Horizon Project

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

Horizon - A New Horizon for Internet

WP4 - TASK 4.2: Overall System Architecture Design
(Annex K)

Institutions

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

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

Contents

1 Introduction **4**
1.1 Document Outline 6

2 Overall Architecture Design **7**
2.1 Xen Architecture View 7
2.1.1 Resource Managers 10
2.1.2 Admission control of new virtual routers 12
2.1.3 Plane separation modules 17
2.1.4 Secure communication module 17
2.1.5 QoS Provision 18
2.1.6 Analysis 20
2.2 OpenFlow Management Architecture 25
2.2.1 Evaluation of the OpenFlow Management architecture 27

3 A hybrid Xen and OpenFlow system architecture design **30**
3.1 Xen and OpenFlow virtualization platforms pros and cons . . 32
3.2 XenFlow architecture design 34
3.2.1 Plane separation and route translation into flows . . . 35
3.2.2 XenFlow virtual topology migration 37
3.3 Experimental results 38

4 Multi-Agent System for Self-Management of Virtual Networks **42**
4.1 Autonomic networks 43
4.2 Proposal of a multi-agent architecture 44
4.3 Implementation 46
4.4 Experimental results 48

5 Conclusions and Ongoing Work **54**

Bibliography **61**

List of Figures

2.1 Horizon architecture design using the Xen platform. 8
2.2 View of physical nodes using the Xen platform and VNEXT. . 9
2.3 Example of substrate histogram shifting, when $N_{int} = 10$ and $\Delta = 1$. 15
2.4 Probability mass function of the shifted substrate histogram shown in Figure 2.3(b). 15
2.5 Creating the secure channel between Dom0 and DomU. 18
2.6 QoS provision inside each virtual network (QoS-operator) and among virtual networks (QoS-provider) in proposed architecture. 19
2.7 Admission control assuming virtual networks with traffic modeled by a Poisson process and maximum blocking probability of 5%. 22
2.8 Admission control assuming virtual networks with on-off traffic and maximum blocking probability of 5%. 23
2.9 Admission control assuming virtual networks with increasing traffic and maximum blocking probability of 5%. 24
2.10 RTT according to the QoS parameters inter virtual networks, assuming that Network 1 is prioritized. 25
2.11 Horizon architecture design using the OpenFlow platform. . . 26
2.12 Migration of the flow x from path A-B-D to path A-C-D. . . . 27
2.13 Results of migration experiment calling migration function by an agent and comparison of control overload between NOX and OMNI. 29

3.1 OpenFlow network virtualization, in which a migration is the redefinition of the network flows in another set of switches. . . 33
3.2 Xen network virtualization, in which a migration is equivalent to a virtual router migration. 33
3.3 Architecture of a XenFlow network element. 35
3.4 Example of a XenFlow network composed of virtual switches and virtual routers. 36

3.5 XenFlow routing, in which packets are directly forwarded by Domain 0. 37
3.6 Three steps of the XenFlow virtual topology migration. 38
3.7 Control plane downtime. 40
3.8 Total Migration Time. 41
3.9 Amount of packet losses as a function of the transmitted packet rate. 41

4.1 Autonomic manager architecture. 44
4.2 Information model of the knowledge base. 45
4.3 Testbed built to validate the multi-agent system for self-management of virtual networks. 46
4.4 Graphical interface of the multi-agent system. 48
4.5 Experiments without the multi-agent system. 49
4.6 Experiments with the multi-agent system and Iperf. 51
4.7 Experiments with multi-agent system and SCP. 52

Chapter 1

Introduction

The Horizon Project is aligned with current research community initiatives which rethink the Internet architecture to address the current issues and support emerging requirements. The Future Internet projections lead to a model in which the infrastructure must support several parallel networks, each one with its own protocol stack, and its own management framework. This model assures great flexibility on the network core. The Horizon Project proposes a virtual router architecture, which supports network virtualization, in order to enable the creation of a high programmable network core, which fits well the next generation network requirements and supports innovation on the network core. Our main goal is designing virtualized infrastructures with increased openness via a flexible, layered architecture using current virtualization technologies. We consolidate the overall system architecture into three main approaches, based on the Xen hypervisor, the OpenFlow switch architecture, and a Xen/OpenFlow combined architecture that integrates machine and network virtualization techniques.

The main objective of this work package is to establish a solid foundation, create the environment of the piloting system, and specify the service control requirements for designing policy-based architectures as the basis for the work package activities. Specifically, the main objective of the task 4.2, addressed in this report, is to develop a network management framework for Post-IP networks that correlates and optimizes each network element in order to form a global network with self-management properties. The work defines a thin management layer within virtualized substrates that is autonomic, capable of situated awareness, learning, inferring, and detecting faults for adaptive monitoring in order to provide the system self-* properties. Besides, in this task, we develop the piloting system, the interworking and the integration of separate self-control functions coming from both the autonomous network elements and the global network level elements.

We introduce a general architecture view of the developed modules, which allows an autonomic and intelligent control of the network. We define the integration of the developed tools in both Xen and OpenFlow platforms. Hence, we now present how the designed modules interact with each other in an efficient fashion. It is worth mentioning that the proposed holistic architecture design facilitates the understanding of which control and management algorithms fit better each platform, creating an autonomous platform.

We implemented the first step for integrating the developed modules in both Xen and OpenFlow, giving the basis for creating the design of a coherent prototype which is secure and presents a high performance. This prototype allows the development of new networks to solve specific problems, such as the provision of mobility, security, quality of service, etc. The developed prototype ensures network isolation and conformance with the service level agreements established between network operators and infrastructure providers.

We also propose a third architecture, which mixes Xen and OpenFlow platforms to sum the advantages of both machine and network virtualization platforms. OpenFlow networks give a wide support for flow migration and virtual network remapping without packet losses, while Xen provides a more flexible packet processing due to the use of virtual machines. The idea is to use OpenFlow to manage flows while Xen provides the platform for controlling the network and routing packets. This approach is called XenFlow and presents many advantages as network virtualization platform. XenFlow is able to use the features developed for both Xen and OpenFlow, achieving a flexible and complete network management tool set.

Finally, we present a distributed architecture for management of virtual networks and a piloting system prototype. The main goal of this architecture is enabling substrate network to self-manage virtual networks. It is part of the piloting plane, described in Work Package 3, and it is based on autonomic networks. The autonomic managers of the network elements have a closed control loop of monitoring, analyzing, planning, and executing that feeds a knowledge base for next iterations. In this stage of the project, we also deploy a testbed to validate our architecture. Our testbed is composed of virtual machines working as routers over a physical network infrastructure. We developed a prototype of our architecture with a multi-agent system based on the Ginkgo platform. The testing scenarios focus on the self-healing of virtual networks, but the distributed architecture for self-management of virtual networks is generic and could be used to other functional areas in autonomic computing: self-configuration, self-optimization, and self-protection. Some experiments were carried out to assess the performance of the recovery process.

1.1 Document Outline

This report is organized as follows. Chapter 2 provides the overall architecture view for both the Xen and the OpenFlow platforms. It provides a detailed view of how the proposed algorithms interact with each other as well as how virtualization management tools are used by the control algorithms. Auxiliary functions, such as plane separation and secure communication are also described. Chapter 3 presents the XenFlow architecture design, which is a hybrid architecture that combines Xen and OpenFlow virtualization platforms. This chapter presents XenFlow architecture and also a performance analysis that demonstrates the main features of this new platform. The Chapter 4 presents our self-management system prototype, described in the report related to the task of workpackage 3.2, in which the concepts of autonomic networks were applied in a virtualized environment through a multi-agent system. In this autonomic self-management environment, experiments were performed with a focus on self-management failure, or self-healing, of virtual networks. Finally, Chapter 5 presents the conclusions and directions for future work.

Chapter 2

Overall Architecture Design

In this chapter, we present the overall architecture design of the developed prototype for both the Xen and the OpenFlow virtualization platforms. We show which algorithms concern each platform and how tools interact. We also provide a view of some modules that were designed to create a consistent tool set for the network. Both prototypes were implemented and are now under test to improve the overall performance, guaranteeing a good integration among modules.

2.1 Xen Architecture View

In the network virtualization paradigm, different virtual routers share a physical router in order to provide different network services simultaneously. Key aspects of this paradigm are isolation and performance on packet forwarding. Isolation ensures independent virtual network operation, preventing malicious or fault virtual routers interference in the operation of other virtual networks. Xen [1] is a virtualization platform that can be used to create virtual routers, each with its own operating system and protocol stack, in commodity computers. The Xen platform, however, does not provide complete isolation and also presents low performance on handling network input/output (I/O) operations [2].

We now present an architecture design for monitoring and ensuring isolation, security, and high performance in each physical node of virtual networks. This architecture is represented in Figures 2.1 and 2.2. Figure 2.1 shows the entities relationship in the proposed architecture for Xen virtual networks, whereas Figures 2.2 shows the software architecture inside each physical node. We opted for a modular architecture to guarantee an easy upgrade of a tool without disturbing other services. Moreover, it is easy to

add new features to our model. The basis for the developed modules is the Virtual NEtwork management for Xen-based Testbeds tool (VNEXT) [3, 4], which aggregates all the proposed control modules.

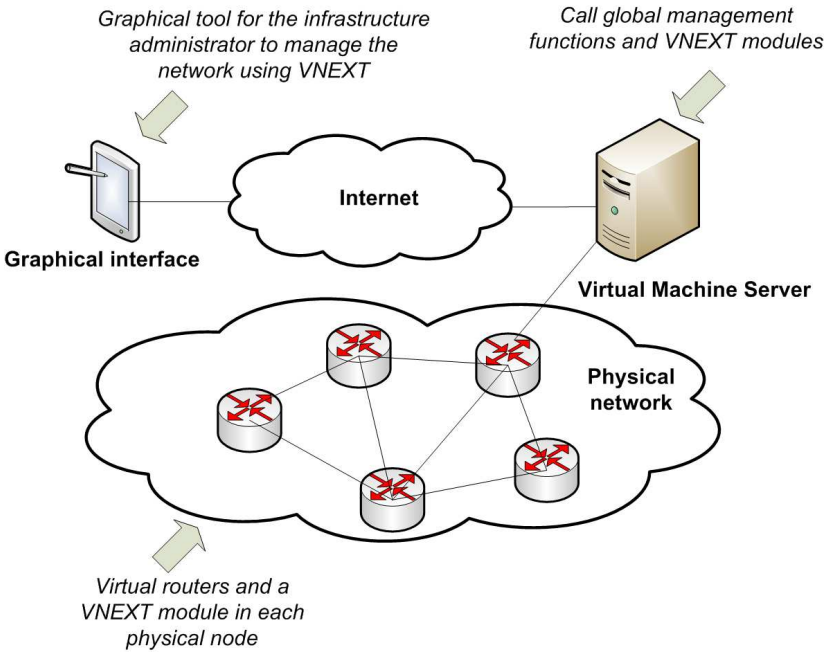


Figure 2.1: Horizon architecture design using the Xen platform.

Our node architecture is based on a client-server model in which virtual machines are clients of the server in Domain 0. It is important noticing that the available services are optional and can be selected according to the administrator requirements. Also, there are services that run only on Domain 0, without interfering in the virtual machine.

Most of the client-server functions used in our prototype are based on the plane separation paradigm [5], in which data forwarding and control mechanisms are decoupled. Thus, control, such as routing, is accomplished inside the virtual machine (DomU), ensuring flexibility in the design of control mechanisms, while packet forwarding is performed in a privileged domain, called Domain 0 (Dom0), providing quasi-native performance. The main advantage of plane separation is the provision of quasi-native packet forwarding performance and the main drawback is lower flexibility in packet forwarding, because the infrastructure administrator imposes a common forwarding plane for all virtual machines. Hence, we offer the virtual network operator the option of using or not the plane separation paradigm. The plane separation module is responsible for creating a valid copy of the forwarding table of

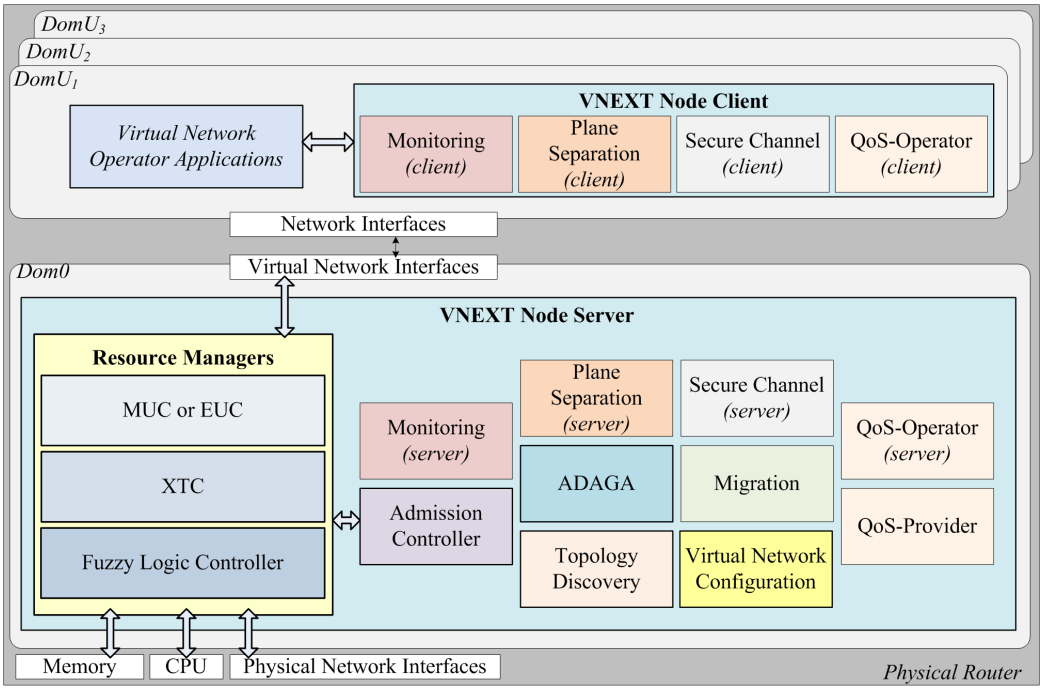


Figure 2.2: View of physical nodes using the Xen platform and VNEXT.

the virtual machine inside Dom0. The QoS operator module creates a copy of the forwarding rules of the virtual machine, such as queuing disciplines and filtering rules, inside Dom0. Hence, these modules accomplish the transfer of the data plane from the virtual machine to Dom0. Both modules use the secure channel module, to ensure a secure communication between DomU and Dom0.

The monitoring module is also based on a client-server model and provides a set of monitoring data for Dom0 and for each virtual machine. This module was described in the technical reports of Work Package 2, as well as the modules for migration [6], topology discovery, and virtual network configuration. All these modules compose the basic tool set for controlling network virtualization.

The ADAGA (Anomaly Detection for Autonomous manaGement of virtuAl networks system) module detects anomalies in the physical and in the virtual nodes. This module was described in Work Package 3. ADAGA interacts with the monitoring module to obtain data about DomUs and the Dom0 and performs actions for creating filtering rules to monitor the virtual networks.

The main modules of the architecture are the resource managers, which dynamically control the resource allocation to the virtual machines. We

developed three main modules (MUC/EUC, XTC, and Fuzzy controller), which were described in Work Package 3. We also added a new module, the admission controller, which operates in parallel with the resource managers to locally judge if new virtual networks can be hosted in the same physical node. This module gives support to global virtual network allocation functions.

Two new modules were added to node architecture, the QoS-Operator and the QoS-Provider. The QoS-operator allows the virtual network operator to set QoS parameters inside the virtual network, even if plane separation is in use. The QoS-provider differentiates the service provided by each virtual network, allowing the infrastructure administrator to set different privileges for each virtual network. Hence, these modules give the basis for developing the QoS parameters, which is one of current requirements of the Internet, as pointed out in Task 4.1.

In the following, we provide more details about each module, specially the admission controller and the QoS modules, and how they interact with each other.

2.1.1 Resource Managers

Our prototype assumes the use of one up to three resource managers, depending on the resources that the infrastructure administrator wants to monitor. Each manager defines a different policy and controls a different parameter in the virtualized system.

The first resource manager module selects a resource policy, according to the selected controller. There are two options: the Maximum Usage Controller (MUC) [7, 8] and the Efficient Usage Controller (EUC) [9]. MUC reserves a minimum amount of resources to each DomU and shares the remaining resources among all DomUs. Hence, all the available resources are offered to the system characterizing the maximum usage of resources. There is no bound of resource usage for the virtual networks. Therefore, this police allows that a virtual network uses more resources than the amount specified in the Service Level Agreement. The sharing of the remaining resources follows a parameter set by the infrastructure administrator to differentiate virtual networks.

EUC specifies a minimum resource reservation rate and a maximum volume of resources to be provided in a long time interval to each virtual network. The controller dynamically adjusts the resource reservation parameters of each virtual network according to the reservation parameters and the network demand. Hence, EUC specifies the maximum amount of resources each virtual network can use, providing a more precise resource reservation policy than MUC.

Both the MUC and the EUC are correlated to the monitoring module, which provides the required information to perform the resource allocation control. The resource manager also interacts with the Virtual Network Configuration, to obtain and modify the resource allocation parameters of the virtual networks. The Virtual Network Configuration module is initially set by the VNEXT Virtual Machine Server, explained in Work Package 2, which exports the parameters selected by infrastructure provider during network creation or management to each physical node, guaranteeing an accurate behavior in the controller modules.

The Virtual Machine server allows the infrastructure administrator to set the resources parameters of MUC or EUC for each node in a virtual network through the graphical interface. The default configuration assumes the same parameters for all nodes, but it also allows per node configuration.

The second resource manager module is XTC (Xen Throughout Control) [10, 11]. XTC is a mechanism to control the bit rate that a virtual machine forwards packets, reducing its impact on Domain 0. Indeed, XTC dynamically adjusts Xen scheduler parameters to give more or less CPU time to each virtual network in order to increase or reduce its packet forwarding rate. The description of XTC and its algorithms can be found at Report 3.1. As XTC limits the bit rate that a virtual router sends, XTC is only effective when the plane separation of VNEXT architecture is not used. Like all VNEXT modules, the actions performed in XTC are done by the Virtual Machine Server. For this, each physical machine executes a daemon, called XTC Manager, that can be connected to the Virtual Machine Server through TCP sockets. The Virtual Machine Server is thus responsible for receiving the user requests and making the appropriated action by exchanging messages with XTC Manager. The XTC actions performed by the Virtual Machine Server can be done using the Graphical User Interface of VNEXT. This module provides a user-friendly interface of XTC making requests through the Virtual Machine Server.

The third resource manager is a fuzzy logic controller [12, 13]. This controller presents a complimentary functionality to EUC and MUC, allowing the infrastructure provider to control other parameters rather than the shared CPU, memory, and bandwidth in Domain 0, such as the robustness to failures and the machine temperature. The idea is to provide an efficient control system for Service Level Agreements (SLAs) in the virtual network environments. The proposed system verifies the physical resource usage, retrieves real-time profiles of virtual routers, and guarantees the SLA requirements. The control is based on nebulous logic and determines the resource allocation according to the system overload and to the profile of routers. The control logic punishes virtual networks that exceed the established SLA. The

punishment depends on the exceeding value and on the system charge.

This controller also uses the monitoring module and generates profiles, represented by histograms, to compactly store the relevant system information. More information about this module is on Report 3.3.

2.1.2 Admission control of new virtual routers

The virtual network admission controller arbitrates the access of new virtual routers to the physical machine. The number of virtual routers hosted in a physical machine influences the provision of long-term volume reservations. Indeed, EUC defines two main types of resource reservation (Report 3.1):

- The *short-term rate reservation* ($R_s[n]$), which is the rate of resources that must be met for the virtual network n whenever there is demand in a short interval I_s.
- The *long-term volume reservation* ($V_l[n]$), which is an amount of resources $V_l[n] = R_l[n] \cdot I_l$ that should be guaranteed during a long interval I_l, where $R_l[n]$ is the average rate of the long-term volume reservation of network n.

Indeed, the admission control mechanism is non-trivial because different network profiles may use the same amount of long-term resources. Then, the difference among network profiles, which include static information, such as disk and memory sizes, and dynamic information, such as throughput and CPU consumption, should be considered for calculating the resource blocking probability. The resource blocking probability is the probability of a virtual network service request be denied due to an overload in the physical resource. The key idea of the proposed admission controller is to estimate the probability of blocking a resource demand before admitting a new virtual network. The controller estimates a value to compensate demand increases in the already hosted networks.

2.1.2.1 Monitoring and storage

The proposed admission control stores a set of histograms representing the physical-substrate resource usage. A new histogram is generated for each dynamic resource in each monitoring period. A set of histograms of the same resource models the load variation along a period of time (e.g., a day).

Physical substrate histograms model the behavior of the aggregate resource usage of virtual networks. The idea is that networks with different traffic profiles lead to different aggregate resource histograms, even if the

short and long term reservations are the same for all virtual networks. Thus, two networks can have the same V_l and R_c and have completely different behaviors, which imply in different aggregated resource demands. Thus, the physical substrate histogram realistically models what is happening with the physical network resources at each monitoring interval.

A monitoring interval is defined as $K_{adm} \cdot I_l$ seconds, where K_{adm} is an arbitrary constant chosen by the infrastructure administrator. To avoid a storage and processing overload, the proposed system randomly selects K_{rand}, which is the number of long intervals that will not be evaluated after K_{adm} long intervals. Moreover, instead of storing all histograms, the admission controller checks the difference between the current and the last histogram. For this, the admission controller normalizes both histograms and calculates the biggest error in the y-axis (e_m) between the two histograms. If condition $|e_m| > E_{adm}$ holds, where E_{adm} is a threshold specified by the infrastructure administrator, then the current profile is stored and a new histogram is started.

The controller algorithm estimates the blocking resource probability after a new virtual network joins the physical substrate. This estimated probability is used as a criterion for accepting or not a new virtual network. The algorithm inputs are the substrate histograms, the long-term volume reservation of each virtual network ($V_l[\ ] = R_l[\ ] \cdot I_l$), and the average resource usage of each virtual network, $R_{avg}[\ ]$.

The admission control algorithm is accomplished in four steps:

- **Step 1 -** Estimate the aggregate resources usage through a histogram.
- **Step 2 -** Estimate how a demand increasing would impact in the aggregate resource usage (assuming that the reservations are respected).
- **Step 3 -** Estimate a probability function to model the new virtual network resource usage.
- **Step 4 -** Calculate the resource blocking probability if the new network were using the physical substrate.

2.1.2.2 Step 1

Step 1 consists of monitoring resource usage and storing histograms as described in Section 2.1.2.1. Thus, in the end of this step, the algorithm knows the substrate histograms for bandwidth, CPU, and shared memory of each monitoring interval.

2.1.2.3 Step 2

In this step, the proposed algorithm estimates the substrate histograms if all virtual networks were using all reserved resources. The idea is to ensure that there will always be physical resources to meet all the virtual networks requirements, even if there are simultaneous peak demands. Solutions based on migration [14] are usually slow, because they depend on observing the resource blocking for a period of time, then searching for a new mapping between virtual and physical topology, avoiding overcharged physical nodes, and finally migrating the selected virtual nodes. Besides, these solutions can cause packet losses, leading to penalties for the infrastructure provider. Migration-based solutions without the use of an appropriate admission control may also overcharge physical nodes, causing more losses and oscillations in the mapping of virtual networks over the physical substrate. Therefore, it is important to estimate the impact of a new network before admitting it into a physical node. The key idea of our proposal is to estimate the resource blocking probability as if the new virtual network was using the physical network and all hosted networks were using fully using their reservations.

The proposed algorithm uses $R_{avg}[\,]$ and $R_l[\,]$ of each monitored resource to estimate the demand increases, represent as

$$\Delta = \sum_{n=1}^{N_H} R_l[n] - R_{avg}[n], \tag{2.1}$$

where N_H is the number of virtual networks. The Δ estimates the impact of a demand increase over the physical substrate. As a basis for this estimate, we assume that virtual networks will request all reserved resources and that the demand increase does not change the way resources are required. Thus, if the function $f_i(t)$ models the resource usage of virtual network i over time, then the estimate of the increased resource usage of this network would be given by $f_i(t) + \Delta_i$, where $\Delta_i = R_l[i] - R_{avg}[i]$. Nevertheless, it is irrelevant for the proposed algorithm the way each network increases its consumption individually, but the way the aggregated usage increases when each virtual network is using its whole reservation. After obtaining Δ (Eq. 2.1), the controller updates each substrate histogram. First, the histogram is shifted according to Δ. Assuming that $I[i]$ is an histogram interval with upper bound $L_s[i]$ and $I[i']$ is an interval that contains the value $L_s[i] + \Delta$, then

$$L_s[i' - 1] < L_s[i] + \Delta \leq L_s[i']. \tag{2.2}$$

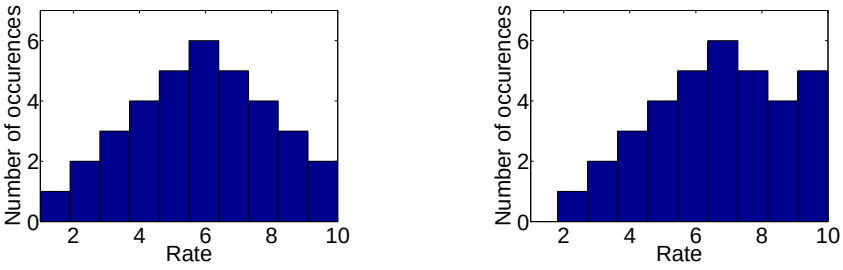
Furthermore, we assume that $H_{sub}(I[i])$ is the number of event occurrences in the interval $I[i]$ in the substrate histogram and that N_{int} is the number

of intervals in substrate histogram. Thus, the shifted substrate histogram, H_{sft}, is computed as $H_{sft}(I[i']) = H_{sub}(I[i])$. The interval $I[N_{int}]$ of H_{sft} is defined as

$$L_s[N_{int} - 1] < I[N_{int}] < \infty, \tag{2.3}$$

to maintain a fixed number of intervals. Figure 2.3 shows an example of histogram shifting when $N_{int} = 10$ and $\Delta = 1$.

After that, the controller calculates a probability mass function for the shifted substrate histogram (PMF_{hist}), assuming that the values of the x-axis are given by $L_s[s]$, as shown in Figure 2.4.



(a) Substrate histogram (H_{sub}).

(b) Shifted substrate histogram (H_{sft}).

Figure 2.3: Example of substrate histogram shifting, when $N_{int} = 10$ and $\Delta = 1$.

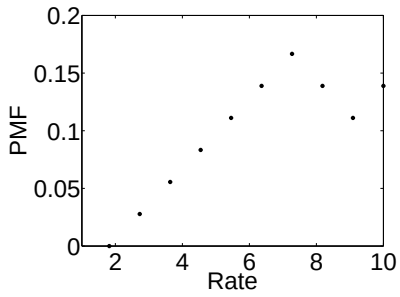


Figure 2.4: Probability mass function of the shifted substrate histogram shown in Figure 2.3(b).

2.1.2.4 Step 3

The controller estimates the demand of the new virtual network based on a predefined distribution. For example, an optimistic admission controller assumes a Poisson distribution, while a pessimistic admission controller assumes a distribution concentrated at peak rates.

The distribution estimated for the new virtual network (PMF_{new}) is represented according to the intervals $I[n]$ of the substrate histogram, assuming that the values of x-axis are given by $L_s[\,]$.

2.1.2.5 Step 4

The controller algorithm estimates the resource blocking probability after the requesting virtual network joins the substrate for each dynamic resource, i.e. CPU, bandwidth, and memory. This probability is used as criteria to accept or not the new virtual network.

The proposed algorithm uses PDF_{hist} to calculate the resource blocking probability, according to the distribution assumed for the new virtual network (PDF_{new}). Considering that

$$A = \{(x_1, x_2) \mid x_1 + x_2 \geq C\}, \quad x_1, x_2 \in [L[0], L[N_{int}]], \tag{2.4}$$

is a set of tuples (x_1, x_2) whose sum $x_1 + x_2$ exceeds the resource capacity C, then the controller computes the resource blocking probability by

$$P_B = \sum_{\forall (x_1,x_2) \in A} PDF_{hist}(x_1) \cdot PDF_{new}(x_2). \tag{2.5}$$

Therefore, the proposed algorithm estimates the probability of the sum of the resources used in the shifted substrate histogram and in the new network exceeds the physical machine capacity.

The new virtual router is accepted if there are enough resources for the static requirements and if the conditions

$$\sum_{n=1}^{N} R_s[n] < C \qquad \text{and} P_B < P_L \tag{2.6}$$

hold for all histograms of all dynamic resources, where P_L is the resource blocking probability accepted by the infrastructure administrator. A small P_L guarantees a low probability of packet losses. Nevertheless, it also reduces the physical resource efficiency, which reduces the infrastructure provider profits.

2.1.3 Plane separation modules

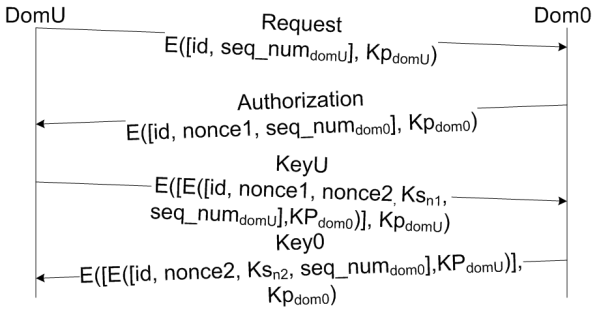
As mentioned earlier, for a higher performance, it is essential to forward packets through Dom0. However, by separating the packet forwarding from the routing control, the virtual routers are unable to update their forwarding tables and their packet filters, because they have no access to Dom0 memory. Our system monitors tables and filters in DomUs and makes a replica of them on Dom0 through the data plane manager module. Therefore, the client in each DomU monitors changes in the forwarding table and packet filters, and the server in Dom0 maps both the forwarding table and the packet filter built in each DomU to Dom0.

Every change in the forwarding table or packet filter in DomU must be immediately updated in its replica in Dom0. For this reason, after every control message arrival, the data plane manager client checks for changes in the forwarding table and packet filters in DomU. If any difference is detected, the client transmits the changes to Dom0 via the secure communication module. When the data plane manager server receives a message notifying a forwarding table change, it searches for the settings of that DomU to find out in which Dom0 table to insert the change. In packet filter updates, the server modifies the received rule, inserting rule parameters that specify the characteristics of the virtual network. This procedure avoids that one virtual router creates rules in the packet filter that influences the traffic of other virtual routers.

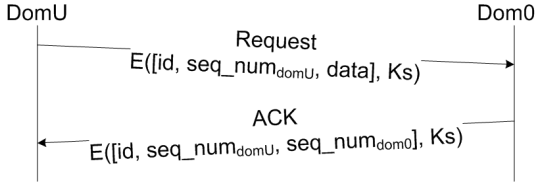
2.1.4 Secure communication module

The secure communication module creates a secure communication channel between the Dom0 and the DomUs, providing mutual authentication and privacy in data transfer. A secure communication channel is required because the data plane aggregates all forwarding tables of all virtual networks and, then, privacy and authentication is mandatory to guarantee the isolation. Since it is often used to update the data plane in Dom0, this must be a light module. Mutual authentication is required to ensure that no opponent domain can forge the identity of a common domain or of Dom0 to generate spoiled information in the data plane that corresponds to the attacked domain.

The secure communication module is composed of two protocols: one based on asymmetric cryptography for exchanging session keys (K_s), as described in Figure 2.5(a), and other based on symmetric cryptography for securely transmitting data between a DomU and the Dom0, described in Figure 2.5(b), where kp is the private key, Kp is the public key, $E([M], key)$



(a) Establishment of a session key, $Ks = f(Ks_1, Ks_2)$.



(b) Message exchange to update data plane.

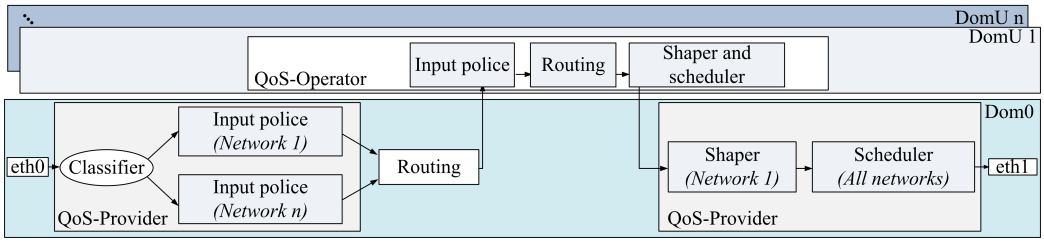
Figure 2.5: Creating the secure channel between Dom0 and DomU.

is the encryption of the message M with the key for symmetric or asymmetric cryptography, $Sign([M], k)$ represents message M and its signature with k, and id is the source node identity. These protocols avoid replay attacks, in which the opponent domain repeats old control messages to spoil information in the data plane of the attacked domain, using sequence numbers and nonces in the control messages. Nonces are randomly chosen numbers that should be used just once. Besides nonces, the proposed mechanism also changes the session key every time a sequence number reaches a maximum value to increase the robustness against replay attacks. Therefore, we can affirm that the communication between Dom0 and DomU is secure because our system checks the authenticity, the privacy, and the non-reproducibility of data.

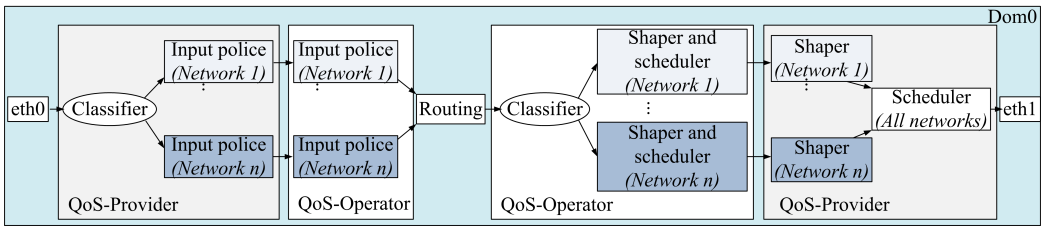
2.1.5 QoS Provision

One important advantage of the proposed architecture is the support for QoS provisioning. Although a virtual router that does not use the plane separation paradigm can set up a traffic control inside its virtual machine, a control at this level is not enough to ensure quality of service. A virtual router has no control over its own traffic while it is being forwarded by Dom0 from the physical device driver to the virtual device driver, which may in-

clude delays and packet losses. The I/O operations in Xen architecture are accomplished at Dom0, which is the driver Domain and, as consequence, DomUs do not have complete control of the forwarding operations. In the plane separation paradigm the data plane is in the Dom0 that offers a fine control of the forwarding functions and can also offer quality of service. Thus, the proposed architecture offers primitives for adding QoS rules inside a virtual network and among virtual networks, as shown in Figure 2.6(a) and 2.6(b), for packet forwarding through the virtual machine and through Dom0, respectively. Figure 2.6(a) describes in details the QoS modules of proposed architecture assuming packet forwarding through the virtual machine. According to this scheme, the infrastructure provider can configure priority access for the physical hardware to some virtual network as well as any virtual network operator can differentiate its own packets being processed inside the virtual machine.



(a) QoS provision assuming packet forwarding through the virtual machine.



(b) QoS provision assuming the usage of the plane separation paradigm, in which the packet forwarding is exclusively performed in Dom0.

Figure 2.6: QoS provision inside each virtual network (QoS-operator) and among virtual networks (QoS-provider) in proposed architecture.

The use of the plane separation paradigm implies that the data plane of each virtual network, which was previously inside DomU, is placed into Dom0. Thus, QoS provision must be adapted to work with the plane separation paradigm. As a consequence, the QoS provision for both the virtual network operator and the infrastructure provider must be supplied inside

Dom0, as shown in Figure 2.6(b). The proposed architecture provides an interface that allows each virtual router configuring its own QoS rules through the QoS-operator client and QoS-operator server components. These components guarantee that one virtual network can configure its QoS rules without interfering with the QoS rules of other virtual networks.

2.1.6 Analysis

In this section, we analyze both the access control and the QoS modules when used with the EUC controller. These modules were introduced in this work package to guarantee a proper architecture for virtual networks, which gives support to address the requirements described in Report 4.1.

2.1.6.1 Admission control module evaluation

The virtual network admission control for the proposed architecture was implemented in C++. The proposed algorithm is evaluated under different traffic patterns through simulations, assuming the use of the EUC controller, described in Section 2.1.1. We compare our proposal to other proposals of the literature, namely Sandpiper [14] and VNE-AC (Virtual Network Embedding Algorithm based on Ant Colony Metaheuristic) [15].

Sandpiper is a system for monitoring virtual machines load in data centers. When a physical machine is overloaded, which is identified when requests for resources of virtual machines are blocked, Sandpiper looks for a new virtual machine that is able to host one or more virtual servers of the overloaded node. The admission control in Sandpiper is based on the current virtual machine peak load and on the average amount of idle resources in the new physical machine. If the resource demand peak, which is estimated by the 95% rate in the cumulative distribution function, is less or equal to the average amount of idle resources, then the new virtual network is admitted.

VNE-AC is a virtual network mapping mechanism, which is also part of Horizon project, concerning the global virtual network control. This algorithm is based on the ant colony metaheuristic, which determines an adequate mapping of virtual networks over the physical substrate. The admission control mechanism proposed in the VNE-AC assumes that resources are statically attributed to each virtual network. Hence, this admission control restricts the SLA specification for each virtual network.

We implemented both Sandpiper and VNE-AC admission control to compare them to our mechanism. The Sandpiper peak rate is chosen as $p_k = 95\%$, as suggested by Sandpiper's authors. VNE-AC reservation threshold is set as R_l, which is the average rate of the long-term volume reservation (V_l) in

EUC. We also evaluate the impact of Δ (Eq. 2.1), which is indicated as 'Hist' in the results, to verify the efficiency of this procedure.

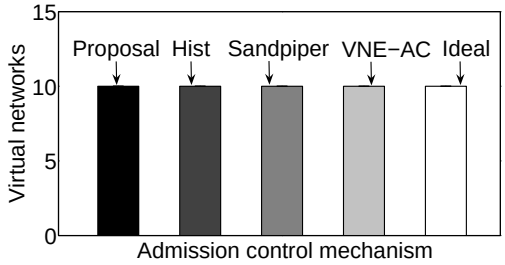
In this evaluation, we measure the number of virtual networks that each mechanism admits to be hosted in a physical router. We assume, for simplicity, that all virtual networks present the same resource reservation parameters and that the number of intervals in the histogram for our proposal is $N_{int} = 30$.

For evaluating the results, we also measured the ideal number of virtual networks in the analyzed physical node in each experiment. This ideal number is chosen as the maximum number of virtual networks that guarantees that the blocking probability threshold specified by the infrastructure administrator (P_L) is not violated. Since $p_k = 95\%$ for Sandpiper, we selected $P_L = 0.05$ for fairness. In one hand, if an admission control mechanism admits less networks than the ideal number, than the physical machine resources are wasted. On the other hand, if the admission control mechanism admits more virtual networks than the ideal number, then the physical node is overloaded and the reserved resources are denied for the virtual networks, generating fines for the infrastructure provider. Hence, admitting more virtual networks than the ideal value is worse than admitting less virtual networks.

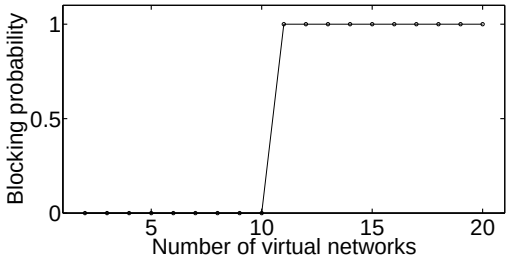
We run 30 rounds for each experiment, analyzing the output link throughput in packets/s. We present in the results the corresponding mean value and the standard deviation.

Traffic pattern impact - In the first experiment, we evaluate virtual networks whose traffic is modeled by Poisson processes. Each virtual network demands ≈ 100 Mb/s, which is also the value of R_l. Figure 2.7(a) shows that all mechanisms admitted the ideal number of virtual networks. Hence, if virtual network traffic presents a small deviation, then all the analyzed mechanisms are able to correctly perform the admission control. Figure 2.7(b) shows that admission control is this scenario is abrupt, because ten networks cause no blocking probability, while eleven networks cause a 100% blocking probability.

To evaluate the proposals in environments with a higher variability, we also simulated virtual networks with a traffic pattern described by an on-off model. These results are on Figure 2.8. In this experiment, all virtual networks present the same traffic pattern. The new virtual network traffic pattern is estimated based on a Poisson process in the proposed mechanism, but it behaves just as all the other virtual networks, with an on-off traffic. The on-off traffic is generated based on an exponential distribution with $\mu = 1/3$. Each value generated with the exponential distribution indicates a time interval in which the traffic will be on or off. The on-traffic is modeled



(a) Number of virtual network admitted by each mechanism.

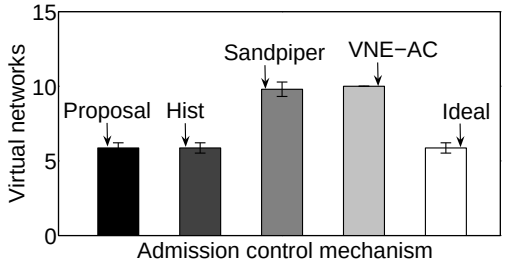


(b) Blocking probability according to the number of virtual networks in parallel.

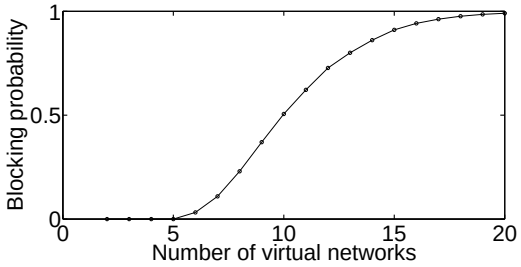
Figure 2.7: Admission control assuming virtual networks with traffic modeled by a Poisson process and maximum blocking probability of 5%.

by a Poisson distribution with $\lambda \approx 200$ Mb/s and the off-traffic is always zero. Figure 2.8(a) shows that Sandpiper and VNE-AC admits more virtual networks than the other proposals in the same physical router. Nevertheless, both mechanisms admit more virtual networks than the ideal number of virtual networks and, by Figure 2.8(b), we observe that the blocking probability for this number of networks is about 50%. Sandpiper overestimated the idle resources due to the on-off nature of traffic and allowed the admission of ≈ 10 networks. VNE-AC behaves similarly to Sandpiper, but for different reasons. VNE-AC does not differentiate the scenario of Figures 2.7 and 2.8, because real data is not used for predicting the demand. Hence, virtual networks with different data patterns are equally treated, generating admission control errors. Our proposal admitted the ideal number of networks, guaranteeing a low blocking probability and an efficient resource usage.

Δ Impact - This experiment evaluates virtual networks whose demand varies with time. In this scenario, the throughput of the virtual networks increases with time up to the threshold of the long-term volume reservation. The admission request for the new virtual network is sent when the



(a) Number of virtual network admitted by each mechanism.

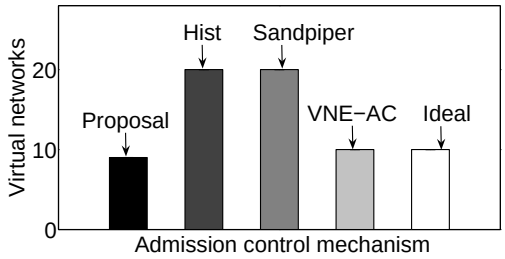


(b) Blocking probability according to the number of virtual networks in parallel.

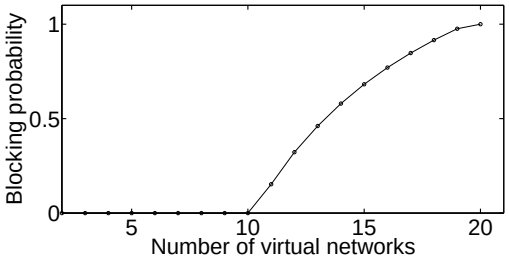
Figure 2.8: Admission control assuming virtual networks with on-off traffic and maximum blocking probability of 5%.

demand of the other virtual networks correspond to half of their reservation. Figure 2.9(a) shows that Hist and Sandpiper accepted all the analyzed networks, because these mechanisms do not consider the demand increase. Thus, the blocking probability is of 100% for both mechanisms when the demands increase up to their reservation threshold in the end of the simulation, as shown in Figure 2.9(b). Both our proposal and VNE-AC consider the demand variation and achieved a number of accepted virtual networks close to the ideal number.

Based on these results, the proposed admission control is the only one that is efficient in all analyzed scenarios, guaranteeing the admission of a high number of virtual networks, without violating the blocking probability threshold imposed by the infrastructure administrator. Sandpiper and VNE-AC are inefficient when network demand presents a high variability, overcharging the physical node.



(a) Number of virtual network admitted by each mechanism.



(b) Blocking probability according to the number of virtual networks in parallel.

Figure 2.9: Admission control assuming virtual networks with increasing traffic and maximum blocking probability of 5%.

2.1.6.2 QoS modules

Finally, we evaluate the QoS components of the proposed architecture. Figure 2.10 shows the impact of the use of QoS premises by the infrastructure provider. In this scenario, composed of two virtual networks, it is assumed that Network 1 hires a small forwarding delay for its traffic. It is also assumed the following parameters in the output link: $R_s[1] = 50$ Mb/s and $R_l[1] = 400$ Mb/s for Network 1 and $R_s[2] = 100$ Mb/s and $R_l[2] = 600$ Mb/s for Network 2. CPU and memory resources are equally divided between the networks. Network input demands are, respectively, $D[1] = 50$ Mb/s and $D[2] = 1$ Gb/s. We chose a high $D[2]$, because a high volume of traffic hinders the provision of QoS for Network 1. The traffic of Network 2, which uses the plane separation paradigm, flows among two external machines. The traffic of Network 1, which is routed through the virtual machine, is generated by a different external machine and shares the output link with the traffic of Network 2.

Figure 2.10 shows the results for the Round Trip Time (RTT) of both networks with and without the proposed QoS support. In the first scenario,

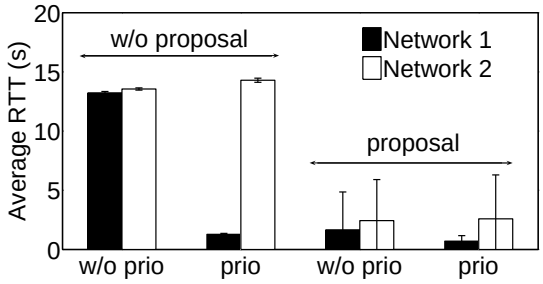


Figure 2.10: RTT according to the QoS parameters inter virtual networks, assuming that Network 1 is prioritized.

called 'w/o prio', both networks have the same priority and, in the second scenario, called 'prio', the traffic of Network 1 is privileged. When we give priority to one virtual network, even without the use of the resource sharing manager EUC, the RTT decreases by more than 10 times for the privileged traffic. The use of the QoS support ensures that Network 2 does not exceed the use of network or CPU, reducing the volume of processed data and thereby further reducing the transmission delay. Hence, the QoS module reduces the RTT by more than 18 times when compared to the scenario without our proposal and without priority and more than 1.8 times when compared to the scenario with privileged traffic but without the resource sharing manager EUC.

2.2 OpenFlow Management Architecture

OpenFlow has an architecture different from Xen. In OpenFlow networks, the control plane is centralized in one node, while the data plane is distributed over the physical OpenFlow switches.

We developed the tool OpenFlow MaNagement Infrastructure (OMNI), which is being used as the basis for integrating OpenFlow management modules [16, 17]. The overall architecture overview is on Figure 2.11.

Due to the centralized nature of the data plane, the proposed architecture to OpenFlow networks is simpler. First, we do not require any modifications in the forwarding nodes. We respect the condition of a simple node that is shared by all virtual networks without even noticing that the network was virtualized. Hence, all proposed functions are restricted to the controllers and the FlowVisor.

The original version of OMNI was designed to work with any OpenFlow controller, providing network view and intra-virtual network management

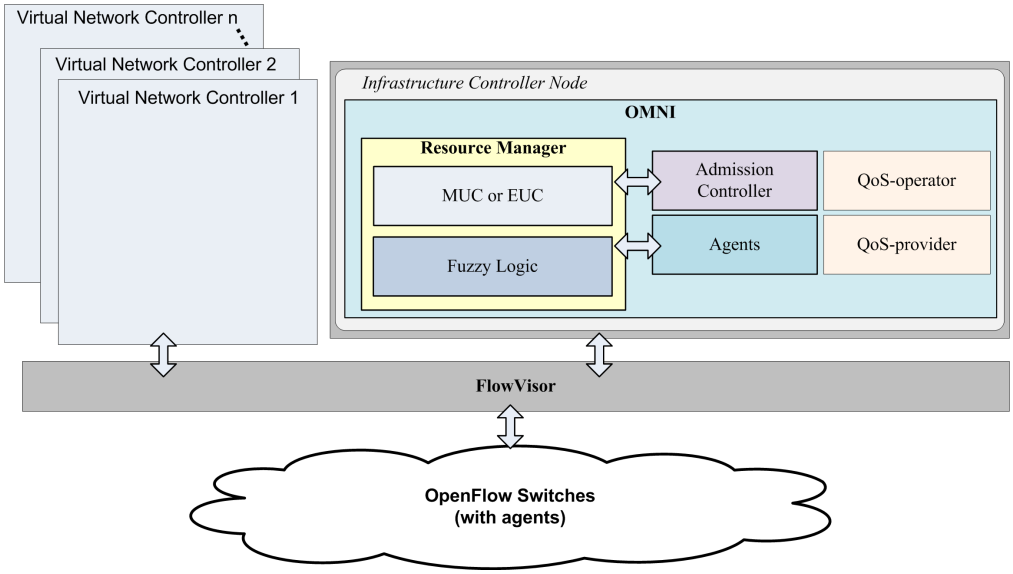


Figure 2.11: Horizon architecture design using the OpenFlow platform.

functions. The OMNI extended version works coupled with FlowVisor to provide management functions to the infrastructure administrator. Hence, this extended version runs algorithms for controlling physical resources that are shared by all virtual networks.

The new OMNI modules added in this work package are described on the right side of Figure 2.11. In this figure, we observe that all controllers interact with FlowVisor, which interacts with the OpenFlow network. The OpenFlow network is composed of OpenFlow switches. Optionally, we can use agents in these nodes to increase management performance as well as to add new monitoring functions. Agents could also be aggregated in middle boxes or even on the infrastructure controller node to provide in an easy way to perform distributed control functions.

FlowVisor is an interface provided by OpenFlow team to share the OpenFlow switches among many controllers [18]. FlowVisor offers an interface to configure physical switch resources, such as memory, queues, etc. It does not, however, provide control algorithms. Hence, we applied the same controller algorithms used in Xen to control the resources in OpenFlow, because these algorithms are platform independent. Hence, we can apply the logic of the modules EUC, MUC, fuzzy logic, admission control, QoS-operator, and QoS-provider in OMNI to control FlowVisor resources. We also add the agents to emulate a distributed control behavior.

2.2.1 Evaluation of the OpenFlow Management architecture

We evaluate OMNI to check its response time and operation. We also compare the number of control packets of OMNI and NOX original applications to estimate the OMNI control overhead. We deployed an experimental network using personal computers running OpenvSwtich software [5], an implementation of OpenFlow. OpenvSwitch works as a Kernel module and assures a high performance in packet forwarding. Our experimental scenario consists of four OpenFlow switches, a FlowVisor entity and a NOX controller, as shown in Figure 2.12. OpenFlow switches and the FlowVisor run on Intel Core 2 Duo computers, with 2 GB of memory. The controller runs on an Intel I7 computer with 4 GB of memory. On this computer, we also run Ginkgo agents, an agent for controlling each OpenFlow switch. We present the results with a 95% confidence interval.

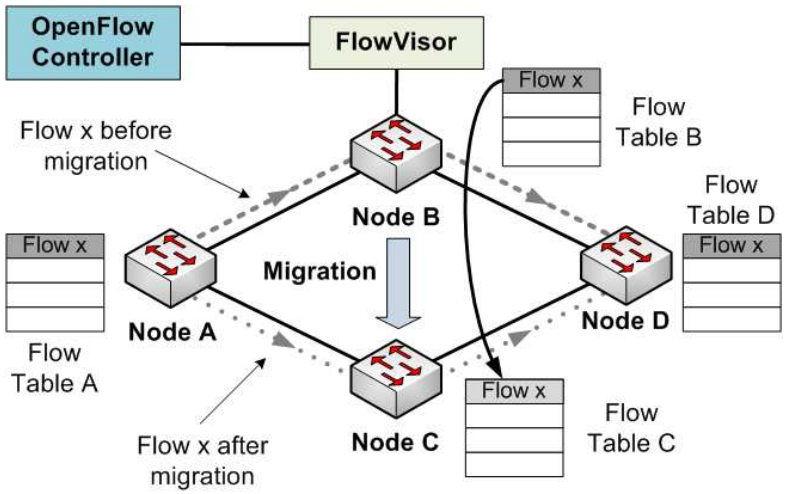
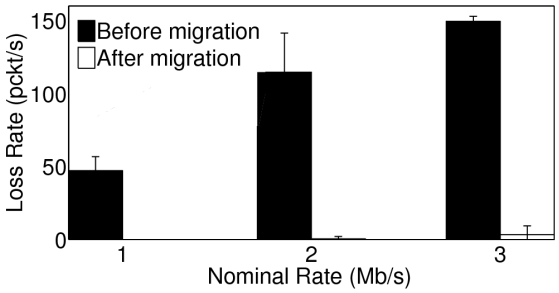


Figure 2.12: Migration of the flow x from path A-B-D to path A-C-D.

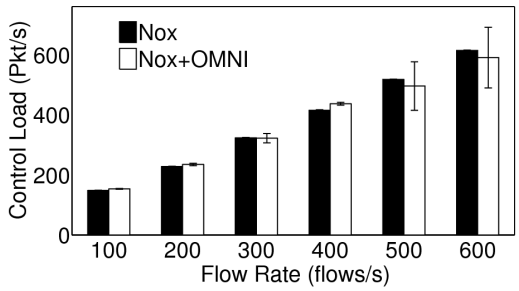
Our first experiment evaluates the migration performed by our multi-agent system. This experiment consists in migrating a flow from the path composed of A, B, and D switches to the path composed of A, C, and D switches, as shown on Figure 2.12. The probing traffic is a UDP flow with 1470 bytes of packet size and rate varying from 0.5 to 3 Mb/s. In this scenario, the throughput of the AB link is upper bounded by OpenFlow at 200 kb/s, while the other links, BD, AC, and CD, link are bounded by the link capacity of 100 Mb/s. Therefore, the original path loses packets when the transmission rate exceeds 200 kb/s. As the UDP flow transmission rate varies, we measure the packet losses until the agent autonomously trigger the flow migration. In

order to take the decision of a flow migration, the agent verifies the packet loss rate of its monitored switch, which must be higher than 200 packets/s, and also compares the local packet loss rate with loss rates exchanged with other agents. In that comparison, the local value should be at least 20 packets/s higher than the others. The packet loss rate and the comparison thresholds are agent parameters and are set according to each network. Our experiment is an example of agent usage. The developed agent senses the network at fixed intervals of 10 s and migrates a flow only after three consecutive observations where the packet loss rate is above the threshold. Figure 2.13(a) shows that the agent properly detects the bottleneck link within the flow path and then migrates the flow to avoid, or reduce, the packet loss rate. Since agents observe the loss rate on links at fixed time intervals, the time between starting the agents and the decision of migrating a flow is independent of the packet transmission rate. The agent triggers the migration on average 29.4 s after starting up.

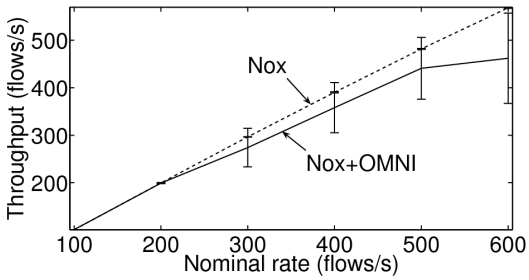
Flow instantiation is one of the main causes of control overhead in an OpenFlow network, because when a packet does not match any flow in a switch, the packet is forwarded to the controller and the controller sends a command to the switch. Thus, our next experiments evaluate the control overhead introduced by OMNI, and the effect of OMNI applications on flow instantiation. We measure the rate of control packets that traverse an OpenFlow network while new flows are instantiated using either NOX or NOX+OMNI. 'NOX' means a NOX controller running its original applications for collecting network statistics and for configuring packet forwarding. 'NOX+OMNI' means running all OMNI applications on the NOX controller. Since the NOX+OMNI flow instantiation mechanism is the same of NOX, the difference between the two curves is due to the statistic monitoring of NOX+OMNI. Figure 2.13(b) compares the control overhead for a varying flow instantiation rate. We observe that the difference between the two systems is negligible for instantiating up to 400 flows/s, as NOX and NOX+OMNI show almost the same control load. When instantiating 500 flows/s and above, the control load of NOX+OMNI is lower than that of NOX, but Figure 2.13(c) shows that NOX+OMNI is unable to instantiate as many flows as NOX. The flow instantiation rate achieved with OMNI is greater than the rate of NOX+OMNI. Also, the error bar size of NOX+OMNI increases because of the test instability. Indeed, NOX+OMNI is trying to process more data than the controller is able to handle. Since OMNI interval for monitoring each resource is configurable, increasing the interval reduces OMNI overhead. Thus, OMNI should achieve higher flow instantiation rate at the cost of increasing the granularity of statistics measures.



(a) Packet losses before/after running the agents.



(b) NOX and OMNI control load.



(c) NOX and OMNI flow instantiation rate.

Figure 2.13: Results of migration experiment calling migration function by an agent and comparison of control overload between NOX and OMNI.

Chapter 3

A hybrid Xen and OpenFlow system architecture design

Xen and OpenFlow present specific advantages and disadvantages. As shown in Report 2.2, OpenFlow presents simple flow configuration mechanisms, whereas Xen presents more flexibility in packet processing. Therefore, we propose a hybrid architecture to sum these advantages, obtaining a more powerful virtualization platform [19]. The management tools [17, 16, 4, 3] described in the previous sections also apply to this architecture with a few modifications.

The deployment of new protocols and services in the core of the Internet is rejected by most service providers, due to the high risk these changes represent for proper network operation and high costs involved in changing hardware platforms. A proposal to couple innovation support and production traffic is the network virtualization [20, 21]. Network virtualization allows that different networks, isolated from each other, share the physical substrate.

Network virtualization introduces a new management primitive, the migration of virtual networks [5]. The migration primitive is used in different contexts, such as maintenance of physical nodes and remapping the logical topology over the physical topology. Maintenance of physical nodes often requires shutting down the device. In case of routers, the maintenance period causes adjacency losses and, consequently, network failures during the convergence of the routing algorithm. The virtual network remapping is used for traffic management and for green networking, in which virtual nodes are reorganized considering the energy demand [22]. Migration can also be used to prevent damages, for instance, under a denial of service attack (DoS). In this scenario, virtual networks that share the same physical substrate with the network under attack are migrated to other nodes, preventing the overload in the input links. The migration of virtual topologies, however, presents

great challenges, such as the relocation of virtual links and the decrease of the damages caused by the service downtime during migration.

There are proposals [23, 6] that perform the migration of the logical topology in a transparent way to the network edges, without packet losses or broken connections. The scenarios in which these proposals are valid, however, are limited. In [23] and [6], the authors assume the existence of a mechanism for link migration that is independent of the mechanism of node migration. They also assume that one virtual router can only be migrated from one physical machine to another in the same local area network (LAN). Otherwise, tunnels between the physical machines must be created to simulate a LAN.

Flow migration in the OpenFlow platform is easy. Pisa *et al.* presents an algorithm that is based on the definition of a flow path in the OpenFlow network [6]. This proposal presents zero packet loss and low control overhead. Nevertheless, OpenFlow migration is not applicable to router virtualization or flow processing systems. The proposal is limited to switched networks.

This chapter introduces the XenFlow architecture, which is a hybrid network virtualization platform based on Xen and OpenFlow. Our proposal describes a flow processing system that allows migration of virtual networks, including the migration of both nodes and links. In this architecture, the plane separation paradigm is used and then the virtual router is divided into two planes, the control and the data planes. The control plane, which runs inside a Xen virtual machine, is responsible for updating the routing table given the routing protocol decisions. The data plane, which is implemented with OpenFlow, is responsible for forwarding the packets according to routing policies. The routing policies are based on defined routes, calculated by the control plane.

The main advantages of this network virtualization technique are:

- Plane separation with a highly flexible data plane.
- Migration without packet losses.
- Migration is not restricted to local network.
- Mapping of a logical link into one or more physical links.
- OpenFlow data plane, but using distributed network control.
- Node and link migration in the same procedure.

A XenFlow prototype was built to validate the system architecture design. Experimental results show that the system is robust to migration, in

the sense that, while migrating, there is no packet loss or routing service interruption. The system is efficient, as it allows the migration of virtual routers and links without connection loss or packet forwarding delay. When we compare XenFlow migration and Xen virtual machine native migration, XenFlow showed zero packet loss, while Xen native migration lost a significant amount of packets and presented a longer downtime during control plane update.

3.1 Xen and OpenFlow virtualization platforms pros and cons

OpenFlow is a switching technology that enables programming packet forwarding by associating actions with flows. A flow is defined as a set of up to twelve fields extracted from the frame header, which include link layer, network layer, and transport layer data [24]. The forwarding table of an OpenFlow switch is the Flow Table. The Flow Table relates a flow with one or more output ports of the switch according to the output actions defined by the centralized controller. The controller processes the first packet of a flow and, then, defines the actions. The Nox [25] is an OpenFlow controller that acts as an interface between the control applications and the OpenFlow network. As soon as a packet arrives at an OpenFlow switch, the switch checks if the packet matches any already defined flow. If so, the actions defined for that flow are applied to the packet. If not, the packet header is sent to the controller, which extracts the flow characteristics from the packet and creates a new flow in the Flow Table of the OpenFlow switch. The example of an OpenFlow network is on Figure 3.1. In OpenFlow networks, the migration is an easy primitive because it is performed by only reprogramming the Flow Tables in the switches.

The two main disadvantages of OpenFlow networks are that the control plane must be centralized and per-hop packet processing is a costly operation.

Xen is a personal computer virtualization platform largely used in and server consolidation. Its architecture is based on a virtualization layer, called Virtual Machine Monitor (VMM) or hypervisor. The Xen virtual environments are called virtual machines, or domains, and present its own resources, such as, CPU, memory, disk and network access. There is also a privileged virtual environment, called Domain 0, which has access to physical devices, and provides access to the Input/output operations from other domains. It also performs management operations in hypervisor. Figure 3.2 shows an example of network virtualization based on Xen. In this scenario, a virtual

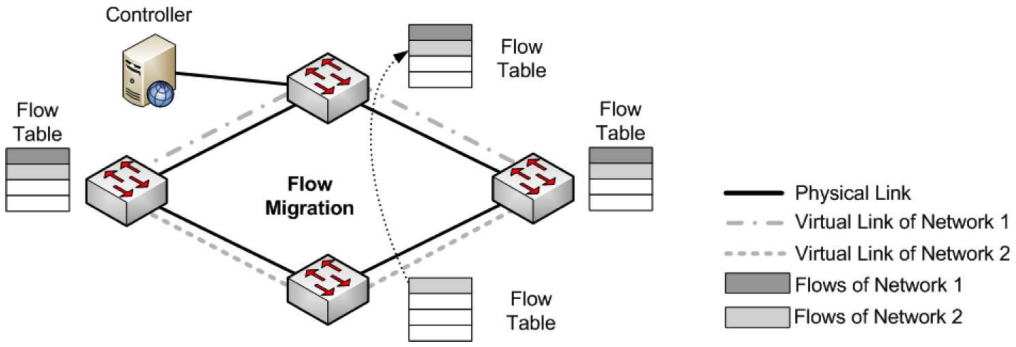


Figure 3.1: OpenFlow network virtualization, in which a migration is the redefinition of the network flows in another set of switches.

router migration is equivalent to migrate a virtual machine. As routers perform real-time service, a virtual router migration demands the minimization of the packet forwarding service downtime.

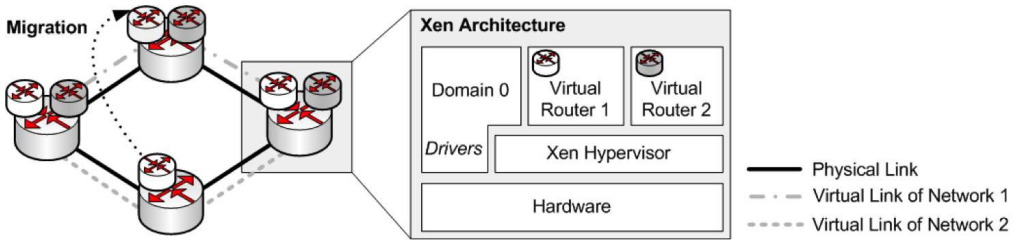


Figure 3.2: Xen network virtualization, in which a migration is equivalent to a virtual router migration.

Xen native migration [26] is based on virtual machine live migration. The virtual machine live migration consists into copying the virtual machine memory from the source physical machine to the destination one. This migration is called live because it tries to reduce the virtual machine downtime. Hence, the virtual machine is running during the first phases of the migration. As the pages of virtual machine memory in the source physical machine are changing as the migration procedure is going on, this procedure uses an iterative copy mechanism of memory pages. In the iterative copy, the modified memory pages are tagged, and, in the next iteration, they are copied to destination. This is repeated until the number of modified memory pages in the last round lower than a certain threshold. At this point, the virtual machine execution is suspended on the source physical machine, the last modified memory pages are copied to destination, and the virtual machine is, then, restored in the destination physical machine. A disadvantage of this

proposal for virtual router migration is the packet loss during the time that the virtual machine is unavailable, between suspension and restoring. This mechanism is limited to a local network, because this approach assumes the existence of a shared hard disk, and link migrations are performed by sending *ARP Reply* packets.

Plane separation can be used in order to avoid packet loss during migration. Pisa *et al.* proposes a virtual machine migration to Xen platforms that makes a copy of the data planes of all virtual routers to Domain 0 [6]. Thus, the data plane migration is performed without affecting the routing and without downtimes. The solution, however, is restrictive because a virtual router migration is performed only between nodes in the same LAN. Hence, the migration scope is limited to just one hop from the source router.

3.2 XenFlow architecture design

Our proposed architecture design combines the advantages of per-hop packet processing and the distributed control of Xen platform with flow processing capability provided by OpenFlow platform. The architecture of a XenFlow network element is shown in Figure 3.3. Each virtual machine hosts the control plane of a different network. The OpenFlow switch on Domain 0 performs packet forwarding, according to the forwarding rules specified by the virtual machines. In this architecture, a network element can be defined as a virtual switch (Layer 2), a virtual router (Layer 3), or a middle box (Layer greater than 3). The function performed by each network element depends on the virtual network protocol stack and applications.

In the XenFlow system, as well as in the Xen platform, physical device drivers are in Domain 0, and, then, all communication between virtual machines and physical devices must pass through the Domain 0. Thus, the Domain 0 multiplexes packets from virtual network elements to physical devices and demultiplexes packets from physical devices to virtual network elements [1]. In XenFlow scenario, the multiplexing and demultiplexing process is performed by an OpenFlow switch.

In XenFlow, packet forwarding is programmed according to the rules defined in the OpenFlow switch controller, which is an application running in Domain 0. This controller interacts with the virtual machines to discover the forwarding rules created by each data plane. If the virtual machine demands a per packet processing, the controller will direct all incoming traffic of that virtual network to the virtual machine. In addition to programming flows, this controller is also able to set policies to the flows, for instance, specifying a minimum bandwidth for each flow or for a set of flows [24].

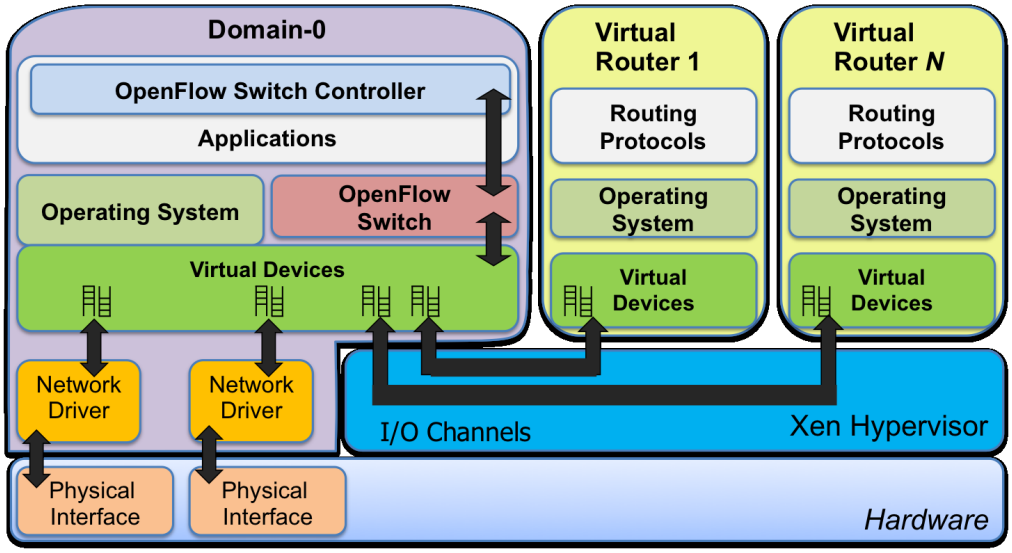


Figure 3.3: Architecture of a XenFlow network element.

Although OpenFlow presents a centralized network control, in XenFlow, a virtual network can opt between a centralized and a distributed control plane. To build a network with distributed control, a virtual router is instantiated in each physical node that belongs to that virtual network. Each virtual router interacts with XenFlow controller in Domain 0 to inform the forwarding rules. To build a centralized control, a middle box containing the controller is instantiated and the controller must interact with XenFlow controller in each Domain 0. In this case, there is no need to instantiate a virtual machine in each physical node, in a model very closer to traditional OpenFlow networks. Figure 3.4 shows the XenFlow architecture with three virtual networks, in which there are virtual switches and virtual routers interoperating. The physical switch data plane is shared using the OpenFlow protocol among different virtual networks.

Figure 3.4 shows the management entities that are aware of the physical network topology and are managed by the infrastructure administrator, which decides when to instantiate or delete a virtual network, as well as the amount of physical resources that each virtual network receives. These entities are also responsible for starting network migrations.

3.2.1 Plane separation and route translation into flows

Virtual switch migration is trivial when using OpenFlow switch. On the other hand, a virtual router migration on Xen platform is a more complex

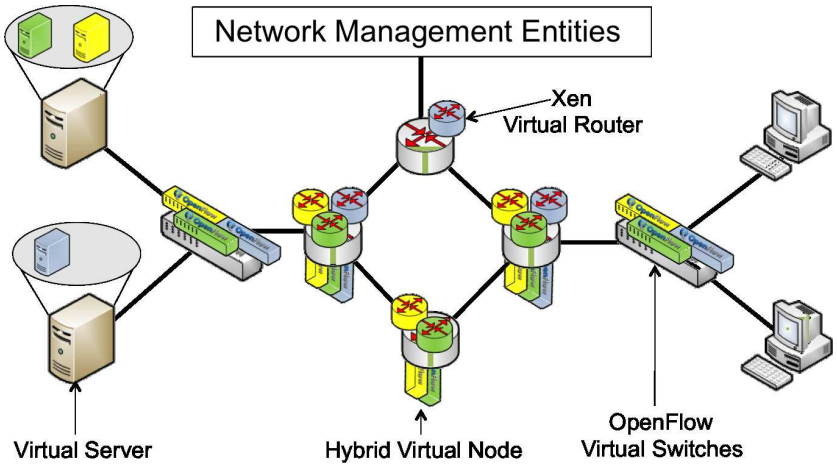


Figure 3.4: Example of a XenFlow network composed of virtual switches and virtual routers.

operation, because turning off a router implies in packet losses. Migration without packet losses is fundamental in router migration and this goal is reached in XenFlow due to the use of the plane separation technique.

The forwarding table of each virtual network is computed inside the virtual router, which is placed inside a virtual machine. This information must be transmitted to Domain 0 to build the correct forwarding rules in the OpenFlow switch. Figure 3.5 shows how XenFlow performs this task. A daemon running inside the virtual machine copies the data plane information to a Nox controller which is inside Domain 0. Then, the controller translates this data into flows using the Rule Table module, which was developed as a Nox application, and configure the OpenFlow switch on demand.

XenFlow forwards data packets as follows. A packet that reaches Domain 0 is directly forwarded, if it matches any flow in the Flow Table; otherwise, the packet is forwarded to the controller, in order to have its path defined by the controller. In this case, Nox controller extracts the twelve OpenFlow fields from the packet, queries a Rule Table to define to which network that packet belong and which is the corresponding forwarding rule of that network. After that, the controller inserts a new flow in the Flow Table of the OpenFlow switch. It is important to notice that the packet arrives in Domain 0 with the destination MAC address of the virtual router and this address has to be modified to the next hop MAC address before being forwarded by the OpenFlow switch. The next hop MAC address is obtained in the Rule Table. In case that node is only a virtual switch, this operation is not performed. Hence, this module guarantees the correctness

when mapping a virtual link into one or more physical links.

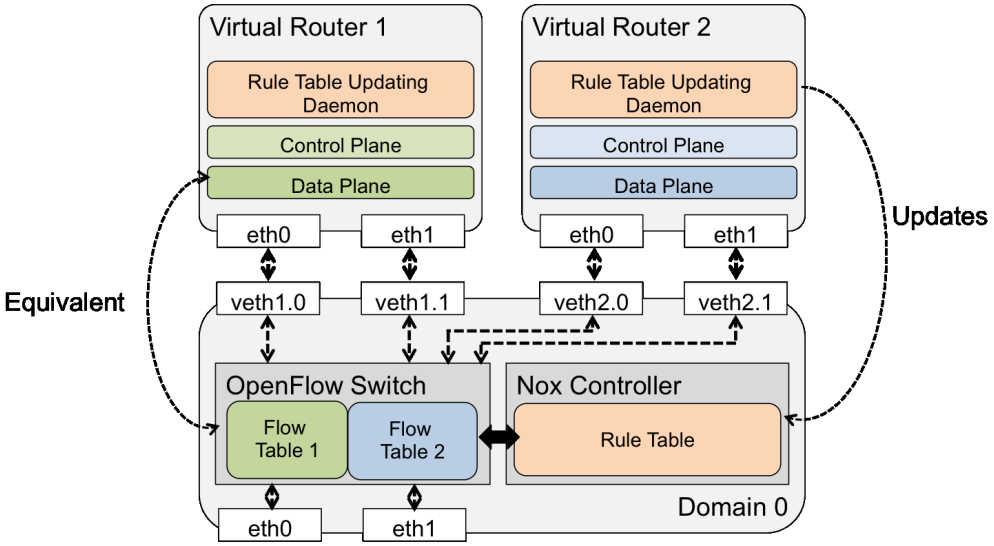


Figure 3.5: XenFlow routing, in which packets are directly forwarded by Domain 0.

3.2.2 XenFlow virtual topology migration

In a XenFlow network, a virtual link can be mapped into one or more physical links. The packet forwarding is accomplished by a flow table dynamically programmed by the Nox controller. Thus, the logical and the physical topology are detached. Therefore, virtual-node migration in a XenFlow network, shown in Figure 3.6, is composed of three steps: control plane migration, data plane migration, and link migration. The control plane migration occurs from the origin physical node to the destination physical node in a similar way as for the Xen conventional live migration mechanism [26]. After the control plane migration, the data plane migration is accomplished as follows: the flows related to the migrated virtual router are selected and they are sent to the destination physical router; in the destination, the flow definitions are mapped to the currently setting up of the physical and the virtual router. Thus, it is kept the correspondence of flow input and output ports, taking into consideration the Domain 0 virtual switches in the source and in the destination of the migration. Then, translated flows are added to the Flow Table of destination Domain 0 OpenFlow switch. After data and control planes migration, the link migration occurs in Domain 0 OpenFlow switches, and in the others network switches. The link migration

creates a switched path between the one logical hop virtual router neighbors to the destination physical router. In order to do so, flows are defined in the physical routers in the path between the destination physical router and the physical routers which host the virtual routers that are one hop distant of the migrated virtual router. Adding the existing flows to the flow table, however, is not enough. An automatic mechanism is needed to create, on demand, new flows on physical routers in the path. This mechanism is deployed through the introduction of new rules on controller Rule Tables of the nodes in the path.

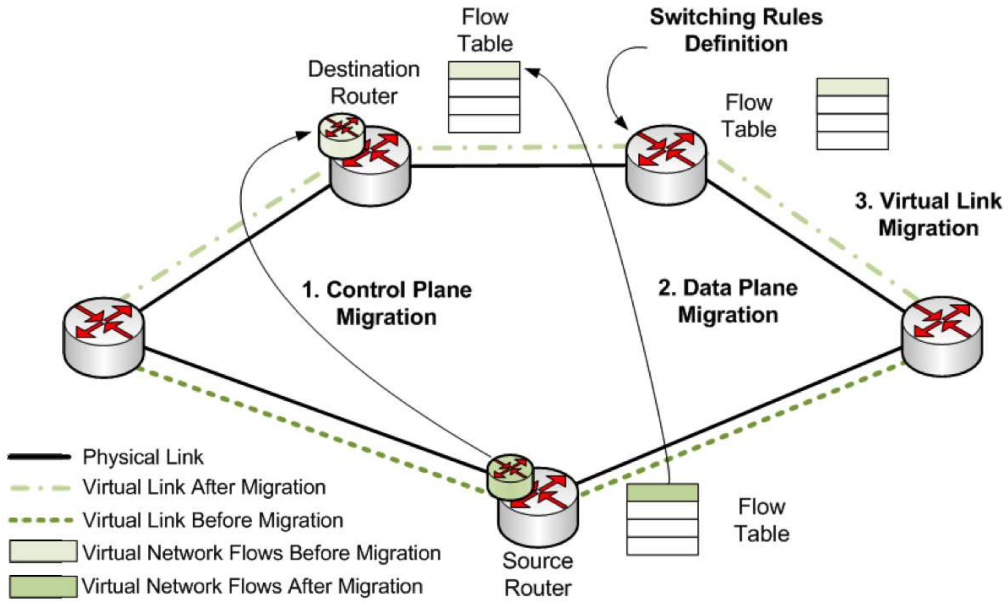


Figure 3.6: Three steps of the XenFlow virtual topology migration.

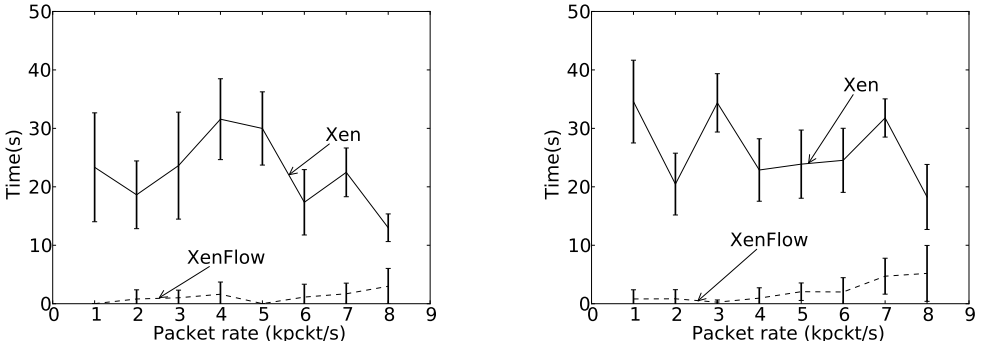
3.3 Experimental results

A prototype of the proposed architecture was developed as a proof of concept of virtual routers migration without packet loss. To evaluate the performance, we adopted the tool *Iperf* [27], as a packet generator tool, and *Tcpdump* [28] to measure the amount of packets generated, received, and lost. The packet loss was measured from the comparison of information collected using *Tcpdump* on network interfaces responsible for generation and for reception of packets. To assess XenFlow performance during migrations, it is compared to the native migration of the Xen virtualization platform.

The experimental scenario is composed of four machines: two for packet forwarding and two for generate/receive packets. Two machines perform the function of forwarding packets, and the prototype was installed in them. Each machine is equipped with an Intel Core 2 Quad processor and three Ethernet network interfaces of 1 Gb/s. The machines run the Xen hypervisor 4.0-amd64. In one of these physical machines, a virtual machine is instantiated with one virtual CPU, 128 MB of memory, two network interfaces, and Debian 2.6-32-5 operating system, to work as a virtual router. The experiments use two additional machines, equipped with Intel Core 2 Duo processors, that generate or receive packets, each one equipped with an Ethernet interface 1 Gb/s, connected to a control network, and two Ethernet network interfaces 100 Mb/s, to communicate with both physical routers. The experiments were performed with the virtual router forwarding UDP packets of 64 and 1500 bytes, which are, respectively, the minimum content of an Ethernet frame and the most common size of MTU (Maximum Transmission Unit).

The first experiment measures the control plane downtime during migration. We send control packets that are forwarded by the virtual machine to determine the average downtime according to the number of lost control packets. Hence, the control plane downtime is given by the difference between the timestamp of the last packet received immediately before the migration, and the timestamp of the control packet received immediately after migration. Figure 3.7 shows the control plane downtime for the XenFlow system and for the native Xen migration, as a function of the transmitted packet rate. The results show that the average downtime of the virtual router is always lower than 5 seconds in XenFlow, no matter the packet size. On the other hand, in native Xen migration, the average downtime of virtual router ranged from 12 to 35 seconds. This difference mainly occurs for two reasons. First, there is no writings on virtual machine memory during the migration using XenFlow, because the packets are sent directly by Domain 0, while in Xen migration, all packets are forwarded by the virtual machine. Hence, packet forwarding in Xen generates memory writings and readings while the virtual machine is migrated. This larger usage of memory leads to a greater number of dirty pages, which implies in a larger downtime when copying the last memory pages of the virtual machine. Secondly, XenFlow migration is performed in two steps. First, the virtual machine is migrated and then the data plane is migrated using OpenFlow migration, which avoids packet losses. On native Xen migration, the link migration is accomplished by sending *ARP Reply* packets, in order to indicate the interfaces in which the migrated virtual machine is now available. However, the proper working of *ARP Reply* mechanism is conditioned on the expiration of the ARP tables

entry. This can add a delay in updating the interfaces used to communicate with the migrated virtual machine.



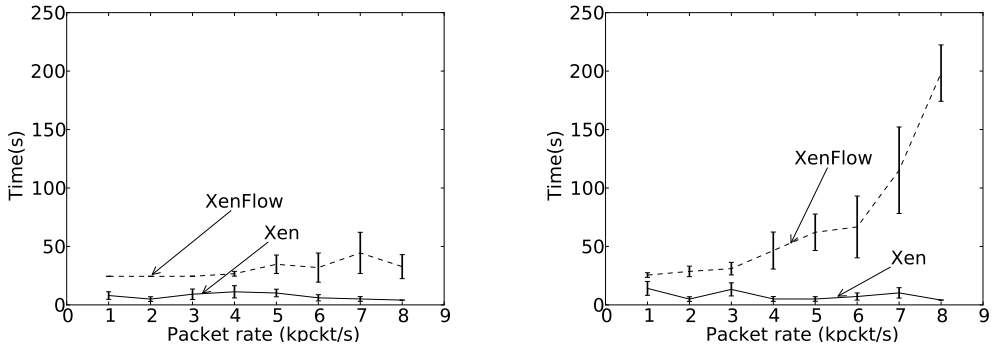
(a) 64 bytes packets.

(b) 1500 bytes packets.

Figure 3.7: Control plane downtime.

The second experiment evaluates the total migration time, which establishes the minimum time between two consecutive virtual network element migrations. The total migration time considers the execution time of all operations related with the migration process. Figure 3.8 presents the total migration time as a function of the transmitted packet rate. The results show that XenFlow migration presents a greater total migration time than Xen. It happens because XenFlow migration involves more steps, and one of them is the native Xen migration itself. Part of the additional time occurs because of the flow migration, in order to rebuild the data plane in the destination physical router, and also because of the link migration, which is responsible for setting up the virtual network new topology over the physical network. Figure 3.8(b) shows that for 1500 bytes packets, there is an increase in XenFlow total migration time, as the transmitted packet rate increases. This is caused due to the fact that, when using packets of 1500 bytes, the link of 100 Mb/s is saturated at the rate of approximately 8,000 packets per second.

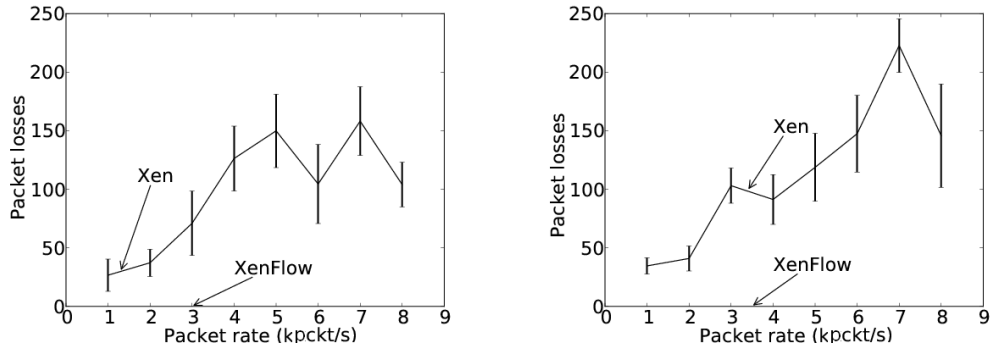
A primitive for router migration is to have no packet loss. Thus, the third experiment shows the number of packet losses during the migration in both systems. Figure 3.9 reveals that, while migrating a router using XenFlow, there is no packet loss. Besides, Figure 3.9 shows that XenFlow zero packet loss is independent of the forwarded packet rate. On the other hand, Xen native migration presents greater packet losses for greater transmitted



(a) 64 bytes packets. (b) 1500 bytes packets.

Figure 3.8: Total Migration Time.

packet rates. This reflects the forwarding service downtime, as we can see in Figure 3.7. As the native Xen migration suspension downtime is almost constant, the amount of lost packets in this interval increases with the sending rate.



(a) 64 bytes packets. (b) 1500 bytes packets.

Figure 3.9: Amount of packet losses as a function of the transmitted packet rate.

Chapter 4

Multi-Agent System for Self-Management of Virtual Networks

In this chapter we applied autonomic computing techniques for the management of virtual networks. We present a distributed architecture to enable a self-management of virtual networks over a substrate network. It makes part of the piloting plane described in Work Package 3. The autonomic managers of the network elements have a closed control loop of monitoring, analysis, planning, and execution which feeds a knowledge base for next iterations. We focus on the self-healing of virtual networks, but the distributed architecture for self-management of virtual networks is generic enough to be used to other functional areas in autonomic computing: self-configuration, self-optimization, and self-protection. In sequence we present some related work.

A multi-agent system to maintain SLAs (service level agreements) in events of resource failure and severe performance degradation is presented in [29]. The agents form groups based on similarity of physical nodes that are managed by them. The dissimilarity function is also used to choose where the virtual nodes will be recovered in case of failure. Our architecture is also based on multi-agent systems and we implement a fast recovery of virtual networks by restoring the virtual routers from backup memory to reduce the convergence time of the routing protocol.

In [30], a distributed management architecture is presented with the goal of self-organizing virtual networks to maintain a good use of physical resources. The algorithm used for self-organization is based on the autonomic control loop. It monitors the link and tries to minimize the traffic load on the network through the migration of virtual nodes. We present similar ar-

chitecture and we implemented a prototype with focus on the self-healing of virtual networks to evaluate our proposal.

4.1 Autonomic networks

The autonomic computing manifesto [31] emphasizes that the complexity has been a major obstacle to the development of IT, because it is growing beyond human ability to manage it. The autonomic computing is bio-inspired on autonomic nervous system that is responsible for regulating the body according to environmental changes without the need for conscious control. Its goal is to reduce or eliminate human intervention in the management through properties of self-configuration, self-optimization, self-healing, and self-protection.

The architecture of [32] is based on distributed autonomic managers. The autonomic manager performs a control loop over the managed element and uses a knowledge base to store the collected information. It performs the activities of monitoring, analysis, planning and execution at each loop, which feeds the next cycle through the knowledge base.

The scenario of increasing complexity, heterogeneity, ubiquity, connectivity, and integration is the reason that leads to the need for development of autonomic networks. The autonomic networks are based on the principles of autonomic computing. They should be able to perform self-management from high-level policy sets by administrators or inferred through knowledge of the application.

The autonomic network should self-configure, self-optimize, self-heal and self-protect itself to be self-managed. These properties can be obtained with the inclusion of autonomic managers in network elements to perform these tasks. The self-management system should be distributed with each autonomic manager responsible for a network resource that is its managed element. This prevents the creation of a single point of failure and enables higher scalability.

Managers must act independently, but share common goals. The operation of individual managers on their autonomic managed elements should provide a greater autonomic control loop of self-management network according to its policies and objectives.

The FOCALE [33] is an autonomic architecture for network management. It is also build on autonomic managers over managed resources, and the control cycle is defined by the managers at runtime, by context and high-level policies. The architecture also defines the use of an MBTL layer (model-based translation layer) to enable the use of legacy equipment management system.

To perform the adaptation, the architecture uses techniques of learning and cognition to compare, based on models and ontologies, if the current behavior is right or whether it should be replaced. The FOCALE architecture was applied as a case study in projects like Beyond 3G Networks and Motorola's Seamless Mobility.

4.2 Proposal of a multi-agent architecture

The agent-oriented modeling is an interesting paradigm for the development of distributed self-managing systems. Agents are autonomous entities that observe the environment and act on it, and may have some level of cognition, as well as communicate with other agents. They can play the role of the manager of an autonomic network resource, its managed element. Therefore, we propose a distributed system to self-manage virtual networks based on a multi-agent architecture.

In our multi-agent system for self-management of virtual networks the agents acts on the physical network nodes. Agents are responsible for monitoring the resources of the physical and virtual nodes: CPU, memory, storage, network interfaces etc. They must also control the virtual routers, i.e., create, destroy and migrate them.

The architecture of agents is based on behaviors, knowledge base, policies and the dynamic planner, as shown in Figure 4.1. The sensing, cognition and activities of agents are performed by behaviors.

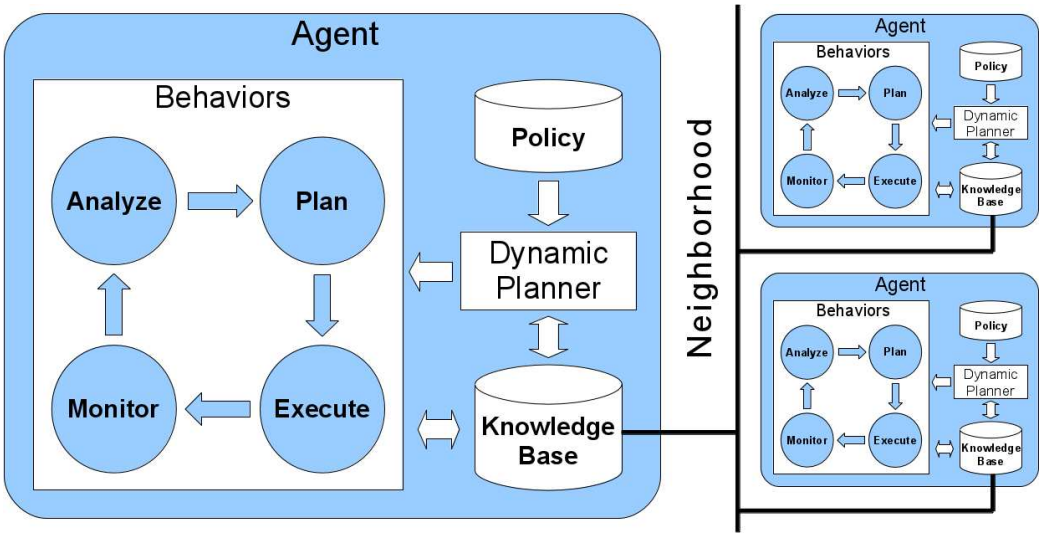


Figure 4.1: Autonomic manager architecture.

The knowledge base serves as a repository for all information of the agents, which can be collected locally by monitoring, or remotely through communication with other agents. The knowledge base has a common information model to enable the agents communication and data interpretation. The information model, in Figure 4.2, represents the virtual network topologies and their mappings in the physical network and it is based on the work [34].

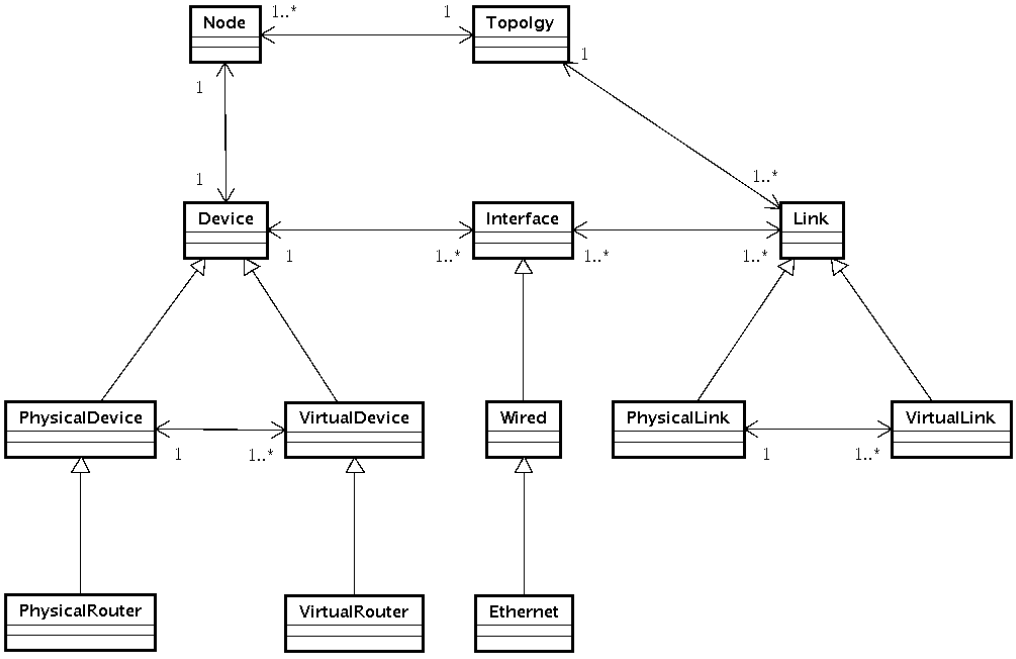


Figure 4.2: Information model of the knowledge base.

We define in the policy file the execution order and parameters of behavior, such as the rate of the loop and the agent downtime to be inferred a physical node failure. The dynamic planner is responsible for interpreting the policy file, changing parameters of behavior, and controlling the life cycle of the agent. It also has access to the knowledge base and can act upon information contained on it.

For the multi-agent system works properly it is necessary that the agents are synchronized to perform some of their actions, e.g., the Execute behavior should ensure that all agents have sent their physical node costs to be sure that only one agent will perform the recovery of the virtual router. We enabled this synchronization through pieces of information in the knowledge base that controls the behaviors.

Agents can be organized into neighborhoods, which limits the scope of dissemination of the knowledge base. Through this selective communication, agents form situated views on the network. An agent may belong to several different neighborhoods and propagate information for each one. In the multi-agent system for self-management of virtual networks there is only one neighborhood with all members of the network, and all the information generated by the agent is diffused to the neighbors.

4.3 Implementation

The Figure 4.3 illustrates the testbed built to test the system in a controlled environment. The network core consists of four machines: *zeus*, *atlas*, *dionisio*, and *cronos*. The machines *zeus* and *dionisio* are also connected to the hosts *apolo*, *hermes*, *nix*, and *artemis* by Giga switches. Over the substrate network two virtual networks with three virtual routers each were created. The images of virtual routers are in a repository accessible by all machines of the substrate network via NFS. Both the physical and the virtual machines have the operating system Debian GNU/Linux with kernel version 2.6.32.

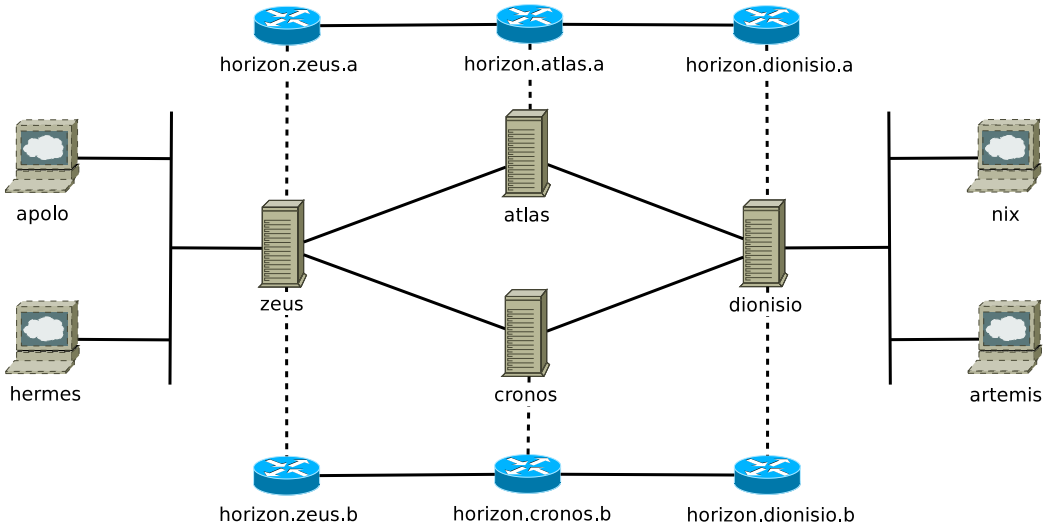


Figure 4.3: Testbed built to validate the multi-agent system for self-management of virtual networks.

The management of virtual machines for both the creation of the testbed and the agents operation uses the Libvirt library [35]. It provides an API for monitoring and control of various virtualization platforms, including Xen.

The use of Libvirt in the system for self-management of virtual networks is important for making it independent of virtualization technology.

The agents run on physical machines at the network core: *zeus*, *atlas*, *dionisio*, and *cronos*. We use the Ginkgo platform [36] for the implementation of the agents. It allows the creation of lightweight and portable agents, which facilitates its deployment in heterogeneous environments. Ginkgo is a framework that has the basic building blocks for our architecture.

The multi-agent system must perform disaster recovery of virtual networks. Failures can occur by problems in virtual routers, in the physical nodes or even in the physical links. In the first case, the agent responsible for the virtual router must diagnose and warn others about the failure. In other cases, the agent may also stop communicating, and therefore the neighbors must diagnose the failure.

We implement an autonomic control loop for self-healing of virtual networks. The loop is controlled by the dynamic planner and is formed by four behaviors: Monitor, Analyze, Plan and Execute. Our agents regularly perform these behaviors in sequence. They are described below:

Monitor: Collects data from virtual and physical nodes and feeds the knowledge base.

Analyze: Performs fault diagnosis in virtual routers or physical nodes in the neighborhood.

Plan: Calculates the cost of the substrate node from the use of its resources. The physical nodes with more virtual routers running have the greater costs. It also disseminates this information to other agents.

Execute: Verifies whether all agents in the neighborhood have already sent their information. If so, the agent in the physical node with lowest cost recovers the failed virtual routers.

We create a special agent running on a host machine. This agent is part of the neighborhood and receives information of the other agents. It executes a behavior to create and update a graphical interface. This agent contributes to prototype an architecture with hybrid agents with different capabilities and levels of cognition. In this case it performs a visualization of the topology and the mappings of virtual networks over the substrate, and charts with the information collected by the Monitor behavior of the other agents. The graphical interface is shown on Figure 4.4.

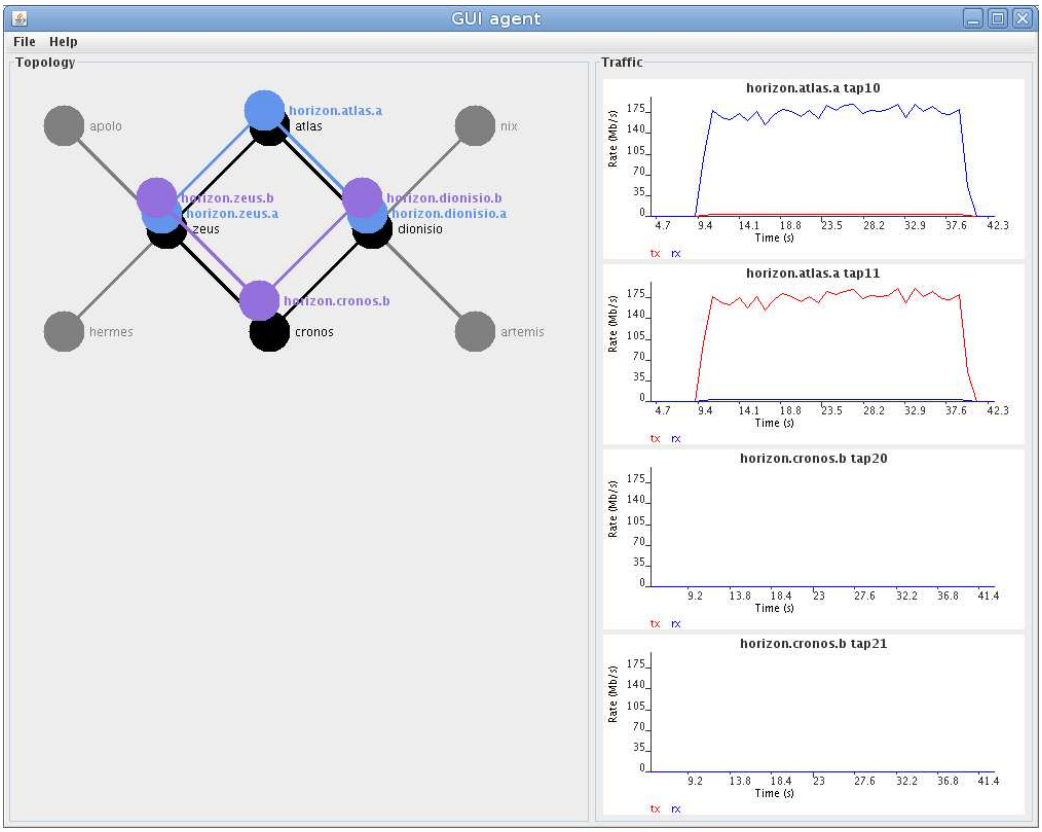


Figure 4.4: Graphical interface of the multi-agent system.

4.4 Experimental results

The experiments aim to validate the architecture for self-management of virtual networks and test different approaches to the recover virtual routers in cases when there are failures on the physical elements. The recovery time T_r from a virtual network can be set according to:

$T_r = T_d + T_p + T_i + T_c$

where T_d is the time of failure diagnose; T_p is the time spent on the planning action, which involves exchange of information between agents; T_i is the time of instantiation of virtual machine and T_c is the convergence time of routing protocol.

We studied two ways to recover the virtual router. One creates a virtual machine from the image file in the repository, booting the guest operating system. The other resumes the machine from backup memory file, also in the repository, generated when the virtual router was operational. In this case, there is no time involved with booting the guest operating system and

the convergence time of routing protocol is smaller.

The executions were performed with static and dynamic routing in the virtual network. Static routing is manually configured in the virtual routers. For the dynamic routing, the Quagga routing suite [37], running OSPF algorithm, was used.

The first experiment was performed in the testbed without the multi-agent system. In this experiment migrations of a virtual router with static routing were performed, while a UDP flow at constant rate of 500Kbps from *apolo* to *nix* passed through the virtual network. The traffic was generated with Iperf tool. The curves of Figure 4.5 show the flow rate and when it is at zero indicates losses in the network. All migrations change the mapping of the virtual router *horizon.cronos.b* from *cronos* to *atlas* and began near 8s.

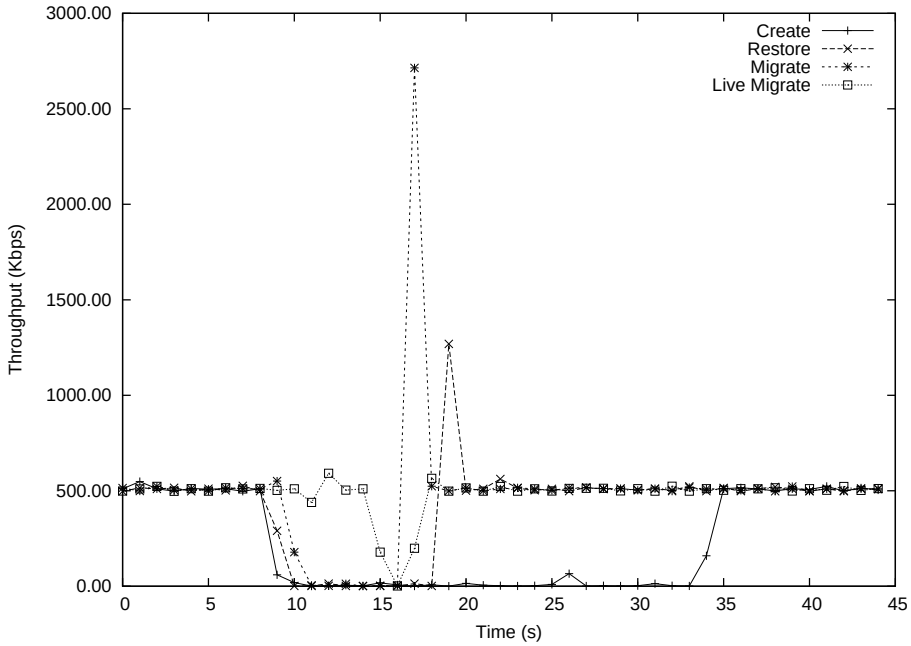


Figure 4.5: Experiments without the multi-agent system.

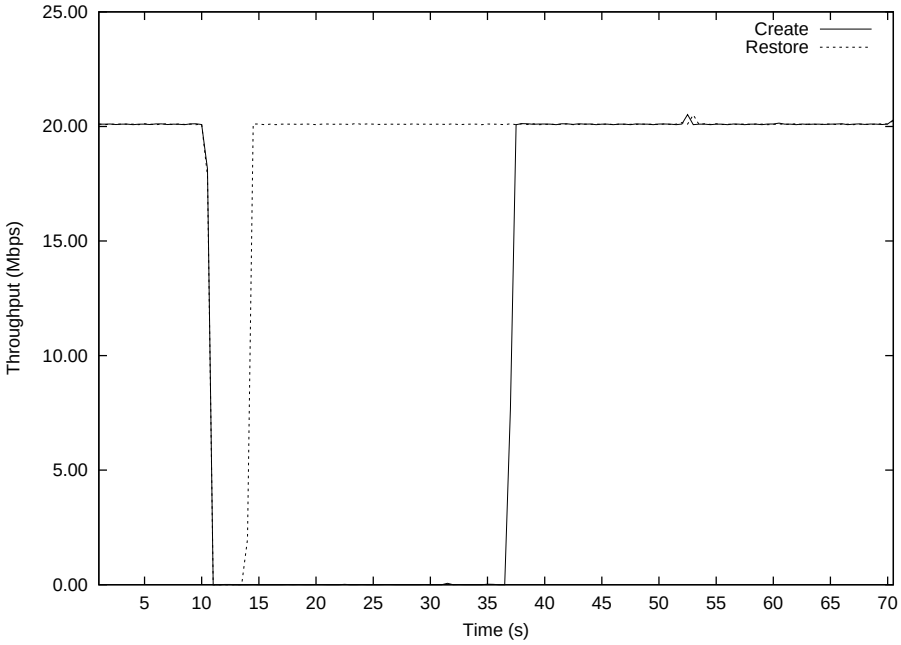
In the first run (curve "Create" in Figure 4.5), the virtual router was destroyed at *cronos* and recreated at *atlas*. In this case, the recovery took 27s mainly due to the time of booting the operating system. In the second run (curve "Restore" in Figure 4.5) the state of the virtual router was saved in the network file repository, the virtual router was stopped at *cronos*, saved at image repository, and restored at *atlas*. Even spending more time with the memory saving, the process took only 45% of the time of the former. The

next runs use migration services offered by KVM. In the basic migration, executed in the third run (curve "Migrate" in Figure 4.5), the virtual machine is stopped, the memory is copied over the network from origin to destination, and then the machine is restored on destination. This situation is similar to the second run, except that the copy is made directly from *cronos* to *atlas* and not through an intermediate machine and therefore it is a little faster, 37% of the time of the first run. In these two situations, when the machines were restored there was a peak in the network. This was because in the instant that the virtual machine memory was saved, there were packets in the buffer that have been added to those that were arriving at the instant that it was reactivated. The forth run (curve "Live Migrate" in Figure 4.5) was carried out with live migration, where the memory is copied from source to destination while the virtual machine is running and only when there are no more modified page, the control is transferred. This is the best way to migrate a virtual router, because the transmission was interrupted for fewer time, less than 15% compared to the first run.

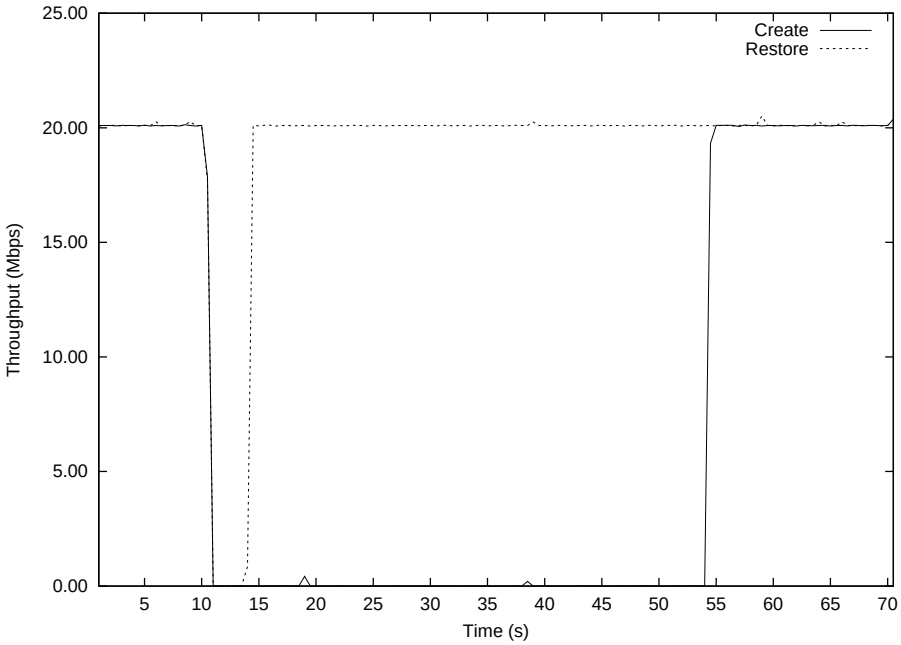
The graphs in Figure 4.6 show the results of the experiment with the distributed management system and the UDP traffic generated with Iperf tool. The graph 4.6 (a) shows the executions with static routing, and graph 4.6 (b), with dynamic routing. In each graph the approaches to create the virtual router from image file, and to restore the virtual machine from the backup memory file, are being compared. In all runs, a UDP flow at constant rate of 20Mbps from *apolo* to *nix* is going through the virtual network B and the curves represent the flow rates over time.

After 10s the machine *cronos* is disconnected from the network. Agents in *zeus*, *atlas* and *dionisio* diagnose the failure of *cronos* because they stop receiving the knowledge base propagation of its agent. In the experiment, the agents perform the Monitor behavior and the knowledge base dissemination every 0.5s and when the information about one physical node is outdated more than 1s, the Analyze behavior reports the problem. So, the time of diagnosis of failure varies between 1 and 1.5s. Thereafter the agent enters in the Plan behavior, where it calculates the physical node cost that will be sent to other agents in the next propagation. The Execution behavior waits all agents send their physical nodes costs. The cost of *zeus* and *dionisio* are higher because they are running two virtual routers. So who will recover *horizon.cronos.b* is the agent in *atlas*, which creates the virtual machine from image or restores from the backup memory.

Another experiment with SCP application over virtual network was performed and the results are presented in Figure 4.7. The SCP generates a TCP traffic. A 1GB file was transferred from *hermes* to *artemis* through virtual network B. For each scenario we performed 10 runs and calculate a



(a) Static routing



(b) Dynamic routing

Figure 4.6: Experiments with the multi-agent system and Iperf.

95% confidence interval. All executions were carried out in a close period, when the network conditions were similar.

In this experiment, we performed some executions without the agents to serve as a basis for comparison. First, we measure the time to send the file by Path A (*zeus* → *cronos* → *dionisio*) and by Path B (*zeus* → *atlas* → *dionisio*). As shown in Figure 4.7, the flow rates of Path A and Path B are different. So, we perform executions with live migration of the *horizon.cronos.b* from *cronos* to *atlas*, 10s after starting the SCP transmission, to serve as a floor result, because the live migration has the lowest downtime. Next, we execute the runs with the agent system to perform the recovery of the virtual network. In these scenarios a failure occurs in physical node *cronos*, also after 10s. Again, the multi-agent system chooses *atlas* to recover *horizon.cronos.b*. We do 4 scenarios with failure, combining the routing scheme: static and dynamic, and the virtual router recovery approach: create the virtual machine from image and restore the virtual machine from backup memory.

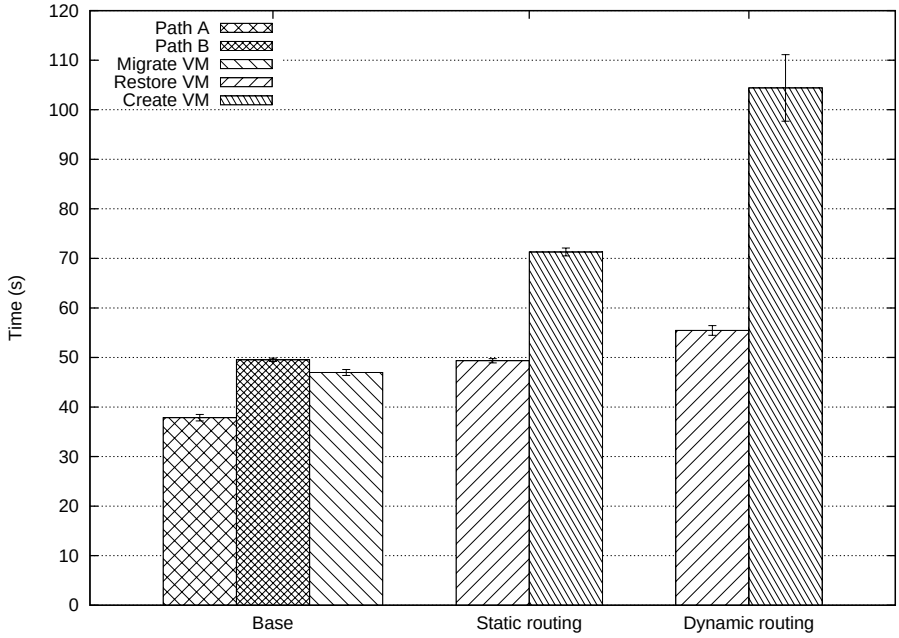


Figure 4.7: Experiments with multi-agent system and SCP.

There is a considerable difference in recovery time of the virtual network according to the way the virtual router is recovered. The difference is greater in dynamic routing, because the convergence time of the routing algorithm. The recovery by creating the virtual machine took 43.6s in this case, and 26.4s with static routing in the Iperf experiments with multi-agent system. The

time to complete the SCP transmission was 52% and 122% higher in average from the floor in static and dynamic scenario respectively. The creation of virtual machine in dynamic routing scenario had a greater variance because the routing protocol convergence time.

In the approach which the virtual machine state is restored from backup memory, the type of routing interfered less. In both the static and the dynamic scenario, the recovery time was about 3.5s in Iperf experiments with multi-agent system. The impact on the SCP transmission time was about 5% and 18% from floor in static and dynamic scenario respectively.

The results were favorable to the approach of restoring the virtual machine from memory backup, especially when dealing with virtual routers to save the convergence time of the routing protocol. Although this approach is faster, saving, transmitting and storing the backup memory may cause significant impacts on network. This backup could occur in moments of idleness of the network to cause less impact on its operation. Distributed storage and memory ballooning can also be used to reduce this overhead.

Chapter 5

Conclusions and Ongoing Work

In this document, we present the overall view of the architectures developed in this project. The first architecture is based on the Xen virtualization platform. We developed a model that integrates the features developed during the project, in addition to new control and management modules. This model uses as basis the virtualization tool developed in work package 2. The main modules in the Xen architecture are the resource managers, which deals with the allocation of physical resources to virtual networks according to the policies previously defined. Hence, we present a controller for observing the use of shared bandwidth, CPU, and memory in Domain 0, a fuzzy logic controller to handle the resource consumption inside virtual machines, and a controller that interacts with Xen scheduler to ensure the bandwidth provision for virtual machines. To integrate these modules, we modified the monitoring functions, performed by the measure gather module and ADAGA. Other modules were also added, such as the admission controller, which evaluates if new virtual networks can be hosted in a specific physical node. The admission controller is the basis for applying global functions that remap virtual networks according to network load or other premises. We also added QoS modules for our architecture, to accomplish the requirements of a new architecture for the Internet, as described in Report 4.1.

We performed tests to evaluate the performance of the new modules added to Xen architecture. These tests show that the developed mechanisms perform better than other mechanisms of the literature. In addition, our analysis shows that the proposed admission control dynamically adapts to different traffic demands, guaranteeing a precise control of the number of virtual network hosted in the physical device. Hence, we avoid both the overcharge and the undercharge of the physical node, differently from other proposals that were analyzed. We also present tests performed with the developed QoS module. The results show the advantages of providing QoS among vir-

tual networks, instead of guarantying QoS primitives only inside the virtual network.

The second developed architecture applies to OpenFlow networks. We used as basis for our architecture the virtualization tool developed in work package 2, called OpenFlow MaNagement Infrastructure (OMNI). Since OpenFlow network control is centralized, we placed the infrastructure control modules inside a special controller that interacts with FlowVisor for controlling physical resources provided to each virtual network. We added control modules initially developed for the Xen platform to control the resources in OpenFlow networks, since the proposed algorithms are platform independent. Another functionality added to OMNI is the agent control, which emulates a distributed control in the network. The performed analysis shows the advantages of using agents, as well as the impact of our management system over the virtualized network.

We also design a third approach that joins the advantages of Xen and OpenFlow in a different virtualization platform. This approach is a hybrid network architecture design, which combines the flexibility of the OpenFlow switching matrix with the distributed control of a network virtualization based on Xen platform. This approach is called XenFlow and implements the concept of flow processing, in which a general purpose network node is able to process any kind of flow. XenFlow provides a robust and efficient way for migrating virtual topologies. The XenFlow main goal is to achieve the zero packet loss virtual router migration and to eliminate the need of tunnels or external mechanisms for migration of links. The presented architecture design uses the plane separation technique deployed as an application over the Nox controller, which controls the data plane based on forwarding rules. These rules are updated by a daemon that runs in each virtual router. The results show that the XenFlow control plane downtime is up to 30 times lower than the native Xen migration downtime. The results also show that the total migration time of XenFlow is greater than the native Xen migration. This result occurs due to new steps introduced by XenFlow into virtual topology migration process, when compared with Xen. However, increasing the total migration time is not a significant factor for the virtual router migration. It only sets the minimum time between two consecutive migrations. The results show that a XenFlow virtual router migration occurs without packet loss, which makes this architecture design appropriate to the scenario of virtual networks, as opposed to the native migration of the Xen virtualization platform.

Besides the overall view of the architectures developed in this project previously mentioned, we presented our self-management system prototype, described in the report related to the task of workpackage 3.2, in which the

concepts of autonomic networks were applied in a virtualized environment through a multi-agent system. We evaluate this autonomic self-management environment with a focus on virtual network self-managing failures, or self-healing. We show how this infrastructure was used to build our prototype.

As one of our ongoing works, we intend to develop new applications in addition to the routing application in XenFlow. In this new context, the virtual machines are intended to provide services of middle boxes, such as, load balancer and firewall. Thus, the XenFlow architecture becomes a platform for the implementation of routers and specialized nodes, which can be migrated to different locations on the network.

Bibliography

[1] N. Fernandes, M. Moreira, I. Moraes, L. Ferraz, R. Couto, H. Carvalho, M. Campista, L. Costa, and O. Duarte, "Virtual networks: Isolation, performance, and trends," *Annals of Telecommunications*, pp. 1–17, 2010.

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

[3] I. M. Moraes, P. S. Pisa, H. E. T. Carvalho, R. S. Alves, L. H. G. Ferraz, R. S. Couto, D. J. S. Neto, V. P. Costa, R. A. Lage, N. C. Fernandes, M. E. M. Campista, L. H. M. K. Costa, and O. C. M. B. Duarte, "VNEXT: Uma ferramenta de controle e gerenciamento para redes virtuais baseadas em Xen," in *Salão de Ferramentas do XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC'2011*, pp. 1–8, may 2011.

[4] P. S. Pisa, R. S. Couto, H. E. T. Carvalho, D. J. S. Neto, N. C. Fernandes, M. E. M. Campista, L. H. M. K. Costa, O. C. M. B. Duarte, and G. Pujolle, "VNEXT: Virtual network management for Xen-based testbeds," in *A ser publicado no 2nd IFIP International Conference Network of the Future - NoF'2011*, pp. 1–5, nov 2011.

[5] 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.

[6] P. Pisa, N. Fernandes, H. Carvalho, M. Moreira, M. Campista, L. Costa, and O. Duarte, "Openflow and xen-based virtual network migration," in *Communications: Wireless in Developing Countries and Networks of the Future* (A. Pont, G. Pujolle, and S. Raghavan, eds.), vol. 327 of *IFIP*

Advances in Information and Communication Technology, pp. 170–181, Springer Boston, 2010.

[7] N. C. Fernandes and O. C. M. B. Duarte, "Xnetmon: Uma arquitetura com segurança para redes virtuais," in *Anais do X Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, SBSEG '10, (Fortaleza, CE, Brazil), pp. 339–352, Oct. 2010.

[8] N. C. Fernandes, M. D. D. Moreira, and O. C. M. B. Duarte, "XNetMon: A network monitor for securing virtual networks," in *IEEE International Conference on Communications (ICC 2011 - Next Generation Networking and Internet Symposium)*, ICC'11 NGNI, 2011.

[9] N. C. Fernandes and O. C. M. B. Duarte, "Provendo isolamento e qualidade de serviço em redes virtuais," in *XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC'2011*, pp. 1–14, may 2011.

[10] R. S. Couto, M. E. M. Campista, and L. H. M. K. Costa, "XTC: a throughput control mechanism for Xen-based virtualized software routers," in *IEEE Global Communications Conference (GLOBECOM'2011) - accepted for publication*, (Houston, Texas, EUA), Dec. 2011.

[11] R. S. Couto, M. E. M. Campista, and L. H. M. K. Costa, "XTC: Um controlador de vazão para roteadores virtuais baseados em Xen," in *XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC'2011*, pp. 1–14, may 2011.

[12] H. E. T. Carvalho, N. C. Fernandes, and O. C. M. B. Duarte, "Um controlador robusto de acordos de nível de serviço para redes virtuais baseado em lógica nebulosa," in *XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC'2011*, pp. 1–14, may 2011.

[13] H. E. T. Carvalho, N. C. Fernandes, and O. C. M. B. Duarte, "SLAPv: A service level agreement enforcer for virtual networks," in *International Conference on Computing, Networking and Communications, Internet Services and Applications Symposium - International Conference on Computing, Networking and Communications, Internet Services and Applications Symposium*, pp. 1–5, jan 2012.

[14] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Sandpiper: Black-box and gray-box resource management for virtual machines," *Computer Networks*, vol. 53, no. 17, no. 17, pp. 2923–2938, 2009.

[15] I. Fajjari, N. Aitsaadi, G. Pujolle, and H. Zimmermann, "VNE-AC: Virtual network embedding algorithm based on ant colony metaheuristic," in *ICC 2011 Next Generation Networking and Internet Symposium (ICC'11 NGNI)*, pp. 1–6, IEEE, June 2011.

[16] D. M. F. Mattos, N. C. Fernandes, L. P. Cardoso, V. T. da Costa, L. H. Mauricio, F. P. B. M. Barreto, A. Y. Portela, I. M. Moraes, M. E. M. Campista, L. H. M. K. Costa, and O. C. M. B. Duarte, "OMNI: Uma ferramenta para gerenciamento autônomo de redes OpenFlow," in *Salão de Ferramentas do XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC'2011*, pp. 1–8, may 2011.

[17] D. M. F. Mattos, N. C. Fernandes, V. T. da Costa, L. P. Cardoso, M. E. M. Campista, L. H. M. K. Costa, and O. C. M. B. Duarte, "OMNI: OpenFlow management infrastructure," in *A ser publicado no 2nd IFIP International Conference Network of the Future - NoF'2011*, pp. 1–5, nov 2011.

[18] R. Sherwood, G. Gibb, K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," tech. rep., Tech. Rep. OPENFLOW-TR-2009-01, OpenFlow Consortium, 2009.

[19] D. M. F. Mattos, N. C. Fernandes, and O. C. M. B. Duarte, "XenFlow: Um sistema de processamento de fluxos robusto e eficiente para migração em redes virtuais," in *XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC'2011*, pp. 1–14, may 2011.

[20] 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, no. 1, pp. 61–64, 2007.

[21] S. Ratnasamy, S. Shenker, and S. McCanne, "Towards an evolvable Internet architecture," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4, no. 4, pp. 313–324, 2005.

[22] R. Bolla, R. Bruschi, F. Davoli, and A. Ranieri, "Energy-aware performance optimization for next-generation green network equipment," in *Proceedings of the 2nd ACM SIGCOMM workshop on Programmable routers for extensible services of tomorrow*, pp. 49–54, ACM, 2009.

[23] Y. Wang, J. van der Merwe, and J. Rexford, "VROOM: Virtual routers on the move," in *Proc. ACM SIGCOMM Workshop on Hot Topics in Networking*, Citeseer, 2007.

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

[25] 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, no. 3, pp. 105–110, 2008.

[26] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 273–286, USENIX Association, 2005.

[27] *Iperf.* http://iperf.sourceforge.net/, Accessed in October, 2011.

[28] *Tcpdump & libpcap.* http://www.tcpdump.org/, Accessed in October, 2011.

[29] I. Houidi, W. Louati, D. Zeghlache, P. Papadimitriou, and L. Mathy, "Adaptive virtual network provisioning," in *Proceedings of the second ACM SIGCOMM workshop on Virtualized infrastructure systems and architectures*, VISA '10, (New York, NY, USA), pp. 41–48, ACM, 2010.

[30] C. C. Marquezan, L. Z. Granville, G. Nunzi, and M. Brunner, "Distributed autonomic resource management for network virtualization," in *IEEE/IFIP Network Operations and Management Symposium (NOMS 2010), Osaka, Japan*, pp. 463 –470, april 2010.

[31] IBM, "Autonomic Computing: IBM's Perspective on the State of Information Technology," 2001. `http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf`, Accessed in October, 2011.

[32] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, pp. 41–50, January 2003.

[33] J. Strassner, N. Agoulmine, and E. Lehtihet, "FOCALE – A Novel Autonomic Networking Architecture," in *Latin American Autonomic Computing Symposium (LAACS 2006), Campo Grande, Brazil*, 2006.

[34] I. Fajjari, M. Ayari, and G. Pujolle, "VN-SLA: A Virtual Network Specification Schema for Virtual Network Provisioning," in *International Conference on Networking (ICN 2010), Menuires, France*, pp. 337 –342, April 2010.

[35] D. Coulson, D. Berrange, D. Veillard, C. Lalancette, L. Stump, and D. Jorm, "Libvirt 0.7.5: Application Development Guide," 2010. `http://libvirt.org/guide/pdf/Application_Development_Guide.pdf`, Accessed in October, 2011.

[36] Ginkgo Networks, "Ginkgo Distributed Network Piloting System: White Paper," 2008. `http://www.ginkgo-networks.com/IMG/pdf/WP_Ginkgo_DNPS_v1_1.pdf`, Accessed in October, 2011.

[37] K. Ishiguro, "Quagga: A routing software package for TCP/IP networks," 2006. `http://www.quagga.net/docs/quagga.pdf`, Accessed in October, 2011.