App:** Implements a spanning tree algorithm that avoids the occurrence of loops in the network. The topology of the defined spanning tree is available as a XML message.

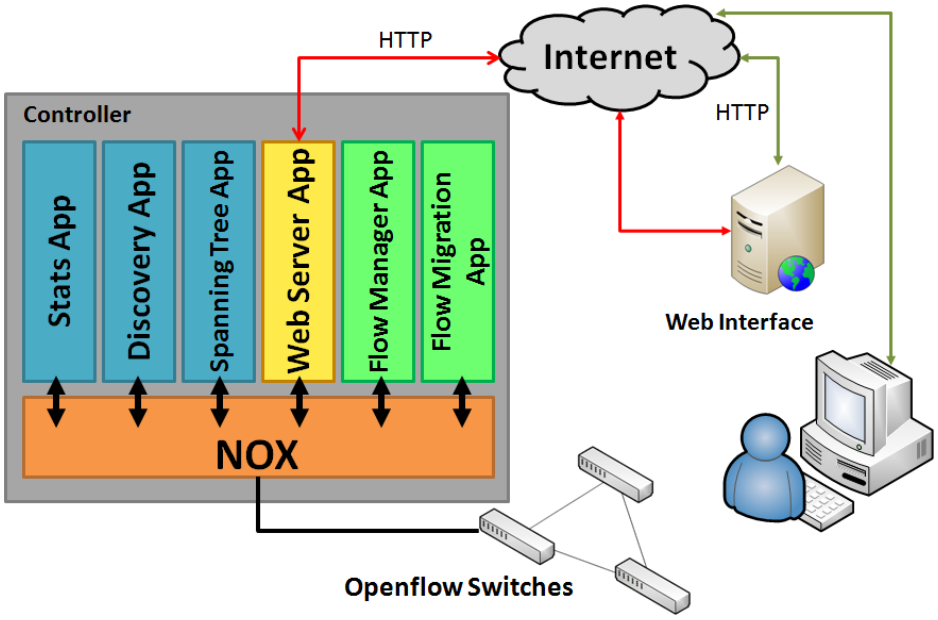


Figure 4.1: OpenFlow applications, NOX and agents interaction.

- **FlowManager App:** Implements flow changes by adding, modifying and deleting flows.
- **FlowMigration App:** Implements flow changes by migrating a flow from one path to another path.
- **WebServer App:** Provides the integration among the features of NOX Controller applications. The WebServer App implements the HTTP protocol to provide an interface between applications of NOX Controller and external applications. Consequently, this application is in charge of handling the HTTP requests, convert them in an application method call and execute the method. Our prototype implements a client for the WebServer App. The client is another web server that enables administrators to control the OpenFlow network. This client takes part in a web application that provides a Graphical User Interface, described latter in Section 4.2. More details about WebServer App implementation is provide in the next section (Section 4.1).

4.1 OpenFlow Web Server

Web Server Application (WebServer App) [29] is a NOX application that is responsible for providing a web interface for other NOX applications. The WebServer App implements the concept of web Service, in which a functionality of other applications can be accessed by an HTTP request and returns XML messages. The implementation of this application is based on the NOX web server default application. The NOX default web server application is set to run on 8080 port, listening HTTP requests. All HTTP requests are handled by the OpenFlow resource. OpenFlow resource is defined on `mywebserver` application. For each Web Service provided by OpenFlow resource, there is a method defined in the `MyWebServerResource` class. We describe below the concepts concerning each WebServer App component.

4.1.1 Default Web Server Application

The NOX default web server application implements a framework for deploying websites as a NOX application. This feature is used to implement the Web Services as a special kind of websites.

4.1.2 `mywebserver` class

It is a NOX application implementation class. It starts all applications that must run simultaneously with the web server application. It also starts the NOX default web server application and defines its default resource as an object of the class `MyWebServerResource`.

4.1.3 `MyWebServerResource` class

This class defines a resource. A resource is a kind of website that is implemented by the NOX default web server application. This class also implements the mapping of URL requests to function calls, providing an interface between the user and the OpenFlow network. There are some services already implemented on `MyWebServerResource`. Each service can be accessed by an HTTP request using an specific URL. The services are described as follows.

4.1.3.1 `getStats`

This service does not take any parameters. It calls the Stats App and returns the statistics and the information about the OpenFlow switch network

in a XML message.

4.1.3.2 `getTopology`

This service does not take any parameters. It calls the Discovery App and returns the topology of the network in a XML message. This service returns a list of all network links.

4.1.3.3 `getNeighbor`

This service does not take any parameters. It calls the Discovery App and returns the topology of the network in a XML message. This service returns the list of all node neighbors for each node in the network.

4.1.3.4 `getSpanningTree`

This service does not take any parameters. It calls the Discovery App and returns the spanning tree of the network in a XML message. This service returns the list of the node neighbors, which are linked to the node by a spanning tree link, for each node in the network.

4.1.3.5 `addFlow`

This service takes as parameters flow characteristics, like the flow match, idle timeout, hard timeout, priority, and action. This service adds a new flow calling the FlowManager App, which performs the required action over the network.

4.1.3.6 `delFlow`

This service takes as parameters flow characteristics, like the flow match, idle timeout, hard timeout, and priority. This service delete a flow calling the FlowManager App, which performs the required action over the network.

4.1.3.7 `migrateFlow`

This service takes as parameters flow characteristics, like the flow match, idle timeout, hard timeout, priority, action, and also the list of switches on which the flow must be set. This service migrates a flow calling the FlowMigration App, which performs the required action over the network.

4.2 Graphical User Interface

As seen before, OpenFlow Switches forwards network traffic according to a flow table containing every active flow. This table contains flow characteristics and rules to be applied to these flows, such as determining the queue and the output port. This table can be configured locally or by a network controller. We then developed a user-friendly interface that allows users to modify the flow tables in order to facilitate configuration and network management.

This user interface was developed based on a web application in which, using a web browser, the user can access the interface and perform commands and queries to manage the network.

The application that provides the Graphical User Interface is divided into three layers. The first one is the *Data Layer*, where are executed user commands and where is performed the collection of data given as command answers. The second one is the *Data Processing Layer*, that processes all received information before sending it to the other layers. The third one is the *Presentation Layer* that organizes and shows the data to the user. The isolation provided by layer structure allows us to modify a specific layer without modifying the others. Figure 4.2 presents the layers of the application and the protocol used to exchange messages.

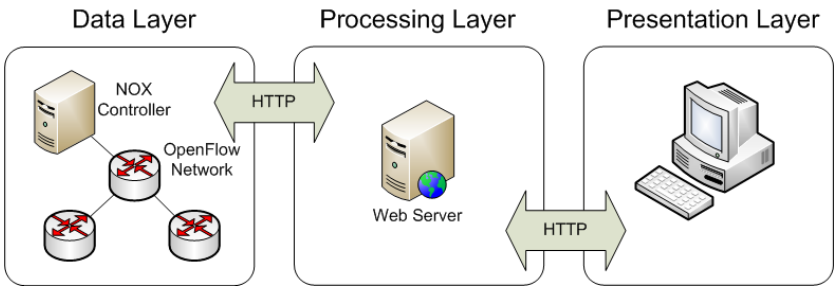


Figure 4.2: Application layers.

4.2.1 Data Layer

The Data Layer is composed by NOX controller and its applications. The application WebServerApp is responsible for providing the communication interface between the Data Layer and the Data Processing Layer. This application communicates with Data Processing Layer over the HTTP protocol.

4.2.2 Data Processing Layer

The Data Processing Layer is composed by a web server, developed in Python language, which processes the requests and commands executed by user. We choose Python because it is a flexible language, it has a good number of modules and classes that can be used and it is ease to debug. Furthermore, the NOX applications are developed using Python language. Future works involves modifying the web Server implemented in this layer in order to be compatible with an Apache Server. We choose to develop our own server, instead of using an existing server like Apache, because we want to have total control of provided services in order to facilitate the system development.

As we shall see, the Data Presentation Layer has web pages that implements an user-friendly interface for network controlling. In that layer there is a CGI (Common Gateway Interface) file for each web page, with HTML page marks and some classes to process the requests. When the user calls a web page or executes a command in Data Presentation Layer, the Data Processing Layer web server calls the appropriated class. This class processes the data, sends the commands to Data Layer, receives the commands execution results, processes these results, prepares the request answer and sends it to the Presentation Layer. The answer is prepared using the HTML marks that are in the CGI file of the called page. If the request is a command execution, then the answer is a XML message.

4.2.3 Data Presentation Layer

The Presentation Layer is composed by marks and scripts files that are interpreted by the web browser, which can be HTML(Hyper Text Markup Language), JS(JavaScript), CSS(Cascading Style Sheets), XML(eXtensible Markup Language), and SVG (Scalable Vector Graphics) files. The HTML files have the web page description that a web browser interprets in order to show this web page.

The CSS files have the style markup to improve the presentation provide by HTML. The CSS file is interpreted by web browser and the style is applied to HTML markup. The JS files have the script functions to make web pages dynamic and interactive. The XML files have information that web browser or JavaScript functions use in order to show or exchange some data. The messages that are provide as user commands answers are XML messages.

The SVG files have a XML description of network topology to provide graphic visualization. A combination between SVG and JavaScript allows us to create animations and interactions with the image generated by SVG.

Whit this combination we implement some actions to be executed into the topology view, such as flow creation and flow migration. With this feature the network management becomes simpler.

In this layer there is some web pages that allow users to acquire network information and execute commands. The provided pages are described below with some images illustrating them.

4.2.3.1 Home

System home page with a system presentation text.

4.2.3.2 Statistics

Web age that provides all statistics and information of the network switches such as: IP address, MAC address, identifier, ports statistics, flows statistics and flow tables.

In this page is also provided a form to add a flow in the flow table of a specific switch.

The statistics and information about switches are stored in the Data Layer and are periodically updated. The Data Layer is responsible to update the data. This works this way to make the queries fast to the user. If the data were obtained by demand, the answer to user would take the time needed for the Data Layer to collect all information of each network switch. Consequently, the user's waiting time will depend on the number of switches on network. We will describe each function of this page with graphical examples.

Figure 4.3 presents the Statistics page. The area indicated by the number 1 on the figure is the button to access the statistics page and the one indicated by the number 2 is the form to acquire the statistics from one or more switches.

Figure 4.4 presents the Switch Description. It presents the switch IP address, MAC address, datapath ID and also information about software, hardware, serial number and vendor of the switch.

Figure 4.5 presents the switch ports and Figure 4.6 presents the statistics of a specific switch port. It presents received and transmitted data statistics, such as received and transmitted bandwidth, bytes and packets count, port name, port status, collisions and other measures.

Figure 4.7 presents the Switch Flow Tables Statistics. It presents table information such as name, matched count, active count, max entries and lookup count.

Figure 4.8 presents the aggregated flows statistics. It presents packet count and byte count of all network flows together, and also presents the

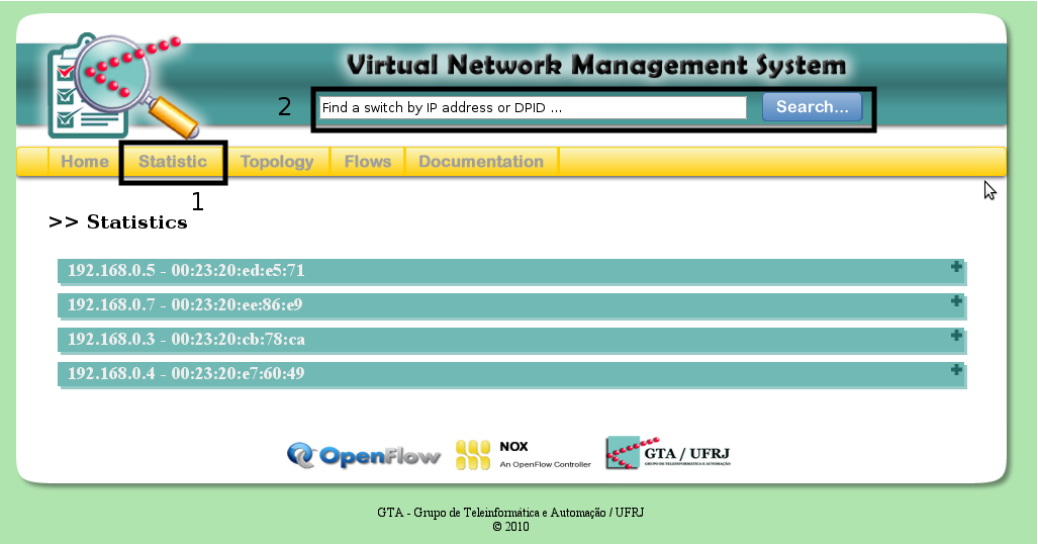


Figure 4.3: Statistics Page.

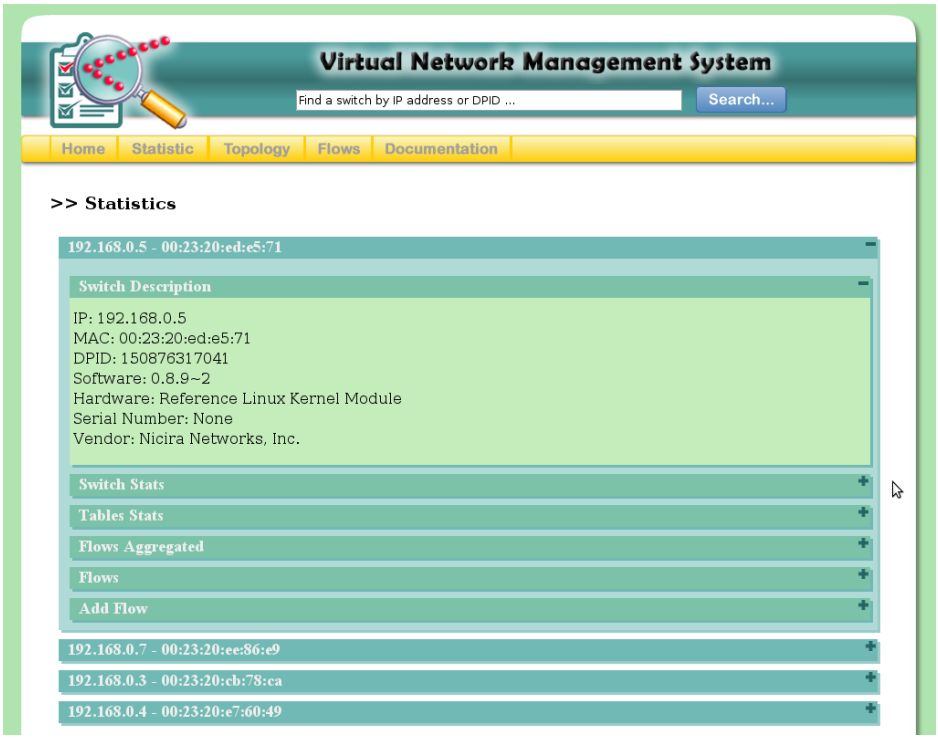


Figure 4.4: Switch Description.

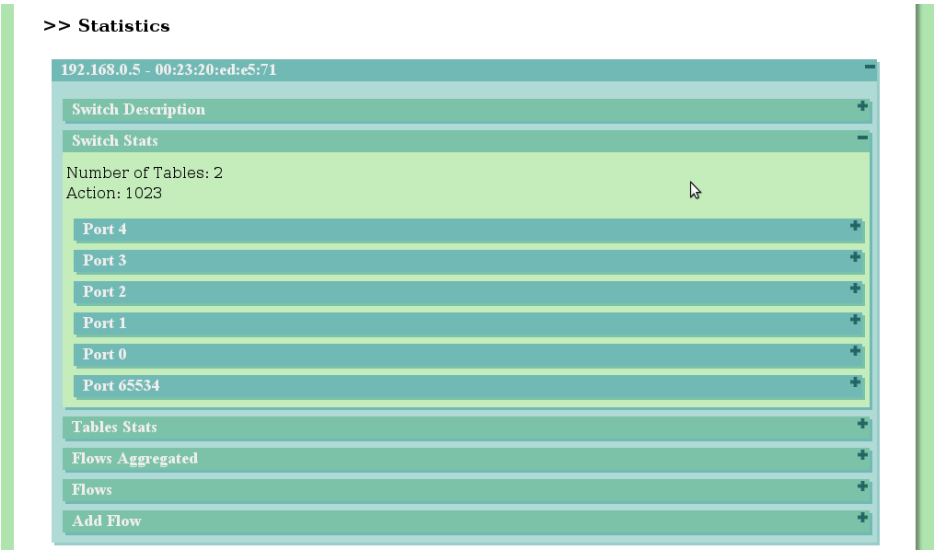


Figure 4.5: Switch Status.

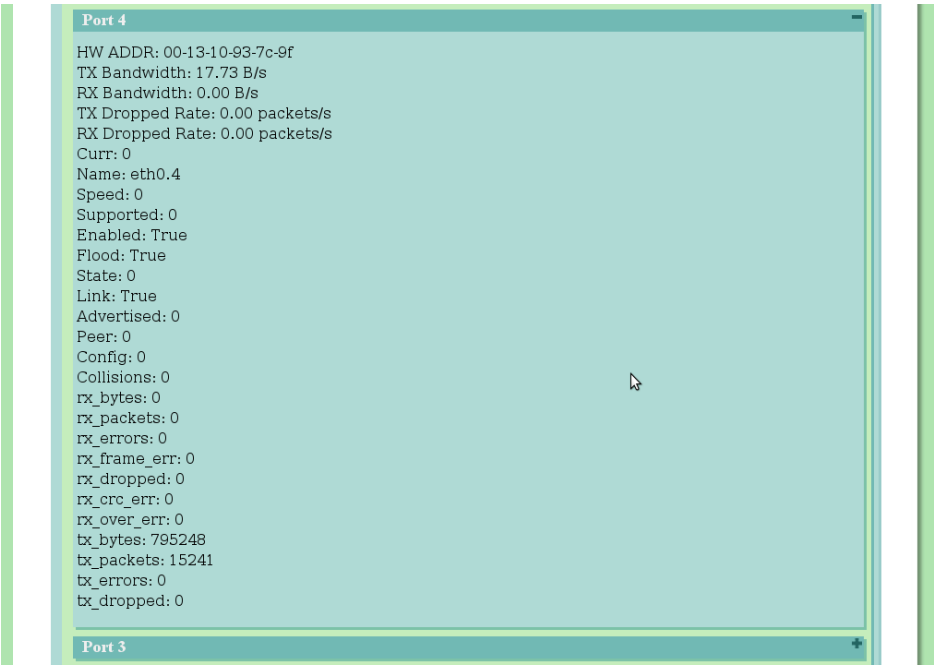


Figure 4.6: Switch Port Statistics.

number of flows in the network.

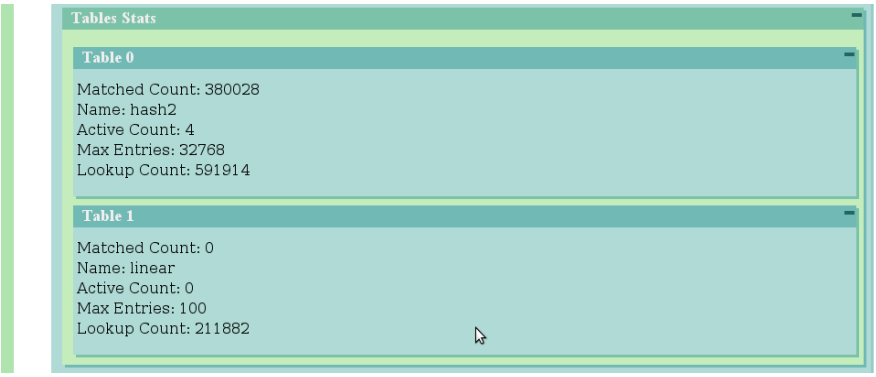


Figure 4.7: Switch Flow Tables Statistics.

Figure 4.8: Aggregated Flows Statistics.

Figure 4.9 presents the flows of a specific switch. It presents the flow count, actions and match.

Figure 4.10 presents a form to add flows in a switch. The form fields correspond to the flow match values and the hard timeout, idle timeout and output values.

4.2.3.3 Topology

This page provides a picture with the network and spanning tree topology view. In this view there are the IP and MAC addresses and the ports that are connected to links. We use a tool called Graphiviz [30] to generate the network topology picture. This tool generates a picture following a graph description that are in a file.

In the same way that occur with statistics and switches information, the network and spanning tree topology are stored in the Data Layer. This works this way because the discovery and spanning tree algorithm that are in the Data Layer are executed periodically, and when the algorithm stop to run they keep this information within them.

Figure 4.11 presents the Button to access the topology page indicated by the number 1 in the figure.

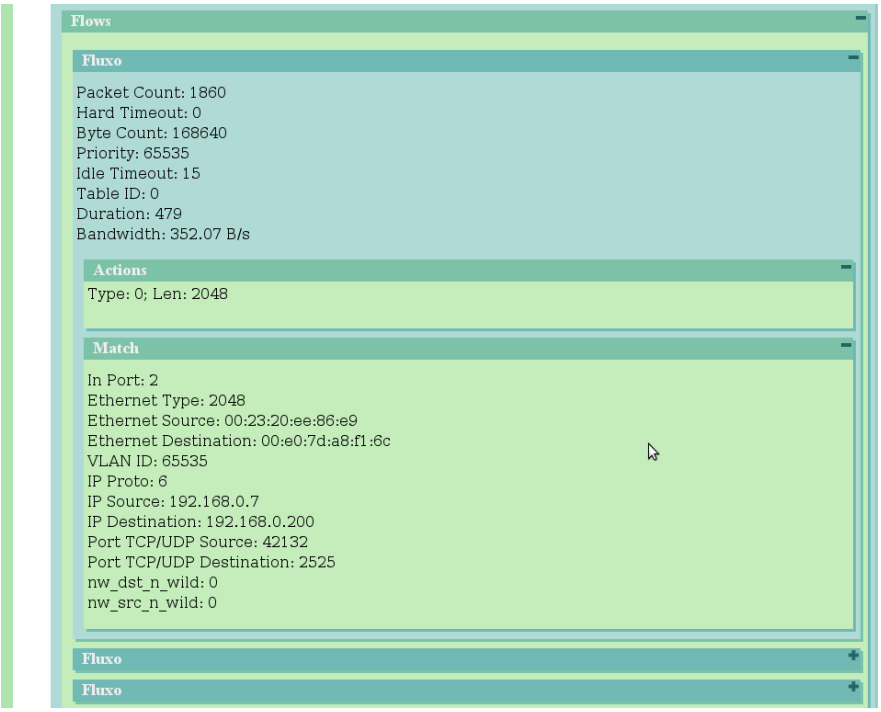


Figure 4.9: Flows of a Specific Switch.

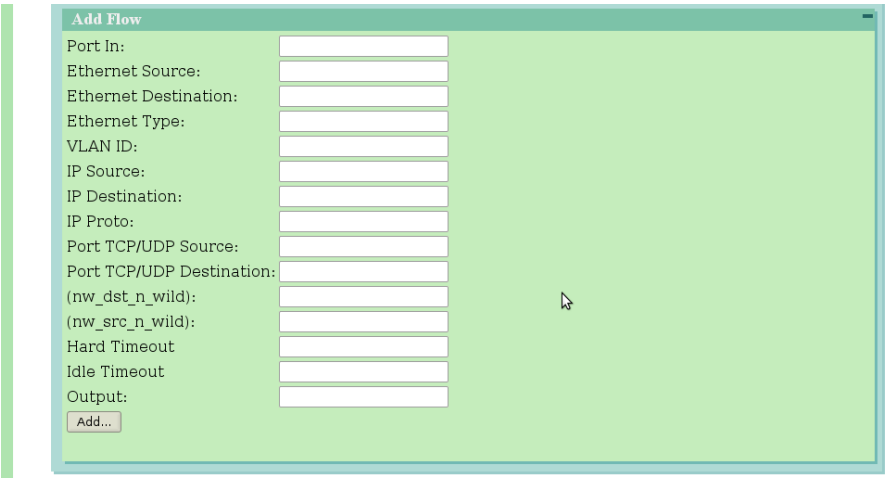


Figure 4.10: Form to add flows into switch.

Figure 4.12 presents the network topology. Each ellipse in the picture represents a connected switch in the network. The lines connecting the switches

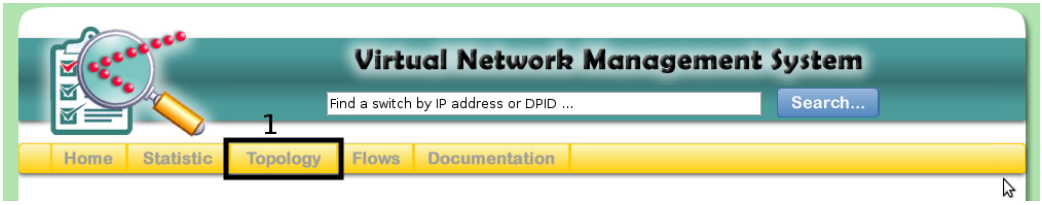


Figure 4.11: Button to access the Topology Page.

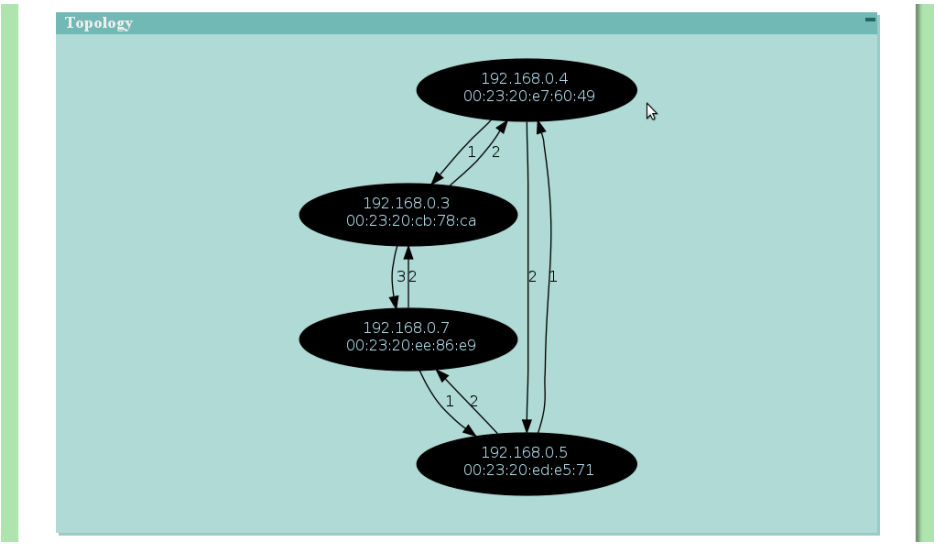


Figure 4.12: Network Topology.

represent the links. The number in the lines indicate the switches ports numbers. In this picture, the switch with IP address of value 192.168.0.5 is connected to switch with IP address of value 192.168.0.7 using the port 2, and the switch 192.168.0.7 is connected to switch 192.168.0.5 using the port 1. These two lines represent a physical link of the network.

Figure 4.13 presents the Spanning Tree. Each ellipse in the picture represents a connected switch in the network. The lines connecting the switches represent the links. The number in the lines indicate the switches ports numbers. In this picture, the switch with IP address of value 192.168.0.4 is connected to switch with IP address of value 192.168.0.5 using the port 2, and the switch 192.168.0.5 is connected to switch 192.168.0.4 using the port 1. These two lines represent a physical link of the network. Note that there are not a link between switch 192.168.0.5 and switch 192.168.0.7because of the use of Spanning Tree algorithm.

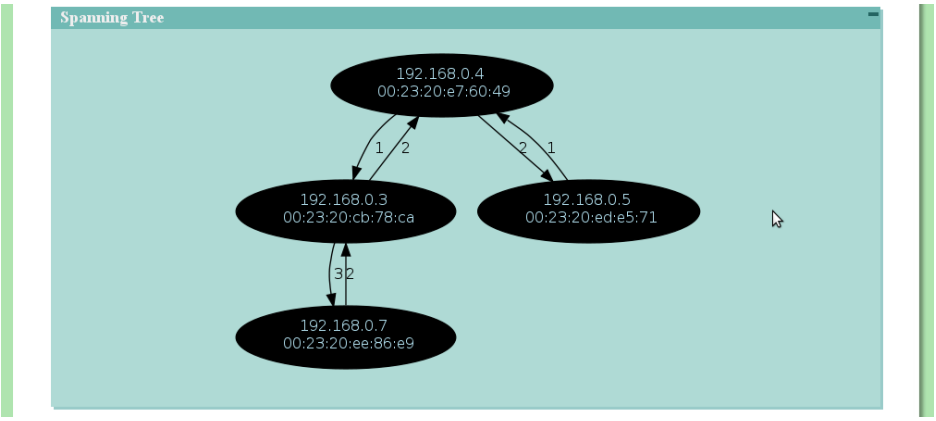


Figure 4.13: Spanning Tree.

4.2.3.4 Flows

This page shows all network flows and the switches that have theses flows, as well as input and output ports of each switch. Figure 4.14 presents the Button to access the topology page indicated by the number 1 in the figure.

In this page there are two forms to filter the flows, presented in Figure 4.16.The answer of the first form, which we call filter form, shows us only the flows founded by filter. The answer of the second form shows us a flow path view in the topology as well as the flows founded by the filter. This view is made using SVG and allow the execution of some actions such as create new flows, remove flows and migrate flows path. The two forms have the same fields and can be filled with some flows characteristics, such as the destination IP address and the source TCP port. The flow path view that are shown as result of the second form can be seen as a virtual network topology defined by filled fields in the filter. The area indicated by the number 1 in the figure is the button to access the filter form and the button indicated by the number 2 is the button to access the logical topology view form.

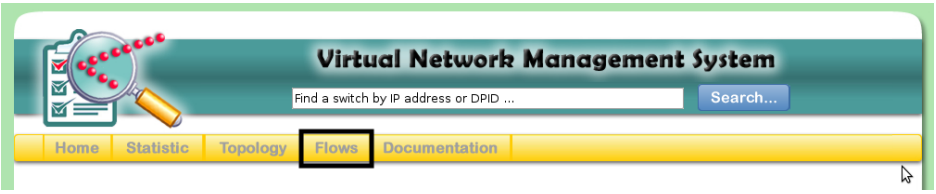


Figure 4.14: Button to access the network flows.

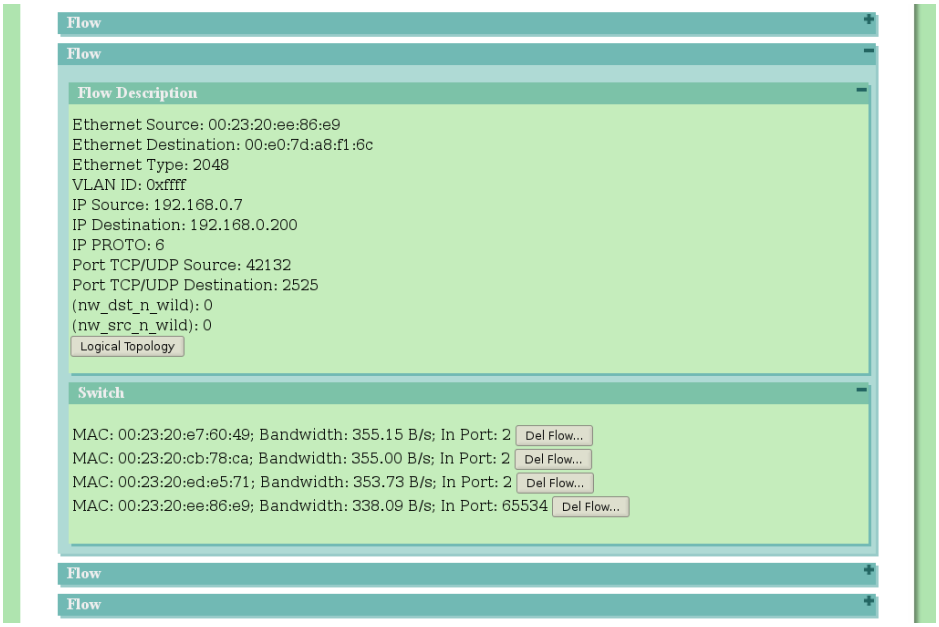


Figure 4.15: Network flows description.

Figure 4.15 presents Network flows description which is the answer of the filter form query. It presents the switches that have the flow, the flow bandwidth on each switch, the flow input port, a button to see the logic topology made by the described flow and the button to remove this flow from specifics switches.

Figure 4.17 presents a flow logical topology that we have mentioned before. The black links show the physical topology and the red links show the logical topology.

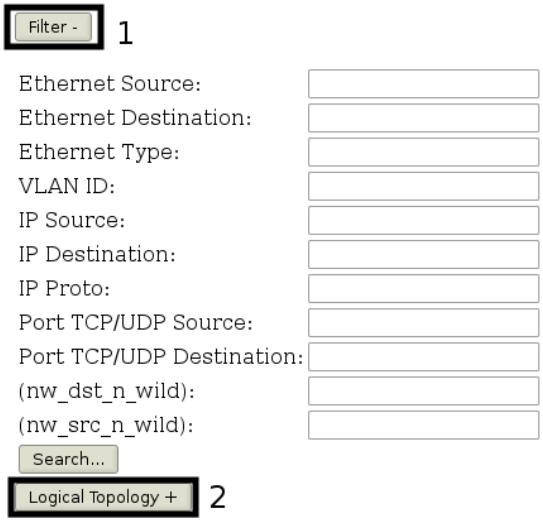


Figure 4.16: Form to filter flows. 1 Button to access filter form . 2 Button to access the logical topology view form.

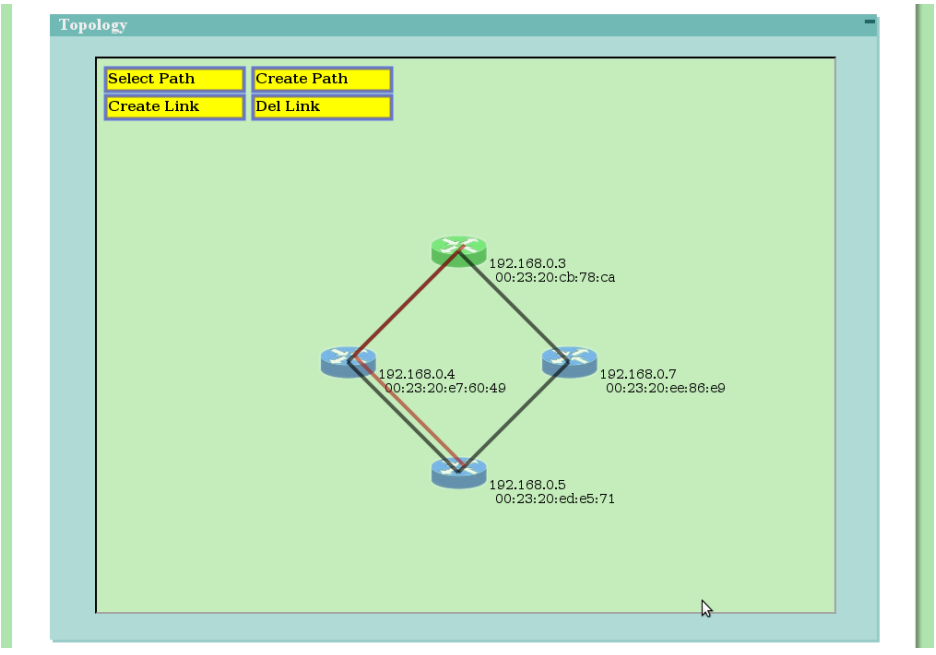


Figure 4.17: A flow Logical Topology.

Chapter 5

Prototype

5.1 Xen Prototype

In the Xen prototype, a Virtual Machine Server offers services that perform actions on the network infrastructure, such as creating new virtual routers. These services are accessed through a Graphical User Interface (GUI). In this section we show the physical indicators of the Virtual Machine Server and the GUI. We also describe the testbed used in our prototype.

5.1.1 Graphical User Interface

The GUI provides an interface to visualize the network topology, retrieve information from the network elements and execute management tasks such as turning elements on/off, changing parameters on routers and even migrating them. Figure 5.1 provides an overview of the GUI. Each number high lighted in the figure corresponds to a specific functionality:

1. Network topology view;
2. Information of the selected virtual or physical router (Properties module);
3. List of the registered physical nodes and each virtual nodes associated with them (Registered Nodes module);
4. Migration of virtual machines and routers (Migration module);
5. Customization interface to change GUI colors, apply different kinds of alpha effects etc. (Network Viewer Selection module).

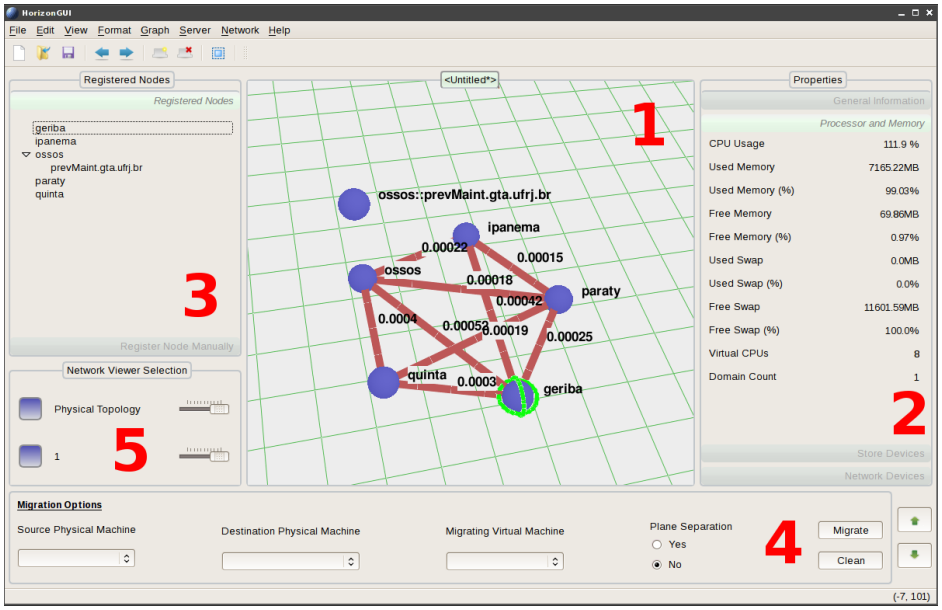


Figure 5.1: Graphical User Interface Overview.

The Topology View, enlarged in Figure 5.2, is automatically loaded when the GUI connects to the Controller, which aggregates information from the nodes. In the topology, we can retrieve the name of each physical machine and some information regarding the latency of the physical links. In a plane above the physical machine, it is possible to visualize the virtual machines that resides on each physical machine. In the given example, we have five physical machines. The chosen physical machine has a virtual machine called ossos::prevMaint.gta.ufrj.br. From the Topology View we can manually select a node and show its information in the Properties Module. This action is indicated in Figure 5.2, were the Graphical User Interface shows the information about the node marked with green hoops. The Properties Module thus provides different machine information like CPU and memory usage, number of virtual CPUs etc.

The Xen prototype allows the migration of virtual routers between physical machines. The GUI provides an easy way to do this action through the Migration module. Figure 5.3 shows an example of this module utilization. In this figure we show a migration example of the virtual router prevMaint.gta.ufrj.br from the machine Ossos to the machine Ipanema. The Migration Module also allows the migration using data/control plane separation [17].

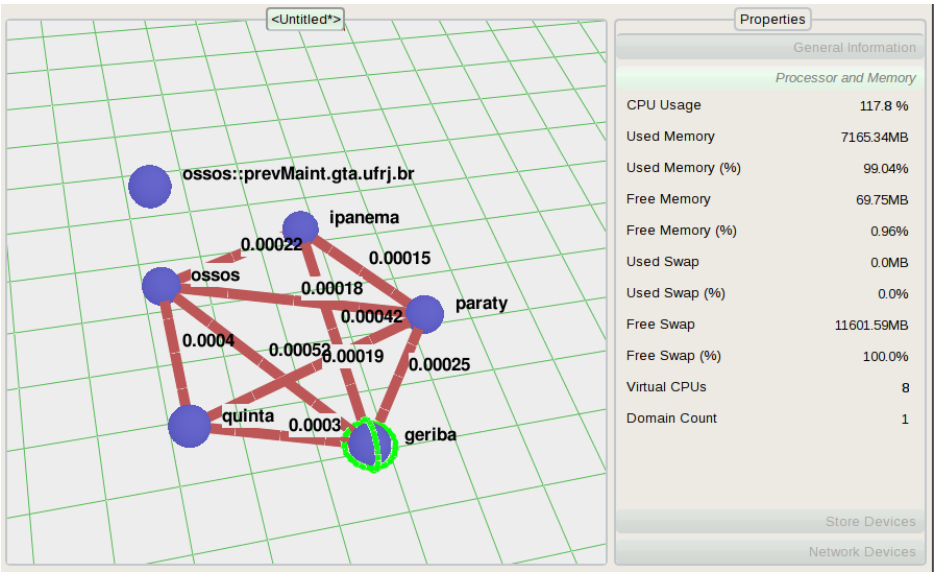


Figure 5.2: Topology View and Properties module.

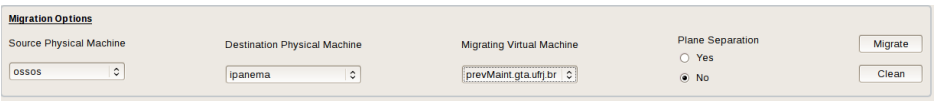


Figure 5.3: Migration module.

5.1.2 Virtual Machine Server

Most of the actions and informations provided by the GUI use services from the Virtual Machine Server, installed on a machine called Controller. This section shows some piece of the Virtual Machine Server source code, exemplifying its implementation. Each Virtual Machine Server service is implemented as a method of server's main class called `VirtualMachineServer`. As an example, Listing 5.1 shows the Java implementation of the method `createVirtualMachine`, that creates a virtual machine. Every service must be implemented as a public method that receives an object of the class `OMElement` (Object Model Element) and returns another object of this class. The `OMElement` class is offered by the Axis2 [31] library and aims at storing an XML (eXtensible Markup Language) element. In other words, each service performs actions by receiving and returning XML elements. These XML messages are carried by SOAP [25] messages. A client must send to the Virtual Machine Server this kind of message with the desired service and its parameters. To ease the development of a client, we implement the `HorizonXenClient` class that creates the payload of SOAP messages specific to each service. The Listing 5.2 shows, as an example, the method `createVirtualMachinePayload` which is used to generate the payload in order to use the method `createVirtualMachine` of the Virtual Machine Server. Finally, Listing 5.3 shows an example of a client implementation that creates a new virtual machine, using the method `createVirtualMachinePayload`.

5.1.3 Xen Testbed

To experiment with the GUI and the Virtual Machine Server presented in the last sections, we built the testbed of Figure 5.4. This testbed consists of 5 machines acting as network nodes, labelled with numbers from 1 to 5, one machine acting as a controller and another one as a NFS Server. The NFS Server stores the virtual hard disk of each virtual machine. All machines are connected to the GTA Lab. production network, and so the Controller and NFS Server use this network to access the nodes. The GUI can be installed on any GTA machine that is connected to the GTA network. Using this network the GUI can make requests to the Virtual Machine Server running on the Controller. The direct connections between nodes are represented in Figure 5.4 with dashed lines and the box with red axes represents the nodes connection using a switch. The connections between all the machines and the GTA network are omitted and use this same switch, but in a different VPN (Virtual Private Network). Using the machines Quinta and Ipanema of Figure 5.4, we can attach traffic receivers and generators to the network.

Listing 5.1: `createVirtualMachine` method.

```
1  public OMElement createVirtualMachine(OMElement element) {
2          element.build();
3          element.detach();
4          Iterator it = element.getChildElements();
5          Vector<OMElement> ele = new Vector();
6          ele.clear();
7          String returnText = "SUCCESS - Virtual Machine created\nAttributes:\n";
8          String result = "SUCCESS";
9          String att = "";
10         String phyServer = "", vmName = "", vmRAM = "", vmIP = "";
11         while (it.hasNext()) {
12             ele.add((OMElement) it.next());
13             if (ele.lastElement().getLocalName().equals("phyServer")) {
14                 phyServer = ele.lastElement().getText();
15                 att = att + "phyServer:" + phyServer + "\n";
16             }
17             if (ele.lastElement().getLocalName().equals("vmName")) {
18                 vmName = ele.lastElement().getText();
19                 att = att + "vmName:" + vmName + "\n";
20             }
21             if (ele.lastElement().getLocalName().equals("vmIP")) {
22                 vmIP = ele.lastElement().getText();
23                 att = att + "vmIP:" + vmIP + "\n";
24             }
25             if (ele.lastElement().getLocalName().equals("vmRAM")) {
26                 vmRAM = ele.lastElement().getText();
27                 att = att + "vmRAM:" + vmRAM + "\n";
28             }
29
30         }
31         try {
32             if (!createVirtualMachine(phyServer, vmName, vmRAM, vmIP)) {
33                 returnText = "ERROR - The Domain could not be created - \nAttributes:\n"
                       ;
34                 result = "ERROR";
35             }
36         } catch (LibvirtException ex) {
37             returnText = "ERROR - The Domain could not be created - Exception: " + ex.
                   getMessage() + "\nAttributes:\n";
38             result = "ERROR";
39         }
40         returnText = returnText + att;
41         OMFactory fac = OMAbstractFactory.getOMFactory();
42         OMNamespace omNs = fac.createOMNamespace(URI, PREFIX);
43         OMElement retElement = fac.createOMElement("createVirtualMachineResponse", omNs)
               ;
44
45         OMElement value = fac.createOMElement("result", omNs);
46         value.addChild(fac.createOMText(value, result));
47
48         retElement.addChild(value);
49         //method.addChild(fac.createOMText(returnText));
50         return retElement;
51 }
```

Listing 5.2: `createVirtualMachinePayload` method.

```
1  public OMElement createVirtualMachinePayload(String phyServer, String vmName, String
       vmIP, String vmRAM, String vmDiskSize) {
2          OMFactory fac = OMAbstractFactory.getOMFactory();
3          // Set the namespace of the messages
4          OMNamespace omNs = fac.createOMNamespace(URI, PREFIX);
5          // Set the required operation
6          OMElement element = fac.createOMElement("createVirtualMachine", omNs);
7
8          // Attributes
9
10         // Physical Server IP
11         OMElement value = fac.createOMElement("phyServer", omNs);
12         value.addChild(fac.createOMText(value, phyServer));
13         element.addChild(value);
14         // Virtual Machine Name
15         value = fac.createOMElement("vmName", omNs);
16         value.addChild(fac.createOMText(value, vmName));
17         element.addChild(value);
18         // Virtual Machine IP Address
19         value = fac.createOMElement("vmIP", omNs);
20         value.addChild(fac.createOMText(value, vmIP));
21         element.addChild(value);
22         // Virtual Machine RAM Memory Size
23         value = fac.createOMElement("vmRAM", omNs);
24         value.addChild(fac.createOMText(value, vmRAM));
25         element.addChild(value);
26         // Virtual Machine Hard Disk Size
27         value = fac.createOMElement("vmDiskSize", omNs);
28         value.addChild(fac.createOMText(value, vmDiskSize));
29         element.addChild(value);
30
31         return element;
32
33 }
```

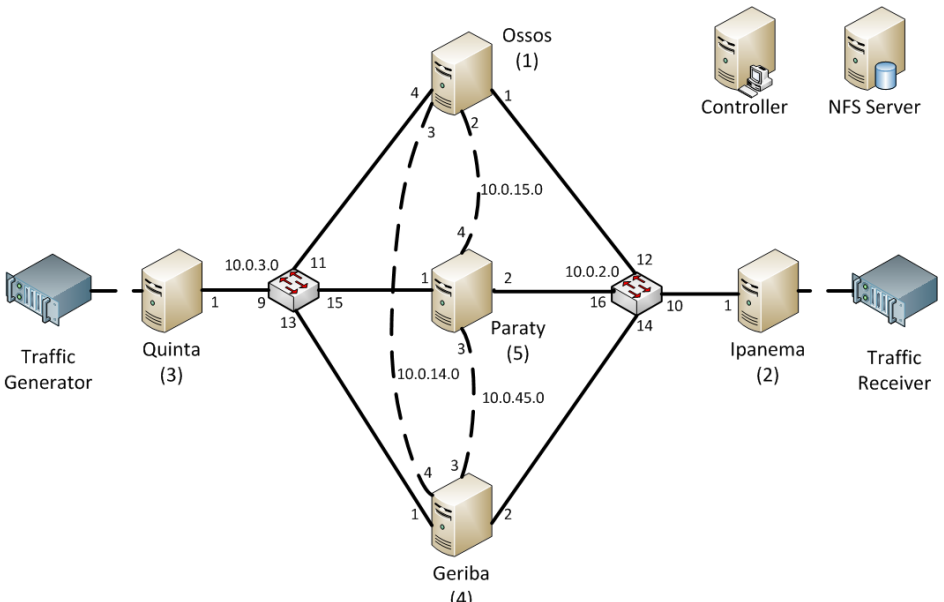


Figure 5.4: Xen testbed topology.

Listing 5.3: Example of a client implementation.

```
1  public static void main(String[] args) {
2    try {
3      HorizonXenClient hxc = new HorizonXenClient();
4      // set options to send message
5      Options options = new Options();
6      options.setTo(hxc.targetEPR);
7      options.setTransportInProtocol(Constants.TRANSPORT_HTTP);
8      // create a Web Service client
9      ServiceClient sender = new ServiceClient();
10     sender.setOptions(options);
11     // creating message payload
12     OMElement messagePayload = hxc.createVirtualMachinePayload("phyServer","vmName","
           10.0.0.1","65536");
13     // send message and wait for the server response
14     OMElement result = sender.sendReceive(messagePayload);
15   } catch (Exception e) {
16     e.printStackTrace();
17   }
18 }
```

Figure 5.5 shows the 5 nodes and the Controller of the testbed physical installation. The NFS server is located in another room and is not shown in this figure. Figure 5.6 shows the switch used in the testbed.

Figure 5.5: Xen testbed physical installation.

Figure 5.6: Xen testbed switch.

5.2 OpenFlow Prototype

The OpenFlow prototype consists of several machines acting as OpenFlow switches, one machine acting as a controller running NOX, and other machines used to generate and receive traffic. This network is managed using some applications developed to run in the NOX controller, and a Graphical User Interface (GUI) that use them. This GUI offers an easy way to manage the OpenFlow network. In this section we show the physical indicator of the NOX applications and the GUI. We also show the testbed used in our prototype.

5.2.1 NOX Applications

We develop in the OpenFlow prototype different network management applications running in NOX. In this section we exemplify with the Flow Migration Application. This application allows the network controller to reorganize the data flow through the network. It is an important application in order to execute load-balancing algorithms and to choose paths based on the Quality of Service (QoS) of links. The main code of the Flow Migration NOX Application is shown on Listings 5.4 and 5.5. This code is a Python module and runs over the NOX platform. This application is also a good example to understand the development of a NOX application. As seen in the source code of Listings 5.4 and 5.5, a NOX application is an extension of the Component class and it should implement, at least, the `__init__(self, context)` and

Listing 5.4: Source code example of the Flow Migration NOX Application.

```
1  from nox.lib.core import *
2  from nox.lib.packet.ethernet import ethernet
3  from nox.lib.packet.packet_utils import *
4  from nox.lib.netinet.netinet import create_datapathid_from_host
5  from nox.coreapps.examples.pyswitch import pyswitch
6  from nox.netapps.discovery.discovery import discovery
7  from nox.netapps.spanning_tree.spanning_tree import Spanning_Tree
8  from nox.netapps.discovery.flow_manager import flow_manager
9  from migrationHandler import MigrationHandler
10 from utils import Utils
11 import logging
12 from time import time
13 from xmlparser import importXmlParser
14
15 class flow_migration (Component):
16   def __init__ (self , ctxt):
17     Component.__init__(self , ctxt)
18     self .logger = logging.getLogger('flow_migration|nox.netapps.flow_manager.
           flow_manager' )
19     self .lastMileTp = {}
20   def install (self ):
21     self .discovery = self .resolve(discovery)
22     self .flow_manager = self .resolve(flow_manager)
23     self .pyswitch = self .resolve(pyswitch)
24     self .spanning_tree = self .resolve(Spanning_Tree)
25     self .logger.info("flow_migration:␣ON" )
```

the `install(self)` methods. The first one is the class constructor, which is responsible for calling the Component constructor. The second one is called as soon as the controller calls the application. The other Flow Migration Application methods implement the flow migration algorithm.

5.2.2 Graphical User Interface

This GUI is based on a web interface, offering a way to control all OpenFlow network resources. The following images are screens of our web application interface. We show the main screenshots of the applications and tools we developed. Figure 5.7 shows the Home screen of the interface developed.

The GUI has different tools to manage the OpenFlow network. Figure 5.8 shows the Topology Visualization tool, where the ellipses represent the OpenFlow switches connected to the network. Figure 5.9 shows the Flow Visualization tool, that lists each instantiated flow in the network. Using this interface we can view each flow's characteristics, like associated network addresses, simply by clicking on the flow. Figure 5.10 shows the Switch Statistics tool that presents the statistics of each OpenFlow switch, like the number of active flows. Each line of this tool represents a switch. Clicking on one line, the

Listing 5.5: Source code example of the Flow Migration NOX Application (cont.).

```
66    def MigrateFlow (self , dpidsStr, match, priority = 65535 ,hardTimeOut = openflow.
          OFP_FLOW_PERMANENT,= openflow.OFP_FLOW_PERMANENT):
67      xml = self .discovery.get_neighbor_xml()
68      migrationHandler = MigrationHandler(dpidsStr, xml)
69      migrationHandler.getEntireDpidPath()
70      migrationHandler.getPorts()
71      dpids = migrationHandler.dpid_intForm
72      output_ports = migrationHandler.output_ports
73      input_ports = migrationHandler.input_ports
74      dpids.reverse()
75      output_ports.reverse()
76      input_ports.reverse()
77      try :
78         in_port = int(match['in_port' ])
79      except :
80         in_port = None
81      self .logger.info(str(dpids))
82      ret = '<root>\n'
83      ret = ret + '<inport>%s</inport>\n' %str(input_ports)
84      for i in range( 0 , len(dpids)):
85         continue_bool = True
86         if (i != (len(dpids)- 1 )):
87           match['in_port' ] = input_ports[i]
88         elif in_port != None :
89           match['in_port' ] = in_port
90         else :
91           del match['in_port' ]
92         if (output_ports[i] == - 1 ):
93           try :
94             output_ports[i] = self .pyswitch.lastMileTp[dpids[i]][match['dl_dst' ]]
95             utils = Utils()
96           except KeyError:
97             continue_bool = False
98         if continue_bool:
99           actionObj = [openflow.OFPAT_OUTPUT, [ 0 , output_ports[i]]]
100          add_args = (idleTimeOut, [actionObj], UINT32_MAX)
101          self .flow_manager.manage_flow(dpids[i],'add',match,priority,add_args,
                 hardTimeOut,i)
102          ret = ret + '<dpid>%s</dpid>\n<match>%s</match>\n<in_port>%s</in_port>\n<output
                 >%s</output>\n' %(str(dpids[i]),str(match),str(match['in_port' ]),str(
                 output_ports[i]))
103        else :
104          ret = ret + '<Error><match>%s</match>\n<type>No␣Last␣Mile␣Info</type>\n</Error
                 >\n' %(str(match)
105     return ret+"<result>Migration␣is␣over</result>\n</root>"
106   def getInterface (self):
107     return str(flow_migration)
108   def getFactory ():
109     class Factory :
110       def instance (self , ctxt):
111         return flow_migration (ctxt)
112     return Factory()
```

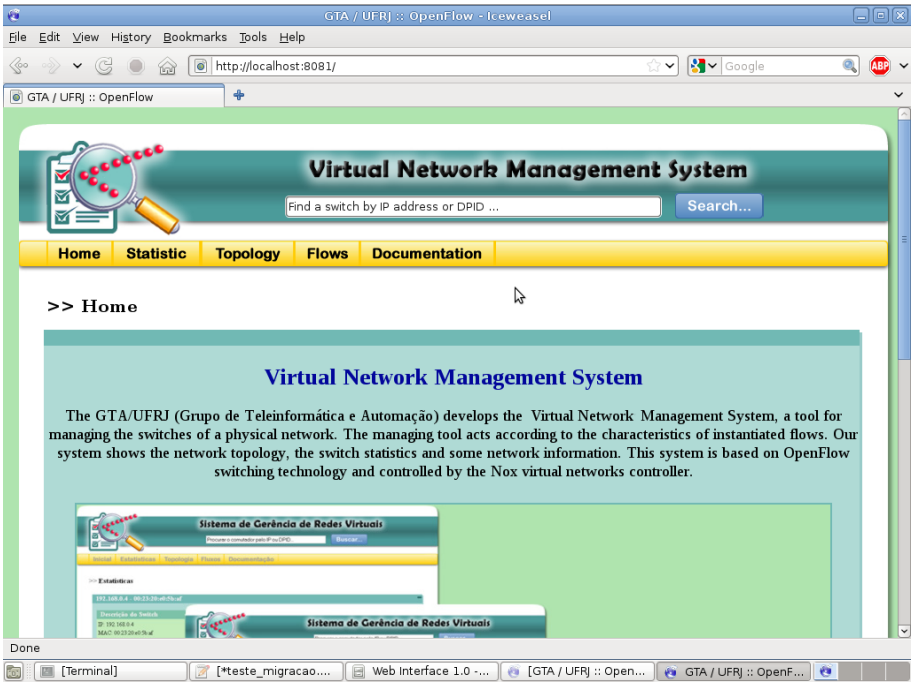


Figure 5.7: Graphical user interface Home screen.

switch statistics are shown. Figure 5.11 shows the Virtual Topology tool. This tool shows how a flow is configured through the physical network. The black lines represent the network physical links and the red lines represent the logical links of the flow. Finally, Figure 5.12 shows the Flow Migration Tool, that is used to migrate a flow to another set of switches. This figure shows the confirmation dialog, that appears when a flow migration is completed.

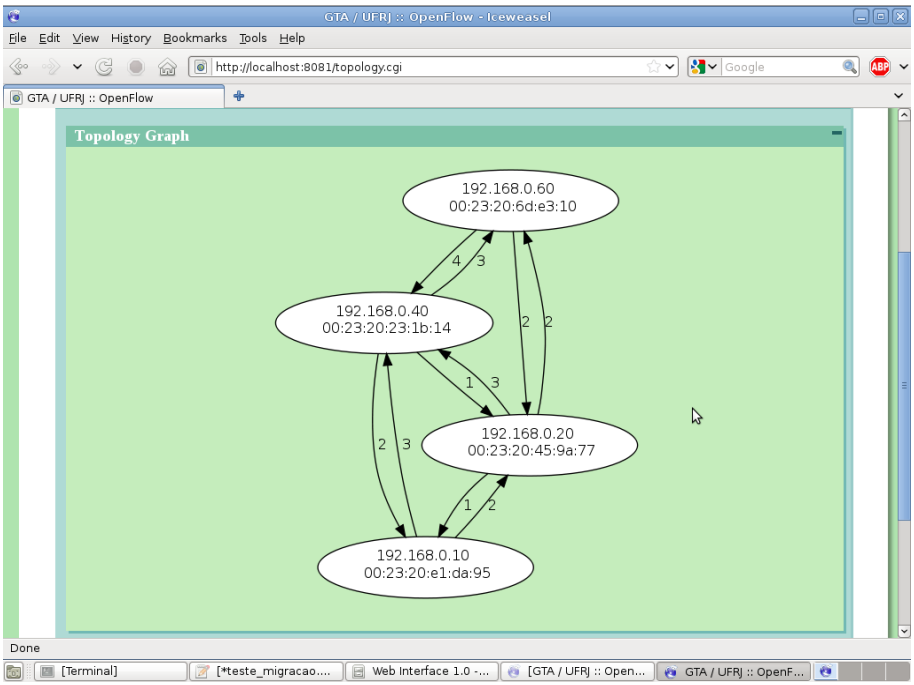


Figure 5.8: Network Topology Visualization tool.

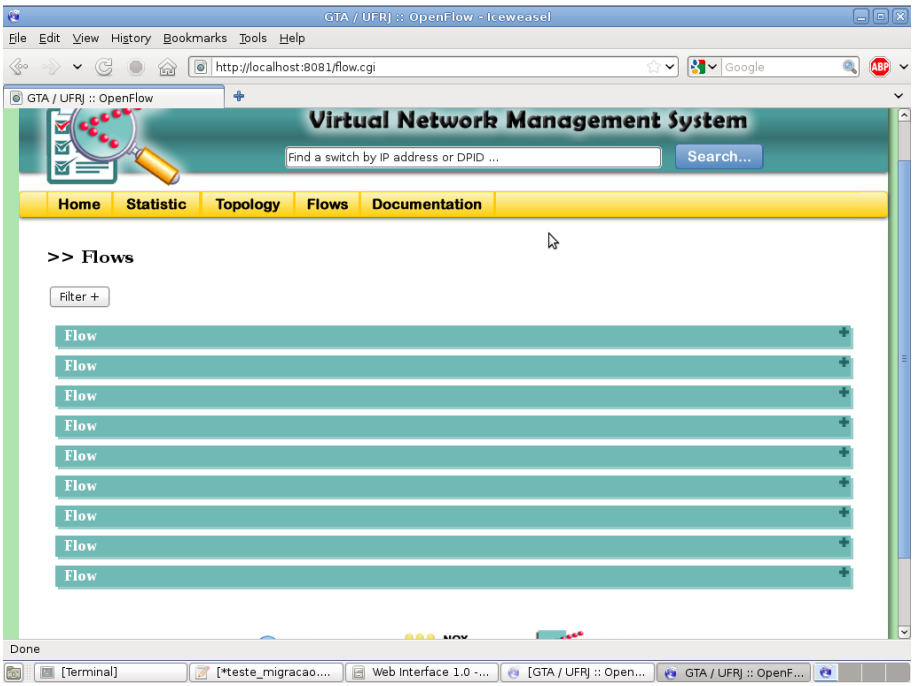


Figure 5.9: Flow Visualization tool.

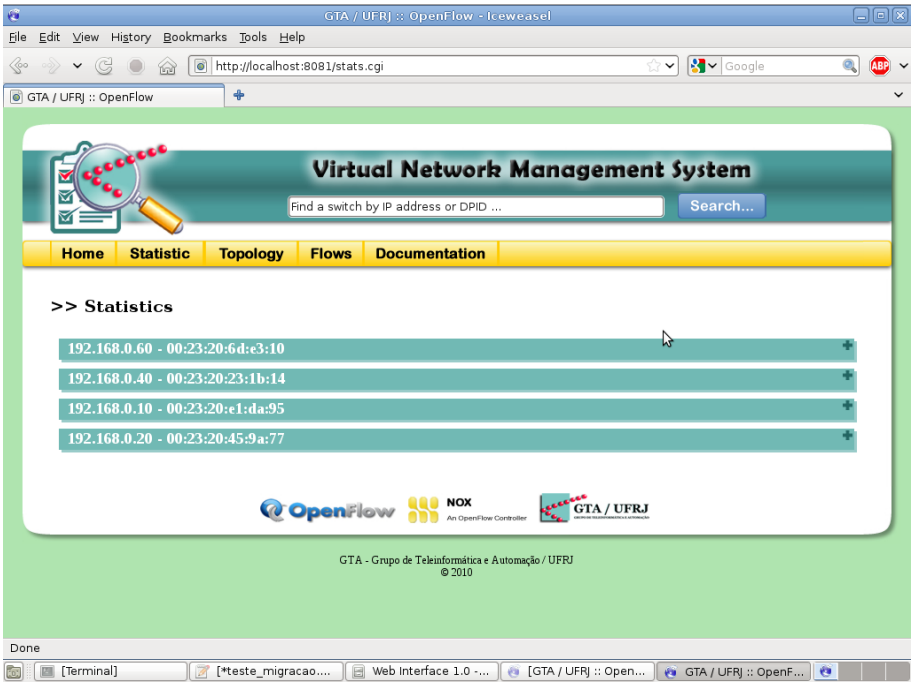


Figure 5.10: Switch Statistics tool.

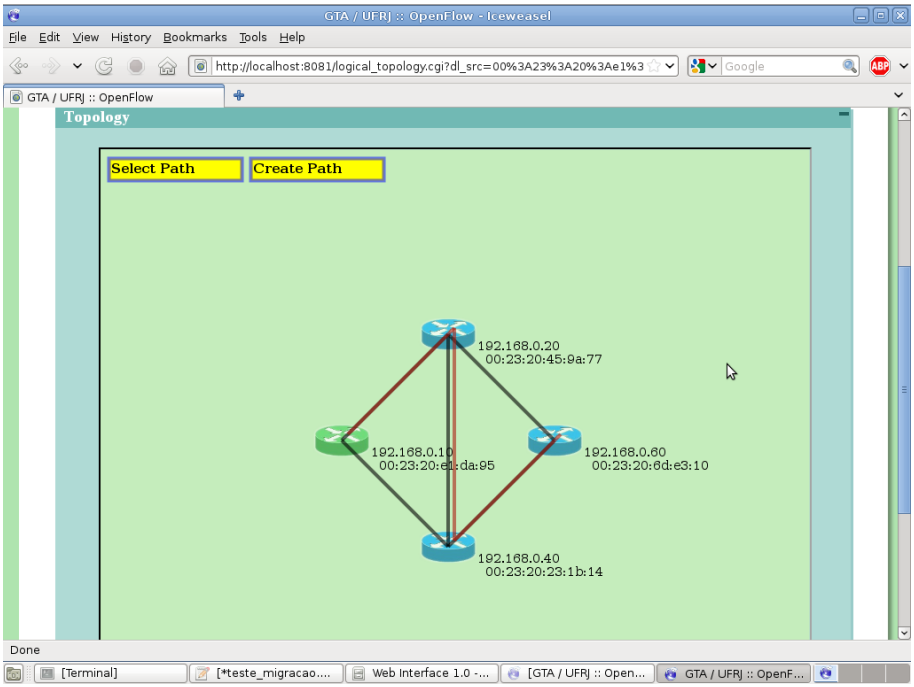


Figure 5.11: Virtual Topology tool.

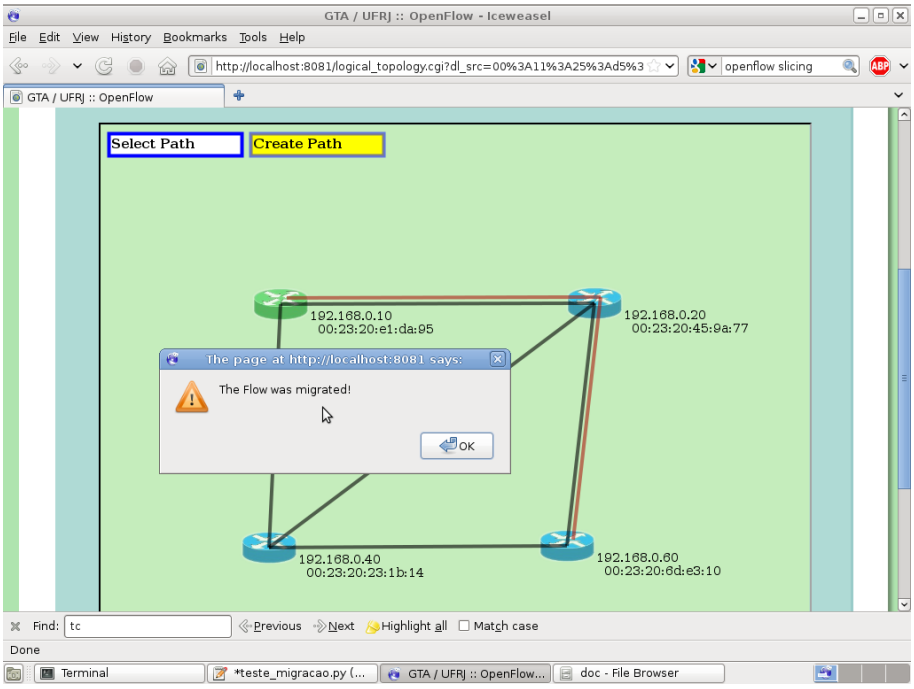


Figure 5.12: Flow Migration tool.

5.2.3 OpenFlow Testbed

In order to use the applications we developed, we deploy a testbed network as shown in Figure 5.13. Our testbed network is composed of four computers, acting as OpenFlow switches, two laptops, which are the traffic generator and the receiver, and a Controller, which is a computer that runs the NOX controller and the web interface server. In this testbed we can configure different topologies and set different points of generation and reception of network traffic.

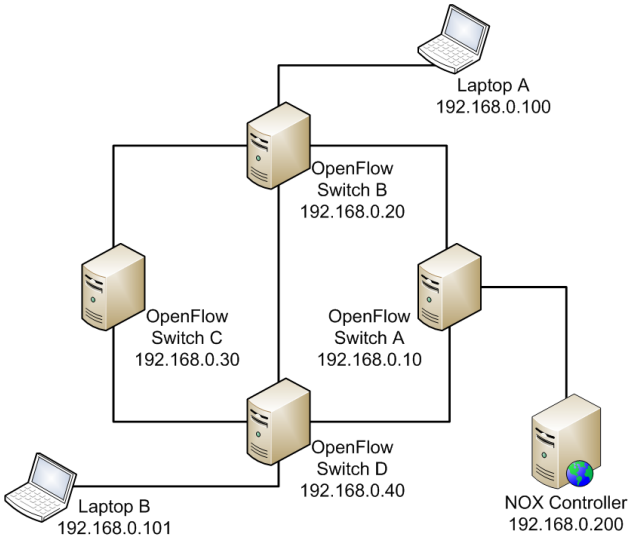


Figure 5.13: OpenFlow testbed topology.

Figure 5.14 shows the testbed physical installation. The four desktop computers are running the OpenFlow Switch software. To perform an experiment, one laptop generates packets, which are forwarded by the OpenFlow network in order to reach the other laptop. Each OpenFlow switch has at least three network interfaces, allowing us to test different network topologies. The switch that is directly connected to the Controller is the one with more network interfaces, in order to improve the testbed topology.

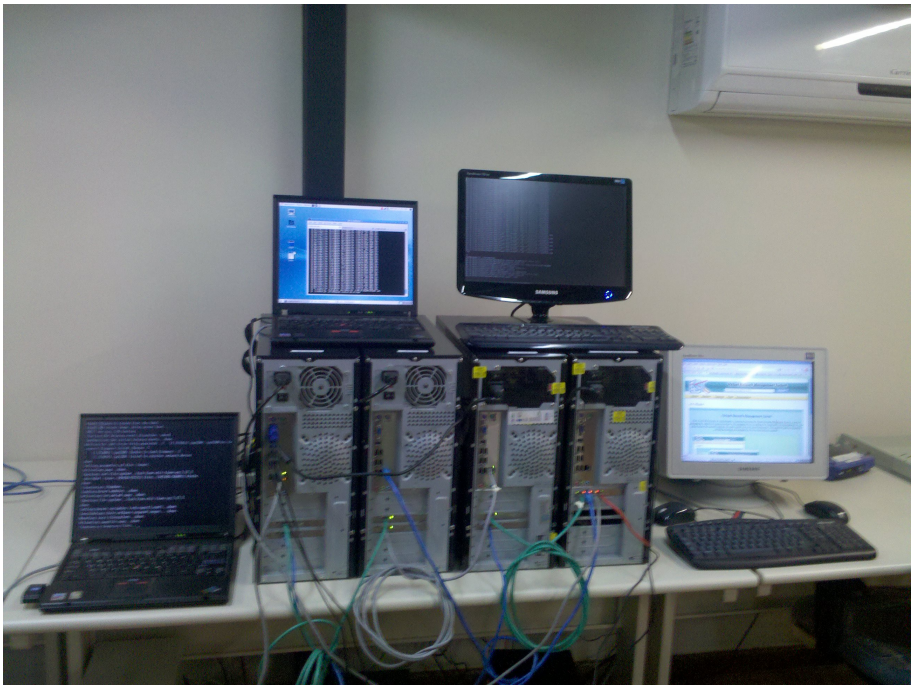


Figure 5.14: OpenFlow testbed physical installation.

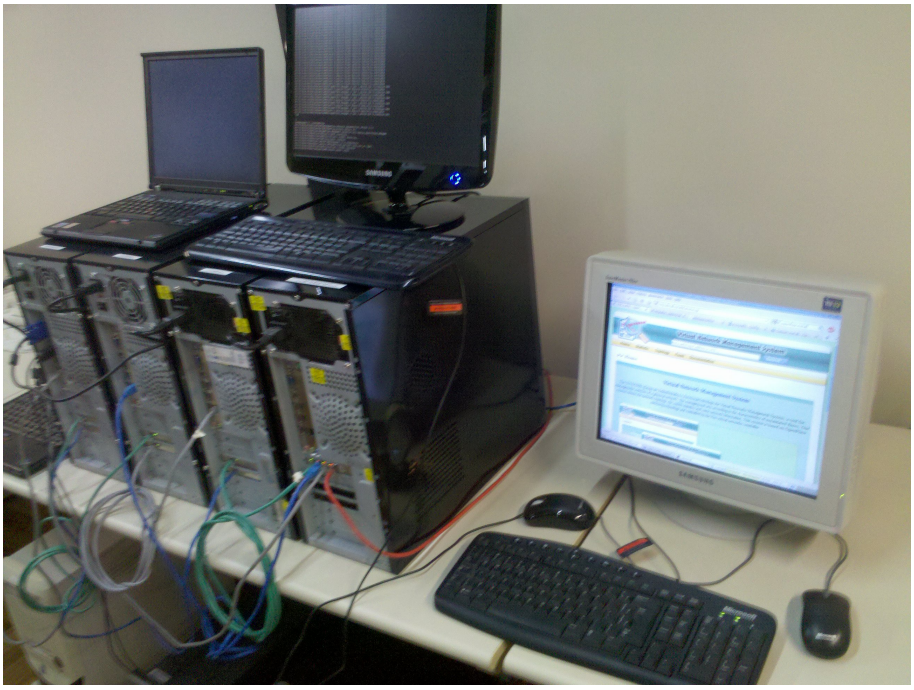


Figure 5.15: The Controller running the administrative web interface.

Chapter 6

Conclusions and Ongoing Work

The Horizon Project assumes a pluralist architecture for the Future Internet. Therefore, we assume that the infrastructure is virtualized. In this report, we conduct performance measurements using two virtualization platforms, OpenFlow and Xen. Our results show that Xen enables a highly flexible environment, with different protocol stacks running in parallel using customized network-data forwarding structures and lookup algorithms. This flexibility has a high performance cost limiting the virtual machine packet forwarding capacity to less than 200 kp/s. On the other hand, OpenFlow shows a packet forwarding performance similar to a native Linux environment.

We developed a prototype for both virtualization platforms. On this report, we describe the interfaces developed for each one of the prototypes. For the Xen platform a Virtual Machine Server was developed using the Web Service concept. Using the Virtual Machine Server, physical and virtual hosts of the network can be controlled. To simplify the network administration by a human agent, a graphical user interface was developed. This interface can be used to show the topology of the network and act on its elements.

In the OpenFlow prototype we developed similar tools to those developed for the Xen prototype. The WebServer App offers a Web interface to agents interested in administrate the OpenFlow network. A graphical user interface was developed. This interface can be accessed using a Web browser to view information related to the OpenFlow network.

We plan to add more services to both virtualization platforms, Xen and OpenFlow. These services must cover other administrative tasks that cannot be done on the current testbeds. Besides, a unique interface will be developed in order to unify the calls to the services on the testbeds. Hence, the underlying virtualization approach becomes transparent for the agent controlling the network infrastructure.

Bibliography

[1] P. Baran, "On distributed communications networks," *IEEE Transactions on Communications Systems*, vol. 12, no. 1, pp. 1–9, Mar. 1964.

[2] D. Clark, R. Braden, K. Sollins, J. Wroclawski, D. Katabi, J. Kulik, X. Yang, T. Faber, A. Falk, V. Pingali, M. Handley, and N. Chiappa, "New Arch: Future generation Internet architecture," tech. rep., USC Information Sciences Institute Computer Networks Division, MIT Laboratory for Computer Science and International Computer Science Institute (ICSI), Aug. 2004.

[3] M. S. Blumenthal and D. D. Clark, "Rethinking the design of the Internet: the end-to-end arguments vs. the brave new world," *ACM Transactions on Internet Technology*, vol. 1, no. 1, pp. 70–109, Aug. 2001.

[4] A. Karouia, R. Langar, T.-M.-T. Nguyen, and G. Pujolle, "SOA-based approach for the design of the future internet," in *Communication Networks and Services Research Conference (CNSR)*, pp. 361–368, May 2010.

[5] T. Anderson, L. Peterson, S. Shenker, and J. Turner, "Overcoming the Internet impasse through virtualization," *IEEE Computer*, vol. 38, no. 4, pp. 34–41, Apr. 2005.

[6] D. F. Macedo, Z. Movahedi, J. Rubio-Loyola, A. Astorga, G. Koumoutsos, and G. Pujolle, "The autoi approach for the orchestration of autonomic networks," *Annals of Telecommunications*, June 2010.

[7] N. Feamster, L. Gao, and J. Rexford, "How to lease the Internet in your spare time," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 1, pp. 61–64, Jan. 2007.

[8] F. L. Verdi, M. F. Magalhães, E. Madeira, and A. Welin, "Using virtualization to provide interdomain QoS-enabled routing," *Journal of Networks*, vol. 2, no. 2, pp. 23–32, Apr. 2007.

[9] N. C. Fernandes, M. D. D. Moreira, I. M. Moraes, L. H. G. Ferraz, R. S. Couto, H. E. T. Carvalho, M. E. M. Campista, L. H. M. K. Costa, , and O. C. M. B. Duarte, "Virtual networks: Isolation, performance, and trends," tech. rep., Electrical Engineering Program, COPPE/UFRJ, June 2010.

[10] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, L. Mathy, and T. Schooley, "Evaluating Xen for router virtualization," in *International Conference on Computer Communications and Networks - ICCCN*, pp. 1256–1261, Aug. 2007.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S., and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[12] M. P. Mateo, "OpenFlow switching performance," Master's thesis, Politecnico Di Torino, Torino, Italy, July 2009.

[13] A. Menon, A. L. Cox, and W. Zwaenepoel, "Optimizing network virtualization in Xen," in *USENIX Annual Technical Conference*, pp. 15–28, May 2006.

[14] B. Pfaff, B. Heller, D. Talayco, D. Erickson, G. Gibb, G. Appenzeller, J. Tourrilhes, J. Pettit, K.-K. Yap, M. Casado, M. Kobayashi, N. McKeown, P. Balland, R. Price, R. Sherwood, and Y. Yiakoumis, "OpenFlow switch specification version 1.0.0 (wire protocol 0x01)," tech. rep., Stanford University, Dec. 2009.

[15] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, July 2008.

[16] R. Sherwood, M. Chan, A. Covington, G. Gibb, M. Flajslik, N. Handigol, T.-Y. Huang, P. Kazemian, M. Kobayashi, J. Naous, S. Seetharaman, D. Underhill, T. Yabe, K.-K. Yap, Y. Yiakoumis, H. Zeng, G. Appenzeller, R. Johari, N. McKeown, and G. Parulkar, "Carving research slices out of your production networks with OpenFlow," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, no. 1, pp. 129–130, 2010.

[17] P. S. Pisa, N. C. Fernandes, H. E. T. Carvalho, M. D. D. Moreira, M. E. M. Campista, L. H. M. K. Costa, and O. C. M. B. Duarte, "Openflow and Xen-based virtual network migration," in *The World Computer Congress - Network of the Future Conference*, Sept. 2010.

[18] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Symposium on Networked Systems Design & Implementation - NSDI*, pp. 273–286, May 2005.

[19] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford, "Virtual routers on the move: Live router migration as a network-management primitive," in *ACM SIGCOMM*, pp. 231–242, Aug. 2008.

[20] R. Olsson, "Pktgen the Linux packet generator," in *Linux symposium*, pp. 11–24, July 2005.

[21] D. Chisnall, *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, 2007.

[22] H. Fathi, R. Prasad, and S. Chakraborty, "Mobility management for VoIP in 3G systems: Evaluation of low-latency handoff schemes," *IEEE Wireless Communications*, vol. 12, no. 2, no. 2, pp. 96–104, 2005.

[23] R. S. Alves, L. H. M. K. Costa, M. E. M. Campista, L. G. Valverde, P. S. Pisa, C. Fragni, T. N. Ferreira, I. M. Moraes, and O. C. M. B. Duarte, "A virtual machine server for the future internet," in *Workshop on Network Virtualizaton and Intelligence For Future Internet (WNetVirt)*, Apr. 2010.

[24] W3C, "Web services activity." http://www.w3.org/2002/ws/. (Accessed march 2010).

[25] D. Box, D. Ehnebuske, G. Kakivaya, A. L. N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple object access protocol (soap) 1.1." http://www.w3.org/TR/2000/NOTE-SOAP-20000508/, 2000. (Accessed march 2010).

[26] "Apache tomcat." http://tomcat.apache.org/. (accessed march 2010).

[27] "Libvirt: The virtualization api." http://libvirt.org/. (accessed march 2010).

[28] D. Menezes, N. Fernandes, C. Gomes, and O. Duarte, "Developing nox applications for network control," in *Workshop on Network Virtualizaton and Intelligence For Future Internet (WNetVirt)*, Apr. 2010.

[29] C. Gomes, D. Menezes, N. Fernandes, and O. Duarte, "A tool for openflow network management," in *Workshop on Network Virtualizaton and Intelligence For Future Internet (WNetVirt)*, Apr. 2010.

[30] "Graphiviz." http://www.graphviz.org. (accessed february 2010).

[31] "Apache axis2/java - next generation web services." http://ws.apache.org/axis2/. (Acessed in January/2011).