# Horizon Project

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

# Horizon - A New Horizon for Internet

WP2 - TASK 2.3: Identified Virtualization Solution Modification/Adoption Report and Tools Tests
(Annex F)

# Institutions


| **Brazil** | **France** |
| --- | --- |
| GTA-COPPE/UFRJ | LIP6 Université Pierre et Marie Curie |
| PUC-Rio | Telecom SudParis |
| UNICAMP | Devoteam |
| Netcenter Informática ltda. | Ginkgo Networks |
| | VirtuOR |

# Contents


**1 General Prototype Modifications Description** **4**

1.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.2 General Prototype Description . . . . . . . . . . . . . . . . . . 6
1.2.1 Sensors . . . . . . . . . . . . . . . . . . . . . . . . . 6
1.2.2 Actuators . . . . . . . . . . . . . . . . . . . . . . . . 7
1.3 Document Outline . . . . . . . . . . . . . . . . . . . . . . . . 7

**2 Xen** **8**

2.1 Xen CPU Scheduler . . . . . . . . . . . . . . . . . . . . . . . . 10
2.1.1 Xen Credit Scheduler . . . . . . . . . . . . . . . . . . . 11
2.1.2 Use of Scheduling Parameters in Piloting Plane . . . . 13
2.1.3 CPU and Scheduling Parameters Adjustment . . . . . 14
2.1.4 CPU and Scheduling Parameters Adjustment in Xen using Libvirt . . . . . . . . . . . . . . . . . . . . . . . 15
2.2 Xen Migration . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.2.1 Standard Xen Migration: Qualities and Flaws . . . . . 17
2.2.2 Proposal for Xen Migration: Data Plane and Control Plane Separation . . . . . . . . . . . . . . . . . . . . . 18
2.2.3 Implementation Overview . . . . . . . . . . . . . . . . 19
2.2.4 Implementation Details . . . . . . . . . . . . . . . . . . 21
2.3 Xen Statistics . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.3.1 Measures Gatherer Handler Component . . . . . . . . 24
2.3.2 Measures Gatherer Main Component . . . . . . . . . . 26
2.3.3 Xentop Gatherer Component . . . . . . . . . . . . . . 27
2.3.4 Memory Gatherer Component . . . . . . . . . . . . . . 30
2.3.5 Ifconfig Gatherer Component . . . . . . . . . . . . . . 31
2.3.6 Latency Gatherer Component . . . . . . . . . . . . . . 34
2.4 Xen Topology . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
2.4.1 Methodology . . . . . . . . . . . . . . . . . . . . . . . 36
2.5 New I/O Virtualization Techniques . . . . . . . . . . . . . . . 38
2.5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . 38

2.5.2 Techniques . . . 40
2.5.3 Implementation . . . 40
2.5.4 Installation . . . 41

**3 OpenFlow** **43**
3.1 FlowVisor . . . 45
3.1.1 Introduction . . . 45
3.1.2 Isolation Mechanisms . . . 46
3.2 OpenFlow Migration . . . 47
3.2.1 Introduction . . . 47
3.2.2 Implementation . . . 48
3.3 OpenFlow Statistics . . . 50
3.3.1 Introduction . . . 50
3.3.2 Stats XML Message . . . 51
3.3.3 Information Classes . . . 51
3.4 Openflow Discovery . . . 55
3.5 Openflow Spanning Tree . . . 58

**4 Performance Tests** **62**
4.1 Migration Tests . . . 62
4.1.1 Introduction . . . 62
4.1.2 Results . . . 63
4.1.3 Conclusions . . . 65
4.2 Bandwidth Control Tests . . . 66
4.2.1 Introduction . . . 66
4.2.2 Results . . . 67
4.2.3 Conclusions . . . 68

**5 Conclusions and Ongoing Work** **70**

**Bibliography** **76**

# List of Figures


1.1 Architecture of the prototype. . . . . 5
1.2 Prototype overview. . . . . 7

2.1 Detailed Xen prototype architecture. . . . . 9
2.2 A physical CPU and its queues. . . . . 12
2.3 The self balancing mechanism. . . . . 12
2.4 Cap Adjustment Experiment. . . . . 14
2.5 Example of router migration in Xen when a virtual link is mapped to a multiple-hop path in the physical network. . . . . 20
2.6 Measures Gatherer architecture, along with its components . . 24
2.7 The Topology Consolidate component. . . . . 37
2.8 The Node Consolidate component. . . . . 38
2.9 The Scanning Neighbors component . . . . . 39

3.1 OpenFlow prototype architecture. . . . . 44
3.2 FlowVisor Working . . . . . 45
3.3 FlowVisor hierarchy . . . . . 46
3.4 Flow Migration Schema. . . . . 47
3.5 LLDP frame format for IEEE 802.3. . . . . 55
3.6 Behavior of the original and the modified Discovery algorithm. 57
3.7 An example of topology with a loop. . . . . 59
3.8 Example of a network graph. . . . . 60

4.1 Experimental scenarios for virtual network migration. . . . . 63
4.2 Migration downtime as a function of the data packet rate. . . 64
4.3 Number of lost packets during downtime as a function of the data packet rate. . . . . 64
4.4 Total migration time as a function of the data packet rate. . . 65
4.5 The Test Environment . . . . . 66
4.6 TCP-UDP Test . . . . . 67
4.7 UDP-UDP Test . . . . . 68
4.8 TCP-TCP Test . . . . . 69

# Chapter 1

# General Prototype Modifications Description

## 1.1 Introduction

The Horizon Project fits within the current research community initiative in rethinking the Internet architecture to cope with its current limitations and support new requirements. Many researchers conclude that there is no one-size-fits-all solution for all requirements from users and network-providers and, thus, advocate for pluralist approaches which allow coexistence of different network architectures and gradual migration from the current architecture to its successor. Network virtualization is a key enabling technology for pluralist architectures because it allows multiple virtual networks to run over the same physical infrastructure [1]. Each virtual network can run specific network protocol stacks independently from the others, and hence network virtualization allows the coexistence of multiple specialized networks running at the same time [2].

The Horizon Project proposes a virtualized infrastructure which requires a piloting system to manage and control virtual networks [3]. The piloting system runs intelligent algorithms [4, 5] to automatically *instantiate/delete* and *monitor* virtual networks and also to *migrate* network elements and *set* its resource-allocation parameters [6, 7].

The network virtualization infrastructure provides five primitives (*instantiate*, *delete*, *migrate*, *monitor*, and *set*) for the piloting system to control and manage the virtual network elements. The relationship between the piloting system and the network virtualization infrastructure is shown in Fig. 1.1.

The piloting system needs information to pilot the network. To acquire the required information, the piloting system uses the *monitor* primitive,

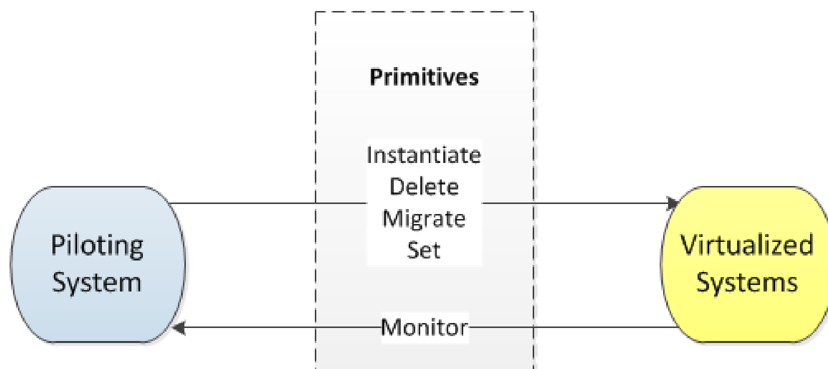



Figure 1.1: Architecture of the prototype. The piloting system uses the *instantiate*, *delete*, *monitor*, *migrate* and *set* primitives from the network virtualization infrastructure.

which calls the monitoring tools required to measure variables of interest, such as available bandwidth, processor and memory usage, link, and end-to-end delay. After monitoring, the piloting system is able to act on the network using the *instantiate*, *delete*, *migrate*, and *set* primitives. Our architecture supports real-time control of virtual network resources, allowing setting resource-allocation parameters of each virtual network element (routers, switches, links, gateways, etc.). The available parameters include low level and specific hardware parameters such as the number of virtual processors given to a virtual router and priority of processor usage in a contention scenario. Thus, the piloting system dynamically adapts the resources allocated to each virtual network according to the current network condition, number of users, priority of each virtual network, Service Level Agreements (SLAs), etc.

This report presents the two virtual network prototypes developed by the Horizon project team [8]. The first prototype is based on Xen [9, 10], a x86 platform virtualization tool, and the second prototype is based on OpenFlow [11], a network virtualization tool. For both prototypes, we present the performed modifications on the virtualization tools to support the piloting system primitives. The modifications are needed because the virtualization tools do not support all the required functionalities [12, 13, 14]. For example, the primitive *monitor*, should inform some measurements about the virtual networks that are not provided by the virtualization tools, such as the Round Trip Time (RTT) to neighbors or the exact virtual router memory usage when running Xen virtualization tool. Another example is the support for live migration with no packet loss, which was another feature we added

to Xen [15].

## 1.2 General Prototype Description

The Horizon network virtualization approach divides a virtual network element into two main planes: the virtualization plane and the piloting plane. The virtualization plane provides the substrate for running logical network elements over a single shared physical network. The piloting plane provides the intelligence for optimizing the network, according to Service Level Agreements (SLAs) and to each virtual network requirements.

Our network virtualization prototype must address the Future Internet requirements [16]. In order to accomplish them, we use two virtualization techniques: router virtualization, which creates logical routers over physical routers; and network virtualization, which virtualizes network flows. Xen hypervisor [9] virtualizes routers and OpenFlow Switches [11] create virtualized networks. Both Xen and OpenFlow are well suited for achieving our architectural goals, therefore, our prototype is divided into two, each one uses a different network virtualization tool: the Xen-based and the OpenFlow-based prototypes. In this sense, both prototypes must support the required primitives for piloting the network. Nevertheless, because they only partially support the required primitives, we modified Xen and OpenFlow by developing applications to provide all the missing features.

The prototypes basic architecture, shown in Fig. 1.2, are independent of the virtualization tool and run over each network. The virtualization system is an abstraction of which virtualization tool is being used, it can be either Xen or OpenFlow. Depending on the virtualization tool adopted, there are specific modifications required on each tool. Network nodes must have their own set of sensors and actuators. The sensors are responsible for collecting information about the element environment and state. The actuators are responsible for actuating on the network elements according to the piloting plane policies. All network nodes are associated to a Controller, which aggregates nodes information and issues commands to each node.

### 1.2.1 Sensors

All nodes should describe the context they belong to as well as they should be able to inform their state. In order to collect information, all nodes implement a set of sensors. Each virtualization technique introduces a different set of sensors. For instance, Xen virtualization tool introduces a set of sensors for monitoring the virtual router state, whereas OpenFlow

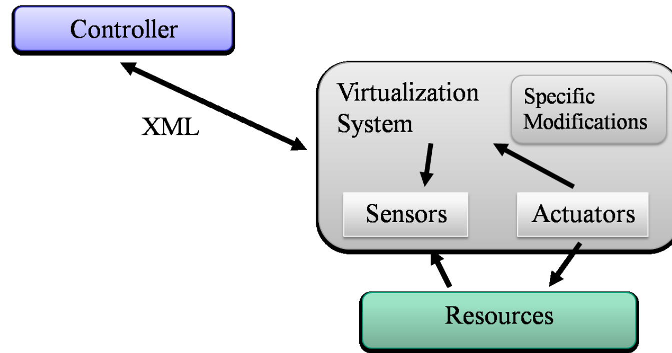



Figure 1.2: Prototype overview.

virtualization tool introduces another set of sensors for monitoring the flows going through a node. There are, however, five composed measures that are common to both virtualization techniques and should be monitored: CPU usage, memory usage, network throughput, dropped packet rate, and network topology.

### 1.2.2 Actuators

All network nodes should provide an interface for the piloting plane network control. The actuators are the software components that apply the piloting plane required actions. Each virtualization system has different means and methods to control the network. Xen provides built-in actuators, such as the creation and destruction of virtual machines, but there are some primitives, such as no packet loss migration, that had to be implemented. OpenFlow also provides some actuators, like flow creation and destruction, but similarly to Xen, it lacks some features that had to be implemented, such as a native flow migration mechanism.

## 1.3 Document Outline

The rest of this report is organized as follows. Chapter 2 presents the Xen virtualization tool modifications and applications developed for our Xen-based prototype. Similarly, Chapter 3 presents the applications developed for our OpenFlow-based prototype. In chapter 4 we present some performance tests of the virtualization tools. Finally, Chapter 5 concludes this work and presents future work directions.

# Chapter 2

# Xen

Xen is an open-source hypervisor proposed to run on commodity hardware platforms, and it uses paravirtualization techniques to improve performance [9]. Xen allows to simultaneously run multiple virtual machines on a single physical machine. Xen architecture is composed of one hypervisor located above the physical hardware and several virtual machines over the hypervisor. Each virtual machine can have its own operating system and applications. The hypervisor controls the access to the hardware and also manages the available resources shared by virtual machines. In addition, device drivers are kept in an isolated virtual machine, called Domain 0, in order to provide reliable and efficient hardware support [17]. Because it has total access to the hardware of the physical machine, Domain 0 has special privileges compared with other virtual machines, referred to as user domains (Domains U).

This chapter presents our Xen-based prototype of a virtual network environment. The Xen-based prototype is composed of three main modules: the Graphical User Interface (GUI), the controller, and the network substrate. The GUI presents prototype information for user monitoring and also a simple control interface for user interaction. The controller, a special node that consolidates network data, acts on the physical and virtual networks issuing commands and interfaces with the piloting plane. Finally, the network substrate is mainly composed of physical and virtual routers in which several modules were developed for coping with our architecture requirements [18]. Fig. 2.1 details the prototype modules and their interfaces.

The User Client Node shown in Fig. 2.1 represents the user computer for manual monitoring and control of the prototype. The User Client Node contains a GUI that shows both physical and virtual network topologies and additionally allows fine grain monitoring by exposing detailed information upon a physical or virtual router selection. The Client node also allows send-

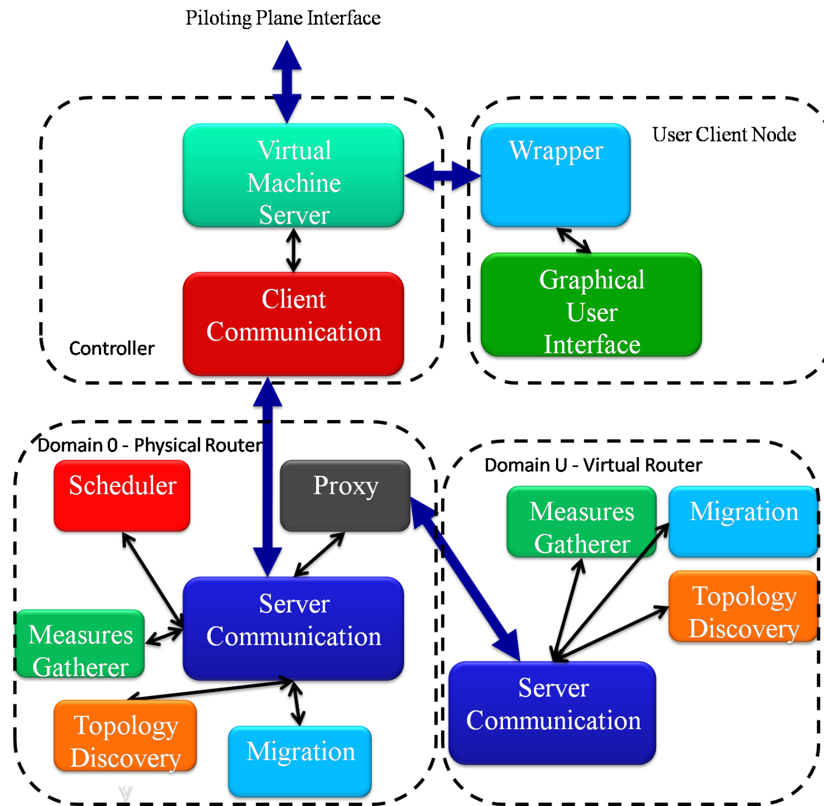



Figure 2.1: Detailed Xen prototype architecture. The modules that compose the prototype parts are shown as well as their interfaces and interactions.

ing commands to the prototype, such as issuing a virtual router migration from a physical router to another one. In order to interact with the controller node, the GUI interfaces with the Wrapper module, which is responsible for converting the GUI issued commands into Simple Object Access Protocol (SOAP) calls the controller node and converts the SOAP responses into its expected response pattern [19].

The controller node is composed of the Virtual Machine Server module and the Client Communication module. The Virtual Machine Server is the core module of the controller. It has a SOAP interface for interacting with the piloting plane and User Client Node. It is the module responsible to consolidate all prototype information, and to execute all the control and maintenance algorithms. The Virtual Machine Server module extensively uses the `libvirt` library to accomplish its tasks using native virtualization tool support and also to accomplish tasks that demand non-native enabled

functionalities. The Virtual Machine Server module uses special modules to run inside the physical and the virtual routers. The Virtual Machine Server uses the Client Communication module in order to interface with these special modules.

The physical router nodes provide the physical infrastructure used by the virtual networks. There is a Server Communication module within each physical router. This module receives requests from the controller node Client Communication module and forwards the request to the appropriate module. If a request is addressed to a virtual router, it is forwarded to the correct virtual router, passing through the Proxy module. Otherwise, it is forwarded to one of the special modules of the physical router. If the received request regards a monitoring task, it is forwarded to the Measures Gatherer module, with the purpose of obtaining measures from the physical and virtual routers. If the received request is to act on the resources division among the virtual routers, the Scheduler module handles the request. Requests of discovering physical or virtual networks topologies are handled by the Topology Discovery module. Migration requests are handled by our Migration module, which provides virtual router migration without packet losses.

In the following sections we describe Xen modifications and our Xen-based prototype modules. Section 2.1 presents the Xen Credit Scheduler. We show the effect of `weight` and `cap` on the CPU assignment and a proposal of using them as a control mechanism. Section 2.2 explains our migration feature that allows reconfigure the network topologies while the routers are running and without losing packets. In Section 2.3, we present our resource monitoring module. Section 2.4 describes the module responsible for discovering the physical and virtual network topologies. Finally, Section 2.5 presents the new virtualization hardware support technologies. These technologies are available to enhance I/O virtualization performance and their effect in virtual network environment.

## 2.1 Xen CPU Scheduler

Xen virtualizes the processor (Central Processing Unit - CPU) by assigning virtual CPUs (vCPUs) to virtual machines. Virtual CPUs correspond to the CPUs that the running processes within the virtual machine can see. The hypervisor maps vCPUs to physical CPUs. Xen hypervisor implements a CPU scheduler that dynamically maps a physical CPU to each vCPU under a certain period, based on a scheduling algorithm. Xen implements two scheduler algorithms, the Simple Earliest Deadline First (SEDF) scheduler and the Credit Scheduler [20].

The SEDF scheduler was implemented on Xen hypervisor first and now it is a legacy scheduler. The main idea is to pick the vCPU that is closest to the deadline. Each domain has time parameters, a time slice, that determines the continuous time that the domain must run, and a period of time, in which this domain must run its time slice. The scheduled domain is the one which is closest to run out of its period without running its time slice.

The Credit scheduler is the default Xen scheduler. This scheduler makes a proportional CPU share. This means that Credit scheduler allocates CPU resources to each virtual machine (or, more specifically, to each vCPU) according to weights assigned to virtual machines. Credit scheduler can also be work conserving on SMP (Symmetric Multi-Processing) hosts. This means that the scheduler permits the physical CPUs to run at 100% if any virtual machine has work to do. In a work-conserving scheduler there is no limit on the amount of CPU resources that a virtual machine can use.

### 2.1.1 Xen Credit Scheduler

Xen Credit Scheduler is the default scheduler on recent Xen Hypervisor versions. The basic idea is to distribute credits among the domains and, when a domain wants to run, the domain must pay for the running time. In order to share the CPU utilization, the scheduler distributes the credits among domains, including the Domain 0, based on two parameters associated with each domain, `weight` and `cap`. When a domain is active executing, it consumes credits. Based on credit accounting, the scheduling determines which CPU is able to execute. The scheduler, however, only takes into account if the domains have spent or not their credits, and does not consider the absolute value of current credit of the domain.

In the basic Xen Credit Scheduler implementation, each physical CPU has two queues. One is for the vCPUs that still have credits to spend, the UNDER queue, and another, to the vCPUs that have over spent their credits, the OVER queue. Each queue works in a round robin manner. The scheduler picks one vCPU from the UNDER queue, and let it run for a given period. After that, it picks another vCPU. Each vCPU is allowed to run up to 30 ms, without the scheduler interruption, as long as the vCPU has sufficient credits to do it. This amount of time is called *quantum*.

The scheduler ticks every 10 ms. Every 10 ms, credits are debited from each domain that has a vCPU running. When a vCPU over spend its credits, it goes to the OVER queue. If a physical CPU has no vCPUs in the UNDER queue, it tries do pull a vCPU from other physical CPUs queue [20]. A vCPU in the OVER queue can only run, if all vCPU in the UNDER queue are blocked. This mechanism allows a vCPU to use more than its share of

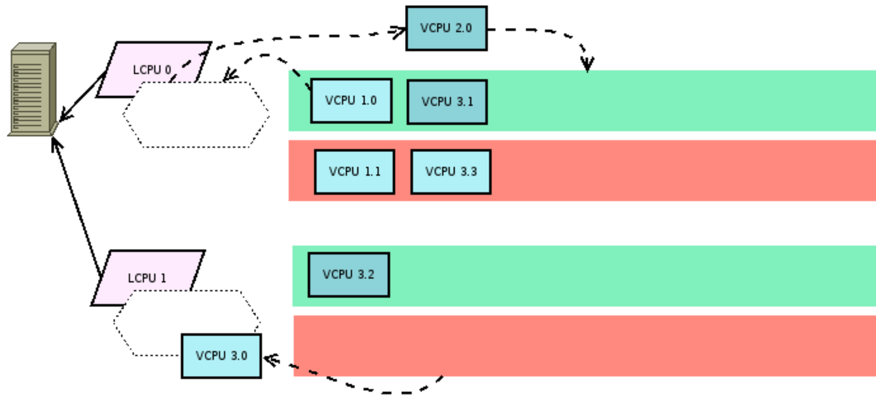



Figure 2.2: A physical CPU and its queues: the UNDER, green, and the OVER, red. The physical CPU picks an OVER vCPU, because there are only blocked vCPUs in the UNDER queue, dark blue.

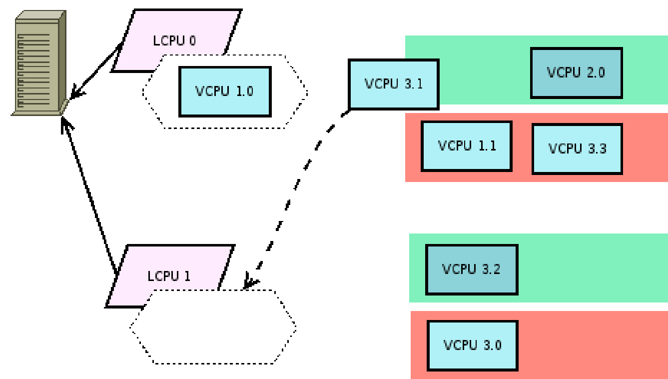



Figure 2.3: The self balancing mechanism. The physical CPU, which is able to run vCPUs, picks one vCPU from the queue of the other physical CPU. The vCPUs, which have credits, always have the priority.

the resources, when a physical CPU is idle.

#### 2.1.1.1 Scheduling Parameters: Weight and Cap

Xen Credit Scheduler has two parameters that influence in credit distribution, `weight` and `cap`. Those parameters can be set while a domain is running. The `weight` parameter determines the proportional share of the physical CPU time that each domain gets. The `weights` of the domains are relative to each other. For example, in an environment with two domains,

in which domain A has a `weight` of 256 and domain B has a `weight` of 128, domain A can get twice more CPU resources than domain B. As the `weight` is relative, the same effect occurs, in an environment that domain A has a `weight` of 512 and domain B has a `weight` of 256. If a domain, however, with high `weight` does not need all the allocated CPU resources, domains with lower `weights` can get the unused resources. The `cap`, in its turn, is an absolute value that limits the amount of physical CPU resource that a domain is able to use. It forces the Credit Scheduler to work in a non-work-conserving mode. The `cap` is expressed in percentage of one physical CPU, e.g. 50 is half of a CPU, 100 is an entire CPU and 400 is 4 CPUs. The `cap` of each domain is set by default to zero. A 0 `cap` value means that the domain CPU utilization is unrestricted [20].

### 2.1.2 Use of Scheduling Parameters in Piloting Plane

Xen Credit Scheduler parameters may be used in the Piloting Plane to control the amount of CPU resources that each virtual router is allowed to run. Taking into account these parameters, we offer different performance levels [21, 22]. To illustrate the use of Xen Credit Scheduler parameters, we conduct an experiment with `cap` adjustment [23]. Our goal is to show the influence of this parameter on the forwarded packet rate of virtual routers sharing CPU resources. We use `cap` instead of `weight`, because `cap` gives more control of the CPU usage to the system administrator. This is a desirable feature, because the Piloting Plane must have the control of CPU usage in order to ensure performance levels. The experiment takes place in a testbed, in which traffic generators (TGs) send packets to traffic receivers (TRs) passing through a traffic forwarder (TF).

The experiment consists on sending packets from a TG to a TR at a fixed rate of 150 kilopackets per second. Virtual routers instantiated over the Xen platform, in TF, forward the packets. TF has two virtual routers that share the same CPU core, and each one has its own network interface pair. Thus, we ensure that the main resource shared by the virtual routers is the CPU. Domain 0 has its own CPU core. We have two virtual networks running in parallel in TF. We assign a fixed `cap` of 100 to a virtual machine (VM2) and we vary the `cap` of the other virtual machine (VM1). Results, shown on Fig. 2.4, evidence that maximum packet rate obtained is less than the total sent. This happens because Domain-0 must forward packets from and to the virtual routers. Furthermore, the packet forwarding rate of VM1 decreases with its `cap` reduction, whereas the packet forwarding rate of VM2 increases. This happens because both virtual routers share the same core and reducing the CPU given to VM1 frees VM2 to use the CPU.

Thus, we show that we assure a maximum packet forwarding rate to each virtual router, adjusting its `cap`. This strategy helps the Piloting Plane controlling the maximum throughput. Limiting the throughput of individual virtual routers is a strategy to ensure a performance level to each virtual on the network. This performance level is based on QoS (Quality of Service), parameters specified in SLAs.

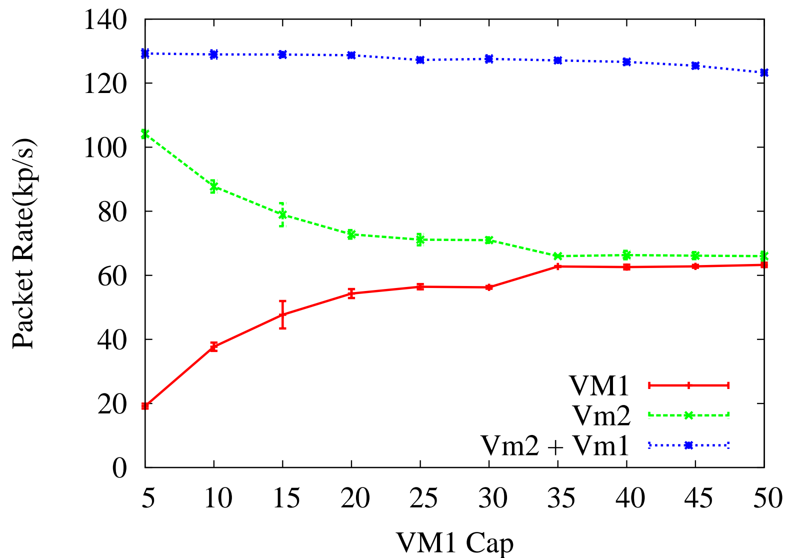



Figure 2.4: Cap Adjustment Experiment.

### 2.1.3 CPU and Scheduling Parameters Adjustment

In this subsection we provide Xen commands to configure physical CPU allocation and scheduling parameters. All commands specified below are executed while a domain is running. Those commands are executed locally.

#### 2.1.3.1 `xm sched-credit` command

This command sets the `weight` or the `cap` of a certain domain. The general format of this command is as follows:

```
$ xm sched-credit [-d <DOMAIN> [-w WEIGHT] [-c CAP]]
```

To adjust the `weight` of a specific domain we use:

```
$ xm sched-credit -d <DOMAIN> -w <WEIGHT>
```

To adjust the `cap` of a specific domain we use:

```
$ xm sched-credit -d <DOMAIN> -c <CAP>
```

In both cases above, the domain is specified by its ID or its name.

To list the current scheduler settings we use:

```
$ xm sched-credit
```

#### 2.1.3.2 `xm vcpu-set` command

This command sets the number of VCPUs of a domain. This command, however, only sets a number equal or less than the number of VCPUs specified in domains configuration file. The format of this command is the following:

```
$ xm vcpu-set <DOMAIN> <#VCPUs>
```

#### 2.1.3.3 `xm vcpu-pin` command

This command assigns a vCPU to one or more physical CPUs. If this command is used, the scheduler cannot assign this vCPU to another physical CPU that is not specified. The format of this command is the following:

```
$ xm vcpu-pin <DOMAIN> <VCPU|all> <PCPUs|all>
```

In this command a vCPU of a certain domain is chosen to be pinned in one or more physical CPU denoted by pCPUs. In order to give the scheduler the independence to assign a vCPU to any physical CPU, we use `all` as the pCPU argument. Nevertheless, this is the default Xen vCPU assignment.

#### 2.1.3.4 `xm vcpu-list` command

This command lists the current CPU assignment. It shows whether a vCPU is pinned in one or more physical CPUs and the current physical CPUs used by the running vCPUs. This command is used as follows:

```
$ xm vcpu-list
```

### 2.1.4 CPU and Scheduling Parameters Adjustment in Xen using Libvirt

Libvirt [24] is an API (Application Programming Interface) used to interact with different virtualization solutions. Using Libvirt, we manage the virtual machine locally or remotely. Libvirt is used with different programming languages, such as python, java or C. The remote management of virtual routers plays an important role in the Piloting Plane of Horizon project. Hence, we develop a python module to simplify the usage of Libvirt. This module, called VirtualManagement, aggregates Libvirt classes and encapsulates them. Currently, the VirtualManagement module has one class called

PhysicalHost. An object of this class represents a physical Xen host. This object is instantiated in any machine that has the Libvirt client and a network connection with the physical host. At the moment of the object instantiation, the connection between the client and Xen host is established.

In the class PhysicalHost we define methods, according to Piloting Plane programming requirements. The methods listed above are to set scheduler parameters. These methods functionality equivalent to the commands specified in Section 2.1.3.1. The methods implement on PhysicalHost class, however, may be called remotely, while the Xen native commands are called locally.

#### 2.1.4.1 `PhysicalHost::setSchedParametersWeight(weight,domainID)`

Sets the `weight` of a domain specified by "domainID".

#### 2.1.4.2 `PhysicalHost::setSchedParametersCap(cap,domainID)`

Sets the `cap` of a domain specified by "domainID".

#### 2.1.4.3 `PhysicalHost::printSchedParameters()`

Prints, in console, the information of current scheduler settings.

## 2.2 Xen Migration

Experimenting new alternatives in the Internet core using production traffic is considered a hard task, as Internet providers do not feel comfortable to perform modifications that could damage their service. This is a problem because, on one hand, Internet must evolve to handle new demands, but, on the other hand, new mechanisms that change the Internet cannot be applied in the underlying infrastructure. Many works tackle this problem by using virtualization techniques [1, 25]. In a virtualized environment, the physical substrate is shared among many virtual networks which are isolated and, consequently, new proposals can run together with the current Internet without disturbing the production traffic.

Network virtualization requires new control and management primitives for redistributing physical resources among the virtual nodes. One of these primitives is the live virtual network migration. The idea is to move virtual networks among the available physical resources without disrupting the packet forwarding and network control services [26]. In Horizon project

scope, live migration allows the dynamic reconfiguration of the network topologies without shutting down the running routers.

Virtual network migration allows dynamic planning of physical resources and traffic management on demand. Network migration can also be applied to create green networks [27], because virtual networks could be placed on different physical routers according to the traffic demand to reduce energy consumption. This concept is also compatible with the idea of cloud computing, which is an attempt to efficiently share power, storage devices, user applications, and network infrastructure over the Internet as if they were services [28].

We analyze the standard Xen migration scheme and we also propose a new migration model that fixes the issues that were found on the standard mechanism. The migration proposal is meaningful to the project because it gives the piloting plane the capacity to dynamically rearrange the logical network topology, without disrupting running services and with no packet loss. After the formal definition of these technologies, we present a detailed documentation of the prototype with its overview and its functionalities.

### 2.2.1 Standard Xen Migration: Qualities and Flaws

When we use Xen to create virtual networks, we assume that each VM works as a virtual router. Hence, migrate a VM is equivalent to migrate a router. Because the VM is running a live service, we need to reduce the migration downtime, which is the time that the virtual machine is unavailable during the migration. It is also important to minimize the total migration time to guarantee that we can quickly free the resources of the initial physical machine. Xen has a built-in mechanism to migrate virtual machines [29]. This mechanism is based on some assumptions: the migration occurs inside a local network and the machine disk is shared over the network[1]. The main idea of this procedure is that migrating a virtual machine is the same of copying the memory of the virtual machine to the new physical location and reconfiguring the network links without breaking connections.

The simplest way to migrate the VM memory is to suspend the VM, transfer all the memory pages to the new physical location, and then resume the VM. To reduce the downtime, this procedure is evolved to a pre-copy migration, in which the memory copy is accomplished through two phases. The

[1]This shared disk assumption can be relaxed in our scenario, because routers from the same vendor usually implement the same small set of applications [26]. Then, we assume that the new physical router also has this set of programs and is able to load them onto the file system of the new VM. Hence, only virtual router memory and configuration files must be migrated.

first phase, called iterative pre-copy, transfers all memory pages to the new physical machine, except for the "hot pages", which are frequently modified pages. Consequently, the downtime is reduced, because only a few pages, instead of the whole memory, are transmitted while the VM is down. Hence, in the first round, all the memory pages are transferred from source to destination with a minimum rate specified by the network administrator. Then, on the other rounds, only the memory pages that were dirtied by the operating system will be transferred. The transfer rate is updated at each round according to an adaptive mechanism based on the "dirtying rate". In each round, the dirtying rate is calculated as the ratio of number of dirtied pages on the last round and the duration of the last round. The maximum rate of the next round is then obtained by adding a constant increment of 50Mb/s to the calculated dirtying rate. The pre-copy ends if the maximum rate of the round is equal to the maximum rate specified by the administrator or less then 256kB of dirtied pages remains to be transferred. The next phase is called stop-and-copy. In this phase the VM is suspended and the hot-pages are transferred with the maximum transfer rate. Then, the new physical node confirms the reception of the whole memory to the old physical node. The Xen built-in migration is inadequate for virtual networks due to the high packet loss rate during the VM downtime. Other problem of Xen built-in migration for virtual routers is that it assumes a migration within a local area network, which does not fulfill our objectives of migrating routers. Indeed, we cannot assume that physical nodes always belong to the same local network.

### 2.2.2 Proposal for Xen Migration: Data Plane and Control Plane Separation

The Horizon Xen-based Live Migration aims at obtaining a virtual router that is able to migrate to different physical network elements with no packet loss. To obviate the packet loss during live migration, we propose a plane separation in Xen [30, 15]. The plane separation is a technique for dividing each network node into two parts: a control plane, responsible for running all the control algorithms and for constructing the routing table; and the data plane, which is responsible for forwarding the traffic. A similar approach was proposed for OpenVZ, a virtualization platform that provides multiple virtual user spaces over the same operating system [26]. Xen, however, presents a more programmable virtualization platform, because each virtual router can have its own software set, including operating system and protocol stack.

We developed a prototype that maintains in the VM the control plane,

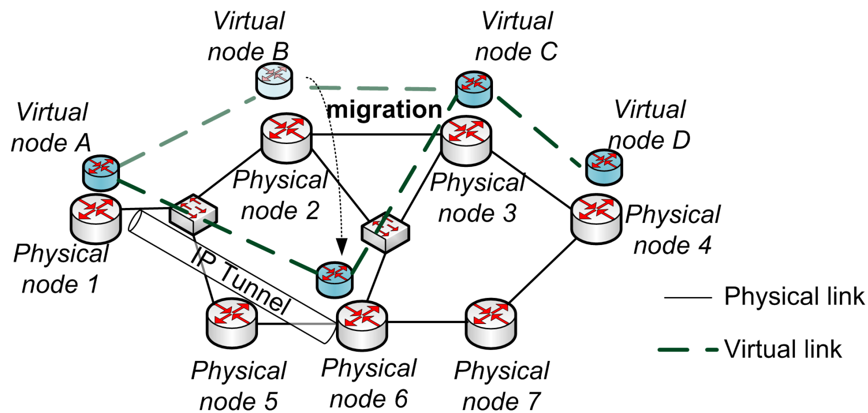
while the data plane is implemented in Domain 0. Each virtual router has its own forwarding table in Domain 0 and each table is a copy of the original forwarding table created by routing software running in the VM. When Domain 0 receives a control message, it checks which network the message belongs to and forwards the message to the corresponding VM. When Domain receives a data message, it is forwarded by Domain 0 using the forwarding table that corresponds to that virtual network.

The proposed migration mechanism works as follows. First, the Xen standard migration is started to migrate the VM. After the iterative precopy phase, the VM is paused and the remaining dirty memory pages are transferred. During this time, the data path is still working at Domain 0, with no interruptions or packet losses. Also, a daemon is started in Domain 0 to buffer the control packets for the VM that is being migrated. When the whole memory is copied, the VM is resumed on the new physical machine (PM) and the network connections are created in the new Domain 0 using a dynamic interface binding module, which is responsible for mapping the virtual network interfaces to the physical network interfaces of the new PM. After that, a tunnel from the old PM to the new PM is created in order to transfer the control packets that were buffered in the old Domain 0 and also the new control packets. Finally, the ARP reply is broadcast to update the links and the data path in the old PM is removed.

This migration mechanism guarantees no packet loss in the data plane during the VM migration, which is an important characteristic for a virtual router. Moreover, there is also no control packet loss. The mechanism inserts only a delay in the control packet delivery. The proposed mechanism, however, is based on Xen default migration, which means that it still needs that routers are within the same local area network. In addition, the mapping of a virtual link over multiple physical links is still an open issue that depends on solutions such as IP tunnels or instantiating new virtual routers on the network. For instance, in Fig. 2.5, we migrate virtual node B from physical node 2 to physical node 6. Physical node 6, however, is not a one-hop neighbor of physical node 1. Consequently, to complete the link migration, we need to create a tunnel from physical node 6 to physical node 1 to simulate a one-hop neighborhood. The other solution is to instantiate a new virtual router to substitute the tunnel. This solution, however, modifies the virtual topology and influences the routing protocol operation.

### 2.2.3 Implementation Overview

As seen in section 2.2.2, our proposal is currently implemented as a prototype built on Xen with plane separation approach. From now on, the proposal




1) Migrate control plane from physical node 2 to 6 maintaining data and control planes.
2) Do interface dynamic binding and create tunnel between physical nodes 1 and 6.
3) Create a new forwarding table in Domain 0 of physical node 6
4) Reconfigure links with ARP reply
5) Delete data plane on physical node 2.

Figure 2.5: Example of router migration in Xen when a virtual link is mapped to a multiple-hop path in the physical network.

will be referred as the Horizon Xen-based Live Migration functionality.

It is necessary to create the separated control and data planes, in order to implement the Xen-based Live Migration functionality. The Domain 0 should know all VMs that are in the same physical machine, with the purpose of sending control messages to the correct VM. Furthermore, control and data planes should interact in order to maintain the forwarding base consistent. Therefore, we create two components: the Hello component and the Route Change component. There is also another component that actually migrate the virtual machine from one physical machine to other, the Migration Advise component. This component runs on both source and destination Domains 0 and it is responsible for the migration of information communication.

When the virtual router is instantiated, a `daemon` is started within the virtual router. This daemon creates a connection to a daemon running in Domain 0. The first step of this communication aims at activating the separated plane mechanism. The first component is the Hello component, which informs to Domain 0 about the virtual router, sending hostname and network interfaces information. The Domain 0, with this information, creates a route table for this virtual router and the rules for using this table. From this moment on, all the packets of this router are forwarded by Domain 0 using the virtual router specific route table. After that, control plane monitoring starts. When a route is changed, this change are also executed in Domain 0.

The Route Change component monitors the route modifications.

The router keeps forwarding packets using data plane in this Domain 0 until the network manager decides to migrate it to another physical machine. The standard Xen migration is our prototype first step. Using this mechanism, we migrate the control plane, but we keep the data plane in its current physical machine, also called source Domain 0. After that, the forwarding environment, the route table and the rules, of this virtual router is created on the destination physical machine, also called destination Domain 0. The new route table is populated with the control plane routes in the migrated virtual machine. At this moment, we have the source Domain 0 forwarding the packets and the destination Domain 0 ready to forward. We start migration of links.

In order to migrate the links, we use the ARP mechanism, which forces our prototype to map one logical hop into two or more physical hops. Because of that, our migration takes place between machines that has connectivity with all the logical neighbors. The ARP reply message is sent for notifying the MAC address of the interface with the given IP address. Hence, the destination physical machine sends the ARP reply message for each interface used by this virtual router, notifying the neighbors that the router is currently in another location. Thereafter, all the links are migrated and the data plane, that is running over the source Domain 0, is dropped. The Migration Advise module accomplishes the migrating process and calls the Hello and Route Change modules following its demands.

These are the main steps of the Horizon Xen-based Live Migration functionality implemented. The Section 2.2.4 gives more implementation details of each module used in the prototype.

### 2.2.4 Implementation Details

In this section, we describe the main objectives and the steps of each message in our services. There are activities executed before sending the request message in the client side and after receiving the request message in the server side. In most cases, the response message is an application acknowledgement message, returning to client whether the task invoked was done correctly. Because of that, we do not talk about the activities done involving the reception of response messages.

#### 2.2.4.1 Hello Component

The Hello message informs the Domain 0 about the existence of the virtual machine which has sent the message. The message contains the virtual

machine name and information about the virtual machine network interfaces.

The Hello message is a request that the virtual machine makes to Domain 0. The virtual machine must prepare the request message and, then, Domain 0 process this information in order to create the environment that allows the virtual machine to forward packets through Domain 0. The procedure of the Hello component is represented in Table 2.1.

| **Client** | **Server** |
|---|---|
| 1. User starts the separated router mechanism inside virtual machine | 1. Save node information |
| 2. Get node hostname and information about interfaces | 2. Create virtual machine route table |
| 3. Send Hello Request Message to Domain 0 | 3. Create rules to use this table |
| | 4. Add routes to reach virtual machine |
| | 5. Bind the virtual interfaces to physical interfaces |
| | 6. Send Hello Response to virtual machine |

Table 2.1: Hello module algorithm.

#### 2.2.4.2 Route Change Component

The Route Change message aims at informing Domain 0 that the control plane has detected a change in one route. Domain 0 is able to change the updated route in the virtual machine forwarding table. A route is defined by the network and the mask of the destination field of the IP protocol and has some parameters such as next hop, metric, and output interface.

The control plane monitors three kinds of modification: adding, altering and removing. The adding modification consists in a new entry to the routing and forwarding table. The altering modification represents that any parameter of a route has been changed when the route was updated. Finally, the removing modification detects the routes that had disappeared from the routing table and must then be removed from the forwarding table.

After Domain 0 finishes the modifications in forwarding table, it must notify the virtual machine that the process was successful. If the process fails, the virtual machine must resend the route change information to Domain 0. The procedure of the Route Update component is represented in the Table 2.2.

| Client | Server |
| --- | --- |
| 1. Monitor FIB changes | 1. Discover the virtual machine that sent the route change |
| 2. Triggered by route changes | 2. Get the virtual machine route table |
| 3. Discover the change type (Add, Alter, Remove) | 3. Change the route in the virtual machine route table |
| 4. Send Route Change Request to Domain 0 | 4. Send Route Change Response to virtual machine |

Table 2.2: Route Change module algorithm.

#### 2.2.4.3 Migration Advise Component

The Migration Advise component, which runs in the migration source Domain 0, interacts with the destination Domain 0. The main objective of this component is to prepare the destination Domain 0 to forward the packets of the migrated virtual machine. When the control plane migration process ends, the module recreates the data plane. This procedure is represented in Table 2.3.

| Client | Server |
| --- | --- |
| 1. User starts the migration process | 1. Forward the message to VM using Communication Channel |
| 2. Get the destination Dom 0 and VM | 2. VM resend Hello Message Request to Domain 0 |
| 3. Get VM communication channel IP | 3. VM re-send the routes to Domain 0 |
| 4. Send Migration Advise Request to destination Domain 0 | 4. VM notify the Domain 0 that the all the messages were sent |
| | 5. Send Message Advise Response to source Domain 0 |

Table 2.3: Migration Advise module algorithm.

## 2.3 Xen Statistics

The Measures Gatherer is an important module of the Horizon prototype. The module retrieves information about resource allocation and resource usage of Domains 0 and Domains U. With this information, the prototype can

be aware of the resource allocation status of the entire Xen Network. The services provided by the Measures Gatherer can be accessed through XML message requests. The response is also a XML message.

The Measures Gatherer module is composed of several components that are specialized in gathering information from different measurement tools. The current version of the module contemplates the Xentop Gatherer, Ifconfig Gatherer, Memory Gatherer and Latency Gatherer Measures components. The Measures Gatherer subsystem has also two special components, the Measures Gatherer Handler, which is responsible for allowing the communication with the other modules of the Xen prototype, and the Measures Gatherer main component, which is responsible for calling specific measurement modules and consolidating the output to fulfill the requests.

The Measures Gatherer module architecture is shown in Fig. 2.6.

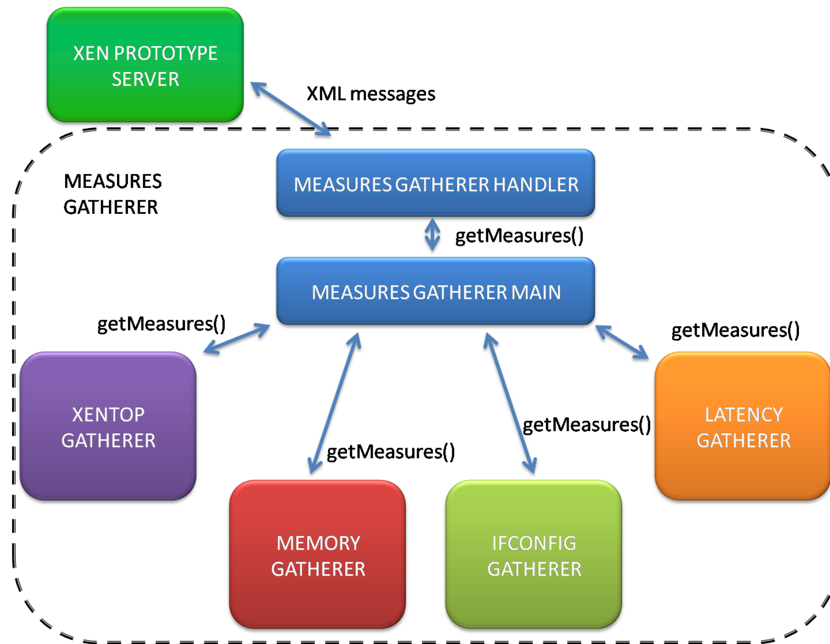



Figure 2.6: Measures Gatherer architecture, along with its components

### 2.3.1 Measures Gatherer Handler Component

The Measures Gatherer Handler component is responsible for interacting with the basic communication module of the prototype. The component receives and answers requests in a XML message exchange fashion. The component has a built-in XML parser, which decodes the requests and passes

it to the Measures Gatherer Main component. The requests must be fulfilled through the `getMeasures()` directive. Upon the receiving of the directive, the Measures Gatherer Main Component retrieves the needed information and returns it as a XML message.

The XML request received by the Measures Gatherer Handler component is in the pattern of the Listing 2.1:

Listing 2.1: XML received by the Measures Gatherer Handler module.

```
<XenPrototype>
    <NodePK>NODE PUBLIC KEY</NodePK>
    <Application>
        <ApplicationId>4</ApplicationId>
        <ApplicationRequest>
            <Measures>
                <UsedGatherers>
                    <ifconfigGatherer>1</ifconfigGatherer>
                    <xentopGatherer>0</xentopGatherer>
                    <latencyGatherer>1</latencyGatherer>
                    <memGatherer>1</memGatherer>
                </UsedGatherers>
                <GathererParameters>
                    <ifconfigGatherer>
                        <iterations>4</iterations>
                    </ifconfigGatherer>
                </GathererParameters>
            </Measures>
        </ApplicationRequest>
    </Application>
</XenPrototype>
```

Meaning of the tags:

- `<XenPrototype>`: Indicates the beginning of a Xen Prototype message
- `<NodePK>`:Contains the public key of the node that generated the message
- `<Application>`:Indicates that the following content is directed to or originated from an application
- `<ApplicationId>`:Identification of the application, which the message is directed to or originated from
- `<ApplicationRequest>`:Indicates that the following content is an application request

- `<Measures>`:Indicates that the following content specifies how the measures should be done
- `<UsedGatherers>`:Indicates that the following content will specify which gatherers to use
- `<ifconfigGatherer>`:Indicates whether the Ifconfig Gatherer should be used. 1 indicates it should be used, 0 indicates it should not.
- `<xentopGatherer>`:Indicates whether the Xentop Gatherer should be used. 1 indicates it should be used, 0 indicates it should not.
- `<latencyGatherer>`:Indicates whether the Latency Gatherer should be used. 1 indicates it should be used, 0 indicates it should not.
- `<memGatherer>`:Indicates whether the Latency Gatherer should be used. 1 indicates it should be used, 0 indicates it should not.
- `<GathererParameters>`:Indicates that the following content will specify gatherer parameters
- `<ifconfigGatherer>`:Indicates that the following content will specify `ifconfig` gatherer parameters
- `<iterations>`:Indicates the number of iterations `ifconfig` should use

In the previous XML example all the gatherers must be specified within the `UsedGatherers` scope in order to know which gatherers should be used. The `GathererParameters` scope should only contain parameters tags for the parameters that should not be used with their default values.

### 2.3.2 Measures Gatherer Main Component

The Measures Gatherer Main component is responsible for accessing the required gatherer modules to collect the desired measures and consolidate their measures in a single XML response. The Measures Gatherer Main component is accessed by the Measures Gatherer Handler whenever a new request arrives. The XML generated by the Measures Gatherer Main component is in the pattern of the Listing 2.2:

Meaning of the new tags:

- `<MeasureToolResponse>`:Indicates that the following data contains the measures of a gatherer

Listing 2.2: XML generated by the Measures Gatherer Main module.


```
<XenPrototype>
    <NodePK>NODE PUBLIC KEY</NodePK>
    <Application>
        <ApplicationId>1</ApplicationId>
        <ApplicationResponse>
            <MeasureToolResponse>
                <MeasureToolId>1</MeasureToolId>
                MEASURE TOOL MEASURES DATA
            </MeasureToolResponse>
            <MeasureToolResponse>
               MEASURE TOOL DATA
            </MeasureToolResponse>
        </ApplicationResponse>
    </Application>
</XenPrototype>
```


- `<MeasureToolId>`:Identification of the gatherer from which the following measures belong

The other tags inside the XML are specific for each gatherer and will be specified in sections describing each specific gatherer.

### 2.3.3 Xentop Gatherer Component

The Xentop Gatherer component is responsible for gathering information acquired from the `xentop` measurement tool. The Xentop Gatherer module executes only in Domain 0 and provides non-invasive information from all the domains running inside the physical machine.

The Xentop Gatherer component receives the parameters of how it should collect the measures and provides a method called `getMeasures()` to retrieve the acquired measures in XML format. The measures are sent to the Measures Gatherer Main module.

The Xentop Gatherer component allows us to decide the number of iterations of the Xentop tool and the sampling interval during these iterations.

The XML message generated by the Xentop Gatherer module the pattern described in Listing 2.3:

Meaning of the new tags:

- `<Iteration>`:Indicates the beginning of the description of a measurement round

- `<IterationNum>`:Identifies the number of the iteration

Listing 2.3: XML part generated by the Xentop Gatherer module.

```xml
<MeasureToolResponse>
    <MeasureToolId>1</MeasureToolId>
    <Iteration>
        <IterationNum>1</IterationNum>
        <IterationDateTime>1263385100.37</IterationDateTime>
        <Domain>
            <DomainName>Domain-0</DomainName>
            <DomainCPUTime>2522</DomainCPUTime>
            <DomainCPUPercentUsage>0.0</DomainCPUPercentUsage>
            <DomainMemory>2824192</DomainMemory>
            <DomainMemoryPercentUsage>90.3</
                DomainMemoryPercentUsage>
            <DomainVCPUCount>4</DomainVCPUCount>
            <DomainNetInterfacesCount>0</
                DomainNetInterfacesCount>
            <NetworkInterface>
                <NetworkInterfaceNumber>0</
                    NetworkInterfaceNumber>
                <ReceivedBytes>34062339</ReceivedBytes>
                <ReceivedPackets>267213</ReceivedPackets>
                <ReceptionErrors>0</ReceptionErrors>
                <ReceptionDrops>19</ReceptionDrops>
                <TransmittedBytes>28</TransmittedBytes>
                <TransmittedPackets>1</TransmittedPackets>
                <TransmissionErrors>0</TransmissionErrors>
                <TransmissionDrops>0</TransmissionDrops>
            </NetworkInterface>
            <NetworkInterface>
                NETWORK INTERFACE DATA
            </NetworkInterface>
        </Domain>
        <Domain>
            DOMAIN DATA
        </Domain>
    </Iteration>
    <Iteration>
        ITERATION DATA
    </Iteration>
</MeasureToolResponse>
```

- `<IterationDateTime>`:Contains date and time of the moment the measures were acquired

- `<Domain>`:Indicates the beginning of the description of a domain measurement

- `<DomainName>`:Contains the name of the domain being measured
- `<DomainCPUTime>`:Contains CPU time in seconds spent by the domain
- `<DomainCPUPercentUsage>`:Contains CPU usage in percentage
- `<DomainMemory>`:Contains the total amount of memory allocated to the domain in kbytes
- `<DomainMemoryPercentUsage>`: Contains the total amount of memory allocated to the domain in percentage
- `<DomainVCPUCount>`:Contains the number of virtual CPUs a domain has
- `<DomainNetInterfacesCount>`:Contains the number of virtual network interfaces a domain has
- `<NetworkInterface>`:Indicates the beginning of the description of a network interface
- `<NetworkInterfaceNumber>`:Number of the virtual network interface
- `<ReceivedBytes>`:Total amount of bytes received by the network interface
- `<ReceivedPackets>`:Total amount of packets received by the network interface
- `<ReceptionErrors>`:Total amount of packet reception errors in the network interface
- `<ReceptionDrops>`:Total amount of packets dropped by the network interface during reception
- `<TransmittedBytes>`:Total amount of bytes transmitted by the network interface
- `<TransmittedPackets>`:Total amount if packets transmitted by the network interface
- `<TransmissionErrors>`:Total amount of packet transmission errors detected in the network interface
- `<TransmissionDrops>`:Total amount of packets dropped by the network interface during transmission

### 2.3.4 Memory Gatherer Component

The Memory Gatherer component is responsible for gathering memory information acquired from within a domain. The Memory Gatherer component executes both in Domains 0 and Domains U providing invasive memory information. It is necessary to use the invasive procedure because Xentop provides the amount of memory allocated to a domain but lacks information about how much memory is effectively in use.

The Memory Gatherer component receives the parameters of how it should collect the measures and provides a method called `getMeasures()` for returning the acquired measurements in XML format for consolidation with the other measurement tools by the Measures Gatherer Main module.

The parameters for acquiring measurements are the number of iterations that should be taken and the interval between iterations. The information is retrieved from the `/proc/meminfo` kernel interface.

The XML message generated by the Memory Gatherer module follows the pattern of Listing 2.4:

Listing 2.4: XML part generated by the Memory Gatherer module.

```
<MeasureToolResponse>
    <MeasureToolId>4</MeasureToolId>
    <Iteration>
        <IterationNum>1</IterationNum>
        <IterationDateTime>1272889043.56</IterationDateTime>
        <FreeMem>1110692.0</FreeMem>
        <UsedMem>1966096.0</UsedMem>
        <TotalMem>3076788.0</TotalMem>
        <FreeSwap>8773324.0</FreeSwap>
        <UsedSwap>239100.0</UsedSwap>
        <TotalSwap>9012424.0</TotalSwap>
    </Iteration>
    <Iteration>
        ITERATION DATA
    </Iteration>
</MeasureToolResponse>
```

Meaning of the new tags:

- `<FreeMem>`:Contains the amount of free RAM in kilobytes
- `<UsedMem>`:Contains the amount of used RAM in kilobytes
- `<TotalMem>`:Contains the total amount of RAM in kilobytes

- `<FreeSwap>`:Contains the amount of free swap memory in kilobytes
- `<UsedSwap>`:Contains the amount of used swap memory in kilobytes
- `<TotalSwap>`:Contains the total amount of swap memory in kilobytes

### 2.3.5 Ifconfig Gatherer Component

The Ifconfig Gatherer component is responsible for gathering network information acquired from within a domain. The Ifconfig Gatherer module executes both in Domains 0 and Domains U providing invasive network information.

The Ifconfig Gatherer component receives the parameters about how it should collect the measurements in its constructor and provides a method called `getMeasures()` for returning the acquired measures in XML format for consolidation with the other tools measurements by the Measures Gatherer Main module.

The parameters for acquiring measurements are the number of iterations that should be taken with `ifconfig` and the interval between iterations.

The XML message generated by the Ifconfig Gatherer component is in the pattern of the Listing 2.5:

Meaning of the new tags:

- `<NetworkInterface>`:Indicates the beginning of the description of a network interface
- `<NetworkInterfaceName>`:Contains the name of the network interface
- `<LinkEncap>`:Contains the link encapsulation type
- `<LinkMAC>`:Contains the interface MAC address
- `<IPv4Addr>`:Contains the IPv4 address of the interface
- `<IPv4BcastAddr>`:Contains the IPv4 broadcast address of the interface
- `<IPv4NetMask>`:Contains the IPv4 netmask of the interface
- `<IPv6Addr>`:Contains the IPv6 address of the interface
- `<IPv6Scope>`:Contains the scope of the IPv6 address
- `<LinkMetric>`:Contains link metric value
- `<LinkMTU>`:Contains link Maximum Transmission Unit

Listing 2.5: XML part generated by the Ifconfig Gatherer module.

```
<MeasureToolResponse>
    <MeasureToolId>2</MeasureToolId>
    <Iteration>
        <IterationNum>1</IterationNum>
        <IterationDateTime>1272886080.72</IterationDateTime>
        <NetworkInterface>
            <NetworkInterfaceName>lo</NetworkInterfaceName>
            <LinkEncap>Local  Loopback</LinkEncap>
            <LinkMAC></LinkMAC>
            <IPv4Addr>127.0.0.1</IPv4Addr>
            <IPv4BcastAddr></IPv4BcastAddr>
            <IPv4NetMask>255.0.0.0</IPv4NetMask>
            <IPv6Addr>::1/128</IPv6Addr>
            <IPv6Scope>Host</IPv6Scope>
            <LinkMetric>1</LinkMetric>
            <LinkMTU>16436</LinkMTU>
            <TransmittedBytes>82280</TransmittedBytes>
            <TransmittedPackets>309</TransmittedPackets>
            <TransmissionErrors>0</TransmissionErrors>
            <TransmissionDrops>0</TransmissionDrops>
            <TransmissionOverruns>0</TransmissionOverruns>
            <TransmissionCarrierErrors>0</
                TransmissionCarrierErrors>
            <TransmissionCollisions>0</TransmissionCollisions>
            <TransmissionQueueLength>0</TransmissionQueueLength>
            <TransmissionRate></TransmissionRate>
            <ReceivedBytes>82280</ReceivedBytes>
            <ReceivedPackets>309</ReceivedPackets>
            <ReceptionErrors>0</ReceptionErrors>
            <ReceptionDrops>0</ReceptionDrops>
            <ReceptionOverruns>0</ReceptionOverruns>
            <ReceptionFrameErrors>0</ReceptionFrameErrors>
            <ReceptionRate></ReceptionRate>
        </NetworkInterface>
        <NetworkInterface>
            NETWORK INTERFACE DATA
        </NetworkInterface>
    </Iteration>
    <Iteration>
        ITERATION DATA
    </Iteration>
</MeasureToolResponse>
```

- `<TransmittedBytes>`:Contains the total amount of bytes transmitted through the interface

- `<TransmittedPackets>`:Contains the total amount of packets transmitted through the interface
- `<TransmissionErrors>`:Contains the total amount of transmission errors from the interface
- `<TransmissionDrops>`:Contains the total amount of packets dropped from the interface during transmission
- `<TransmissionOverruns>`:Contains the total amount of transmission overruns from the interface
- `<TransmissionCarrierErrors>`:Contains the amount of transmission carrier errors from the interface
- `<TransmissionCollisions>`:Contains the total amount of transmission collisions from the interface
- `<TransmissionQueueLength>`:Contains the length of the transmission queue
- `<TransmissionRate>`:Contains the transmission rate in bytes per seconds calculated using the iterations, value not present in the first iteration
- `<ReceivedBytes>`:Contains the total amount of bytes received through the interface
- `<ReceivedPackets>`:Contains the total amount of packets received through the interface
- `<ReceptionErrors>`:Contains the total amount of reception errors from the interface
- `<ReceptionDrops>`:Contains the total amount of packets dropped from the interface during reception
- `<ReceptionOverruns>`:Contains the total amount of reception overruns from the interface
- `<ReceptionFrameErrors>`:Contains the amount of reception frame errors from the interface
- `<ReceptionRate>`:Contains the reception rate in bytes per seconds calculated using the iterations, value not present in the first iteration

### 2.3.6 Latency Gatherer Component

The Latency Gatherer component is responsible for gathering latency information for all the neighbors of a domain. The Latency Gatherer component executes both in Domains 0 and Domains U providing invasive network information. The Latency Gatherer uses information from the Topology Discover component to know which addresses should be probed using the ping profiling tool.

The Latency Gatherer component receives the parameters of how it should collect the measures in its constructor and provides a method called `getMeasures()` for returning the acquired measurements in XML format for consolidation with the other tools measures by the Measures Gatherer Main component.

The parameters for acquiring measurements are the number of iterations that should be taken, the interval between iterations and also the number of iterations and delay used in the ping probing.

The XML message generated by the Latency Gatherer component is in the pattern of Listing 2.6:

Meaning of the new tags:

- `<IterationNeighbor>`: Indicates the beginning of the description of the latency measures to a neighbor
- `<NeighborIP>`:Contains the IP address of the neighbor
- `<NeighborData>`:Indicates the beginning of the measures data
- `<PacketsTransmitted>`:Contains the number of transmitted probe packets
- `<PacketsReceived>`:Contains the number of received probe packets
- `<PercentageOfPacketLoss>`:Contains the packet loss percentage value
- `<TimeTaken>`:Contains the time spent for measures of the iteration
- `<RttData>`:Indicates the beginning of the description of the Round Trip Time data
- `<Min>`:Contains the minimal RTT value
- `<Avg>`:Contains the average RTT value
- `<Max>`:Contains the maximum RTT value
- `<MDev>`:Contains the mean deviation of the RTT value

Listing 2.6: XML part generated by the Latency Gatherer module.

```
<MeasureToolResponse>
    <MeasureToolId>3</MeasureToolId>
    <Iteration>
        <IterationNum>1</IterationNum>
        <IterationDateTime>1272889043.56</IterationDateTime>
        <IterationNeighbor>
            <NeighborIP>192.168.1.110</NeighborIP>
            <NeighborData>
                <PacketsTransmitted>20</PacketsTransmitted>
                <PacketsReceived>20</PacketsReceived>
                <PercentageOfPacketLoss>0</
                    PercentageOfPacketLoss>
                <TimeTaken>3801</TimeTaken>
                <RttData>
                    <Min>0.161</Min>
                    <Avg>0.229</Avg>
                    <Max>0.260</Max>
                    <MDev>0.026</MDev>
                </RttData>
            </NeighborData>
        </IterationNeighbor>
        <IterationNeighbor>
            NEIGHBOR DATA
        </IterationNeighbour>
    </Iteration>
    <Iteration>
        ITERATION DATA
    </Iteration>
</MeasureToolResponse>
```

## 2.4 Xen Topology

The objective of this module is to discover the network topology. The topology cannot be inferred using only the list of registered nodes because it does not present the interconnection among network elements. The new architecture for the Internet, in the current project proposal, suggests that different services have different requirements, and independent virtual networks implement these requirements. Therefore, it is necessary to know both how the virtual networks organize themselves over the physical network and the physical network topology. In our prototype, we created a module to provide physical and virtual network topologies called Xen Topology. To

reach this goal, the module probes all neighbors [2] of each network element using the Nmap Security Scanner [31].

### 2.4.1 Methodology

The Xen Topology module of our prototype has three components. The first one, called Scanning component, is responsible for discovering all the neighbors of one network element. This module runs on every physical or virtual element. The second module aims at consolidating the information of one physical network element, which includes the neighbors in the physical network and the neighbors of all virtual elements running over this physical element. The last one, which runs on the controller node [3], consolidates the topology information of all network elements and creates the network topology graph of the physical and of the virtual networks.

#### 2.4.1.1 Topology Consolidate Component

The Virtual Machine Server calls the Topology Module in order to discover the network topology. The Virtual Machine Server has a list of the registered nodes of the physical network, but there is no information about the interconnections among these nodes. The Topology Module will provide the interconnection information. In order to reach this goal, the Topology Module asks every physical node in the registered node list about the interconnection information of the physical and virtual elements. After receiving all the interconnection information, the Topology Consolidate component creates the topology of the networks (Fig. 2.7. We model our network topology as a graph. Therefore, the Topology Consolidate component returns a graph representation to the Virtual Machine Server. The graph is represented by the adjacency matrix of each network. Thus, the Topology Module allows the Virtual Machine Server to provide the current topology of physical and virtual networks.

#### 2.4.1.2 Node Consolidate Component

The Node Consolidate component runs on each physical element of the network and works as Fig. 2.8 illustrates. This component aims at obtaining the information about the neighbors of this element in both physical and

[2] The term “neighbor” means those nodes which have direct interconnection through one of the network interfaces of the network element.

[3] The controller node is the Virtual Machine Server, described in the report 2.2.

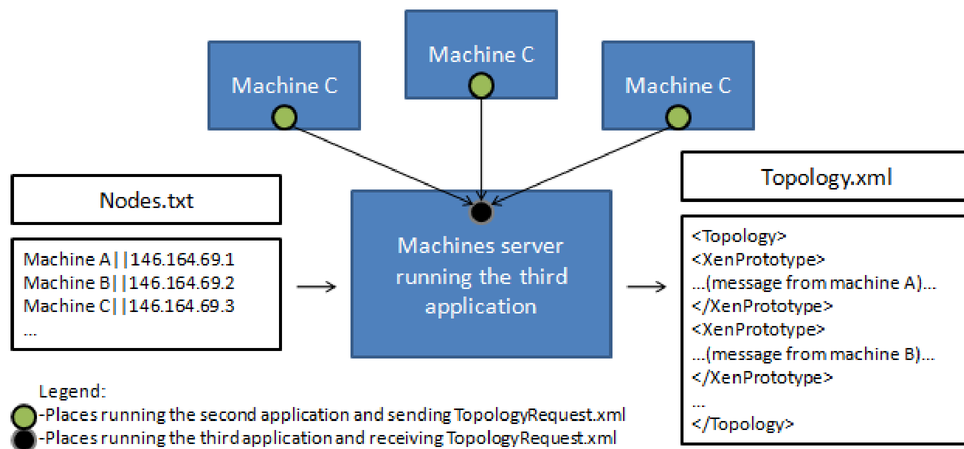



Figure 2.7: The Topology Consolidate component receives the registered node list, asks all the network elements about its neighbors and provides the network topology.

virtual networks. The Topology Consolidate component calls the Node Consolidate component, which has three tasks. The first one is to discover the physical neighbors of the network element. The second one is to discover all the neighbors of each virtual network element running over this physical network element. Both discoveries use the Scanning component presented in Section 2.4.1.3. The first two tasks run at the same time because there is no dependency between them.

The last task of the Node Consolidate component is to consolidate information. The consolidation consists of the neighborhood information of the physical network element and a list of virtual network elements with its neighbors. The module returns the consolidated information to the controller, which runs the Topology Consolidate component using the messages in XML format.

#### 2.4.1.3 Scanning Neighbors Component

The Scanning Neighbors component provides a list of all neighbors of one network element. Both physical and virtual network elements run this component. The neighbor discovery uses the Nmap Security Scanner tool, which probes all the IP range of each network interface of the network element. We add to the neighborhood list all IP addresses that respond the probes. After that, we have all the neighbors of the network element, which are connected by all network interfaces. The neighborhood information has

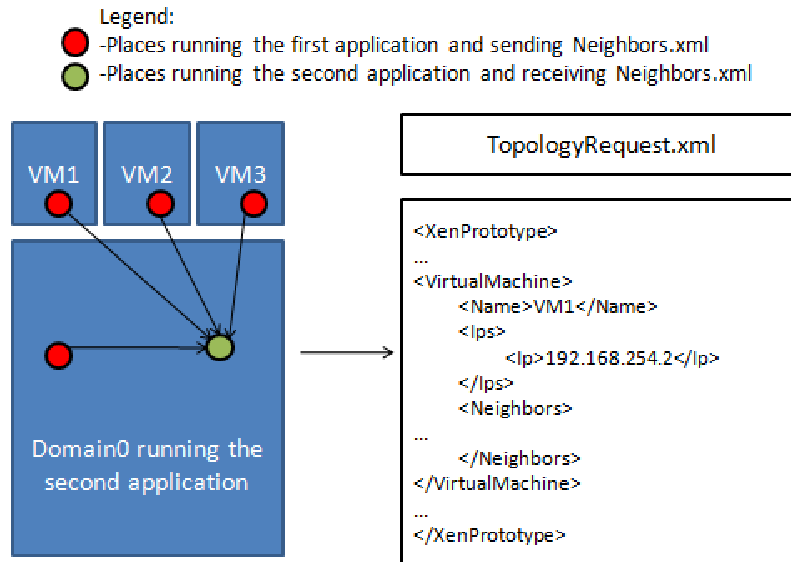



Figure 2.8: The Node Consolidate component discovers the consolidated neighborhood information of the physical network element and all virtual elements running on this physical element.

the neighbor's IP and MAC addresses and the latency of the link. After discovering the neighbors of all network interfaces, the component creates the neighborhood list and transfers it to the Node Consolidate component through XML messages, as we see in Fig. 2.9.

## 2.5 New I/O Virtualization Techniques

### 2.5.1 Introduction

Virtualization has resurged as a way to increase resource utilization efficiency [32, 33] because of the increasing power of modern servers which are leading to underutilized hardware. Using virtualization, it is possible to migrate a service to a Virtual Machine (VM) and to consolidate several VMs into a single server, achieving better resource utilization and saving on maintenance costs, such as cooling and energy consumption. The use of virtualized environments, however, brings performance drawbacks. The virtualization layer introduces overhead because it requires an extra task for virtual machine multiplexing. This overhead is particularly critical for I/O intensive loads [34].

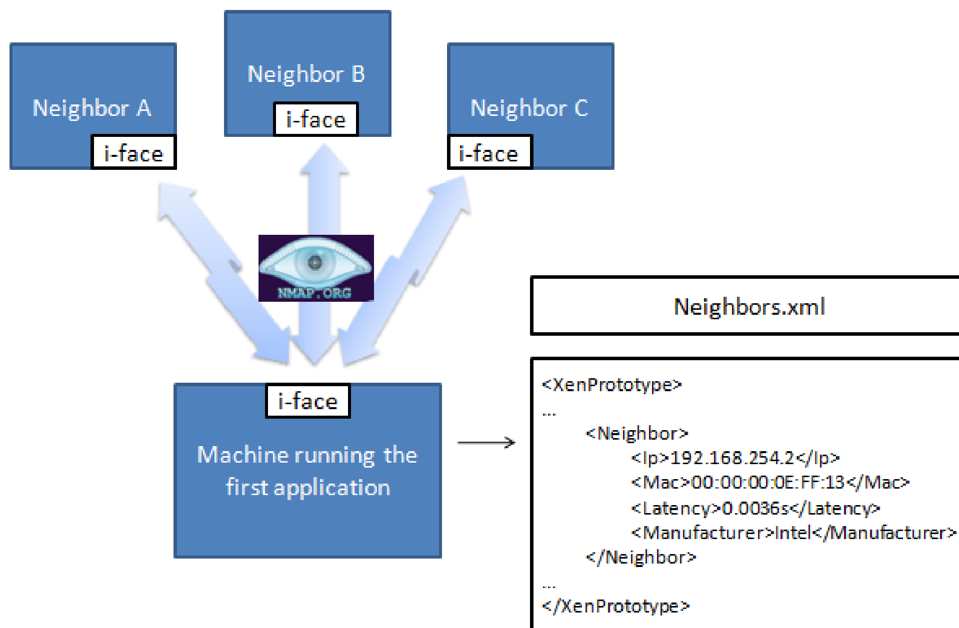



Figure 2.9: The Scanning Neighbors component uses the Nmap tool to discover all the neighbors connected though all network interfaces.

One of the critical cases of heavy I/O utilization is network intensive environments. In this case, providing high throughput and low delay are both required. Moreover, each VM network traffic must be isolated from each other.

Nowadays, I/O virtualization is under responsibility of the Virtual Machine Monitor (VMM), which must multiplex outgoing data flows and demultiplex incoming data flows. Regarding network virtualization, the VMM must share the link, controlling its access and multiplexing incoming packets to the correct virtual network interface. These tasks must be fair considering all VMs.

In order to improve overall network virtualization performance, several techniques have been proposed: direct assignment of device [35], multiqueued devices [36] and single root I/O virtualization (SR-IOV) [37]. These techniques are briefly discussed in the following sections. We also discuss our current efforts on incorporating these new technologies in our prototype.

### 2.5.2 Techniques

#### 2.5.2.1 Direct I/O Access

The Direct I/O technology is a new functionality provided by modern motherboards to safely allow direct device access from VMs. This technique allows device Direct Memory Access (DMA) to different memory areas. Hence, it provides the ability for a device transfers data directly to a VM, without VMM intervention. Memory accesses are assisted by the chipset of the motherboard, which intercepts device memory accesses and makes use of I/O page tables to verify whether the access is permitted and translate the required address to the physical memory address. This mechanism, however, has scalability problems, since a physical device cannot be shared between several VMs. It can only be assigned to one VM.

#### 2.5.2.2 Multiple Queues

The VMM has an important task of classifying packets. Packet classification incurs in great VMM processing overhead, since it demands that the VMM defines the destination VM of all incoming packets and multiplex all outgoing packets. Modern Network Interface Cards (NICs) address this problem by having multiple queues and doing packet classification by themselves. To accomplish this feature, the NIC classifies a packet using a certain pattern (VLAN tag or MAC destination address) and pushes it into the appropriate queue. One or more queues can be assigned to VMs. Thus, they have their traffic isolated from each other.

#### 2.5.2.3 Single Root I/O Virtualization (SR-IOV)

The SR-IOV standard allows sharing PCI-Express devices, such as NICs, among several VMs and accesses them as if they were native. The standard provides a way of bypassing the VMM involvement in data movement. This standard approach also defines a way of sharing a NIC to several VMs. NIC access, using multiple queues and direct I/O, follows the SR-IOV standard.

### 2.5.3 Implementation

We are currently developing a prototype to analyze the performance of the new I/O virtualization techniques. These techniques provide native device sharing support for virtual machines, minimizing overheads in network virtualization. Our final goal is to provide a high performance network infrastructure, capable of fulfilling all requirements of virtual networks.

### 2.5.4 Installation

The newest version of the Xen hypervisor, at the time of this document, is 4.0.0. Xen 4.0.0 has several new features and some of them are related to the new I/O virtualization techniques. This version of Xen is used in this prototype.

An important issue is the management domain Kernel, the Domain 0 Kernel. We need a compatible kernel supporting the new features to configure the devices and hypervisor, making the environment fully functional. The newest Kernel that works with Xen 4.0.0 hypervisor is 2.6.32 Linux Kernel (stable version). Several new features are still under development and are not currently supported.

Currently, it is possible to give a virtual machine full control of a device with Direct Access. It is also possible to create virtual functions, used to access PCI-Express devices, of SR-IOV specification, but it is not possible to give control of these virtual functions to a virtual machine.

#### 2.5.4.1 Direct Access

With Direct Access, a network interface is assigned to a VM, having full control of the interface. The process, that enables the Direct Access in Xen architecture, occurs as follows. The first step is unbinding the device from the default driver. Next, the device is bound to a special driver in Domain 0, the `pcibackend` Xen driver. These steps are necessary for providing Xen with the capability of writing and reading into the configuration memory space of the device. Once the VM has started, the device is assigned to it. The assignment is made using a special parameter in the VM configuration file or a special command for PCI hot-plug, a standard for adding or removing peripherals without restarting the platform or OS.

In our current efforts, the device is successfully assigned to a domain, but using a legacy interrupt mode for interfacing the device with the VM. For this reason, the performance we are experiencing is not comparable to native Linux or Domain 0 performance. Future patches are expected to enable the MSI-X (and MSI) interrupt model, which improves I/O virtualization performance.

#### 2.5.4.2 Multiple Virtual Functions

An SR-IOV compliant interface creates multiple instances of virtual PCI express functions, used for the VMs to share the virtualized device. These virtual functions appear to Domain 0 as new Ethernet interfaces. Although

these virtual functions are configured like an Ethernet interface, some operations are restricted to the physical PCI express function, the physical device. In order to configure and manage SR-IOV compliant interfaces, there are two network drivers: the master driver, which controls the physical function; and the virtual function driver (VF driver). Our NIC master driver is igb (or ixgbe for the 10GB NIC) and its VF driver is igbvf.

When the physical driver is instantiated, the number of virtual function is determined, and they appear as new devices. These new devices are controlled using the VF driver. These virtual devices are created using either native Linux or Domain 0.

Currently these devices cannot be assigned to the VMs, because the entire SR-IOV capability is not yet implemented.

# Chapter 3

# OpenFlow

This chapter presents our OpenFlow-based [11] prototype to build virtual networks [38]. The OpenFlow-based prototype is based on a set of applications running over the NOX Controller [39] (Network Operating System). The NOX is an OpenFlow controller that configures and manages the flows in OpenFlow switches.

Our OpenFlow-based prototype is composed of NOX applications [40]. These applications implement five required primitives: instantiate flow (*instantiate* primitive), delete flow (*delete* primitive), migrate flow (*migrate* primitive), manage and change flows (*set* primitive), statistics acquisition about switches and discover the physical network topology (*monitor* primitive), and also a function to avoid loop within the network. The WebService[1] [41] interface provides the five primitives to the piloting system. The architecture of the OpenFlow prototype is represented in Fig. 3.1.

The OpenFlow-based prototype accomplishes both sensors and actuators defined in the primitives for the piloting system. The actuators are the Flow Manager and the Flow Migration applications. The Flow Manager application offers the *instantiate*/*delete*/*set* primitives. Flow Manager application implements an interface between other NOX applications and the OpenFlow commands. This application is responsible for adding, modifying and deleting flows. Flow Manager application receives a flow operation request and translates it into an OpenFlow command. Other actuator is the Flow Migration application, which implements the *migration* primitive. The Flow Migration migrates a flow from one path to other with no packet loss.

The Stats and the Discovery applications implement the sensors in this prototype. Both applications satisfy the *monitor* primitive. The Stats application measures the network and collects statistics about switches. The

[1] The report 2.2 defines and explains the OpenFlow WebService.

Discovery application discovers the network topology and builds a graph to represent the network. There is also the Spanning Tree application that avoids occurrence of loops in the network and unnecessary broadcast retransmissions.

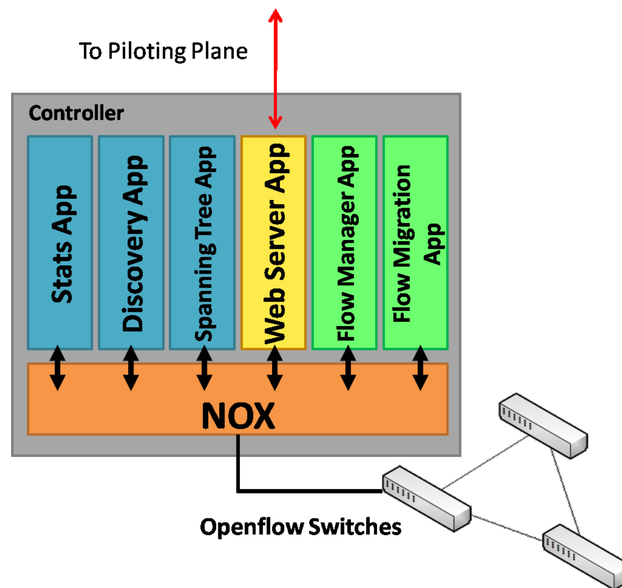



Figure 3.1: OpenFlow prototype architecture.

In addition, we use a tool that enables several NOX controllers to run in parallel in order to slice the network in several virtual networks. This tool, which is called FlowVisor [42], controls the network resource sharing, as bandwidth, topology seen by the controllers, the traffic of each share, the switch CPU usage, and the flow table. It implements the *set* primitive to configure the parameters of each virtual network.

This chapter is organized as follows. Section 3.1 presents the FlowVisor, that allows network resource sharing among NOX controllers. Section 3.2 presents the Migration application, which migrates the flow path with no packet lost. Section 3.3 presents the Stats application. We show the statistics and information that we gather from network and the switch. In section 3.4 we present the Discovery application that discovers the network topology and build a graph representation of the network. Finally, section 3.5 presents the Spanning Tree application that is responsible for avoiding loop occurrence and unnecessary broadcast messages.

## 3.1 FlowVisor

### 3.1.1 Introduction

The FlowVisor [42] is a special type of OpenFlow [11] controller. It works as a transparent proxy between network devices and other controllers, such as NOX controllers. The main feature of FlowVisor is the ability to slice the network and share network resources in a controlled and isolated fashion.

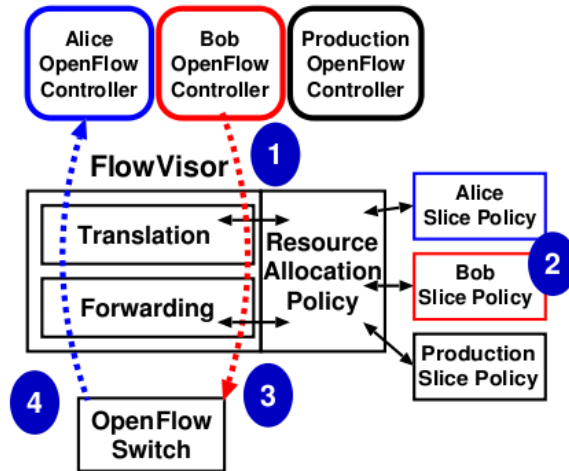



Figure 3.2: FlowVisor Working (Extracted from [42]).

As shown in Fig. 3.2, FlowVisor intercepts the OpenFlow messages sent by guest controllers (1) and, using the user slicing policy (2), transparently rewrites (3) the message, in order to delimit the control to a slice of the network. FlowVisor only forwards messages from switches (4) to guests, if the messages match the slice policy. FlowVisor slices the network, keeping each slice isolated from each other.

FlowVisor can virtualize the network by slicing the use of switches between several controllers. Additionally, FlowVisor allows the creation of hierarchies of FlowVisors, varying the network architecture, as shown in Fig. 3.3. To slice the network among the controllers FlowVisor focus the sharing on five primitive network resources: bandwidth isolation, topology discovery, traffic engineering, device CPU monitoring and forwarding tables control.

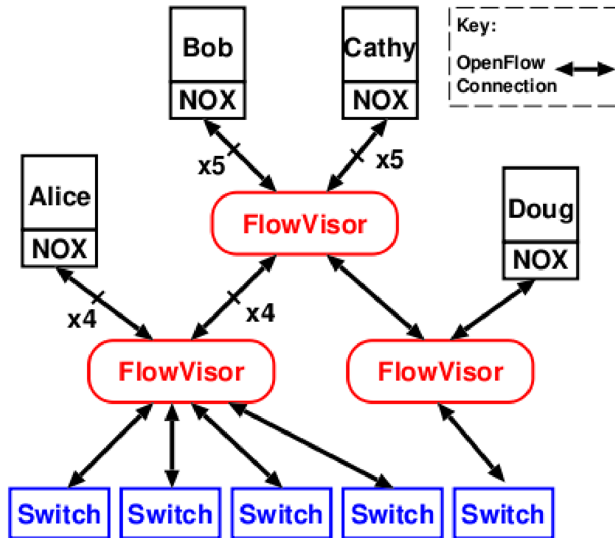



Figure 3.3: FlowVisor hierarchy (Extracted from [42]).

### 3.1.2 Isolation Mechanisms

#### 3.1.2.1 Bandwidth Isolation

To ensure bandwidth isolation, FlowVisor can modify the traffic from a slice to match priority queues on switches. This can be done by three ways: Vlan Priority Code Point (PCP), IP Type of Service (ToS) and OpenFlow QoS.

Vlan tag has a three bit field, the Vlan PCP. This is a standard mechanism to map a packet into one of eight priority queues. The OpenFlow protocol has the ability to manage Vlan tags and the priority bits. Thus, it is possible to mark all packets in a flow with a certain priority. IP ToS can be used in the same way. OpenFlow QoS system may be used to map all traffic from a slice to a certain queue in the OpenFlow Switch.

It is also important to know that what is guaranteed is the minimum bandwidth. The distribution of the extra bandwidth is specific of each implementation of the protocol. In the case of OpenFlow QoS, extra bandwidth is divided in equal parts for each slice.

The exact meaning of each traffic class, or queue, must be configured out-of-band. The network administrator must configure it using switch Command Line Interface (CLI) within each switch.

#### 3.1.2.2 Topology

Controllers discover the network nodes and links them via the OpenFlow protocol. In a non-virtual scenario, a controller listen a TCP port in order to discover a network device when this device actively connects to the controller.

It is possible because the network element communicates with the controller through a TCP connection. Since the FlowVisor acts as a proxy between switch and controller, it proxies connections to a guest controller for the switches in the virtual topology.

The OpenFlow Protocol also defines a message to list the available physical ports on a switch. The FlowVisor modifies the topology discovery message response to report only ports that appear as available in the virtual topology.

## 3.2 OpenFlow Migration

### 3.2.1 Introduction

Flow Migration Application is a NOX [39] application that is responsible for defining a new path and redirect an existent flow into it, with no packet loss during the transaction. This application defines a new path between source and destination OpenFlow switches, and changes the path of the flow to the new one. To migrate a flow, the Flow Migration Application takes as parameter a list of switches. This list defines an ordered group of switches through which the new path flow must pass. In order to calculate the best path from the source to the destination passing through the selected switches, it is used a generalization of the Dijkstra Algorithm [43].

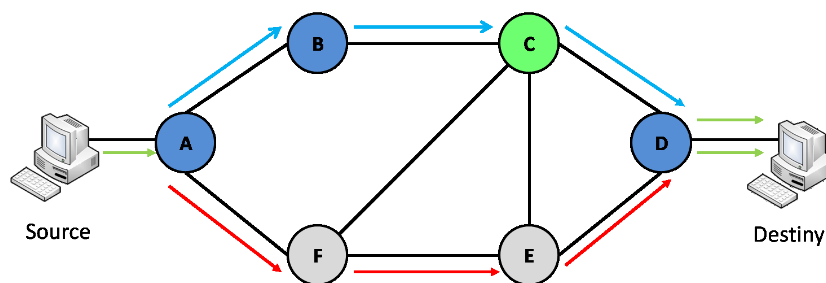



Figure 3.4: Flow Migration Schema.

Fig. 3.4 shows a flow migration schema. Initially, the source sends packets to the destiny through the red path, composed by the nodes A-F-E-D. A new flow from switch A to D is defined, with the requirement that it must pass through the switches A, B, and D, the blue nodes in Fig. 3.4. A, B, and D is

not a complete path between A and D. Thus, the Flow Migration Application calculates a complete path between source and destination switches using the Dijkstra Algorithm. The Dijkstra Algorithm minimizes the number of links in the new path. According to the algorithm, the node C must be included in the path. After defining all nodes that compose the entire path, the application starts to configure the flow in the switches backwardly, that is, it configures the flow from the nearest destiny switch to the farthest. Since all path flow from destiny to source is already configured but the link of the source computer to the switch A, which is the last link to be changed, it avoids packet loss.

### 3.2.2 Implementation

The Flow Migration Application is a NOX application composed of two information classes: the `flow_migration` class and `MigrationHandler` class. These information classes do the flow migration and communicate with the service supplicant, the Webserver Application.

The Flow Migration Application has two interfaces: one with the Webserver Application, by which the control parameters are changed; and the switch interface, where the application sends the flow configuration commands.

This Section describes the the `flow_migration` and `MigrationHandler` information classes, the methods and parameters.

#### 3.2.2.1 `flow_migration` Class

This Flow Migration information class is a NOX Application. It inherits from the `Component` class, and rewrites some parent methods, such as `install` and `getInterface`. The main purpose of this class is the flow migration functionality.

The Webserver requests the migration of a flow and the class calculates the new path and sets it. This functionality is implemented by the `MigrateFlow` method which takes as parameters:

- `dpidsStr`: a string composed by the DPIDs of the switches that must be in the path. Each switch DPID must be separated of others by a '|';
- `match`: a Python dictionary that contains the flow characteristics, as it is in the first switch in the path. The first switch is the one in which the source computer is linked in;

- `priority`: Flow priority. Its default value is the maximum value of priority (65535);
- `hardTimeOut`: Flow hard timeout. Its default value is infinity;
- `idleTimeOut`: Flow idle timeout. Its default value is infinity;

The `MigrateFlow` method return is a XML message that contains the entire flow path, an ending message which shows the ending of migration process. If the migration process has an error, the XML message shows the error and the erroneous switch.

#### 3.2.2.2 `MigrationHandler` Class

`MigrationHandler` class is a general purpose class. It has methods which implement some that are important for the migration process. The routines implemented by `MigrationHandler` class are described below.

`getDpidsList`

This method gets a string containing a sequence of DPIDs separated by a '|'. It returns a Python list of the switch DPIDs, in the order of the new flow path.

`getDictFromXML`

This method gets a XML message, processes the message and returns a Python dictionary containing all information of the XML message. It maintains the XML hierarchy, as it converts a XML tag in a dictionary recursively. It makes dictionaries items of a dictionary, maintaining XML hierarchy.

`CreateTopology`

This method converts the network topology, given by a XML message, in a matrix of 0's and 1's, where the value 1 means that there is a link from row to column switch (and the value 0 means that there is not). Switches are mapped in integer values to index the matrix columns and rows.

`Dijkstra`

This method calculates the best path between a node and the others. It takes as parameters the topology graph and the source node. It returns a tuple composed by a distance vector, that has the distance values of the source node and the node correspondent to the index of the array and a vector of previous nodes, that has the previous node in the path between the source and the node correspondent to the index in the array.

`getPath`

This method gets the two vectors from the Dijkstra method and returns a vector that represents the entire path from one node to another node.

`getEntireDpidPath`

For each pair of switches in the list of switches that the flow must pass through, this method calculates the best partial paths. After that, this method composes the partial paths in an entire path and converts the index notation to the DPID value of its index.

`getPorts`

This method gets the pair of ports used to link switches in the vector returned by the `getEntireDpidPath` method. This method returns a list of DPIDs that defines the order of implementation of the flow through the network, a list of output ports that defines the output port of each switch in the DPIDs list and the input port, that defines the input port for the correspondent switch in the DPIDs list.

## 3.3 OpenFlow Statistics

### 3.3.1 Introduction

The Stats Application is a NOX application responsible for gathering OpenFlow switches information and converting them into XML messages. The Stats application sends commands to the OpenFlow switched network, querying each switch about its statistics and other information. Each switch receives a request to report its description, datapath, table, aggregated flows, and flows information to Nox controller. Each kind of information is obtained with an appropriate OpenFlow Protocol command. The Stats Application interfaces with the Web Server Application and Switchstats Application. The Web Server Application provides an interface between the Stats Application and an http client. The client can be a user interface or another application. The Switchstats Application is responsible for sending OpenFlow commands to switches and handling the responses. From now on, we will refer to Switchstats Application and Stats Application both together as Stats Application, for simplicity purpose.

Stats Application periodically gathers information from the OpenFlow switched network. Whenever a `stats request` is received, the Stats Application responds with a XML message with the currently available information.

### 3.3.2 Stats XML Message

The XML message returned by the Stats Application contains information of every switch registered on the Nox controller. The XML message begins with a root tag named `openflowstats` and has a `datapath` tag for every registered switch. Inside every `datapath` tag there is the switch identifier number (`dpid`), the switch MAC address (`dp_mac`), and specific tags containing information and statistics retrieved from the switch. These specific tags and the internal structures that stores switch information and statistics, referred as *Information Classes*, are described in the next section.

### 3.3.3 Information Classes

#### 3.3.3.1 Description of the Switch Device (`dp_desc_stats`)

This information class stores switch IP Address (`ip`), software description (`sw_desc`) (e.g. software version), hardware description (`hw_desc`), serial number (`serial_num`) and manufacturer name (`mfr_desc`).

#### 3.3.3.2 Statistics about the Switch Datapath (`dp_stats`)

A switch datapath is a logical structure that contains the Flow Table and actions associated with each flow entry. This information class stores datapath information in information subclasses, detailed in the following sections.

**General Information**

This information subclass stores the number of switch implemented tables (`n_tables`); the number of simultaneous supported actions (`actions`) and switch caps number (`caps`). This information subclass also stores the timers for polling switch for flows and ports.

**Port Information**

This information subclass stores statistics about each port state and description. The most important information stored in this information subclass are:

- port number (`port_no`): identifies which physical port is described by the data structure;
- hardware address (`hw_addr`): MAC address of interface port;
- port name (`name`): interface port name;

- port speed (`speed`): nominal port speed;
- enabled flag (`enabled`): indicates if port is enabled or not;
- flood flag (`flood`): indicates port should forward or not flood flows;
- port state (`state`): shows the port state;
- link flag (`link`): indicates whether there is or not an active link in this port;
- received packets number (`rx_packets`): total amount of packets received at this port;
- amount of received bytes (`rx_bytes`): total amount of received bytes at this port;
- errors in CRC check (`rx_crc_err`): amount of packets received at this port that had CRC errors detected;
- dropped received packets count (`rx_dropped`): total amount of packets dropped upon reception at this port;
- received packets errors (`rx_errors`): amount of packets received at this port containing errors;
- error frames (`rx_frame_err`): amount of received frames at this port containing errors;
- transmitted packets (`tx_packets`): total amount of transmitted packets on this port;
- transmitted bytes (`tx_bytes`): total amount of transmitted bytes on this port;
- dropped transmitted packets (`tx_dropped`): total amount of packets that were dropped due to transmission buffer overflow;
- transmitted packets errors (`tx_errors`): amount of transmitted packets errors;
- collisions count (`collisions`): amount of collisions detected at this port;

#### 3.3.3.3 Table Information (`dp_table_stats`)

This information class stores information about the forwarding tables of the switches. Each switch has two forwarding tables by default: hash table, in which the flows are fully described and a flow match is done using a hash function; and the linear table, in which partial flows are described and searching for a flow in this table is uses a linear complexity function. The main information in this class are:

- Table identifier (`table_id`): number used to identify the kind of table, hash tables have table_id equal to 0, while linear tables have table_id equal to 1;
- Matched count (`matched_count`): count of how many flows have matched an entry in this table;
- Name (`name`): table name. Typical values are "hash" or "linear";
- Active count (`active_count`): amount of active flows in the table;
- Maximum number of entries (`max_entries`): maximum number of flow entries in a table;
- Look up count (`lookup_count`): amount of flows already looked up in this table.

#### 3.3.3.4 Aggregated Information (`dp_aggr_stats`)

This information class stores aggregated data about:

- sent and received packets (`packet_count`): sum of sent and received packet for all flows for all ports;
- sent and received bytes (`byte_count`): sum of sent and received bytes for all flows for all ports;
- flows count (`flow_count`): total amount of active flows taking into consideration all switch tables.

#### 3.3.3.5 Flow Information (`dp_flow_stats`)

This information class stores information about flows. A flow entity is composed by the following attributes:

- Packet count (`packet_count`): count of how many packets passed through this flow;
- Hard timeout (`hard_timeout`): flow expiration time, does not depend on flow being used or not;
- Byte count (`byte_count`): total amount of bytes passed through this flow;
- Actions (`actions`): possible controller actions for a packet matching this flow. There can be one or more actions and their main attributes are:
  - Type (`type`): type of the action. The most common is type 0, which corresponds to the `output` action;
  - Lengh (`len`): length in bytes of the action structure;
- Flow priority (`priority`): defines flow priority;
- Idle Timeout (`idle_timeout`): idle flow expiration time;
- Table Identifier (`table_id`): identifier of the flow table this flow belongs to.
- Duration (`duration`): counts for how long this flow exists in table table_id;
- Match (`match`): defines flow characteristics. A flow match is composed by the fowling entities:
  - In port (`in_port`): entrance port of the flow in the switch;
  - Data layer type (`dl_type`): layer 2 used;
  - Data layer source address (`dl_src`): layer 2 source station;
  - Data layer destination address (`dl_dst`): layer 2 destination station address;
  - Data layer Vlan tag (`dl_vlan`): layer 2 protocol header Vlan value;
  - Network layer protocol (`nw_proto`): layer 3 used protocol;
  - Network layer source address (`nw_src`): layer 3 protocol source station address;
  - Network layer destination address (`nw_dst`): layer 3 protocol destination station address;

– Transport protocol source port (`tp_src`): transport layer source port, e.g. TCP source port.

– Transport protocol destination port (`tp_dst`): transport layer destination port, e.g. TCP destination port.

## 3.4 Openflow Discovery

The understanding of the network topology is important to develop control applications. As examples of these applications we can mention the creation of a spanning tree in the network, the choice of routes, the traffic management and the security mechanisms based on access control. The topology of both physical and virtual networks must be available for the network piloting system to allow the network piloting.

The network topology discovery is implemented in Python as an NOX application called Discovery. We had modified the original Discovery application to make it compatible with any OpenFlow device and also to provide an interface for the developed graphical interface. Our Discovery implementation also gets the topology of physical networks and virtual networks.

Discovery implements the LLDP protocol (Link Layer Discovery Protocol) [44]. LLDP is a link layer protocol that allows nodes to transmit information about the capabilities and the current status of network devices. The LLDP implementation is optional in the protocol stack of an IEEE 802 LAN station. The LLDP frame follows the model described in Fig. 3.5. Inside LLDPDU, which is the data unit of LLDP, we can find the device information stored in type-length-value (TLV) structures.

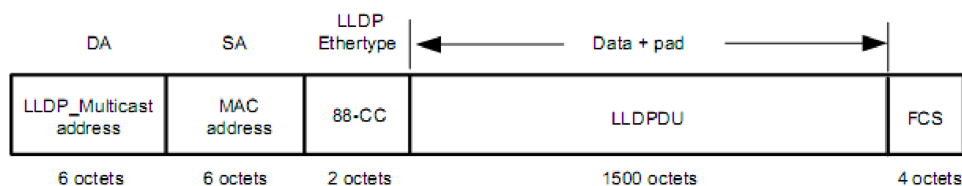



Figure 3.5: LLDP frame format for IEEE 802.3 (Extracted from [44]).

The Discovery application creates an LLDP frame for all the ports of a given switch. The switch propagates those frames through its ports.

Upon the receiving of the first LLDP frame, the switch does not know any forwarding rules for that frame. The switch forwards the frame for the

controller to analyze its contents. After receiving the frame, the controller requests the running instance of Discovery to process this frame. Thus, Discovery analyzes which switch received the frame, from which port the frame was received, which switch sent the frame, and in which port the frame was sent. With this information, the application creates a data structure with source switch, source port, destination switch, and destination port. This data structure identifies a link, and the aggregation of all links of a network characterizes the network topology. The Discovery algorithm was modified to guarantee correct topology discover on Type-1 OpenFlow switches. On Type-0 switches, which are the simplest model of OpenFlow switch, used for running OpenFlow on personal computers, a frame is forwarded only if the controller has already configured a flow for that frame. If there is no configured flow, the frame is dropped. On switch type 1, used on commercials switches, there is the possibility of processing the traffic as if there was no OpenFlow in the case where the controller has not configured a specific flow rule. When the automatic drop is not available, the original Discovery generates an incorrect representation of network topology. In the original algorithm, after processing an LLDP frame that was sent from switch A to switch B, the controller will not send the frame drop command. For this reason, when the network is made of commercials switches, switch B forwards the LLDP frame to other switches connected to it, instead of dropping the frame. Consequently, when the frame is forwarded to other switch C not connected to the source switch A, the controller will identify a link between A and C. This link, however, does not exist physically, inducing the creation of false topologies. To fix this problem, we have modified the algorithm in order to allow it to always send the drop command to the switches. Both algorithms are represented in Fig. 3.6.

The top of the Fig. 3.6 represents the original discovery algorithm. The number 1 indicates that switch A had sent a LLDP packet to swith B. The number 2 indicates that switch B had received the packet from A and then sent it to the controller. In this step, the controller identifies the existence of a link between switches A and B. The number 3 indicates that the switch B had not received a drop command from the controller and due to that, switch C receives the frame from B. The number 4 indicates that switch C has received the LLDP packet and consequently has sent it to the controller. The controller identifies the existence of a link between switches A and C. The bottom of Fig. 3.6 represents the modified discovery algorithm. The number 1 indicates the switch A sending a LLDP packet. The number 2 indicates that switch B has received the packet from A and sent it to the controller, that identifies a link between switches A and B. The number 3 indicates that controller has sent the drop command to switch B. The number

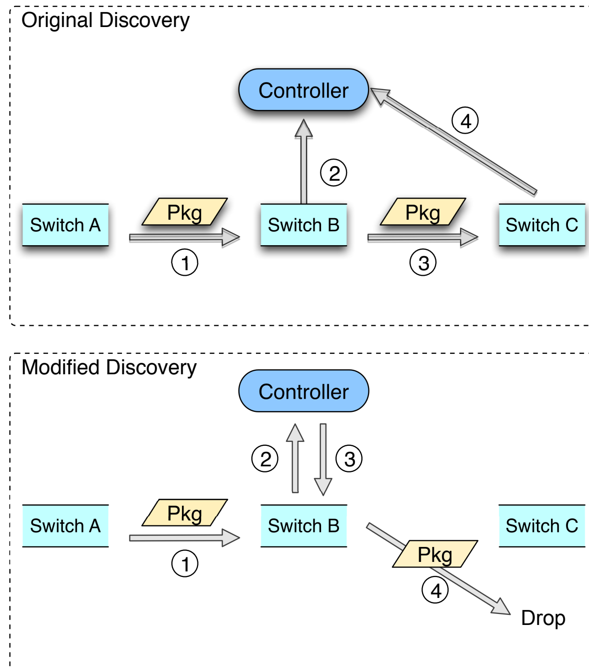



Figure 3.6: Behavior of the original and the modified Discovery algorithm. The modified algorithm uses the drop command to avoid the creation of false topologies. The drop command is represented in steps 3 and 4 in the modified Discovery algorithm. In step 3, the controller sends the drop command to switch B. On step 4, switch B, upon the receiving of the drop command, drops the packet.

4 indicates that switch B drops the packet before it arrives onto switch C.

Another functionality in our implementation is the virtual network topology discovery. The original application only provides the physical network topology. Our application provides the topology of any virtual network running over the physical substrate. To achieve this goal, our application receives the definition of the virtual network, for instance, a VLAN identifier or an IP range, and then it provides a structure with the virtual network topology, which comprises all the switches that are forwarding traffic for that virtual network.

The original Discovery application executes periodically to maintain the topology updated. Whenever a switch is connected to the controller, its ports are added into a special structure. When the Discovery is called, LLDPs frames are sent through these new ports and the new links are recognized in

the topology discovery algorithm. Likewise, whenever a switch disconnects from the network, its ports are removed from the structure and the links associated to these ports are removed from the topology in the next discovery period. We have also changed this part of the algorithm. In the modified version, instead of only storing the new data and waiting for a new Discovery execution, whenever a switch joins or leaves the network, the topology update algorithm is immediately called. Thus, the topology structure is always updated.

Discovery is based on three timers, each one controlling an interval or limiting code execution. The first timer determines the interval between rounds of transmission of LLDP frames, and the default value of this timer is 0.1 seconds. The second timer determines the interval between checks of maximum idle time of links. This timer has a default value of 5 seconds. In Discovery, links do have a maximum idle time. Hence, if a link already identified is not identified again after this maximum idle time, it is removed from the topology. The third timer sets the maximum idle time of the link, and its default value is 10 seconds. In the original Discover, these values cannot be modified by the user. We created an interface to allow the configuration of these timers.

To enhance the compatibility of our prototype, we have also created an interface between Discovery and other applications developed for Horizon project. We have modified the original algorithm output to display topology data in two XML formats. The implementation is based on XML to standardize the transmission of structures between NOX applications and Web servers. The first XML format contains a list of the network links with switches and ports that comprises each link. In the second XML format, we can find a list of switches with its neighbors and the ports that connects them. With these two XML formats, it is possible for any other application to access the physical and virtual topologies and execute commands according to it. These two descriptions can be represented as a graph, easing the visual representation of the topologies.

## 3.5 Openflow Spanning Tree

In most networks, there are redundant paths. This is important to prevent that a link failure damages network operations. If there is a redundant path, this path routes the frames after the link failure. Although redundant paths increase network reliability, it causes a problem during broadcasts procedures in Ethernet networks. In fact, if a node connected to a switch broadcasts a frame, the switch forwards this frame to all ports but the incoming port.

If the network has redundant paths, as shown in Fig. 3.7, the frame will be re-broadcasted by all switches until it returns to the first switch. Because the first switch is not able to identify that the frame was already forwarded, it forwards the frame again. Hence, the frame will be continuously forwarded due to a loop in the network topology. In order to avoid these loops, we use the Spanning Tree algorithm [45] to control the physical topology of our OpenFlow prototype. This algorithm creates a topology for broadcasting frames without creating loops. In Fig. 3.7, when we disable broadcast for the link between A and C, we do not produce any loops in the network.

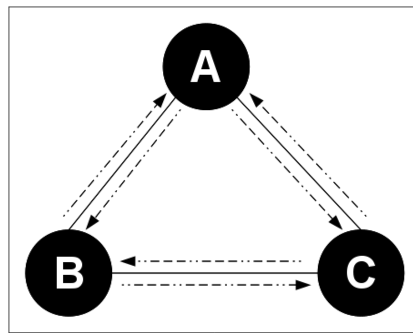



Figure 3.7: An example of topology with a loop. If node B sends a broadcast message, A and C will receive and forward the message. When node A receives the message forwarded by node C, it will forward the message again, and both B and C will receive and forward the message, creating a loop.

The Spanning Tree algorithm runs as a NOX application [39]. The algorithm builds a graph that maps the network topology. Each node and each edge of the graph represents a switch and a link, respectively. The algorithm analyzes the graph and builds a tree containig all the nodes of the graph and the selected edges, as shown in Fig. 3.8. Disabling all links (edges) outside of the spanning tree means these disabled links will not forward broadcast messages. This mechanism prevents frames from passing through the disabled links, loop occurrence and unnecessary broadcast traffic retransmission.

The Spanning Tree algorithm uses the topology provided by the Discover application (section 3.4) Our algorithm works as follows. First, it sorts the switches (nodes) ordered by its identifiers (Ids). After that, it selects the switch that has the lowest identifier as the root of the spanning tree. Then, it checks the switches connected to the root switch and marks them as visited, which means that the node and the link between these nodes and the root also compose the spanning tree. Finally, it checks the switches connected to the already visited switches. If any of these switches is marked as visited

the link between them does not compose the spanning tree. Otherwise, the algorithm inserts both the node and the corresponding link in the spanning tree. It uses a breadth first search algorithm [46]. If we start at a particular node (say, $n1$), the breadth first search algorithm searches all nodes $M-hops$ away from $n1$ before all nodes $M+1-hops$ away from $n1$. For instance, in Fig. 3.8, if $n1 == B$, then the algorithm searches $D$ and $E$ before $G$ and $H$.

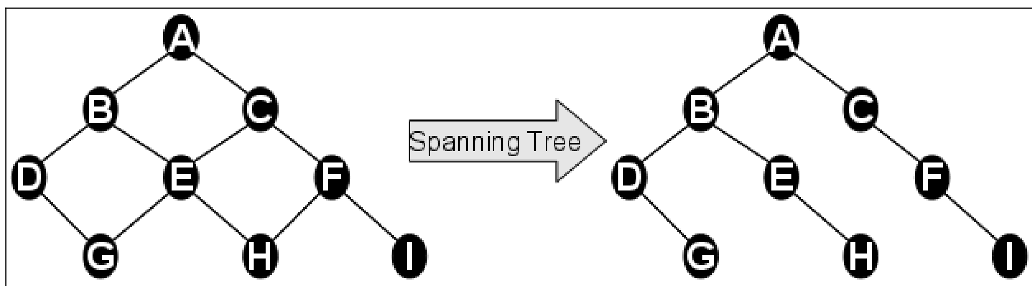



Figure 3.8: Example of a network topology and its corresponding spanning tree, assuming A as the root of the spanning tree.

The Spanning Tree algorithm runs periodically to keep the spanning tree updated. Connecting a new switch to the controller demands the Spanning Tree to disable all ports of this switch until it analyzes the graph and includes the new switch in the spanning tree. For security, after a minimum amount of time, the Spanning Tree application enables these ports to ensure they will be identified by Discovery application, and, consequently, included in the network topology.

The topology that the Discover application provides shows each link twice, one for each direction. When a problem occurs in the topology discovery or when wireless links are being used, unidirectional links may appear in the link list. The application does not use these links when creating the spanning tree, therefore it disables them for broadcasting messages..

When a switch receives a new broadcast data flow, it forwards the frame to the controller. After receiving the frame, the controller calls the running instance of Spanning Tree to analyze the link through which the broadcast frame was received. If this link is in the spanning tree, the switch processes and forwards the frame. Otherwise, the application drops the frame. When the frame received is a LLDP (Link Layer Detection Protocol) frame , the processing and forwarding happens independently of the source link because the Discover application uses that frame to discover the network topology. Also, if the controller has already defined a rule in the switch for the received broadcasted frame, the switch does not forward that frame to the controller,

but processes the frame according to the defined rule instead, even if the link is not in the spanning tree.

The Spanning Tree application has three timers. The first one timer determines the interval between consecutively execution rounds of the building spanning tree algorithm, and its default value is 5 seconds. The second timer determines how long a port on a new registered switch must be disabled until it can be included in the tree, and its default value is 10 seconds. The third timer manipulates one of the Discovery timers, which determines the interval between the algorithm execution to topology discovery, and its default value is 0.05 second. We created an interface to configure these timers that receives messages from other applications running on NOX.

We also created an interface between Spanning Tree and other applications developed for Horizon Project. Therefore, we modified the original algorithm output to display Spanning Tree data in a XML format. The messages are based on XML format to standardize the transmission of structures between NOX applications and the Web Server application. In this XML format, we enumerate the list of switches with its associated neighbors and the ports between them. With this XML format, any application can access and control the Spanning Tree application.

# Chapter 4

# Performance Tests

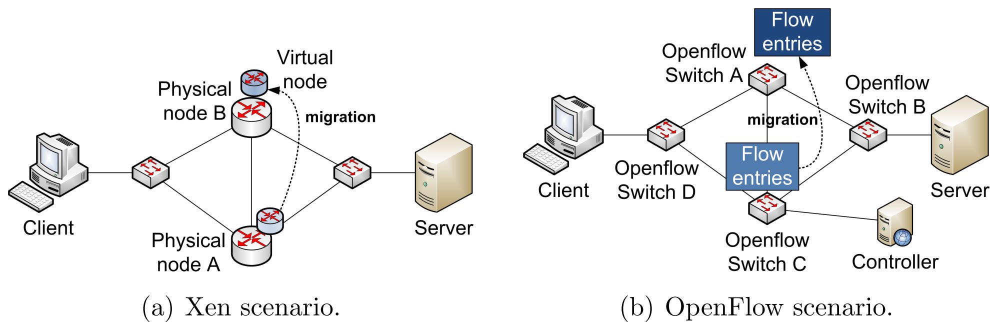
This chapter presents some performance tests to evaluate the virtualizations tools. In section 4.1 we present some tests made with migration tools from both prototypes, Xen and OpenFlow.

## 4.1 Migration Tests

### 4.1.1 Introduction

This section describes the performance tests of migration functionality of both prototypes, Xen and OpenFlow. In Xen prototype, we test the two virtual approaches for router virtualization, with and without plane separation, as described in section 2.2. In OpenFlow prototype, we use the NOX migration application described in section 3.2 to migrate a flow from "Client" to "Server". Accordingly, there is nopacket loss during OpenFlow migration.

In Fig. 4.1, we show the migration scenarios. In the Xen scenario, we have two physical machines, Physical Node A and Physical Node B, which house the virtual routers. The experiment consist of generating an UDP data traffic and migrating a virtual router from Physical Node A to Physical Node B while the virtual router forwards the data traffic from the client to the server. To not disturb the data traffic, we use a separated link to transmit the migration traffic during the migration process. Fig. 4.1(a) shows this process. In the Xen standard migration, the data traffic is forwarded by the virtual router. In the Xen with plane separation, the data traffic is forwarded by the shared data plane in Domain 0 and the control traffic is forwarded by the virtual machine. In the Openflow scenario, the two physical machines with Xen are replaced by OpenFlow switches. The controller defines the flow path from "Client" to "Server" which passes through switches D, C and B.




(a) Xen scenario.

(b) OpenFlow scenario.

Figure 4.1: Experimental scenarios for virtual network migration.

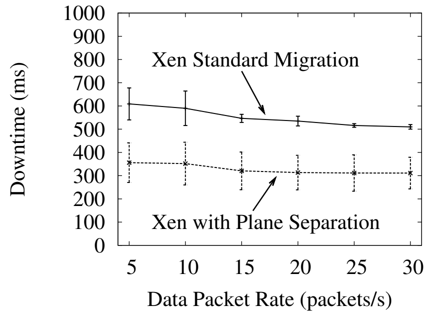
After that, the controller migrates the flow path to now traverse switch A, as shown in Fig. 4.1(b).

### 4.1.2 Results

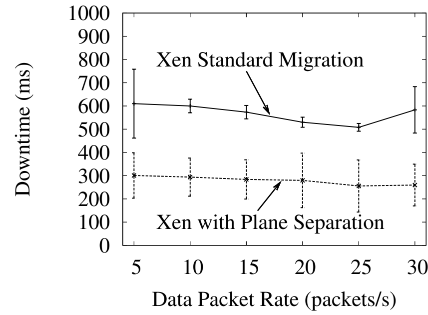
In our experiments, we measured the delays and losses due to the migration with different data packet rates. We present experiments with 64-byte and 1500-byte packets, which respectively represent the minimum Ethernet payload and the most common Ethernet MTU, but there were no significant differences in the results when the packet size varies.

Fig. 4.2 presents the time elapsed during the downtime stage. The results show that the downtime is roughly constant with the growth of the data packet rate for all migration schemes. Compared with the Xen standard migration, the Xen with plane separation has a downtime that is 200 milliseconds lower on the average. This difference exists because in Xen standard migration, the virtual machine must forward all data and control packets, which increases the number of 'hot pages' in virtual machine memory and, consequently, raises the downtime. In the Xen with plane separation, the data traffic is forwarded by Domain 0 and the memory pages dirtied in the virtual machine come only from the routing software running in the virtual machine, which reduces the downtime.

Fig. 4.3 shows the number of data packets that were lost during downtime in the experiments. As expected, in the Xen standard migration, the number of lost packets increases linearly with the packet rate, because the downtime is constant and the data packet rate grows linearly. OpenFlow and Xen with plane separation tackle this issue. Xen with plane separation has a downtime only during the control plane migration and the data plane is not frozen during migration.. In OpenFlow, the migration mechanism moves



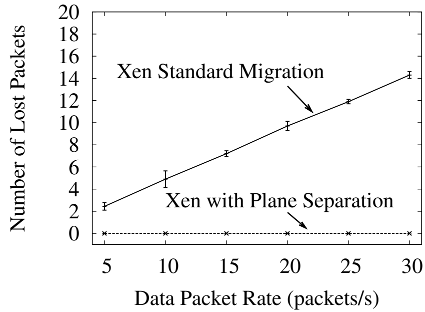
(a) Packet size: 64 bytes.



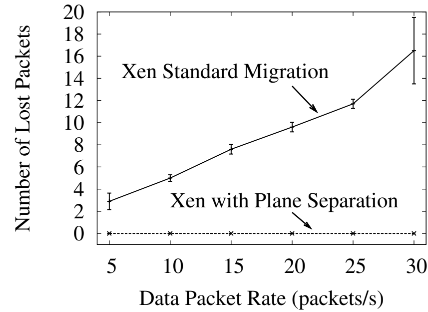
(b) Packet size: 1500 bytes.

Figure 4.2: Migration downtime as a function of the data packet rate.

the data traffic to a new path without migrating the control plane. Indeed, OpenFlow has no downtime, since neither the data plane nor the control plane are stopped due to the migration.



(a) Packet size: 64 bytes.

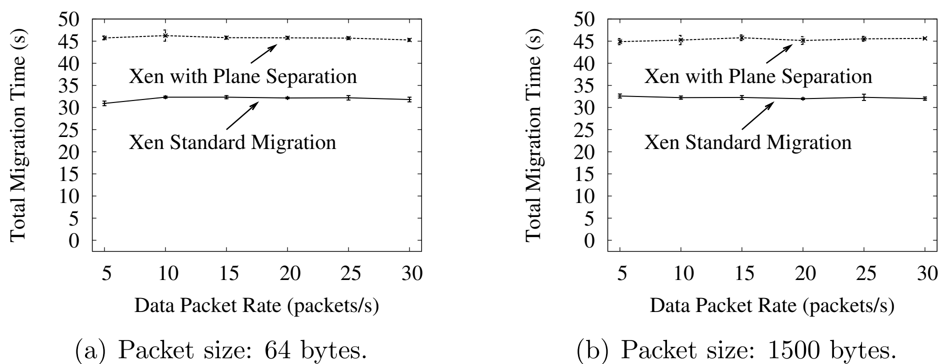


(b) Packet size: 1500 bytes.

Figure 4.3: Number of lost packets during downtime as a function of the data packet rate.

Fig. 4.4 shows the total migration time, which consists of the time between the migration triggering and the moment when the data packets start to pass through the destination machine. The difference between Xen standard mechanism and Xen with plane separation is about 15 seconds. This variation occurs in our prototype due to data plane synchronization after control plane migration and the remapping of the virtual interfaces to the appropriate physical interfaces, which does not exist in the Xen standard migration. OpenFlow total migration time is about 5 milliseconds, because

OpenFlow only migrates the data plane. This time comprises the interval between the beginning and the ending of sending flow change messages from the controller to all switches. In our scenario, the controller reaches all the switches with two hops at the most, as we can see in Fig. 4.1(b). Nevertheless, if the distance between the controller and the OpenFlow switches rises, the total migration time increases. Besides, if the number of migrated flows grows, the total migration time also increases. This happens due to the increased number of messages sent to the switches.




(a) Packet size: 64 bytes.

(b) Packet size: 1500 bytes.

Figure 4.4: Total migration time as a function of the data packet rate.

### 4.1.3 Conclusions

We evaluated the impact of different virtual network migration models with Xen and OpenFlow virtualization platforms. We observed that data and control plane separation is a key feature for reducing packet losses during the migration. Moreover, in the proposed migration mechanism for Xen, the packet forwarding through Domain 0 also reduced the number of dirtied pages in the virtual machine. Therefore, the downtime of the control plane during migration is reduced by 200 milliseconds. We obtained a significant difference in the packet loss rate and control plane downtime during migration when comparing the Xen standard migration and the proposed migration mechanism with plane separation for Xen. Besides, the analysis of the proposed migration algorithm for OpenFlow showed that it is an efficient approach for migrating virtual networks, because it provides a downtime of less than 5 milliseconds and no packet losses during the migration process.

The control plane downtime and the mapping of a virtual link over multiple physical links are the main observed drawbacks in the Xen migration.

OpenFlow has none of these disadvantages, but it is based on a centralized controller, which can restrict the size of the network. Hence, we sketched a hybrid approach to use OpenFlow for migrating data plane in Xen. As future work, we intend to better design this approach and investigate its performance on the data plane migration.

## 4.2 Bandwidth Control Tests

### 4.2.1 Introduction

This section describes the bandwidth control tests using the FlowVisor, described in section 3.1.

The test environment can be seen in Fig. 4.5, it consists of 5 machines: 2 data senders, 1 data receiver, 1 OpenFlow switch and 1 NOX Controller. We use iperf to create two UDP data flows from sender A and sender B to the receiver, and we perform the bandwidth control using queue mechanisms on the OpenFlow switch.

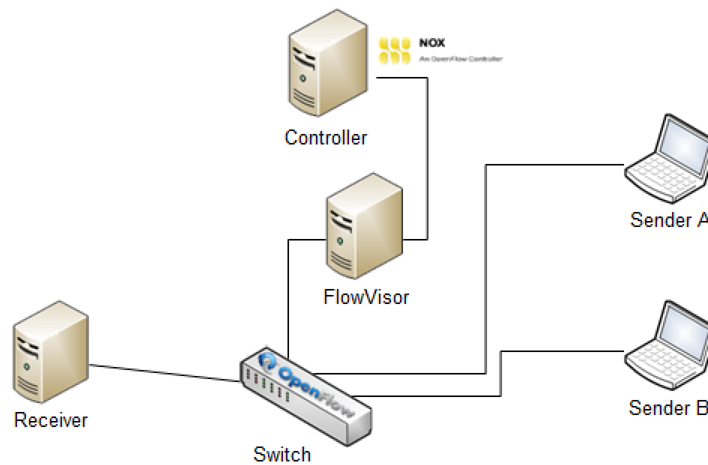



Figure 4.5: The Test Environment

The bandwidth control is organized as following:

- 0s to 30s:
  - Flows with no control (default)
- 30s to 60s:
  - Sender A flow limited to 6 Mb/s
  - Sender B flow limited to B Mb/s
- 60s to 90s:
  - Both flows limited to 4 Mb/s
- 90s to 120s:
  - Flows with no control (default)

### 4.2.2 Results

In our experiments, we measure the bandwidth of the data flows from Sender A and Sender B. We have performed tests according to the proposed schedule mixing TCP and UDP data for Sender A and Sender B.

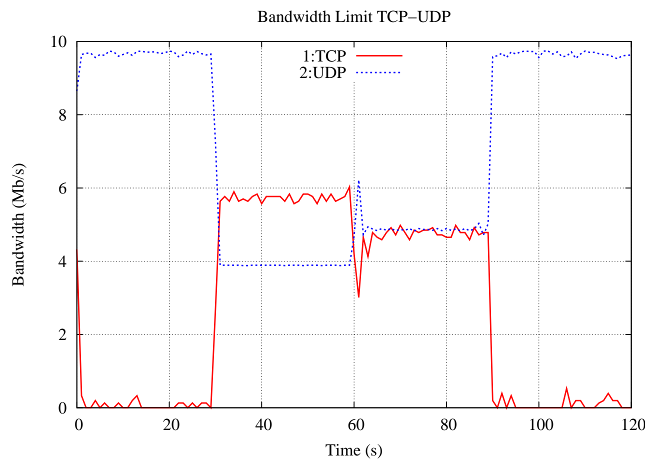



Figure 4.6: TCP-UDP Test

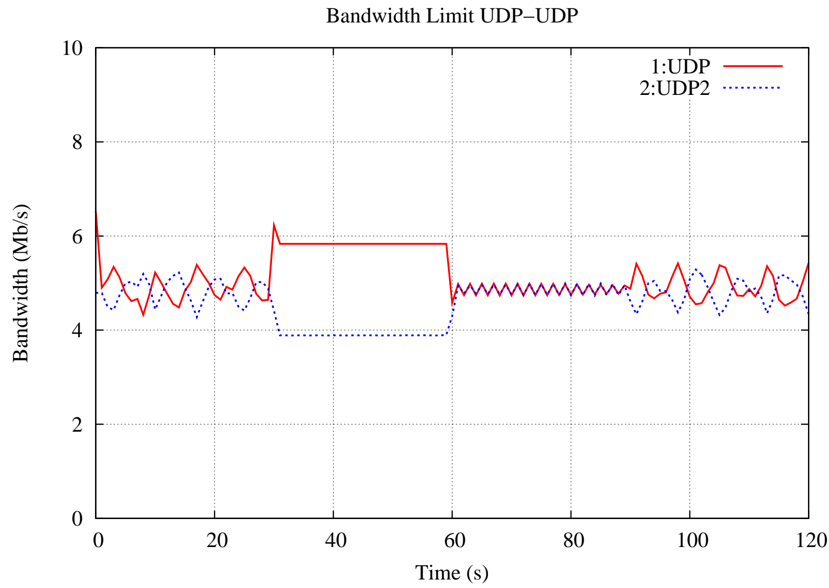



Figure 4.7: UDP-UDP Test

### 4.2.3 Conclusions

By these experiments, we were able to observe FlowVisor's control mechanisms behavior. In each experiment can be seen some instability around the flow change instant, wich is due to the OpenFlow switch's queue mechanisms. The overall conclusion is that the FlowVisor was able to control flow bandwidth as expected.

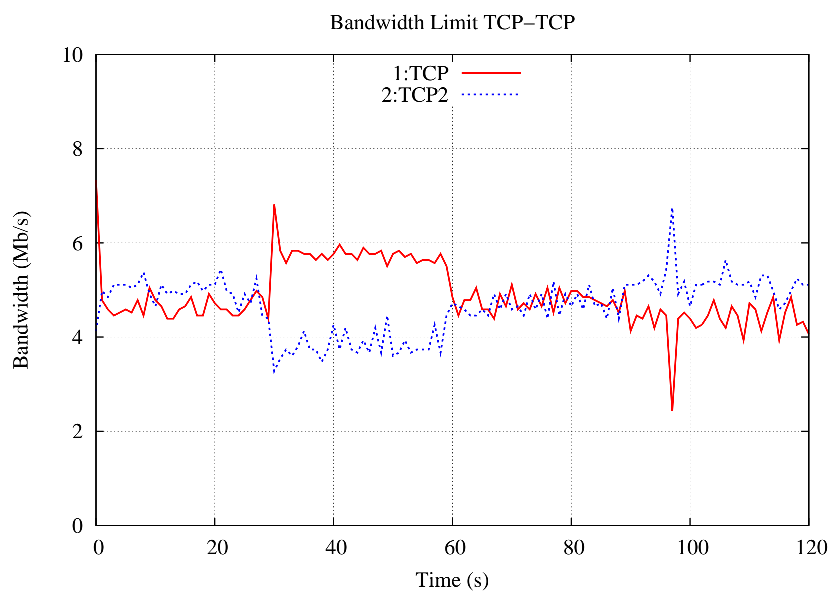



Figure 4.8: TCP-TCP Test

# Chapter 5

# Conclusions and Ongoing Work

In this document we explain how we develop two prototypes to provide an environment suited for the Future Internet requirements. We detail the internal mechanism of our prototypes and expose how they provide to the piloting plane an interface for controlling the virtual networks environment according to the desired policies and primitives. Piloting develops an important role in the Horizon project because the embedded intelligence in the network must be aware of what happens in the environment to react under attacks, to fix wrong configurations and to act pro-actively in the global optimization of the network performance. The piloting capability is provided by means of sensors and actuators that can inspect and modify network elements. We perceive the environment, reason about what is happening and actuate to improve it.

Two different perspectives implement the sensing capability, the non-intrusive and the intrusive. In the non-intrusive perspective, it is possible to sensor information such as the amount of memory allocated to a virtual network element and how many processors can a specific element use. This kind of information is useful in many different situations. Nevertheless, there are cases in which these information are not enough for understanding the environment. This scenario leads to an intrusive perspective, where a `daemon` must execute inside network elements to retrieve more accurate measurements, e.g., the amount of memory is being used or which processes are executing on an element.

In the Xen-based prototype, the sensing capability occurs in components that use different tools to gather information about the environment, such as the Xentop Gatherer, the Ifconfig Gatherer, the Latency Gatherer and the Memory Gatherer. There is also the Xen Topology module that allows us to discover the physical and virtual network topologies. In the OpenFlow-based prototype, the sensor and actuating capabilities rely on the developed

NOX applications. The Stats application allows us to retrieve information about the status of each OpenFlow switch, such as the flows that are passing through the switch and its characteristics. The Discovery application discovers the network topology and the SpanningTree application avoids the unnecessary broadcast traffic retransmission and occurrence of loops in the network.

The actuating capability allows us to actuate on the network and perform tasks such as modifying element configurations on the fly and changing a virtual topology with the aid of the *migrate* primitive. In the Xen-based prototype, we propose a new migration tool, which allows dynamic reconfiguration of a virtual topology with no packet loss, differently from the original Xen migration tool in which the high loss rate raises many issues in the running services. In the OpenFlow-based prototype, we develop a flow migration application. This application can modify a flow path with no packet loss as well. Through these actuators, the piloting plane can dynamically reorganize the topology, avoiding dangerous paths or guaranteeing Quality of Service on selected services.

Given the sensors and actuators, our work walks towards the creation of the piloting plane itself. Currently, we are focusing on the development of algorithms and control systems to use the sensors, retrieve information and use knowledge to pilot the Future Internet through the actuators. For instance, we are developing a control prototype based on the Xen scheduler, which lets us to guarantee SLAs and QoS on the virtual networks by sensing the network behaviors and actuating through the reconfiguration of the scheduler parameters. This will reflect in the slice of resources that a given element can use in a specific moment. We are also developing a control system to manage the power usage of the network elements. This algorithm decides when is the best moment to migrate an element or a flow, or the best moment to suspend elements or rearrange them. Our preliminary tests show that the proposed algorithm can reduce significantly the power usage without lowering the network performance.

As we can see, the sensors and actuators are an important part of the Horizon project. The sensors feed the piloting plane with the necessary information to make management decisions that the actuators apply. The developed modules provide us a substrate network that supports this kind of operations.

# Bibliography

[1] 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.

[2] I. Fajjari, M. Ayari, and G. Pujolle, "VN-SLA: A virtual network specification schema for virtual network provisioning," in *Networks (ICN), 2010 Ninth International Conference on*, pp. 337 –342, 11-16 2010.

[3] C. R. Senna, D. M. Batista, E. R. M. Madeira, and N. L. S. da Fonseca, "Experiments with virtual network management based on ontology," in *WNetVirt'10 : Proceedings of the First Workshop on Network Virtualization and Intelligence for Future Internet*, Apr. 2010.

[4] M. A. S. Jr. and E. R. M. Madeira, "Autonomic management of resources in virtualized networks," in *WNetVirt'10 : Proceedings of the First Workshop on Network Virtualization and Intelligence for Future Internet*, Apr. 2010.

[5] R. B. Freitas, L. B. de Paula, E. Madeira, and F. L. Verdi, "Using virtual topologies to manage inter-domain qos in next generation networks," *International Journal of Network Management*, vol. 20, no. 3, no. 3, pp. 111 – 128, 2010.

[6] F. N. C. van 't Hooft and E. R. M. Madeira, "Resource allocation policies in future multi-agent based virtual network," in *WNetVirt'10 : Proceedings of the First Workshop on Network Virtualization and Intelligence for Future Internet*, Apr. 2010.

[7] G. P. Alkmim and N. L. S. da Fonseca, "Virtual network mapping on a physical substrate," in *WNetVirt'10 : Proceedings of the First Workshop on Network Virtualization and Intelligence for Future Internet*, Apr. 2010.

[8] 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, 2010.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles - SOSP03*, Oct. 2003.

[10] D. M. F. Mattos, C. Fragni, M. D. D. Moreira, L. H. G. Ferraz, L. H. M. K. Costa, and O. C. M. B. Duarte, "Evaluating virtual router performance for the future internet," tech. rep., Electrical Engineering Program, COPPE/UFRJ, June 2010.

[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] H. Carvalho, M. Moreira, N. Fernandes, L. Ferraz, R. Souza, I. Moraes, M. Campista, L. H. Costa, and O. Duarte, "Packet forwarding using Xen," in *WNetVirt'10 : Proceedings of the First Workshop on Network Virtualization and Intelligence for Future Internet*, Apr. 2010.

[13] M. Moreira, N. Fernandes, H. Carvalho, L. Ferraz, R. Souza, I. Moraes, M. Campista, L. H. Costa, and O. Duarte, "Packet forwarding using OpenFlow," in *WNetVirt'10 : Proceedings of the First Workshop on Network Virtualization and Intelligence for Future Internet*, Apr. 2010.

[14] C. Fragni, M. Moreira, D. Menezes, L. H. Costa, and O. Duarte, "Evaluating Xen, VMware, and OpenVZ Virtualization Platforms for Network Virtualization," in *WNetVirt'10 : Proceedings of the First Workshop on Network Virtualization and Intelligence for Future Internet*, Apr. 2010.

[15] 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 2010 - Network of the Future Conference to appear*, Sept. 2010.

[16] M. D. D. Moreira, N. C. Fernandes, L. H. M. K. Costa, and O. C. M. B. Duarte, "Internet do futuro: Um novo horizonte," *Minicursos do*

*Simpósio Brasileiro de Redes de Computadores - SBRC2009*, pp. 1–59, 2009.

[17] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, L. Mathy, and T. Schooley, "Evaluating Xen for router virtualization," in *ICCCN'07: International Conference on Computer Communications and Networks*, pp. 1256–1261, Aug. 2007.

[18] R. dos 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 *WNetVirt'10 : Proceedings of the First Workshop on Network Virtualization and Intelligence for Future Internet*, Apr. 2010.

[19] 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), 2010 Eighth Annual*, pp. 361 –368, 11-14 2010.

[20] D. Chisnall, *The Definitive Guide To The Xen Hypervisor*. Prentice Hall, 2008.

[21] D. M. Batista, C. G. Chaves, , and N. L. S. da Fonseca, "Scheduling virtual machines and grid tasks on clouds," in *WNetVirt'10 : Proceedings of the First Workshop on Network Virtualization and Intelligence for Future Internet*, Apr. 2010.

[22] M. Bourguiba, K. Haddadou, and G. Pujolle, "Evaluating and enhancing xen-based virtual routers to support real-time applications," in *Consumer Communications and Networking Conference (CCNC), 2010 7th IEEE*, pp. 1 – 5, 9-12 2010.

[23] R. Souza, L. Ferraz, M. Campista, L. H. Costa, and O. Duarte, "CPU resource allocation on Xen virtual network environments," in *WNetVirt'10 : Proceedings of the First Workshop on Network Virtualization and Intelligence for Future Internet*, Apr. 2010.

[24] "Libvirt: The virtualization api." http://libvirt.org/. (Accessed June 2010).

[25] "Geni: Global environment for network innovations." http://www.geni.net/.

[26] Y. Wang, E. Keller, B. Biskeborn, J. V. 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.

[27] R. Bolla, R. Bruschi, F. Davoli, and A. Ranieri, "Energy-aware performance optimization for next-generation green network equipment," in *PRESTO'09: Proceedings of the 2nd ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of Tomorrow*, pp. 49–54, 2009.

[28] S.-M. Han, M. M. Hassan, C.-W. Yoon, and E.-N. Huh, "Efficient service recommendation system for cloud computing market," in *ICIS'09: Proceedings of the 2nd International Conference on Interaction Sciences*, pp. 839–845, 2009.

[29] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI'05: Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation*, (Berkeley, CA, USA), pp. 273–286, USENIX Association, 2005.

[30] P. Pisa, M. Moreira, H. Carvalho, L. Ferraz, and O. Duarte, "Migrating Xen virtual routers with no packet loss," in *WNetVirt'10 : Proceedings of the First Workshop on Network Virtualization and Intelligence for Future Internet*, Apr. 2010.

[31] M. Wolfgang, "Host Discovery with nmap," *Insecure. org, URL http://insecure.org/nmap/docs/discovery.pdf, Accessed*, vol. 1, 2007.

[32] R. Figueiredo, P. Dinda, and J.Fortes, "Guest editors' introduction: Resource virtualization renaissance," *Computer*, vol. 38, no. 5, pp. 28 – 31, may 2005.

[33] S. Rixner, "Network virtualization: Breaking the performance barrier," in *ACM Queue*, vol. 6, pp. 36–52, Jan. 2008.

[34] L. Ferraz, H. Carvalho, P. Pisa, and O. Duarte, "New I/O virtualization techniques," in *WNetVirt'10 : Proceedings of the First Workshop on Network Virtualization and Intelligence for Future Internet*, Apr. 2010.

[35] Intel Corporation, *Intel Virtualization Technology for Directed I/O*, Sept. 2008.

[36] Intel Corporation, *Intel 82576 Gigabit Ethernet Controller Datasheet*, Oct. 2009.

[37] Intel LAN Access Division, *PCI-SIG SR-IOV Primer*, Dec. 2008.

[38] N. Fernandes, D. Menezes, C. Gomes, L. Panzariello, V. Torres, M. Moreira, I. Moraes, M. Campista, L. H. Costa, and O. Duarte, "Multinetwork control using OpenFlow," in *WNetVirt'10 : Proceedings of the First Workshop on Network Virtualization and Intelligence for Future Internet*, Apr. 2010.

[39] 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.

[40] D. Menezes, N. Fernandes, C. Gomes, and O. Duarte, "Developing NOX applications for network control," in *WNetVirt'10 : Proceedings of the First Workshop on Network Virtualization and Intelligence for Future Internet*, Apr. 2010.

[41] C. Gomes, D. Menezes, N. Fernandes, and O. Duarte, "A tool for OpenFlow network management," in *WNetVirt'10 : Proceedings of the First Workshop on Network Virtualization and Intelligence for Future Internet*, Apr. 2010.

[42] R. Sherwood, M. Chan, A. Covington, G. Gibb, M. Flajslik, N. Handigol, T. Huang, P. Kazemian, M. Kobayashi, J. Naous, *et al.*, "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.

[43] J. Moy, *OSPF Version 2*. IETF, RFC 2328, 1998.

[44] "802.1ab IEEE standard for local and metropolitan area networks. station and media access control connectivity discovery," tech. rep., IEEE Institute of Electrical and Electronics Engineers, 2005.

[45] G. Gibb, "Basic spanning tree." http://www.openflowswitch.org/wk/index.php/Basic_Spanning_Tree. (accessed January 2010).

[46] R. Zhou and E. Hansen, "Breadth-first heuristic search," *Artificial Intelligence*, vol. 170, no. 4-5, no. 4-5, pp. 385–408, 2006.