Network virtualization: performance, sharing and applications

Anhalt Fabienne

NNT: 2011ENSL0630

Thèse

en vue d'obtenir le grade de

Docteur de l'Université de Lyon délivré par
l'École Normale Supérieure de Lyon
Spécialité : Informatique

Laboratoire de l'Informatique du Parallélisme
École Doctorale Informatique et Mathématiques

présentée et soutenue publiquement le 7 Juillet 2011 par

Mademoiselle **Fabienne Anhalt**

Titre :

Virtualisation des réseaux : performance, partage et applications

---

Directeur de thèse :		
Monsieur Paulo	Gonçalves	
Co-directrice de thèse :		
Madame Pascale	Vicat-Blanc	
Après avis de :		
Monsieur Kurt	Tutschku	
Madame Véronique	Vèque	
Devant la commission d'examen formée de :		
Monsieur Paulo	Gonçalves	Membre/Directeur
Madame Isabelle	Guérin-Lassous	Membre
Monsieur Daniel	Kofman	Membre/Président
Monsieur Kurt	Tutschku	Membre/Rapporteur
Madame Véronique	Vèque	Membre/Rapporteur
Madame Pascale	Vicat-Blanc	Membre/Directrice

Remerciements

Tout d'abord, je tiens à remercier mes directeurs de thèse qui m'ont permis d'effectuer mes recherches dans l'équipe RESO dans un environnement excellent. Toute ma gratitude va à Pascale pour m'avoir introduit à la recherche, pour m'avoir transmis de son enthousiasme, pour avoir partagé son savoir et son expérience avec moi, pour avoir su m'orienter, pour son investissement et pour m'avoir toujours donné confiance, soutien et assurance. Merci également à Paulo pour avoir accepté d'assurer la direction de ma thèse pendant cette dernière année, pour son grand engagement et sa disponibilité, et pour ses nombreux suggestions et conseils judicieux.

De plus, je voudrais remercier les rapporteurs et membres du jury pour avoir accepté d'évaluer cette thèse.

Mes remerciements vont aussi à Jean-Patrick Gelas pour son aide au début de ma thèse, et à Thomas Bégin, Isabelle Guérin-Lassous et Laurent Lefèvre pour leurs conseils avisés.

D'autre part, merci à mes co-auteurs, collègues de bureau, et amis sans faille, Guilherme et Dinil, pour avoir travaillé et vécu l'entière expérience d'une thèse avec moi. Merci à Dinil pour m'avoir enseigné sur la recherche et la vie; et à Marcos pour tous ses conseils précieux.

Merci à tous mes co-auteurs, en particulier Tram Truong-Huu, Johan Montagnat et Lucas Nussbaum.

De plus, merci à toute l'équipe RESO et l'équipe Lyatiss, en particulier Sébastien, Romaric, Ludovic, Marcelo, Hugo, Suleyman, Damien, Anne-Cécile, Pierre-Solen, Matthieu, Philippe, Armel, Abderhaman, Doreid, Ghanem, Landry et Attilio, pour les bons moments et l'aide que l'un ou l'autre m'a apporté au cours de ces trois années. Plus spécifiquement, merci à Augustin pour son soutien en anglais et à Olivier pour ses coups de main techniques.

Finalement, un chaleureux merci à Susanne, Sandra, Maykel Ange, Mahmoud, Leila et à tous mes amis qui m'ont accompagnée pendant ces années d'études, ainsi qu'à la famille Lambert, pour avoir toujours été ma famille à Lyon.

Mein größter Dank richtet sich an meine Eltern, Großeltern, Onkel Gerald, Ursula und Marleen, die mich in allen meinen Entscheidungen beraten und unterstützt haben und trotz meiner Abwesenheit immer die nächsten an meiner Seite waren.

Contents

List of Figures

List of Tables

Abstract

Virtualization appears as a key solution to revolutionize the architecture of networks, such as the Internet. The growth and success of the Internet have eventually resulted in its ossification, in the sense that ubiquitous deployment of anything into this network is hardly possible, thus impeding innovations. This is exactly where virtualization comes as a solution, by adding a layer of abstraction between the actual hardware and the 'running' network. These virtual networks can be managed and configured flexibly and independently by different operators, thus creating a competitive environment for stimulating innovations. Being 'de-materialized' in such a way, networks can be deployed on demand, configured, started, paused, saved, deleted, etc., like a topology of programmable objects, each representing a virtual switch, router or link. The flexibility introduced into the network provides the operator with options for topology reconfiguration, besides allowing it to play with the software stacks and protocols. Achieving such a high degree of decoupling, that leads to disruptive changes, is one of the ultimate goals of network virtualization—envisioned as a key to the 'future' of the Internet—but it is still far from reality.

Today, network virtualization has been realized in research testbeds, allowing researchers to experiment with routing, and interconnecting virtual computing nodes. The industry proposes virtual routers for network consolidation and saves in equipment cost. However, introducing virtualization in a production network such as those of the Internet raises several challenges, that have not yet been addressed. The additional layer interposed between the actual network hardware and the virtual networks is responsible for sharing the physical resources among the virtual networks. It potentially introduces performance overhead.

In this manuscript, we concentrate on these issues, namely the *performance* and the *sharing* in virtualized networks. These are in particular relevant, when the network data-plane is virtualized, for maximum isolation and configurability in virtual networks. Then, we investigate the *applications* of virtualized networks, sharing the physical network at the data-plane level. In this context, the contributions presented in this manuscript can be summarized as follows.

Analysis and evaluation of the impact of virtualization mechanisms on communication performance. In order to evaluate the impact of virtualization on the performance of a virtualized network, we analyze different technologies that allow to virtualize the data-plane, and we build a virtual router prototype using virtual machine techniques. The network performance of such a virtual router is evaluated in detail, using different software configurations [7] [9] [2]. The results show that the performance of the communication in virtual servers has improved over the last few years, to reach up to 100% throughput on 1 Gb/s network interfaces, thanks to optimizations in software. Hence, virtualization in software is a promising approach for experimentation, but for production networks such as the Internet, dedicated hardware is required, for reaching very high speeds (> 10 Gb/s).

Virtualizing the switching fabric. We propose a virtualized switch architecture that allows flexible sharing of the hardware resources of a switch among several virtual switches [5].

This virtualized switch architecture enables users to set up virtual switches with a configurable number of ports, dimensionable capacity per port and buffer sizes on top of the physical switch. In addition, each virtual switch can have different packet-scheduling and queuing mechanisms. A virtual switch scheduler controls the sharing of the physical resources among the virtual switches and provides performance isolation. The proposed architecture is evaluated through simulations. This architecture has been patented [VxSwitch].

Isolating and programming virtual networks. When virtualized, the network resources are shared among different virtual networks. We propose a virtual network service for controlling the amount of resources that is conferred to each virtual network, and ensuring performance isolation. This service consists in interconnecting nodes by a virtual network composed of virtual routers and links. The routers can be configured so that each virtual network controls which paths its traffic uses. The virtual links can be dynamically provisioned with bandwidth. The underlying physical resources control that each virtual link provides the configured bandwidth, thus ensuring a guaranteed service level [1]. This service is implemented in two ways, first using software virtual routers and links [HIPerNet], and second creating virtual routers and links using OpenFlow switches. Evaluations show that either approach can provide bandwidth guarantees, as well as routing configuration functionality for each virtual network.

Application of virtual networks. Finally, the virtual network service is applied to generalized virtual infrastructures, combining network and IT (Information Technology) virtualization. The users can request a virtual computing infrastructure, whose resources are interconnected through a controlled and isolated virtual network. This is a promising approach, e.g., for providing a network service in Clouds. We deploy such a service in the Grid'5000 testbed, and evaluate it using a large-scale distributed application [6]. The results show that the configuration of different service levels in the virtual network impacts directly on the application execution time [3] [12]. Hence, we validated the importance of control and isolation in virtual networks to provide predictable performance to Cloud applications.

Résumé

La virtualisation apparaît comme étant une solution clé pour révolutionner l'architecture des réseaux comme Internet. La croissance et le succès d'Internet ont fini par aboutir à son ossification : le déploiement de nouvelles fonctionnalités ou la mise à jour de son architecture ne sont guère possibles, entravant ainsi les innovations. La virtualisation apporte une réponse à cette problématique, en ajoutant une couche d'abstraction entre le matériel et le réseau tel qu'il est vu par les utilisateurs. De tels réseaux virtuels peuvent être gérés et configurés de manière flexible et indépendamment les uns des autres, par des opérateurs différents. Ceci crée donc un environnement compétitif qui stimule l'innovation. En étant dématérialisés de cette façon, les réseaux peuvent être déployés à la demande, configurés, démarrés, suspendus, sauvegardés, supprimés, etc., comme un ensemble d'objets eux-mêmes programmables, organisées dans une topologie, où chaque objet représente un commutateur, un routeur ou un lien virtuel. La flexibilité qui en résulte donne à l'opérateur la possibilité de configurer la topologie du réseau, et de modifier les piles protocolaires. Parvenir à un tel degré de découplage, permettant d'aboutir à des changements fondamentaux, est l'un des buts ultimes de la virtualisation des réseaux—qui est envisagée comme une clé pour le 'futur' de l'Internet—mais le concept est pour l'instant loin d'être réalisé.

Jusqu'à présent, la virtualisation du réseaux a été déployée dans des plateformes de test ou de recherche, pour permettre aux chercheurs d'expérimenter avec les protocoles de routage, notamment dans les réseaux interconnectant des nœuds de calcul, eux-mêmes virtuels. L'industrie propose des routeurs virtuels pour la consolidation des réseaux afin de réduire les coûts des équipements. Pourtant, dans le but d'introduire la virtualisation dans les réseaux de production comme ceux de l'Internet, plusieurs nouveaux défis apparaissent. La couche supplémentaire interposée entre le matériel et les réseaux virtuels doit prendre en charge le partage des ressources physiques entre les différents réseaux virtuels. Elle introduit potentiellement un surcoût en performance.

Dans ce manuscrit, nous nous concentrons sur ces problématiques, en particulier la *performance* et le *partage* dans les réseaux virtualisés. Ces deux questions sont particulièrement pertinentes, lorsque le plan de données du réseau lui-même est virtualisé, pour offrir un maximum d'isolation et de configurabilité dans des réseaux virtuels. Puis, nous examinons les possibles *applications* des réseaux virtuels, partageant le réseau physique au niveau du plan de données. Dans ce contexte, les contributions présentées dans ce manuscrit peuvent être résumées de la manière suivante.

Analyse et évaluation de l'impact des mécanismes de virtualisation sur la performance des communications. Afin d'évaluer l'impact de la virtualisation sur les performances d'un réseau virtualisé, nous analysons d'abord différentes technologies qui permettent de virtualiser le plan de données. Puis nous construisons un prototype de routeur virtuel en utilisant des techniques logicielles de virtualisation. La performance réseau d'un tel routeur virtuel est évaluée en détail, en utilisant des configurations logicielles différentes [7] [9] [2]. Les résultats montrent que la performance des communications des serveurs virtuels a augmenté durant les dernières années, pour atteindre jusqu'à 100% du débit maximum sur des interfaces réseau de 1 Gb/s, grâce aux optimisations logicielles. La virtualisation en

logiciel est donc une approche prometteuse pour l'expérimentation, mais pour un réseau de production tel qu'Internet, du matériel dédié est requis, pour atteindre de très hauts débits (> 10 Gb/s).

Virtualisation de la matrice de commutation. Nous proposons une architecture de commutation virtualisée, qui permet de partager les ressources matérielles d'un commutateur de manière flexible entre plusieurs commutateurs virtuels [5]. Cette architecture de switch virtualisée permet à des utilisateurs de mettre en place des commutateurs virtuels, instanciés au dessus du commutateur physique, chaque commutateur ayant un nombre de ports configurable. De plus, la capacité par port et la taille des tampons mémoire peuvent être dimensionnées. En outre, chaque commutateur virtuel peut disposer de mécanismes d'ordonnancement et de mise en file d'attente de paquets différents. Un ordonnanceur de commutateurs virtuels contrôle le partage des ressources matérielles entre les commutateurs virtuels et assure l'isolation des performances. L'architecture proposée est évaluée par des simulations. Un brevet a été déposé pour cette architecture [VxSwitch].

Isolation et programmation de réseaux virtuels. Étant virtualisées, les ressources réseau sont partagées par des réseaux virtuels différents. Pour contrôler la quantité de chaque ressource qui est attribuée à chaque réseau virtuel, nous proposons un service de réseaux virtuels. Ce service consiste en l'interconnexion de nœuds par un réseau virtuel, composé de routeurs et de liens virtuels. Les routeurs peuvent être configurés afin que chaque réseau virtuel puisse contrôler quel chemin est utilisé par son trafic. La bande passante peut être allouée dynamiquement aux liens virtuels. Les ressources physiques sous-jacentes vérifient que chaque lien virtuel fournit la bande passante configurée et qu'il assure un niveau de service garanti [1]. Ce service est implémenté de deux façons. En premier lieu, des routeurs virtuels logiciels interconnectés par des liens virtuels sont utilisés [HIPerNet]. Puis, des routeurs et liens virtuels sont créés en utilisant des commutateurs OpenFlow. Des évaluations montrent qu'avec chacune des deux approches, des garanties en termes de bande passante, ainsi que des fonctions de configuration du routage peuvent être fournies à chaque réseau virtuel.

Application des réseaux virtuels. Finalement, le service de réseaux virtuels est appliqué au contexte des infrastructures virtuelles généralisées, combinant la virtualisation du réseau et des nœuds de calcul. Les utilisateurs peuvent demander une infrastructure virtuelle de calcul, dont les ressources sont interconnectées par un réseau virtuel contrôlé. Cette approche est prometteuse pour les réseaux du *Cloud* ou nuage en français. Nous déployons un tel service dans la plateforme de test Grid'5000, et l'évaluons en utilisant une application distribuée à grande échelle [6]. Les résultats montrent que la configuration de niveaux de service réseau différents impacte directement le temps d'exécution de l'application [3] [12]. Nous validons donc l'importance du contrôle et de l'isolation dans les réseaux virtuels, dans le but de fournir des performances prévisibles à des applications Cloud.

1 Introduction

1.1 Virtualizing the network

The growth and popularity of the Internet have resulted in its ossification, in the sense that a ubiquitous deployment of any new technology into this network is hardly possible, thus impeding innovations [Handley, 2006]. This is exactly where virtualization comes as a solution, by adding a layer of abstraction between the actual hardware and the 'running' network [Anderson et al., 2005] [Keller and Rexford, 2010].

Virtualization, a technology introduced in 1973 [Popek and Goldberg, 1973], consists in using a single physical resource to host several virtual machines that share and access concurrently the actual hardware. Improvements in hardware and virtualization technologies such as Xen [Barham et al., 2003], KVM [Kivity et al., 2007] and VMware [VMW] have made server virtualization very commonplace. Their benefits include configurability, better resource utilization, mobility, isolation, and fault tolerance. Since similar benefits can be derived when virtualizing the network infrastructure, virtualization appears as a solution to the architectural issues of the Internet, and is promoted by many projects and research activities [Chowdhury and Boutaba, 2009]. Being 'de-materialized' from the actual network, virtual networks could be deployed on demand, configured, started, paused, saved, deleted, etc., like a set of programmable objects.

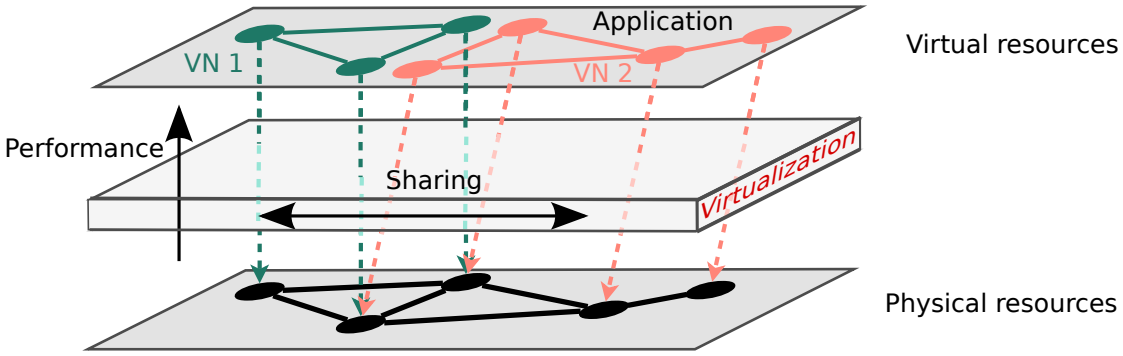


Figure 1.1: Two virtual networks, VN 1 and VN 2, allocated on a physical network infrastructure, through a virtualization layer.

Figure 1.1 illustrates the concept of a virtualized network, where virtual networks (VN) 1 and 2 are allocated atop the physical infrastructure. A virtualization layer, interposed between the virtual and the physical resources, provides the mechanisms for sharing the physical infrastructure among the virtual networks. Applications run on virtual resources unaware of the shared physical hardware. The performance that each virtual network achieves depends on the overhead introduced by the virtualization layer. In addition, depending on their implementation and performance, virtual networks can have different applications, and allow the introduction of new features into the network.

1.2 Problem and Objectives

We define network virtualization as follows:

Network virtualization refers to the transformation of N physical networks into M virtual networks. A network can be defined as a topology of routers and switches interconnected by links, or as an interconnection of ports at the scale of a single device (e.g., a router or a switch). All the virtual entities of a virtual network share the underlying physical resources, such as ports, CPU and memory (packet buffers, routing and forwarding tables). The physical entities control how much of each resource is attributed to each virtual entity, so that each virtual resource has the same or a subset of the functionalities (e.g., routing mechanism, queuing paradigm, etc.) of the physical resource where it is allocated. This functionality can be configured differently on a per virtual device basis. A virtual network is a logical entity that can be created and torn down on demand.

Depending on the layer of the network where virtualization is implemented, more or less flexibility can be obtained for virtual networks. Virtualizing the data plane, which deals directly with the traffic, storing and forwarding packets, allows not only to control the routing in each virtual network, but also the forwarding of packets. This provides maximum isolation between virtual networks as well as maximum configurability. Thus, QoS can be set up for each virtual network independently, e.g., by configuring its packet queuing and scheduling mechanisms. Such a level of configurability makes it possible to introduce new features into the network, and adapt each individual virtual network to its traffic.

Nevertheless, virtualizing a network also raises several new challenges. We identify three major challenges, namely *performance*, *sharing* and *applications*, as described below.

Performance. As illustrated in Figure 1.1, a vertical consequence of the virtualization layer is a potential *performance overhead*, meaning that a virtual network may require more resources to achieve the same performance of a physical network due to hardware abstraction and sharing mechanisms. Thus, virtualization may decrease the performance. The deeper the virtualization is performed—i.e., the more resources and functionality of the physical network are exposed—the more performance overhead can be expected. For example, virtualizing the data plane (the packet queuing and forwarding mechanisms) is more expensive than virtualizing only the control plane (e.g., the routing mechanisms), as it requires to push all packets up to the virtual resources for a per virtual network treatment.

Sharing. The controlled sharing of network resources is crucial to provide performance guarantees to virtual networks. It is up to the virtualization layer, as shown in Figure 1.1, to manage how these resources are shared. They can be divided equally among virtual networks, or be split into variable-size partitions. The latter means that a virtual network can obtain a different amount of the same resource over time, depending on the use of this resource by concurrent virtual networks. Such variable sharing can better exploit the

physical resources. As a drawback, it may not give the necessary performance guarantees to each virtual network. However, performance guarantees are crucial in contracts such as Service Level Agreements (SLAs). If we consider that a virtual network can be leased over a certain period as a service, performance is of great importance in determining the cost of the lease.

Applications. Thanks to their de-materialization, virtual nodes and links become themselves a service. They can be created on demand, and reconfigured with new capacity, new protocols, etc. A network of configurable virtual routers interconnected by virtual links, can be provisioned with capacity. Even latency can be controlled by the way virtual links are allocated atop the physical infrastructure. Such controlled virtual networks can have many applications, enabled by their possibility of customization in terms of functionality and performance.

From these challenges, we derive the following questions, summarizing open issues in network virtualization.

[Q_1] *How does virtualization impact network communication performance?*

[Q_2] *Where and how can virtualization be implemented in the network, in order to reduce the performance impact while enabling configurability?*

[Q_3] *When virtualizing a network, which are the resources that need to be shared?*

[Q_4] *How to provide deterministic performance and QoS to virtual networks?*

[Q_5] *What are the applications of controlled virtual networks?*

The objective of this thesis is answers these questions and hence address the challenges described above.

1.3 Contributions and thesis organization

The contributions described in this manuscript are organized in performance, sharing mechanisms, and applications of virtual networks. Table 1.1 positions each contribution in relation to the thematic and the type of virtualized component. On the one hand, we consider software routers, i.e., routers, implemented on classical servers, for virtualization. On the other hand, we propose to virtualize the data plane of hardware switches. Finally, the virtualization of the network as a whole, i.e., the routers and the links, is investigated. These contributions are further described as follows.

Analysis and evaluation of the impact of virtualization mechanisms on communication performance. To get a first insight into virtualization technologies and their *performance* in a network context, we start by building virtual routers inside virtual machines on commodity server hardware. Our goal here is to virtualize the data plane, so that virtual routers get full control over the packets, in order to maximize isolation and configurability. For realizing this, classical Linux routers are set up on virtual machines. In this setup,

	Performance	Sharing	Applications
Software router	Analysis and evaluation of a software virtual router (Chapter 3)	Evaluation of rate sharing in virtualized software routers (Chapter 5)	Customizable routing in virtual networks (Chapter 5)
Hardware switch		Sharing of the switching fabric (Chapter 4), and evaluation.	Use case of a configurable switching plane (Chapter 4)
Network		Bandwidth service in virtual networks (Chapter 5)	Isolation on-demand virtual infrastructures (Chapter 6)

Table 1.1: Contributions organized by performance, sharing and applications of software virtual routers, hardware virtual switches, and virtual networks as a whole.

each virtual machine gets virtual network interfaces. The virtual network interfaces of all virtual machines share the same physical network interfaces. Moreover, the packets go through an additional layer, before reaching the physical interfaces. We extensively analyze this performance overhead over time with different versions of Xen [7] and KVM [9], and conclude that data plane virtualization is costly in terms of processing overhead, especially when emulating the network interfaces [2]. However, virtualization technology has improved over time, as our results show. These contributions are detailed in **Chapter 3**. We conclude that the software virtual routers are promising for experimental networks, where high configurability is required, or for small edge networks. Nevertheless, for sustainable virtual routers on the Internet, it is necessary to consider dedicated equipment for virtualization, which leads us to our second contribution.

Virtualizing the switching fabric. As of the writing of this manuscript, manufacturers propose equipment with virtualized control planes, running multiple parallel routing instances (Layer 3). The virtualization of the data plane (Layer 2) is left to technologies such as VLAN and queuing (IEEE 802.1p). For pushing the de-materialization paradigm further and virtualizing also the switching fabric in order to obtain virtual routers with their own configurable data plane, we design VxSwitch, a virtualized switch architecture [5]. VxSwitch enables to *share* the physical switching resources among several virtual switches. Users can allocate virtual switches, specifying their number of ports, the exact amount of capacity of each port, and the amount of virtual buffer space. Moreover, each virtual switch has a configurable scheduler on each output port, which dequeues packets from the crosspoints. The physical switch controls the different schedulers and virtual buffers to ensure performance isolation. In this way, QoS can be finely configured in each virtual switch, as the virtual queues can be sized, and the schedulers programmed with custom policies. The architecture of VxSwitch has been patented [VxSwitch] and is detailed in **Chapter 4**. Simulations evaluate the relative performance impact of virtualizing the switching fabric, and use cases demonstrate the new features that VxSwitch can introduce

to the network.

Isolating and programming virtual networks. A virtual network consists in a topology of virtual routers and virtual links that can interconnect virtual end-hosts. Our contribution, detailed in **Chapter 5**, consists in a provisioning service for virtual networks. It ensures performance isolation based on the controlled *sharing* of the physical network. The goal is that end-hosts, connected to one other through a virtual network, obtain deterministic service levels. The service combines programmable routing inside virtual routers, and configurable bandwidth on virtual links [1]. In order to implement such a service, traffic control mechanisms are evaluated on virtual links—between virtual routers or virtual routers and virtual end-hosts—for providing performance isolation. Service modules have been implemented to manage the automatic configuration of virtual networks, by programming virtual routers and configuring virtual links according to service requirements [HIPerNet] [13]. The modules are able to interact with software routers and with OpenFlow switches [McKeown et al., 2008], to set up virtual routers and virtual links.

Application of virtual networks. The success of Clouds [Rosenberg and Mateos, 2010] has brought the virtual infrastructure concept to the foreground. A critical issue in virtual infrastructures is the network, since it is responsible for communication among virtual nodes and can impact the performance of distributed applications. Hence, we propose to interconnect virtual infrastructures through isolated virtual networks, such as those provided by the service that we defined. By using such virtual networks inside virtual infrastructures, the computing and storage nodes get predictable communication performance. In addition, the virtual network can be reconfigured over time, adding or removing virtual links and routers, adjusting their bandwidth and routing, so as to continuously satisfy the requirements of the applications running on the virtual infrastructure. We first propose the application of the virtual infrastructure concept for executing isolated distributed experiments on Grid'5000 [6], the French national Grid and Cloud research platform. We then validate the concept and demonstrate the benefit of network control in virtual infrastructures through the execution of a distributed application over a virtualized infrastructure. The virtual network is reconfigured over time to meet the exact application requirements. The results show that an efficient network configuration can reduce the application execution time [3] [12]. This contribution is described in **Chapter 6**.

2 Network virtualization: techniques and applications

Abstract. The goal of network virtualization is to enable several applications to run independently and simultaneously over a differently shared physical network. Users can then enjoy isolated virtual networks with customized configurations. This chapter surveys current research towards reaching these goals. It discusses network virtualization concepts, implementations and evaluations, involving different network layers and equipment types. The main issues identified in many current network virtualization techniques are: i) they do not enable performance and configurability at the same time, and ii) control in the resource sharing mechanisms is mostly limited or absent.

2.1 Introduction

Virtualization has been applied to networks for over a decade to help virtualize connectivity, i.e., the links, with technologies such as VLAN and VPN. However, the functionality of a network relies on its nodes, the routers and switches, which decide how traffic is routed and switched across the topology. Therefore, a more recent approach has been to virtualize also these devices, in order to enable the setup of different virtual networks with their own functionalities atop a common physical infrastructure. Considering virtualization in this new light, it appears as one of the key solutions to make networks such as the Internet more flexible and remove their current architectural locks [Anderson et al., 2005].

In this Chapter, we survey the different technologies that have been proposed for virtualizing a network or part of its components. For each technology, we identify the type of resource that is virtualized, how it is virtualized and what its application is. The goal of this chapter is to provide an overview of the current status of network virtualization, serving as a toolbox to build virtual networks; and identify gaps in current research and technologies that prevent them from heading towards a new sustainable network architecture.

The rest of this chapter is organized as follows. In the next section, we define a formalization of virtualization, that is used throughout the rest of the chapter to describe the virtualization mechanisms of each discussed technology. Thereafter, Sections 2.3 and 2.4 describe the technologies for virtualizing respectively the network connectivity and the network functionality. At the end of each of these two sections, a table summarizes all discussed technologies, their goal and the associated virtualization function. Section 2.5 discusses current applications of the previously discussed virtualization technologies, whereas Section 2.6 positions this manuscript in the context of the discussed research and technologies. Finally, Section 2.7 concludes this chapter.

2.2 Formalization of virtualization

In general, 'virtualization' means abstracting a resource from the actual hardware, resulting in virtual resources. A virtual resource, a partition of one or a combination of several physical resources, inherits from the functionality of the physical resource(s).

2.2.1 Types of transformation

Representing virtualization as a function transforming a resource, it can be defined in different ways. We define the three types of transformation, associated to network virtualization, as *sharing*, *aggregation* and *concatenation*:

Sharing: The hardware of a physical resource can be shared among N virtual resources, as represented in Figure 2.1(a). For example, a router can be shared by several virtual routers. These virtual routers can share the ports and hence the capacity of the physical router. The goal of sharing is to use the hardware of a device more efficiently, and to allow to run different configurations in parallel on a single device. Different virtual routers hosted on the same device can run different routing protocols and route the traffic of several virtual networks independently.

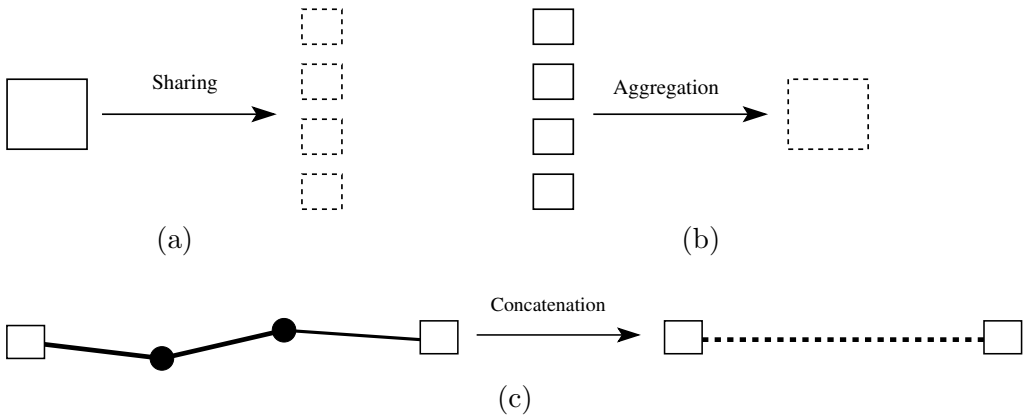


Figure 2.1: Transformation of resources by virtualization (Solid lines represent physical resources, dashed lines represent virtual resources.): (a) Sharing: transforming 1 resource into n virtual resources; (b) Aggregation: transforming n resources into 1 virtual resource; (c) Concatenation: transforming n organized resources into 1 virtual resource.

Aggregation: Aggregation consists in building a virtual resource using several physical resources, as represented in Figure 2.1(b). Typically, different memory devices of computers are presented to the operating system as a single virtual memory. In general, the goal of aggregation is to provide a virtual device that can achieve higher performance than a single physical device.

Concatenation: Similar to aggregation, concatenation consists in building a virtual resource from several physical devices, as represented in Figure 2.1(c). However, concatenating resources consists in assembling and organizing them in a particular way. A virtual link can be build as a concatenation of physical links (i.e. a path). This hides complexity, as users of a virtual link do not need to care about intermediate nodes on the actual physical path.

2.2.2 Formalization

We formalize virtualization as a function, transforming resources in one of the ways described above. Hence, we infer three virtualization functions, namely V_s, V_a and V_c, that respectively *share*, *aggregate* and *concatenate* resources to build virtual resources. We define these functions in the following way:

1. $V_s : r \rightarrow \{r_{v_1}, r_{v_2}, ..., r_{v_n}\}$: The resource r is shared by a set of virtual resources r_{v_i}, each of which inherits from the functionality of r, and has a subset of its capacity.

2. $V_a : \{r_1, r_2, ..., r_n\} \rightarrow r_v$: A set of resources r_i are aggregated in a virtual resource r_v that has the aggregate capacity of all r_i and inherits from their functionality.

3. $V_c : [r_1, r_2, ..., r_n] \rightarrow r_v$: An ordered set of resources r_i is concatenated to form a virtual resource r_v with inherited functionality of the physical resources. Its capacity corresponds to the capacity of the smallest resource r_i.

Note that these functions can be applied on physical resources, as well as on virtual resources. Conceptually, a virtual resource can itself be virtualized, hence combining several of the described functions. Combining for example V_a and V_s is formalized as $V_a \circ V_s : \{r_{p_1}, r_{p_2}, ..., r_{p_n}\} \rightarrow \{r_{v_1}, r_{v_2}, ..., r_{v_m}\}$, where multiple physical resources r_{p_i} are aggregated to a virtual resource, which is then re-partitioned into multiple virtual resources r_{v_i}, as represented in Figure 2.2.

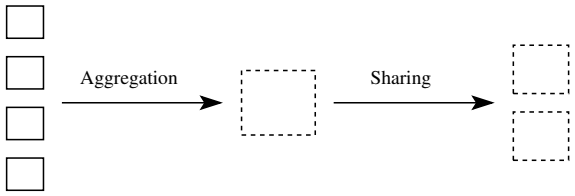


Figure 2.2: Combined aggregation and sharing: 1) transforming n resources into a virtual resource, and 2) transforming the virtual resource in to m virtual resources.

2.2.3 Applying the formalization in this chapter

In this chapter, we survey the different concepts and technologies to virtualize the different components of a network, the links, the interfaces, the routing and control mechanisms. For each described concept or technology, we identify:

- x: the element or set of elements that is virtualized;
- V: the virtualization function (sharing, aggregation, concatenation or a combination of them) applied to it;
- y: the resulting virtual element or set of virtual elements.

and represent the type of virtualization in the form of

$$V : x \rightarrow y$$

We chose this representation to enable the reader to identify clearly for each technology which elements are virtualized, how they are virtualized, and what is the result of the virtualization.

2.3 Virtualizing Connectivity

The connectivity of a network relies on links that interconnect different routing and switching devices in a topology, using their ports. Connectivity also refers to the internal links interconnecting pairs of ports inside a single router or switch, as shown in Figure 2.3. A link, between devices or internal, is always defined by an interconnection of two ports. Hence, virtualizing the connectivity of a network technically consists in virtualizing its ports.

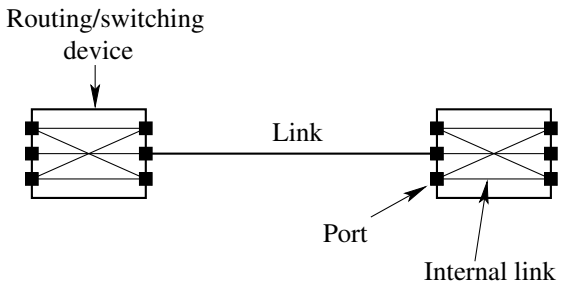


Figure 2.3: Connectivity of a network: links interconnecting routing and switching devices, and links that are internal to a single device.

Virtualizing network links and ports allows one to partition the network into multiple virtual networks with logically isolated links. Each virtual network offers individual connectivity, but functionalities such as routing and switching control remain globally defined. Hence different virtual networks hosted on the same physical network equipment cannot run their own control algorithms.

This section describes the different techniques that enable to virtualize the elements of the network, that are responsible for connectivity.

2.3.1 Virtual Local Area Networks

Virtualizing the Local Area Networks (LAN) into VLANs, standardized as 802.1Q by IEEE [IEE], is a widespread approach to separate networks into several logically isolated networks [VLAN, 2005]. For a given switch, only ports belonging to the same VLAN can communicate with one another. This has the advantage of limiting the broadcast domain of a VLAN to only links that belong to the same network, decreasing broadcast traffic and improving performance and security. In addition, a VLAN can be easily reconfigured when the network topology changes. For example, a new link can be added to the VLAN or moved from one VLAN to another without the necessity to intervene physically on the equipment.

Typically, a VLAN is identified based on *ports* (layer 1), *MAC addresses* (layer 2), *IP addresses* (layer 3) or even *protocols*. When a packet enters a switch, it is tagged depending on these criteria, and henceforth its tag determines to which VLAN it belongs. A port-based VLAN consists in a set of physical ports. For a given switch, these ports cannot communicate with ports that belong to other VLANs. Hence, a port-based VLAN allows one to partition a switch into several virtual switches, each exploiting a dedicated set of physical ports.

On the contrary, a layer 2 or 3 VLAN is abstracted from the physical network. No matter through which port a packet enters a switch, it obtains a VLAN tag based either on its IP or MAC address. Thus even if a host is moved from one port to another, or from one switch to another, it will stay in the VLAN, obtain connectivity with other nodes of the same VLAN, and remain isolated from other VLANs. Consequently, this approach is more flexible than port-based VLANs.

The virtualization in VLANs is implemented at the forwarding table, which is part of the forwarding control module of a switch as depicted in Figure 2.4. The entries of the forwarding tables contain an additional VLAN identifier that decides on the output port

a packet takes. Although the functionality, e.g., the queuing and forwarding mechanism,

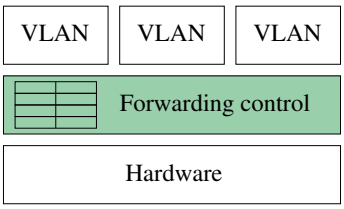


Figure 2.4: Switch configured with several VLANs.

of each VLAN is the same, some exceptions allow to configure different VLANs with their own functionality. For example, spanning tree protocol[1] (STP) [STP, 2004] can be set up per VLAN on some switches.

To summarize, the switch resources that are virtualized depend on the type of VLAN (layer 1, 2, 3). Port-based VLANs divide a switch into subsets of ports, where each subset corresponds to one VLAN. For layer 2 or 3 VLANs, the physical ports can be shared by several VLANs. Such a shared port is called *trunk port*. Hence, we formalize the virtualization of switches' ports for setting up VLANs as follows:

$$V_s : Trunk\ port \rightarrow Set\ of\ virtual\ ports$$

V_s, means that the trunk port is shared among a set of virtual ports.

Despite their benefits, VLANs also have some drawbacks. The fact that different IP networks running inside their own VLANs may share the same physical switches and links can lead to performance interferences between these IP networks. This prevents one from managing VLANs independently, when it comes to congestion or failures on a physical link or switch [Bin Tariq et al., 2009].

For giving a VLAN performance guarantees, many of current switches have several queues per port. The number of queues varies from eight up to thousands in recent equipment. On such switches, a VLAN port can be attributed a dedicated queue of the physical port. Priorities or even rates can be configured per queue and hence per VLAN. The scheduling of the different queues allows such priorities. An example of a switch port with several queues where each is used by a different VLAN is represented in Figure 2.5. Thus, VLANs can benefit from basic QoS such as prioritization and sometimes traffic shaping, depending on the implemented scheduling algorithms. However, the number of VLANs with QoS is limited to the number of available queues. Moreover, relative scheduling, such as priority scheduling, leads to VLAN interdependencies in terms of performance. On the other hand, if VLANs are configured with absolute rates, the capacity of the links may not be fully used at some moments, or some VLANs may experience contention, while others do not use their full capacity.

2.3.2 Virtual Private Networks

Virtual Private Networks (VPN) enable communication over a public network such as the Internet in a secure way. In a VPN, physical paths (concatenations of possibly several

[1] STP prevents switching loops.

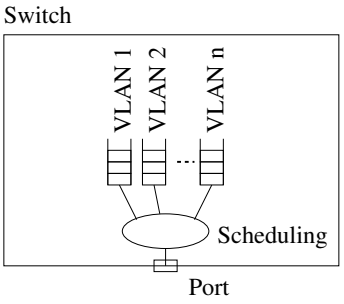


Figure 2.5: Switch port shared by several VLANs, where each VLAN has access to a separate queue.

links) throughout the network are presented to the VPN user as a single virtual link. Hence, we formalize the virtualization function for obtaining a VPN from a *concatenation* of physical links as follows:

$$V_c : Sequence\ of\ links \rightarrow Virtual\ link$$

A user connecting to a remote host through a VPN gets the impression that he uses a single link. Thus, remote hosts profit from the same connectivity as if they were in the same LAN, and can exchange data over the public network.

The physical path underlaying the VPN virtual link is shared with other traffic over the Internet, but the VPN traffic is logically isolated. In Figure 2.6, two VPNs are shown as an example, where VPN links consist of tunnels set up either between the provider edge

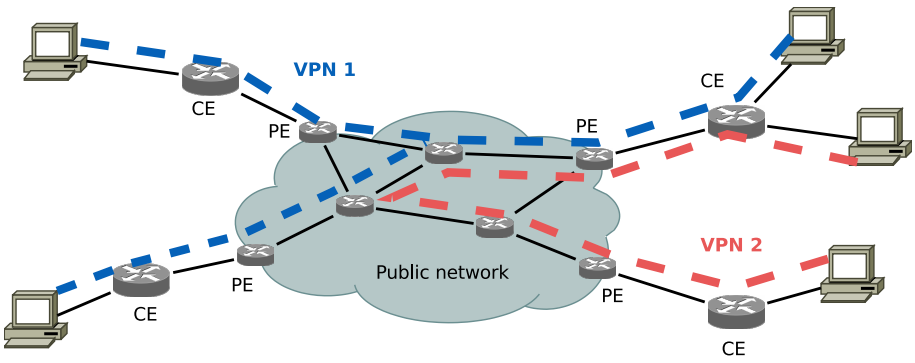


Figure 2.6: Two VPNs over a public network.

routers (PE), or between the customer edge routers (CE).

2.3.2.1 Implementation techniques

Although all VPNs use tunneling mechanisms to build virtual links over the Internet, they are implemented using technologies that operate at different layers of the OSI model.

At layer 3, one can use tunnels created using IPsec [Kent and Atkinson, 1998], IP-in-IP [Simpson, 1995] or GRE [Farinacci et al., 2000] to implement the VPN between the CE routers or between the PE routers. Another common technology to implemented VPNs in the provider network is the Multi-Protocol Label Switching (MPLS) [Rosen

et al., 2001] [Muthukrishnan and Malis, 2000], which establishes circuits termed as Label Switched Paths (LSP). Packets are routed inside the MPLS network along these LSPs by so called label switched routers (LSR) that take forwarding decisions based on labels. These labels are attributed to the packets at the network edge, and are changed (namely *switched*) at each LSR in the MPLS core network. A particular implementation, VPLS (Virtual Private LAN Service) [Lasserre and Kompella, 2007] allows one to set up tunnels with point to multi-point connection.

VPNs can also be implemented directly at layer 1 [Fedyk et al., 2008]. Layer 1 VPNs are based on the Generalized Label Switching Protocol (GMPLS) [Berger, 2003], and can be provisioned with a wavelength, or a partition of timeslots [Wu et al., 2006].

2.3.2.2 Provisioning and QoS

Provisioning is an important feature to deliver performance guarantees to a VPN, as it shares network segments with other traffic. QoS has to be configured at the edge of the network, on the access switches and routers, as the core is generally overprovisioned to prevent congestion. The classical solution also called the customer-pipe model, is to allocate point-to-point provisioned links that interconnect all the VPN endpoints in a full-mesh topology. As it is hard to specify QoS in this model under VPNs with large numbers endpoints, the hose-model has been proposed [Duffield et al., 1999]. This model, it does not guarantee a bandwidth between each two end-points, but guarantees an aggregate in/out bandwidth on each end-point called hose. Figure 2.7 represents a VPN based on the hose model [Duffield et al., 1999]. The interconnection between end-points inside the

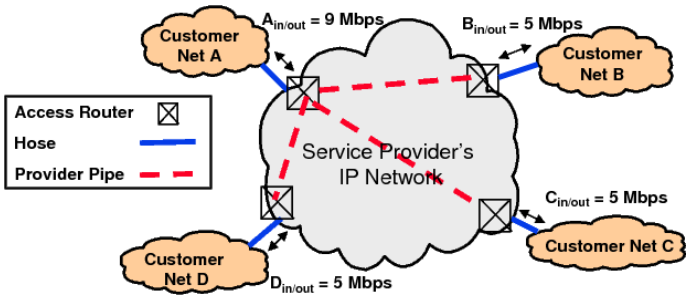


Figure 2.7: VPN based on the hose model [Duffield et al., 1999].

service provider's network must guarantee that each hose gets the required bandwidth towards all other end-points.

For giving such guarantees, the different VPNs must be efficiently mapped to the service provider's network. In Ethernet Metropolitan Area Networks, this can be performed by mapping VPNs to spanning trees. A spanning tree is a sub-graph of the network, connecting nodes without loops through shortest paths. Solutions have been proposed to map several VPNs to several spanning trees created with Multiple Spanning Tree Protocol (MSTP) [MSTP, 2005]. Optimizations in the creation of the multiple spanning trees and the mapping of the VPNs to them have resulted in improving load balancing while providing VPNs with bandwidth guarantees between endpoints [Brehon et al., 2007].

2.3.3 Overlay networks

In an overlay network, the nodes are interconnected by links spanning over a path of physical links. The intermediate hops on the path are transparent to the overlay nodes[2]. The basic property of an overlay network is that it virtualizes the connectivity by making paths appear as links. Hence, in an overlay, a virtual link is obtained by concatenating physical links. We formalize this type of virtualization as follows:

$$V_c : Sequence\ of\ links \rightarrow Virtual\ link$$

The goal of overlay networks is to delegate some of the network functionality to the edges. This is possible as overlays can direct traffic through specific nodes called overlay nodes, e.g., fully configurable end-hosts. Users can configure the overlay nodes to perform specific operations on the traffic (e.g., monitoring, crypting, coding, firewalling, etc.), a feature that has been used differently to introduce new functionalities to the network. A few examples are given here.

One of the first overlay strategies was Detour [Savage et al., 1999], an overlay network composed of Internet end-hosts serving as routers, interconnected by tunnels. The routers can control the network and make packets take more efficient routes, in terms of delay and packet loss rate.

Also seeking to improve network QoS, OverQoS is an overlay architecture that aims at fairly sharing links between customers [Subramanian et al., 2003]. The rate of a virtual link is determined according to loss rate and bandwidth of the physical link and a specified maximum loss level. Customers get a minimum guaranteed service for the sum of their flows, without impacting the underlying best-effort traffic.

Optimizing bandwidth in overlay networks has been proposed in [Lee et al., 2008]. Bandwidth is monitored between the different nodes of the network, and the monitored data is used to find alternative overlay-paths through relay nodes in case of a bottleneck. Alternative paths are chosen according to the capacity improvement compared to the original path.

One application of overlay networks are for example content delivery services. Servers that deliver content over the network in a multicast way are deployed as an overlay network. Given that content should be close to clients, that network cost should be reduced, but that the type of user requests are not know in advance, it is difficult to choose the right server locations when deploying a content delivery service in a big network. In [Busson et al., 2007], the authors propose a stochastic-geometry based approach to determine appropriate locations for deploying servers, minimizing the deployment cost.

Overlay networks are also used to deal with network failures. As an example, Resilient Overlay Networks (RON) are formed by a set of nodes located inside different routing domains on the Internet [Andersen et al., 2001]. They communicate via a WAN-overlay to detect failures or performance degratation of the underlying paths. Packets are forwarded through RON nodes to route around failures or other detected link problems. The rerouting with RON is much faster (several seconds) than the recovery of normal routing protocols (several minutes). Figure 2.8 illustrates this concept, where all traffic from A to B is redirected over the overlay routing node C, when the direct link from A to B fails.

Platforms for automatic deployment and management of overlays have been proposed. The X-Bone [Touch, 2000] is a distributed system allowing automatic deployment and

[2] A VPN is for example a particular type of overlay.

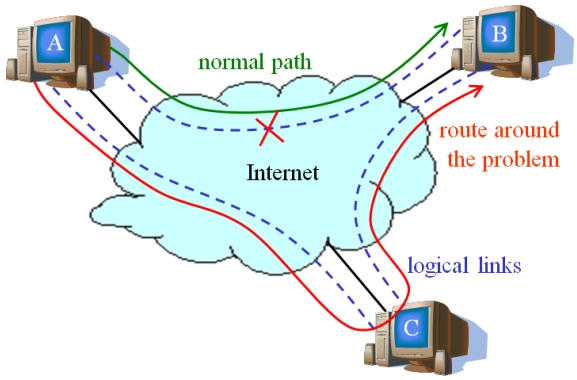


Figure 2.8: Routing in an overlay network [Jennifer Rexford, 2005].

management of overlays. A user specifies a request about the network he wants to create, and the X-Bone automatically discovers the necessary resources, deploys and monitors the requested overlay. It supports IPsec and dynamic routing, and recursive overlays (overlays over overlays). Two levels of IP encapsulation allow a single node to represent several distinct nodes on the same overlay and permit existing dynamic routing. Fault detection mechanisms make X-Bone failsafe.
One of the largest overlays, PlanetLab [Bavier et al., 2004], comprises virtual machines distributed over the world, that researchers can reserve and configure for network-wide experiments. The virtual nodes are implemented with VServer [VSR], and can be used as end-hosts or configured as network nodes. Links are not only concatenated, but also shared. Therefore, we formalize the virtualization as follows:

$$V_s \circ V_c : Link \rightarrow Virtual\ link$$

The rise of overlay networks shows the need for configuring the network for testing routing protocols, for innovations, which is not possible inside the current Internet, whose devices are closed boxes. Overlays are one of the technologies, that first used virtualization to enable network configuration and programmability.

2.3.4 Virtual machine connectivity

Server virtualization allows a physical host to be virtualized, and its resources to be shared among multiple virtual machines. To provide network access to virtual machines, the network stack of the physical host needs to be virtualized as well. The main difficulty in realizing this is to handle the concurrent I/O (Input/Output) requests of the different virtual machines on the same network interface card (NIC). A complex multiplexing mechanism is needed to assign packets from virtual machine network interfaces to physical interfaces and vice versa.

Server virtualization technologies can be classified into three main categories, namely *OS-level virtualization*, *full virtualization* and *paravirtualization*. The following sections describes how the network stack is virtualized by these technologies.

2.3.4.1 OS-level virtualization

A lightweight way of virtualizing a server is doing so at the operating system (OS) level. A virtual machine consists in a confined user space environment for executing applications

on top of the host kernel. The advantage is that virtual machines access hardware just like applications do. On the other hand, as a virtual machine runs the OS of the host and generally does not have its own kernel, isolation between virtual machines is limited.

The main OS-level virtualization technologies include OpenVZ [OVZ], Linux VServer [VSR], Virtuozzo [VRZ], FreeBSD Jails [JAI] and User-Mode-Linux (UML) [UML]. VServer and FreeBSD Jails are also referred to as *container-based* virtualization technologies. In this terminology, a virtual machine is a container that hosts an OS userspace and shares the host kernel with other virtual machines. Virtual machines access the network through sockets bound to dedicated IP addresses [Soltesz et al., 2007] [Kamp and Watson, 2000]. In both VServer and FreeBSD Jails the virtual machine user cannot configure the routing table or change the IP addresses of the virtual machine network interfaces on the fly. Hence, different virtual machines share the network stack in the same way that applications share the network stack on a classical Linux OS. That is, different virtual machines access the same NIC through different TCP/UDP sockets, and the network stack is virtualized as:

$$V_s : TCP/IP\ stack \rightarrow Set\ of\ sockets$$

Each virtual machine hosted on a VServer can get specific network bandwidth. The traffic of the different virtual machines sharing a physical NIC is shaped using the classical Hierarchy Token Bucket (HTB) of the Linux Traffic Control tools [TC]. The main advantage of OS-level virtualization in terms of network performance is that, without incurring any CPU overhead, virtual machines can achieve the same network throughput than on a classical (non virtualized) Linux system, which is not the case with other virtualization technologies.

Another OS-level virtualization technology, OpenVZ [OVZ], works basically the same way as VServer, but an alternative option on the network configuration allows virtual machine users to set up their own IP addresses on specific virtual Ethernet interfaces. These interfaces are bridged to the physical interface. Here, virtualization takes place at the Ethernet NIC driver level, so that the virtualization function applied to the driver exposes a virtual Ethernet NIC to each virtual machine:

$$V_s : Ethernet\ driver \rightarrow Set\ of\ virtual\ ethernet\ drivers$$

While virtualizing the Ethernet driver, gives more flexibility to virtual machines—they can have configurable virtual NICs—its drawback is network performance. Network throughput on such virtual interfaces is less than with the default configuration, using different sockets.

Another OS-level virtualization technique, different from those described above, is User-Mode Linux (UML) [Dike, 2001] [UML]. It enables virtualization by running Linux kernels in userspace. The NIC is virtualized by exposing TUN [TUN] interfaces to virtual machines. UML can fully virtualize the kernel network stack, exposing a virtual network stack to each virtual machine. Hence, the virtualization function in UML is:

$$V_s : Kernel\ network\ stack \rightarrow Set\ of\ virtual\ network\ stacks$$

The additional processing causes a huge impact on performance. UML virtual machines achieve less than half of the throughput of a classical Linux server [Koh et al., 2006]. In such systems, rate can also be controlled using software queuing and traffic control mechanisms.

2.3.4.2 Full virtualization

Full virtualization is based on a *hypervisor* that manages virtual machines and their access to the hardware of a host. Common full virtualization technologies include KVM [Kivity et al., 2007], VMware [VMW], VirtualBox [VBX], Hyper-V [HYV], and Xen (with a hardware virtual machine option) [Pratt et al., 2005]. Contrary to OS-level virtualization, full virtualization presents completely emulated NICs to virtual machines.
Emulation is a technology, that can abstract a device from its hardware, presenting it as an *emulated device* to a virtual machine which perceives it as a physical device. When a machine accesses an emulated NIC, all instructions are translated to actual hardware instructions, and executed on the NIC, as represented on the left part of Figure 2.9. Hence, emulated NICs expose the same functionality than physical NICs, and virtual machines can use native NIC drivers to access these emulated NICs. A common emulation technology used for this purpose is Qemu [QEM]. In summary, emulation virtualizes the whole kernel network stack in order to expose several full-featured NICs to virtual machines. Hence, several virtual machines have each their own virtual network stacks, that result from sharing the Kernel network stack:

$$V_s : \textit{Kernel network stack} \rightarrow \textit{Set of virtual network stacks}$$

These full virtualized systems use hardware virtualization support of the CPU, running unmodified OSes inside virtual machines and providing improved virtual machine isolation. However, it does not improve the network performance as NICs are emulated and emulation is very costly in terms of processing. Taking the example of Kernel Based Virtual Machines (KVM), the NIC is emulated using Qemu and exposed to the virtual machines. As this requires the translation of all instructions between the emulated NIC and the hardware NIC, very high CPU overhead is expected, which can rapidly become a bottleneck to virtual network performance, besides overall virtual machine performance during high network loads. This virtual machine performance is quantified in Chapter 3. Therefore, many full virtualization solutions come now with special network drivers, and perform paravirtualization.

2.3.4.3 Paravirtualization

It requires to modify the virtual machine OS to integrate special drivers to access the hardware, especially disks and NICs, as represented on the right part of Figure 2.9 Paravirtualization is similar to full virtualization with some changes. One of the first paravirtualization technologies was Denali [Whitaker et al., 2002], using a Virtual Machine Monitor (VMM) or hypervisor that virtualizes the hardware and exposes them to the virtual machines. The most commonly used paravirtualization solution today is Xen [Barham et al., 2003]. Note that Xen also offers the option to be used in full virtualization mode, running so called Hardware Virtual Machines (HVM) [Pratt et al., 2005]. Yet, by default, it performs paravirtualization. But also KVM [Kivity et al., 2007] allows paravirtualization with the specialized *virtio* driver [Russell, 2008], that virtualizes the access to network and storage devices. Other technologies, like VMware and VirtualBox also propose to patch the virtual machine OSes to update the I/O drivers in virtual machines, and achieve better I/O performance. In a paravirtualized system, the physical NIC is exposed through special virtual NIC drivers to the virtual machine. Hence, the virtualization function is applied to the NIC driver and several virtual NICs are exposed to the virtual machines:

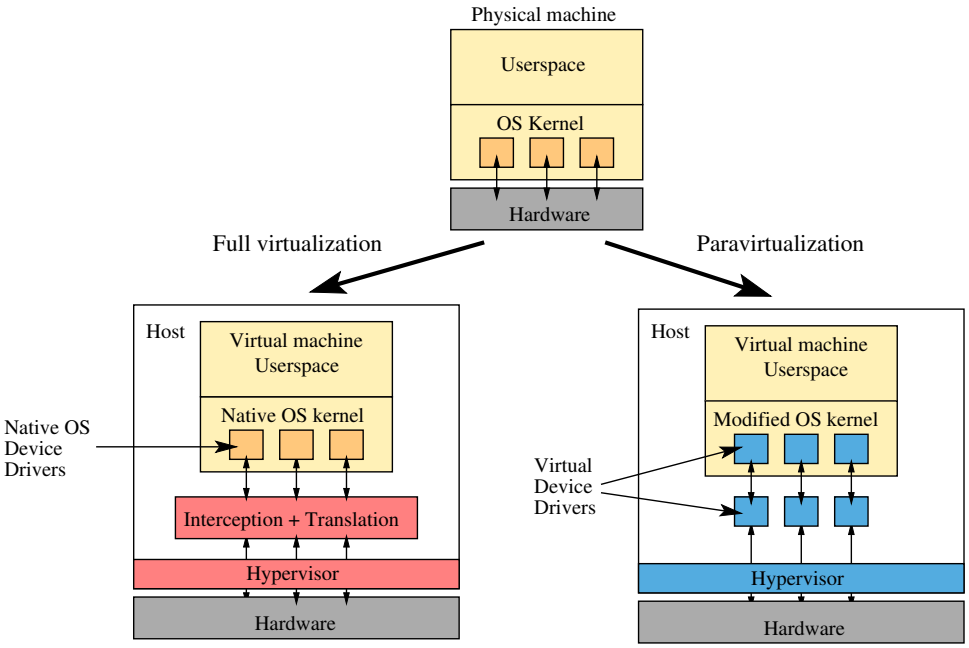


Figure 2.9: Full virtualization and paravirtualization.

$$V_s : \textit{NIC driver} \rightarrow \textit{Set of virtual NIC drivers}$$

The main performance improvement, compared to full virtualization is that instructions do not need to be translated one by one to the physical NIC. Instead, packets are multiplexed to the physical NIC with a software bridge or router. While this significantly reduces CPU overhead, it is still a costly procedure due to the multiplexing, the interrupts of packets at the NIC and at the virtual NICs, and the additional copy. In fact, a packet is copied from the NIC to the host, from where it is copied to a special shared memory that can be accessed by the authorized virtual machine [Chisnall, 2007].

Xen appears to be the hypervisor that has been most extensively evaluated by the research community, resulting in numerous I/O performance improvements over the different versions. To give an idea, network throughput on a virtual machine increased from roughly below 250 Mb/s per 1-Gigabit NICs using Xen 2.0 [Menon et al., 2006], to up to 941 Mb/s on today's version of Xen [7]. A detailed discussion on this evolution and on paravirtualization technology is given in Chapter 3.

With the performance issues of the software switch being obvious, direct access to the network interface by virtual machines, called *passthrough I/O*, would be a more efficient solution than multiplexing several virtual machine interfaces on one physical interface. However, passthrough I/O enables only one virtual machine to use the physical NIC at a time. Virtual Passthrough I/O (VPIO) has been proposed to allow different virtual machines to share a physical interface [Xia et al., 2009]. It is a tradeoff between dedicating a NIC to a virtual machine and multiplexing virtual NICs to hardware NICs. The physical interface is used in passthrough mode as much as possible and handed over between virtual machines when needed. If passthrough is not possible at a given time, the virtual machine monitor takes care of the processing.

Another solution to improve the network performance of virtual machines is to 'replace' the software switch that interconnects virtual machines on a host with a hardware switch. This means that all virtual machine traffic is by default sent out of the host towards a classical hardware switch. This switch then forwards the traffic towards the destination host, which can be a virtual machine located on the same host of the source of the traffic [Congdon et al., 2010]. The advantage is that not only forwarding, but various costly operations, such as address learning, access control, or other can be handled by the hardware switch.

Another recent software network virtualization solution is Crossbow [Tripathi et al., 2009], proposed initially for OpenSolaris. Crossbow allows one to build virtual NICs using so-called *virtualization lanes*, which are network stacks with dedicate receive and transmit buffers (called rings). Packets are directly classified on layer 2 into the right virtualization lane. These virtualization lanes can be implemented in software, but this does not enable any performance isolation or fairness in the resource sharing. Therefore, NICs with virtualized hardware are required, where each virtualization lane has its own receive ring, and hence does not impact the packet rate of other lanes.

Reviewing these different software virtualization techniques, we note that passthrough is the most efficient solution as it causes no performance overhead. This brings us to the conclusion that specialized NICs are needed, which can be directly accessed by several virtual machines in parallel.

2.3.5 Virtualized NICs

When talking about hardware-assisted virtualization, such as KVM [Kivity et al., 2007], it generally means that the CPU is virtualized, to give several parallel virtual machines independent and secure access to it. While this is meant to improve computing performance it does not necessary improve the network performance, as network I/O does not benefit from CPU virtualization. Therefore, NICs with hardware virtualization support have been developed.

Arsenic is such a type of NIC, with an extended interface to the OS and to applications [Pratt and Fraser, 2001]. It exposes virtual interfaces to the applications, where each has its own transmit and receive queue. This discharges the OS, as the multiplexing between virtual interfaces and the real physical interface is implemented directly in the Arsenic NIC. In addition, traffic shaping and scheduling enable the control of application traffic on the transmission queue.

More recently, Intel has implemented solutions to dedicate transmit and receive queues to virtual machines on 1 to 10 Gb/s Ethernet NICs. VMDq (Virtual Machine Device queues) is one such technology dedicating one queue to a virtual machine, handled by a dedicated CPU core [VMDq, 2008]. An incoming packet triggers an interrupt and is directly sent to the right queue and handled by the CPU core dedicated do the queue. This replaces the costly multiplexing mechanisms, that would otherwise take place in the hypervisor. Instead, the workload is spread among multiple cores. In summary, with VMDq, the NIC is shared into virtual NICs, where each has a dedicated queue. The virtualization function of VMDq can hence be formalized as:

$$V_s : NIC \rightarrow Set\ of\ virtual\ NICs$$

An even more efficient solution is SR-IOV (Single-Root I/O Virtualization) specified by PCI SIG [IOVb] [SR-IOV, 2009]. Likewise VMDq, each virtual machine has access to a dedicated queue on the hardware NIC. In addition, it has access to dedicated registers. Through a specific driver, each virtual machine accesses the so called *Virtual Functions*, which communicate with *Physical Functions* handling the Ethernet port. Using these functions, packets on a physical NIC are directly copied to the virtual machine's memory space, without the hypervisor getting involved. We formalize the virtualization mechanism used by SR-IOV as:

$$V_s : \textit{NIC driver function} \rightarrow \textit{Set of virtual functions}$$

SR-IOV is implemented on some Intel controllers, and similar technologies are provided by other vendors, e.g., Myricom [MYR], Neterion [NET]. Software support exists for Linux [IOVa] and hypervisors, such as Xen [Dong et al., 2008].

2.3.6 Virtual optical connectivity

All technologies discussed so far are designed to virtualize the network at its edges, from the host's NICs to the edge switches. Virtualizing also the connectivity over the Internet's backbone must consider the sharing of the optical links and cross-connects. However, the optical network is governed by a completely different paradigm. In general, direct connections are established between edges over optical circuits.

Recent research started to extend virtualization to the optical network. OpenFlow, which allows one to program a switch's flow tables as detailed in Section 2.4.1.2, has been implemented in Ciena optical switches. Layer 1 attributes have been added to the standard protocol, enabling the specification of particular wavelength, on which traffic should be sent out of the switch, or to select a virtual concatenation group or a starting time-slot and a signal type for the traffic [Das et al., 2010]. This allows users to configure how packets are sent out to the optical network. Together with FlowVisor [Sherwood et al., 2009], it allows them to virtualize the optical network.

Bandwidth virtualization has been proposed in optical networks as a way to decouple bandwidth from wavelength [Melle et al., 2008]. Services can be mapped to subwavelengths, with a fraction of the bandwidth of the wavelength, or to superwavelengths, that combine the bandwidth of several wavelengths. Digital Virtual Concatenation (DVC) is used to map these sub- and superwavelengths to the actual optical network.

Within the GENI project [GEN], the virtualization of optical links has been proposed using OFDM/OFDMA (orthogonal frequency multiplexing/orthogonal frequency multiplexing access) technology [Wei et al., 2009]. This enables virtualization links at subwavelength granularity, by assigning so-called subcarriers to virtual links. Consequently, we formalize the virtualization using OFDM/OFDMA as:

$$V_s : \textit{Wavelength} \rightarrow \textit{Set of optical subcarriers}$$

Using this technology, the bandwidth of a wavelength can be shared by carrying numerous subcarriers on it—e.g., 256 subcarriers on a 10 Gb/s wavelength.

2.3.7 Summary of technologies

The above described techniques to virtualize the different connectivity elements of network components are summarized in Table 2.1. The next Section describes how virtualization goes a step further into the network, virtualizing also its functionality.

Technology	Goal	Concerned component	Virtualization function	Result	Bandwidth Service
VLAN	Logical isolation	Port	$V_s : Trunk\ port \rightarrow Set\ of\ virtual\ ports$	Port partitions	Priority or minimum guaranteed rate (depends on number of queues)
VPN	Secure connection over a network	Sequence of links	$V_c : Sequence\ of\ links \rightarrow Virtual\ link$	Virtual link over a path of links	Bandwidth per access link or L1 provisioning
Overlay	Simplified connectivity	Sequence of links	$V_c : Sequence\ of\ links \rightarrow Virtual\ link$	Virtual link over a path of links	QoS aware routing and/or overlay nodes providing traffic control
OS-level virtualization	Run several virtual machines as processes on a single host	TCP/IP stack	$V_s : TCP/IP\ stack \rightarrow Set\ of\ sockets$	Sockets, several TCP/UDP ports	Software queue scheduling (e.g., htb in VServer)
Full software virtualized NICs	Access the NIC with its native driver	Kernel network stack	$V_s : Kernel\ network\ stack \rightarrow Set\ of\ virtual\ network\ stacks$	Set of virtual network stacks	Depends on software QoS
Paravirtualized-virtualized NICs	Lightweight NIC access through a virtual driver	NIC driver	$V_s : NIC\ driver \rightarrow Set\ of\ virtual\ NIC\ drivers$	Virtual NIC drivers	Depends on software QoS
VMDq	Direct access to the NIC hardware and DMA to VMs	NIC	$V_s : NIC \rightarrow Set\ of\ virtual\ NICs$	Virtual NICs with dedicated queues	
SR-IOV	Direct access to the NIC and registers	NIC and driver function	$V_s : NIC\ driver\ function \rightarrow Set\ of\ virtual\ functions$	Virtual NICs and virtual driver functions	
OFDM/ OFDMA	Share the bandwidth of an optical wavelength	Wavelength	$V_s : Wavelength \rightarrow Set\ of\ optical\ subcarriers$	Optical subcarriers (SC)	Guaranteed bandwidth per virtual link at the granularity of an SC

Table 2.1: Summary of techniques to virtualize connectivity (links).

2.4 Virtualizing Functionality

Virtualizing the connectivity is a traditional way to share a network among different users or applications. It allows networks to better exploit resources, e.g., the available bandwidth on the links, and to logically isolate the traffic of different users by grouping them into virtual networks. However, virtualizing only the connectivity does not provide each virtual network with individual functionality, such as routing and forwarding mechanisms. Hence, it is necessary to virtualize functionality so that each virtual network user or operator can set up its own protocol stack, or even specify packet queuing and scheduling different from other virtual networks. In this way, a virtual network could be configured to provide specific QoS levels.

A virtual network should not only be usable, but also configurable or ideally, programmable.

Traditionally, an operator configures an equipment, its functionality (e.g., routing, VLANs) and QoS. The users only connect to the network and transfer data. Virtualizing network functionality creates a new role,a virtual network operator (VNO), in charge of configuring the virtual equipment [Feamster et al., 2007]. The physical equipment is still maintained and configured by a physical network provider or operator. Figure 2.10 illustrates the layers in this new concept.

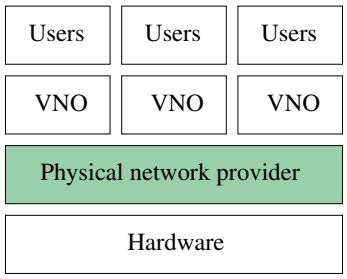


Figure 2.10: Virtualization of network functionality; a new role—the Virtual Network Operator (VNO)—can configure his virtual network.

The novelty of such an approach is that the network devices are no longer black boxes. VNOs can configure their virtual network partitions in parallel by accessing the virtual routers and switches. This can move the programmability of overlay networks from end-hosts to network devices, avoiding to deviate traffic through overlay hosts, which are end-hosts.

Network functionality can be configured at the network or routing layer, and at the forwarding layer. The following section starts by introducing the concept of programming routers, and then surveys the different technologies to virtualize the functionality of the network and to configure virtual network devices.

2.4.1 Network programmability

This section introduces the concept of programmability and its has two (actually complementary) important roles:

- It is a main *motivation for network virtualization*: with virtualization, different virtual networks can run on top of a physical infrastructure, and each virtual network

can be programmed differently. That is, each virtual network can run its own protocol, forwarding mechanism, etc.

- It is an *enabler of network virtualization*: Some components of the network can be programmed so as to run several virtual networks in parallel, in different partitions of the physical network.

In this section, we briefly survey approaches to program the network, and that can *enable virtualization* in the network.

2.4.1.1 Programmable devices

Click router is a widely used router programming platform, consisting of a modular software router running on Linux or FreeBSD platforms [Morris et al., 1999]. A user can build its own router by chaining a set of *click elements* that perform the classical tasks executed on a packet on its path throughout a router. Such elements implement for example queues, dequeuing processes, filtering modules, modules to retrieve packets from NICs, among others [CLI]. Users can also program custom elements and insert them on the data path, building their own fully programmable and configurable router. This tool enables experimentation and proof of concept, and may even be used in production networks on today's high-performance servers. It has for example been used at different levels to build virtual routers, as later described in Section 2.4.4

Considering the hardware, even though boxes are mostly closed, research effort has been made to expose part of their features for configuration, and for programmability. Orphal is an example of an open router platform with an API for programming custom modules on hardware routers [Mogul et al., 2008]. These modules, also called *switchlets*, have their own address spaces and control threads, and can operate on packets or flows or implement control plane functions. For example, switchlets can perform monitoring and implement firewalls, NAT, or even OpenFlow; a protocol discussed in the next section. Using Orphal, a user can program even packet processing on a hardware vendor platform. This becomes really interesting in a virtualized context, where users can manipulate confined environments. Similarly, JunOS SDK enables the creation of custom service modules in JunOS [Kelly et al., 2010]. Plug-ins that perform for example packet manipulation, monitoring, traffic classification, are added to these service modules. Other vendors propose APIs and SDKs for hosting third-party applications in hardware routers, e.g., Cisco's Application eXtension Platform [CSD, 2008]. Jointly with virtualization, these features become interesting as different users could program their own modules to run in parallel.

Programming the lookup of hardware in in high-speed routers has been proposed with a flexible modular lookup module called PLUG [De Carli et al., 2009]. It enables programming a customized pipeline of lookup tiles where each tile can be adapted to the type of lookup performed. In this efficient design, performance is close to that achieved with specialized lookup modules. Such programmable lookup modules could be used to virtualize the lookup mechanism in routers and switches.

2.4.1.2 Moving the control plane out of the box

One step towards enabling the programmability of the network as a logical entity is to decouple the control plane from the data plane, taking the control plane out of the routers,

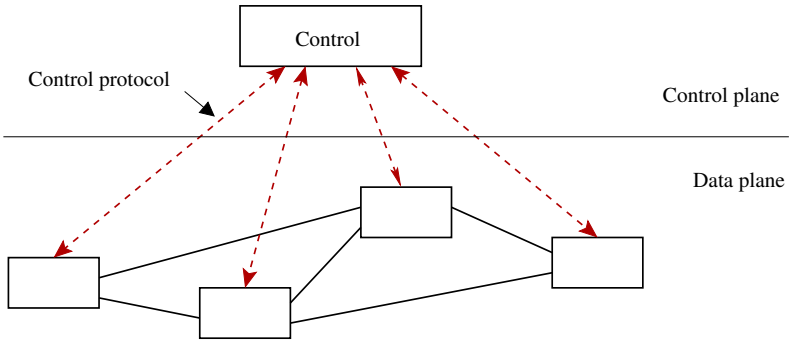


Figure 2.11: Separating control and data plane in a network.

as represented in Figure 2.11. This separation of data and control plane has been proposed in different ways.

ForCES. The IETF defined the Forwarding and Control Element Separation (ForCES) [Khosravi and Anderson, 2003] [Yang et al., 2004] to abstract the network control from the physical resources. As opposed to classical routers, the control such as routing does not necessarily have to take place in the same device of packet forwarding. Instead, *forwarding elements* (e.g., switches) can be controlled out of the box by *control elements* that do not need to be equal in number to forwarding elements. For example, one control element can be in charge of several forwarding elements in a network, as shown in Figure 2.11. This offers great flexibility in network management, and if the control plane is programmable, virtualization is possible. The programmability depends on the interface that forwarding elements expose to the control elements. The communication between forwarding and control elements is done with the ForCES protocol [Doria et al., 2010].

OpenFlow. It is a specific protocol for the communication between control and forwarding elements [McKeown et al., 2008] [OF]. It is part of a pioneer project enabling any user equipped with a general purpose PC to program the forwarding tables of commercial routers and switches, which run a slightly modified operating system, understanding and speaking the OpenFlow protocol. Hereafter, we refer to such switches and routers as *OpenFlow switches*. An OpenFlow switch operates on flows, that are defined by a subset of IP source and destination addresses, MAC source and destination addresses, TCP/UDP source and destination ports, as well as VLAN tags, and other Ethernet and IP packet header fields. OpenFlow switches are equipped with a TCAM[3] flowtable that contains entries that define how flows are routed. *OpenFlow controllers*, which can be written in any programming language and run on general purpose PCs, are responsible for configuring the switches flow tables, and handling messages from the switches, e.g., informing on their state or on the arrival of a packet. In fact, when a new flow that is not yet registered to the flow table arrives, the first packet is sent to the controller, which can then decide how to route the flow and which flow to add to the flowtable. The OpenFlow principle is

[3]TCAM (Ternary Content Addressable Memory) is a particular type of memory, that is used for programmable lookup tables.

described in detail in Chapter 5, where we use the programmability of the flowtable to set up virtual networks.

The following sections describe the different propositions for introducing programmability to the network when virtualizing it.

2.4.2 Hardware router virtualization

Virtualization has been proposed for hardware routers, mainly for consolidation purposes, i.e., running several virtual routers inside a single physical device to reduce equipment cost and space. Moreover, the routing hardware has been virtualized to run different virtual routers with their own routing protocols on the same device. Some examples of how router manufacturers propose virtualization are discussed next.

2.4.2.1 Virtual Routing and Forwarding

Virtual Routing and Forwarding (VRF) partitions a single physical router, where each partition or logical router can run a different routing instance. This is achived by running several parallel routing tables, one per VRF. We formalize the virtualization of a routing table as follows:

$$V_s : \textit{Routing table} \rightarrow \textit{Set of virtual routing table instances}$$

With this, several routers can run in parallel on the same equipment. Hence, at a given time, several virtual networks can share the devices using VRF. Some types of VPN exploit this technology to route traffic differently in each VPN, e.g., based on MPLS [Rosen and Rekhter, 1999].

2.4.2.2 Virtual Router Redundancy Protocol

The inverse approach to sharing consists in aggregating several physical routers into a virtual router. Virtual Router Redundancy Protocol (VRRP) [Hinden, 2004] manages a set of aggregated physical routers that belong to the same LAN. It ensures that, if one router fails, another takes over the control. Only one of the physical routers, the 'master', is actually running and routing the network traffic, the other are backup routers. In case the master fails, VRRP ensures that the routing task is delegated to one of the backup routers, keeping the network running. The set of redundant physical routers, managed by VRRP, is seen by the network like a single logical or virtual router. We can hence formalize the virtualization (aggregation of routers) performed by VRRP as follows:

$$V_a : \textit{Set of routers} \rightarrow \textit{Virtual router}$$

2.4.2.3 Router consolidation

Consolidation is a term frequently used in the context of server virtualization. Servers are often underutilized, and with virtualization one can consolidate several virtual servers into a single physical server, resulting in better resource usage as well as savings in equipment cost and floor space.

Trying to reap similar benefits, consolidation is proposed by manufacturers of network equipment. With improvements in hardware (10 Gb/s links, high switching power), it is appealing to consolidate routers and switches into a single equipment—a virtualized router.

Major vendors propose virtualized network routers that aggregate a whole network of routers and switches inside a single device, reducing the complexity of the cabled network, and saving cost and space. Figure 2.12 illustrates this concept, as realized by Cisco[4] in Internet access routers, also called Points of Presence (PoPs) [HVR, 2008]. In this exam-

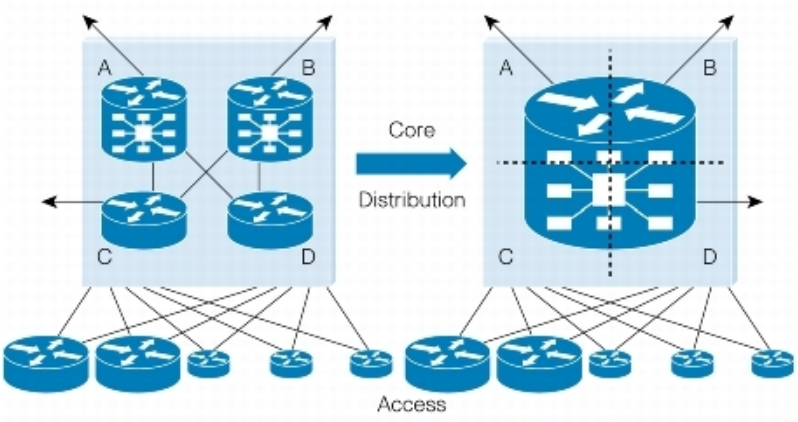


Figure 2.12: Consolidation of network resources in a PoP [HVR, 2008].

ple, a network of routers and switches (A, B, C and D on the left part of Figure 2.12) are consolidated as virtual entities into a single device (right part of Figure 2.12). Such a large network device, hosting the different virtual routers and switches, implements virtualization in different ways depending on the performance requirements. One implementation consists in dedicating a hardware control engine and a hardware forwarding engine to a virtual routers [JCS, 2009] [HVR, 2008]. The control engine can be configured with the suitable routing protocol. This is not really virtualization, but rather the segmentation of a hardware platform into several routing and switching modules. Yet, as it shares a hardware platform, we formalize this transformation with the sharing function as follows:

$$V_s : \textit{Router platform} \rightarrow \textit{Set of dedicated routing and forwarding hardware}$$

With this approach, virtual routers benefit from high performance and isolation as they have dedicated hardware resources. As a drawback, the number of virtual routers is limited to the number of hardware routing and forwarding engines present on the platform. The Juniper JCS1200 technology features for example 12 routing engines [JCS, 2009]. Moreover, it does not offer a high degree of flexibility in the granularity of resources allocated to a virtual router. A router can for example only allocate entire ports, even if it requires less capacity than what the port offers.

Another type of implementation manufacturers propose consists in virtualizing only the software part of the equipment. This means that a virtual router does not have dedicated hardware, but shares the hardware with other virtual routers. As an example, on a Juniper router, up to 16 *logical routers* can be implemented on top of a single physical routing engine [Kolon, 2004]. Similarly, Cisco can virtualize routers in the software part of their equipment [HVR, 2008]. Hence, with this approach, a single routing engine is virtualized to run several virtual routing instances, as formalized below:

[4]Note that other equipment manufacturers adopted similar implementations.

$$V_s : \textit{Routing engine} \rightarrow \textit{Set of virtual routing instances}$$

For isolating virtual routers in terms of performance, a class of service per logical router can be configured to control their rate. However, the control is performed in software and does not offer the same degree of isolation as dedicated hardware virtual router.

2.4.3 Distributed virtual switches

A distributed virtual switch abstracts a network of switches, hiding its complexity and allowing it to be managed in a simple way, as a single switch. This concept is mainly used in the context of server virtualization, as shown in Figure 2.13. The left side shows

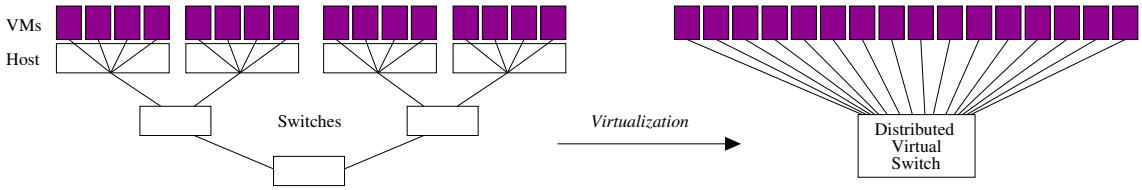


Figure 2.13: Distributed virtual switch.

a network interconnecting four hosts, each running a set of virtual machines (VMs). In this interconnection of virtual machines, there are two levels of switching. First, a switch inside each host multiplexes and forwards the traffic of the virtual machines and the NIC. Second, a network of switches interconnects the hosts. With distributed virtual switches, the complexity of this network is hidden, as represented on the right side of Figure 2.13.

With this type of virtualization, a network of switches is concatenated, i.e., aggregated but with respect to a specific topology, in order to hide complexity. Hence, we formalize the virtualization as follows:

$$V_c : \textit{Network of switches} \rightarrow \textit{Virtual switch}$$

Cloud solutions propose distributed switches within their management platforms, which facilitate the management of a cluster of virtual machines that are distributed over different physical machines.
For example, VMware vSphere 4 platform [VSP] has introduced vDS [vDS, 2009] (virtual Network Distributed Switch). vDS enables the centralized management of the switching between virtual machines hosted on different hardware servers, like a single switch. For example, private VLANs can be set up to interconnect only a subset of the virtual machines connected to the vDS. Moreover, virtual machines can be migrated transparently from one host to another, without loosing connectivity as they remain on the 'same' switch, the vDS. An interesting feature for controlling the traffic of virtual machines is traffic shaping, which can be set up on the virtual ports of vDS. Moreover, virtual machine traffic can be classified into different types, such as virtual machine migration traffic and management traffic, each of which is provisioned with different rates. A specific implementation of vDS for VMware, that supports up to 64 hosts, is the Cisco Nexus 1000V [NEX, 2010].

Another such technology is Brocade's Virtual Cluster Switching (VCS) [VCS, 2010]. Likewise vDS, it manages a topology of networked switches, including hardware switches and software switches (those interconnecting virtual machine inside a single host). This topology is viewed and managed as a single logical switch, just like the virtual distributed

switch represented in Figure 2.13. The particularity of VCS is a specific protocol that allows routing the traffic over the network of switches in a way that optimizes resource utilization. This specific protocol achieves better switching performance than if the paths over the different switches of the topology were computed using the classical spanning tree protocol (STP) [STP, 2004].

A very evolutionary distributed software switch is Open vSwitch [Pfaff et al., 2009] [OVS]. It is compatible with most standard virtual machine platforms, e.g., Xen, KVM, VirtualBox [VBX], and can interconnect the virtual machines of different hosts, like the previously described technologies. It exposes interfaces to NetFlow [Claise, 2004], sFlow [Phaal et al., 2001] and other types of management standards. It also features OpenFlow, a protocol to program switch's forwarding tables (see Section 2.4.1.2). Virtual machines can be transparently migrated across different hosts, while being centrally managed by an Open vSwitch spanning them.

In summary, the concept of distributed virtual switches is very interesting to hide the complexity of the interconnection of virtual machines in a LAN. Especially with the trends in Clouds, where clusters of virtual nodes need to be interconnected on demand whenever virtual machines are instantiated, the approach is useful. In [Pettit et al., 2010], the authors predict that virtual switching will expand, and more management interfaces like Netflow and sFlow will be added to switches like Open vSwitch, making them indistinguishable from hardware switches. This will be possible with the improvements in hardware, enabling more functionality to be offloaded.

2.4.4 Software router virtualization

Different types of virtual routers have been implemented in software, mainly for research, experimentation and proof of concept. While performance generally represents a weakness, the ability to configure and program software routers offers virtually unlimited flexibility. Such virtual software routers enable to implement and approve new designs, before deploying them on hardware. They are among the first, to virtualize the network functionality. This section overviews of the research background on virtual routers.

Virtual routers have been envisioned several years ago, for their advantages, like rapid deployment and reconfiguration, which could be useful in research. At that time, servers had not reached high performance, and virtualization technologies were not yet very popular. Virtual routers were first proposed for emulating real IP routers and for building test networks for experimenting on one or several general purpose machines, rather than investing in dedicated network equipment [Baumgartner et al., 2002] [Puljiz and Mikuc, 2006]. Then, virtual routers were proposed as a tool for research and education, where new functionalities could be tested [Baumgartner et al., 2003]. They were run inside Linux processes emulating IP routers, and interconnected by a network of UDP tunnels. Students and researchers could configure queue management and other router functionality for investigating on subjects such as QoS.
Later, a virtual router implementation, called QuaSAR (QoS aware router), was proposed [McIlroy and Sventek, 2006]. Its main purpose was to separate traffic into virtual forwarding-planes that could achieve differentiated treatments of traffic, directing it over different virtual routers. In this router, the virtualization function can be formalized as:

$$V_s : \textit{Software router} \rightarrow \textit{Set of virtual routers with separate data planes}$$

Virtual routers were divided into one Best-Effort router and several MPLS routers. Each of the MPLS routers was configured to fulfill a specific RSVP-TE [Awduche et al., 2001] requirement. This concept was realized with Xen and Click router [Morris et al., 1999], and virtual software routers hosting control and data planes were proposed.

Xen and Click have further been used to build other variants of virtual routers. In [N. Egi et al., 2007], the authors used Xen and evaluated extensively the performance overhead introduced by the virtualization layer, the hypervisor. These initial results showed the high rate of packet loss in virtual machines, due to CPU saturation, and concluded on the difficulty of virtualizing the forwarding plane. Later, Egi at al. proposed to statically bind virtual routers to CPUs to obtain better network performance in virtual routers hosted on commodity servers [Egi et al., 2008a] [Egi et al., 2008b]. Finally, in order to build high-performance software virtual routers, they proposed to virtualize only the control plane, meaning that each software virtual router has a separate routing mechanism. However, the forwarding of all packets is not on the responsibility of the virtual routers. Instead, packets are directly moved from input NICs to output NICs on the physical host, according the the routing decisions from the virtual routers. Hence, in this approach, we formalize the virtualization, which applies only to the control plane, as follows:

$$V_s : \textit{Software router control plane} \rightarrow \textit{Set of virtual software router control planes}$$

For leveraging performance, the authors proposed to bind explicitly forwarding paths in the physical host to CPU cores and organize them in a particular hierarchy to keep a single packet on the same CPU and the closest cache as long as possible [Egi et al., 2009].

However, for keeping the possibility to configure the data plane, on a per virtual network basis, packets should be forwarded inside the virtual routers. This has been proposed in [Liao et al., 2010], where the expensive interruption mechanisms that trigger copying packets from NICs to virtual routers and reciprocally, have been replaced by polling, i.e., active checking if packets are waiting to be treated. In this case, each virtual router has a dedicated memory space that it shares with the host system kernel. Packets are copied to this memory by the NIC and the virtual routers poll it constantly to see if there are packets to process. This is naturally very CPU consuming, but for virtual machines whose activity is dedicated to routing, it can represent an interesting approach. Being implemented with OpenVZ and Click, the obtained throughput is close to classical Linux forwarding throughput.

In general, performance is the main issue preventing software routers from being deployed in production networks, even if they are not virtualized. To cope with this issue, one approach is to aggregate the resources of several machines into a cluster to build a virtual software router. This can be formalized as follows, if a cluster of interconnected servers where each one hosts a software router is aggregated into a single virtual software router:

$$V_a : \textit{Set software routers} \rightarrow \textit{Virtual software router}$$

In [Argyraki et al., 2008], the authors propose such a clustered software-router architecture built of servers, organized in a particular topology. This design is extended to *RouteBricks*, where workload is parallelized among servers and cores [Dobrescu et al., 2009]. With RouteBricks, a 35 Gb/s aggregate router could be built from several servers.

Parallelization has also been used for improving the performance of virtual routers. In [Liao et al., 2009], the authors propose to use multiple parallel data planes (PdP) for virtual routers. In the proposed implementation, a user control plane runs on a virtual machine on a given host. The data plane controlled by this control plane is implemented on several parallel forwarding engines, whose number depends on the required performance. Each forwarding engine is hosted inside a virtual machine. These different virtual machines can be allocated on different physical machines to aggregate their performance and provide the required forwarding rate. Thus, the virtualization function in this approach consists in first virtualizing a host into several virtual machines, each of which hosts a packet forwarding engine. Then, several of these virtual forwarding engines are re-aggregated together, to obtain a more powerful virtual forwarding engine. Hence, sharing and aggregation are combined in the following way:

$$V_s \circ V_a : \textit{Software forwarding engine} \rightarrow \textit{Parallel virtual software forwarding engine}$$

Evaluations of this model show that multiplying the forwarding engines also multiplies the packet forwarding rates, which is a main issue in traditional virtual software routers.

Packet rates are not the only scalability concern in virtual routers. The size of the lookup tables for routing and forwarding may also be a concern. In a virtualized router, these lookup tables must be shared among numerous virtual routers. The sum of all the routes to be stored in all virtual routers may largely exceed the number of routes needed in a single physical router. Hence, it has been proposed to optimize the way of virtualizing the lookup mechanism and sharing the lookup table by creating a common prefix set for different virtual routers [Fu and Rexford, 2008]. This means that some entries of the routing table can be shared by different virtual routers. With a similar scope, *trie-braiding* was proposed for virtualized routers [Song et al., 2010]. A trie is a tree structure organizing routing prefixes for performing lookups. Trie-braiding combines these lookup structures from several virtual routers into a single trie, which has a smaller memory footprint than the set of virtual router tries. Depending on the braiding strategy, the memory footprint can be reduced significantly.

A particular application of virtual routers is to implement fault tolerance. The idea is to run multiple virtual router instances in parallel to form one bug tolerant router (BTR) [Caesar and Rexford, 2008]. This concept is similar to VRRP (see Section 2.4.2.2), but uses virtual routers instead of dedicated hardware boxes. Hence, such a BTR is a virtual router formed of a set of routers, which are already virtual themselves (as opposed to VRRP, where they are physical). In other words, first, a server hosting a software router is virtualized into several virtual routers. Then, several of these virtual routers are aggregated to form a BTR. This process can be formalized as follows:

$$V_s \circ V_a : \textit{Software router} \rightarrow \textit{Redundant virtual router}$$

In a BTR node, if one virtual router has bugs in its software, another virtual router can take over the control of the node. This mechanism is handled by a *router hypervisor* that

detects faults and chooses the right routing instance and virtual router to be booted and used [Keller et al., 2009].

As another type of application, virtual routers have been proposed in the context of active networks as Active Virtual Routers (AVR) [Louati et al., 2009]. AVRs can be programmed and provisioned automatically to create virtual networks on demand. These networks benefit from the reconfigurability and the mobility of virtual routers.

In summary, virtual software routers offer a lot of flexibility to virtual networks, as they allow to configure and even program the functionalities of the network, in particular the routing. However, as they also implement packet forwarding in software, they may incur important performance drawbacks, depending on the type of implementation. For improving performance, an interesting approach is to implement virtual routers on programmable hardware, as described below.

2.4.5 Virtual routers on FPGA

Virtual routers can be programmed in hardware using special-purpose devices with configurable circuits, such as FPGAs (Field Programmable Gate Arrays). Programming an FPGA is less straightforward than a software router, but it appeals for performance and isolation in terms of both performance and configurability of layer 2 functionality, i.e., packet queuing and forwarding. Several virtual router and switch design implementations on FPGA are described below.

The programmability of FPGA enables to configure packet forwarding, as proposed to the router implemented in [Lu et al., 2009]. It can be configured to classify packets so that they are forwarded according to different schemes or protocols. This classification and differentiated treatment allow on to virtualize the network forwarding mechanism, formalized as follows:

$$V_s : \textit{Forwarding scheme} \rightarrow \textit{Set of forwarding schemes}$$

Such a router is useful in the context of data centers, where special protocols are used for routing, and where high performance is required.

Another type of router more specifically designed for virtualization is presented in [Anwer and Feamster, 2009]. It has a virtualized data plane and is built on a NetFPGA board, a special type of FPGA that has several network ports, each port having several queues [Naous et al., 2008] [NFP]. In this implementation, virtual routers' ports are allocated on such queues. The queues, which represent virtual ports, are associated to dedicated virtual forwarding environments, which forward the packets on a per-virtual-router basis. Consequently, in this design the data plane is virtualized, providing each virtual router with a virtual forwarding environment. This can be formalized as follows:

$$V_s : \textit{Forwarding environment} \rightarrow \textit{Set of virtual forwarding environments}$$

The routing of virtual routers is controlled by users, and routing instances run in OpenVZ [OVZ] containers on the host system. This virtualized router can support eight virtual forwarding engines in hardware. MAC addresses are mapped to 'virtual environment IDs', which determine for each packet which virtual forwarding environment has to be used to forward it. Unlike virtual software routers, these virtual routers can achieve 1

Gb/s throughput on each of their four interfaces without incurring CPU overhead. However, performance isolation is not yet supported in this design. Consequently, a virtual router can be impacted by another virtual router consuming an excessive amount of bandwidth. This has been improved in a later design, called *SwitchBlade* [Anwer et al., 2010], also based on NetFPGA. It enables the rapid deployment of different virtual forwarding environments, which can run in parallel and operate on custom protocols. On SwitchBlade, different programmable modules are pipelined. An incoming packet is first directed to its specific virtual forwarding environment, then to a traffic shaper module, before it is switched and sent to the output port. This also ensures performance isolation per virtual forwarding environment.

An even more flexible design for a virtual router also implemented on NetFPGA and OpenVZ has been proposed [Unnikrishnan et al., 2010]. It distinguishes two types of routers: those requiring high throughput are mapped to hardware, while routers requiring lower throughput are implemented in slow software data planes inside virtual machines. This enables better scalability as the number of routers is not limited by the hardware. If virtual routers change from low to high throughput or vice versa, they can be migrated between hardware and software. Moreover, this design stands out by the fact that hardware virtual routers can be reconfigured dynamically. For a reconfiguration, a virtual router is temporarily migrated to software, where the forwarding is implemented with Click [CLI]. Then, the FPGA is reprogrammed, and the router is migrated back. As a drawback, the scalability of such a system is limited because it is based on a Xilinx Virtex II FPGA board that supports only four virtual hardware routers able to deliver 1 Gb/s throughput each. The other virtual routers must be implemented in software, where throughput drops far below 100 Mb/s especially with small packets, and latency increases.

An improvement of this solution has been implemented later, on a Virtex 5 FPGA [Yin et al., 2010]. This new board supports up to 20 virtual routers. In addition, subregions of the FPGA can be reconfigured dynamically while other subregions stay operational. The main advantage of this kind of partial reconfigurability is that only the reconfigured virtual router will be affected by a short performance drop, while the others can continue running. On the other hand, with the Virtex II board, all virtual routers had to be migrated to software, hence being affected by a performance drop of over 90% during the reconfiguration time [Yin et al., 2010].

FPGA enables powerful virtual routers due to the virtualization of the data plane, the configurability, and programmability of the hardware. It is a very powerful tool to implement prototypes of new router or switch designs, with near hardware performance.

2.4.6 Virtual network-wide control plane

Moving the control plane out of routers, as discussed in Section 2.4.1.2, adds a completely new dimension to network configurability. It allows one i) to manage a network as a whole with a single network-wide control plane, and ii) to program the routing, which is not possible in traditional routers as they are black boxes exposing only some configuration interfaces.

Virtualizing such programmable networks allows several different entities to control their own partitions of the physical network, as represented in Figure 2.14. In an OpenFlow

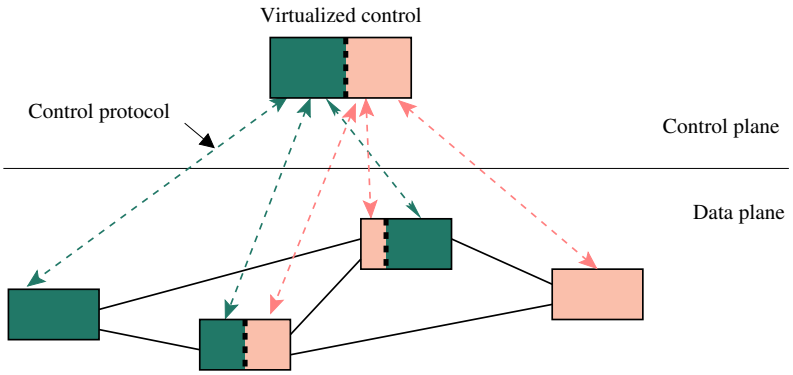


Figure 2.14: Virtual network slices with separate control and data plane.

network (as discussed in Section 2.4.1.2), where control and data planes are separated and communicate using the OpenFlow protocol, this type of virtualization is possible using FlowVisor [Sherwood et al., 2009] [Sherwood et al., 2010]. FlowVisor is interposed between OpenFlow switches and OpenFlow controllers. It 'slices' the network of OpenFlow switches, and each slices transports another subset of flows. FlowVisor operates like a proxy and sends OpenFlow messages to the right switch and back to the right controller based on the slice configuration. Hence, FlowVisor virtualizes an OpenFlow switch as follows:

$$V_s : \textit{OpenFlow switch} \rightarrow \textit{Set of virtual OpenFlow switch instances}$$

FlowVisor virtualizes and shares the layer 2 control between different users. Each user can run its own controller to manage part of the flowtables of the switches where its network slice is allocated. In addition, a less dynamic and flexible way of virtualizing an OpenFlow switch consists in running several OpenFlow instances on the same switch inside distinct VLANs.

When virtualizing an OpenFlow network, a challenge is to provide performance isolation between network slices. This challenge has been addressed by switches in different ways.

The OpenFlow protocol enables one to control the packet forwarding inside OpenFlow switches, or more concretely to which output port an incoming packet is sent. Starting from version 1.0 [OF1, 2009], the OpenFlow protocol allows specifying at which queue of a port a packet has to be enqueued. Depending on the equipment, these queues can then be configured, e.g., with rate or a priority, for isolating traffic. Using this feature for isolating virtual networks, different slices can have their own queues.

As an alternative, in the OpenFlow switches from HP, *rate limiters* can be set up in hardware. They can be managed and assigned flows, using a special extension of the OpenFlow protocol. A QoS framework was proposed to automatically install such rate limiters for flows, depending on their performance requirements and their priority compared to other flows [Kim et al., 2010]. Applying QoS to bundles of flows that constitute a network slice enables performance isolation between virtual networks. However, the degree of isolation depends on the scheduling algorithms implemented by the rate limiters.

2.4.7 Summary of technologies

The various propositions for virtualizing the network functionality through the virtualization of control and data planes in routers and switches are summarized in Table 2.2.

Technology	Goal	Concerned component	Virtualization function	Result	Bandwidth Service
VRF	Run different virtual routers on the same device	Routing table	V_s : *Routing table* $\rightarrow$ *Set of virtual routing table instances*	Several virtual routers with different routing table instances	
VRRP	Ensure network fault tolerance with redundancy	Router	V_a : *Set of routers* $\rightarrow$ *Virtual router*	Virtual router managing a set of physical routers	
Router consolidation (hardware)	Save equipment cost and rack space	Parallelized router platform	V_s : *Router platform* $\rightarrow$ *Set of dedicated routing and forwarding hardware*	Set of dedicated routing and forwarding hardware	Ports are dedicated
Router consolidation (software)	Save equipment cost and rack space	Routing engine	V_s : *Routing engine* $\rightarrow$ *Set of virtual routing instances*	Set of virtual routers sharing a data plane	Software traffic control
Distributed virtual switch	Simplify a distributed virtual machine network	Network of switches	V_c : *Network of switches* $\rightarrow$ *Virtual switch*	Virtual switch spanning over a topology of switches	Centralized traffic control
QuaSAR	Isolated virtual router with different QoS	Software router	V_s : *Software router* $\rightarrow$ *Set of virtual routers*	Set of virtual softwares router with separate data planes	QoS through MPLS RSVP-TE per virtual router
High-perf. software virtual router	Routing control on a per virtual network basis	Software router control plane	V_s : *Software router control plane* $\rightarrow$ *Set of virt. soft. router ctrl.-planes*	Set of virtual software routers sharing a common data plane	
RouteBricks	High performance in software routers	Topology of software routers	V_a : *Set of routing servers* $\rightarrow$ *Virtual router*	Virtual router spanning over a network of servers	
PdP	Forwarding performance of virtual software routers	Virt. soft. forwarding engine	$V_s \circ V_a$: *Software forwarding engine* $\rightarrow$ *Parallel virtual software forwarding engine*	Multiple virtual software forwarding engines in parallel	
BTR	Redundancy for bug tolerance	Software router	$V_s \circ V_a$: *Software router* $\rightarrow$ *Set of virtual routers*	Redundant virtual router	
SwitchBlade	High-performance virtual forwarding paths	Forwarding environment	V_s : *Forwarding environment* $\rightarrow$ *Set of virtual forwarding environments*	Set of different virtual forwarding environments	Shaping at the ingress of the switch
FlowVisor	Concurrent programming of forwarding in switches	OpenFlow switch	V_s : *OpenFlow switch* $\rightarrow$ *Set of virtual OpenFlow switch instances*	Individually configurable virtual OpenFlow switches	Equipment-specific rate-limiting or queuing

Table 2.2: Summary of techniques to virtualize network functionality (routing and forwarding).

To summarize the two first sections of this chapter, Figure 2.15 shows where in the network the different virtualization technologies are employed. Starting from the edge,

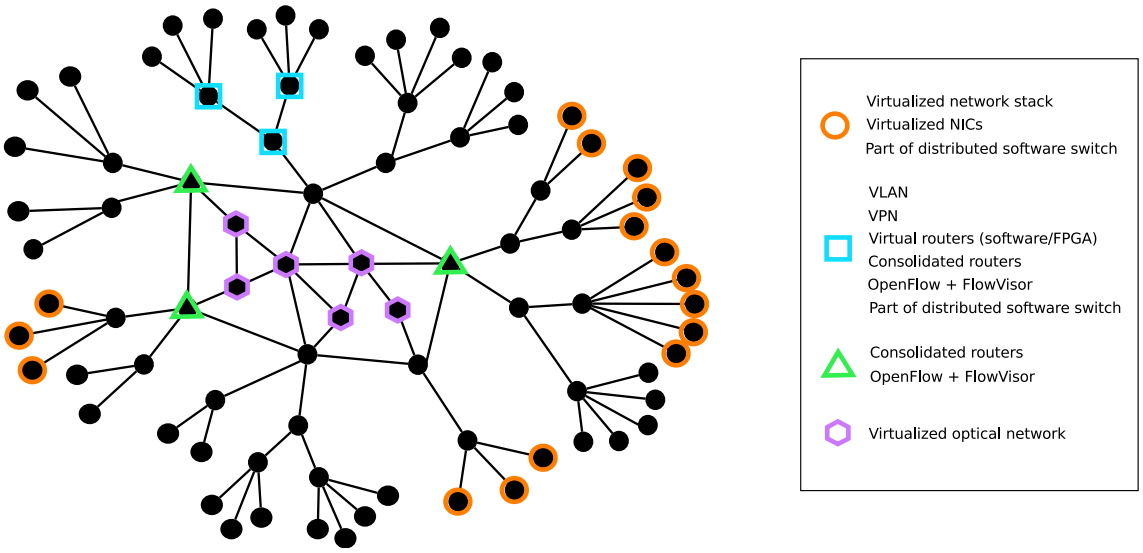


Figure 2.15: Network virtualization from the edge to the core.

server virtualization, i.e., virtualized network stacks, as well as VLANs and VPNs have been around for a long time. Modern technologies such as router consolidation and OpenFlow in optical switches, combined with FlowVisor can virtualize the PoP level, i.e., at the access routers to the backbone. Finally, ongoing research on optical equipment is pushing virtualization toward the network core.

The next section describes how the above discussed technologies are applied in today's networks.

2.5 Application examples

Interest in network virtualization has increased over the last years, as result of the popularity of server and data center virtualization, and the arrival of the Clouds [Rosenberg and Mateos, 2010]. Server virtualization has been taken as an example on how to bring flexibility and isolation to resources, and more particularly the routers—the functionality of the network—for enabling network innovation.

This section describes to which extent the previously described technologies have been applied for creating programmable virtual networks and virtual network infrastructures.

2.5.1 Mobility in networks

One of the benefits of virtualization is *mobility*. By abstracting virtual resources from the hardware, they can be migrated from one host to another. Migrating virtual network nodes, links and paths brings important features to virtual networks. It enables for example fault tolerance, where in case a node or link fails, the virtual links and nodes can be relocated to allow quick recovery from a virtual network outage. In addition, router migration can improve the physical resource usage, e.g., link capacity [10]. This can be

required in the following situation. As virtual networks can be dynamically created, at some moment the combination of all the virtual networks on the physical network may not be optimal. To cope with this issue, in [Yu et al., 2008], the authors propose to migrate virtual paths, to improve virtual network embedding and re-balance the mapping of virtual networks on the substrate. Figure 2.16 illustrates this concept, where virtual networks 1

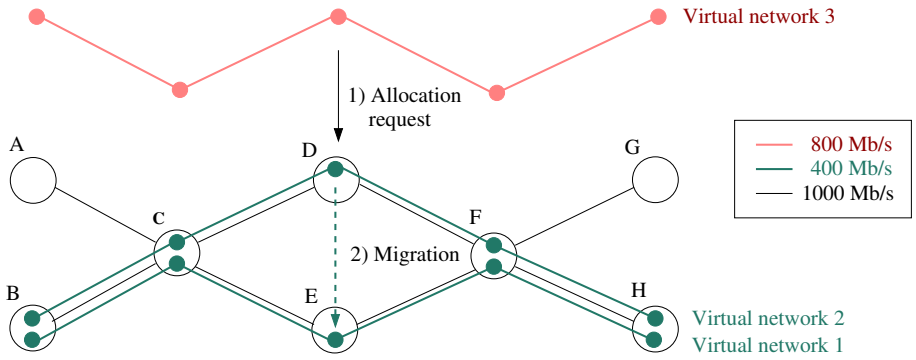


Figure 2.16: Migration of virtual routers for improving the virtual network embedding.

and 2 are allocated and running on a physical substrate network with 1000 Mb/s links. Virtual network 3 cannot be allocated as it requests a bandwidth of 800 Mb/s on each link. In this case, a migration of the virtual router from the physical substrate node D to node F can solve the problem. After the migration, virtual network 3 can be allocated on the physical substrate nodes A, C, D, F and G.
In this case, a re-organization of virtual resources can allow more virtual networks to be allocated and to use the physical resources more efficiently. Besides, the reorganization could be made to save energy, by minimizing the dispersion of virtual routers on the physical infrastructure.
In addition, Yu et al. propose to split virtual paths that require an amount of bandwidth that is not available on any physical path [Yu et al., 2008]. A fraction of the virtual path can for example be migrated to another physical path to optimize the allocation. A migration algorithm is proposed to take such decisions on virtual network reallocations. As an example, VROOM (Virtual ROuters On the Move) allows live migration of virtual routers without incurring an important performance impact on the running network [Wang et al., 2008]. The migration process is as follows. First, routing messages are tunneled to the new host where the virtual router control plane is then migrated. The data plane keeps running on the initial host, while being copied to the new host. Second, after all traffic has been redirected to the new host, the data plane is disabled and removed from the initial host. Hence using VROOM, a virtual link allocated over a path containing migrating virtual routers, appears to the user as a continuous service.

2.5.2 Research and experimentation

Multiple research initiatives are presently attempting to re-architecture Internet at different levels. In Europe, the Future Internet initiative [FI] aims at addressing challenges such as mobility, reconfigurability, extensibility, flexibility, business value and energy efficiency, to make the Internet sustainable. These challenges emerged especially from the fact that the Internet is no longer only a communication facility, but an infrastructure to host and

share content and applications [Tutschku et al., 2008].

The Clean-Slate program in the USA also aims at revolutionizing the Internet. The goal is to design a network from scratch without basing it on the current Internet architecture, as opposed to the incremental evolution, adopted over the past 30 years [Feldmann, 2007]. To finally enable major architectural changes, virtualization is expected to be one of the main paradigms of the future Internet [Tutschku et al., 2011].
Numerous projects aiming at virtualizing networks started over the last few years, especially from the future Internet and the Clean-Slate programs. In particular, various experimental platforms emerging from different research projects support resource virtualization to deploy confined experiment environments qualified as virtual infrastructures. A *virtual infrastructure* consists in a set of virtual resources allocated on the physical infrastructure and interconnected over the network. These can be deployed on demand, and used for computing or for storage for example. The particularity of *virtual network infrastructures* is that they also include virtual routers and links, which are configurable.

One of the first implementations of a virtual network infrastructure is VINI [Bavier et al., 2006] [VIN]. Researchers can run experiments in virtual network partitions, interconnected by virtual routers, that run on UML (User Mode Linux) [UML] instances. XORP routing software [Handley et al., 2005] is deployed on each virtual router, so that it can run its own routing protocol, independently of other virtual routers. VINI has been evaluated on top of the PlanetLab platform [Bavier et al., 2004]. More recently, specialized nodes equipped with network processors have been deployed on PlanetLab for enabling the deployment of high-performance network nodes [Turner et al., 2007]. Network traffic of different slices can be assigned to their own queues in order to control the share of network bandwidth each slice obtains.

Later, Trellis was designed as a platform to host virtual networks such as those created with VINI [Bhatia et al., 2008]. Physical hosts can run several virtual routers. Physical links can run several virtual links, which can span several physical link segments as overlay links. The virtual routers are implemented using OS-level virtualization, which offers better throughput compared to full virtualization, but allows less configurability as the data-path is not virtualized, only the routing mechanism.

Another virtual network facility is Mobitopolo [Potter and Nakao, 2009], a service that adds mobility to virtual network infrastructures. It is also implemented in UML with a focus on portability to enable snapshot-based deployment and live migration on Linux based systems. Virtual nodes are interconnected using UDP tunnels, and can move around the locations where they are needed.

Within the Emulab testbed [White et al., 2002], virtualization of hosts, routers and network has been implemented for allocating experiments more flexibly [Hibler et al., 2008]. Virtualization is used at a minimum degree, setting up virtual nodes inside FreeBSD Jails, as discussed in Section 2.3.4.1. As this does not virtualize the network interfaces, special virtual Ethernet interface devices have been developed to interconnect virtual networks, which can perform traffic shaping.

Another virtual network infrastructure that is supposed to enhance PlanetLab is CoreLab [Nakao et al., 2008]. As opposed to VINI, it is based on a hosted virtual machine monitor, KVM [Kivity et al., 2007]. The reason for this choice is the increased isolation and flexibility, which enable the deployment of any software in the virtual infrastructure nodes as though they were physical nodes. This naturally impacts the performance due

to heavier procedures to access the hardware. However, as the technology is promising and expected to improve, this solution may be more advantageous for enabling network innovations.

Virtualizing production network infrastructures creates a new business model. As described in [Feamster et al., 2007], infrastructure providers are different from service providers. The former run the physical network equipments, while the latter lease virtual slices from the infrastructure. This idea is developed in [Schaffrath et al., 2009] in the context of virtualizing the infrastructure to enable network innovation. The authors propose to layer the network architecture horizontally. Each layer is managed by a different entity, namely physical infrastructure providers and a virtual network providers, as well as a virtual network operators and a service providers. Each stakeholder takes a different business role. While the physical provider operates the equipment, the virtual provider creates the virtual network infrastructure and leases it to a virtual network operator. The latter configures it for specific service providers. In such a layered model, a virtual network operator can individually configure a network, deploy anything and innovate. A prototype of such an infrastructure service has been implemented using Xen and Click on commodity servers.

Shadownet [Chen et al., 2009] is yet another platform that enables to dynamically create logical network topologies. As opposed to the previously described virtual infrastructures, Shadownet nodes can be allocated on hardware network devices that feature partitioning (e.g., Juniper logical routers [Kolon, 2004]). Virtual links are established using VLAN and VPN technologies. The primary goal of Shadownet is to enable the deployment of new tools directly on a production network made of slices of real resources.

Projects such as 4WARD [4WA] and SAIL [SAI] investigate network virtualization for provisioning virtual networks, e.g., in Clouds and distributed data-centers.
Within the GENI project [GEN], network virtualization is implemented at several levels. As an example, FlowVisor [Sherwood et al., 2010] is used to virtualize the network into slices and allow individual OpenFlow experiments. Moreover, some optical nodes within the GENI experimentation facility allow virtualization with OFDM/OFDMA as described in Section 2.3.6.

One of the first all-optical virtualized testbeds is currently being developed within GEYSERS project, which aims at virtualizing the Internet core [4] [GEY]. The joint provisioning of end hosts and end-to-end links enables to dimension the network performance exactly to the needs of applications and provides them the network as a service.

Following the described results of research, some parts of the network virtualization technologies have moved to production platforms, in particular to the Clouds.

2.5.3 Virtualization in production networks and Clouds

Clouds are distributed infrastructures that provide virtual computing and storage nodes on demand, as a service [Rosenberg and Mateos, 2010]. While research starts virtualizing the optical network, production platforms virtualize the network, especially at the edges, for interconnecting virtual machines. Virtualized data centers are for example frequently interconnected by virtual networks in today's production platforms. Cloud providers, such as Amazon [AMA], use common server virtualization technologies with a management framework to deploy and configure virtual machines over a distributed infrastructure. Management operations in Clouds also include migration, snapshot creation, backups,

etc., as discussed above.

Examples of middleware for setting up and managing Clouds include VMware vSphere [VSP], Eucalyptus [EUC] and OpenNebula [NEB]. In vSphere ESXi servers, virtual machines are interconnected by vSwitches inside a host. A distributed switch, vDS (discussed in Section 2.4.3), spans over several physical servers to interconnect virtual machines of different vSwitches. Virtual machine's network interfaces can be grouped into portgroups that can be configured with QoS attributes such as average- and peak-bandwidth, and burst size. Hence basic traffic control can be set up between and inside virtual infrastructures.
Eucalyptus is a private Cloud or data center solution, that integrates different hypervisors, like Xen, KVM, VMware vSphere ESX/ESXi [EUC]. It can also be interfaced with Amazon's EC2 public Cloud service [AMA] into a hybrid Cloud. Traffic isolation of different virtual machines can be performed through VLAN security groups. While the goal is not to isolate performance, classical VLAN priority mechanisms can provide basic traffic control features. Also, the traffic shaping mechanisms available on the hosts, e.g., Linux Traffic Control on Xen and KVM, or shaping on ESXi servers can be used to control virtual machine traffic. The same features are exposed by OpenNebula, which is also a private Cloud solution [NEB]. In addition, it can be deployed as a public Cloud using the OGF Open Cloud Computing Interface (OCCI) [OCC].

The example of Clouds emphasizes the need for network virtualization, since Clouds are already running production platforms. They need a virtualized network, so that virtual machines in virtual infrastructures can enjoy isolated, programmable networks, with customizable and guaranteed performance. In fact, the network should also become a service within the Clouds.

2.6 Positioning of the thesis

Among the different network virtualization technologies, described in this chapter, we identify some open issues, that currently prevent virtualization from being implemented in production networks. These are in particular related to a lack of performance guarantees inside virtual networks that share a physical infrastructure. We summarize these issues as follows:

— Most of existing technologies virtualize only the control plane of network devices, i.e., the routing. Yet, for providing performance guarantees to virtual networks, the data plane has to be fully virtualized since it is to the way to control the amount of resources attributed to each virtual network.

— The technologies that virtualize the data plane share only ports among virtual network devices. However, port capacity is not the only criterion for performance guarantees and QoS in a virtual network. The buffer sizes are also of great importance.

— The configurability of virtual networks is limited to the routing control plane. Nevertheless, for configuring QoS per virtual network, the forwarding control and hence the packet scheduling mechanisms must also be configurable.

— Most technologies share the resources without constraints, i.e., the amount each resource conferred to a virtual network can not be configured. This does not provide performance guarantees to a virtual network.

These issues motivated us to *i) virtualize all the resources of the network data plane*, *ii) enable the configuration of forwarding and packet scheduling in virtual networks*, and *iii) parametrize the virtualization function with the amount of each resource the virtual network should obtain*, when sharing the network. Hence, we propose to virtualize the different resources of the data plane with a new parameter specifying the capacities of all the virtual resources, also worth nothing in a formal way as:

V_s: *Resource, Set of capacities* $\rightarrow$ *Set of virtual resources with specific capacities*

In particular, we propose to virtualize the switching fabric of network devices, to share the ports and buffers between virtual switches, and to enable each virtual switch to have its own scheduling mechanism. As an example, the above function applies respectively in the following ways to share ports and buffers:

V_s: *Port, Set of capacities* $\rightarrow$ *Set of virtual ports with specific capacities*

V_s: *Buffer, Set of buffer sizes* $\rightarrow$ *Set of virtual buffers with specific sizes*

Throughout this manuscript, we propose different technologies for virtualizing the resources of the data plane, controlling the parametrized sharing of resources among virtual networks, and enabling the configuration of scheduling mechanisms at the data plane level.

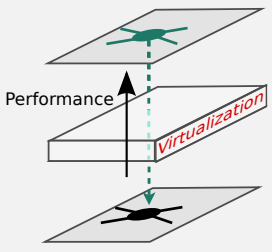
2.7 Conclusions

Traditionally, networks have been virtualized to offer connectivity independently to different groups of users, with the help of technologies like VLAN and VPN. However, to provide virtual networks with the same configurable functionality as physical networks, the network devices and their functionality also need to be virtualized. This has engaged research mainly on how to build virtual routers and programmable switches. It is crucial, to control the sharing of the physical resources by virtual routers, in order to provide each virtual network with performance guarantees. In addition, each virtual router should enable the configuration of fine-grained QoS, to be adaptable to application requirements. For providing virtual routers with this feature and make them behave like physical ones with full functionality and performance guarantees, virtualization needs to take place at layer 2, exposing routing and switching functionality to a virtual device. The first step towards building such virtualized networks is analyzing and comparing mechanisms that can implement virtualization at layer 2—the data plane. This is the subject of the next chapter, where a virtual router with a virtualized data-plane is prototyped and evaluated.

3 Analysis and evaluation of the impact of virtualization mechanisms on communication performance

The work presented in this chapter has been published and awarded as a top paper at the International Conference on Networking and Services (ICNS) in 2009 [7], at the Linux Symposium 2009 [9] and at the International Journal On Advances in Intelligent Systems 2009 [2]. In addition, a research report has been published [11].

Abstract. Virtualizing the network brings new challenges, especially addressing the potential performance overhead induced by the virtualization layer. For quantifying these, this chapter presents a *performance* study, whose main contributions are:



- An extensive analysis and comparison of technologies that enable the virtualization of the network data plane;
- A prototype implementation of a virtual software router on commodity server hardware; and
- The evaluation of the virtual end-host and router performance in terms of throughput, packet loss and processing power under various conditions.

3.1 Introduction

A wide variety of virtualization technologies, e.g., VMware [VMW], Xen [Barham et al., 2003], KVM [Kivity et al., 2007] are commonly used nowadays for virtualizing servers. Introducing these to the network offers great flexibility, since virtualizing links *and* routers, i.e., connectivity *and* functionality, allows running and configuring different virtual networks independently on the same physical infrastructure. Virtual routers can execute customized routing protocols that affect only *their* virtual network environment. Hence, a virtual network is an ideal environment for experimentation 'in the wild' because it can be deployed in parallel with a production network without damaging the latter.

As stated in the previous chapters, virtualization was initially made available to end-hosts to virtualize resources such as storage and computing. In this chapter, virtualization is pushed one step farther into the network. A virtual router prototype is built in software, and evaluated in the edge network. The precondition while designing the edge virtual router—intended for experimentation—is to have full functional isolation from other virtual routers. For realizing this, it has to have a virtual control-plane *and* a virtual data plane, so that packets from different virtual routers can take different customizable forwarding paths. The goal is to obtain a fully programmable virtual router, able to control routing and forwarding of the traffic in virtual infrastructures (detailed in Chapter 5).

However, virtualizing the data plane adds a layer of processing between the hardware and the system. This layer, responsible for virtualizing packet I/O mechanisms, is costly as described in the previous chapter. In addition, the layer controls the resource sharing and isolation between different virtual routers, so that the activity of one virtual router does not affect any other virtual router. For having better insight into these issues, in the next section we analyze the current technical solutions that allow virtualizing the network data plane. Based on this analysis, a virtual router prototype with a virtualized data plane is implemented. Its properties and potential in terms of network performance like throughput, packet loss, latency and the induced processing cost are evaluated in Section 3.3. Finally, Section 3.4 compares these results with related studies.

The results presented in this chapter were obtained over a large period of time, evaluating progressively different releases of virtualization software as they were launched—first Xen 3.1, then Xen 3.2 and later KVM 84—to have an overview of the technological evaluation [7] [9] [2]. A summary of all performance evaluations of this chapter is given in Table 3.1.

3.2 Virtualizing the data plane

Our primary concern in designing of a virtualized network is the ability to configure routing *and* forwarding, e.g., protocols as well as specific operations on packets, such as filtering, queueing, scheduling, etc., on a per virtual network basis. For addressing this concern, it is necessary to virtualize the data plane.

3.2.1 Virtual router design

To evaluate the performance of communication mechanisms in virtual networks, we start by building a virtual router in software, with a virtualized data plane. We then evaluate its performance. This section describes its design requirements.

Configuration	Sending		Receiving		Forwarding	
	NET Rate	CPU Overhead	NET Rate	CPU Overhead	NET Rate	CPU Overhead
Xen	Section 3.3.3.1		Section 3.3.3.2		Section 3.3.4.1	
	100% (Fig. 3.6)	> 150% (Fig. 3.7)	94-96% (Fig. 3.11)	> 290% (Fig. 3.12)	80-100% (Fig. 3.16, Tab. 3.3)	290-415% (Fig. 3.17)
KVM	Section 3.3.3.1		Section 3.3.3.2			
	30-80% (Fig. 3.8)	> 220 to > 400% (Fig. 3.9)	> 64% (Fig. 3.13)	> 340% (Fig. 3.14)		

Table 3.1: Summary of performance evaluations in this chapter. Percentage of the network throughput compared to classical Linux throughput (NET) and the corresponding CPU overhead (CPU) on Xen and KVM.

Virtual routers are de-materialized instances, running in parallel on the hardware of a router. Each virtual router must have its own data plane with a dedicated forwarding engine, and its own routing plane, just like a standard software router. Figure 3.1 shows an example of such an architecture with software routers uploaded (control and data path) into virtual machines to create virtual routers. In this example, two virtual routers share the resources (NICs, CPU, memory) of a single physical machine. The governing principle

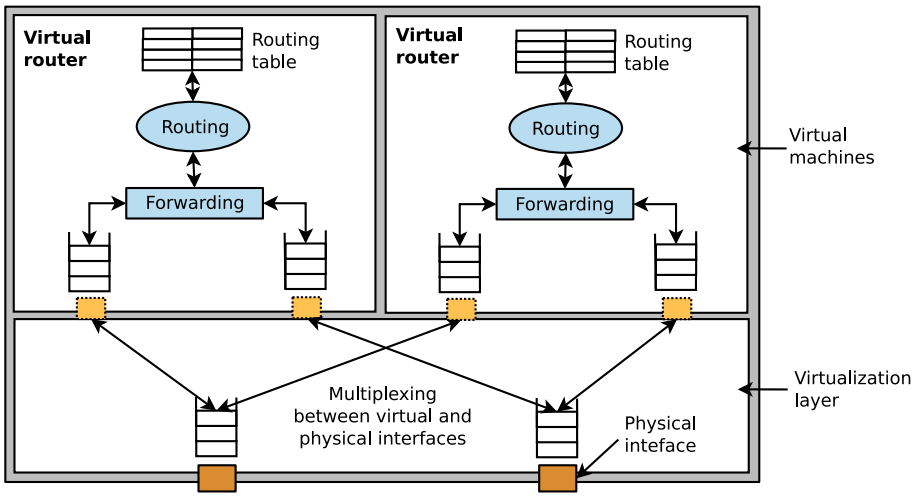


Figure 3.1: Machine with two virtual routers sharing the two NICs.

inside such a virtual router is the same as inside a standard software router, except that the virtual routers do not have direct access to the physical hardware interfaces. The packets are forwarded between the virtual machine interfaces and the corresponding physical interfaces thanks to a multiplexing and demultiplexing mechanism. This is implemented

in an intermediate layer located between the hardware and the virtual machines. As a consequence, additional computing is required in this design, whose impact on the network performance needs to be analyzed.

We decide to implement this preliminary virtual router design, using common virtualization techniques running on commodity hardware, which enable to virtualize the data plane. This choice is made for the sake of programmability and portability to any computing platform. Besides, today's general purpose hardware is powerful enough to make routing and forwarding in software a quite promising approach, at least at the edge of a network.

For implementing such a virtual router prototype, different available technologies are investigated in the following.

3.2.2 Available technologies

Virtualizing the data plane means having separate packet queues, forwarding tables, routing tables and routing daemons in each virtual router. Hence, while choosing an appropriate technology for virtualizing commodity hardware, our constraints are 1) that each virtual machine must be able to run its own OS with its own kernel, and 2) that each virtual machine has its own hardware abstractions, such as packet queues that can be configured. These constraints limit the choice of the technology to *full-* or *paravirtualization* (this choice results from the comparison of the different virtualization technologies in Section 2.3.4 of Chapter 2). Both enable virtualization to take place at the lowest level in software, directly above the hardware, so that each virtual machine performs network operations down to layer 2 of the OSI model.

The most popular full virtualization technologies are KVM [Kivity et al., 2007] and VMware [VMW], but also VirtualBox [VBX], while the mostly used paravirtualization technology is Xen. These technologies are compared in Figure 3.2, according to their level of configurability, and their I/O optimizations or network performance expectation. KVM offers great flexibility for experimentation, as it uses a Linux kernel that enables

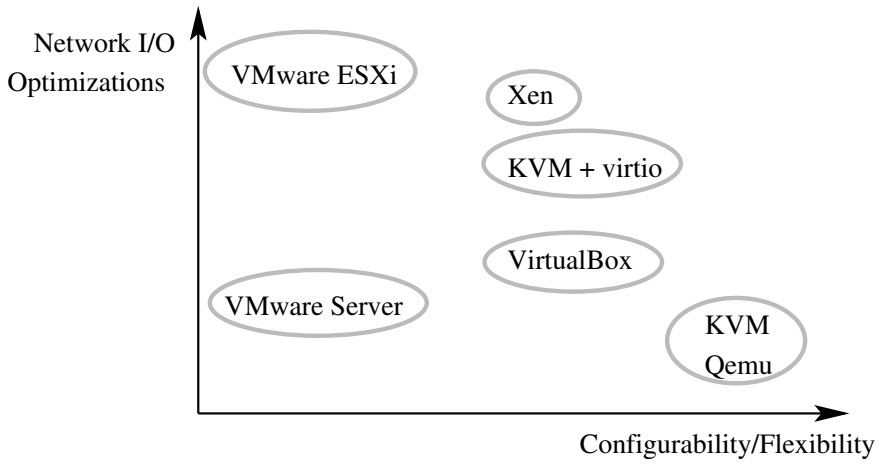


Figure 3.2: Comparison of full- and paravirtualization technologies.

full programmability. However, due to the emulation of the network devices, performance is potentially rather low. Yet, with the appearance of the *virtio* [Russell, 2008] tools,

consisting of a set of virtual I/O drivers, KVM became, as well, able to perform paravirtualization. VMware is probably one of the most optimized full virtualization technology and gives good network performance according to some preliminary tests we performed on ESXi and VMware Server. But it is less configurable as it is a closed system, and hence less adapted to our experimental context. VirtualBox in turn was designed at a first place for desktop virtualization, hence it may not be adapted to high network loads.

As both, Xen and KVM, combine promising performance with a high configurability, we retain these for building the virtual router prototype, analyze them further and evaluate them.

3.2.3 Virtualized data path

In this section, we examine the data path that packets take throughout a virtual router for each of the two technologies, Xen and KVM.

3.2.3.1 Data path in Xen

The virtual machines in Xen access the network hardware through the virtualization layer, called the virtual machine monitor or the hypervisor. Each domU has a virtual interface for each physical network interface it wants to use on the physical machine. This virtual interface is accessed via a *split-device driver* composed of two parts, the frontend driver in domU and the backend driver in dom0 [Chisnall, 2007]. Figure 3.3 illustrates these components, and the path followed by a network packet sent from a virtual machine residing inside a domU to the physical NIC. The memory page where a packet resides in the domU

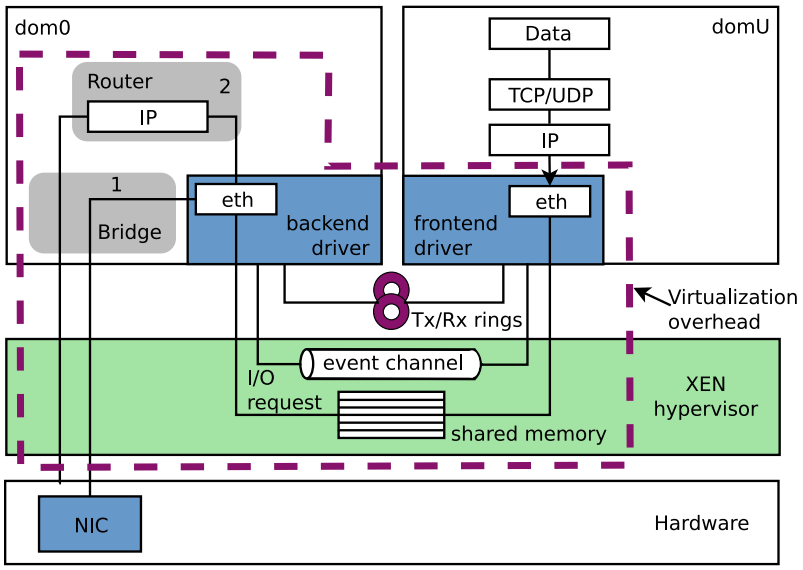


Figure 3.3: Path of a network packet with Xen, from a domU to the NIC.

kernel is either mapped to dom0 or the packet is copied to a segment of shared memory by the Xen hypervisor from where it is transmitted to dom0. Inside dom0, packets are bridged (path 1) or routed (path 2) between the virtual interfaces and the physical ones. The reception of packets on a domU is similar. To receive a packet, domU gives a grant to dom0 so that dom0 can access the grant page and copy the packet to domU's kernel

space [Santos et al., 2008].
The dashed line encircles the additional path a packet has to go through due to virtualization. A significant processing and latency overhead can be expected due to the additional copy to the shared memory between domU and dom0. Moreover, the multiplexing and demultiplexing of the packets in dom0 can be expensive.

3.2.3.2 Data-path in KVM

In KVM, the virtual machines are created as device nodes, and the host system operates as a hypervisor. It runs two KVM kernel modules, a modified Qemu [Bellard, 2005] module performing hardware-device emulation, and a processor-dependent module to manage hardware virtualization. When the virtual machines are network-intensive, an important performance overhead is expected, as emulation is a very costly procedure. Similar to Xen, KVM enables to use a virtual driver in paravirtualization mode. This virtual driver is part of the *virtio* drivers used within the Linux kernel [Russell, 2008]. Virtual machine kernels use also a frontend driver with particular code to communicate with a backend driver which interfaces with the KVM module inside the Linux kernel [Laor, 2007]. This architecture is represented in Figure 3.4. For communicating between frontend and back-

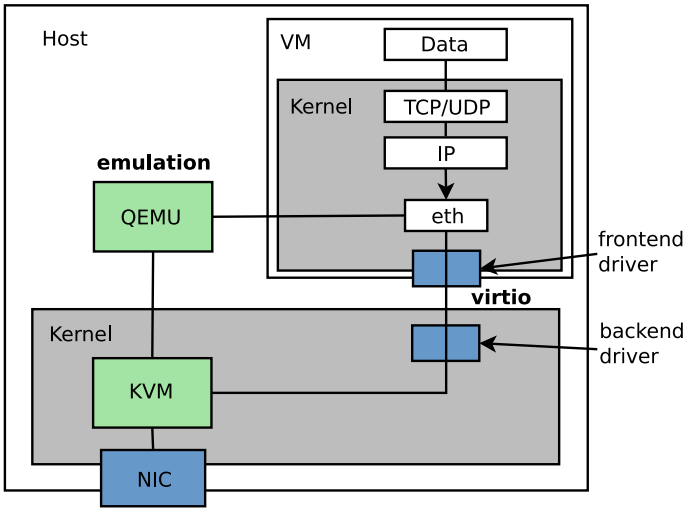


Figure 3.4: Path of a network packet in KVM using paravirtualization through virtio or full virtualization through emulation.

end, ring-buffers are used for the implementation of so called 'net channels' that allow the communication between virtual machines and the host, like in the virtual drivers in Xen.

3.3 Performance evaluation and analysis

This section describes the experiments, we carried out to examine the impact of the virtualization layer on the network performance. The throughput and processing cost of sending and receiving traffic in virtual machines is first evaluated separately. Results obtained on Xen and KVM are compared. As Xen shows more promising performance,

it is further used for building the virtual router prototype and evaluating its forwarding rate and latency.

3.3.1 Metrics

An efficient usage of virtual machines for networking requires a certain number of non-functional properties like *efficiency*, *fairness* in resource sharing and *predictability* of performance. We define and evaluate these properties using the metrics summarized in Table 3.2.

Metric	Description
R_i	Throughput of virtual machine i, $i \in [1, N]$
$R_{aggregate}$	$\sum_{i=1}^{N} R_i$: aggregate throughput on all N virtual machines
$R_{aggregate}/N$	Effective mean throughput
C_i	CPU cost on virtual machine i, $i \in [1, N]$
$C_{aggregate}$	$\sum_{i=1}^{N} C_i$: total CPU cost on all N virtual machines
L_i	Latency on virtual machine i, $i \in [1, N]$
$R_{classical(T/R))}$	Throughput for sending/receiving on classical Linux
$R_{classical(F)}$	Throughput for forwarding on classical Linux without virtualization
$C_{classical(T)}$	CPU cost of sending on classical Linux
$C_{classical(R)}$	CPU cost of receiving on classical Linux
$C_{classical(F)}$	CPU cost of forwarding on a classical Linux software router
$L_{classical(F)}$	Latency of forwarding on a classical Linux router
$R_{theoretical}$	Theoretical data throughput on 1 Gb/s Ethernet interfaces (941 Mb/s for TCP and 952 Mb/s for UDP)

Table 3.2: Metrics for measuring network performance on a physical machine hosting $N \in \mathbb{N}$ virtual machines.

For quantifying the performance of a virtual system, we define its *efficiency* as the ratio between its achieved metric and the the same metric achieved on a classical Linux system under the same workload.

For example, we define the efficiency in terms of throughput by (1)

$$E_{throughput} = \frac{R_{aggregate}}{R_{classical}} \quad (1).$$

The fairness of the inter-virtual machine resource sharing is derived from the classical Jain index [Jain et al., 1984], defined by (2).

$$Fairness(x) = \frac{\left[\sum_{i=1}^{N} x_i\right]^2}{N \times \sum_{i=1}^{N} x_i^2} \quad (2)$$

Here, N represents the number of virtual machines sharing the physical resources and x_i the metric achieved by virtual machine i, for example the throughput or the CPU percentage.

Predictability and scalability of the system are inferred by analyzing the performance based on the number of virtual machines.

3.3.2 Experimental setup

The experiments were all executed on the French national testbed Grid'5000 that hands over full control and configuration rights on its machines to the user during a reservation [Cappello et al., 2005] [G5K]. Hence, different types of machines were installed with Linux, as well as Xen and KVM for the purpose of these experiments. For the Xen end-hosts we used IBM eServers 325, with 2 AMD Opteron 246 CPUs (2.0 GHz/1 MB) with one core each, 2 GB of memory and a 1 Gb/s NIC. Virtual routers were deployed on IBM eServers 326m, with 2 AMD Opteron 246 CPUs (2.0GHz/1MB), with one core each, 2 GB of memory and two 1 Gb/s NICs. The experiments on KVM were executed on more recent machines provided with hardware virtualization enabled processors. These were Dell PowerEdges 1950 with two dual-core Intel Xeon 5148 LV processors (2.33 Ghz), 8 GB of memory, and 1 Gb/s NICs. In each experiment, machines inside one LAN interconnected by a single switch were used to avoid any additional latency or concurrent network load.

The precise software configurations used were Xen 3.1.0 and 3.2.1 with respectively the modified 2.6.18-3 and 2.6.18.8 Linux kernels. Comparative experiments were performed on KVM 84 with the Linux 2.6.29 kernel in the host system as well as in the virtual machine. With KVM, the default full virtualization (FV) with the emulated network driver, as well as paravirtualization (PV) with the virtio driver were evaluated.

Measurement tools were *iperf* [IPE] for the TCP throughput, *netperf* [NPE] for the UDP rate, *xentop* and *sar* for the CPU utilization, and the *ping* utility for measuring latency.

3.3.3 Sending and receiving performance

In the following experiments, each, sending and receiving performance for default maximum sized (1500 Bytes) packets, was evaluated on virtual machines implemented with Xen 3.1, Xen 3.2 and KVM. As some results with Xen 3.1 were not satisfying, dom0 being the bottleneck, a second run of the experiments on Xen 3.1 was performed, attributing more CPU time to dom0 (up to 16 times the part attributed to a domU). This choice was made as dom0 is in charge of forwarding all the network traffic between the physical and the virtual interfaces. In the case of 8 virtual machines, dom0 needs to handle the traffic of all 8 virtual machines on the physical interface, as well as on each of its virtual interfaces. Its load is hence estimated to be $8 + 8 = 16$ times higher than the load of a single domU. This setup will be called *Xen 3.1a*. KVM was used either in its native full hardware virtualization setup or using lightweight paravirtualization. The experiments were repeated ten times and avereage results are presented.

3.3.3.1 Sending performance

In this first experiment, the TCP sending throughput on 1, 2, 4 and 8 virtual hosts inside one physical machine, as well as the corresponding CPU overhead, were evaluated. Figure 3.5 shows the test setup, for the example with two virtual machines, acting as

senders from the same physical machine.

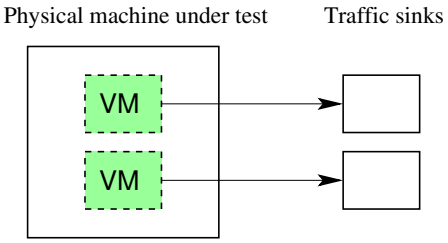


Figure 3.5: Experimental architecture for evaluating virtual machine sending performance.

The throughput per virtual machine and the aggregate throughput with Xen are represented on Figure 3.6. In both Xen configurations, 3.1 and 3.2, performance was close

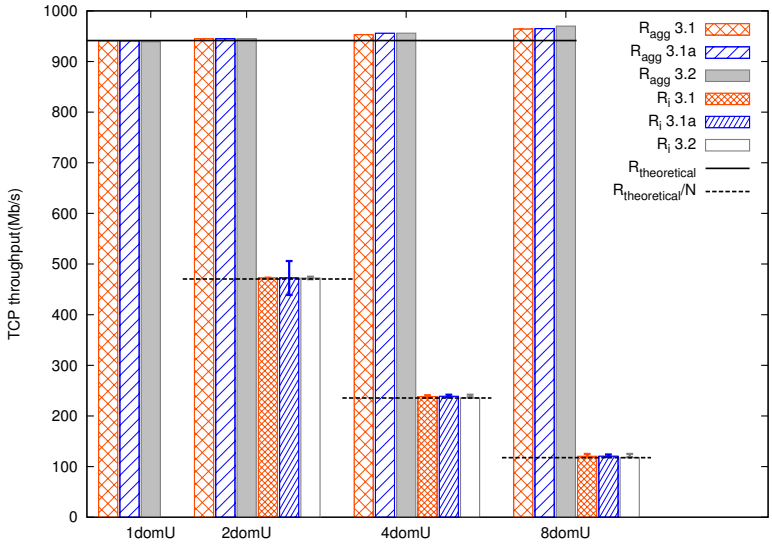


Figure 3.6: TCP Sending throughput on respectively 1, 2, 4 or 8 domUs with Xen versions 3.1 and 3.2.

to classical Linux throughput $R_{classical(T/R)} = 938$ Mb/s. In 3.1a and 3.2 setups, the aggregated throughput obtained by all the virtual machines was barely higher than on 3.1. We conclude that in the three cases (3.1, 3.1a, 3.2), the system is efficient and predictable, in terms of sending throughput. In addition, the sharing is fair as the throughput per virtual machine corresponds to the fair share of the available bandwidth of the link ($R_{theoretical}/N$) and the fairness index is equal to 1.

The associated average CPU utilization for each Xen guest domain is represented in Figure 3.7. For a single domU, around half the processing power of the two CPUs (i.e., 100%) was used in the three setups (Xen 3.1, 3.1a and 3.2), whereas on a native Linux system without virtualization, we measured that only $C_{classical(E)} = 64\%$ of a single CPU was in use running the same network benchmark. In the experiment with 8 domUs, the CPUs were used at over 140%. The overall CPU overhead did not differ much between Xen 3.1 and 3.2 setups. However, by increasing dom0's CPU weight (setup 3.1a), the overall

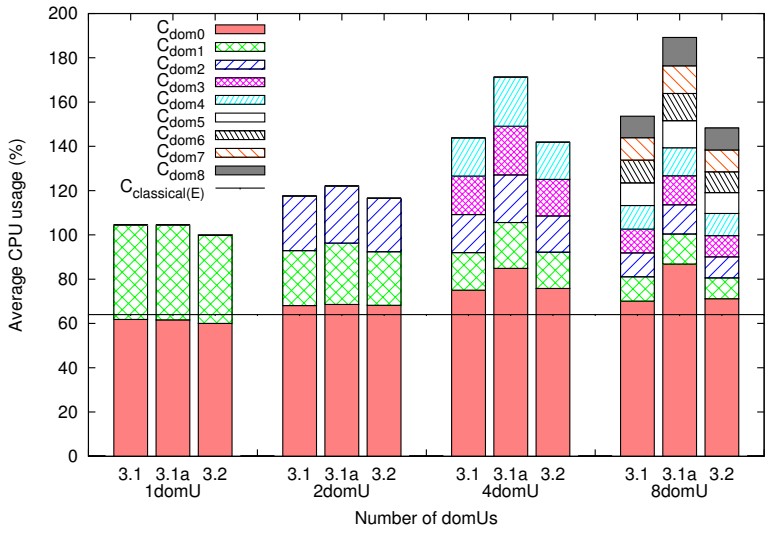


Figure 3.7: Average utilization of the two CPUs during TCP sending on 1, 2, 4 or 8 domUs with Xen.

CPU cost also increased while improving the throughput insignificantly. We notice that even though virtualization introduced a processing overhead, two processors like the ones used in these experiments achieved a throughput equivalent to the maximum theoretical throughput on 8 concurrent virtual machines, sending TCP flows of default maximum-sized packets on a 1 Gb/s link. Here, the fairness index was close to 1, bandwidth and CPU time were fairly shared between the different domUs.

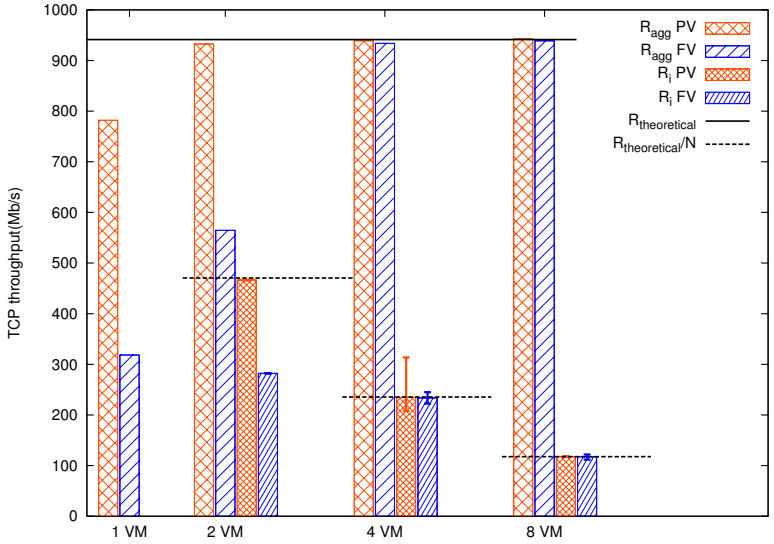


Figure 3.8: TCP Sending throughput on 1, 2, 4 or 8 virtual machines with KVM 84 using paravirtualization (PV) or full virtualization (FV).

The same experiment was executed on virtual machines implemented with KVM. KVM

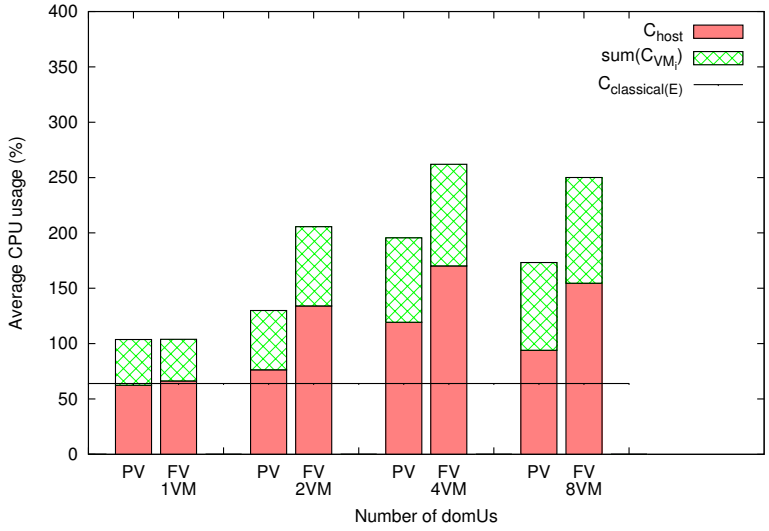


Figure 3.9: Average utilization of the four CPU cores during TCP sending on 1, 2, 4 or 8 VMs with KVM.

was used in two different configurations: paravirtualization (PV) using the virtual network driver from *virtio*, and native full hardware virtualization (FV) where network drivers are emulated with Qemu. The results in terms of throughput are represented in Figure 3.8. Paravirtualization clearly outperformed emulation in terms of network performance. For a single virtual machine, KVM with the emulated driver reached only around $E_{throughput} = 30\%$ of the native Linux throughput, while using and 100% of a CPU. In this case, the bottleneck was obviously the processing overhead. In the case of several virtual machines, where each one was assigned a different CPU core, the throughput increased, as more CPU cores were used. The CPU overhead was higher than 220% for paravirtualization, and over 400% for full virtualization.

3.3.3.2 Receiving performance

In this experiment, the TCP receiving throughput on 1, 2, 4 and 8 concurrent virtual machines and the corresponding processing overhead were evaluated. Figure 3.10 shows this test setup, for the example with two virtual machines, allocated on the same physical machine, and receiving data simultaneously.

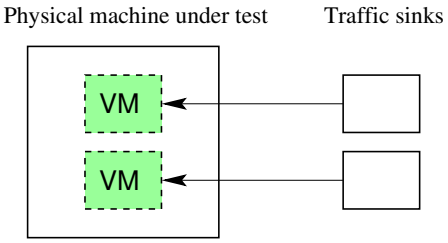


Figure 3.10: Experimental architecture for evaluating virtual machine receive performance.

Figure 3.11 represents the results of this experiment in terms of TCP throughput per

domU and aggregate throughput with Xen. We notice that the aggregate throughput

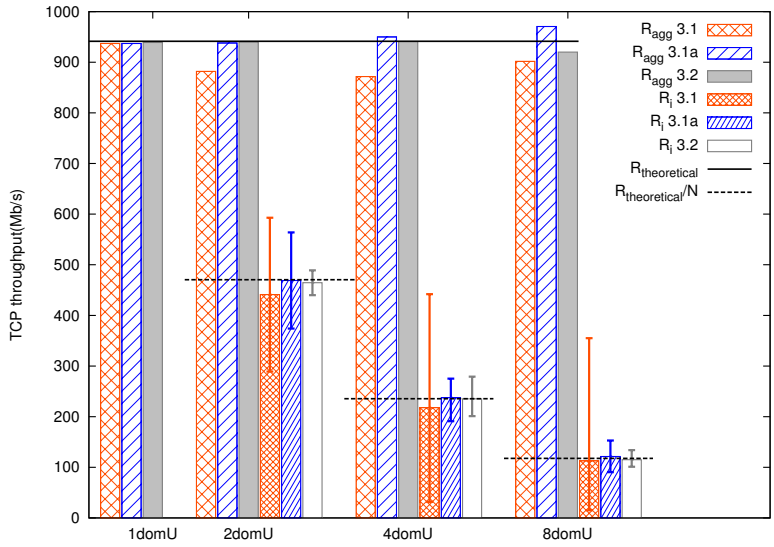


Figure 3.11: TCP Receiving throughput on respectively 1, 2, 4 or 8 domUs with Xen versions 3.1 and 3.2.

decreased slightly according to the number of virtual machines on Xen 3.1. It reached only 882 Mb/s on two domUs and only 900 Mb/s on a set of 8 concurrent domUs, which corresponds to about 96% of throughput $R_{classical(T/R)} = 938\ Mb/s$ on a classical Linux system. Efficiency $E_{throughput}$ varied between 0.96 for 8 domUs and 0.94 for two domUs. By changing the scheduler parameters (Xen 3.1a), we managed to improve the aggregate throughput to reach about 970 Mb/s on 8 virtual machines, thus making the system efficient.

Besides, the sharing of the bandwidth between the domUs in Xen 3.1 was increasingly unfair when incrementing the number of domUs. This problem was related to an unfair treatment of the events (domUs created later got less chance to get their events treated than those created earlier, as in each round, the scheduler started with the first created domU, even if a round was interrupted before all domUs were served). It was fixed in Xen 3.2 by modifying the scheduling algorithm [Ongaro et al., 2008]. Our solution that provided dom0 with more CPU time simply (3.1a setup) allowed also to improve fairness in Xen 3.1 by giving dom0 enough time to treat all the events before the scheduler ran out of credits and started switching unnecessarily between dom0 and domUs. The resulting fair resource sharing made performance much more predictable. The measured aggregate receiving throughput in Xen 3.2 was more similar to the Xen 3.1a results with the modified scheduler parameters. The throughput increased by about 6% compared to the default 3.1 version. Figure 3.12 gives the CPU time distribution among the guest domains. The total CPU cost of the system varied between 140% and 150% in the default Xen 3.1 and 3.2 versions, which represents an important overhead (over 290%). Indeed, on a Linux system which is not virtualized network reception at maximum speed takes only $C_{classical(R)} = 48\%$ of the CPU using the same benchmark. We notice that on the default Xen 3.1, the efficiency in terms of throughput decreased, while the available CPU time

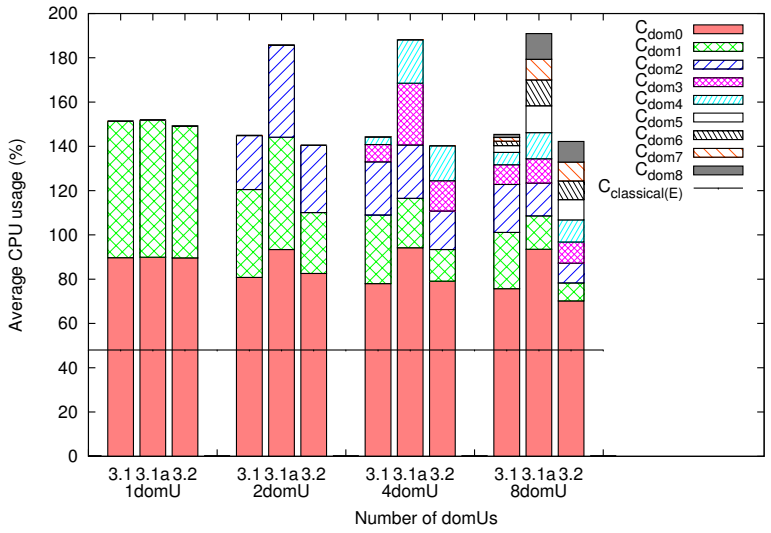


Figure 3.12: Average utilization of the two CPUs during TCP receiving on 1, 2, 4 or 8 domUs.

was not entirely consumed. Also, the distribution of the CPU time consumption among the domUs followed the same unfairness pattern than for the throughput. This shows that the virtual machine scheduler on the CPU looses efficiency when stressed with networking. The fairness index decreased to only 0.46 on 8 concurrent domUs on Xen 3.1 because of the described scheduling problem.

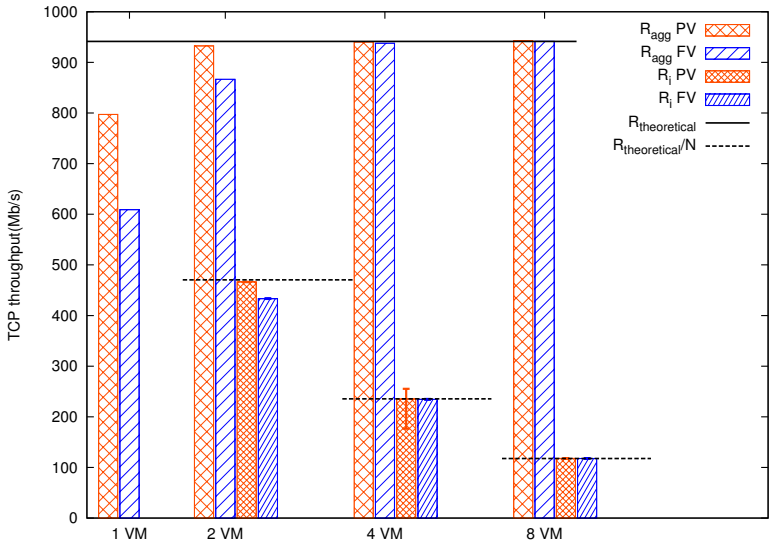


Figure 3.13: TCP Receiving throughput on 1, 2, 4 or 8 virtual machines with KVM 84 using paravirtualization (PV) and full virtualization (FV).

In summary, our proposal to readjust the scheduler parameters (setup 3.1a) improved fairness in Xen 3.1, but increased CPU consumption. The Xen 3.2 version showed sim-

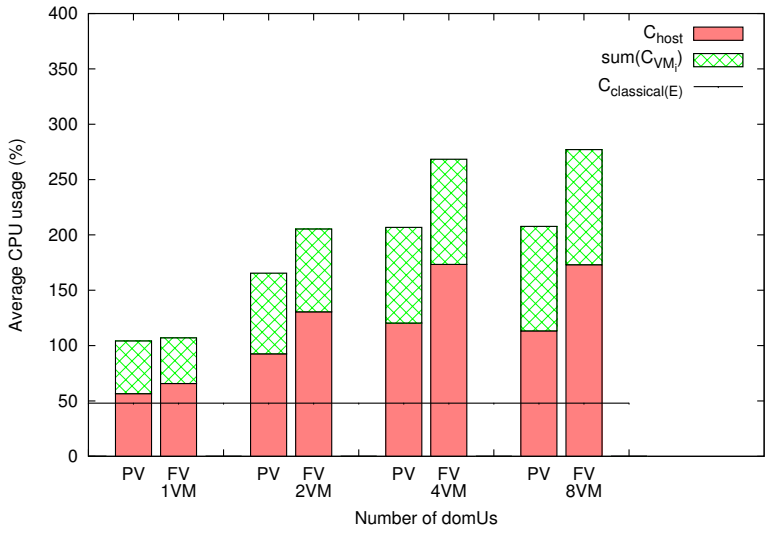


Figure 3.14: Average utilization of the four CPU cores during TCP receiving on 1, 2, 4 or 8 virtual machines with KVM.

ilar trends, increasing throughput and removing unfairness. Moreover, CPU utilization decreased slightly. It was still more efficient than in Xen 3.1, showing less total CPU overhead while achieving even better throughput. We conclude that important improvements have been implemented in Xen 3.2 to decrease the excessive dom0 CPU overhead.

In comparison, KVM using the virtualized network driver achieved very similar results in sending performance. Using a single CPU core, as in the case of one virtual machine, is not enough to achieve maximal Linux throughput as shows Figure 3.13. Throughput reaches only around 800 Mb/s in paravirtualization mode, and barely more than 600 Mb/s in full virtualization mode, which corresponds to an efficiency $E_{throughput}$ of around 64%. Figure 3.14 shows that the CPU overhead for receiving is slightly more important than for sending using virtio paravirtualization driver. This is similar to the results obtained with Xen 3.2, which nevertheless used between 10% and 30% less processing to achieve the same throughput. The CPU overhead corresponds to respectively 340 and 560% for achieving 100% throughput efficiency with paravirtualization and full virtualization. In the case of full virtualization, KVM's receiving mechanism is more efficient than its sending mechanism, but it still does not reach Xen's performance, needing three of the available CPU cores to achieve maximum Linux throughput.

3.3.4 Forwarding performance

To evaluate the forwarding performance of virtual routers, we sent UDP traffic over virtual routers with 2 NICs, as depicted in Figure 3.15, and measured the throughput obtained on the receiver, and the packet loss. We repeated the experiment using TCP traffic, and measured also throughput, as well as latency.

For this experiment, only Xen 3.2 was used, which was the best performing technology in the previous experiments. The results were obtained on Xen 3.2 in its default configuration and with an increased weight (up to 32 times the weight of a domU) for dom0 in

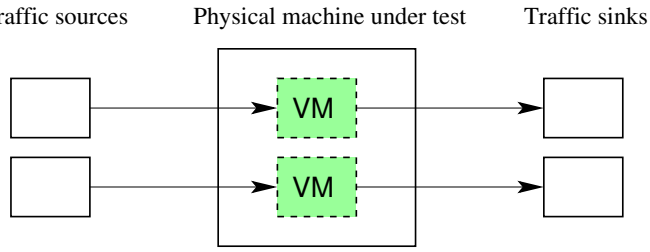


Figure 3.15: Experimental architecture for evaluating virtual machine forwarding performance.

CPU scheduling. We call this setup *Xen 3.2a* in the following.

3.3.4.1 Throughput and processing cost

For determining the performance of virtual routers, UDP traffic was generated with either maximum- (1500 bytes) or minimum- (64 bytes) sized packets over one or several virtual routers (from 1 to 8) sharing a single physical machine. All the flows were sent at maximum rate from distinct physical machines to avoid bias. Next, end-to-end TCP throughput was also measured.

Figure 3.16 shows the obtained UDP bit rate with maximum-sized packets and the TCP throughput. The corresponding CPU cost is represented in Figure 3.17. Table 3.3 details the UDP packet rates and the loss rates per domU with maximum- and minimum-sized packets.

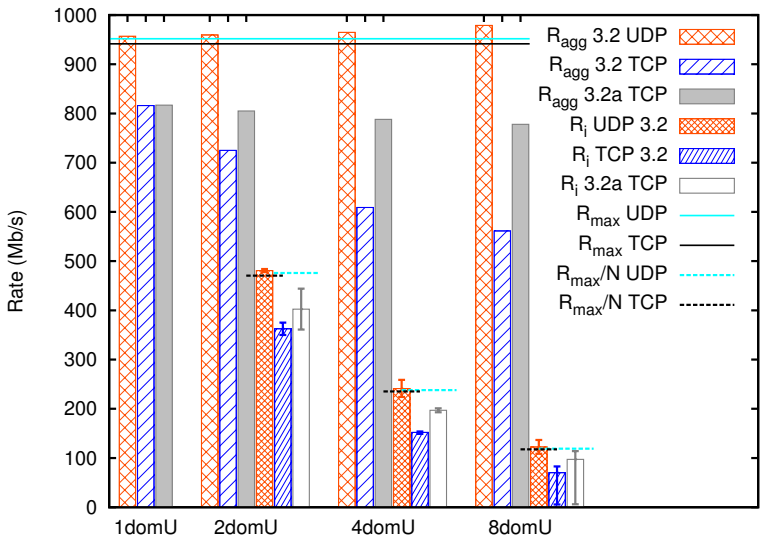


Figure 3.16: Receiver side throughput over 1, 2, 4 or 8 virtual routers with Xen 3.2 forwarding 1500-byte packets.

With UDP, the percentage of loss with maximum-sized packets on each virtual machine corresponds to $1-(1/N)$. This means that the bandwidth is fairly shared between the virtual routers. The results show efficiency, the throughput corresponds to the value obtained

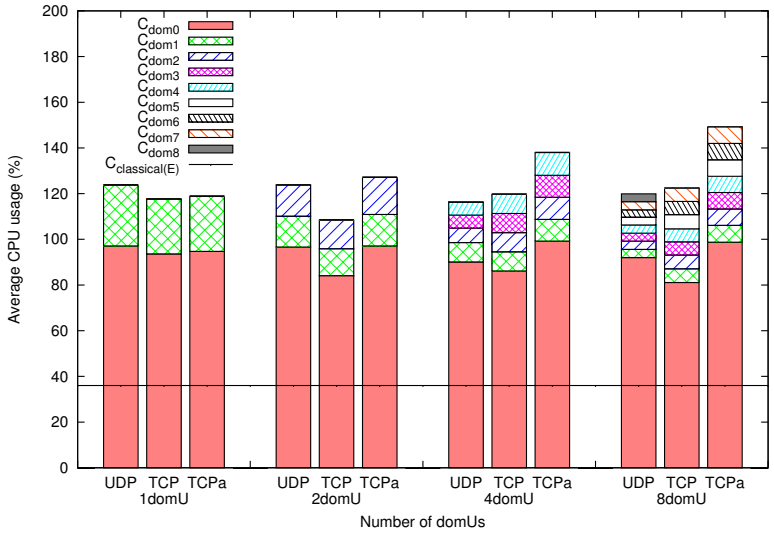


Figure 3.17: CPU cost of 1, 2, 4 or 8 virtual routers with Xen 3.2 forwarding 1500-byte packets.

	1500 byte packets		64 byte packets	
	pps/domU	loss/domU	pps/domU	loss/domU
Classical Linux	81277	0.00 %	356494	0.06 %
1 VR	81284	0 %	109685	60 %
2 VR	40841	50 %	12052	96 %
4 VR	20486	75 %	/	/
8 VR	10393	87 %	/	/

Table 3.3: Average UDP packet-forwarding rate (pps) and loss rate per domU hosting a virtual router (VR).

on a classical Linux router $R_{classical(F)} = 957\ Mb/s$. The aggregate UDP throughput was in some cases a bit higher than the theoretical value due to little variation in the start times of the different flows. Resource sharing was fair: in this case, performance is predictable. With maximum-sized packets, dom0 used an entire CPU, forwarding at maximum rate with UDP and only 80% of the maximum throughput with TCP, having an efficiency of $E_{throughput} = 0.80$. With minimum-sized packets on 4 or 8 virtual routers, dom0 became too overloaded, not being able to forward on all virtual routers anymore.

Regarding the processing of the router, dom0 used much more CPU resources than the domUs, compared to the simple sending or receiving scenario. On a classical Linux router, we measured that the forwarding of a flow at maximum speed generated a CPU load of $C_{classical(F)}$ =36%. However, using virtual routers inside Xen domUs generated a CPU load between 105 and 150% as shown in Figure 3.17. Hence, forwarding inside virtual machines generates an overhead between around 290 and 415%. This high overhead is due to the fact that it has to forward the traffic of twice as many virtual interfaces than before (16 in the case of 8 virtual routers). In the case of UDP and TCP with the

modified scheduling parameters (named TCPa), the usage of domU's CPU was pushed to its maximum. It used an entire CPU. Hence, the TCP throughput was obviously limited by dom0's processing limitation. In the case of TCP forwarding with the default CPU scheduling parameters, dom0 did not get enough processing resources. This is especially the case when the number of virtual routers, and hence the number of virtual interfaces to treat, were increased. As a consequence, the throughput decreased. With the important overload generated by 8 virtual routers, the last domU was not even able to forward packets anymore, and thus used no CPU.

With TCP, even on maximum-sized packets, the throughput throttled down, especially with an increasing number of virtual routers.

3.3.4.2 Latency

In this experiment, the latency on one virtual router (VR) was measured, while concurrent virtual routers (1, 3 or 7) sharing the same physical machine were either idle or stressed forwarding maximum-rate TCP flows. Table 3.4 represents the results in both cases. The

	Linux	1 VR	2 VR	4 VR	8 VR
idle	0.084	0.147	0.150	0.147	0.154
stressed			0.888	1.376	3.8515

Table 3.4: Latency in ms over one VR among 1, 2, 4 or 8 VRs idle or stressed with TCP forwarding.

latency over a virtual router sharing the physical machine with other idle virtual routers was about 0.150 ms, no matter the number of virtual routers, which was almost the double of the latency on a classical Linux router $L_{classical(F)} = 0.084ms$. In the case of a stressed system, the latency on the considered virtual router increased with the number of concurrent virtual routers forwarding maximum throughput TCP flows. The average latency reached nearly 4 ms on a virtual router sharing the physical machines with 7 virtual routers forwarding TCP flows. The more virtual machines asking for the scheduler, the more the latency on the virtual router increased.

3.3.5 Discussion

The above results confirm our initial assumption that emulation is more costly than paravirtualization for the network. This is shown since the simplest scenario, where a single virtual machine sends a flow to a distant machine. In this scenario, KVM using the emulated network interface consumes the power of an entire CPU for achieving only little more than 300 Mb/s of network throughput. On the contrary, on Xen, the same throughput as on a classical Linux system can be achieved with a single CPU. Using virtio instead of the emulated interface improves KVM's network performance significantly. A virtual machine achieves almost 800 Mb/s of TCP throughput. However, this is still less than the throughput on Xen, and the generated CPU cost is higher in KVM.

In all the configurations, with KVM as well as Xen, a more or less important CPU overhead has been observed, as forwarding packets from physical NICs to the virtual machines is a costly operation. This is mainly related to the additional copy that is needed to deliver a packet from the host memory space to the virtual machine memory space.

A fundamental problem is that virtual machines are not designed to deal with excessive network loads. The arrival of a network packet triggers an interrupt, so that the machine temporarily stops its current task and treats the packet. Interrupts, when occurring too frequently, are difficult to handle by a virtual machine scheduler such as Xen's credit scheduler [XCS]. When having consumed its credits, a domU or the dom0 can be interrupted at any time by a packet arrival until its credits are reprovisioned. Such an interrupt causes a context switch to the domain (domU or dom0), which is responsible of handling the packet. Credits are reprovisioned once all domUs and dom0 finished them. The interrupt mechanism implemented in software routers to allow them handling packet arrivals while executing other tasks, reduces efficiency, as interrupt handling is an expensive operation. Therefore, dedicated network equipments rather poll the NICs to retrieve packets, which would also be more appropriate in virtual routers for improving their performance.

In the interrupt model, one solution to get 100% efficiency is to use a 'powerful enough' (this depends on speed of the network interfaces, the number of the network interfaces, etc.) CPU. Yet, it would be better to treat the problem at the root, improving the implementation of I/O mechanisms to reducing the overhead, and/or using special virtualized network cards that enable direct access by virtual machines. All these approaches are progressing: Hardware becomes more and more powerful, and many optimizations have been brought to the software since virtualization became popular. The following gives an overview of this evolution.

3.4 Comparison to previous results and follow up

Even though now both, Xen and KVM are commonly used for virtualization, more research has been performed on Xen. The main reason for this is probably that Xen, was released long before, in 2003, while KVM featuring hardware virtualization became available only in 2007. As a consequence, the Xen I/O drivers have been subject to much more optimization over the time than the KVM network stack. This section discusses the different research results over time, on the successive Xen versions, as well as KVM. From these and our results, the trends in the evolution of virtual network performance on commodity hardware can be deduced.

On Xen 2.0, Menon et al. [Menon et al., 2006] measured default transmit and receive TCP throughput on a domU below 1000 Mb/s using four 1-Gigabit NICs, which is much less than the throughput on a non virtualized Linux system (941 Mb/s per 1G NIC). The authors improved Xen's networking performance by modifying the virtual network interfaces to include hardware NIC features and optimize the I/O channel between dom0 and domU. After optimization, they obtained results for transmissions closer to Linux throughput: 3310 Mb/s on four NICs which corresponds to around 830 Mb/s per NIC, but still less for receptions (only 970 Mb/s on four NICs) which corresponds to only 26% of the receiving throughput we measured on Xen 3.2.

A study about scheduling on Xen 3 (unstable version, changeset 15080) [Ongaro et al., 2008] confirms our results with Xen 3.1, of the unfair bandwidth distribution among the domUs in the default configuration, with the credit scheduler [XCS]. The authors measured an aggregate throughput of less then 800 Mb/s on 7 domUs, varying from less than 25 Mb/s to around 195 Mb/s per domU. They proposed event-channel improvements which enhanced the fairness in the sharing of the bandwidth between the virtual machines,

to vary only about ±25 Mb/s on the different domUs. We noticed this improved fairness in the Xen 3.2 version.

An evaluation of Xen 3.0 for routers [N. Egi et al., 2007] shows that the aggregate forwarding throughput on two to six domUs reaches less than 25% of what is achievable on classical Linux for 64-byte frames.
In a more recent paper [Egi et al., 2008a], the authors showed that memory access time is the main system bottleneck with Xen. They finally proposed a system to map forwarding paths to CPU cores so that as many packets as possible can be kept in the closest CPU cache to limit costly memory accesses [Egi et al., 2009]. This system is an interesting step to improve the performance of forwarding in software, but there is no real data plane virtualization as packet data remains in dom0 and only routing is performed in domU. It is an interesting tradeoff between performance and the level of virtualization in software virtual routers.

In parallel to these evaluations and optimizations of Xen, KVM appeared as a new full virtualization solution. In [Zeng and Hao, 2009], the authors showed that the network driver emulation generated high CPU cost, related to the use of a tap device, the data copy between kernel and user space, and the bridging module in the host. These explain the low performance in the fully virtualized machines. To improve its low I/O performance, due to the emulation of the drivers, paravirtualization was also included to KVM within the *virtio* drivers. Nevertheless, as our results show, Xen outperforms KVM with virtio when it comes to the network performance. Virtio performs a sort of tradeoff between performance and genericity, as it can be integrated with different technologies. On the contrary, Xen's virtual drivers are tailored to the Xen technology and, as described before, have gone through several optimizations. This can explain their better performance to date.

Very recent evaluations of Xen 4.0 have shown similar results to those we obtained on Xen 3.2, achieving 100% throughput efficiency only with 1500 Byte packets, and suffering from poor network performance under high CPU load [Schlosser et al., 2011]. But Xen follows up improving its performance. More recently, a major improvement to Xen's network virtualization mechanisms has been proposed, namely *netchannel2* [Santos et al., 2008]. It aims at reducing overhead due to the copy of packets between host and virtual machines, scheduling and packet fragmentation. While its implementation in the more recent Xen 4.0 version is not yet definitive, it could bring major improvements in the future.

To summarize these evolutions, Table 3.5 lists the main issues of the successive Xen versions and KVM, and the key improvements of each version, from a network point of view.

3.5 Conclusion

In this chapter, we evaluated virtual end-host and router performance in terms of TCP and UDP transmission rates, the corresponding CPU cost and latency on different Xen versions. The results were compared to those we obtained on KVM. Virtualization mechanisms like additional copy and I/O scheduling of virtual machines sharing the physical devices were shown to be costly in terms of processing. Nevertheless, our results show that virtualizing the data plane by forwarding packets inside the virtual machines becomes an

	Evaluated problem	Network Throughput Efficiency	Key improvement
Xen 2.0	Simultaneous sending/receiving throughput on a virtual machine on 4 1-Gb NICs	≈25% of native Linux throughput	Offloading of functionality to the vNICs and I/O channel modification leading to over 300% improvement [Menon et al., 2006]
Xen 3.0	Forwarding rate on several virtual machines [N. Egi et al., 2007]	less than 25% of native Linux due to a CPU bottleneck	The credit-scheduler: an SMP load balancing CPU scheduler for sharing the CPUs among virtual machines [XCS]
Xen 3.1	Unfairness between virtual machines to access CPU resources	Less than 80% of native Linux throughput	Improvement of the Credit scheduler, event channel modification [Ongaro et al., 2008]
Xen 3.2-4.0	Important CPU overhead while forwarding, especially for I/O of small packets [7]	Nearly 100% with 1500-byte packets, but less than 5% with 64-byte packets when sent at maximum rate	Main improvements in netchannel2 [Santos et al., 2008]
KVM 84	Huge CPU overhead due to tap device and copy [9]	Less than 30% with emulated NICs	*virtio* [Russell, 2008] driver for paravirtualization of the network I/O, leading to nearly 80% of efficiency

Table 3.5: Comparison and performance evolution of Xen versions and KVM.

increasingly promising approach as technology improves. We show that throughput efficiency improved in Xen 3.2 compared to 3.1, reaching now close to 100% throughput for big packets. Xen, with its virtual network drivers, outperforms KVM, requiring significantly less processing power. Our virtual router prototype build with Xen is suitable for network experimentation virtualizing the data-path. It has been integrated in the HIPerNet[1] platform (cf. Chapter 6), where it is involved in traffic control and forwarding customization. However, we are aware that maximum Linux throughput can not be taken for granted on this virtual router, especially for the transmission of small packets.

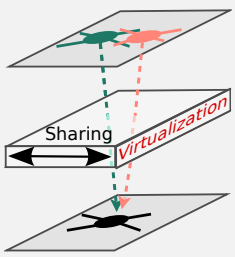
[1] http://www.ens-lyon.fr/LIP/RESO/Software/hipernet

In addition, for pushing virtualization further inside the network, ensuring good performance is crucial, especially as 10 Gb/s links move closer and closer to the edges. We conclude from the results of this chapter that general purpose OSes and server hardware are not adapted to high-speed networking. Hence, to bring virtualization to the Internet, it has to be implemented directly in hardware, on special purpose devices; the subject of the next chapter.

4 Virtualizing the switching fabric

The work presented in this chapter has been published at the 11th International Conference on High Performance Switching and Routing (HPSR) in 2010 [5]. In addition, the work is part of the solutions patented by the LYaTiss company (`http://www.lyatiss.com`) and the Institut National de Recherche en Informatique et en Automatique (INRIA, `http://www.inria.fr/`) [VxSwitch].

Abstract. For bringing full virtualization to production networks, such as the Internet, it is necessary to upgrade the current hardware architectures to cope with high-speed transmissions. The contributions of this chapter are:



- The design of a virtualized switching fabric, enabling controlled layer 2 *resource sharing* and configurability;
- A simulator and the evaluation of VxSwitch, especially on the impact of resource sharing; and
- Use cases showing how virtualizing the data plane in routers and switches can enable realizing new features in the network.

4.1 Introduction

Current virtual router implementations allow either high configurability *or* high performance, but not both at the same time. Highly configurable virtual routers are those that virtualize all their elements, including the data plane, and that allow one to customize these elements. The prototype we presented in the previous chapter is a high configurable virtual router, but its current performance makes it unsuitable for production networks such as the Internet. High performance virtual routers are those that enable network consolidation, e.g., Juniper logical routers [Kolon, 2004] [JCS, 2009] or Cisco Nexus [HVR, 2008]. In contrast to programmable routers, they enable only standard configuration, and their scalability is tightly bound to the hardware, e.g., the number of queues, the number of forwarding engines. For enabling innovation in the Internet, both, *configurability and performance* are required.

Improving the performance of fully configurable, e.g., software routers, is one approach that has been explored. Parallelization has been suggested as a solution, as it can be realized aggregating commodity hardware [Liao et al., 2009] [Dobrescu et al., 2009]. However, such hardware is not designed for packet switching: a limited number of network interfaces is interconnected by a bus, hence bounding the switching performance. Thus, the opposite approach is needed for bringing virtualization to the Internet. It consists in adding the configurability needed for virtualization to hardware dedicated to switching.

Such configurability requires to *share buffers and ports* among virtual buffers and virtual ports, so that each virtual router can provide configurable and deterministic link capacities. Moreover, strong isolation between virtual networks at the data plane level is ensured. Each virtual router can have customized switching and routing. We propose VxSwitch as a virtualized hardware switch whose switching fabric design allows the controlled sharing of the switching hardware [VxSwitch] [5]. *VxSwitch enables creating virtual switches that are dimensioned in terms of capacity and buffer size, and configurable in terms of layer 2 functionalities.* The next section motivates the need for virtualization at layer 2, and Section 4.3 describes the design of the virtualized switch. VxSwitch is evaluated in Section 4.4 and applications are shown in Section 4.5.

4.2 Virtualizing the fabric

In the previous chapter, we introduced the need for virtualizing the network data plane. In hardware, this means virtualizing the switch ports, buffers and operations such as packet queueing and scheduling. Besides isolation and scalability, this would introduce invaluable *configurability* to the network *data plane*, while maintaining performance. By configurability at this level, we mean the ability to, 1) *dynamically dimension a virtual device's resources*, such as port density, port capacity and buffer, and 2) *customize its layer 2 operations*, such as packet scheduling and buffer management.

Considering that different operators exploit virtual networks leased on infrastructure provider's equipment at a time, as proposed in [Feamster et al., 2007], the switching resources could be more efficiently used. If providers would expose routers with a virtualized switching fabric, virtual network operators could reserve the exact amount of resources they need, in addition to deploying custom functionality.

4.2.1 Controlled sharing

The way of sharing the hardware between virtual switches is of primary importance to ensure the following fundamental features:

- *Isolation between virtual switches:* A virtual switch must not affect the performance of another virtual switch allocated on the same physical device.
- *Determinism inside virtual switches:* Each virtual switch must behave like a dedicated physical switch. It must deliver deterministic performance and latency, according to the configuration requested for a lease. That is required as a basis for a contract. A virtual switch user must be exposed exactly the hardware resources he needs.
- *Efficient resource sharing:* The switching hardware can be shared in a more efficient way, avoiding overprovisioning of buffer and capacity, as well as preventing from underprovisioning.

Given these constraints, involving switching capacity and buffer, it is necessary to virtualize a network device at the lowest level—at the switching fabric. This means virtualizing the hardware (for sharing in space), as well as the layer 2 operations (for sharing in time) for enabling configuration. Like this, each virtual network device can use dedicated virtual hardware, and has dedicated time to operate it.

To the best of our knowledge, this is not possible in current devices. Traditional approaches include queuing and priority scheduling in VLANs. Ports have a fixed number of queues and each VLAN can use a different queue. However, the size of the queues can not be configured dynamically for each VLAN, and the number of queues is limited by the hardware implementation. This violates our constraint of efficient resource sharing. Especially that queues are generally overprovisioned to be able to deal with bursty bulk traffic. But this configuration may not be suitable for any type of VLAN.
An interesting approach that allows to assign capacity on a per flow basis has been proposed in [Kim et al., 2010]. Burst size can be set for a flow, but not buffer size, and no layer 2 operations can be configured. This brings to the second requirement on a virtualized switch: Each virtual switch should be flexibly configurable.

4.2.2 Configurability

The possibility to configure virtual network devices at layer 2 would allow operators to not only set up the software stacks and protocols of their networks at their will, but also to provision equipment with the exact amount of resources they would prefer to. Virtual equipment could be provisioned through the dimensioning of its ports, i.e., their number and capacity, and the dimensioning of its buffers. Both are relevant to QoS. Indeed, for configuring precise performance levels in terms of rate and loss, as well as latency and jitter, not only the capacity and the operation of the switch, e.g., the scheduling, but also the size of the buffers are of great importance [Vishwanath et al., 2009]. This brings the need for exposing buffer as a service. For example, small buffers are better for traffic requiring low latency, while big buffers are needed for high-speed bulk traffic, with a bursty pattern. Hence, depending on the type of traffic and the protocol used, buffer requirements are different, as shown in the multitude of research that has been done on buffer sizing.

Based on the property that most of the traffic on the Internet is TCP [Lee et al., 2010], the impact of buffer size on TCP flows has been investigated in various work. For example, for TCP flows, packet loss and latency are important caveats to their performance. For minimizing them, buffer sizing plays an important role, but that depends on multiple criteria such as the size and the number of flows, besides the transmission rates. In [Prasad et al., 2009], the authors show for example that the ratio between output and input capacity of a router determines the relationship between loss probability and buffer size. As another example, there is the well approved rule of sizing buffers to $C \times RTT/\sqrt{N}$, as a derivative of the bandwidth-delay-product, which holds for a large number of flows N, on a path with a capacity of C and a round-trip time RTT [Appenzeller et al., 2004]. The important revelation from this rule is that the buffer-size dimensioning depends on the number of flows multiplexing. Applied to core routers, carrying millions of flows, buffer size can be considerably reduced. However, by the definition of N, sizing queues according to this rule is not suitable at the edges, where the routers multiplex much less number of flows. Moreover, it has been shown that the rule does not apply on short-lived TCP flows, hence increasing their packet loss and latency [Tomioka et al., 2007]. Otherwise, buffer sizes impact the loss synchronization of TCP flows, as demonstrated in [Hassayoun and Ros, 2009]. An experimental study on SLAs, considering latency, loss rate, throughput, jitter, shows that router buffers must be sized differently, depending on the number of flows, their sizes and congestion control algorithms [Sommers et al., 2008]. It has moreover been considered, to adapt buffer sizes dynamically to the minimum value satisfying particular utilization or loss constraints [Zhang and Loguinov, 2008]. Considering the diversity of traffic and the impact of the buffer size on the performance of each type of traffic, we conclude that different virtual networks that carry different types of traffic should also be able to be configured with different virtual buffer sizes.

A part from resource dimensioning, configurability is also required in the operations of virtual switches, so that each can set up a specific QoS. The operations in the switching fabric consist in packet queuing and scheduling. In order to meet specific latency and rate requirements on a per flow basis, the packet schedulers of a virtual switch need to be configurable, to set up different policies. A virtual switch forwarding homogeneous traffic on all ports may for example be configured with a simple round-robin scheduling algorithm, while a switch transporting different types of traffic may be configured in order to implement differentiation. Priority scheduling could for example enable a switch to prioritize a specific flow.
Moreover, configurability of the queueing mechanisms could enable virtual switches to enqueue packets following different policies. A virtual switch could for example set up algorithms performing active queue management such as RED (Random Early Detection) [Floyd and Jacobson, 1993], to better control its queue sizes and satisfy specific QoS requirements.

To introduce such configurability—i.e., resource dimensioning and operation customization—to virtual networks, the architecture of router's and switch's data planes have to be upgraded.

4.3 VxSwitch: A virtualized switch

In this Section, *VxSwitch*, the design of a virtualized switch architecture is proposed. It can be used for virtualizing the network data plane and building virtual network devices such as routers with dedicated virtual hardware and switching functionality.

4.3.1 Design goals

From the possibilities emerging of virtualizing a network device at layer 2 that were described in the previous section, the following design goals have been extracted for VxSwitch:

- *Resource dimensioning:* A user leasing a virtual switch should be able to specify the number of ports, the capacity of each port and the buffer sizes.
- *Configurability of functionality:* Packet queuing and scheduling algorithms should be configurable on a per virtual switch basis, so that each user can adapt his virtual switch to the type of traffic it forwards.
- *Controlled sharing:* A switch must not affect any other in performance. Each switch should get the exact amount of resources it requires.

These design decisions require to virtualize the architecture of the switch, i.e., its hardware organization, as well as its operations. The first step towards such a new design is to chose an architecture, which is suitable for virtualization.

4.3.2 Overview of switch architectures

Different switch architectures have been proposed over time. In this section, we choose a design as a basis for a virtualized switch architecture.

4.3.2.1 From bus to crossbar architectures

In the past, switches happened to be built on a shared bus that interconnects input and output ports. However, this bus being shared by all ports is a bottleneck to switching power, as only two ports can communicate at a time. Hence, today, commonly-used switch architectures are shared-memory and more recently crossbar [Divakaran et al., 2009].

The shared-memory is known to have scalability issues challenged by the need to access the memory at a speed equivalent to the product of the line rate and the number of ports [Iyer and McKeown, 2001]. Overcoming these inhibitions, the crossbar architecture permits N pairs of I/O ports to communicate, with memories running at speeds independent of the number of ports.

The traditional crossbar switches deploy a centralized, complex matching algorithm to decide which among the contesting ports should communicate. All the selected ports transfer fixed-size packets at the same time, in a synchronous manner. This also requires segmentation of variable-size packets into fixed-size 'cells' at the input, and complementary reassembly at the output. These constraints are removed with the introduction of buffers at the crosspoints (CP), and leave the choice to perform cell- or packet-scheduling.

4.3.2.2 Buffered crossbar

On a buffered crossbar, the schedulers are distributed at each output port, and can operate independently to switch packets of variable size asynchronously. Having distributed schedulers is an important argument for choosing this architecture for virtualization. The complexity of scheduling algorithms in centralized crossbar architecture (with no buffering at crosspoints) is usually greater than $O(N^2)$ [Tarjan, 1983]. Virtualization of such an architecture brings up scalability issues as the number of virtual ports increases. On the other hand, the distributed schedulers in the Crosspoint-Queued (CQ) switch architecture pose no such problem.

Exploiting this technology, a CQ switch [Kanizo et al., 2009] has queues neither at the inputs nor at the outputs, but only at the crosspoints. Figure 4.1 represents such a CQ switch. Besides, when virtualizing this architecture, its simplicity removes the need to virtualize input/output queues. For more details on CQ switches, we refer the readers to [Kanizo et al., 2009]. Each crosspoint of the switching fabric implements a queue.

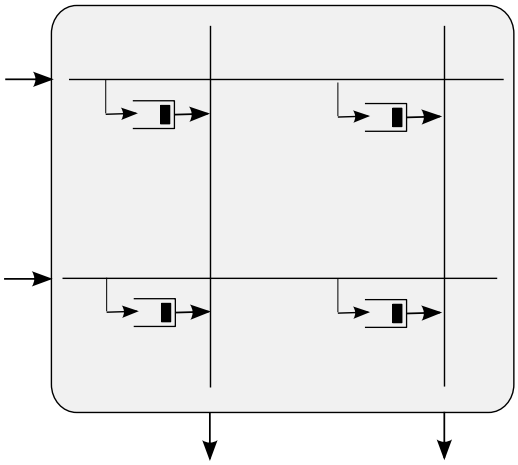


Figure 4.1: A Crosspoint-Queued switch.

Incoming packets are sent to the crosspoints according to their destination output port. A scheduler runs at each output port, selecting packets from one of the different crosspoints connected to that output. Once a packet is scheduled, it is sent out through the output line. In the whole process of crossing the switch, the packets go through a single buffer only.

4.3.3 Virtualizing a buffered crossbar

The resources to be shared in a switch are link capacity (in time) and buffer size (in space). As said before, each crosspoint of a CQ switch implements a queue. Virtualizing this architecture into V virtual switches (VSes), each crosspoint buffer used by these VSes is divided into V virtual buffers, each one implementing a virtual queue. If $b_{i,j}$ is the physical queue size at crosspoint (i, j), it is shared among the VSes, with each $VS^k, k \in [1, V]$ owning $b^k_{i,j}$ of the queue at crosspoint (i, j). We refer to the virtual buffers as 'virtual crosspoint buffers' (VXBs). Table 4.1 lists some of the commonly used notations in this chapter.

The sharing of input/output ports is similar. That is, the link capacity of a physical

Notation	Description
$1 \leq N^k \leq N$	Number of ports of VS^k
$\mathbf{C}^k(in) = [c_i^k(in)]$, $i = 1 \dots N^k$	Capacity vector for input ports of VS^k
$\mathbf{C}^k(out) = [c_i^k(out)]$, $i = 1 \dots N^k$	Capacity vector for output ports of VS^k
$\mathbf{F}^k = [f_{i,j}^k]_{N^k \times N^k}$	Matrix of forwarding paths of VS^k
$\mathbf{B}^k = [b_{i,j}^k]_{N^k \times N^k}$	Matrix of CP buffer sizes of VS^k
$\mathbf{Q}^k = [q_{i,j}^k]_{N^k \times N^k}$	Matrix of queue managers of VS^k
$\mathbf{S}^k = [s_i^k]$, $i = 1 \dots N^k$	Output scheduler vector for output ports of VS^k
$\mathbf{L}^k = [l_i^k]$, $i = 1 \dots N^k$	Lookup vector for input ports of VS^k
τ^k	Lifetime of VS^k

Table 4.1: Table of notations

port is shared among different VSes. Virtualizing ports gives rise to 'virtual ports', each receiving a part of the capacity of the physical port.
Virtual ports are dimensioned by user-defined input and output rates $c_i^k(in)$ and $c_i^k(out)$ for each port i. Moreover, as opposed to classical switches, in a VxSwitch, all inputs may not be connected to all outputs. As an example, a switch interconnecting a server with several clients may not want to provide connectivity between clients. To take this into account, forwarding paths are explicitly represented by a matrix $\mathbf{F}^k$. If there are V VSes all requiring a share of port i, then each virtual port of VS^k has a different capacity c_i^k. This sharing is performed by a scheduler whose service discipline is adjusted to satisfy the criteria of all VSes owners while optimizing the productivity of the physical host-switch. Details on scheduling are given in Section 4.3.4.1.

While combining different VSes on the same physical switch, the allocation of any VS^k has to satisfy respectively conditions 4.1 and 4.2 for each of its ports and each of its buffers:

$$c_i^k \leq c_i - \sum_{x=1, x \neq k}^{V} c_i^x, \forall i \in [1, N] \tag{4.1}$$

$$b_{i,j}^k \leq b_{i,j} - \sum_{x=1, x \neq k}^{V} b_{i,j}^x, \forall i, \forall j \in [1, N] \tag{4.2}$$

These constraints guarantee isolation, ensuring that each VS sharing the VxSwitch obtains the required amount of resources and predictable performance, and that VSes do not affect each others performance.

For functional customization, users can also configure the lookup mechanisms, $\mathbf{L}^k$ at the input ports, just like the schedulers $\mathbf{S}^k$ at the output ports and the queue managers QM^k at the crosspoints. Thus, each VS has control over the packets, once they enter the switch. This exposes dedicated lookup to upper-layer virtual routers, each of which may run a different protocol. Such a flexible lookup module can be implemented using for example a modular architecture like PLUG [De Carli et al., 2009]. A lifetime τ^k is associated with each VS.

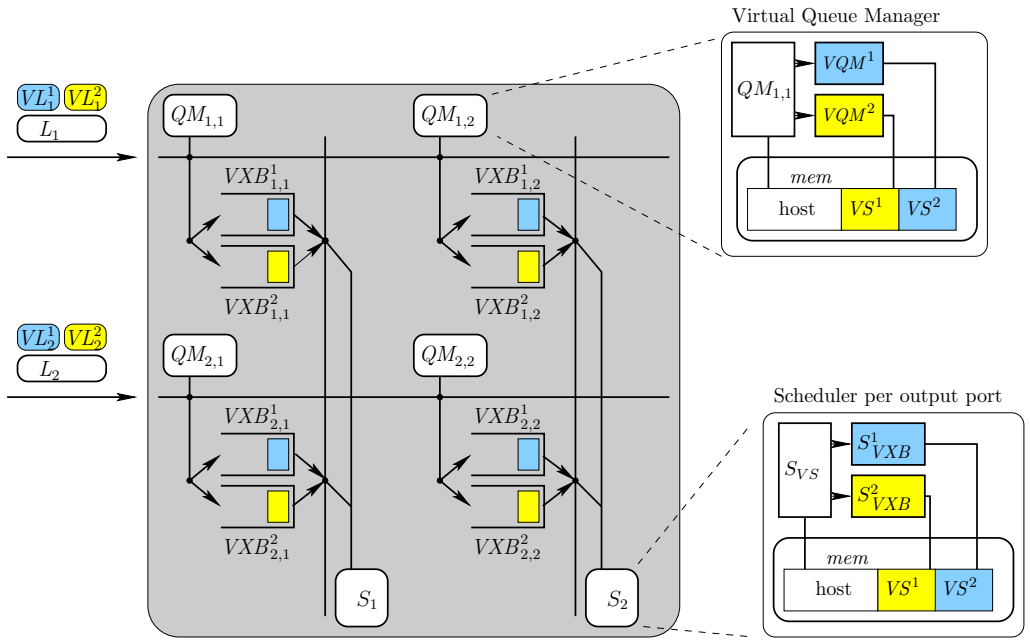


Figure 4.2: Architecture of VxSwitch's virtualized switching fabric.

Figure 4.2 depicts a VxSwitch based on a virtualized CQ architecture. In this simplified example, the switch hosts two VSes, each one using all the physical ports. Incoming packets are demultiplexed to the corresponding VS after a lookup action. This classification can depend on header fields, for example, the IP source/destination addresses, VLAN tag, ToS, or an additional shim header like in MPLS. One could also imagine using a controller such as OpenFlow [McKeown et al., 2008] to dynamically change and identify virtual networks. After the lookup, a packet is sent to the corresponding VXB if it has enough space left. Scheduling takes place at each output port. A scheduler per physical output port chooses among the virtual switches at each time-slot, then another scheduler, inside the chosen VS and the output port, selects a VXB to be dequeued.

The mechanism of classification and queuing to different virtual buffers is derived from QoS management in routers, where flows are grouped into service classes and memory is partitioned into logical queues. While, in classical QoS implementations, shaping and policing are usually performed on an output port with a set of virtual buffers, the virtualized crosspoint-queue switch implements shaping in a decentralized manner scheduling several crosspoint queues per output port.

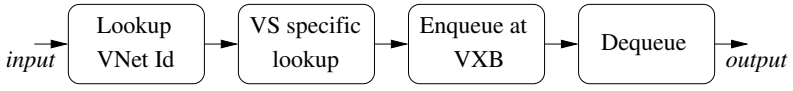


Figure 4.3: Successive actions performed on a packet traveling throughout VxSwitch.

Figure 4.3 depicts the different operations a packet undergoes while traveling throughout a VxSwitch. It enters the line card, where a two-level lookup is performed. The first lookup figures out to which virtual network (VNet), and thus VS, the packet belongs.

The second lookup is specific to the VS. According to the result, the linecard sends the packet to the selected VXB, from where it is dequeued once it is at the head and has been selected by a scheduler.

4.3.4 Resource sharing and configurability

The resources of the virtualized switch are shared in time and space. Sharing in time is assured by scheduling algorithms determining at which moment each virtual buffer is dequeued. Sharing in space takes place in the buffers which are split into multiple virtual queues. Each crosspoint buffer is shared into VXBs, one per VS using this crosspoint. Implementing queue management and packet scheduling on the crossbar requires additional memory, but this should not be the bottleneck, especially with the increasing density of VSLI. It is rather the number of pins, whose number is constrained, which makes virtualization of a crossbar even more appealing [Mhamdi et al., 2006].

4.3.4.1 Virtualized scheduling

In the example of Figure 4.2, the virtualized switch has two input ports and two output ports, and four crosspoints, used by two VSes. Incoming packets at each port are classified by the responsible queue manager (QM) to the corresponding VXB. For a given output port i, a scheduler (S_i) decides from which crosspoint and from which VXB the next packet will be dequeued. For isolation and customization of VS scheduling, there are two levels of scheduling. An *Inter-VS* scheduler selects among the VSes, which one to select. It is responsible of performance isolation between the different VSes. In addition, each VS has an *Intra-VS* scheduler, which selects the VXB from which to dequeue. Both, the inter- and the intra-VS schedulers can operate using different policies. The choice of the policy will depend on the type of each hosted VS, and on the switch maintainer's interest in maximizing the utilization.

As an example, a switch provider hosting several VSes, each one switching bulk traffic that needs high throughput, and having no strict constraints on delay and burstiness, can use a variable sharing policy. This means, it allows VSes to temporarily exceed their bandwidth, and risks in turn to get smaller bandwidth at another moment while other VSes exceed their bandwidth. For the switch providers, this brings the advantage of maximizing their equipment use, and for the VS operators, of not paying for peak rate guarantees if it is not strictly required for them.
In another scenario, one of the hosted VSes might have delay-sensitive traffic and require strict minimum bandwidth guarantees. Hosting such a VS would imply for the VxSwitch to use another intra-VS scheduling model in order to satisfy the VS's delay constraints and avoid impact from other VSes on the capacity.
Inside a VS, an operator can also choose different strategies, e.g., Round-Robin (RR), Longest Queue First (LQF), or First Come, First Served (FCFS), depending on the requirements in terms of delay, jitter, buffer-size, or scheduling complexity. In an operational network, the choice of the scheduling and sharing strategies comes with a business and pricing model which are not developed here. It focuses primarily on the architecture allowing this flexibility in scheduling.

The internal architecture of this two-level scheduling module within the crossbar is shown in Figure 4.2 for scheduler S_2. The inter-VS scheduler S_{VS} shares capacity between the different VSes. At each scheduling decision, it selects which VS is scheduled and for

how long, or for how many packets or bytes. Each VS has an internal VXB scheduling module S^k_{VXB} responsible for this, which is called by the inter-VS scheduler. Inter-VS and intra-VS schedulers logically share the scheduler's memory to register state information. This requires V times more memory for the scheduler, when the switch hosts V VSes. When a VS is scheduled, it acts like a physical switch, but chooses only from its own VXBs, and dequeues only during the time period it has been attributed. Each VS having at maximum V VXBs, each intra-VS scheduler has a complexity of $O(V)$. As only one intra-VS scheduler operates at a time, the overall scheduling complexity is also of $O(V)$.

4.3.4.2 Virtualized buffer management

While scheduling aims at guaranteeing a specific rate for the 'virtual link', the 'virtual buffer' size is defined by the different possible buffer-sharing policies.

The possibility to change memory and queue sizes on crossbars [Mhamdi et al., 2006] enables VXBs to be flexibly dimensioned. Each such VXB is accessible only by packets of the VS to which it belongs. Moreover, a virtual queue manager (VQM) for each VXB can be configured, to perform specific queue management, deciding on where to enqueue a packet and from where to drop a packet, or performing active queue management. A crosspoint queue manager (QM), selects for an incoming packet which VQM is responsible for enqueuing it. Its internal architecture is shown in Figure 4.2 for $QM_{1,2}$.

The physical crosspoint queue can be shared among different VSes in a strict way or in a flexible way. If the sharing is strict, the VXBs have each a fixed size. Such a setting may lead to an inefficient use of buffers, as packets belonging to a VS are dropped when the corresponding VXB is full, even though there is space for it in the physical crosspoint queue.

This is where another mode of buffer sharing is helpful. In the flexible buffer sharing policy, a VXB can be resized provided that the other VXBs sharing the crosspoint allow such resizing. For example, for increasing a VXB of n Bytes, there must be n Bytes of free space in the concurrent VXBs. Such a dynamic buffer sharing reduces the number of losses (compared to the strict buffer-sharing policy). This allows a number of dynamic buffer-sharing policies—every VS^k can decide to set aside a fraction of its queue size, say $F^k_{i,j}$, that can be attributed to other VSes that share the crosspoint (i, j). The order in which packets are dequeued can also be an interesting decision criteria. A VS operator would like to give priority to its packets before dequeuing $F^k_{i,j}$.

4.4 Simulations

For evaluating the described virtualized switch architecture and its scheduling paradigms, we developed a simulator that runs multiple VSes in parallel, simulating the sharing of common physical resources. It is used to evaluated the performance of VxSwitch, and explore the individual configurability of virtual switches.

Virtualization, and hence the sharing of the resources such as buffer and capacity, introduce new challenges to a switch. For example, several small buffers may accept less packets than one big buffer. In addition, scheduling needs to take into account and ensure the capacities of each of the virtual switches, while enabling each virtual switch to execute its own scheduling algorithm. This simulation sheds light into these issues and the resulting performance impact, compared to a non virtualized buffered crossbar.

The first objective of this study is to investigate the relative the loss of (physical) switch performance due to the resource sharing. Another objective is to evaluate according to VS efficiency. The different resource-sharing policies that we discussed earlier will lead to different performance levels. With an example of strict sharing, compared to priority scheduling, we intend to quantify these differences here.

4.4.1 Virtual switch simulator

For simulating the behavior of VxSwitch, we developed a simulator. It simulates a switch hosting a variable number of virtual switches with different characteristics. It simulates variable sized packet arrival on the different ports of the switch and the sharing of the physical resources. It is composed of a traffic generator module, a hosting switch module and several virtual switch (VS) modules.

The traffic generator module generates an input traffic matrix. The main component, the VS hosting module, inserts packets into the VXBs from the input traffic matrix. Different instances of a VS module in different configurations, scheduled by the hosting module, dequeue the packets. Figure 4.4 illustrates the software architecture of the simulator. The

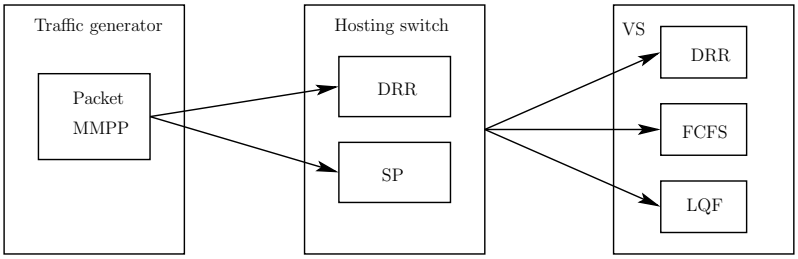


Figure 4.4: Software architecture of the VxSwitch simulator

modules composing it are described in the following.

4.4.1.1 Traffic generator

This module generates input traffic to the simulator. This traffic consists in bursty network traffic, as described in Section 4.4.2.1, using a Markov Modulated Poisson Process (MMPP, refer Figure 4.5). The arrivals are in packets taken from bimodal uniform distribution with the first mode ranging from 64 to 100 bytes and the second mode ranging from 1350 to 1500 bytes. The traffic load on each input port can be set to a value between 0 and 1. For each VS, the proportion of traffic on a port belonging to this VS can be configured as a weight.

4.4.1.2 Hosting switch module

The role of the hosting switch module is to share the resources between the VSes in a controlled way: each VS needs to get the amount of capacity and buffer it requires. The module enqueues the packets from the input traffic matrix to the appropriate VSes and selects the VSes from which to dequeue packets according to a scheduling policy that determines the type of sharing. Selecting a VS means calling the corresponding VS module. The host switch has presently two scheduling modes, deficit round-robin (DRR) and strict priority (SP), which are detailed further.

- DRR hosting switch. The DRR hosting switch schedules the different VSes using deficit round-robin [Shreedhar and Varghese, 1995], according to a vector of weights $< \omega_1, \omega_2, ..., \omega_V >$ of size V, the number of VSes. Each VS_i gets a quantum of ω_i bytes at the beginning of a scheduling round, if it has at least one packet to dequeue. It can then dequeue until finishing its quantum or until it has no more packet to dequeue, then moves to VS_{i+1} and calls its dequeuing module for ω_{i+1} bytes. This enforces the sharing of the capacity according to the weights, and guarantees minimum capacity and isolation. Yet, for providing absolute performance and limiting the maximum capacity, a slight modification is needed in the algorithm to limit the number of bytes dequeued over time.

- SP hosting switch. The SP module enables to give strict priority to a VSes. It schedules the different VSes according to their priorities. These are specified by a vector of priorities $< p_1, p_2, ..., p_V >$ of size V, where $p_i = 0$ is the highest priority. If several VSes have the same priority, these are scheduled in round robin. If for two VSes, VS_i and VS_j, $p_i > p_j$, then VS_i is only scheduled when VS_j has no cell to dequeue, as VS_j has strict priority over VS_i.

Either of these modules (DRR or SP) account the number of transmitted and sent packets, and the delay, respectively of each VS. Throughput is computed as the amount of bytes transmitted to the amount of bytes arrived at the input ports. Inversely, loss corresponds to the number of packets (respectively bytes) lost to the number of packets (respectively bytes) arrived. Delay corresponds to the average time a packet spends in the switch, between its arrival instant and its dequeuing instant.

4.4.1.3 Virtual switche module

Each time a VS is scheduled by the hosting switch, it has the control over the physical output port for a certain duration and can dequeue packets. VSes can have different packet scheduling disciplines, some of which are listed below:

- DRR VS. This VS schedules VXBs using deficit round-robin. When it is interrupted by the host switch, because it is the turn of another VS, it saves the identifier of the last dequeued VXB to start the dequeuing process at the next VXB, the next time it will be scheduled.

- FCFS VS. The FCFS (First Come, First Serve) VS, each time it is scheduled, browses all the queues that hold at least one packet, and dequeues a packet from the queue with the oldest head of queue packet.

- LQF VS. The LQF (Longuest Queue First) VS dequeues always the VXB with the biggest amout of data. It has a memory to register from which queue it dequeued last time. Thus, if at successive scheduling periods, several queues hold the same number of bytes, it will not dequeue twice the same queue.

4.4.1.4 Extensibility

The actual version of the simulator offers a set of scheduling policies, which appeared to be the most relevant to us. But the simulator's modularity allows to easily extend it with other policies. The hosting switch module's scheduling can be programmed independently

of the VSes, and new scheduling policies can easily be programmed independently of the VS modules. Moreover, a different module exists for each VS type (in terms of scheduling). Thus, the simulator can easily be extended with new VS types by adding new VS modules. Any of the VS modules can be combined to simulate a switch hosting different types of VSes simultaneously and observe the performance due to traffic arrival and scheduling. In the same way, other input traffic generator modules can be added to the simulator.

4.4.2 Experiments

This section gives simulation results on the overall performance of a virtualized switch in terms of throughput, as well as the individual VS performance, in case of heterogeneous VSes sharing the same VxSwitch. The different simulations have two main goals:

- Determine the consequence of sharing switching resources into fixed sized virtual resources, instead of sharing them in an uncontrolled way between all traffic in a physical switch;
- Experiment with the configurability of VxSwitch and show how customized scheduler and buffer configurations can satisfy individual requirements of different virtual switches, inside a single VxSwitch.

4.4.2.1 Metrics

The metrics we consider are throughput, loss, and delay. The throughput of a switch is defined as the percentage of arrived packets that departed the switch. In a virtualized switch, to calculate the total throughput, we use the sum of the arrived packets which were admitted in any of the VSes. The loss rate is complementary to the throughput, being the percentage of arrived packets which are dropped due to queue overflow. The delay is the time spent by a packet in the switch. The scalability of the virtualized switch is analyzed by varying the number of VSes it hosts for different buffer sizes.

The generated input traffic emulates bursty network traffic. A Markov Modulated Poisson Process (MMPP, refer Figure 4.5) is used for the arrival process. The arrivals are in packets, with sizes taken from bimodal uniform distribution, where the first mode varied from 64 to 100 bytes and the second mode varied from 1350 to 1500 bytes.

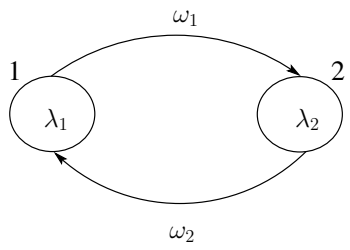


Figure 4.5: MMPP model: λ_1 and λ_2 are the arrival rates; ω_1 and ω_2 are the transition rates

The simulations are executed using the described input traffic scenario. Each experiment is run tun times. The average values are presented. The physical crosspoint buffer (CB) sizes vary as multiples of 1500 bytes, since it is the minimum size necessary to hold one maximum-size packet. The simulated physical switch has $N = 32$ ports. The input

traffic is generated at each physical input port with a load of $\rho = 1/N$, in order to charge the output port by a load of 1. It is then split into the different VSes according to weights. Similarly, we assume that the traffic at each input port is uniformly distributed to the output ports. Thus, for simplicity, we explain the details with respect to a single output port.

4.4.2.2 VxSwitch overall performance

The first experiment evaluates a virtualized switch hosting homogeneous VSes (they all have the same VXB size and the same virtual port capacities). Input traffic for each VS arrives with a load of $1/(N * V)$ on each port, where N is the number of ports and V the number of VSes, to have a total load of 1 on the output port. VSes are served using a deficit round-robin (DRR) [Shreedhar and Varghese, 1995] scheduler to ensure the fair sharing of output link rate, serving each VS during at least $1/V$ of a scheduling round, as long as it has packets to dequeue. All the VSes have the same quantum $q = 1500$. Each VS has a deficit counter (DC), which gets increased by one quantum each time it is scheduled and has packets to dequeue. It is allowed to dequeue at maximum DC unities of data before the next VS gets scheduled. This small quantum size, allowing to dequeue only a single 1500-byte packet, minimizes rate variation and delay in the VSes and maximizes isolation between them. Inside each VS, a separate instance of DRR is used to dequeue packets in a fair way from the different VXBs. The quantum of each VXB is also chosen equal to 1500 bytes. Note that, independently of the host switch's VS scheduling, a VS could also use another VXB scheduling algorithm like FCFS, LQF, etc. But for homogeneity, each VS here uses the same algorithm with the same parameters.

In the different runs, the number of VSes as well as the size of the crosspoint buffers (CBs) are varied to evaluate the scalability of the switch, and the impact of buffer size on the performance. Figure 4.6 shows the throughput of the virtualized switch hosting 2 to 32 VSes and having crosspoint buffer sizes between 3 and 192 kB. The result numbered '1' on the x axis corresponds to the performance of the physical switch. Per VS throughput is not represented here, as it is equivalent to the total throughput of all VSes. Mean deviation is of negligible significance in these results.

In this configuration, with up to 32 VSes, the total throughput of the physical switch is always above 90%, given a VXB size (CB/V) of at least 1500 bytes. The throughput is highly dependent on the CB size. By segmenting the buffers into fixed-size VXBs, the usable buffer capacity shrinks by a fraction, and this inefficiency increases with the number of VXBs. Thus, enabling the hosting of several VSes with isolated VXBs requires the host switch to scale in buffer size, at least to give each VXB the size of one maximum size packet. A CB size of 96 KB allows over 99% throughput for up to 32 VSes. For a crossbar with 32 1 Gb/s-ports, this would require a total of about 100 MB of buffer.

Figure 4.7 shows the average delay per VS on one VxSwitch, for different numbers of VSes sharing the VxSwitch, and different VXB sizes. When the buffers are too small, i.e., the configurations where the throughput is significantly below 100% (see corresponding results in Figure 4.6), the delay increases with an increasing number of VSes. Indeed, if all VS's VXBs are full, a VS needs to wait longer for its turn to come with DRR scheduling and its packets experience more delay.
When the buffers are big enough, or bigger than necessary (the cases where close to 0% packet loss occurs), delay does not increase with the number of VSes. This is for example

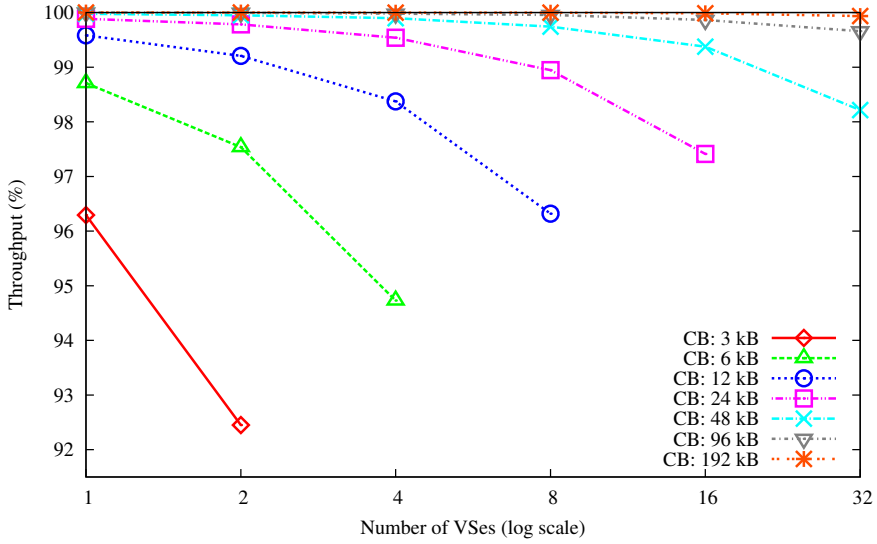


Figure 4.6: Throughput of a VxSwitch with different CB sizes, hosting different numbers of VSes.

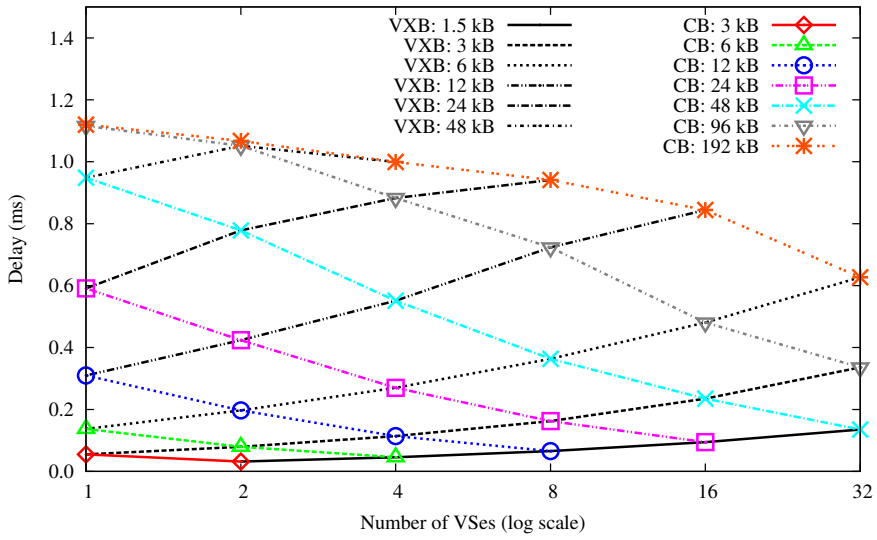


Figure 4.7: Delay on a VxSwitche with different VXB sizes.

the case for a VXB size of 48 kB, where increasing from 2 to 4 VSes does not increase the delay. This is probably related to the fact that each of the 4 VXBs at each crosspoint contains on average only half of the number of packets than each of the 2 VXB in the case of 2 VSes, as rate is divided by two.

In conclusion, with this type of bursty packet traffic, sharing a buffer impacts on the performance as more losses occur. On the other hand, delay increases naturally with the number of VSes, as one VS must wait for others to complete their jobs. In this example, delay increases only when packet loss occurs, i.e., the buffers are not big enough. But in general, buffer size, and hence delay will depend on the type of traffic, the burst sizes, and the rate.

In general, inefficiency in sharing can be decreased by using variable buffer sharing, and also by more traffic-aware scheduling. In this example of homogeneous sharing, VSes are served in a fixed order. Hence, a VS might be scheduled at a moment where it has few packets to dequeue, thus blocking other virtual switches with full queues. This might be particularly penalizing for 'small' VSes using only a small part of the resources and being scheduled after very big intervals, thus suffering from high delays in round-robin.

4.4.2.3 Sharing resources differently

In this section, the advantages of configurability are evaluated. A VxSwitch is shared differently, by two VSes where each has other performance requirements. The experiments show how the adaptation of the scheduling of the VSes can improve the performance of each of them. VS^1 requires little bandwidth—10% of the available capacity—on each port, and minimal delay. VS^2 needs high bandwidth—90% of the available capacity—on each port, and has no constraints on delay. In this scenario, DRR scheduling like before would cause high delays to VS^1, giving it only one quantum in 10, to dequeue a packet. As an alternative, strict-priority (SP) scheduling is implemented in the physical switch, always giving VS^1 priority over VS^2. Hence, VS^1 can dequeue packets when they arrive, thus momentarily exceeding the 10% of reserved average rate. The resulting variable-sharing scheduling policy is compared to the previous fair sharing with DRR. Inside each VS, the same scheduler (DRR) is used, to focus only on the impact brought by different inter-VS scheduling strategies.

In both cases, the throughput and delay are evaluated as a function of different allocations of 15-kB CBs among the two VXBs. Figures 4.8 and 4.9 show, respectively, the average throughput and the delay obtained on each VS, in the ten runs. Giving priority to VS^1 increases the loss on VS^2 by less than 1%, compared to DRR, while throughput of VS^1 is highly increased. It reaches 100% for a VXB size starting from 3 kB, while with DRR, VS^1 obtains 99% of throughput starting only from a buffer size of 6 kB, due to increased latency, as VS^1 is scheduled only once in 10 times.

Regarding throughput, giving priority to a low-bandwidth VS improves, on the average, the individual throughput of both VSes, even if the total physical switch throughput decreases very slightly (by less than 1%). Regarding the delay, the improvement using SP is even more important. Figure 4.9 shows that VS^1, using SP scheduling, decreases its delay by 0.1 to nearly 0.6 ms. Meanwhile, the delay of VS^2 is increased by less then 0.2 ms in the worst case of having only a 1500-bytes VXB. Also, standard deviation is depicted in both scheduling scenarios. Giving VS^1 priority over VS^2 allows it to achieve an almost constant delay with very slight variation, while with DRR, it varies by up to 0.15 ms around the average. The delay deviation of VS^2 is negligible and similar in both scenarios, varying around less then 0.05 ms. For VS-operators, in this configuration, priority scheduling sounds to be the better solution, at the cost of a very small decrease in the overall throughput.

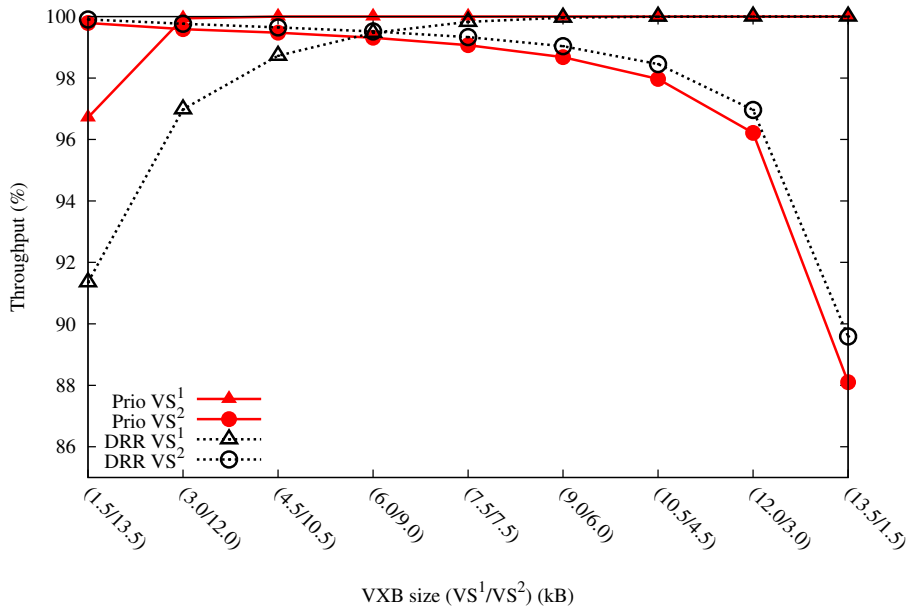


Figure 4.8: Throughput of a VxSwitch hosting two VSes, using respectively 10% and 90% of the capacity.

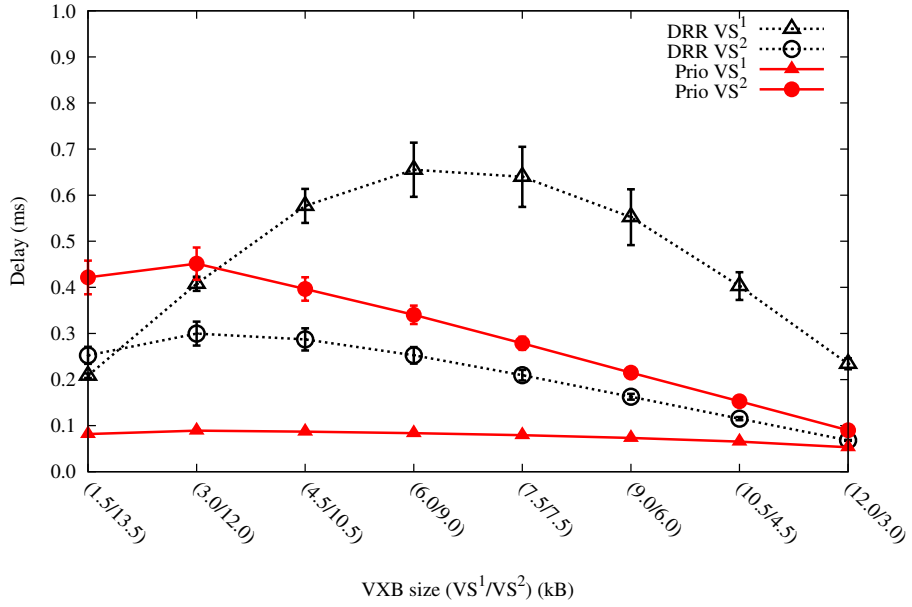


Figure 4.9: Delay of two VSes hosted on a VxSwitch and using respectively 10% and 90% of the capacity.

These experiments give an insight on the variation of VS performance, depending on the type of resource sharing that is employed. Further investigation on this virtualized architecture is necessary to choose the best sharing strategies.

4.5 Application

The flexibility introduced to the network by the ability to flexibly dimension and configure virtual routers and switches brings new use cases and allows to implement features that remained concepts until now. In this section, we show a particular application of VxSwitch enabling transparent virtual path-splitting.

4.5.1 Virtual network context

Virtualization has been proposed to decouple virtual network operators from network infrastructure providers [Feamster et al., 2007]. In such an application, virtual service operators can exploit virtual networks and configure them independently from each other. For realizing this, we imagine that the different virtual networks are hosted on a physical network infrastructure or federation of infrastructures, and are managed through a management plane, like suggested, e.g., in [Greenberg et al., 2005]. Figure 4.10 illustrates such a scenario. The management plane exposes the configuration facility to administrators

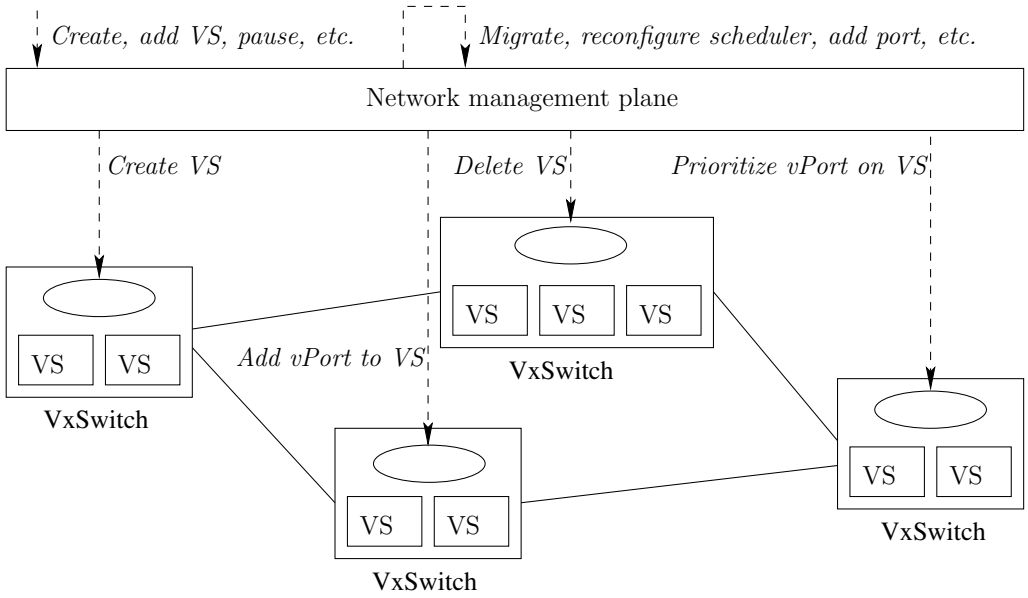


Figure 4.10: Management of a network of VxSwitches.

to deploy, reconfigure, save or release virtual networks. When an administrator issues a virtual-network configuration request to the network-management plane, the latter translates it into operations to execute on the different VSes. The management plane can also decide by itself to reconfigure a virtual network, transparently to the users, e.g., migrate a running VS for load balancing. Thanks to layer 2 virtualization in VxSwitches and the resulting isolation and customization flexibility of VSes, virtual networks can be configured through elementary operations on VSes such as adding or deleting virtual ports, reconfiguring their capacities, adding or deleting buffers and reconfiguring their size, as well as choosing scheduling and lookup mechanisms at any time. The synchronized execution of such operations on the different VSes composing a virtual network can result in precise virtual-network configuration, fitting to specific use cases, some of which are described below.

4.5.2 Use case: Paths splitting

A new problem that arose with network virtualization consists in how to map virtual networks on a physical substrate, satisfying all constraints such as processing and memory requirements in nodes and bandwidth on links. One interesting solution has been proposed, that decomposes virtual links, and allocates them on parallel physical paths, to overcome the hardware limitations in terms of capacity [Yu et al., 2008]. The practical issue of this solution is that such a decomposed virtual link may deliver packets in disorder, due to different latencies on the different physical paths. This problem is mostly solved at the transport layer, e.g., using multi-path TCP, a version of TCP that is designed to deal with transporting data over several parallel paths [Barré et al., 2010] [Ford et al., 2011]. In the context of virtual networks, path splitting has also been proposed as transport virtualization (TV) [Zinner et al., 2010]. In the following, we show how the configurability of VxSwitch can remedy these issues.

4.5.2.1 Concept

Figure 4.11 illustrates the concept of virtual path splitting, representing a virtual link, we call *VLink*, allocated on two parallel physical paths. VLink represents the virtual link

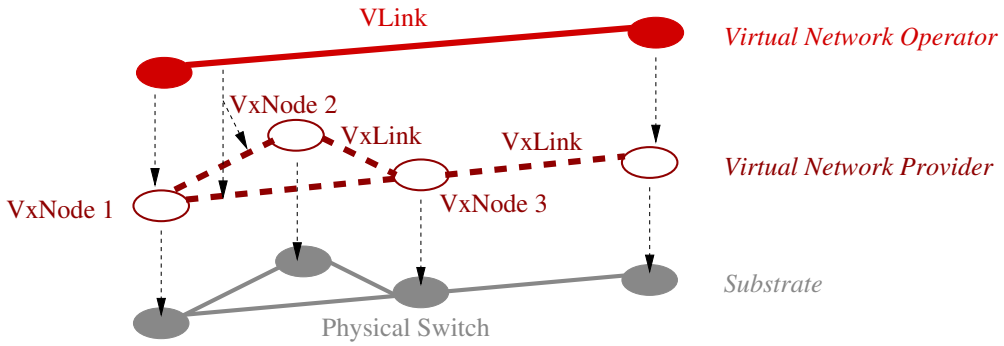


Figure 4.11: Layers of virtualization required to efficiently provide a virtual link on top of a physical substrate. VLink is split between two physical paths. The network provider configures a set of VxLinks and VxNodes to implement the VLink splitting and to provide the QoS requirements.

seen from the perspective of virtual network operator. They are not aware of the physical allocation, and instead, see the virtual link like if it was a physical link, configured with an SLA, specifying for example guaranteed capacity and latency. The bottom layer represents the physical network where the VLink is mapped to. The intermediate layer represents the network as it is seen by the virtual network provider, who is in charge of supplying the VLink with the requested SLA to the virtual network operator. It is composed of partitions of all physical resources involved in the implementation of the VLink. We refer to these resource partitions as VxLinks (partitions of physical links) and VxNodes (virtual routers or switches). In this example, the VLink is split over two physical paths, allocated between the VxNodes 1 and 3. In order to fulfill the SLA requested by the virtual network operator on the VLink, VxLinks have to be provisioned with specific bandwidth, while VxNodes have to be provisioned with specific buffer sizes and scheduling algorithms.

Such a virtual link embedding solution may lead to performance drawbacks, due to the possible disorder of packets, and difference of latency on the parallel physical paths. This will have an important impact, especially on TCP flows that are sensitive to loss and latency. That is, where VxSwitch can come as a solution, to configure the switching plane of the different VxNodes underneath the VLink, in order to deal with packet disorder. In fact, it allows to flexibly configure buffer sizes, one main aspect to performance and to recover from packet disorder. The following section describes the possible configuration and illustrates it with simulations.

4.5.3 Implementation and simulations using VxSwitch

In this section, we use the simulator to implement the previously described use case and play with the configurability of VxSwitches, to see how they can realize a split virtual link allocation, depending on the physical network.

4.5.3.1 Simulation setup

Two VxSwitches, 1 and 2, were created using the simulator, each hosting one VS, respectively VS^1 and VS^2, as represented in Figure 4.12. Each VxSwitch x has two input ports, $p^{x,1}$ and $p^{x,2}$, and two output ports $p^{x,3}$ and $p^{x,4}$. They are interconnected by two physical paths, L_1 and L_2. VS^1 and VS^2 are interconnected over a virtual link that is split over

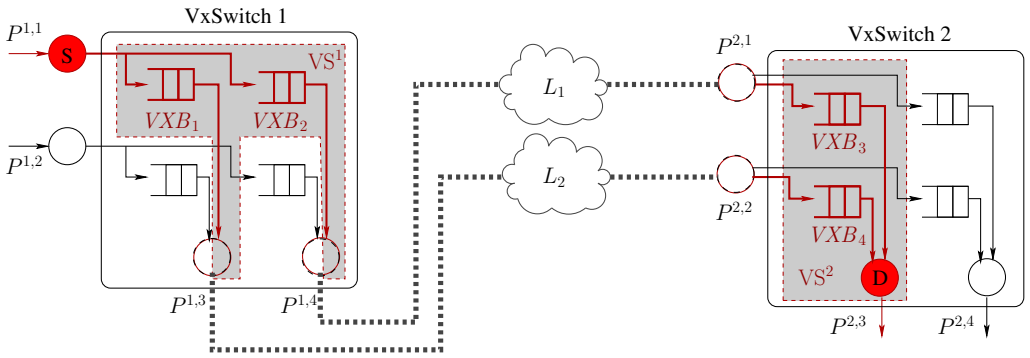


Figure 4.12: Simulation architecture: Two VSes (represented in red), allocated on two VxSwitches, interconnected over two paths over a physical network.

the two physical paths, L_1 and L_2, allocating half of the capacity of each. The traffic on input port $p^{1,1}$ of VS^1 is equally split by the classifier S to the two output ports, $p^{1,3}$ and $p^{1,4}$. For assuring this task, we added a classifier module to the VxSwitch simulator. This module, besides splitting the traffic, assignes a sequence number to each packet. On VS^2, the traffic of the two paths, L_1 and L_2 is re-aggregated by the output scheduler D on port $p^{2,3}$.

Each VS has two dedicated VXBs, at the interconnection points of their input and output ports. The output schedulers of VxSwitch 1 are configured with Deficit Round Robin (DRR), attributing 50% of the processing on each output port to each, VS^1 and the cross-traffic, for dequeuing packets. The output scheduler D of VS^2 on $p^{2,3}$ is configured so as to dequeue packets according to their order of arrival at VS^1, to hide the impact of splitting. For this, we implemented an additional sub-module in the hosting switch module of the simulator that dequeues packets sequentially according to their sequence numbers.

Capacities are allocated for the VSes as follows: they use 100% of the capacity on input $p^{1,1}$ and output $p^{2,3}$, and 50% of each of the two paths L_1 and L_2. The other resources of the two VxSwitches are used by cross traffic in order to use 100% of its physical capacity.

The simulator operates on variable-size packets. The input traffic was generated with a Markov Modulated Poisson arrival process, and packet sizes were taken from a bimodal uniform distribution, where one mode varied from 64 to 100 bytes, the other from 1350 to 1500 bytes, like for the previous simulations. The input rate at each port is of 1 Gb/s.

Sizing VXBs to absorb VS internal latency differences. In the first simulation, we consider paths L_1 and L_2 having the same latency. Hence packet reordering at D in VS^2 may be required only due to disorder introduced by VS^1's internal scheduling.

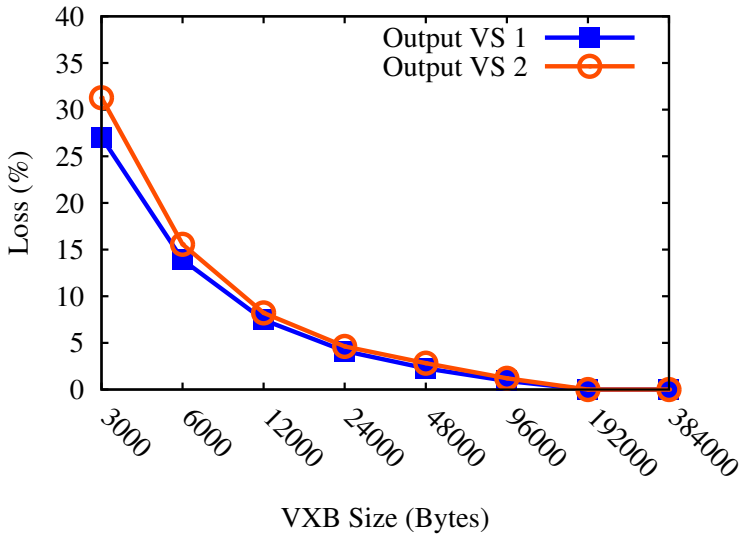


Figure 4.13: Loss on a path split into two paths with equal latency and equal capacity.

Figure 4.13 shows that while varying the VXB sizes of the two VSes, it can be observed that the ratio of lost packets on the two output ports of VS^1 and the ratio of lost packets on output $p^{2,3}$ of VS^2 (to the number of arrived packets at $p^{1,1}$ of VS^1), both reach zero for a VXB size greater or equal to 192000 Bytes.
This means that the reordering scheduler does not need additional buffer, when there is no difference in latency on the two physical paths.

Sizing VXBs to absorb latency on the path. In this experiment, constant additional latency is introduced to path L_2, simulating that the path contains for example more hops then L_1. In order to avoid buffer overflow on VXB_3, additional buffer needs also to be added. The results in Figure 4.14 show how adjusting VXBs according to the additional latency (in this case of a single flow, using the bandwidth-delay-product), allows to obtain zero loss, while packets are delivered in order by the scheduler D. Depending on the traffic pattern and the splitting ratio and the paths selection for virtual links, the buffer sizes must be adjusted differently. For this, analytical models such as proposed in [Zinner et al.,

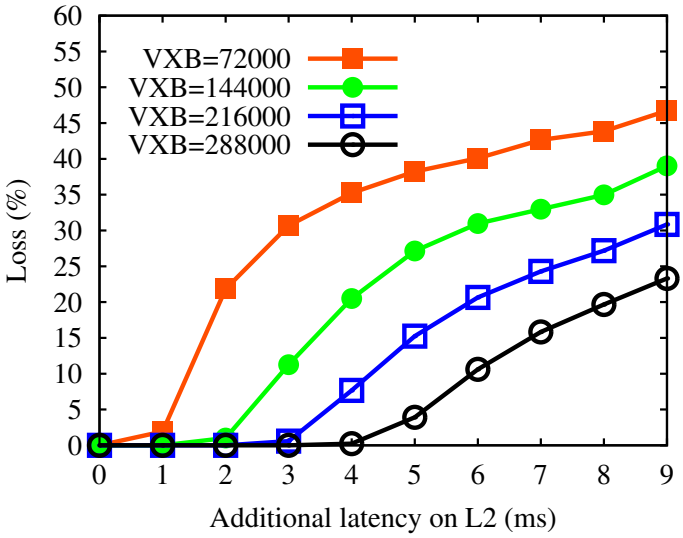


Figure 4.14: Loss due to different latencies on a split path. All VXBs are sized to 192000 Bytes, apart VXB_3 whose size is increased by different values starting from 72000 up to 288000 Bytes.

2010] could be used, to find the most efficient VxSwitch buffer configurations.

These simulation results gave an example of how the flexible buffer and scheduler configuration of VxSwitch can solve issues in virtual networks, directly on the data plane. In the given example, the notable benefit is that a particular embedding, mapping one virtual link to parallel physical paths, can be configured directly on layer 2 using VxSwitch, in a way completely transparent to a virtual network service provider and to upper layers, especially the transport layer, which will perceive the required capacity without loss and packed disorder.

4.6 Conclusion

This chapter proposed a design of a virtualized switch, VxSwitch, that allows dynamic creation of virtual switches, with customized buffer sizes, port capacities, scheduling and queue management algorithms. As for its de-materialized aspect, and its flexibility in configuration, VxSwitch is aligned with the virtual network concept, and the idea of networks as a service. Here, the first design of VxSwitch was discussed and simulated. To implement and integrate VxSwitch to production networks, there are multiple challenges and new research directions to take.

The example of VxSwitch is based on the crosspoint-queued (CQ) switch architecture [Kanizo et al., 2009] for its simple and efficient design, which is appealing for virtualization. Currently, the biggest drawback of a crosspoint-only-buffered switch is the size of the crossbar chip which is not capable of integrating big buffer sizes for high port densities. However, the number of virtual switches that can be hosted on a virtualized CQ switch depends on the features defining different virtual switches. In addition, SRAM density

is doubling each year. Moreover, one can imagine other switch designs for building a virtualized fabric, obtaining the same facility of deep slicing; an example is the CQ switch with input buffers.

To optimize resource utilization and give virtual switches performance guarantees, there are a variety of possibilities. While strict sharing gives the best guarantees, it may lead to a waste of resources. To solve this issue, the best balance of performance guarantees of virtual switches, resource utilization and pricing has to be found.

Security is another problem raised when sharing the switching hardware and handing over the fabric's configuration to users via a management plane. Access to virtual switches needs to be strictly controlled, and code and data of different virtual switches need to be isolated from each other. Also, the customization of a VxSwitch should provide maximum flexibility while limiting the configuration complexity.

Yet another possible direction is to explore how a management plane can cope with heterogeneity in network when VxSwitches are interconnected with other types of equipments.

For integration with existing solutions, an interesting approach would be to implement an OpenFlow module in VxSwitch, and extend the OpenFlow API to add flows directly to virtual crosspoint buffers and virtual input/output ports to enable per-flow QoS in virtual switches. This is one of our plans for future work, after implementing VxSwitch on FPGA; to evaluate and employ it in virtual network infrastructure deployment, and explore which new functionality could be integrated in VxSwitch.

In summary, when sharing the resources a layer 2, virtualized networks can become reality, different entities can rent virtual networks, customize them, and use them in private environments where everyone can develop new features. The network could evolve in a secure way, opening up a huge number of research directions to explore.

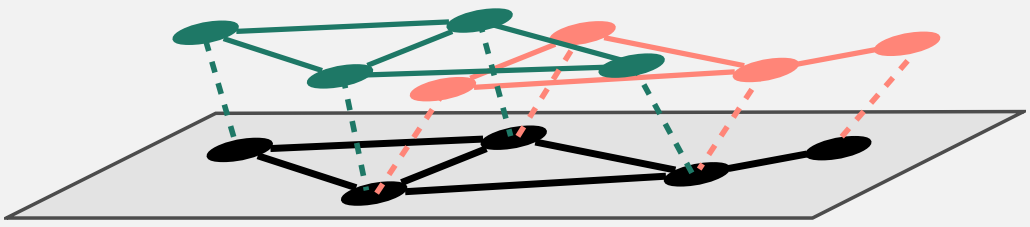
Our principal motivation for virtualizing the network at layer 2 is the need for *isolation* between virtual resources partitions and operations. One consequence of the isolation is *deterministic performance*. In the next chapter, we investigate how isolation and performance guarantees can be provided using current virtualization techniques, in order to offer virtual networks as a service.

5 Isolating and programming virtual networks

The work presented in this chapter has been published at the International Journal of Network Management - special issue on Network Virtualization and its Management (IJNM 2010) [1]. Modules developed in the context of this chapter are part of a deposited software [HIPerNet].

Abstract. When allocating and interconnecting virtual routers on top of a physical infrastructure to build a virtual network, resource sharing has a major impact on the service delivered by the virtual network. In this chapter, we propose

- A link sharing mechanism to provide differentiated bandwidth services to virtual networks, with guarantees;
- A virtual routing service that offers customizable routing per virtual network; and
- An evaluation of the virtual link bandwidth service with software virtual routers and OpenFlow switches.



5.1 Introduction

After having extensively analyzed the performance of software virtual routers in Chapter 3, and demonstrated a possible hardware design for virtualizing switches and share their resources in Chapter 4, this Chapter proposes a virtual network service based on controlled resource sharing.

As discussed earlier, a virtual network is a set of virtual resources such as virtual routers, virtual switches *and virtual links* created atop a physical infrastructure to interconnect virtual nodes, e.g., computing or storage nodes. For delivering a predictable service, it is necessary to configure virtual network resources with a particular size (e.g., capacity for a link) and functionality. In this chapter, the idea of virtual routers is combined with virtual links to create a virtual network concept. A bandwidth sharing and routing mechanism is proposed to expose configurable services to hosts that connect to virtual networks. Finally, an implementation and evaluation with virtual software routers and OpenFlow switches is proposed.

In this chapter, we proceed as follows. The next section identifies the problem of delivering service guarantees in virtual networks as well as configurability and describes our proposal of integrating these two features into virtual networks, by introducing virtual links with bandwidth guarantees and virtual routers with configurable routing. We implement both features, namely VxLinks and VxRouters, as described thereafter. Section 5.3 describes the implementation in software of VxRouters and VxLinks and the evaluation of the associated bandwidth control mechanisms. As an alternative, we implement VxRouters and VxLinks using OpenFlow switches, and evaluate also their bandwidth control mechanisms as detailed in Section 5.4. Finally, Section 5.5 concludes this chapter. An application of the proposed virtual network service is proposed in the following chapter.

5.2 Virtualizing networks for service provisioning

Virtual routers, as described in the previous chapters, can enable to programm the routing individually per virtual network. In this Section, we propose to combine this feature with virtual link configuration, to provide users with isolated virtual networks, offering precise services.

5.2.1 The need for networks as a service

A network like the Internet is shared by lots of users and types of traffic. When initiating a communication, a user does not know, which path his traffic will take, nor does he know which other traffic takes the same path, and hence, which is the performance (throughput, latency, jitter) that his traffic can achieve on the paths. He only knows that he will get a connection. This is the principle of the *best-effort* service. Best-effort is the universal sharing paradigm in the Internet, realized by TCP congestion control mechanisms. It allows to share the network in a fair way (in terms of bandwidth), between all users, whose traffic comes together in the network core.

With the increasing popularity of virtualization at the end-hosts, i.e., the computing and storage nodes in data centers, even the outermost edges of the network are shared by different users of different virtual hosts. A link that connects a virtualized computing or storage node to the network, can automatically be considered like ‘virtualized’. It actually

connects several virtual machines to the network, where each has for example a different IP address. An example of virtualized hosts on 4 physical machines is represented in Figure 5.1. In this example, the virtual machines belong to two different users, A and

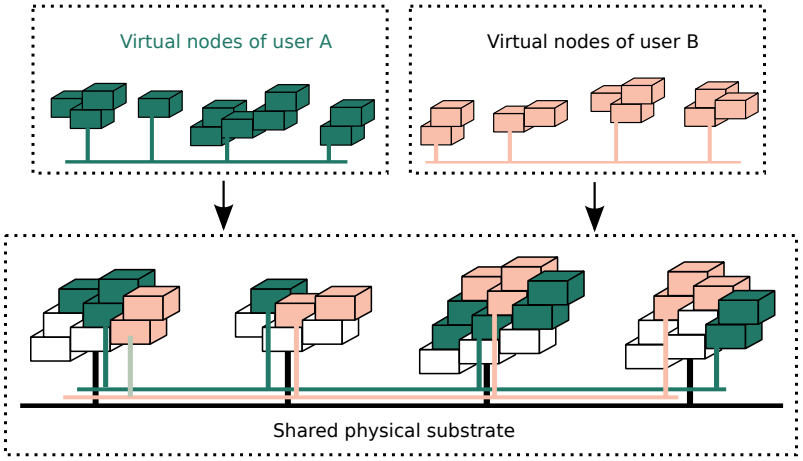


Figure 5.1: Virtual nodes sharing a physical network.

B. The virtual machines of each user are interconnected sharing the underlying physical substrate network. The topology and characteristics of this network are hidden to the end-hosts. In such a scenario, the way of sharing the physical network is also unknown. However, *in order to get predictable network performance between virtual machines, the network must be shared in a controlled way.* Hence, we advocate that a virtual network must be delivered to the user as a resource, together with a service that he can configure. This service is globally characterized by a bandwidth and a latency. To deliver such a service, we propose to virtualize all resources of the physical network, i.e., routers and links, and to control the resource sharing.

5.2.2 A new network architecture

For getting predictable service in a network, the two basic requirements we define are, to control:

1. *Which resources are used*: This is defined by the routing mechanism that defines, which path a traffic takes, and hence through which nodes and links it travels;

2. *The amount of resources*: This is defined by the capacity and latency of routers and especially links, and by the queuing and scheduling mechanisms in routers.

For controlling these in a virtual network, it is necessary to enable the programmability of routers and the configurability of bandwidth of links. For enabling this, we propose to virtualize the routers, as well as the links. Hence, the network is no more transparent, but its different elements are exposed for configuration and programming and can be provisioned with a service. Figure 5.2 shows the architecture of such a virtualized network. It represents the same resources than in Figure 5.1. In addition, it shows the virtualized network that interconnects them.

The goal of virtualizing the whole network topology like this is that for each user, A and B, the routing mechanism and forwarding paradigms in the virtual network can be

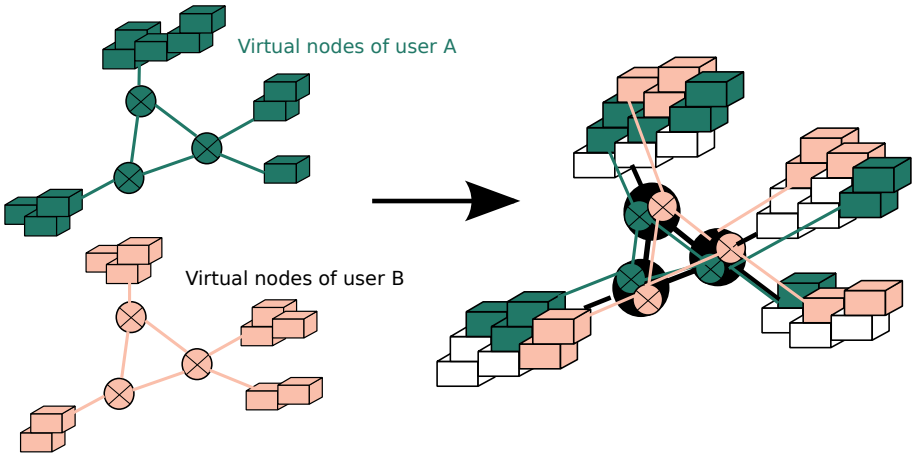


Figure 5.2: Virtual nodes with a dedicate virtual network.

programmed. Jointly with this configuration, bandwidth can be attributed to the different virtual links, to offer a well-defined and predictable communication service.

With such a virtual network service, any user of a virtual network has the illusion that he is using his own dedicated system, while in reality, he is using multiple systems, part of the global physical network.

The precise topology, the per virtual link bandwidth and the routing mechanisms of a virtual network can be specified by the user, e.g., using VXDL, the Virtual Network Description Language [Koslovski et al., 2008]. Such a specification results in a topology of *VxRouters (virtual routers)* and *VxLinks (virtual links)*, with the following features.

- VxRouters: Virtual Routers.

  We define *VxRouters*, as fully customizable virtual routers interconnecting virtual links in a virtual network. In our approach, all traditional network planes (data, control and management) are virtualized. Hence, users can deploy customized routing protocols. In addition, they can even configure the packet-queuing disciplines, packet filtering and monitoring mechanisms they want. Consequently, the user is provided with a fully isolated virtual network.

- VxLinks: Virtual Links.

  Virtual links, we call *VxLinks*, interconnect virtual routers (e.g., VxRouters) and virtual end-hosts. Each VxLink consists in a temporarily allocated partition of a physical link. A VxLink can be configured with bandwidth for each direction, according to the applications' requirements. Taking into account the direction of a VxLink is useful, as some links of a virtual network may be used mainly in one direction. Thus, it is not necessary to allocate the same capacity on the physical link in both directions. Hence, some physical link capacity can be saved. The advantage of having controlled virtual links is twofold: users get strong guarantees on bandwidth, while the physical network can be better exploited by sharing it efficiently but differently.

This de-materialization of routers and links enables to define specific routing and bandwidth services on a per-virtual network basis. These are described in the following.

5.2.3 A virtual network routing service

Building virtual networks with virtual routers, the path the traffic takes across the virtual network can be configured differently for each virtual network. Virtualizing routers to enable their configuration has a similar goal than the overlay network approach (see Section 2.3.3 of Chapter 2). But instead of controlling the network from the edges, virtualization allows to control the nodes inside the network, the VxRouters.

Customized routing and traffic-engineering functions can be set up on the VxRouters. A user can, for example, choose to adapt the routing of his virtual network in order to satisfy specific Quality of Service requirements, as illustrated in Figure 5.3.

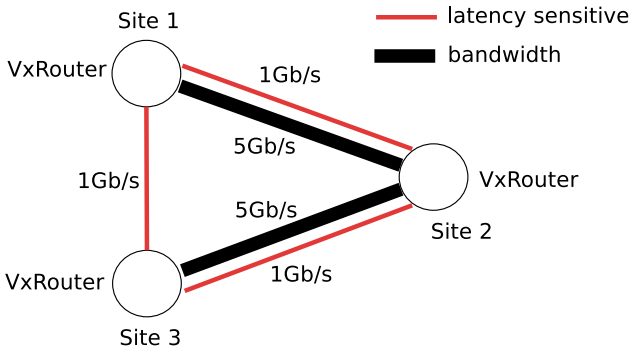


Figure 5.3: Example of bandwidth allocation for latency-sensitive and high-bandwidth flows.

In this example, latency-sensitive traffic and bandwidth-aware traffic are routed on different paths. 1 Gb/s links are allocated between each one of the three VxRouters to transmit latency sensitive traffic. All high-throughput traffic, which can be pipelined, is redirected over Site 2. The advantage of this routing scheme is that the user needs to provision only 2 links with 5 Gb/s instead of 3. As presented in [Divakaran and Vicat-Blanc Primet, 2007], this is an example of efficient channel provisioning combining routing and traffic engineering. To realize such a scenario, a bandwidth service on VxLinks is required. Such a service is defined below.

5.2.4 A virtual network bandwidth service

Virtual networks share a physical substrate network. For ensuring their isolation in terms of performance, the resource sharing between different virtual networks needs to be finely controlled. This consists in particular in the link and hence bandwidth sharing.

For enabling a the control of bandwidth on VxLinks, we defined different services. A virtual network user can request bandwidth R_{req} per VxLink, associated to one of these services, defined as:

- Guaranteed minimum: The user specifies R_{req}, as the minimum rate that has to be available on the VxLink at any moment. The service must ensure that the allocated rate R_{alloc} is always at least equal to this specified minimum:

$$R_{alloc} \geq R_{req}$$

- Allowed maximum: The user specifies R_{req} as the maximum capacity the VxLink must provide at any moment. The service must ensure that the allocated rate R_{alloc} never exceeds this maximum:

$$R_{alloc} \leq R_{req}$$

- Static reservation: The user specifies R_{req} as the exact capacity required on the VxLink. The service must ensure that exactly this capacity is allocated as R_{alloc}:

$$R_{alloc} = R_{req}$$

Specifying bandwidth and one or several of these services (minimum guaranteed, maximum allowed) per VxLink, a user can obtain QoS for applications he executes over the network, typically distributed applications, e.g., running on Grids and Clouds. For example, a user who wants to execute a distributed application communicating with MPI could specify a guaranteed minimum rate which can not be decreased during negotiation. If the user specifies an allowed maximum, the negotiation will result in a bandwidth equal or below this maximum. This link is shared in a best-effort way with other traffic. Specifying a static rate gives the user the impression that he works in a dedicated physical network whose links have the specified capacity. He will be able to obtain the specified bandwidth at any moment but can never exceed it. This kind of service allows no negotiation and could for example help with conducting reproducible experiments where link availability does not vary.
In addition, buffer queue lengths and hence delay in routers are critical to the execution time of distributed applications. Hence, following the model to adjust input rate on a router to control its buffer queue length, proposed in [Zitoune et al., 2009], we could adjust VxLink rate depending on the size of the VxRouter queues, in order to minimize packet loss and control delay. In fact, by dynamically adjusting VxLink bandwidth, the execution time of distributed applications can be controlled as shown in the next chapter.

The above described bandwidth service is implemented as follows. When a user requests a VxLink with a particular bandwidth and service, a decision needs to be taken on the ability to map a virtual network on the physical underlay in a way to provide the requested bandwidth and the associated service. When a virtual network is deployed, rate control is activated or configured to i) guarantee the desired minimum rate and/or ii) limit the rate to the desired maximum for each VxLink. To guarantee a minimum rate on a physical link, all the concurrent virtual links using it are limited.

Let C be the capacity of the link, N the number of VxLinks sharing it, $m_{req}(i)$ and $M_{req}(i)$ be respectively the minimum and the maximum requested bandwidth of VxLink i. A bandwidth $R_{alloc}(i)$ needs to be allocated to the VxLink so that $m_{req}(i) \leq R_{alloc}(i) \leq M_{req}(i)$. In this case, we consider that the best solution is to maximize R_{alloc}, so as to improve performance for the user and revenue for the link operator. Yet, many other strategies are possible to allocate bandwidth on virtual links, according to different criteria.

But for maximizing R_{alloc} for all VxLinks sharing a physical link, the resulting objective function is:

$$\begin{aligned} \text{Maximize} \quad & \sum_{i=1}^{N} R_{alloc}(i) \\ \text{subject to} \quad & m_{req}(i) \leq C - \sum_{j \in [1,N], j \neq i} R_{alloc}(j), \forall i \in [1,N] \\ & m_{req}(i) \leq R_{alloc}(i) \leq M_{req}(i), \forall i \in [1,N] \\ & \sum_{i=1}^{N} m_{req}(i) \leq C \end{aligned} \tag{5.1}$$

Given the user specifications for each VxLink i sharing the link, $m_{req}(i)$ and $M_{req}(i)$ (by default, $m_{req}(i) = 0$ and $M_{req}(i) = C$), the optimum values for the maximum bandwidth ($R_{alloc}(i)$) per VxLink are calculated.

The following example will illustrate such an allocation schema: An available capacity of 2 Gb/s on a physical links is shared by three VxLinks. For VxLinks 1 and 2, the users request respectively a minimum (m_{req}) of 100 Mb/s and 800 Mb/s and a maximum (M_{req}) of 500 Mb/s and 1500 Mb/s. For VxLink 3, the user makes a static reservation of $m_{req} = M_{req} = 300$ Mb/s. At timestamp t_1, all of those three VxLinks are running. At timestamp t_2, VxLink 3 decommissions and only VxLinks 1 and 2 remain, so the rate allocations are recalculated. The allocation is calculated according to the objective function 5.1, using the simplex algorithm. The resulting allocation is illustrated in Table 5.1.

		VxLink 1	VxLink 2	VxLink 3
t_1 (VxLinks 1-3 allocated)	m_{req}	100	800	300
	M_{req}	500	1500	300
	R_{alloc}	500	1200	300
t_2 (VxLinks 1-2 allocated)	m_{req}	100	800	
	M_{req}	500	1500	
	R_{alloc}	500	1500	

Table 5.1: Rate allocated per VxLink (Mb/s).

In this example, during the first phase, from t_1 to t_2, all the virtual links can get the desired minimum rate, but it is not possible to allocate the maximum desired rate for each one. So VxLink 2 can attempt only a rate of 1200 Mb/s instead of 1500 Mb/s. At t_2, VxLink 3 finishes, and VxLinks 1 and 2 can share the remaining bandwidth. From this moment on, both VxLinks 1 and 2 can use their maximum desired bandwidth on the specified VxLinks.

According to these user specifications and the free capacity on the physical substrate, a static bandwidth reservation is made. Then, the control mechanism allocates the maximum bandwidth for each virtual link in order to guarantee the negotiated bandwidth. The configuration takes place on both ends of the virtual link configuring the physical interfaces, depending on the concerned direction of the VxLink.

Our analysis in Chapter 3 has demonstrated that modern virtualization techniques are improving rapidly, making the VxRouter approach if software interesting and promising. Yet, maximum link rate can not be expected on all hardware. Hence, an upper limit needs to be set to the available rate.

The next section gives two types of different implementations of VxRouters and VxLinks. We developed software modules that help deploying VxRouters and VxLinks on top of two different types of hardware. The first allows to deploy VxRouters and VxLinks on commodity hardware installed with Xen. Later, a software module has been implemented to manage virtual routers also on top of OpenFlow [McKeown et al., 2008] switches. The rate control mechanisms in both implementations are further evaluated.

5.3 Implementation in software

This first section describes our analysis and implementation of the mechanisms that allow to control VxLink bandwidth. VxLinks interconnect VxRouters and both are implemented in software.

5.3.1 Implementation of virtual routers and links

In this first implementation, VxRouters are instantiated inside Xen virtual machines, as described in Chapter 3. This gives us the necessary flexibility to experiment and implement rate control mechanisms to provide bandwidth guarantees on VxLinks.

For controlling the rate of a VxLink, we choose among software rate limiting technologies, especially those provided by the Linux `traffic control` tool [TC]. Different locations can be identified inside the physical machine installed with Xen that host the VxRouters, where rate control can take place. Considering the path of a packet through the physical machine as represented in Figure 5.4, there are four possible places on this path to implement traffic control:

1) At the ingress of the physical interface of the physical machine (dom0);

2) At the egress of the virtual interface of dom0;

3) At the egress of the virtual interface of the VxRouter (domU);

4) At the egress of the physical interface of the physical machine (dom0).

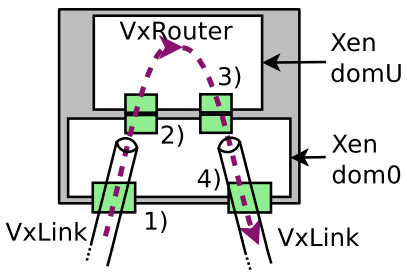


Figure 5.4: Potential locations of rate-control mechanisms in a virtual router.

Limiting rate at the incoming physical interface (1) with the *traffic control* ingress policy would result in dropping packets when the allocated bandwidth is exceeded. This solution is unsatisfactory, causing too much packet loss when the traffic is bursty. Instead, shaping mechanisms would be preferable for controlling the traffic rates on VxLinks. Limiting at the outgoing virtual interfaces of dom0 seems to be a good solution, as the traffic is shaped as soon as possible, before even entering the virtual routers. Limiting at the outgoing virtual interface of the virtual routers would be a solution too, but shaping as soon as possible, i.e., before entering the virtual router, would be preferable. The last shaping opportunity occurs at the outgoing physical interface of dom0 (4), before the packets leave the interface. The advantage, compared with solutions 1, 2, and 3, is that the shaping function knows the traffic of all the VxLinks, since it is concentrated on this interface. This could help in adapting the treatment of one VxLink's traffic according to that of the other VxLinks. We thus focus on limiting the traffic at the outgoing interfaces of dom0, either the virtual ones or the physical ones. To shape the traffic on the physical interface, a classful queuing discipline is required in order to treat the traffic of each VxLink as a different class.

5.3.2 Evaluations

This section presents experimental results on the rate control on VxLinks implemented in software, as described above. Bandwidth provisioning and performance isolation are investigated. To instantiate virtual end hosts and virtual routers, Xen is used. All the experiments are executed within the Grid'5000 [Cappello et al., 2005] platform, using Xen 3.2 and IBM Opteron servers with one 1 Gb/s physical interface each. The machines have two CPUs (one core each).

The goal of these experiments is to evaluate the isolation between several VxRouters, and between several VxLinks when they share physical devices.

5.3.2.1 Experimental setup

Two VxRouters, VR_1 and VR_2 share one physical machine to forward the traffic of two different virtual networks (VNet 1 and 2). Each VxRouter i is connected to four virtual nodes (VN) through four VxLinks ($VL_{i,1}$, $VL_{i,2}$, $VL_{i,3}$ and $VL_{i,4}$). This setup is represented in Figure 5.5.

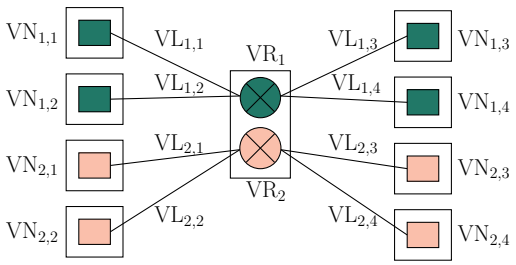


Figure 5.5: Experimental setup with 2 VxRouters (VR) hosted on a single physical machine. Each VR interconnects 4 virtual end-hosts (VN), each located on a dedicated physical machine, through VxLinks (VL) with different bandwidth requirements.

Each VxLink $VL_{i,j}$ is configured to allow a maximum bandwidth $R_{i,j}$. The values we use for $R_{i,j}$ in these experiments are given in Table 5.2. We distinguish user traffic

following three different profiles:

	$VL_{1,1}$	$VL_{1,3}$	$VL_{1,2}$	$VL_{1,4}$	$VL_{2,1}$	$VL_{2,3}$	$VL_{2,2}$	$VL_{2,4}$
$R_{i,j}$ (Mb/s)	50	50	100	100	100	100	200	200

Table 5.2: Maximum rate $R_{i,j}$ of each VxLink $VL_{i,j}$.

1. *in-profile* traffic: the sum of the traffic sent on VxLink $VL_{i,j}$ is smaller than the allowed maximum rate $R_{i,j}$;
2. *limit* traffic: the sum of the traffic on sent on VxLink $VL_{i,j}$ is equal to the allowed maximum rate $R_{i,j}$;
3. *out-of-profile* traffic: the sum of the traffic sent on VxLink $VL_{i,j}$ exceeds the allowed maximum rate $R_{i,j}$.

This experiment aims at determining if the traffic-control techniques limit the traffic to the desired rate and if isolation is guaranteed. Isolation means that limit or out-of-profile traffic on one VxLink should have no impact on the bandwidth available on other VxLinks sharing the same the physical link.

Table 5.3 lists four testcases to validate our implementation. In this experiment, we suppose a congestion factor (CF) of 0.8 for *in-profile* traffic which means that traffic is sent at 80% of the maximum allowed rate. *Limit* traffic has a CF of 1 and for *out-of-profile* traffic, a CF of 1.2 is chosen.

	VNet 1	VNet 2
Case 1	in profile	in profile
Case 2	limit	limit
Case 3	in profile	limit
Case 4	in profile	out of profile

Table 5.3: User traffic profiles for different test cases.

We send four flows simultaneously. In VNet 1, flow 1 is sent from host $VN_{1,1}$ to host $VN_{1,3}$, and flow 2 is sent from host $VN_{1,2}$ to host $VN_{1,4}$. In VNet 2, flow 1 is sent from host $VN_{2,1}$ to $VN_{2,3}$, and flow 2 ist sent from host $VN_{2,2}$ to host $VN_{2,4}$. We perform one experiment using UDP flows and another one using TCP flows. The flows are sent with different rates in each experiment to provoke different congestion factors (CF) per VNet, as specified in Table 5.3.

5.3.2.2 Results

The results of these experiments show that the desired rate on each virtual link can be obtained with our configuration and that *out-of-profile* traffic does not impact other traffic sharing the same physical link.

In case 1, where all the traffic is *in profile*, requiring less bandwidth than allowed, each flow gets its desired bandwidth and no packet loss is detected with UDP. For case 2, all

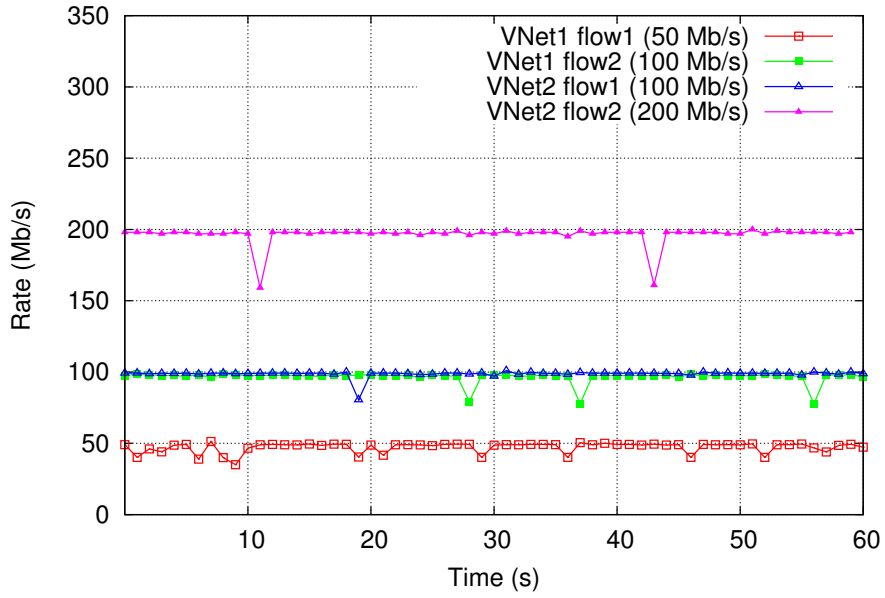


Figure 5.6: Test case 2: TCP Rate with VNet 1 and 2 being at the *limit* rate (with a congestion factor(CF) of 1).

flows try to get 100% of the maximum available bandwidth. This causes some sporadic packet losses which cause some instantaneous throughput decreases in TCP as illustrated in Figure 5.6. The overall throughput still reaches the allowed maximum value. For test case 3, where the traffic of VNet 1 is sent *in profile* and the traffic of VNet 2 at *limit* rate, none of the flow experiences packet loss like in case 1.

Figures 5.7 and 5.8 show respectively the results with TCP and UDP flows in test case 4 where the two flows of VNet 1 are in profile. Their overall congestion factor corresponds to 0.8. They are sent at respective rates of 30 and 90 Mb/s, corresponding to a total of 120 Mb/s. The allowed bandwidth on the VxLinks is equal to respectively 50 and 100 Mb/s. With TCP and UDP, both flows attempt their desired rate and with UDP, no loss is detected. The two flows of VNet 2 try to exceed the allowed value of 300 Mb/s by sending at $120 + 240 = 360$ Mb/s. As a result, they see some of their packets dropped at the incoming interface of the physical router. TCP reacts in a sensitive way to this packet drop. It tries to share the available bandwidth by the two flows. So flow 1 gets generally the 120 Mb/s as it is less than half of the available 300 Mb/s. Flow 2 requiring 240 Mb/s varies a lot to get also an average throughput of about half of the 300 Mb/s.

With UDP (Figure 5.8), the packet drop of the flows of VNet 2 causes a regular decrease of the bandwidth. Flow 1 looses on average 13% of its packets and flow 2 an average of 18% of its packets. The average of these two values is slightly smaller than the percentage of exceeding packets ($1 - (1/1.2) = 16.6\%$) implied by the congestion factor of 1.2. These results show that only flows which try to exceed the fixed limit bandwidth are penalized in this configuration. *In profile* and even *limit* flows are not impacted. In this way, individual rate sharing is efficient in this model and isolation is guaranteed.

The evaluated mechanisms have been implemented in the HIPerNet framework [1], for

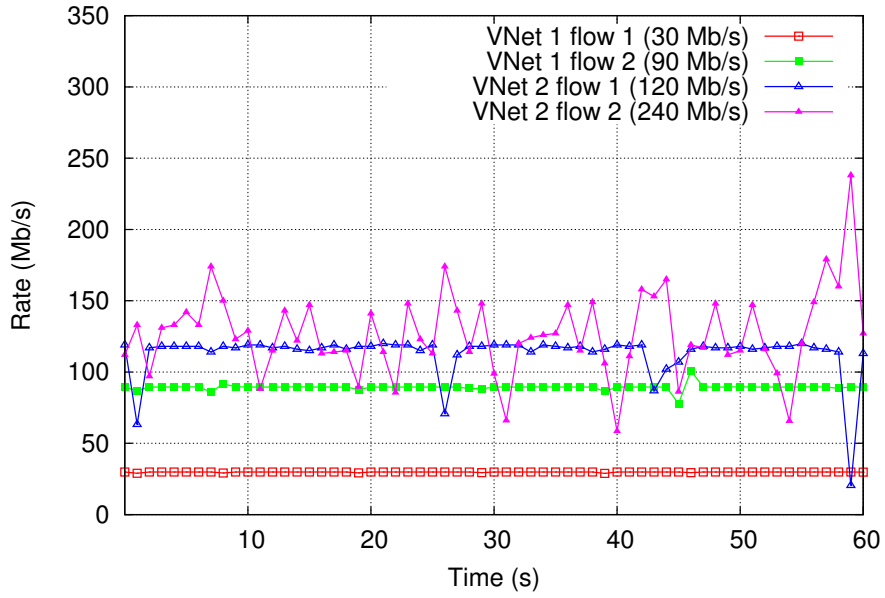


Figure 5.7: Test case 4: TCP Rate with VNet 1 being *in profile* and VPXI 2 out of profile (with a congestion factor(CF) of 1.2).

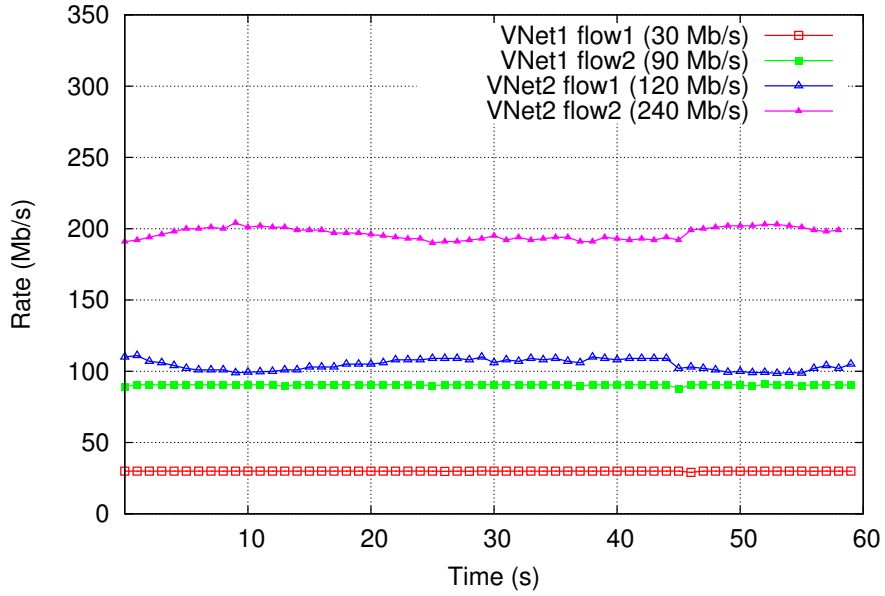


Figure 5.8: Test case 4: UDP Rate with VNet 1 being *in profile* and VNet 2 *out of profile* (with a congestion factor(CF) of 1.2).

application in the context of virtual infrastructures. This application is further detailed in the next chapter. Virtual network users can specify the exact bandwidth for each virtual link, using VXDL, hence creating an optimal execution environment for their application.

5.4 Implementation with OpenFlow

In this second part, as an alternative to the software solution that has scalability issues when high network speed is required, as described in Chapter 3, the setup of configurable VxRouters and VxLinks is investigated on OpenFlow switches. These are interesting for implementing VxRouters, as they allow to configure their flowtables, and hence the routing, as explained in the following. A specific OpenFlow controller was implemented that enables to manage VxRouters and VxLinks in this type of equipment.

5.4.1 The OpenFlow technology

OpenFlow switches are standard hardware switches that are equipped with a TCAM flowtable, which can be programmed using the OpenFlow protocol [McKeown et al., 2008]. This possibility, as described previously in Section 2.4.1.2 of Chapter 2, enables also to implement virtualization on such types of switches. In the flowtable, each entry is composed of the three following elements:

- A flow specification: A set of fields specifying a flow. These corresponds to the classical IP and TCP/UDP header fields in an Ethernet packet;
- Actions: A set of actions to perform on a packet whose header fields match the flow. These actions specify the port to which the packet must be sent out, or can modify its header, e.g., set a VLAN tag, change the IP address, etc.
- Statistics: Statistics relative to the flow, e.g., how many packets have been received.

A complete specification of the OpenFlow protocol can be found at [OF]. Essentially, an OpenFlow switch allows programming the forwarding of flows according to their layers 2, 3 and 4 characteristics.

Messages formatted in the OpenFlow protocol, which are sent to an OpenFlow switch allow to program its flow table, e.g., adding, modifying and deleting one or several entries. They also allow to request information to the switch, for example about the already implemented flows in the flowtable, or statistic, etc.

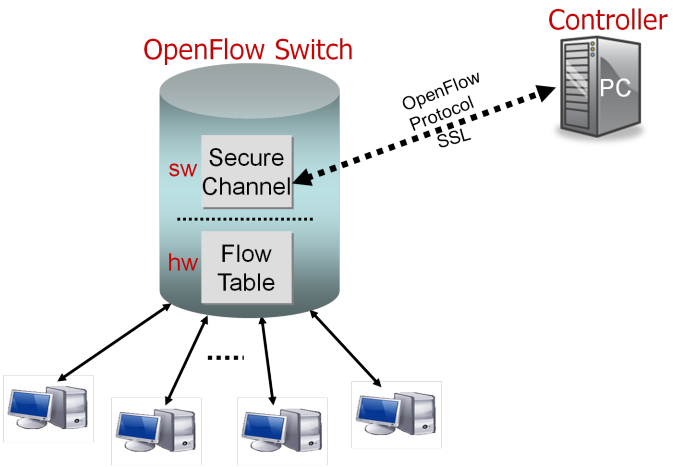


Figure 5.9: The OpenFlow concept [OF].

The OpenFlow concept is depicted in Figure 5.9. OpenFlow switches can be programmed using so-called *controllers* that can be written in any programming language, on any general purpose PC, provided that the controller formats all messages to the OpenFlow protocol, and is connected through a secure TCP connection to the switch. This porting of the control plane out of the box offers enormous flexibility in configuring hardware switches.

An OpenFlow switch can route flows based any packet header field. One rule that a controller could for example set on the OpenFlow switch is "every packet destined to TCP port 80 must leave the switch at port 16". This allows for example to redirect HTTP traffic through a specific network paths. But a controller could also run a routing daemon that takes decision on the IP layer. Hence, it could send rules to the switch like "every packet destined to IP address 10.12.34.56 must be sent out of port 11", and so on. Hence, by choosing a header field as a criteria to identify the traffic of a particular virtual network, e.g., a MAC address, an IP address or a VLAN tag, traffic could be routed differently, according this criteria, thus allowing virtualization. Note that this type of virtualization needs to be implemented outside the switch, inside the software module that controls the programming of the flowtable.

OpenFlow switches can be programmed in two ways: to be *active* or *pro-active*. The active approach consists in setting up flows on the switch in response to a packet arrival. Actually, when a packet arrives at an OpenFlow switch, and matches no entry of the flow table, it is by default sent to the controller. Then, based on the information present in the packet, the controller can decide if it adds an entry to the flow table that matches all the following packets of the same flow. The flow entry can be set with a timeout, so as to disappear from the flowtable after a certain time. This enables to setup dynamic routing or switching.

The pro-active approach consists in pre-programming the flow table, so that flow entries already exist before the first packet of a flow arrives. Packets that do not match any flow entry can be dropped. This is comparable with static routing.

Moreover, depending on the vendor, some OpenFlow switches allow to set up traffic control mechanisms. The latest specification of the OpenFlow protocol includes an action that consists in sending packets out to a particular queue. On switches that have ports with several queues, and that allow to setup different rates or priorities on the different queues, basic QoS is possible by sending flows explicitly to queues, which have been pre-configured, e.g., via SNMP or directly on the switch. Other vendors propose particular implementations of QoS. HP proposes for example a specific extension to the OpenFlow protocol that enables to set up rate limiters, and to send flows explicitly to these rate limiters [Kim et al., 2010].

Such OpenFlow switches are a very interesting alternative to software routers, as they are also programmable, even if it is less, but they offer the performance of dedicated switching hardware. Actually, there is no virtualization layer that causes overhead during packet forwarding.

5.4.2 An OpenFlow controller for a virtual network service

This section describes our investigations on how to build VxRouters inside OpenFlow switches, and how to control VxLinks interconnecting such VxRouters.

5.4.2.1 VxRouter and VxLink management with OpenFlow

For providing a virtual network service, we propose to use the OpenFlow protocol to set up VxRouters and VxLinks on OpenFlow switches in the following way:

- VxRouters: VxRouters allocated on a physical device use a partition of its ports, and have a specific routing configuration. Building such VxRouters on top of OpenFlow switches is possible through the configuration of their flowtables. Actually, a VxRouter's routing table can be translated into a set of rules that correspond to flowtable entries. The subset of ports a VxRouter is allowed to use can also be controlled through the flowtable configuration. All the flowtable entries corresponding to a specific VxRouter must have the output ports set only to the ports allowed to be manipulated by this VxRouter.
  Besides, VxRouters implemented on OpenFlow switches are no more limited to perform layer 3 routing. We keep the name as routers, but what they actually do is forwarding the traffic on any criteria that can be specified by OpenFlow, i.e., layer 2, 3 or 4 packet header fields. Note that layer 1 is also included in specific OpenFlow implementations for optical equipment [Das et al., 2010]. In summary, we define a VxRouter on an OpenFlow switch by a set of forwarding rules.

- VxLinks: In order to implement VxLinks with specific capacities between VxRouters allocated on OpenFlow switches, rate control must be performed on the OpenFlow switches for guaranteeing the requested rate on each VxLink. Rate control on switches output ports can be performed through queuing. The OpenFlow protocol enables to send packets of a particular flow to a specific queue of an output port. Hence, it is possible to send all flows that are supposed to use the same VxLink, to the same queue that is configured with the rate required by the VxLink.
  As an alternative to queuing, rate limiting tools are provided by the extended OpenFlow implementation in HP switch's firmware. This extension consists in the ability to set up rate limiters on the switch that are independent of ports, and that limit all traffic that is sent through them to a specific maximum rate. These rate limiters are implemented in hardware. An additional *action* in the OpenFlow protocol allows to send flows through these rate limiters. We set up one rate limiter per port and per VxLink, and all flows that use a VxLink are sent though the corresponding rate limiter. This enables bandwidth isolation between VxLinks.

For configuring VxRouters and VxLinks on OpenFlow switches, as described above, we developed a controller module that allows to translate generic VxRouter or VxLink configurations into OpenFlow configurations. This module interfaces with a generic VxRouter and VxLink configuration module, responsible for configuring virtual networks. The role of the OpenFlow controller in configuring virtual networks is represented in Figure 5.10. Creating and managing VxRouters and VxLinks consists on the one hand in sending configurations to the physical devices, here the OpenFlow switch, to configure them. Generic requests we term *'create VxRouter'* or *'create VxLink'* must be translated into OpenFlow protocol specific flow or rate limiter configuration requests together with a flow or rate limiter specification. We simplify these as *'add flow'* and *'add rate limiter'* on Figure 5.10. The controller is responsible for this translation. On the other hand, the management of virtual networks consists also in retrieving information from the VxRouters and VxLinks

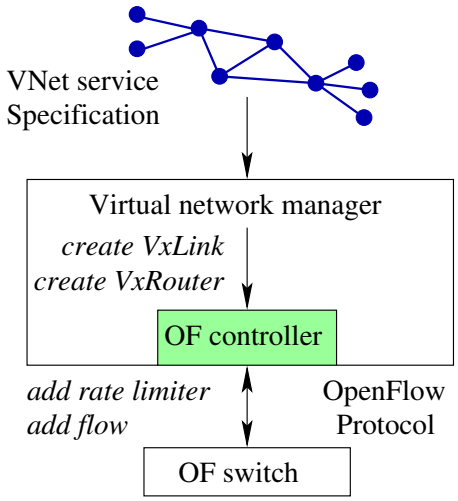


Figure 5.10: VxRouter and VxLink management module for translating virtual network (VNet) service specifications to OpenFlow (OF) switches.

either by sending requests to get the device status, or by receiving spontaneous messages from the device. For handling these two functionalities when managing virtual networks, we implemented the specific controller module described in the following.

5.4.2.2 Controller architecture

For sending configurations to the switches, as well as receiving on replies or spontenious messages from the switches, e.g., resource states, the controller must be running permanently. It listens to messages from the switch to propagate them upwards, as well as to messages from the virtual network manager, to propagate them downwards, to the switch. The controller's role is also to ensures synchronization of the data between VxRouters's internal data and the switch configuration, e.g., between the routing tables of all VxRouters, and the flowtable of the switch.

We built the controller using the library provided within the reference controller that is released within Open vSwitch [OVS]. This library provides all the necessary primitives to setup connections between controller and switches, and to send messages to the switch and receive messages from the switch. Hence, OpenFlow messages, formed according to VxRouter and VxLink configurations, can be sent to the switch, and status messages can be read from the switch and be exposed to the upper layer, the VxRouter or virtual network management layer. The controller performs these two tasks in parallel, listening to the switch, and listening to VxRouter configuration messages, or information requests, as represented in Figure 5.11. Typically, when a VxRouter configuration is issued by the VxRouter manager, and requests to the controller to add a route, the controller sends an *add flow* message to the OpenFlow switch, and then a *get flow info* message to ask for the presence of the flow. Then it waits until the process listening on the switch returns the reply and according to the information, sends back to the VxRouter manager if the flow could be added successfully or not. This additional check is performed in order to be sure the route could be added and information is consistent between the internal data of the VxRouter manager and the actual switch flowtable.

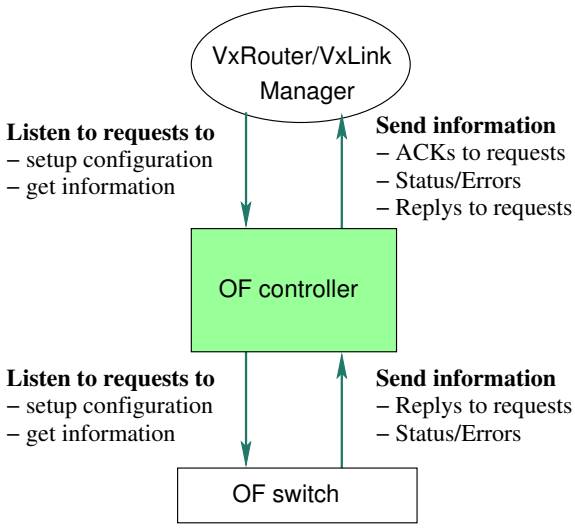


Figure 5.11: OpenFlow controller interacting with VxRouter and VxLink manager and the switch.

5.4.2.3 Alternative implementations

An existing solution for virtualizing an OpenFlow switch is FlowVisor [Sherwood et al., 2010]. FlowVisor operates as a proxy between one or several OpenFlow switches and one or several controllers. It allows different controllers to control different subsets of flows on the same switch, hence virtualizing the latter. For example, one controller could control only HTTP flows. Another controller could control for example only flows that enter at ports 1 to 7 of a switch, etc. Any type of filtering based on the packet header fields is possible. Controllers connect to FlowVisor and get the impression that they are directly connected to the switch. FlowVisor then sends the OpenFlow messages from the controller to the switch. The messages coming from the switch are also sent to FlowVisor, which distributes them to the responsible controller. This is useful if several independent entities want to run their own controllers on a single OpenFlow switch.

In our first VxRouter implementation, the VxRouter manager itself is in charge of performing the virtualization. It is actually the only entity that controls and configures all the VxRouters of a switch, and hence does some of the tasks of FlowVisor. Before sending a request to the controller, it verifies that the required configuration does not conflict with another VxRouter's configuration.

In a further version of a VxRouter manager that includes the possibility for users to deploy their own OpenFlow controllers (e.g., we could imagine a separate controller per VxSwitch, say a *VxController*) it could be interesting to use FlowVisor. It could divide the OpenFlow messages between such VxControllers and the default OpenFlow controller we use presently for pre-configured VxRouters. However, this would require to configure FlowVisor in a way to make sure, that different VxControllers could not configure the same entries in the flowtable.

Regarding the controller, NOX [Gude et al., 2008] is the most popular existing platform, which operates like a network operating system, able to control a network of switches. It features an API, which enables developing controller modules using simplified primi-

tives, which allow the configuration of several switches at the same time. We chose to build our own controller directly above the OpenFlow protocol, so as to have the possibility to fully exploit the features of the protocol, some of which are hidden by the NOX API for simplication. In addition, we used a specific protocol version patched with the rate limiter feature of the HP switch, which was not supported in NOX by default. Besides, our goal was not to manage several switches at the same time with the controller, as we suppose that the virtual network logic is implemented in a layer above, for example using a manager such as HIPerNet [1], discussed in the next chapter.

In the following, some preliminary experiments are described that have been performed on OpenFlow switches, to evaluate the rate limitation mechanisms and the performance that can be expected on VxLinks.

5.4.3 Evaluations

The aim of these experiments is to calibrate the OpenFlow switch performance, so as to know which is the available capacity, and which is the latency to expect when configuring VxLinks and VxRouters. These parameters particularly important as an input for a virtual network allocation process [Koslovski et al., 2011]. Hence, in these experiments, the rate control mechanisms on OpenFlow switches are evaluated, as well as the latency. All experiments are executed on HP Procurve 6600 switches, installed with the OpenFlow enabled firmware from HP (version K_14_63/2.02h).

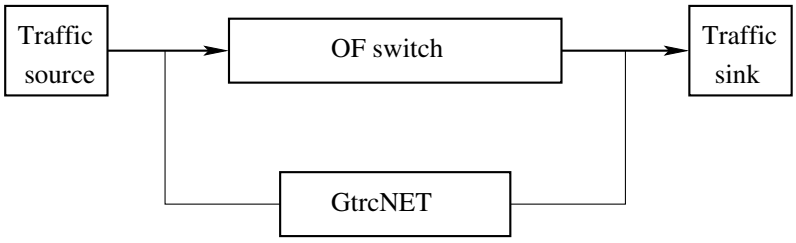


Figure 5.12: Test setup for OpenFlow experiments.

5.4.3.1 Latency

For evaluating the eventual overhead of the OpenFlow flow matching process, we measure the latency on an OpenFlow switch. In a simple setup, flows are sent over an OpenFlow HP Procurve switch from one Linux machine to another Linux machine that act as traffic sources and sinks. This setup is represented in Figure 5.12. In addition, a GtrcNET-1 device [Kodama et al., 2004] is plugged between traffic source and switch input, and between switch output and traffic sink, in order to measure the latency on the switch. In this experiment, different switch configurations are used. First, latency is measured on the switch running the native HP firmware, OpenFlow being disabled. Second, OpenFlow is enabled and flows are added in order to allow the traffic source to communicate with the traffic sink. In this specific switch model we use, most of the OpenFlow functionality is implemented in hardware, while some of it is implemented in software modules of the switch. For instance, actions that modify packet headers, e.g., setting a VLAN tag, or changing source or destination addresses, are typically implemented in software. However,

such type of modifications can be useful in the management of virtual networks. Hence, in a third experiment, we force the traffic to use the software path by modifying one if its header fields. In all experiments, traffic consisting in 128 Byte packets is generated using iperf [IPE].

The results of the three experiments are represented on Figure 5.13. They represent

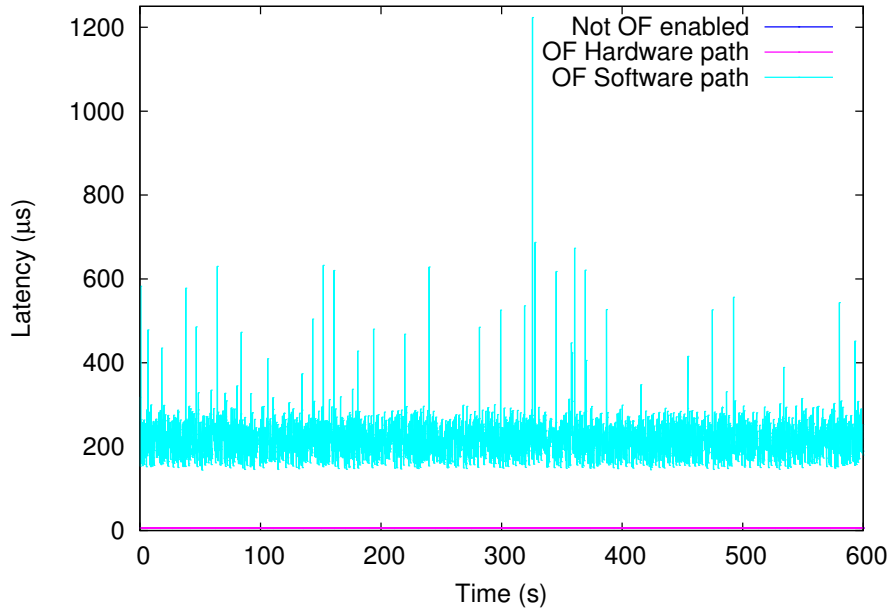


Figure 5.13: Latency on an OpenFlow enabled switch.

average latencies over intervals of one second. Latency is almost the same on the non OpenFlow enabled switch, and the OpenFlow enabled switch where the flow is set up in hardware. It corresponds respectively to about 5.8 µs and 6.8 µs for 128 Byte packets. Using the software paths, packets undergo on average a latency of over 200 µs, which varies by over 200 µs. Hence, when implementing header field update functionality on VxRouters, the corresponding latency has to be taken into account on this particular type of switch.

5.4.3.2 Rate limiting

For calibrating the available VxLink capacity, we evaluate, the rate limiting mechanisms. The same test setup is used than the one represented in Figure 5.12, but without using the GtrcNET device. Flows are generated using iperf [IPE]. In a first experiments, UDP traffic is sent at maximum rate through the 1 Gb/s ports of the switch. The experiment is then repeated with TCP traffic. In the switch, flows are set up to allow the communication between the two hosts, and a rate limiter is set up to limit the traffic of these flows. The maximum rate of the rate limiter is changed every 10 minutes, starting from 100 Mb/s up to 1000 Mb/s by increments of 100 Mb/s.

Figure 5.14 shows the rate, to which the rate limiter is configured, and the obtained UDP and TCP throughput. The results obtained correspond to the average rates over one second of data inside maximum sized IP packets (1500 Bytes). Converting these to

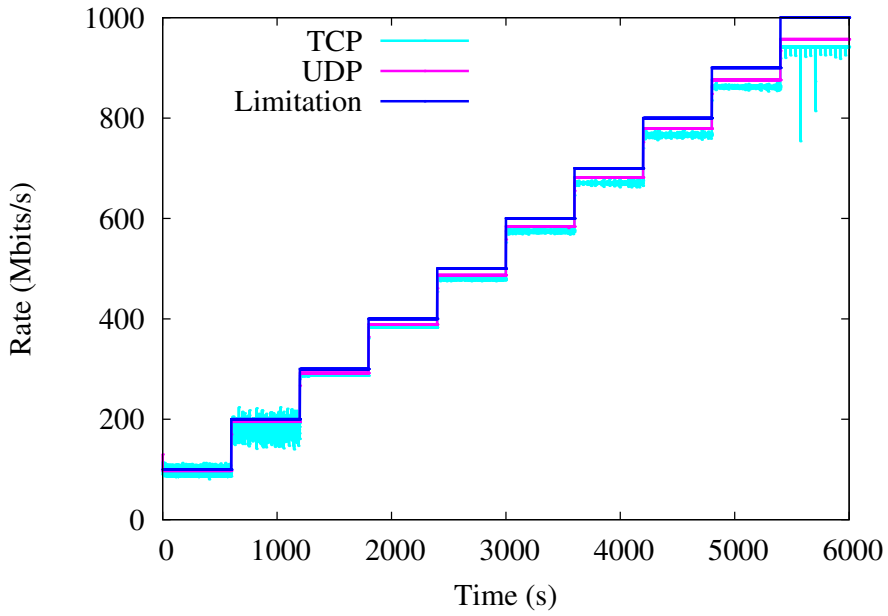


Figure 5.14: Throughput on an OpenFlow switch with rate control.

the Ethernet frame rates, i.e., multiplying them by *Ethernet frame size/data size*, we find that UDP traffic is transmitted very precisely at the rate specified at the rate limiter[1]. This is also the case for TCP on average, but TCP reacts differently to the limitation and hence throughput variation is much higher, which we relate to the type of scheduling performed for limiting the rate. Yet, in both cases, we consider that the rate on VxLinks can be efficiently controlled using the rate limiters.

Results using the software path are not represented here. The limitation works the same way, but as it is using more processing, its rate is actually limited to 100 packets per second by default, and can be increased to no more than 10000 packets per second. Hence, available VxLink capacity can not exceed these thresholds that result from this rate and the packet sizes.

These preliminary results allow to be aware of performance when allocating VxRouters and VxLinks on the physical substrate, i.e., the routers and switches. Hence, different allocation strategies can be taken according to the performance required by the virtual network [Koslovski et al., 2011]. Indeed, it is very important to ensure the required performance on VxLinks as shows the application of virtual networks, described in the following chapter.

[1]The Ethernet frame size corresponds to 1514 Bytes (IP packet size + Ethernet header), while the data size corresponds to $1500 - 20 - 32 = 1445$ Bytes in TCP, where 20 Bytes is the size of the IP header, and 32 Bytes the size of the TCP header; and to $1500 - 20 - 16 = 1464$ Bytes, where 16 is the size of the UDP header.

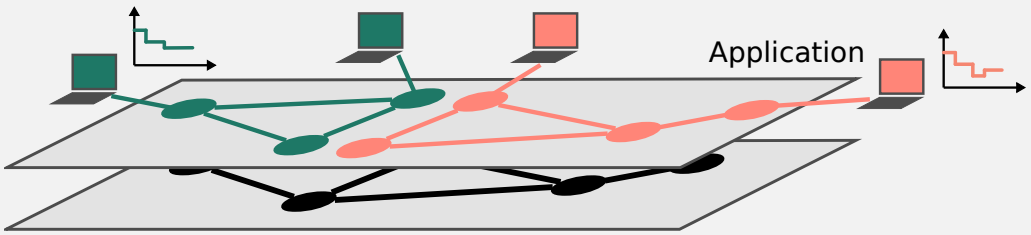
5.5 Conclusion

After having built a virtual router prototype in Chapter 3, able to run on commodity hardware, we integrated the router to the virtual network concept in order to control routing between distributed virtual nodes. As a promising and powerful alternative, virtual routers based on OpenFlow technology were developed. Moreover, a virtual link (VxLink) concept was defined to interconnect virtual routers. Rate control mechanisms were designed and implemented, allowing users to adjust the capacity of VxLinks exactly to the required amount at the desired time. The virtual network implementations in software and with OpenFlow were evaluated, and both are able to isolate the traffic of different VxLinks. Virtual routers implemented on OpenFlow switches can deliver higher performance, while still offering high configurability. In the following chapter, this virtual network service is applied to the context of virtual infrastructures with the goal of enabling the precise configuration of the network inside virtual infrastructures, and providing strict performance guarantees.

6 Application of virtual networks

Part of the content of this chapter has been published at the 20th ITC Specialist Seminar 2009 [6], at the Journal of Grid Computing (JoGC 2010) [3] and has been presented at the Grid'5000 Spring school 2009 [16]. In addition, a research report has been published [12].

Abstract. With the Infrastructure as a Service (IaaS) paradigm and the Clouds, geographically distributed virtual IT (computing and storage) resources can be requested on demand. The interconnection network plays an important role in the quality of service delivered. This chapter investigates virtual infrastructures combining IT and network. The previously presented virtual network service is used to interconnect virtual infrastructure's IT resources in a controlled way. A use case and experimental validation show the resulting isolation and its impact on the performance of virtual infrastructures.



6.1 Introduction

The concept of virtual infrastructures has emerged from the virtualization of end-hosts, and the convergence of computing and communication. The Internet is not only a black box providing pipes between edge machines, but has become a world-wide reservoir increasingly embedding computational and storage resources to meet the requirements of emerging applications [RES, 2008]. The promising vision of grid computing—to bring together geographically distributed resources to build large-scale computing environments for data- or computing-intensive applications—and the service wave led naturally to the "Infrastructure as a Service" (IaaS) paradigm [IaaS, 2008].

Currently, virtual infrastructures, e.g., those enabled by Clouds, are composed of virtual computing and storage nodes. These virtual infrastructures have two limitations: i) they share the network that interconnects them in an uncontrolled way, and ii) the network is not included in the service and not exposed to the user for configuration. Thus, users have no control neither on the path their traffic uses, nor on the fraction of bandwidth it gets. This can result in unpredictable performance of applications that use the network for communication among several virtual computing and storage nodes. This is exactly where the virtual network service proposed in the previous chapter can come as a solution.

Hence in this chapter, we proceed as follows. First, Section 6.2 describes the context. The concept of virtual infrastructures as a service is detailed and the problem of providing network services in such infrastructures is identified. Then, in Section 6.3, we describe our proposal of integrating network control to virtual infrastructures using the virtual network service we presented in the previous chapter. This proposal includes first the implementation of the virtual network service on Grid'5000 in the context of the HIPerNet virtual infrastructure manager. Moreover, it includes the validation of this service, with a focus on performance isolation in virtual infrastructures. This validation is performed through experimentation with a large-scale distributed application.

6.2 Background on virtual infrastructures

A virtual infrastructure is a set of virtual resources, such as computing and storage nodes, which are interconnected. For example, the data-centers of companies are virtualized today. Data-bases, web servers, file servers, etc., are hosted on virtual machines to decrease the hardware cost and need for maintenance. Such a set of virtual machines is a typical example of a virtual infrastructure that emerged with the expansion of server virtualization technologies. Today, this concept is pushed further towards virtual infrastructures hosted somewhere *in the Cloud* that are exposed as a service.

6.2.1 Infrastructures as a Service and the Cloud

An emerging concept of the IT world is the "Everything as a Service" paradigm, abbreviated as XaaS, where software as well as hardware and its administration, and management are available on demand—*as a service*.

New software models provide software over the web as a service (SaaS) [Turner et al., 2003]. Hence, users do need only a minimal local configuration to run the software. Also, they do not need to administer it, which is provided by the service. They only just use it. This trend evolved further towards Platforms as a Service (PaaS), which makes a whole

platform available for users, containing different underlying hardware and the tools to use it. This service model goes along with the recent trends in outsourcing to the Cloud. Clouds have been defined in [Mell and Grance, 2009]. A Cloud consists basically in a set of computing resources, distributed over different geographical locations or sites. These computing resources are in general organized in virtual machines, due to their flexibility and ease of reconfiguration over time. The Cloud concept emerged together with the Grid concept, and the increasing bandwidth on the Internet.
What is mostly not considered in these services is the network. Users connect to their software or platform through the public Internet, as they connect anywhere else. They do not get particular performance guarantees. However, as the network is fully involved in each of these services, it should also be provided to the users as a service, with attributes such as bandwidth and latency.

The PaaS paradigm has been extended to the Infrastructure as a Service (IaaS) paradigm, where a user can request a whole IT infrastructure as a service, for a given duration, deploy his data and applications, and start computing. This is what is mostly provided by public Clouds, e.g., Amazon [AMA], Microsoft Azure [AZU]. This is typically the case of a company that wants to outsource their IT infrastructure. When it comes to infrastructures, not only the network connection towards the users plays an important role in the performance of the service, but also the quality of the connection between the different resources of the infrastucture. Here, it is even more important to include the network as a service. Depending on the quality of the service, users may experience differences, e.g., in processing data that is stored in a data base that is located on a different resource than the software that uses the data for computing. Especially that oftenly, the different resources are located in the same LAN, hence not traversing the public network where control is difficult.

6.2.2 Cloud networking

The network changed is purpose, especially with the expansion of server virtualization and the outsourcing to the Clouds [Rosenberg and Mateos, 2010]. It is no more solely a data-transmission facility, but participates in some way in the computing, as data that is computed on different virtual machines in a Cloud is exchanged over the network during the execution of the application. Figure 6.1 represents a Cloud network of virtual resources of two different clients, distributed over different locations.

In such a case, the communication paradigm between these resources becomes of great importance. Therefore, new techniques to virtualize the network stack in servers are appearing, as well as new types of switches and routers that are able to deal with parallel networks, for virtualized server parks and private Clouds. This is presently possible only at the edge or inside LANs with particular QoS configurations. However, this does not cope with the network performance in geographically distributed virtual infrastructures, e.g., on public Clouds.
The only service that is offered by the network today is connectivity. Anything else, such as security, is implemented at the edges. While security at the edge, like crypting data, secures it also while traversing the core, which is not the case of performance. Traffic shaping can also occur at the edges, but what the traffic experiences in the core is hardly predictable. It is up to now accepted that other users can impact the quality of the connection to the service.

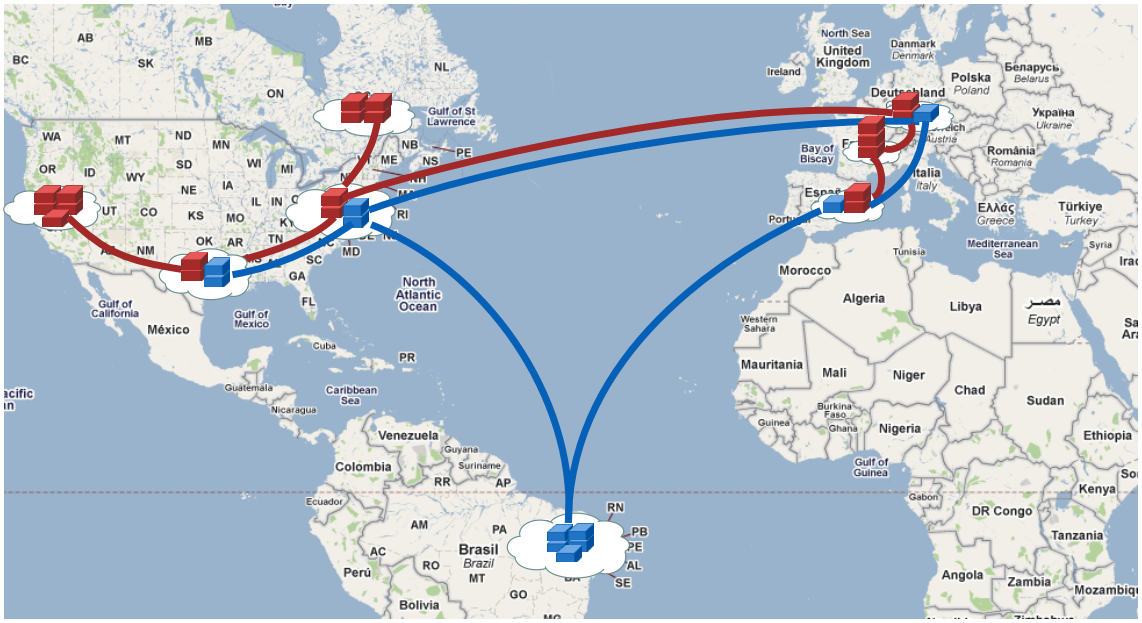


Figure 6.1: Distributed network in the Cloud (the map is a courtesy of Google maps).

Evaluations showed that network performance in Clouds is indeed limited. Throughput is less than on classical non virtualized servers, e.g., decreasing up to less then 50% in a KVM setup with an e1000 driver [Shafer, 2010]. Moreover, throughput is unstable, and latency, varies in an unpredictable way [Wang and Ng, 2010]. This may highly impact network sensitive applications, and thus, could prevent some applications to run on the Cloud. Moreover, the agreement on a business contract between Cloud provider and user depend on performance. Hence it is important to provide predictable network conditions.

This brings two important challenges to consider: i) The performance of the network in virtual machines on servers (as it has been investigated in Chapter 3); and ii) The sharing of the network between virtual infrastructures, and hence Cloud clients (cf. Chapter 5). Gathering these two challenges, we propose to manage virtual IT resources, i.e., the computing and storages nodes, together with a virtualized network, as proposed in the previous chapter. This enables to introduce network control to virtual infrastructures.

6.3 Network control in virtual infrastructures

In this section, the application of controlled virtual networks is investigated in the context of virtual infrastructures. The goal is to provide isolation between virtual infrastructures in terms of network performance, which is a crucial criteria to obtain a predictable execution environment inside a virtual infrastructure. The network must be provided to the virtual infrastructure as a service.

6.3.1 Virtual infrastructures on Grid'5000

Taking the example of Grid'5000 [Cappello et al., 2005], the French national research Cloud, the virtual infrastructure concept is appealing, combined with a controlled net-

work service. It allows researchers to run experiments in confined virtual execution environments, with predictable network performance.

6.3.1.1 The platform

Grid'5000 [Cappello et al., 2005] is an experimental facility, which gathers large scale clusters and gives access to 5000 CPUs distributed over 9 sites and interconnected by 10 Gb/s-dedicated lambdas over the French research and education network RENATER [REN]. It is actually a typical example of a Cloud gathering several computing clusters located on different geographical sites. This infrastructure is represented in Figure 6.2. Grid'5000

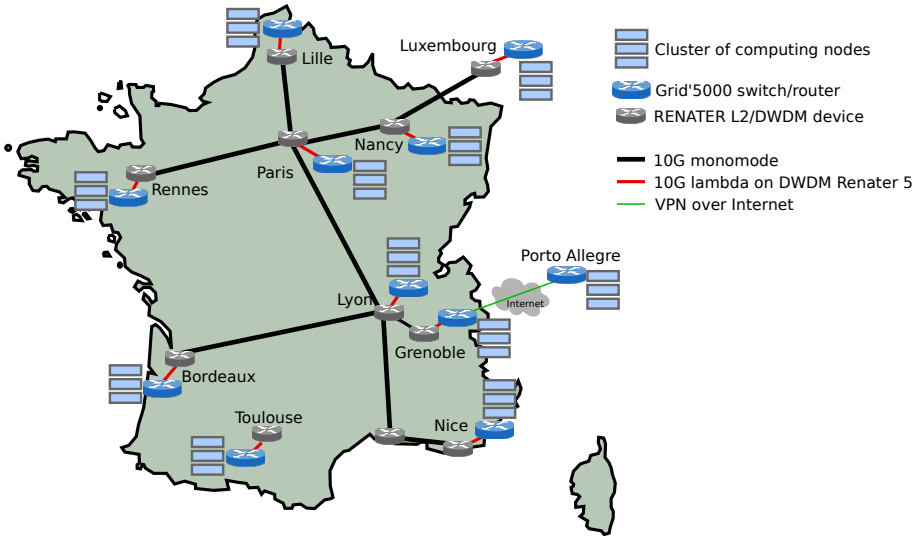


Figure 6.2: Grid'5000 infrastructure.

provides a deep reconfiguration mechanism allowing researchers to deploy, install, boot and run their specific software images, possibly including all the layers of the software stack. This reconfiguration capability leads to the experiment workflow followed by Grid'5000 users: i) reserve a partition of Grid'5000, ii) deploy a software image on the reserved nodes, iii) reboot all the machines of the partition using the software image, iv) run the experiment, v) collect the results and vi) release the machines. Grid'5000 allows users to reserve the same set of resources across successive experiments, to run their experiments on dedicated nodes (obtained by reservation), and to install and run their own experimental condition injectors and measurement software.

While computing nodes can be entirely configured and are dedicated to the user who reserved them, this is not the case for the network. The Grid'5000 backbone, built of 10 Gb/s links, is shared by all users in an uncontrolled way. Neither the bandwidth nor the routing of the network can be controlled by users. Hence, throughput and jitter are not predictable and experiments are not reproducible.

In the following, we describe how the controllable virtual networks, proposed in the previous chapter, can be used in Grid'5000, to expose the network as a reservable, configurable and controlled resource, just as the nodes are, to enable fully controllable experiment infrastructures [6].

6.3.1.2 The network as a resource of excellence

The high configurability of the nodes in Grid'5000 allows installing virtualization software and hence to deploy virtual machines such as VxRouters (as defined in Chapter 5) inside virtual machines. Therefore, it is appealing to use virtual infrastructures with network control on top of Grid'5000 for providing users with confined experiment environments. This means that *a Grid'5000 experiment that requires network guarantees can run inside a virtual infrastructure (VI) with a controlled network service.*

Single-site VI First, a simple example consists in an experiment running within a VI on one site of Grid'5000. Instead of reserving, e.g., simply 4 computing nodes on Grid'5000 for 3 hours, a researcher who needs no more than 100 Mb/s of network bandwidth could reserve a VI, composed of 4 virtual computing nodes for 3 hours, which are interconnected by a VxRouter and VxLinks providing 100 Mb/s bandwidth. When reserving such a VI on a single site of Grid'5000, i.e., all resources are interconnected over a LAN and do not share the backbone with others, each pair of physical nodes is able to communicate at 1 Gb/s, and hence the bandwidth of VxLinks, can be easily guaranteed to reach 100 Mb/s, by configuring the bandwidth as proposed in the previous chapter. This example is illustrated in Figure 6.3. Running an experiment in a controlled VI would on the one hand give the

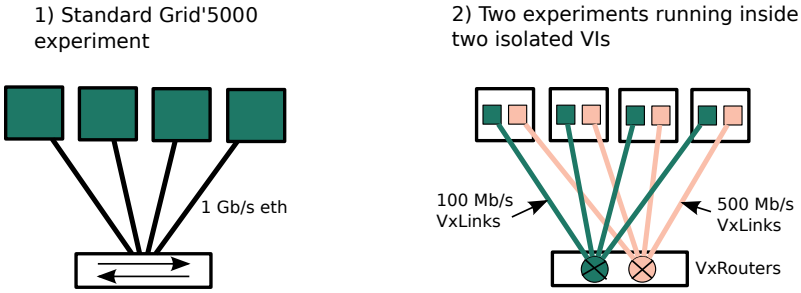


Figure 6.3: Porting Grid'5000 experiments to VIs with network control.

researcher the network performance his experiment requires, and on the other hand, share the resources more efficiently. This example shows that virtualization allows running a second experiment on the same physical resources, and that each could get the required network performance using well-configured VxLinks.

Multi-site VI When reserving VIs with resources that are located on different Grid'5000 sites, such guarantees can not be offered. Neither bandwidth, nor routing can be controlled by a user on the backbones links. Therefore, our approach consists in deploying access VxRouters for each VI on each site it uses, and to interconnect them over the backbone by VxLinks that are also configurable. The traffic of each VI could then be routed differently by different VxRouters. This could allow one to control bandwidth and latency. It requires only one single change to Grid'5000: the traffic using VxLinks over the backbone needs to get rate guarantees. This ability relies on the Grid'5000 access routers and switches to the backbone. One approach consists in deploying all access VxRouters of all VIs of the same site in one single dedicated physical machine. One such machine could be deployed on each site, as represented in Figure 6.4. These machines hosting the VxRouters would

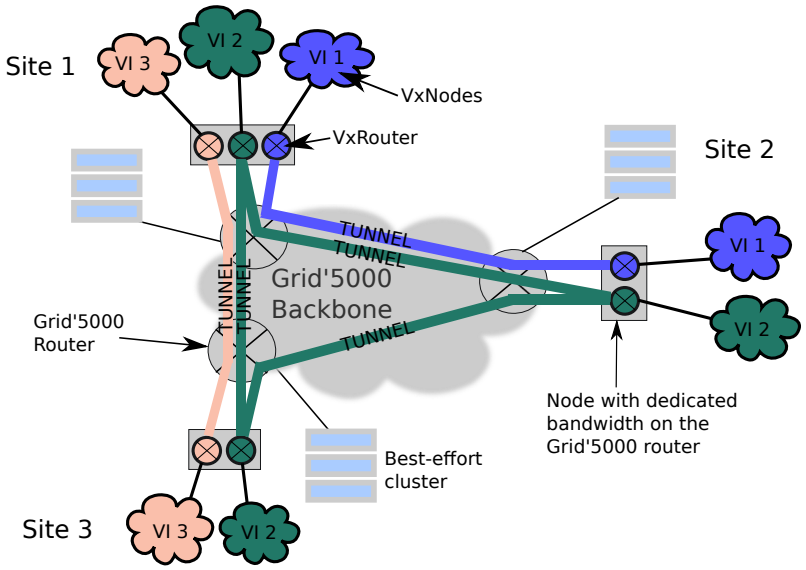


Figure 6.4: Allocation of 3 VIs in the Grid'5000 infrastructure.

be connected to the Grid'5000 access router with a privileged connection, as represented in Figure 6.5. The switch, where the VxRouter-traffic and the Grid'5000 best-effort traffic come together distinguishes the two traffics and gives a strict priority to the VxRouter-traffic. All traffic of VxNodes would be redirected over the access VxRouter, and from

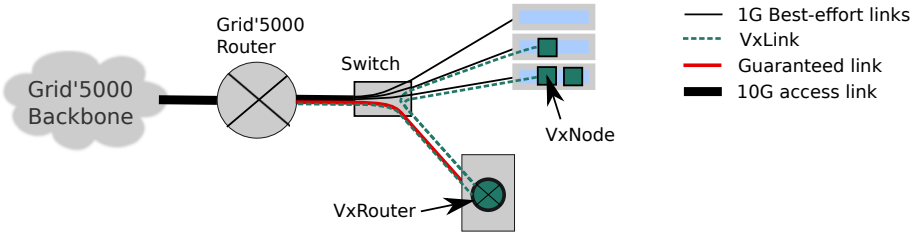


Figure 6.5: Privileged access to Grid'5000's backbone.

there, sent to the backbone over a VxLink, allocated on a physical link with guaranteed rate. This rate can be agreed upon, and the dedicated link can be configured dynamically depending on the number of deployed VIs. The switch, where the machine hosting the VxRouters is connected, needs to ensure that this machine gets the guaranteed rate. The background traffic, i.e., all the traffic that does not require specific treatment, is forwarded through the classical best effort path. With this model, the VxRouters are able to route the traffic of each VI independently, and hence to control which VxLinks it uses. Moreover, the bandwidth sharing of the physical links over the backbone network can be controlled, and isolation can be provided.

In practice, to face scalability issues, we assume that at a given time, only a limited number of experiments will request for a confined VI and dedicated channels (for example, 20% = 2 Gb/s). The others, which are considered not 'communication sensitive', will run without any network control in the classical and fully-transparent best effort mode. Hence,

the aggregate bandwidth allocated for the VI is limited to a configurable percentage of the access link's capacity. The shaped VI-traffic leaving the physical routers hosting the VxRouters is fully isolated from the remaining best-effort traffic of Grid'5000.

Based on this this design, a network reservation service is presently being implemented by Grid'5000's technical comity. In this first implementation, VxRouters are replaced by software router instances and the bandwidth is reserved through VLAN priority configurations.

6.3.2 Implementation in HIPerNet

For the automatic management of isolated VIs with network control, the virtual network management mechanisms described in Chapter 5 have been integrated to HIPerNet [1]. HIPerNet [Laganier and Vicat-Blanc Primet, 2005] is a framework, responsible for the creation, management and control of dynamic VIs, providing a generalized infrastructure service. It supervises these VIs during their whole lifetime, to reprovision resources as required. We integrated the VxLink and VxRouter configuration mechanisms, designed previously, to offer dynamic networking and computing infrastructures as services.

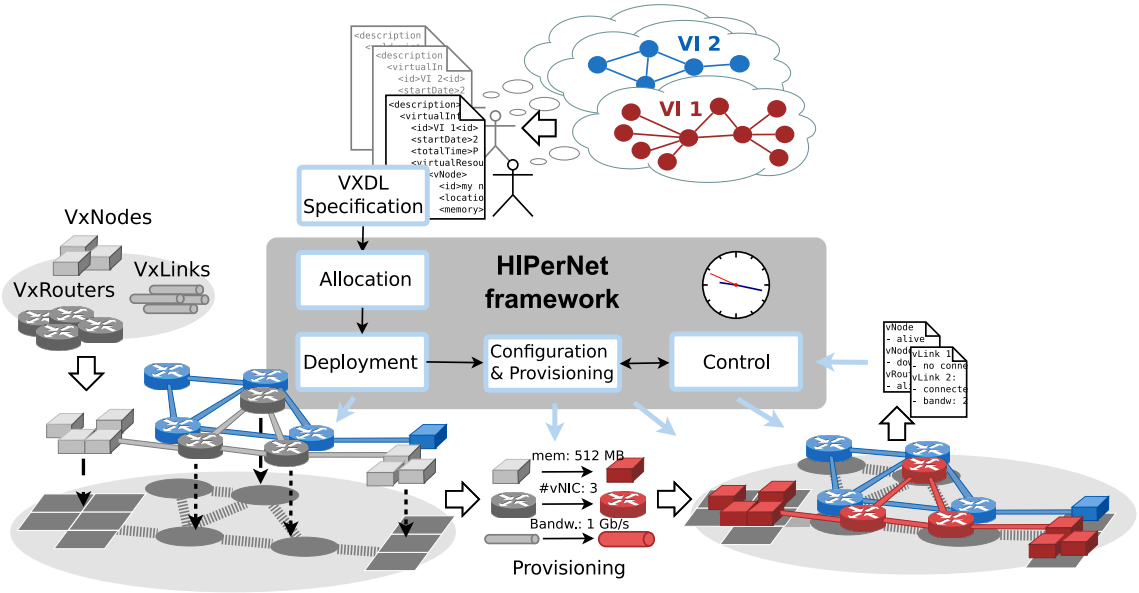


Figure 6.6: VI management with HIPerNet.

A complete usage scenario of the actual version of HIPerNet is depicted in Figure 6.6. In a typical scenario, HIPerNet receives a user's VI request, specified in VXDL, the Virtual Infrastructure Description Language [Koslovski et al., 2008] [VXD]. VXDL allows describing all virtual resources of a VI and their attributes, e.g., the amount of memory of virtual computing nodes (VxNodes), the bandwidth of VxLinks, etc., and their topology. The software interprets the VXDL request, then checks its feasibility and potentially negotiates security, performance, planning and price with the customer or its representative. When an agreement has been found, the VI is allocated and scheduled as proposed in [Koslovski et al., 2011]. Then, the specified virtual entities (VxNodes, VxRouters and VxLinks) are instantiated on the physical nodes and links and provisioned according to the

schedule with the specified amount of resources. The virtual network service ensures that they are interconnected in the required topology and configured with the right bandwidth before being exposed to the user. Our control mechanisms ensure that the resources of the VI are interconnected in the required topology and that VxRouters and VxLinks use no more than the requested amount of resources. Below, we describe the mechanisms that are deployed to virtualize the network resources.
Our contribution to HIPerNet consists in the network configuration module that enables the creation and management of virtual routers with controlled routing, and virtual links with controlled bandwidth.

During the deployment phase, the control tools are invoked at each involved substrate node, in order to enforce allocations during the VI lifetime. At runtime, the control tool's actions can be reconfigured when the VI's requirements change. This happens for example when the execution moves from one *stage* to the next. A stage can be characterized by a certain amount of bandwidth requirements on VxLinks. As an example, a distributed application can have several stages: a first stage where data is transferred to computing nodes, requiring high bandwidth, and a second stage, where computation takes place, requiring high computing capacity, etc. [Koslovski et al., 2008]. At the moment where an execution moves from one stage to the next, HIPerNet updates the configuration of the deployment according to the allocation corresponding to the new stage. Thus, it reconfigures the VxLinks and VxRouters, in order to re-adapt the network to the new requirements.

The resulting VIs are kept isolated from each other and a user has a consistent view of a single private TCP/IP network. The following describes experiments that allow us to validate the concept of controlled VIs.

6.3.3 Evaluation of virtual infrastructure isolation

In this section, the network performance isolation of VIs deployed with HIPerNet is evaluated using a distributed application from the biomedical domain on Grid'5000 [3] [Koslovski et al., 2009].

Distributed applications are necessarily using the network while running. Hence, the network is a resource which takes indirectly part in computing as it allows the application to move data from one processor to another at certain speeds. Running inside a VI, where all network connections are virtualized, such a distributed application will have an execution time that depends directly on the rate allocated to the VxLinks.

6.3.3.1 Tailoring the VI

One of the basic characteristics of a VI is that its resources can be dimensioned and configured, and hence tailored to application requirements. From the network point of view, this means configuring the bandwidth on VxLinks, and the routing on VxRouters.

Application requirements Applications from the biomedical domain frequently require to be executed on several machines in a distributed way, due to the huge amount of data they process. It is difficult to dimension the resources such as CPU, memory, but also network bandwidth, for executing the different tasks of the application on several computing nodes interconnected over a network. In fact, the dimensioning of these resources, as well as their scheduling that depends on the application workflow, directly impacts on the application

execution time [Truong Huu and Montagnat, 2010].

In this experiment, an application from the area of the medical image analysis, Bronze Standard [Glatard et al., 2006], is executed inside a VI, deployed and configured with HIPerNet. This application has a workflow of several services that can be run in parallel or sequentially, depending on their interdependency. Figure 6.7 represents the workflow of Bronze Standard, used in these experiments. This workflow does for example impose that

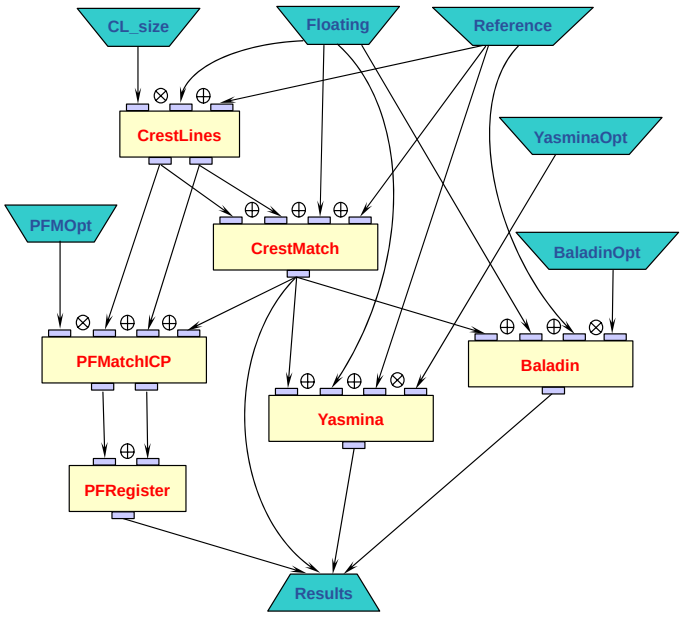


Figure 6.7: Bronze Standard workflow [Truong Huu and Montagnat, 2010].

CrestMatch service starts only once *CrestLines* service finished. Likewise, *PFMatchICP*, *Yasmina* and *Baladin* can start only once *CrestMatch* finished processing all data. Finally, *PFRegister* can start when *PFMatchICP* finished. For running this workflow inside a VI, the different services need to be mapped to virtual resources, in space and time, while VxLinks need to be allocated between the resources, which host services that exchange data. Such a mapping strategy consists in choosing the number and dimension of virtual resources in order to realize the above given workflow. T. Truong Huu and G. Koslovski elaborated different strategies for translating the Bronze Standard workflow into a VI specification, to execute the application inside a VI and minimize the virtual resource costs and the application makespan [3]. According to these strategies, we configured VIs and their bandwidth service in order to evaluated the application makespan.

VI configuration. Three workflow translation strategies were considered in these experiments:

- Naive: All VxLinks interconnecting the computing resources were configured with a capacity of 1 Mb/s;
- Optimized: The different VxLinks were configured with different capacities, which T. Truong Huu had estimated using the number of service invocations, the execution time and the data transfer time. The values are represented in Table 6.1;

- High: All VxLinks interconnecting the computing resources were configured with a capacity of 10 Mb/s.

Moreover, in the optimized strategy, the execution is separated into several consecutive computation stages. In each stage, the required VxLink bandwidth is different.

For each of these strategies, we configured VIs and provisioned all the VxLinks according to the values given by the used strategy. VXDL specifications describing the topology of the virtual resources and their dimensions, were defined for the different stages, and the application was executed using the MOTEUR workflow engine [Glatard et al., 2008].

We developed a tool that allowed a VI to move from one stage to another, by dynamically reconfiguring its topology, its VxRouters, and the capacities of its VxLinks, depending on the given VXDL specification. This allowed us to re-adapt the VI at each stage to the new application requirements, for the optimized strategy.

6.3.3.2 Results and consequences

All experiments were carried out on the HIPerNet instance running on Grid'5000. 30 virtual machines were used for executing the application with the different strategies. Table 6.1 shows the makespan for each workflow translation strategy and the corresponding VI configuration, during each stage, as well as the execution cost. The execution cost is calculated as a function of the unitary cost of a computing resources c_r and the unitary cost of bandwidth c_b. This function is inferred from the cost model presented in [3], which determines the coefficients of c_r and c_b according to the number of resources and their period of utilization. With increasing capacity of VxLinks, the makespan decreases, and

Bandwidth strategy	Stage 1 (s)	Stage 2 (s)	Stage 3 (s)	Makespan (min)	Execution cost ($\times 10^5$)
Low (1 Mb/s)	222.59 ± 2.51	316.57 ± 40.37	2.91 ± 0.50	34.78 ± 0.67	$0.65 \times c_r + 0.31 \times c_b$
Optimized (VxLink capacity)	53.8 ± 4.56 (4.9 Mb/s)	171.72 ± 24.66 (1.95 Mb/s)	1.53 ± 0.23 (3.87 Mb/s)	22.93 ± 0.58	$0.42 \times c_r + 0.48 \times c_b$
High (10 Mb/s)	30.79 ± 3.85	42.68 ± 9.55	1.09 ± 0.18	17.72 ± 0.23	$0.33 \times c_r + 1.61 \times c_b$

Table 6.1: Bandwidth control mechanism evaluation

also its variation. With the optimized strategy, the overall result is the most interesting. Even if the makespan is around 5 minutes longer than in the 'High' strategy, increasing also the computing resources cost by about 20%, the cost of bandwidth is reduced by 70%, leading to an overall cost improvement, considering same unitary prices for bandwidth and computing. These results validate the concept of a controlled network. Consequences on the link between application execution time and VI provisioning cost could be inferred.

6.4 Conclusion

As of writing, virtual infrastructures are broadly know as Infrastructures as a Service (IaaS) in the Cloud. As one of their current limitations is the network performance, we proposed to combine network virtualization with IT virtualization, to obtain virtual infrastructures with a controlled virtualized network. These controlled virtualized infrastructures are isolated from one another, not only logically, but also in terms of performance,

especially of the network. Hence, they present an ideal environment for executing applications, where the network performance is of great importance. In the context of a Cloud, they could also add value by selling the network as an accountable resource delivering guaranteed services. We proposed to apply such virtual infrastructures to the Grid'5000 experimental cloud, in order to carry experiments. In this context, a virtual network service was added to HIPerNet; a virtual infrastructure management framework, one instance of which runs presently on Grid'5000. Finally, this first implementation of virtual network infrastructures, with controlled virtual link rate, allowed us to demonstrate and validate the importance of controlled network virtualization, the sharing mechanisms, and the need for performance control and isolation. This validation strengthens the need for a virtualized network, providing isolation and flexibility, and dissipating the static nature of the network, and hence 'de-ossifying' it.

7 Conclusions and future work

7.1 Summary and conclusions

The Internet has changed its purpose over the last decades, evolving from a communication facility to a shared pool of resources exposed to users as services. This change creates new communication patterns, and a need for a fundamental architectural change in networks becomes more and more imminent. Virtualization is viewed as a solution to make networks, and in particular the Internet, more flexible by abstracting the network from its hardware and literally 'de-materializing' routers and links. This eases the deployment of new features in the network, based on new protocols and new forwarding mechanisms, hence stimulating innovation. In this context, the main concerns addressed in this manuscript are *sharing mechanisms*, necessary when virtualizing the network, and the resulting *performance*. Furthermore, it investigated the *application* of virtualized networks with controlled performance.

In order to obtain maximum configuration flexibility in a virtual network, including packet queuing and scheduling, we set the condition that all network resources should be virtualized at the data-plane.

As a first step, we analyzed and extensively evaluated the performance of techniques that could virtualize the network data-plane. A Linux-based virtual router prototype was built inside a virtual machine to perform routing and forwarding. We evaluated the performance of this virtual router when co-located with other virtual routers on the same host, sharing the hardware. Our results indicate that on the one hand, virtualization in software is a promising approach for experimentation, even from the network point of view. The evolution of software virtualization technologies over the last few years improved to enable, as of the writing, close to 100% network throughput on 1 Gb/s network interfaces. This is moreover eased through the increasing power of hardware. On the other hand, high processing overhead is associated with networking in virtual machines, due to additional packet copies and concurrent resource access. Hence, we concluded that to virtualize high-speed production networks, such as the Internet, dedicated switching hardware is required.

This need for high performance switching led us to examine the current switching architectures and to design a novel architecture that virtualizes the network at layer 2. This architecture, termed as VxSwitch, is based on a buffered crossbar, and its layer 2 resources, such as ports and buffers, can be flexibly shared. In addition, schedulers and queuing mechanisms are virtualized. Thus, VxSwitch allows to create multiple virtual switches sharing the same hardware. Each virtual switch can have a configurable number of ports, capacity per port, buffer sizes, as well as scheduling and queuing mechanisms. The benefit of the high configurability is twofold: through resource dimensioning and sharing, it allows to isolate virtual networks at the lowest level, hence giving them strict performance guarantees; and the configuration of packet queuing and scheduling allows users to finely tune QoS with custom algorithms.

Furthermore, we investigated the application of virtual routers and the sharing of

physical networks into isolated virtual networks. A virtual network service, combining programmable virtual routers with dimensionable virtual links was proposed, to deliver configurable and guaranteed service levels to end users of virtual networks. For realizing this service, different technologies were explored to set up virtual routers and virtual links with controlled bandwidth. One approach consisted in implementing the service using the previously built prototype of a virtual software router. As an interesting alternative, delivering better performance and high configurability, we investigated the OpenFlow technology and proposed a realization of virtual routers on top of OpenFlow switches. These gather configurability and performance in a unique way, enabling the virtualization of the forwarding logic.

We validated the concept of isolated virtual networks by applying it to virtual infrastructures. As discussed beforehand, virtual infrastructures, have been promoted by Clouds and consist in sets of virtual computing and storage nodes made available as a service over a certain period of time. The interconnection network plays a key role in the communication performance achieved by virtual nodes. However, the network is not integrated as a service of the Cloud. To provide virtual infrastructures with performance guarantees, we proposed to combine IT (computing and storage) virtualization with network virtualization. This allowed us to build virtual infrastructures that share the network in a controlled way, and that are isolated from each other in terms of performance. This concept was implemented in the Grid'5000 testbed, as an automatic virtual network management software module. It was validated through experimentation with large scale distributed applications.

Finally, we answer the questions raised in Chapter 1.

[Q_1] *How does virtualization impact network communication performance?*

— The additional layer introduced by virtualization causes additional processing while forwarding packets between NICs and virtual network devices. For virtual routers implemented in software, additional packet copies between NICs and virtual machines cause such processing overhead. Yet, with modern hardware and powerful CPUs, close to 1 Gb/s throughput can be obtained on software virtual routers for big packets, at a CPU cost which is several times higher than on a classical software router.

[Q_2] *Where and how can virtualization be implemented in the network, in order to reduce the performance impact while enabling configurability?*

— For enabling maximum configurability, virtualization needs to be implemented at the data plane, so that each virtual network has the ability to forward, filter, shape, etc., its traffic. Nevertheless, due to the performance overhead introduced by virtualization in software, it is necessary to virtualize the data plane of routers in hardware, i.e., the switching fabric.

[Q_3] *When virtualizing a network, which are the resources that need to be shared?*

— The resources of a network that are relevant to performance and QoS are essentially link capacity and buffer size. Hence, an efficient approach to provide virtual networks with performance guarantees and QoS is virtualize these, so

that each virtual network benefits from virtual links with configurable capacity, and virtual buffers with configurable sizes. This can be achieved with VxSwitch.

[Q_4] *How to provide deterministic performance and QoS to virtual networks?*

— It is the sharing of the link capacity, which is crucial to provide a virtual network with deterministic performance. Hence, it is necessary to configure virtual links, and thus the ports of virtual network devices, with bandwidth. For ensuring that each virtual network device obtains the configured bandwidth, their access to the hardware must be scheduled accordingly. For providing QoS and different service levels inside each virtual network, we advocate to virtualize the switching fabric of network devices, and to enable each virtual network device to be configured with different virtual buffer sizes and different packet scheduling algorithms, as proposed in VxSwitch.

[Q_5] *What are the applications of controlled virtual networks?*

— Thanks to their de-materialization, virtual networks can be deployed on demand when needed. Hence, they can be deployed to interconnect computing and storage nodes of a temporary virtual infrastructure, and provide it with deterministic network performance. This makes the execution time of applications running inside the virtual infrastructure predictable. Such performance guarantees can be especially useful in the context of Clouds, where virtual infrastructures have a business value and are leased as a service. Virtual networks, when controlled, can also be leased as a service with different performance levels.

7.2 Perspectives for future work

The presented work investigated the performance and the sharing mechanisms for virtualized networks. While research and industry now turned towards network virtualization—e.g., research resulted in multiple virtual network testbeds, manufacturers propose virtualized routers for consolidation—large-scale virtualized production networks are still a step away from reality. This is, in our opinion, related to performance and configuration issues. Therefore, we advocate that virtualization must move to the lowest level in the networks, the closest possible to the hardware. We expect a lot of research in this direction. To select a few possible subjects, related with our work, we retain the following.

Scheduling and queuing strategies for VxSwitch. We believe VxSwitch is an approach to take when virtualizing the network hardware. Its high configurability in terms of resource dimensioning and scheduling enables multiple new types of configuration. Resource sharing and QoS can be programmed and realized in many ways. Hence, VxSwitch is a support for innovation, and we plan to investigate new algorithms for differentiated resource sharing between virtual switches. There are many possibilities between strict sharing and variable sharing. In particular, the best tradeoff between guarantees and resource utilization must be found, depending on the context. The resource sharing and scheduling

between all virtual switches and inside each virtual switch should for example be analyzed more specifically. One of our next goals is to implement a first prototype of VxSwitch, for example on FPGA.

Combining VxSwitch with OpenFlow. One very promising solution for programming, and even virtualizing switches is OpenFlow. However, currently it only enables the virtualization of the forwarding logic, meaning the forwarding decisions, through the programming of the flowtable. We plan to combine OpenFlow with VxSwitch, as we think it could be a compelling approach to provide customized routing and forwarding on a per flow-level, where each flow obtains specific QoS. For example, by mapping OpenFlow instances to virtual switches, a flow could be mapped directly to a virtual buffer of a virtual switch. Output scheduling could be adapted to provide specific flow-level service guarantees.

Virtualization in the optical network. Optical fiber is the present technology in the Internet backbone. It moves closer and closer to the edges with technologies such as *Fiber to the home*. Hence, one of our next goals is to examine the optical network components for virtualization. Optical fiber starts providing 100 Gb/s connectivity. Thus, it is appealing to virtualize links into smaller virtual channels. Together with this, virtualizing cross-connects could allow to establish different circuits with those virtual channels, and organize them into different virtual networks. We are currently investigating on the virtualization of the optical network combined with IT virtualization within the Geysers [GEY] project.

Publications

International Journals & Book chapter

[1] Fabienne Anhalt, Guilherme Koslovski, and Pascale Vicat-Blanc Primet. Specifying and provisioning virtual infrastructures with HIPerNet. *ACM International Journal of Network Management (IJNM) - special issue on Network Virtualization and its Management*, 20:129–148, May 2010.

[2] Fabienne Anhalt and Pascale Vicat-Blanc Primet. Analysis and experimental evaluation of network data-plane virtualization mechanisms. *International Journal On Advances in Intelligent Systems*, 2009.

[3] Tram Truong Huu, Guilherme Koslovski, Fabienne Anhalt, Johan Montagnat, and Pascale Vicat-Blanc Primet. Joint elastic cloud and virtual network framework for application performance-cost optimization. *Journal of Grid Computing*, 9:27–47, 2011. 10.1007/s10723-010-9168-6.

[4] Pascale Vicat-Blanc, Sergi Figuerola, Xiaomin Chen, Giada Landi, Eduard Escalona, Chris Develder, Anna Tzanakaki, Yuri Demchenko, Joan A. García Espín, Jordi Ferrer, Ester López, Sébastien Soudan, Jens Buysse, Admela Jukan, Nicola Ciulli, Marc Brogle, Luuk van Laarhoven, Bartosz Belter, Fabienne Anhalt, Reza Nejabati, Dimitra Simeonidou, Canh Ngo, Cees de Laat, Matteo Biancani, Michael Roth, Pasquale Donadio, Javier Jiménez, Monika Antoniak-Lewandowska, and Ashwin Gumaste. Bringing Optical Networks to the Cloud: an Architecture for a Sustainable future Internet, *Chapter in FIA book. Springer Lecture Notes*, May 2011.

Conferences with Proceedings

[5] Fabienne Anhalt, Dinil Mon Divakaran, and Pascale Vicat-Blanc Primet. A virtual switch architecture for hosting virtual networks on the Internet. In *11th International Conference on High Performance Switching and Routing (IEEE HPSR)*, Dallas, Texas, USA, 6 2010.

[6] Pascale Vicat-Blanc Primet, Fabienne Anhalt, and Guilherme Koslovski. Exploring the virtual infrastructure service concept in Grid'5000. In *20th ITC Specialist Seminar on Network Virtualization*, Hoi An, Vietnam, May 2009.

[7] Fabienne Anhalt and Pascale Vicat-Blanc Primet. Analysis and experimental evaluation of data plane virtualization with Xen. In *ICNS 09 : International Conference on Networking and Services*, Valencia, Spain, April 2009. Top paper award.

[8] Dinil Mon Divakaran, Fabienne Anhalt, Eitan Altman, and Pascale Vicat-Blanc Primet. Size-Based flow scheduling in a CICQ switch. In *11th International Conference on High Performance Switching and Routing (IEEE HPSR 2010)*, Dallas, Texas, USA, June 2010. Accepted for publication.

[9] Lucas Nussbaum, Fabienne Anhalt, Olivier Mornard, and Jean-Patrick Gelas. Linux-based virtualization for HPC clusters. In *Linux Symposium 2009*, July 2009.

[10] Fabienne Anhalt, Guilherme Koslovski, Marcelo Pasin, Jean-Patrick Gelas, and Pascale Vicat-Blanc Primet. Les Infrastructures Virtuelles a la demande pour un usage flexible de l'Internet. In *JDIR 09: Journées Doctorales en Informatique et Réseaux*, Belfort, France, February 2009.

Research Reports

[11] Anhalt, F., Vicat-Blanc Primet, P.: Analysis and evaluation of a XEN based virtual router. Technical Report 6658, INRIA Rhône Alpes (Sep 2008)

[12] Vicat-Blanc Primet, P., Koslovski, G., Anhalt, F., Truong Huu, T., Montagnat, J.: Exploring the Virtual Infrastructure as a Service concept with HIPerNet. Research Report RR-7185, INRIA (2010)

[13] Anhalt, F., Blanchet, C., Guillier, R., Truong Huu, T., Koslovski, G., Montagnat, J., Vicat- Blanc Primet, P., Roca, V.: HIPCAL: final report. Research Report, INRIA (2010)

[14] García-Espin, J.A., Ferrer Riera, J., López, E., Figuerola, S., Donadio, P., Buffa, G., Peng, S., Escalona, E., Soudan, S., Anhalt, F., Robinson, P., Antonescu, A.F., Tzanakaki, A., Anastasopoulos, M., Tovar, A., Jiménez, J., Chen, X., Ngo, C., Ghijsen, M., Demchemko, Y., Landi, G., Lopatowski, L., Gutkowski, J., Belter, B.: Deliverable D3.1: Functional Description of the Logical Infrastructure Composition Layer (LICL). Research Report, ICT-2009.1.1 (2010)

Miscellaneous

[15] Anhalt, F., Guillier, R., Koslovski, G. and Vicat-Blanc Primet, P. HIPerNet: virtual infrastructure manipulation. Practical Session in Grid'5000 Spring School (April 2010, Lille, France).

[16] Anhalt, F., Guillier, Mornard, O. and Vicat-Blanc Primet, P. HIPerNET network performance isolation techniques for confined virtual infrastructures. Presentation in Grid'5000 Spring School (April 2009, Nancy, France).

[17] Vicat-Blanc Primet, P., Koslovski, G., Anhalt, F.: Hipcal: Combined network and system virtualization (January 2010) Colloque STIC de l'Agence Nationale de la Recherche (ANR), Paris, France.

[18] Anhalt, F., Gelas, J.P., Vicat-Blanc Primet, P.: Exploration of router performance with data-plane virtualization (June 2009) Poster in Rescom 2009 Summer School, La Palmyre, France.

Patent

[VxSwitch] Anhalt, F and Vicat-Blanc Primet, P. VXSwitch Patent - INPI: N° 10/00368, 2010. LYaTiss, INRIA ENS Lyon - Available: http://www.lyatiss.com

Software

[HIPerNet] Vicat-Blanc Primet, P. and Koslovski, G. and Anhalt, F. and Soudan, S. and Guillier, R. and Martinez, P. and Mornard, O. and Gelas, J.-P. HIPerNet APPcode: IDDN.FR.001.260010.000.S.P.2009.000.10700, 2009, LYaTiss, INRIA ENS Lyon - Available: http://www.lyatiss.com

References

Online References

[4WA] The FP7 4WARD Project. Available: http://www.4ward-project.eu/.

[AMA] Amazon web services. Available: http://aws.amazon.com/.

[AZU] Windows Azure. Available: http://www.microsoft.com/windowsazure/.

[CLI] The Click Modular Router Project. Available: http://read.cs.ucla.edu/click/click/.

[EUC] Eucalyptus. Available: http://www.eucalyptus.com/.

[FI] What is FIA?: Future Internet. Available: http://www.future-internet.eu/.

[G5K] Aladdin/Grid'5000. Available: http://www.grid5000.fr/.

[GEN] The Global Environment for Network Innovations (GENI). Available: http://www.geni.net/.

[GEY] Generalized Architecture for Dynamic Infrastructure Services. Available: http://www.geysers.eu/.

[HYV] Hyper-V Server. Available: http://www.microsoft.com/hyper-v-server/.

[IEE] IEEE - Advancing Technology for Humanity. Available: http://www.ieee.org/.

[IOVa] PCI: Linux kernel SR-IOV support. Available: http://lwn.net/Articles/324612/.

[IOVb] PCI-SIG I/O Virtualization (IOV) Specifications. Available: http://www.pcisig.com/specifications/iov/.

[IPE] Iperf. Available: http://iperf.sourceforge.net/.

[JAI] FreeBSD Jails. Available: http://www.freebsd.org/doc/handbook/jails.html/.

[MYR] Myricom. Available: http://www.myri.com/.

[NEB] OpenNebula. Available: http://opennebula.org/.

[NET] Neterion. Available: http://neterion.com/.

[NFP] NetFPGA. Available: http://www.netfpga.org/.

[NPE] NetPerf. Available: http://www.netperf.org/netperf/.

[OCC] OGF Occi. Available: http://occi-wg.org/.

[OF] The OpenFlow Switch Consortium. Available: http://www.openflowswitch.org/.

[OVS] Open vSwitch. Available: http://openvswitch.org/.

[OVZ] OpenVZ Wiki. Available: http://wiki.openvz.org/.

[QEM] Qemu. Available: http://wiki.qemu.org/.

[REN] Le Réseau National de télécommunications pour la Technologie l'Enseignement et la Recherche.

[SAI] Scalable and Adaptive Internet Solutions project (FP7 SAIL). Available: http://www.sail-project.eu/.

[TC] Traffic Control. Available: http://lartc.org/.

[TUN] Universal TUN/TAP driver. Available: http://vtun.sourceforge.net/tun/.

[UML] The User-mode Linux Kernel. Available: http://user-mode-linux.sourceforge.net/.

[VBX] VirtualBox. Available: http://www.virtualbox.org/.

[VIN] VINI A Virtual Network Infrastructure. Available: http://www.vini-veritas.net/.

[VMW] VMware. Available: http://www.vmware.com/.

[VRZ] Parallels Virtuozzo. Available: http://www.parallels.com/fr/products/pvc46/.

[VSP] VMware vSphere. Available: http://www.vmware.com/products/vsphere/.

[VSR] Linux-VServer. Available: http://linux-vserver.org/.

[VXD] VXDL: Virtual private eXecution infrastructure Description Language. Available: http://www.ens-lyon.fr/LIP/RESO/Software/vxdl/home.html.

[XCS] Xen Credit Scheduler. Available: http://wiki.xensource.com/xenwiki/CreditScheduler/.

References

[RES, 2008] (2008). RESERVOIR - An ICT Infrastructure for Reliable and Effective Delivery of Services as Utilities. Technical Report H-0262, IBM Research Division.

[HVR, 2008] (2008). Router Virtualization in Service Providers. Technical report, Cisco Systems.

[CSD, 2008] (2008). The Cisco Application eXtension Platform: Doing More with Less. Intel Systems.

[JCS, 2009] (2009). Control plane scaling and router virtualization. Technical Report 2000261-001-EN, Juniper Networks.

[OF1, 2009] (2009). OpenFlow Switch Specification Version 1.0.0 (Wire Protocol 0x01). .

[vDS, 2009] (2009). What's New in VMware vSphere 4: Virtual Networking. WMware Inc. White Paper.

[VCS, 2010] (2010). Introducing Brocade Virtual Cluster Switching. Brocade Communications Systems. http://www.brocade.com/downloads/documents/white_papers/Introducing_Brocade_VCS_WP.pdf.

[NEX, 2010] (2010). Virtual Machine Networking: Standards and Solutions. Cisco.

[Andersen et al., 2001] Andersen, D., Balakrishnan, H., Kaashoek, F., and Morris, R. (2001). Resilient Overlay Networks. pages 131–145.

[Anderson et al., 2005] Anderson, T., Peterson, L., Shenker, S., and Turner, J. (2005). Overcoming the Internet Impasse through Virtualization. *Computer*, 38(4).

[Anwer and Feamster, 2009] Anwer, M. B. and Feamster, N. (2009). Building a fast, virtualized data plane with programmable hardware. In *Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, VISA '09, pages 1–8, New York, NY, USA. ACM.

[Anwer et al., 2010] Anwer, M. B., Motiwala, M., Tariq, M. b., and Feamster, N. (2010). SwitchBlade: a platform for rapid deployment of network protocols on programmable hardware. *SIGCOMM Comput. Commun. Rev.*, 40:183–194.

[Appenzeller et al., 2004] Appenzeller, G., Keslassy, I., and McKeown, N. (2004). Sizing router buffers. *SIGCOMM Comput. Commun. Rev.*, 34:281–292.

[Argyraki et al., 2008] Argyraki, K., Baset, S., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Kohler, E., Manesh, M., Nedevschi, S., and Ratnasamy, S. (2008). Can software routers scale? In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, PRESTO '08, pages 21–26, New York, NY, USA. ACM.

[Barham et al., 2003] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA. ACM.

[Barré et al., 2010] Barré, S., Bonaventure, O., Raiciu, C., and Handley, M. (2010). Experimenting with multipath TCP. *SIGCOMM Comput. Commun. Rev.*, 40:443–444.

[Baumgartner et al., 2002] Baumgartner, F., Braun, T., and Bhargava, B. K. (2002). Virtual Routers: A Tool for Emulating IP Routers. In *LCN'02*, pages 363–371.

[Baumgartner et al., 2003] Baumgartner, F., Braun, T., Kurt, E., and Weyland, A. (2003). Virtual routers: a tool for networking research and education. *SIGCOMM Comput. Commun. Rev.*, 33:127–135.

[Bavier et al., 2004] Bavier, A., Bowman, M., Chun, B., Culler, D., Karlin, S., Muir, S., Peterson, L., Roscoe, T., Spalink, T., and Wawrzoniak, M. (2004). Operating system support for planetary-scale network services. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 19–19, Berkeley, CA, USA. USENIX Association.

[Bavier et al., 2006] Bavier, A., Feamster, N., Huang, M., Peterson, L., and Rexford, J. (2006). In VINI veritas: realistic and controlled network experimentation. *SIGCOMM Comput. Commun. Rev.*, 36:3–14.

[Bellard, 2005] Bellard, F. (2005). QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA. USENIX Association.

[Bhatia et al., 2008] Bhatia, S., Motiwala, M., Muhlbauer, W., Mundada, Y., Valancius, V., Bavier, A., Feamster, N., Peterson, L., and Rexford, J. (2008). Trellis: a platform for building flexible, fast virtual networks on commodity hardware. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 72:1–72:6, New York, NY, USA. ACM.

[Bin Tariq et al., 2009] Bin Tariq, M., Mansy, A., Feamster, N., and Ammar, M. (2009). Characterizing VLAN-induced sharing in a campus network. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, IMC '09, pages 116–121, New York, NY, USA. ACM.

[Brehon et al., 2007] Brehon, Y., Kofman, D., and Casaca, A. (2007). Virtual Private Network to Spanning Tree Mapping. In Akyildiz, I., Sivakumar, R., Ekici, E., Oliveira, J., and McNair, J., editors, *NETWORKING 2007. Ad Hoc and Sensor Networks, Wireless Networks, Next Generation Internet*, volume 4479 of *Lecture Notes in Computer Science*, pages 703–713. Springer Berlin / Heidelberg.

[Busson et al., 2007] Busson, A., Kofman, D., and Rougier, J.-L. (2007). A new service overlays dimensioning approach based on stochastic geometry. *Perform. Eval.*, 64:76–92.

[Caesar and Rexford, 2008] Caesar, M. and Rexford, J. (2008). Building bug-tolerant routers with virtualization. In *PRESTO '08*, pages 51–56. ACM.

[Cappello et al., 2005] Cappello, F., Caron, E., Daydé, M., Desprez, F., Jégou, Y., Primet, P., Jeannot, E., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Quetier, B., and Richard, O. (2005). Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *Grid Computing, 2005. The 6th IEEE/ACM International Workshop on*, pages 8 pp.+.

[Chen et al., 2009] Chen, X., Mao, Z. M., and Van Der Merwe, J. (2009). ShadowNet: a platform for rapid and safe network evolution. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 3–3, Berkeley, CA, USA. USENIX Association.

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

[Chowdhury and Boutaba, 2009] Chowdhury, N. M. M. K. and Boutaba, R. (2009). Network virtualization: state of the art and research challenges. *Comm. Mag.*, 47:20–26.

[Congdon et al., 2010] Congdon, P., Fischer, A., and Mohapatra, P. (2010). A Case for VEPA: Virtual Ethernet Port Aggregator. In *ITC 22*.

[Das et al., 2010] Das, S., Parulkar, G., Singh, P., Getachew, D., Ong, L., and McKeown, N. (2010). Packet and Circuit Network Convergence with OpenFlow.

[De Carli et al., 2009] De Carli, L., Pan, Y., Kumar, A., Estan, C., and Sankaralingam, K. (2009). PLUG: flexible lookup modules for rapid deployment of new protocols in high-speed routers. In *SIGCOMM '09*. ACM.

[Dike, 2001] Dike, J. (2001). User-mode Linux. In *Proceedings of the 5th annual Linux Showcase & Conference - Volume 5*, pages 2–2, Berkeley, CA, USA. USENIX Association.

[Divakaran et al., 2009] Divakaran, D. M., Soudan, S., Vicat-Blanc Primet, P., and Altman, E. (2009). A survey on core switch designs and algorithms. Research Report RR-6942, INRIA.

[Divakaran and Vicat-Blanc Primet, 2007] Divakaran, D. M. and Vicat-Blanc Primet, P. (2007). Channel Provisioning in Grid Overlay Networks (short paper). In *Workshop on IP QoS and Traffic Control*.

[Dobrescu et al., 2009] Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. (2009). RouteBricks: exploiting parallelism to scale software routers. In *SOSP '09: Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM.

[Dong et al., 2008] Dong, Y., Yu, Z., and Rose, G. (2008). SR-IOV networking in Xen: architecture, design and implementation. In *Proceedings of the First conference on I/O virtualization*, WIOV'08, pages 10–10, Berkeley, CA, USA. USENIX Association.

[Duffield et al., 1999] Duffield, N. G., Goyal, P., Greenberg, A., Mishra, P., Ramakrishnan, K. K., and van der Merive, J. E. (1999). A flexible model for resource management in virtual private networks. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 95–108, New York, NY, USA. ACM.

[Egi et al., 2008a] Egi, N., Greenhalgh, A., Handley, M., Hoerdt, M., Huici, F., and Mathy, L. (2008a). Fairness issues in software virtual routers. In *PRESTO '08*, pages 33–38. ACM.

[Egi et al., 2008b] Egi, N., Greenhalgh, A., Handley, M., Hoerdt, M., Huici, F., and Mathy, L. (2008b). Towards high performance virtual routers on commodity hardware. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 20:1–20:12, New York, NY, USA. ACM.

[Egi et al., 2009] Egi, N., Greenhalgh, A., Hoerdt, M., Huici, F., Papadimitriou, P., Handley, M., and Mathy, L. (2009). A Platform for High Performance and Flexible Virtual Routers on Commodity Hardware. SIGCOMM 2009 poster session.

[Feamster et al., 2007] Feamster, N., Gao, L., and Rexford, J. (2007). How to lease the internet in your spare time. *SIGCOMM CCR*, 37(1).

[Feldmann, 2007] Feldmann, A. (2007). Internet clean-slate design: what and why? *SIGCOMM Comput. Commun. Rev.*, 37:59–64.

[Floyd and Jacobson, 1993] Floyd, S. and Jacobson, V. (1993). Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1:397–413.

[Fu and Rexford, 2008] Fu, J. and Rexford, J. (2008). Efficient IP-address lookup with a shared forwarding table for multiple virtual routers. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 21:1–21:12, New York, NY, USA. ACM.

[Glatard et al., 2008] Glatard, T., Montagnat, J., Lingrand, D., and Pennec, X. (2008). Flexible and efficient workflow deployement of data-intensive applications on grids with MOTEUR. *Int. Journal of High Performance Computing and Applications (IJHPCA)*, 22(3):347–360.

[Glatard et al., 2006] Glatard, T., Pennec, X., and Montagnat, J. (2006). Performance evaluation of grid-enabled registration algorithms using bronze-standards. In *Medical Image Computing and Computer-Assisted Intervention(MICCAI'06)*, LNCS 4191, pages 152–160, Copenhagen, Denmark. Springer.

[Greenberg et al., 2005] Greenberg, A., Hjalmtysson, G., Maltz, D. A., Myers, A., Rexford, J., Xie, G., Yan, H., Zhan, J., and Zhang, H. (2005). A clean slate 4D approach to network control and management. *SIGCOMM Comput. Commun. Rev.*, 35:41–54.

[Gude et al., 2008] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. (2008). NOX: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110.

[Handley, 2006] Handley, M. (2006). Why the Internet only just works. *BT Technology Journal*, 24(3):119–129.

[Handley et al., 2005] Handley, M., Kohler, E., Ghosh, A., Hodson, O., and Radoslavov, P. (2005). Designing extensible IP router software. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 189–202, Berkeley, CA, USA. USENIX Association.

[Hassayoun and Ros, 2009] Hassayoun, S. and Ros, D. (2009). Loss synchronization, router buffer sizing and high-speed TCP versions: Adding RED to the mix. In *Local Computer Networks, 2009. LCN 2009. IEEE 34th Conference on*, pages 569 –576.

[Hibler et al., 2008] Hibler, M., Ricci, R., Stoller, L., Duerig, J., Guruprasad, S., Stack, T., Webb, K., and Lepreau, J. (2008). Large-scale virtualization in the Emulab network testbed. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 113–128, Berkeley, CA, USA. USENIX Association.

[IaaS, 2008] IaaS (2008). Evaluation of current network control and management plane for multi-domain network infrastructure. FEDERICA Deliverable DJRA1.1.

[Iyer and McKeown, 2001] Iyer, S. and McKeown, N. (2001). Techniques for Fast Shared Memory Switches. Technical Report TR01-HPNG-081501, Stanford University HPNG.

[Jain et al., 1984] Jain, R. K., Chiu, D.-M. W., and Hawe, W. R. (1984). A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems. *ACM Transaction on Computer Systems*.

[Kamp and Watson, 2000] Kamp, P.-H. and Watson, R. N. M. (2000). Jails: Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*.

[Kanizo et al., 2009] Kanizo, Y., Hay, D., and Keslassy, I. (2009). The Crosspoint-Queued Switch. In *IEEE INFOCOM 2009*.

[Keller and Rexford, 2010] Keller, E. and Rexford, J. (2010). The "Platform as a Service" Model for Networking. In *INM/WREN '10*.

[Keller et al., 2009] Keller, E., Yu, M., Caesar, M., and Rexford, J. (2009). Virtually eliminating router bugs. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, CoNEXT '09, pages 13–24, New York, NY, USA. ACM.

[Kelly et al., 2010] Kelly, J., Araujo, W., and Banerjee, K. (2010). Rapid service creation using the JUNOS SDK. *SIGCOMM Comput. Commun. Rev.*, 40:56–60.

[Kim et al., 2010] Kim, W., Sharma, P., Lee, J., Banerjee, S., Tourrilhes, J., Lee, S.-J., and Yalagandula, P. (2010). Automated and scalable QoS control for network convergence. In *Proceedings of the 2010 INM/WREN*. USENIX Association.

[Kivity et al., 2007] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. (2007). kvm: the Linux Virtual Machine Monitor. In *Linux Symposium*.

[Kodama et al., 2004] Kodama, Y., Kudoh, T., Takano, R., Sato, H., Tatebe, O., and Sekiguchi, S. (2004). GNET-1: gigabit Ethernet network testbed. *Cluster Computing, IEEE International Conference on*, 0:185–192.

[Koh et al., 2006] Koh, Y., Pu, C., Bhatia, S., and Consel, C. (2006). Efficient Packet Processing in User-Level OSes: A Study of UML. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 63 –70.

[Kolon, 2004] Kolon, M. (2004). Intelligent Logical Router Service. Technical Report 200097-001, Juniper Networks.

[Koslovski et al., 2011] Koslovski, G., Soudan, S., Gonçalves, P., and Vicat-Blanc, P. (2011). Locating Virtual Infrastructures: Users and InP Perspectives. In *12th IEEE/IFIP International Symposium on Integrated Network Management - Special Track on Management of Cloud Services and Infrastructures (IM 2011 - STMCSI)*, Dublin, Ireland. IEEE.

[Koslovski et al., 2009] Koslovski, G., Truong Huu, T., Montagnat, J., and Vicat-Blanc Primet, P. (2009). Executing distributed applications on virtualized infrastructures specified with the VXDL language and managed by the HIPerNET framework. In *First International Conference on Cloud Computing (CLOUDCOMP 2009)*, Munich, Germany.

[Koslovski et al., 2008] Koslovski, G., Vicat-Blanc Primet, P., and Charão, A. S. (2008). VXDL: Virtual Resources and Interconnection Networks Description Language. In *GridNets 2008*.

[Laganier and Vicat-Blanc Primet, 2005] Laganier, J. and Vicat-Blanc Primet, P. (2005). HIPernet: A Decentralized Security Infrastructure for Large Scale Grid Environments. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 140–147, Washington, DC, USA. IEEE Computer Society.

[Laor, 2007] Laor, D. (2007). KVM Para-Virtualized Guest Drivers. KVM Forum 2007.

[Lee et al., 2010] Lee, D., Carpenter, B. E., and Brownlee, N. (2010). Observations of UDP to TCP Ratio and Port Numbers. In *Proceedings of the 2010 Fifth International Conference on Internet Monitoring and Protection*, ICIMP '10, pages 99–104, Washington, DC, USA. IEEE Computer Society.

[Lee et al., 2008] Lee, S.-J., Banerjee, S., Sharma, P., Yalagandula, P., and Basu, S. (2008). Bandwidth-Aware Routing in Overlay Networks. In *Proceedings of IEEE INFOCOM*, Phoenix, AZ.

[Liao et al., 2009] Liao, Y., Yin, D., and Gao, L. (2009). PdP: parallelizing data plane in virtual network substrate. In *Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, VISA '09, pages 9–18, New York, NY, USA. ACM.

[Liao et al., 2010] Liao, Y., Yin, D., and Gao, L. (2010). Europa: Efficietn User Mode Packet Forwarding in Network Virtualization. In *Proceedings of the 2010 INM/WREN*. USENIX Association.

[Louati et al., 2009] Louati, W., Houidi, I., and Zeghlache, D. (2009). Autonomic Virtual Routers for the Future Internet. In Nunzi, G., Scoglio, C., and Li, X., editors, *IP Operations and Management*, volume 5843 of *Lecture Notes in Computer Science*, pages 104–115. Springer Berlin / Heidelberg. 10.1007/978-3-642-04968-2_9.

[Lu et al., 2009] Lu, G., Shi, Y., Guo, C., and Zhang, Y. (2009). CAFE: a configurable packet forwarding engine for data center networks. In *Proceedings of the 2nd ACM SIGCOMM workshop on Programmable routers for extensible services of tomorrow*, PRESTO '09, pages 25–30, New York, NY, USA. ACM.

[McIlroy and Sventek, 2006] McIlroy, R. and Sventek, J. (2006). Resource Virtualisation of Network Routers. In *HPSR 06: 2006 Workshop on High Performance Switching and Routing*.

[McKeown et al., 2008] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). OpenFlow: enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74.

[Mell and Grance, 2009] Mell, P. and Grance, T. (2009). The NIST Definition of Cloud Computing.

[Melle et al., 2008] Melle, S., Dodd, R., Grubb, S., Liou, C., Vusirikala, V., and Welch, D. (2008). Bandwidth virtualization enables long-haul WDM transport of 40 Gb/s and 100 Gb/s services. *Communications Magazine, IEEE*, 46(2):S22 –S29.

[Menon et al., 2006] Menon, A., Cox, A. L., and Zwaenepoel, W. (2006). Optimizing network virtualization in Xen. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 2–2. USENIX Association.

[Mhamdi et al., 2006] Mhamdi, L., Kachris, C., and Vassiliadis, S. (2006). A reconfigurable hardware based embedded scheduler for buffered crossbar switches. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, FPGA '06, pages 143–149, New York, NY, USA. ACM.

[Mogul et al., 2008] Mogul, J. C., Yalagandula, P., Tourrilhes, J., McGeer, R., Banerjee, S., Connors, T., and Sharma, P. (2008). API Design Challenges for Open Router Platforms on Proprietary Hardware. In *Proceedings of the Seventh ACM Workshop on Hot Topics in Networks (HotNets-VII)*.

[Morris et al., 1999] Morris, R., Kohler, E., Jannotti, J., and Kaashoek, M. F. (1999). The Click modular router. *SIGOPS Oper. Syst. Rev.*, 33(5).

[MSTP, 2005] MSTP (2005). Multiple Spanning Trees, IEEE Std 802.1S. IEEE Computer Society.

[N. Egi et al., 2007] N. Egi et al. (2007). Evaluating Xen for Router Virtualization. In *ICCCN*, pages 1256–1261.

[Nakao et al., 2008] Nakao, A., Ozaki, R., and Nishida, Y. (2008). CoreLab: an emerging network testbed employing hosted virtual machine monitor. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 73:1–73:6, New York, NY, USA. ACM.

[Naous et al., 2008] Naous, J., Gibb, G., Bolouki, S., and McKeown, N. (2008). NetFPGA: reusable router architecture for experimental research. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, PRESTO '08, pages 1–7, New York, NY, USA. ACM.

[Ongaro et al., 2008] Ongaro, D., Cox, A. L., and Rixner, S. (2008). Scheduling I/O in virtual machine monitors. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 1–10. ACM.

[Pettit et al., 2010] Pettit, J., Gross, J., Pfaff, B., Casado, M., and Crosby, S. (2010). Virtual Switching in an Era of Advanced Edges. In *ITC 22*.

[Pfaff et al., 2009] Pfaff, B., Pettit, J., Koponen, T., Amidon, K., Casado, M., and Shenker, S. (2009). Extending Networking into the Virtualization Layer. In *HotNets-VIII*, 2nd Workshop on Data Center – Converged and Virtual Ethernet Switching (DC-CAVES).

[Popek and Goldberg, 1973] Popek, G. J. and Goldberg, R. P. (1973). Formal requirements for virtualizable third generation architectures. In *SOSP '73: Proceedings of the fourth ACM symposium on Operating system principles*, page 121, New York, NY, USA. ACM.

[Potter and Nakao, 2009] Potter, R. and Nakao, A. (2009). Mobitopolo: a portable infrastructure to facilitate flexible deployment and migration of distributed applications with virtual topologies. In *Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, VISA '09, pages 19–28, New York, NY, USA. ACM.

[Prasad et al., 2009] Prasad, R. S., Dovrolis, C., and Thottan, M. (2009). Router buffer sizing for TCP traffic and the role of the output/input capacity ratio. *IEEE/ACM Trans. Netw.*, 17:1645–1658.

[Pratt and Fraser, 2001] Pratt, I. and Fraser, K. (2001). Arsenic: a user-accessible gigabit Ethernet interface. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE.*

[Pratt et al., 2005] Pratt, I., Fraser, K., Hand, S., Limpach, C., Warfield, A., Magenheimer, D., Nakajima, J., and Mallick, A. (2005). Xen 3.0 and the Art of Virtualization. In *Proceedings of the Linux Symposium*, volume 2, pages 65–78.

[Puljiz and Mikuc, 2006] Puljiz, Z. and Mikuc, M. (2006). IMUNES Based Distributed Network Emulator. *International Conference on Software in Telecommunications and Computer Networks*, 0:198–203.

[Rosenberg and Mateos, 2010] Rosenberg, J. and Mateos, A. (2010). *The Cloud at Your Service*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

[Russell, 2008] Russell, R. (2008). virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103.

[Santos et al., 2008] Santos, J. R., Turner, Y., Janakiraman, G., and Pratt, I. (2008). Bridging the gap between software and hardware techniques for I/O virtualization. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 29–42, Berkeley, CA, USA. USENIX Association.

[Savage et al., 1999] Savage, S., Anderson, T., Aggarwal, A., Becker, D., Cardwell, N., Collins, A., Hoffman, E., Snell, J., Vahdat, A., Voelker, G., and Zahorjan, J. (1999). Detour: a case for informed internet routing and transport. *IEEE Micro*, 19:50–59.

[Schaffrath et al., 2009] Schaffrath, G., Werle, C., Papadimitriou, P., Feldmann, A., Bless, R., Greenhalgh, A., Wundsam, A., Kind, M., Maennel, O., and Mathy, L. (2009). Network virtualization architecture: proposal and initial prototype. In *Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, VISA '09, pages 63–72, New York, NY, USA. ACM.

[Schlosser et al., 2011] Schlosser, D., Duelli, M., and Goll, S. (2011). Performance Comparison of Hardware Virtualization Platforms. In *IFIP/TC6 NETWORKING 2011 (NETWORKING 2011)*, Valencia, Spain.

[Shafer, 2010] Shafer, J. (2010). I/O virtualization bottlenecks in cloud computing today. In *Proceedings of the 2nd conference on I/O virtualization*, WIOV'10, pages 5–5, Berkeley, CA, USA. USENIX Association.

[Sherwood et al., 2009] Sherwood, R., Gibb, G., Yap, K.-K., Apenzeller, G., Casado, M., McKeown, N., and Parulkar, G. (2009). FlowVisor: A Network Virtualization Layer. Technical Report OPENFLOW-TR-2009-1, OpenFlowSwitch.org.

[Sherwood et al., 2010] Sherwood, R., Gibb, G., Yap, K.-K., Appenzeller, G., Casado, M., McKeown, N., and Parulkar, G. (2010). Can the production network be the testbed? In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA. USENIX Association.

[Shreedhar and Varghese, 1995] Shreedhar, M. and Varghese, G. (1995). Efficient fair queueing using deficit round robin. *SIGCOMM CCR*, 25(4):231–242.

[Soltesz et al., 2007] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. (2007). Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41:275–287.

[Sommers et al., 2008] Sommers, J., Barford, P., Greenberg, A., and Willinger, W. (2008). An SLA perspective on the router buffer sizing problem. *SIGMETRICS Perf. Eval. Rev.*, 35.

[Song et al., 2010] Song, H., Kodialam, M., Hao, F., and Lakshman, T. V. (2010). Building scalable virtual routers with trie braiding. In *Proceedings of the 29th conference on Information communications*, INFOCOM'10, pages 1442–1450, Piscataway, NJ, USA. IEEE Press.

[SR-IOV, 2009] SR-IOV (2009). Achieving Fast, Scalable I/O for Virtualized Servers. Intel Corporation.

[STP, 2004] STP (2004). Media Access Control (MAC) Bridges, IEEE Std 802.1D-2004. IEEE Computer Society.

[Subramanian et al., 2003] Subramanian, L., Stoica, I., Balakrishnan, H., and Katz, R. H. (2003). OverQoS: offering Internet QoS using overlays. *SIGCOMM Comput. Commun. Rev.*, 33(1):11–16.

[Tarjan, 1983] Tarjan, R. E. (1983). *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[Tomioka et al., 2007] Tomioka, T., Hasegawa, G., and Murata, M. (2007). Router buffer re-sizing for short-lived TCP flows. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*.

[Touch, 2000] Touch, J. (2000). Dynamic Internet overlay deployment and management using the X-Bone. In *ICNP '00: Proceedings of the 2000 International Conference on Network Protocols*, page 59, Washington, DC, USA. IEEE Computer Society.

[Tripathi et al., 2009] Tripathi, S., Droux, N., Srinivasan, T., and Belgaied, K. (2009). Crossbow: from hardware virtualized NICs to virtualized networks. In *VISA '09: Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, pages 53–62, New York, NY, USA. ACM.

[Truong Huu and Montagnat, 2010] Truong Huu, T. and Montagnat, J. (2010). Virtual resources allocation for workflow-based applications distribution on a cloud infrastructure. In *2nd Int. Symp. on Cloud Computing*, Melbourne, Australia.

[Turner et al., 2007] Turner, J. S., Crowley, P., DeHart, J., Freestone, A., Heller, B., Kuhns, F., Kumar, S., Lockwood, J., Lu, J., Wilson, M., Wiseman, C., and Zar, D. (2007). Supercharging PlanetLab: A High Performance, Multi-Application, Overlay Network Platform. *SIGCOMM Comput. Commun. Rev.*, 37(4):85–96.

[Turner et al., 2003] Turner, M., Budgen, D., and Brereton, P. (2003). Turning Software into a Service. *Computer*, 36:38–44.

[Tutschku et al., 2011] Tutschku, K., Müller, P., and Tran, F. D. (2011). Special issue on network virtualisation: Concepts and performance aspects. *International Journal of Communication Networks and Distributed Systems (IJCNDS)*, 6(3).

[Tutschku et al., 2008] Tutschku, K., Tran-Gia, P., and Andersen, F.-U. (2008). Trends in network and service operation for the emerging future internet. *AEU - International Journal of Electronics and Communications*, 62(9):705–714.

[Unnikrishnan et al., 2010] Unnikrishnan, D., Vadlamani, R., Liao, Y., Dwaraki, A., Crenne, J., Gao, L., and Tessier, R. (2010). Scalable network virtualization using FPGAs. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, pages 219–228, New York, NY, USA. ACM.

[Vishwanath et al., 2009] Vishwanath, A., Sivaraman, V., and Thottan, M. (2009). Perspectives on router buffer sizing: recent results and open problems. *SIGCOMM CCR*, 39:34–39.

[VLAN, 2005] VLAN (2005). Virtual Bridged Local Area Networks, IEEE Std 802.1Q-2005. IEEE Computer Society.

[VMDq, 2008] VMDq (2008). Intel VMDq Technology, Notes On Software Design Support for Intel VMDq Technology. Intel Corporation.

[Wang and Ng, 2010] Wang, G. and Ng, T. S. E. (2010). The impact of virtualization on network performance of amazon EC2 data center. In *Proceedings of the 29th conference on Information communications*, INFOCOM'10, pages 1163–1171, Piscataway, NJ, USA. IEEE Press.

[Wang et al., 2008] Wang, Y., Keller, E., Biskeborn, B., van der Merwe, J., and Rexford, J. (2008). Virtual routers on the move: live router migration as a network-management primitive. In *SIGCOMM '08*, pages 231–242. ACM.

[Wei et al., 2009] Wei, W., Hu, J., Qian, D., Ji, P. N., Wang, T., Liu, X., and Qiao, C. (2009). PONIARD: A Programmable Optical Networking Infrastructure for Advanced Research and Development of Future Internet. *J. Lightwave Technol.*, 27(3):233–242.

[Whitaker et al., 2002] Whitaker, A., Shaw, M., and Gribble, S. (2002). Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In *Proceedings of the 2002 USENIX Annual Technical Conference*.

[White et al., 2002] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An integrated experimental environment for distributed systems and networks. *SIGOPS Oper. Syst. Rev.*, 36:255–270.

[Wu et al., 2006] Wu, J., Savoie, M., Campbell, S., Zhang, H., and Arnaud, B. S. (2006). Layer 1 virtual private network management by users. *Communications Magazine, IEEE*, 44(12):86 –93.

[Xia et al., 2009] Xia, L., Lange, J., Dinda, P., and Bae, C. (2009). Investigating virtual passthrough I/O on commodity devices. *SIGOPS Oper. Syst. Rev.*, 43:83–94.

[Yin et al., 2010] Yin, D., Unnikrishnan, D., Liao, Y., Gao, L., and Tessier, R. (2010). Customizing virtual networks with partial FPGA reconfiguration. In *Proceedings of the second ACM SIGCOMM workshop on Virtualized infrastructure systems and architectures*, VISA '10, pages 57–64, New York, NY, USA. ACM.

[Yu et al., 2008] Yu, M., Yi, Y., Rexford, J., and Chiang, M. (2008). Rethinking virtual network embedding: substrate support for path splitting and migration. *SIGCOMM CCR*, 38(2).

[Zeng and Hao, 2009] Zeng, S. and Hao, Q. (2009). Network I/O Path Analysis in the Kernel-Based Virtual Machine Environment through Tracing. In *Proceedings of the 2009 First IEEE International Conference on Information Science and Engineering*, ICISE '09, pages 2658–2661, Washington, DC, USA. IEEE Computer Society.

[Zhang and Loguinov, 2008] Zhang, Y. and Loguinov, D. (2008). ABS: Adaptive Buffer Sizing for Heterogeneous Networks. In *Quality of Service, 2008. IWQoS 2008. 16th Int. Workshop on*, pages 90 –99.

[Zinner et al., 2010] Zinner, T., Tutschku, K., Nakao, A., and Tran-Gia, P. (2010). Using Concurrent Multipath Transmission for Transport Virtualization: Analyzing Path Selection. In *Proceedings of the 22nd International Teletraffic Congress (ITC)*, Amsterdam, Netherlands.

[Zitoune et al., 2009] Zitoune, L., Hamdi, A., Mounier, H., and Veque, V. (2009). Flatness-based controller for expressive-based SLA applications in Internet Computing. In *Control Applications, (CCA) Intelligent Control, (ISIC), 2009 IEEE*, pages 31 –36.

Standards, Recommendations & RFCs

[Awduche et al., 2001] Awduche, D., Berger, L., Gan, D., Li, T., Srinivasan, V., and Swallow, G. (2001). RSVP-TE: Extensions to RSVP for LSP Tunnels. RFC 3209 (Proposed Standard). Updated by RFCs 3936, 4420, 4874, 5151, 5420.

[Berger, 2003] Berger, L. (2003). Generalized Multi-Protocol Label Switching (GMPLS) Signaling Functional Description. RFC 3471 (Proposed Standard). Updated by RFCs 4201, 4328, 4872.

[Claise, 2004] Claise, B. (2004). Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational).

[Doria et al., 2010] Doria, A., Hadi Salim, J., Haas, R., Khosravi, H., Wang, W., Dong, L., Gopal, R., and Halpern, J. (2010). Forwarding and Control Element Separation (ForCES) Protocol Specification. RFC 5810 (Standards Track).

[Farinacci et al., 2000] Farinacci, D., Li, T., Hanks, S., Meyer, D., and Traina, P. (2000). Generic Routing Encapsulation (GRE). RFC 2784 (Proposed Standard). Updated by RFC 2890.

[Fedyk et al., 2008] Fedyk, D., Rekhter, Y., Papadimitriou, D., Rabbat, R., and Berger, L. (2008). Layer 1 VPN Basic Mode. RFC 5251 (Proposed Standard).

[Ford et al., 2011] Ford, A., Raiciu, C., Handley, M., Barre, S., and Iyengar, J. (2011). Architectural Guidelines for Multipath TCP Development. RFC 6182 (Informational).

[Hinden, 2004] Hinden, R. (2004). Virtual Router Redundancy Protocol (VRRP). RFC 3768 (Draft Standard).

[Kent and Atkinson, 1998] Kent, S. and Atkinson, R. (1998). Security Architecture for the Internet Protocol. RFC 2401 (Proposed Standard). Obsoleted by RFC 4301, updated by RFC 3168.

[Khosravi and Anderson, 2003] Khosravi, H. and Anderson, T. (2003). Requirements for Separation of IP Control and Forwarding. RFC 3654 (Informational).

[Lasserre and Kompella, 2007] Lasserre, M. and Kompella, V. (2007). Virtual Private LAN Service (VPLS) Using Label Distribution Protocol (LDP) Signaling. RFC 4762 (Proposed Standard).

[Muthukrishnan and Malis, 2000] Muthukrishnan, K. and Malis, A. (2000). A Core MPLS IP VPN Architecture. RFC 2917 (Informational).

[Phaal et al., 2001] Phaal, P., Panchen, S., and McKee, N. (2001). InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176 (Informational).

[Rosen and Rekhter, 1999] Rosen, E. and Rekhter, Y. (1999). BGP/MPLS VPNs. RFC 2547 (Informational). Obsoleted by RFC 4364.

[Rosen et al., 2001] Rosen, E., Viswanathan, A., and Callon, R. (2001). Multiprotocol Label Switching Architecture. RFC 3031 (Proposed Standard).

[Simpson, 1995] Simpson, W. (1995). IP in IP Tunneling. RFC 1853 (Informational).

[Yang et al., 2004] Yang, L., Dantu, R., Anderson, T., and Gopal, R. (2004). Forwarding and Control Element Separation (ForCES) Framework. RFC 3746 (Informational).

Glossary

AVR	Active Virtual Router	36
BTR	Bug Tolerant Router	35
CB	Crosspoint Buffer	81
CE	Customer Edge	17
Click	Programmable modular software router	28
DRR	Deficit Round Robin	79
DVC	Digital Virtual Concatenation	25
ForCES	Forwarding and Control Element Separation	29
FPGA	Field Programmable Gate Array	36
GMPLS	Generalized Multi-Protocol Label Switching	18
Grid'5000	French national Grid and Cloud research platform	119
HIPerNet	Combined virtual IT and Network resource deployment framework	122
HTB	Hierarchy Tocken Bucket	21
HVM	Hardware Virtual Machine	22
IaaS	Infrastructure as a Service	116
IPsec	Internet Protocol security	17
KVM	Kernel Virtual Machine	22, 50
LSP	Label Switched Path	18
LSR	Label Switched Router	18
MPLS	Multi-Protocol Label Switching	17
NIC	Network Interface Card	20, 24
OCCI	Open Cloud Computing Interface	45
OFDM	Orthogonal Frequency Multiplexing	25
OFDMA	Orthogonal Frequency Multiplexing Access	25
OpenFlow	Open protocol for programming a flowtable of a switch	29, 105
OS	Operating System	20
PaaS	Platform as a Service	116
PE	Provider Edge	17
PoP	Point of Presence	31
QoS	Quality of Service	18
SaaS	Software as a Service	116
SDK	Software Development Kit	28
SP	Strict Priority	79
SR-IOV	Single Root In Out Virtualiation	25
STP	Spanning Tree Protocol	16, 33

TCAM	Ternary Content Addressable Memory	29
UML	User Mode Linux	21
VCS	Virtual Cluster Switching	32
vDS	virtual Network Distributed Switch	32
VI	Virtual Infrastructure	120
VINI	Virtual Network Infrastructure	43
virtio	Virtualized I/O driver	22
VLAN	Virtual Local Area Network	15
VMDq	Virtual Machine Device queues	24
VPIO	Virtual Passthrough I/O	23
VPLS	Virtual Private LAN Service	18
VPN	Virtual Private Network	16
VQM	Virtual Queue Manager	78
VRF	Virtual Routing and Forwarding	30
VROOM	Virtual ROuters On the Move	42
VRRP	Virtual Router Redundancy Protocol	30
VS	Virtual Switch	74
VXB	Virtual Crosspoint Buffer	74
VXDL	Virtual Infrastructure Description Language	122
VxLink	Virtual Link	96
VxRouter	Virtual Router	96
VxSwitch	Virtualized switching fabric	70
XaaS	Everything as a Service	116
Xen	Virtual machine hypervisor	22, 50