Master's Degree in Embedded Computing Systems

Design and Implementation of a Performance Testing Framework for High-Performance Inter-Container Communications

Gabriele Ara

October 14, 2019

Prof. Tommaso Cucinotta
TeCIP Institute
Scuola Superiore Sant'Anna

Academic Year 2018/2019

Abstract

As an emerging technology and business paradigm, cloud computing has seen a stable growth in the past few years, becoming one of the most interesting approaches to high-performance computing. Thanks to the high flexibility of these platforms, more applications get redesigned every day to follow distributed computing models.

In the domain of network operators, recent technological trends led to replacing traditional physical networking infrastructures with more flexible cloud-based systems, which can be dynamically instantiated on demand to provide the required level of service performance when needed. In this context, the paradigm represented by Network Function Virtualization (NFV) aims to replace most of the highly specialized hardware appliances that traditionally would be used to build a network infrastructure with software-based Virtualized Network Functions (VNFs), which are equivalent implementations of the same services provided in software instead of hardware. This brings new flexibility in physical resources management and allows the realization of more dynamic networking infrastructures.

In this context, a number of network functions need high-performance and low end-to-end latency, where a key role is played by the communication overheads experienced by the individual software components participating in each deployed VNF. Such requirements are so tight that NFV has already moved on from traditional Virtual Machines (VMs) to Operating System (OS) containers to deploy VNFs on the designated infrastructure. Primary research focus is now into reducing per-packet processing overheads by using user-space networking techniques, allowing applications to avoid the kernel when exchanging data between containers, either on the same machine or between different hosts. These techniques are generally indicated as *kernel bypass* mechanisms.

Contribution

In this thesis, a benchmarking framework has been designed and realized, for the purpose of comparing different *kernel bypass* mechanisms that can be used to exchange data between VNFs deployed on OS containers within a private cloud infrastructure, to determine which is the most suitable to

build efficient network infrastructures in the cloud. Among these mechanisms, this work focuses on the evaluation of the Data Plane Development Kit (DPDK) framework and other tools that are built on top of it (e.g. software virtual switches), as DPDK occupies a prominent position in the industry and provides most of the functionalities needed to bypass the kernel when exchanging network packets, either locally or with an actual hardware Network Interface Controller (NIC). Evaluations are done comparing software virtual switches against a Single-Root I/O Virtualization (SR-IOV) enabled network card, which provides a hardware implementation of local switching functionalities. This study compares the performance achieved by each different solution with respect to a number of key metrics, namely network throughput, latency and scalability, when subject to synthetic workloads simulating the behavior of real VNF components communicating with each other.

Ringraziamenti

Innanzitutto vorrei esprimere un ringraziamento particolare per i miei genitori, che in tutti questi anni mi hanno sempre incoraggiato a dare il massimo e che mi hanno supportato nei momenti più difficili. Assieme a loro vorrei poi ringraziare tutto il resto della mia famiglia, che con il loro affetto mi ha permesso di superare diversi periodi bui in questi anni di crescita personale e professionale. Grazie Annalisa e Alessia, per essermi sempre state vicine come due sorelle.

Non avrei mai raggiunto questo traguardo senza tutti gli amici che sin dall'inizio del mio percorso universitario mi sono stati più vicini, condividendo con me molti momenti importanti. Grazie quindi a Gabriele, Gabriele e Filippo, che più di tutti sono stati al mio fianco in questi ultimi anni. Grazie a Chiara, Mattia, Lorenzo, Daniele, Alessio e Francesca, per la loro compagnia. Grazie a Salvatore, Roberto, Silvio, Andrea, Giovanni e Michele, perchè senza la loro simpatia le lezioni della magistrale non sarebbero state le stesse.

Un enorme grazie a Courtney, per farmi sentire sempre la sua vicinanza nonostante la distanza che ci separa. Non la ringrazierò mai abbastanza per i suoi infiniti incoraggiamenti e l'affetto che mi ha dimostrato in questi ultimi anni.

Ringrazio il Prof. Cucinotta, per la fiducia che mi ha riservato durante tutti questi mesi di lavoro sulla tesi. Per tutti gli spunti, gli incoraggiamenti e gli stimoli a lavorare sempre meglio e sempre di più in questi ultimi mesi, ringrazio tutti al RETIS Lab. In particolare, senza Daniel, Daniel, Paolo, Francesco, Marco, Agostino e Biruk quest'ultimo percorso non sarebbe stato lo stesso. Inoltre, è doveroso ringraziare Ericsson, che ha finanziato questo lavoro di ricerca.

Infine, vorrei ringraziare tutte le persone che mi hanno accompagnato fino a qui e che, per un verso o per un altro, hanno contribuito a modellare la persona che sono oggi. Anche se le nostre strade si sono divise in passato, il vostro contributo rimarrà per sempre con me.

Contents

List of Tables

List of Listings

List of Figures

Chapter 1

Introduction

Over the past few years, more and more applications shifted to a distributed computing paradigm to provide lots of new services, thanks to the widespread availability of affordable high-speed Internet connections. In a relatively short period of time, the Internet has become a critical infrastructure for global commerce, media and other services. However, the increased complexity of modern network infrastructures represents an obstacle to their evolution into more flexible and dynamic systems.

In the current Internet, IP datagrams have to get through a number of intermediate nodes whose purpose is no longer simply forwarding them until they reach their final destination. Instead these middle-boxes are responsible for implementing a great number of other features directly within the network infrastructure itself, such as address translation, packet inspection, filtering, Quality of Service (QoS) management, and so on. This way, network operators try to offer a greater number of services at the lowest level, removing the need for developers to support some features at a higher level of the stack, usually in the application layer.

These new services are traditionally supported in a network infrastructure by introducing highly specialized network appliances that rely on custom hardware to inspect and transform IP datagrams as they traverse them towards their destination. However, the adoption of this approach has increased the resilience of network infrastructures to the introduction of disruptive new technologies (like would be for example a new transport-level protocol), due to the proprietary nature of existing hardware appliances (which may modify not only the header but also the content of an IP datagram with their processing, dipping into contents of higher-level protocols) and the cost of offering the space and energy to host a variety of middle-boxes within a network infrastructure. This contributes to a phenomenon called Internet ossification, which means that it is difficult to introduce new changes to the current state of the whole Internet infrastructure [2].

In this context, an interesting approach to tackling the problem of In-

ternet ossification implies the use of virtualization techniques to transition from a mostly hardware-oriented approach to a software-oriented one when implementing new network functions. This is done mostly to leverage the characteristics of modern cloud infrastructures, that are able to provide high levels of flexibility in resource management, especially for those applications that are subject to big variations in service demand over time. Given the cost-effectiveness of this approach, cloud computing solutions are progressively replacing traditional dedicated infrastructure management.

In the case of network operators, the adoption of the cloud model to introduce new functions to be implemented within the network led to the introduction of Network Function Virtualization (NFV). NFV aims to move those functionalities that were once implemented with middle-boxes to software applications that can run on off-the-shelf programmable hardware (i.e. industry standard servers, storage, and switches) within a private cloud infrastructure. This can solve a whole group of problems that traditionally affect network operators, like dimensioning correctly the whole infrastructure.

Since service unavailability is typically thought unacceptable, network carriers usually overprovision their services [3], thus the utilization of the overall allocated resources is normally low, both because of the fluctuations in demand of traffic and the offered redundancy in the case of service failure.

Taking advantage of the cloud model, the network infrastructure could dynamically resize itself depending on the amount of traffic that needs to be processed, concentrating the workload on a reduced number of servers and allowing the rest to be either turned off to save energy or be used to provide other services, like general web services, content and software distribution, etc.

Since the overall goal is to implement in software virtual functions to be executed through virtualization technologies on general-purpose servers, one of the most important aspects to be considered is whether performance, such as throughput and latency, is affected by this transition from custom hardware appliances. For many of the network functions, the performance of virtualized network appliances is the first concern [4].

Traditional cloud-based platforms rely on full virtualization, through the use of Virtual Machines (VMs), to provide a standard environment for each application running in the cloud, such as network functions, and to abstract the actual organization of the infrastructure itself. This allows for high levels of manageability of the whole virtual infrastructure, with the possibility to dynamically allocate and scale hardware resources for each virtual machine and even migrate them from one host to another without

disrupting the continuity of service, thanks to live migration.

To achieve these results it is also necessary to attain resource separation with a sufficient level of isolation. This can impact the overall performance of each network function, especially when performing operations that involve shared hardware resources like I/O devices and peripherals. Full virtualization techniques, which rely on Virtual Machines (VMs) to achieve isolation and virtualization, can result in significant slowdowns compared to performance of bare-metal applications, due to big per-packet overheads when exchanging data among VMs. While these costs can be better amortized using bigger packets, most applications that exchange very small packets suffer unbearable costs when deployed within VMs.

Solutions like hardware-assisted virtualization and para-virtualization can improve the overall performance of applications inside virtual machines, but most systems that adopt NFV paradigm prefer to adopt Operating System (OS) containers as virtualization providers (e.g. Docker [5], LXC [6], etc.), as they represent a more lightweight solution to the virtualization problem, despite introducing some limitations with respect to actual virtual machines [7].

Moreover, in order to reduce even further the per-packet processing overheads, and at the same time allow for the maximum flexibility in terms of packet processing by each Virtualized Network Function (VNF), plenty of experimentation is being done on the use of user-space networking, as opposed to traditional TCP/IP based management of network packets within an OS kernel or hypervisor. For this purpose, a number of different *kernel bypass*[1] techniques have been introduced over the years, among which a prominent position in the industry is played by the Data Plane Development Kit (DPDK) [8], which is a set of data plane libraries and drivers developed by Intel for fast packet processing in user-space.

Considering in these scenarios more than one VNF may be allocated on the same host, a key functionality that needs to be preserved is the virtual switching among multiple containers within the same host, both for local communications (between two containers on the same host) and remote ones (two containers located on different hosts). This led to the introduction of a number of software network switches implementations (e.g. Open vSwitch), as well as the introduction of hardware support for virtual switching, like the one provided by Single-Root I/O Virtualization (SR-IOV) Ethernet controllers [9].

[1]For more info see https://lwn.net/Articles/629155/

1.1 Problem Addressed in This Work

Since there are a variety of different solutions that may be considered when building an infrastructure using the VNF paradigm, we have witnessed an increasing interest over the past few years in the analysis of the actual performance that these solutions are able to achieve with respect to each other, to guide network operators towards this or that choice when moving to the new paradigm. The goal of this work is to provide a tool that can be easily deployed on a private cloud infrastructure to evaluate the actual performance that different virtual switching solutions (either software or hardware) are able to achieve with respect to each other.

Part of the contents of this Thesis are also included in a paper [1] presented at the 14th Workshop on Virtualization in High-Performance Cloud Computing (VHPC 2019), as part of the International Supercomputing Conference - High Performance (ISC 2019), held on June 20, 2019 in Frankfurt, Germany. In this occasion, this work stimulated the interest of the audience, proving its importance for representative members of both HPC industry and academic community.

1.2 Document Outline

This thesis is organized as follows. Chapter 2 describes the characteristics of NFV, the advantages with respect to legacy middle-boxes and the challenges that it introduces when moving from hardware to software-based network functions. Chapter 3 illustrates various techniques that can be used to achieve high performance communications between virtualized environments, with particular focus on high-performance oriented solutions. Chapter 4 shows the characteristics of the framework that has been developed to evaluate the performance of various virtual switching solutions when applied in NFV scenarios. Chapter 5 contains the experimental results obtained using said framework in a real use-case. Finally, Chapter 6 presents our conclusions and shows the improvements planned in our future work.

Chapter 2

Network Function Virtualization

In this chapter we describe more in detail the concepts behind Network Function Virtualization (NFV), the motivation behind its introduction, the terminology introduced by NFV used throughput this document, and what are its advantages and the challenges that need to be faced when adopting to this new approach to build a network infrastructure.

2.1 Motivation Behind NFV

In traditional networking infrastructures there are generally a lot of forwarding or processing devices that transmit, transform, filter, inspect or control network traffic for the purpose of network control and management [10], generically called middle-boxes. Examples of such devices are Network Address Translators (NATs), firewalls, Intrusion Detection Systems (IDS) etc. A more detailed taxonomy of middle-boxes can be found in [11]. Due to the heterogeneity of services in use within a network, the number of middle-boxes is increasing constantly over the years. Recent studies showed that for many enterprise networks, differing in network sizes, the number of middle-boxes is comparable to its number of hosted routers [12]. All these proprietary devices require a lot of management effort to effectively be configured, upgraded, monitored and integrated with all the other components of the network infrastructure via long and complex deployment processes, which inevitably increases the time needed to launch new network services [13].

In addition to the growing amount of middle-boxes, their characteristics represent a problem for the dynamicity of the network infrastructure, since they need to be physically placed in between other components and they cannot be easily moved or shared, let alone scaled depending on the dynamic needs of the network traffic. This contributes more and more to networks ossification and represents an obstacle to their flexibility [2] and forces network operators to overprovision their services, achieving usually

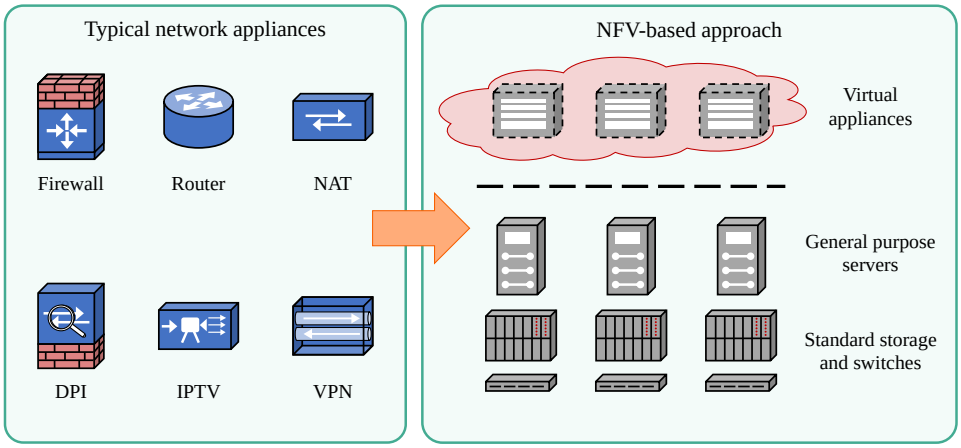


Figure 2.1: NFV main goal is to transition from complex physical network equipment to equivalent virtual appliances deployed on a cloud.

very low utilization of allocated resources [3].

Network Function Virtualization (NFV) aims to improve the flexibility of network services by leveraging virtualization technologies and commercial off-the-shelf programmable hardware, such as general-purpose servers, storage and switches [14]. The main goal of NFV is to decouple the software implementation of network functions from the underlying hardware by decoupling functionality from location (to some extent, and compatibly with the latency constraints) for faster networking service provisioning [15], as shown in Fig. 2.1.

Since NFV implements network functions through software virtualization techniques and runs them on commodity hardware (i.e. standard servers, storage, and switches), these virtual appliances can be instantiated on demand without the installation of new equipment, thus reducing the expenditure needed to upgrade or maintain the whole network infrastructure. This would definitely benefit network service providers, which need to periodically scale up their physical infrastructure, incurring in high Capital Expenditures (CAPEX) and Operating Expenses (OPEX) [16]. Due to the separation of network function from hardware, NFV can effectively reduce both CAPEX and OPEX. With NFV, the infrastructure can be usually dimensioned on the aggregate demand, which is more stable as it can leverage phenomena like statistical multiplexing, and dynamically scale it on demand using techniques typically adopted in cloud computing.

For this purpose, general-purpose Commercial Off-the-Shelf (COTS) network equipment (e.g., x86 based hardware), can provide far more capac-

ities than required with less cost than specialized network equipment [13]. Many works demonstrated over the past few years that it is feasible to implement network functions on general-purpose processor-based platforms, for example, for physical layer signal processing [17] and components in cellular core networks [18].

The introduction of NFV is also motivated by other situations (e.g. the difficulty to provide reliable IP multicast services on heterogeneous networking infrastructures, functional split for C-RAN scenarios, etc.), however it is out of the scope of this document to show them all. For other purposes refer to European Telecommunications Standards Institute (ETSI) white papers [15, 19].

2.2 Key Concepts and Characteristics

Given the problematic situation of the industry related to traditional middleboxes integration and management, over twenty of the world's largest telecommunication operators, including American Telephone & Telegraph (AT&T), British Telecom (BT), Deutsche Telekom (DT), China Mobile, Orange, Telefónica, Verizon, and others, formed an Industry Specification Group (ISG) within the European Telecommunications Standards Institute (ETSI) to define Network Function Virtualization (NFV) in 2012 [3]. Over the years, the ISG has grown into a large community, spanning over four working groups and two expert groups: Infrastructure Architecture, Management & Orchestration, Software Architecture, Reliability & Availability, Security, and Performance & Portability.

After the initial NFV specification [15], updated and improved over the years [20, 21], ETSI has published several specifications and guidelines for network operators that intend to move to a NFV-based infrastructure. In particular, they published works regarding management and orchestration [22], architectural framework [23], infrastructure overview [24] (including descriptions of compute domain [25], hypervisor domain [26] and network domain [27], service quality metrics [28], resiliency [29], and security and trust [30].

2.2.1 Main Terminologies

In this section we present the main terminologies introduced with NFV that are also used throughput this document:

Physical Network Function (PNF) It is the implementation of a specialized network function via a tightly coupled software and hardware

system. This term can be used either to refer to a network node or to other physical appliance.

Network Function Virtualization Infrastructure (NFVI) It is an environment in which network components, either hardware or software, can be deployed, managed and executed. A single Network Function Virtualization Infrastructure (NFVI) can span over multiple geographic locations and the connections between these places are also part of the NFVI itself.

Management and Orchestration (MANO) The Management and Orchestration (MANO) is the entity that is in charge of managing the new capabilities introduced by NFV inside the network infrastructure. In particular it is responsible for NFVI management, resource allocation and configuration, function virtualization, etc.

Virtualized Network Function (VNF) A VNF is a software implementation of a network function, which can be deployed on the NFVI. The VNF shall provide an equivalent functional behavior with respect to a physical implementation and it may be composed of one or more components. Each component is usually deployed as a VM, which means that the VNF can be deployed on a single VM if composed by only one component. Components for each VNF can be deployed independently across the NFVI.

Network Point of Presence (N-PoP) A Network Point of Presence (N-PoP) represents a single location where Physical Network Functions (PNFs) and VNFs can be implemented/deployed.

The relationships among these terminologies are shown in a graphical way in Fig. 2.2, in which they are placed in a layered structure.

The overall goal of NFV is to have a whole NFVI where PNFs are represented only by industry-standard L2/L3 switches, which mostly support only conventional or other kinds of routing protocols (e.g. OpenFlow), and all other network functions are implemented with VNFs. Of course, it is unreasonable to expect for an entire infrastructure to have all specialized network functions implemented as VNFs overnight, thus the coexistence between VNFs and PNFs is inevitable [13]. It is a duty of MANO to maintain a global view of the whole infrastructure and coordinate both PNFs and VNFs for fast and cost-effective service provision.

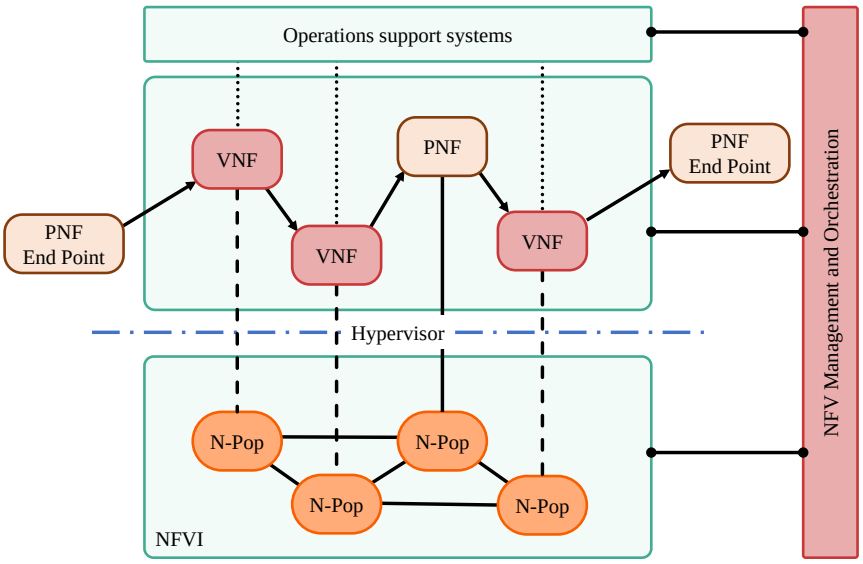


Figure 2.2: The relationship of different terminologies within an end-to-end service.

2.2.2 NFV Architecture

In networking infrastructures we often have a differentiation between what are called the data plane and the control plane. The former is composed of all the resources to run all network services, where each function is responsible for data forwarding and processing. The latter is made of all the orchestrators and the actors within the infrastructure that take decision about traffic management, resource allocation, etc. In NFV, the data plane corresponds to the NFVI, while the control plane corresponds to MANO. In addition, since most actual components within the infrastructure are virtualized, a network based on NFV requires a virtualization layer, which hosts various kinds of VNFs hosted as virtualized components.

The complete architectural scheme of a NFV-based network is depicted in Fig. 2.3, where relationships between all components have been highlighted.

Network Function Virtualization Infrastructure

NFVI provides fundamental services for fulfilling the objectives of NFV [24]. To do that, a set of general-purpose network devices are deployed in distributed locations, depending on the requirements in terms of both latency

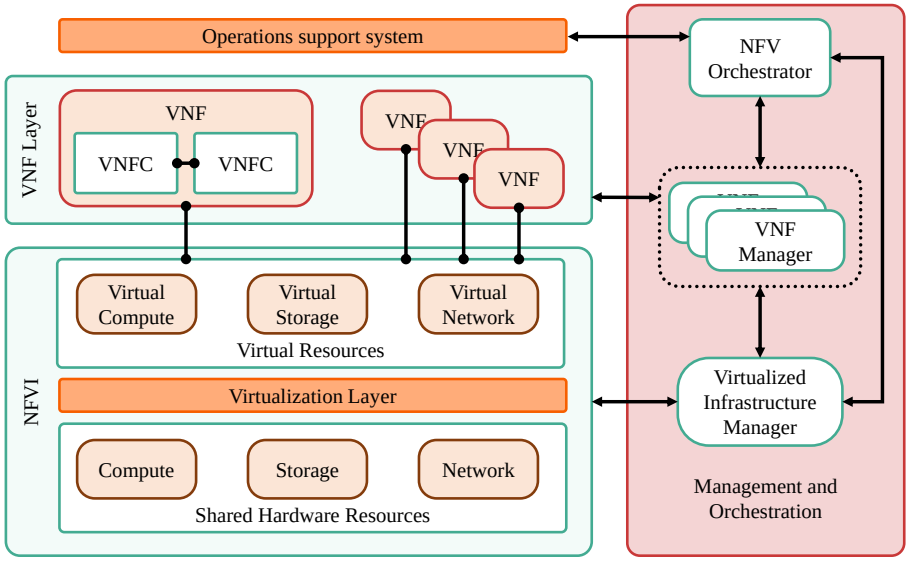


Figure 2.3: ETSI NFV Reference Architecture.

and throughput, on which VNFs are deployed in virtualized environments. Starting from this statement it is natural to divide the whole infrastructure in three distinct layers, that is, physical infrastructure layer, the virtualization layer and the virtual infrastructure layer, as shown in Fig. 2.3.

Physical Infrastructure Layer It is the set of all general-purpose servers that provide the computation and storage capacities for all the network services. In particular, we usually distinguish between compute nodes and storage nodes, interconnected by means of networking interfaces. Compute nodes are represented by general purpose servers, realized in the form of single-core or multi-core processors. Storage nodes are devices capable of storing information temporally or permanently, and they can be separate devices with respect to compute nodes, often implemented through Network Attached Storage (NAS) or Storage Area Network (SAN). Finally. Networking hardware is composed ideally of industry-standard L2/L3 switches, providing just basic forwarding functionalities for the more complex functions, which are virtualized, although in a transitory phase some PNF may still be present within the network, waiting to be replaced by equivalent software implementations.

Virtualization Layer It is the software platform on which all VNFs are

deployed, each within one or more isolated environments (e.g. VM or containers). It is usually constituted by hypervisors (e.g. Linux KVM [31], Citrix Hypervisor (formerly XenServer) [32], VMware vSphere [33], etc.) or other forms of OS-level virtualization platforms, (e.g. Docker [5], LXC [6], etc.). The difference between the two, explained more in detail in Section 3.1, is that the latter do not need to instantiate completely separated systems to run virtualized applications: virtual functions run directly on the host OS and they are isolated by means of access control policies like control groups and namespaces. In that they are able to save resources and reduce the overhead on the hosted applications, which can run faster, but they are also subject to other security issues due to the lack of actual isolation between VNFs (see Section 3.1.2).

Virtual Infrastructure Layer It is the collection of all virtual resources (i.e. virtual compute, storage, and networking) that are allocated to each VNF from the bottom layers. These resources are created from the allocation of actual physical resources from the hypervisor or other management software, which allow resources to be managed in virtual pools and assigned to virtual functions on demand. Examples of these resources are compute resources (i.e. CPUs), storage (e.g. by means of SAN, NAS, etc.), and networking equipment. The latter is particularly relevant for the performance of most virtual functions, which often need to exchange data between components efficiently both in terms of throughput and latency. Virtual networking is similar to traditional computer networking, but its implementation is mostly software-driven, usually in the form of virtual Ethernet adapters and virtual switches. For more details refer to Chapter 3.

Although the basic structure and functionalities associated with NFVI are well outlined by ETSI NFV ISG, implementations may differ greatly from each other, since the specifications do not enforce any precise indication about how infrastructure requirements should be achieved [13].

Management and Orchestration (MANO)

The role of NFV Management and Orchestration (MANO) is to act as a manager for the entire virtualized context within the network infrastructure. In particular, this includes managing all hypervisors and virtualization mechanisms, resources allocation, life cycle management of VNF instances, etc. According to ETSI, there are three main responsibilities assigned to MANO within NFV: the Virtual Infrastructure Manager (VIM),

VNF Managers (VNFMs), and the NFV Orchestrator (NFVO). In particular, NFV Orchestrator (NFVO) is mainly responsible for orchestrating NFVI resources and managing VNF life cycles, which are chained according to NFVO specification to provide the requested network service. Of course in terms of performance this organization must be performed taking into account the optimal VNF placements and forwarding paths that are formed to provide the demanded service.

Depending on the instructions of the NFVO, the Virtual Infrastructure Manager (VIM) and the VNF Managers (VNFMs) comply by managing respectively how hardware resources are allocated to each virtualized environment and where and how VNFs and Virtual Network Function Components (VNFCs) are deployed within the virtualized infrastructure. More in details, VNFMs are responsible for VNF instantiation, updating, migration and termination, while the VIM manages and controls NFVI resources, such as network, compute and storage.

Some responsibilities of NFVO, VNFM and VIM are partially overlapping, which means that actual implementations may differ in how these features are distributed between these three components. The specification describes the three as tightly coupled components that interact with each other to provide the aforementioned functionality.

Virtualized Network Functions

NFV is intended to abstract PNFs and finally implement them in the form of software (i.e. VNF), which means that all the other parts of the infrastructure are necessary to support the actual virtual functions that shall be deployed. The structure of the VNF layer is then composed by a collection of many isolated VNF instances, as illustrated in the top of Fig. 2.3. In the figure it is also shown how some VNFs may also be implemented as a collection of multiple Virtual Network Function Components (VNFCs), which can be independently deployed on isolated environments and together provide the combined functionalities of the whole VNF. All VNFs and VNFCs are managed by MANO, which is responsible for VNFs instantiation, update, query, scaling and termination.

In principle, all network functions and other network elements can be considered for virtualization. As PNFs provide network functions in the physical network, VNFs play the same role within the virtual network environment, thus can be organized and interconnected to make up the desired service chain. However, VNFs can be chained even when deployed on different positions in the physical network, and the resulting network is thus more flexible and dynamic, since VNFs location can be dynamically

selected depending on the current requirements of the network users.

It is important to notice that VNFs may have a number of different roles within a network infrastructure, not all of them related to the data plane. For example, some VNFs may act as software-based controllers and monitors on behalf of network orchestration services. For the rest of our dissertation however we will refer only VNFs in the application layers, that is all VNFs that are actually involved in the forwarding and processing of network traffic.

Most VNFs can be implemented in two kinds of virtualized environments. The first approach is to use full fledged VMs like the ones offered by VMware vSphere [33], Linux KVM [31], Citrix Hypervisor [32], etc. The second is to use OS-level virtualization, that is to deploy VNFs inside containers provided by Docker [5] or LXC [6]. Each solution has its own advantages and disadvantages. While VMs can be better isolated from each other, since each application runs in an isolated environment of a real computing machine, containers only duplicate the necessary applications and resources that cannot be shared with other applications. The most obvious results of these different approaches are that VMs can provide greater levels of isolation security between applications and can even run applications that were designed for different operating systems/architectures in emulation mode, at the cost of higher overheads. Containers are a more lightweight solution [34, 35], but since they share some resources, most notably the operating system kernel, security concerns may arise due to possible interference between containerized applications [34, 36, 37]. These characteristics of the two technologies could be used to determine whether they could be of use when designing a network architecture. For example, containers may not be suitable for being used within public cloud infrastructures [7, 38], since they do not provide enough isolation guarantees among tenants, but they could be suitable for private cloud environments, in which applications could be considered more trustworthy. For a more detailed exposition about virtualization techniques applied in this context, refer to Chapter 3.

In addition to the environments illustrated above, it is also important to notice that part of the networking operations between VMs or containers are performed by hypervisors and other virtual networking software, thus some VNFs may be actually implemented as part of the hypervisor itself, while most of the more specialized network functions are usually implemented as applications to be deployed within a VM. For example, VMware vSphere [33] provides a virtual NAT implemented directly within the hypervisor to allow VMs to communicate with the host without polluting the

global address space.

2.3 NFV Challenges

While the benefits of NFV captured the interest of many major network operators, there are a certain number of challenges that need to be solved when deploying virtual appliances in a real scenario.

When talking about software-based implementation of network functions through virtualization technologies on general-purpose servers, it is extremely important to know whether the performance, such as throughput and latency, will be affected. The per-instance capacity of a VNF may be less than the corresponding physical version on dedicated hardware. Although it is hard to completely avoid performance degradation, we should keep it as small as possible while not impacting the portability of VNFs on heterogeneous hardware platforms. What's more, many network functions are designed to work with protocols developed for physical implementations of such functions and thus they have very tight requirements both in throughput and latency between various components. Previous works showed that virtualization may lead to abnormal latency variations and significant throughput instability even when the underlying network is only lightly utilized [39]. Ensuring that network performance achieved by virtualized functions are at least as good as the ones of purpose-built hardware implementations is the most crucial challenge in realizing NFV [4].

The separation of functionality from location also creates the problem of how to efficiently place the virtual appliances and dynamically instantiate them on demand. The NFV infrastructure should also be able to instantiate VNFs in the right locations at the right time, dynamically allocate and scale virtualized resources for them, and interconnect them to achieve service chaining. This introduces a new challenge, as the cost and value of resources may vary significantly between network points and customer premises. MANO functionalities should take in account all these variations and optimize resource usage across the whole infrastructure, compatibly with the timing constraints and performance requirements of each service. To do so, tools and technologies developed for Software-Defined Networking (SDN) may be extremely useful, as NFV and SDN are closely related to each other. SDN is a networking paradigm that decouples the control plane (where MANO decisions take place) from the underlying data plane and consolidates the control functions into a logically centralized controller. NFV and SDN are mutually beneficial, as both promote the innovation of new technologies for a more dynamic approach to networking.

Another relevant challenge that network operators will face when migrating existing network infrastructures to NFV is the integration between already-existing PNFs inside the infrastructure, since not everything can be immediately virtualized at the same time [40]. The smoothness of the migration depends on the size of the existing network infrastructure and the capability of managing a solution that requires both custom PNFs and VNFs to operate together.

Chapter 3

High Performance Communications in Virtualized Environments

One of the key aspects that need to be addressed when moving to a NFV-based architecture is network performance when exchanging data between VNFs, both from throughput and latency point of views.

In this chapter we address that problem from the virtualization perspective. After a brief illustration of the different virtualization techniques that can be applied to build a NFV-based infrastructure, we show why some of them are more suitable than others from a performance point of view. Then we move on to the various techniques available to exchange data between virtualized components, either on a single host or between multiple hosts, focusing on the options that optimize system performance. Finally, some related work in the field of performance analysis among virtual networking solutions for NFV is presented.

3.1 Components and Virtualization

Virtualization is a technique that has various applications and advantages when deploying software components within a networked infrastructure. It has been widely accepted because of its characteristics such as elasticity and flexibility in delivering on-demand resources (e.g. computational power, storage, etc.). Nowadays, virtualization is used on cloud computing platforms to provide a multitude of services and it is constantly evolving to improve their efficiency and cost-effectiveness.

One of the key characteristics of virtualization is that it allows the creation of multiple virtualized computing environment within the same host. This way, the resources available to a single host can be fully exploited,

solving the problem of resources under utilization. In addition, these virtualized environments, usually referred to as guests, are typically isolated from each other. Applications running in different guests perform their activities by interacting with an abstraction layer provided by the virtualization platform instead of the actual resources they are using; this is a key point to avoid any interference among different applications deployed on the same host and to decouple each application from its actual deployment location. Furthermore, virtualizing applications increases the overall security of the system, as additional checks can be performed by the virtualization platform whenever an actual resource is accessed from any of the virtualized environments.

3.1.1 Main Virtualization Techniques

In this section we present the various environment virtualization techniques that can be applied to deploy applications on a general-purpose server architecture. Each of them comes with its own advantages and drawbacks, which will be discussed with respect to the NFV point of view.

Full Virtualization

A Virtual Machine (VM) instance represents an entire isolated environment for one or multiple applications (e.g. hardware devices, storage, operating system, etc.); usually multiple isolated environments can run independently on the same host and multiplex its resources. This requires more resources than the ones needed by the isolated applications, as the virtualization mechanism needs to mediate interactions with the underlying hardware, but the benefits of multiplexing applications in isolation usually justify the overhead. VMs are usually instantiated, terminated and migrated between hosts according to specifications of MANO, in order to provide the required level of service. Since the decoupling of VMs from the actual hardware present on a single host is necessary if we want to preserve the ability to migrate them freely among hosts, it is often necessary to adopt what is called hardware-level virtualization (or full virtualization), in which VMs are provided an abstract execution environment in terms of hardware by the Virtual Machine Manager (VMM), also called hypervisor. In this model, each VM contains its own OS, which can even differ from the one running on the host, and each interaction between the VM and the underlying hardware of the actual machine is mediated by the hypervisor.

An obvious advantage of this technique is that VMs are fully isolated from each other, at the point in which a system crash in one of the VMs

(like a kernel panic) won't affect the behavior of other unrelated VMs deployed on the same host. Essentially, the hypervisor can emulate almost every component of the hardware platform [41], although for NFV applications the two most common cases of emulation involve only I/O instructions and privileged CPU instructions. Another key characteristic is that hypervisors can dynamically adjust the mapping relationship between physical resources and virtual resources allocated to each VM to dynamically adapt the system over time and enable higher levels of portability of the single VM instances.

The most clear drawback of this scenario is that execution times of applications within VMs are inevitably enlarged, both because of the emulation of the instruction set when run on a different architecture and because of the required mediation from the hypervisor whenever hardware resources are accessed (like when exchanging data via network devices). This loss of performance can be very troublesome from a NFV perspective, where a key role is played by the communication overheads experienced by the individual software components participating in each deployed VNF. Despite the various techniques to increase performance of VMs developed over the years, like hardware-assisted virtualization or para-virtualization, the requirements of typical NFV applications are so tight that NFV has already focused on lightweight virtualization solutions based on OS containers, rather than traditional VMs.

OS-level Virtualization

OS-level virtualization, as opposed to full virtualization, achieves the objective of providing multiple isolated environments to applications, called containers, by separating a single operating system environment into multiple user space instances. To do so, containers are created by recurring to proper kernel-level encapsulation and isolation techniques. Thus, containers can be more efficient compared to VMs, because the former run directly on the host OS while the latter run on the guest OS [38].

Although both containers and VMs are virtualization techniques, they solve different problems: while VMs can provide to an application a whole infrastructure if required (usually emulated), containers can only ship the required software tools on top of the existing OS kernel to a target application, like software shipped within a certain Linux distribution (e.g. Debian, RHEL, etc.). This means that VMs act as a form of Infrastructure as a Service (IaaS), while containers can only provide Platform as a Service (PaaS) [42].

Examples of containers include LXC [6], Docker [5], OpenVZ [43]. Other

companies also offer support for containers like Amazon Elastic Container Service (ECS) [44] and Google Kubernetes Engine [45]. In addition, solutions for containers orchestration and scheduling are becoming increasingly popular [7], such as Google Kubernetes, Mesos, Cloudify, Docker Orchestration and Docker Swarm.

Mixed Virtualization Techniques

Based on OS containers and VMs, other virtualization technologies that can be applied in the field of VNF have been developed over the past few years, although their diffusion is still limited.

One of these approaches is to use what are called Clear Containers, developed as part of the Clear Linux project and then included as part of Kata Containers [46]. The main idea behind clear containers is to enhance performance as primarily measured using two metrics: startup time and memory overhead. These are the two most expensive operations when a multitude of VMs must be spawned to perform very short tasks (less than a second of actual execution time) for which a startup time of over 2 seconds and the memory footprint of an entire guest OS is too much. The approach taken consists into distributing slightly modified versions of the Linux kernel that support the execution of light VMs (usually with QEMU-lite or a similar hypervisor) running in each VM the same version of the Linux kernel already running on the host. In this way, the corresponding boot-up time and image size are close to those of a container, while still maintaining a high degree of isolation[1].

Another interesting approach to application encapsulation and isolation makes use of special VMs that are called Unikernels [47]. These are in fact specialized VMs characterized by a single address space and constructed by using library operating systems [48]. This means that each core functionality provided by the OS is encapsulated within a library which can be linked to the actual application. Within an unikernel image then only the libraries actually used by the target application are included and nothing else; basically an unikernel image is made of a single monolithic binary that contains both the OS (stripped to the bare minimum required) and the application [47]. The resulting VM images have both size and startup times similar to a container image while maintaining the strong isolation properties of a traditional VM. However, due to the special characteristics of unikernels, the act of encapsulating an application within a VM is not so straightforward and it requires a recompilation of a customized image

[1]For more info see https://lwn.net/Articles/644675/

for each application, which has been demonstrated to be quite challenging, although automated build tools have been subject of recent research activities[2].

Since both clear containers and unikernels are solutions that address the problem of VM instantiation and footprint, they are likely to be interesting for applications in which VM instances are created and destroyed frequently; however, this seems unlikely in NFV scenarios, for which it is more important that very high-performance applications are executed for quite some time after their instantiation. Hence we will not consider neither clear containers nor unikernels in our next arguments.

3.1.2 Virtual Machines vs OS Containers

Many studies considered containers against VMs with respect to High Performance Computing (HPC) applications performance, either CPU or I/O intensive. Results show that LXC or Docker containers performance are much higher than KVM VMs for HPC activities, both for CPU and communication performance (network and inter-process), since the amount of overhead introduced by containers is almost negligible with respect to bare OS performance [34, 35]. The superior performance of LXC has been observed for cluster environments, where there is more cooperation between processes.

Although containers offer a great opportunity to reduce the overhead, they also introduce many potential security issues due to the lack of isolation from the hosts [34, 36, 37]. In particular, sharing the kernel among various isolated application exposes each of them to the interference of the others, in the sense that a single kernel crash caused by one of the applications can lead to an entire host shutdown, affecting the other VNFs deployed on the same host. That's why containers have a more practical application for NFV infrastructures realized with private cloud models rather than on public infrastructures: for public cloud providers, interference among multiple tenants is an unacceptable fault and applications deployed within a private infrastructure can be considered more trustworthy among each other. While VMs may still be one of the primarily solutions adopted by public cloud providers, private infrastructures can potentially leverage the performance increase introduced by containers in a mostly secure way.

[2]For more info see https://xenproject.org/developers/teams/unikraft/

3.1.3 OS Containers and Linux

The Linux kernel supports containers by proper configuration of control groups and namespaces, which in combination create an environment in which an application can only see and access those resources that are actually allocated to its container and nothing else. This creates a certain degree of isolation among applications sufficient to deploy multiple applications on the same host. In particular, a process running in a namespace has the illusion to interact with a dedicated copy of the resources within the namespace and control groups regulate how these interactions can be carried out without breaking the contract between the container and rest of the OS. Various namespaces are included within the Linux kernel, each associated with a different kind of hardware or software resource that needs to be isolated/virtualized. For instance, a network namespace can encapsulate resources used by kernel network stack, like hardware or virtual interfaces, routing tables, etc.

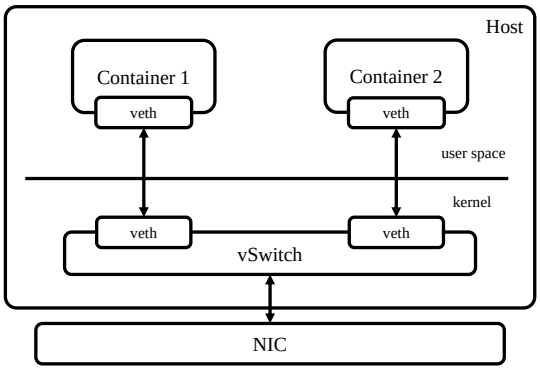
3.2 Inter-Container Communication Techniques

When deploying applications within virtualized environments, including OS containers, there are a number of different techniques that can be adopted to interconnect each encapsulated component with the outside world. Usually, such techniques leverage network virtualization to provide each virtualized environment a set of gateways that can be used to exchange data, as if each VM/container was connected to a virtualized network infrastructure.

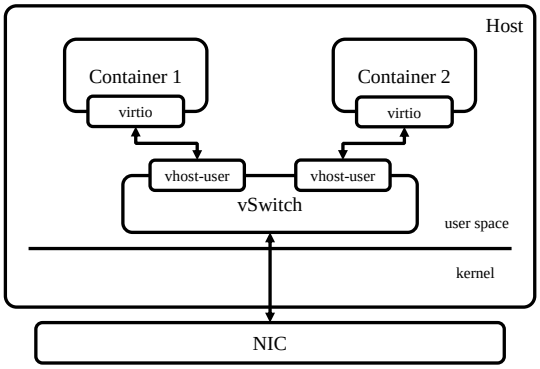
The scope of this section is to illustrate the various techniques that can be adopted to connect OS containers in a virtual network. Since NFV applications have very demanding requirements in terms of performance, the focus of this section is to compare the various techniques with respect to achievable performance on a general-purpose computing machine. Refer to Fig. 3.1 for a visual comparison among the various techniques described in this section.

3.2.1 Containers Networking Through the Linux Kernel

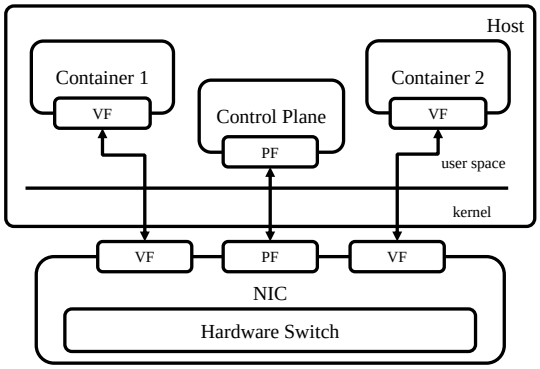
Using the standard techniques provided by the Linux kernel, applications deployed in containers can be connected in a virtual network within the



(a)



(b)



(c)

Figure 3.1: Different approaches to inter-container networking: (a) kernel-based solution; (b) using DPDK with *vhost-user* to bypass the kernel; (c) using SR-IOV support.

same host by creating virtual Ethernet ports and connecting them together, either in pairs (when only two containers need to communicate with each other) or to a virtual switch included within the kernel, called "*linux-bridge*". Each virtual port has no corresponding hardware interface, they are purely implemented in software as endpoints for networked communications within the same host, and they are each deployed in a separate namespace (usually one per container) to allow applications in different namespaces to exchange data. Traffic can also be exchanged with the host namespace, in particular when it is necessary to send or receive data from an actual network interface present on the host machine; *linux-bridge* can connect these virtual ports to the actual interface and forward traffic among all the ports according to the forwarding rules provided by the administrator.

Traditionally, virtual Ethernet ports are managed by the Linux kernel (shared among components), which means that their endpoints inside containers are simply accessed via blocking or nonblocking system calls through the POSIX Socket API. The typical usage of such APIs is to either send or receive one packet per system call, such as when using plain `send()` or `recv()` functions (or their more general forms `sendmsg()` and `recvmsg()`). As a result, at least two system calls are required when exchanging a simple UDP datagram, along with the various user-to-kernel-space switches, data copies and scheduling decisions. Therefore, the overheads associated to these system calls grow to prohibitive values when small packets are exchanged over the network. These costs can be amortized by increasing the size of each packet, but they are still present nonetheless; performance is still limited by the necessity to copy data from the sender to the receiver address spaces during each call.

To better amortize the cost of each system call, batch APIs have been introduced, allowing applications to send or receive multiple packets with a single system call: for this purpose the `sendmmsg()` and `recvmmsg()` APIs can be used. Even using these APIs, the problem of data copies is still present; the only way to avoid this cost is to connect an application directly with the underlying Network Interface Controller (NIC) hardware by exploiting memory-mapped I/O, scatter-gather primitives or using zero-copy APIs, such as the `MSG_ZEROCOPY` flag.

The latter solution is often adopted in combination with raw sockets, which allow applications to bypass most of the network stack implementation within the Linux kernel and construct/process themselves the actual Ethernet frames that will be sent or received through the NIC device. This allows to skip higher levels of the operating system stack, which do not

have full hardware control and thus add up cycles on each packet processing. That is why many packet capture solutions reduce the amount of processing required from the OS kernel using raw sockets and adopt memory-mapping techniques to reduce packet transmission costs from kernel to user space through system calls [49].

This approach is actually effective, as it introduces a significant speedup when a network device can be accessed directly by an application, and even in the case of virtual Ethernet devices it can lead up to 2x performance improvements with respect to entirely kernel-based UDP sockets; many works demonstrated that in modern operating systems moving a packet between the wire and the application can take 10-20 times longer than the time needed to send a packet over a 10 Gigabit interface [50]. However, this approach introduces also a number of limitations. First of all, an application requires to access in an exclusive fashion the network interface (either physical or virtual) by preventing any other application to use that same interface at all; this problem is not very relevant for containerized applications, since each virtual port associated to a container can be accessed only by one processing application, which is the only one deployed within that container. Second, this approach shifts the burden to write all network stack layers, from data-link up to application layer on the application developer; for this purpose, some efficient user space implementations of the network stack have been developed. Another problem of this solution is that the Linux kernel is only partially avoided, since the NIC device (either physical or virtual) needs to be accessed through system calls to actually exchange data. Lastly, the `MSG_ZEROCOPY` option is most effective only when the NIC device in use is an actual physical device which supports buffering in the network driver, hence its usage with virtual Ethernet ports (like the ones usually adopted when deploying applications within containers) does not influence very much networking performance.

3.2.2 Inter-Container Communications with Kernel Bypass

The main problem when using the Linux kernel to achieve network communications is the kernel itself: even if most of the networking stack can be bypassed or device buffers can be accessed directly from within applications through memory-mapped I/O, various parts of the kernel are still in use throughout communications (unavoidable parts of the network stack, kernel-based device drivers for NICs, etc.). In addition, the same Ethernet device abstraction is used both for local and remote communication, which

means that other forms of optimizations that could be applied by applications in certain scenarios (like actual zero-copy data transfers between virtual devices) cannot be adopted at all, since all they see are kernel-based networking devices. To achieve significant performance improvements for inter-container network communication, system calls, context switches, and data copies should be avoided as much as possible. For examples, if two containers are co-located on the same physical node, they could exchange data by using shared memory buffers. If not, applications within containers could access Ethernet devices directly and drive them using user space device drivers implementations. Of course, this requires the introduction of new mechanisms that bypass the in-kernel networking stack and drivers.

Different I/O frameworks allow *kernel bypassing* to exchange batches of raw packets between applications without a single system call.

Local Communications and *vhost-user*

A first optimization that can be adopted when containers are co-located on the same host (a situation which is fairly common given the stringent requirements of networking performance between adjacent VNFs in the forwarding graph) is to switch to para-virtualized network interfaces based on the *virtio* standard [51, 52]. These interfaces expose "virtual queues" for incoming/outgoing packets that can be shared among different guests on the same host, allowing the implementation of efficient host/guest communications. The para-virtualization of these interfaces is much more efficient than the full device virtualization provided by virtual Ethernet ports, as it operates at a much higher level than the latter and both the host and the guest can cooperate to achieve more efficient communications. Still, *virtio* interfaces can be used with traditional socket APIs when handled by the Linux kernel.

While *virtio* network devices are typically implemented for hypervisors (e.g. QEMU, KVM), the recent introduction of a complete user space implementation of such interfaces called "*vhost-user*" allows complete *kernel bypassing* and it can be used within containerized environments to improve significantly local communications network performance. When using the kernel-based version, the *vhost* services [53] are implemented through kernel threads created by the *vhost-net* kernel module, while the user space implementation uses a simple daemon [54] to provide the same services.

While *virtio* is useful when communications between co-located containers are required, it does not provide any functionality when remote hosts need to be reached, which means that alone it cannot be used to achieve dynamic and flexible communications between independently deployed con-

tainers. Thankfully, other frameworks are also available to take advantage of *kernel bypass* techniques for both local and remote communications.

Netmap

Netmap [55] is a framework which allows commodity hardware (without modifying applications or adding custom hardware) to process the millions of packets per second which can be transmitted/received on 1 or 10 Gigabit links. Netmap's high performance is achieved by removing three main packet processing costs, namely per-packet dynamic memory allocation (removed by pre-allocating resources), system call overheads (amortized over large batches), and memory copies (by sharing metadata and buffers between kernel and user space), while access protection for device registers and other kernel memory areas still remains. The main goal of Netmap is to build a fast path between NICs and the applications [56].

According to previous works [55], using Netmap a speed of 14.88 Mbps can be reached with a single core 900 MHz processor, which represents also the maximum packet rate for a 10 Gbps link and the same core can reach well above the capacity of 1 Gigabit links just by running at 150 MHz. In fact, for the minimum Ethernet frame size of 64 bytes there are also 7 bytes of preamble, 1 byte which represents the start of the frame delimiter and 12 bytes of the inter packet gap, which in total adds up to 160 bits to each frame. This means that on a 10 Gbps link at most the rate of 14.88 Mpps can be achieved, well below Netmap capabilities.

To use Netmap, a network interface must be switched from the regular mode, in which the packets are exchanged between the NIC and the host networking stack, to Netmap mode, in which NIC ring buffers are directly connected to Netmap-defined buffer rings, implemented in shared memory.

Netmap provides a few interesting features to achieve high performance for both local and remote communications, including support for multiple hardware queues, zero-copy data transfer between supported interfaces, zero-copy between applications and interfaces (via direct and protected access to packet buffers), and memory pre-allocation for packet buffers to avoid the cost of per-packet allocations/deallocations.

Data Plane Development Kit (DPDK)

Data Plane Development Kit (DPDK) [8] is a set of data plane libraries and drivers which are used for fast packet processing. Initially developed by Intel for its x86 CPUs, it is now extended to support other architectures, like IBM Power and ARM. DPDK main goal is to provide a framework for

fast packet processing in data plane applications that is simple and complete, providing an abstraction level that allows applications to be easily distributed on a multitude of different platform with relative ease, resulting in more scalable and simplified infrastructure solutions [57].

The high-level programming abstraction provided by DPDK is called Environment Abstraction Layer (EAL) [58] and it provides a generic interface that applications can use to gain accesses to low-level resources as memory space and hardware devices in a simple and generic way, without any intervention from the kernel. Each application can spawn one or more threads based on the pthreads library that continuously run on one core each to produce or consume data from network interfaces. Performance is improved adopting a series of techniques, all implemented in user space, that allow to reduce delays between the network device (physical or virtual) and the application itself [57]; the most common techniques include using cache alignment on most memory structures, setting core affinity for threads, disabling interrupts (accessing device buffers in polling), transferring data from and to devices in batches, avoiding overheads related to context-switches (using non-blocking APIs to avoid the intervention of the scheduler), and implementing memory buffers in huge pages residing in memory, thus avoiding delays introduced by memory addresses translation and swapping.

Some key components provided by DPDK include:

Memory Pool Manager used to allocate Non-Uniform Memory Access (NUMA)-aware pools of objects in huge-page memory space.

Buffer Manager greatly reduces the amount of time needed for allocating and de-allocating packet buffers.

Queue Manager implements safe lockless queues that do not rely on spinlocks, to avoid waiting for other threads when managing queues.

Poll Mode Driver (PMD) the main driver used to access 1 Gigabit and 10 Gigabit Ethernet controllers (either physical or virtual), which improves the efficiency of packet transfers by avoiding asynchronous interrupt-based signaling mechanisms.

In addition, DPDK supports *virtio*-based network devices (thanks to its implementation of *vhost-user* interfaces); this means that applications within containers can communicate with others by using DPDK libraries, either using *virtio*-based virtual devices or by accessing actual Ethernet devices present on the host, without any changes to their implementation. During the initialization phase, the Environment Abstraction Layer

(EAL) will simply parse a set of parameters and setup all the required ports accordingly; later, each port can be used in the same way, regardless the underlying interface used. This simplifies considerably the developers work, which can simply build their applications on top of the standardized interface and then deploy them on any platform, connecting each component to any kind of DPDK-supported interface. For this reason, DPDK has become extremely popular over the past few years when it is necessary to implement fast data plane packet processing.

3.2.3 High-Performance Switching Among Containers

The realization of a virtual network infrastructure on a single host can be achieved in multiple ways, either by placing a direct communication link between each pair of VNFs that need to communicate (e.g. placing each end of a *virtio* pair within each VNF namespace) or by interconnecting a group of components to a virtual switch component, usually running in the host namespace. Between the two approaches, the first one is more efficient from a performance point of view, as no additional packet processing is needed to forward traffic from one component to another. However, it is also subject to exponential growth with the number of VNF components, since a private communication link is assigned to each pair of VNF components connected within the forwarding graph. This is a viable approach when the communication is only local to a single machine (i.e. multiple containers on the same host), but it cannot be used in the more general case where VNFs can be deployed anywhere on the cloud infrastructure.

For these reasons, over the past few years a number of software implementations of L2/L3 switches have been developed. These components act like any actual switch, only their implementation is purely software and they can connect together both physical and virtual interfaces. The mentioned *linux-bridge* is one example of such component implemented directly within the Linux kernel. Since these switches are placed in between network endpoints, they must be able to process packets on the data plane very efficiently to satisfy performance requirements both in terms of throughput and latency between components, to leave as much flexibility as possible to MANO when deploying VNFs.

Implementing a virtual switch inside the kernel using *linux-bridge*, as depicted in Fig. 3.1a, can lead to a series of problems that may impact the overall performance of the NFV infrastructure. As already mentioned before, adopting techniques that avoid the kernel at all is the only effective solution to reach traffic volumes of millions of packets per second and above. That's why many virtual switches are implemented as user space

applications that use DPDK to accelerate packet processing, both bypassing the kernel when accessing physical NICs and using *virtio* endpoints to communicate with each container. While this is not the only solution possible to implement a virtual switch, it is widely diffused, with many virtual switches adopting *vhost-user* to communicate with local containers and using user space drivers to connect to physical network interfaces.

The following is a list of the most commonly adopted solutions in the industry for inter-container communications using virtual switches. For each of these scenarios (with the exception of SR-IOV), each container is connected to a software-based virtual switch via a *vhost-user* based connection, as shown in Fig. 3.1b.

DPDK Basic Forwarding Sample Application

DPDK provides a set of example applications alongside its installation on a machine. One of such applications is Testpmd [59], which is a software that can be used to test the functionality of DPDK PMD driver by connecting ports in pairs. On top of that, Testpmd provides other sets of functionalities (e.g. statistics collection, port management operations, etc.), which can be used to get familiar with the various functionalities provided by DPDK. This application acts as a simple bridge between pairs of virtual or physical ports; albeit configurable, the default pairing between ports is performed on their ordering (ports 0 and 1, ports 2 and 3, and so on), which means that this software does not contain any actual switching functionality nor packet inspection programmed in it.

Due to this limitation this software cannot be used in a production environment, as it is not possible for a VNF to be connected to more than one other VNF without opening multiple ports, but it is also very efficient with respect to throughput and latency performance, as no packet processing operation is necessary to forward traffic, the only information needed is the port from which the packet was received.

In this sense, Testpmd is equivalent to another sample application provided by DPDK, which is called the Basic Forwarding Sample Application [60]. As this is less configurable than Testpmd, it is even more lightweight and thus it can be used in its place when a simple bridge between pair of ports is needed.

Open vSwitch (OVS)

Open vSwitch (OVS) [61] is a multi-layer, open-source virtual switch developed to be compatible with all major hypervisor and container systems [62].

In contrast with early virtual switches used for this purpose, OVS does not use static forwarding pipelines to provide L2 connectivity among VMs (or containers) and the physical network. It is instead designed for flexibility and general-purpose usage and it is even programmable remotely via *OpenFlow*.

OpenFlow [63] is a simple binary protocol that can be used to monitor remotely a supported switch, either software or hardware, by collecting statistics on flow tables. In addition, it allows software controllers to change the behavior of the monitored switch by adding, removing or updating flow tables remotely.

The flexibility of *OpenFlow* allows OVS to be used in combination with SDN controllers and makes it extremely suitable for NFV MANO, but it also means that OVS needs to adopt a few precautions to be as efficient as less flexible solutions [62]; one example of these precautions is flow caching, which is an important feature within OVS implementation.

Recently, OVS has been updated to support DPDK (and hence also *virtio*-based ports), which accelerated considerably packet forwarding operations by performing them in user space rather than within a kernel module [64].

FD.io Vector Packet Processing (VPP)

FD.io Vector Packet Processing (VPP) [65] is an extensible framework which provides out-of-the-box production quality switching and routing functionality released by the Linux Foundation Fast Data Project (FD.io). Since it is developed on top of DPDK, it can run on various architectures and it can be deployed in VMs, containers or bare metal environments.

It is a general purpose framework which can be configured on deployment, simplifying the deployment of a set of already available VNFs implemented as software components (thus not encapsulated within containers). VPP runs as a Linux user space application and it is conceived as a "packet processing graph", to which customized graph nodes can be easily plugged into complex and dynamic service chains.

While VPP is not the only framework of this kind (e.g. Click [66], FastClick [67], etc.), an advantage of VPP over similar counterparts is that it implements an open-source version of Cisco VPP technology as its core processing paradigm, from which the framework gets its name. The core idea behind VPP is to process more than one packet at a time, taking advantage of both instruction and data cache localities to achieve lower latency and higher throughput [68]. VPP is very efficient when the number of packets per vector increases, as the processing cost per packet of I-cache

misses is amortized over the large number of packets processed by each function in a single vector.

Snabb

Snabb [69] is a simple and fast packet networking toolkit that can be used to program user space packet processing flows [54]. This is done by connecting functional blocks in a directed acyclic graph, each block performing a specific action or representing a custom driver for an interface. Differently from previous solutions, Snabb is not based on DPDK but instead it has its own implementation of both *virtio* and NIC drivers in user space (although it lacks support for some Ethernet devices).

A certain Snabb configuration can be deployed by writing an appropriate program written using Lua language, whose only purpose is to setup and connect together the components that will form the packet processing graph. This is another key difference with respect to other solutions: while others accept a simple configuration file and do not require users to write any program, a simple Snabb application can be started only if the appropriate configuration program is supplied. On the other hand, the fine grain control over the resulting graph allows for greater optimization, including in the graph the required blocks only.

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

Single-Root I/O Virtualization (SR-IOV) [9] is a specification that allows a single PCIe device to appear as multiple physical PCIe devices. This is achieved by introducing the distinction between Physical Functions (PFs) and Virtual Functions (VFs): the former are capable of using the full list of features of the PCIe device, while the latter are "lightweight" functions that have only the ability to move data between an application and the device.

VFs can be individually exposed in passthrough to VMs or containers, which can access directly the hardware device without any need for virtual switching. Most SR-IOV devices contain a hardware L2 switch to forward traffic among PFs and VFs. This is depicted in Fig. 3.1c.

SR-IOV devices can either be accessed trough OS drivers or via DPDK APIs, which allow applications to gain complete control over a VF directly in user space. In this case, an instance of DPDK Testpmd application must run on the host to handle configuration requests from applications, to assign or release VFs, and to configure the control plane associated with the hardware switch, which will then perform all forwarding operations in

hardware.

3.3 Performance Comparison Among Virtual Switches

In this section we include and slightly expand the comparison published in our previous work [1] among various techniques that make use of *kernel bypass* for high-performance inter-container communications.

The proliferation of different technologies to exchange packets among containers has created the need for new tools to evaluate the performance of these virtual switching solutions with respect to throughput, latency and scalability. Several works appeared in prior research literature address the problem of network performance optimization for virtual machines and containers. For example, some authors [70] investigated packet forwarding performance achievable in virtual machines using different technologies to implement the virtual NICs and to connect them to the host network stack or physical NIC.

Many solutions exist to greatly reduce the overheads due to interrupt handlers, like interrupt coalescing and other optimizations available on Linux through Linux New API (NAPI) [71], including hybrid interrupt-handling techniques that switch dynamically between interrupt disabling-enabling and polling depending on the actual traffic on the line [72].

Comparisons among traditional sockets, DPDK, and Remote Direct Memory Access (RDMA) [73] already exist, mainly focusing on the achievable minimum round-trip latency between two different machines [74]. These works show how both RDMA and DPDK can outperform kernel-based UDP sockets, achieving much smaller latency for small UDP packets; their drawback is that they force applications to operate in poll mode, leading unavoidably to high CPU utilization. Also, authors point out that DPDK can actually be used in combination with interrupts, saving energy, but before sending or receiving packets the program must switch back to polling mode. This reduces CPU utilization during idle times, at the cost of a greater latency when interrupts must be disabled to revert to polling mode, when the first packet of a burst is received.

A survey among common networking setups for high-performance NFV exists [75], accompanied by a quantitative comparison addressing throughput and CPU utilization of SR-IOV, Snabb, OVS with DPDK and Netmap. In this work, the authors highlight how each different solution has remarkable differences in security and usability with respect to each other, and

they show that for local VM to VM communications Netmap is capable of reaching up to 27 Mpps (when running on a 4 GHz CPU), overcoming SR-IOV due to its limited internal switch bandwidth that becomes a bottleneck.

Another work [76] compared the various trade-offs between throughput and latency when adopting three different frameworks for fast packet I/O: DPDK, Netmap and PF_RING [77]. In this work, authors illustrated that the main hardware characteristics that can potentially limit packet processing performance for inter-machine communications are the CPU (which is considered often the dominating bottleneck, as most of these solutions require applications to fully utilize CPU cores), NIC maximum transfer rate (determined by the Ethernet standard), the PCIe bus connecting the NIC to the rest of the system, and RAM memory. In particular, the first two characteristics are considered the most limiting features of the system, thus as long as the processing cost per packet is kept low the main limitation for inter-machine communication remains the one introduced by the Ethernet standard onto the NIC device. However, as per-packet processing cost increases the main limitation in terms of throughput is often represented by the CPU (which becomes fully loaded). From their evaluations, DPDK is considered to be the most lightweight solution, both with respect to PF_RING and Netmap, and it is also able to achieve the highest throughput in terms of packets per second for small burst sizes (about 32 packets per burst); on the contrary, Netmap reaches its highest throughput for bigger burst sizes (about 128 packets per burst), but with lower throughput compared to both DPDK and PF_RING.

In another interesting work [78], VPP, OVS and SR-IOV are compared with respect to scalability in the number of VMs on a single host. The results of such evaluation show that SR-IOV offers a highly linear contribution to the total system throughput as VNFs are added to a certain machine, while both VPP and OVS are able to scale throughput linearly with the number of VNFs up to a certain plateau, which is directly influenced by the amount of CPU resources allocated to VPP and OVS virtual switching functions. The more VNFs are added, the more system performance are degraded if no more resources are allocated for the active virtual switch.

While in this work we will provide a thorough performance analysis for many of the networking solutions described in other works, some of them have not been included; in particular, we did not evaluate NetVM [79] and Netmap (along with its associated virtual switch, VALE [80]), because our focus is to evaluate solutions commonly adopted in current NFV industrial

practice.

Other works exist in the area, but a complete state of the art review is out of the scope of this thesis. While many of these works provide quite systematic reviews of some of the technologies that can be used for NFV, a standardized way to carry out such experiments does not exist, as there is no high-performance tool generic enough to be highly portable on virtually any kind of platform that can be used for NFV. For this reason, each different work relies on custom configurations of one or more machines.

For this reason, the main purpose of this thesis is to develop such a tool, to speed up the evaluation process of existing and potentially future technologies for NFV, with a particular focus of industrially viable solutions for high-performance networking for Linux containers in a NFV scenario.

This thesis is also the conclusion of our preliminary work [1], where a basic comparison of throughput performance among communication techniques for co-located Linux containers was performed, again in the context of NFV.

Chapter 4

Framework Description and Implementation

The main purpose of this work is the design and implementation of a software framework that can be used to evaluate the various technologies that can be adopted to achieve high-performance networking among Linux containers in NFV scenarios. In this chapter we thoroughly describe the requirements and characteristics of this framework.

The designed framework shall be used to setup and perform a series of tests to evaluate performance of networking solutions for NFV on one or more general-purpose servers. The framework shall be easy to configure and deploy on any kind of host, with little to no customization effort on behalf of the user and it shall take care of installing its own dependencies. However, the generality and portability of the framework must not impact negatively its performance, as that is its main purpose. For that, the portability of the DPDK framework will be an important factor, as this work will rely on the libraries included with DPDK to implement its testing applications.

Once deployed on one or multiple hosts, the framework can be used to instantiate OS containers and start a few applications (one per container) that will exchange data via network primitives. These applications will be also provided by the framework and they will serve a dual purpose, to both generate synthetic workload to simulate real NFV applications and to collect statistics that will be used to evaluate system performance in a particular configuration. Each test will execute for a specific amount of time, after which the framework will provide to the user the desired statistics and return. To ease the evaluation of a system when varying certain parameters, the framework shall also be designed to accept multiple tests configurations at once: in that situation, it will simply execute them one after the other autonomously and it will signal the user once all tests are done.

The applications that will run inside each OS container are provided by

the framework to evaluate network performance under the following points of view:

Throughput Dimension As many VNFs deal with huge amounts of packets per second, it is important to evaluate the limits of each networking solution with respect to the number of packets it is able to effectively process and deliver each second.

Latency Dimension Many VNFs shall operate with networking protocols that have been designed for hardware implementations, and as such they expect very low round-trip latency between components, in the order of single-digit microsecond latency. In addition, in NFV infrastructures it is quite common to have relatively long chains of services that communicate with each other; in these situations, what matters the most is the end-to-end latency, thus it is crucial to keep the latency of individual interactions as little as possible.

Scalability Evaluations from this point of view are orthogonal with respect of the two previous dimensions, in particular with respect to throughput: to fully utilize available resources it is necessary to deploy multiple VNFs on each host within the infrastructure, and as such it is extremely important to evaluate how this affects performance in the mentioned dimensions.

This section will show the whole framework that has been designed, first by describing what are the framework dependencies and installation process, and then moving on for a bottom-up description of all framework components.

4.1 Framework Installation

An automated installation script is provided to ease the gathering and installation of all framework dependencies and the configuration of the host for testing. The list of framework dependencies installed and configured by the installation script is the following:

LXC The backend used to instantiate and deploy containers on the host. The Linux distribution that will be deployed will be based on a simple *rootfs* built from a basic *BusyBox* and containing only the resources necessary to start the testing applications provided by the framework. The installation script will both install LXC and configure a set of containers based on an configuration file provided by the user (a set

of sample configuration files is provided with the framework itself). All containers created by the installation script will share the same *rootfs*, resulting in a very low disk occupation. The parameters that can be provided for each container include the name of the container, list of CPU cores to use, IP and MAC address, and the *virtio* socket or SR-IOV VF to use when started.

DPDK The framework will download DPDK sources from the official repository and build it from sources. The installation process will also build some example applications provided with DPDK, including Testpmd application and the Basic Forwarding Sample Application.

OVS The installation script will download OVS sources from the official repository and build it on the local host. This is necessary to compile OVS with DPDK support.

VPP This software switch will be installed from the official *APT* repository, which also includes DPDK support out-of-the-box. The framework includes a base configuration file that can be used to start with the desired parameters; that file can be easily modified to change such parameters with relative ease.

Snabb This software is downloaded from the official GitHub repository and compiled from sources. Since this software switch is implemented as a *BusyBox* including multiple applications and the only way to actually deploy the switch itself is to write a small program in Lua language to setup and connect the required components, a small Lua script is provided by the framework. The script will setup a simple learning switch component with a certain number of ports to which applications can connect to communicate.

Since DPDK relies on resident huge pages memory to allocate data structures and buffers used for packet processing, the system is configured to allocate a configurable amount of huge pages during boot through additional kernel options. In particular, it is configured to allocate a certain amount N resident pages of 1 GB with the following options:

```
hugepagesz=1G hugepages=$N intel_iommu=on iommu=pt
```

It then creates a huge page Translation Lookaside Buffer (TLB) File System that can be used by applications to index and allocate huge pages on demand[1].

[1]For more info see https://lwn.net/Articles/375098/

In addition, the installation script builds and installs the applications that will be used for performance testing inside the containers *rootfs*.

4.2 Testing Applications Included in the Framework

The applications used to generate synthetic traffic among containers are provided as a single executable file built from sources that act like a *BusyBox*, accepting as first argument the name of the application to run. Each application accepts the same list of parameters, although some specific applications may ignore some of the parameters. In general, each application generates or consumes a certain amount of packets each second. The applications are all configurable to handle packets in bursts and they all fully utilize the CPUs on which they are pinned, as they access network devices (virtual or physical) in polling mode.

Each application expects to be connected with another application over the local network, each pair independent from the others. The first four parameters expected by each application are in order the local IP and MAC address to use and the IP and MAC addresses of the other application in the pair. Then a list of optional parameters may follow, as shown in Table 4.1. In addition to these parameters, applications that make use of DPDK will also accept other parameters after a double dash (--), for example the name of a *virtio* interface to use or the PCIe address of a SR-IOV VF. These parameters are used to configure DPDK EAL [58] and thus are simply ignored by those applications that use only syscall-based API. For further information about EAL parameters refer to the official documentation [58].

There are two kinds of application pairs that can be used to test the networking performance of one or multiple hosts:

Sender/Receiver This pair of applications respectively generate/consume traffic with the purpose of evaluating the throughput performance of the networking among containers. In this scenario, the direction of traffic is only unidirectional, from the sender to the receiver.

Client/Server This pair of applications generate traffic with the purpose of evaluating round-trip latency of packets. In this case, traffic is bidirectional, as each packet sent to the server by a client is sent back from the server to the corresponding client.

Table 4.1: List of parameters accepted by each testing application. Some parameters may be ignored, refer to each application description for details.

Parameter	**Has Argument**	**Description**
`-r`	YES	The sending rate in packets per second
`-p`	YES	The size of each packet in bytes
`-b`	YES	The number of packets in each burst
`-R`	YES	If using syscall-based API, use raw sockets instead of normal sockets; the parameter is the name of the interface to use
`-B`	NO	If using syscall-based API, use blocking sockets instead of non-blocking API; ignored otherwise
`-c`	NO	If provided, the application will generate the content of the packet and it will touch each byte; if not, only packet headers are actually manipulated
`-m`	NO	If using syscall-based API, enable the use `sendmmsg()`/`recvmmsg()` API; ignored otherwise
`-s`	NO	Do not print any statistics until a signal is received

Multiple application pairs can be deployed at the same time to evaluate the scalability of the performance as the number of applications increases (and consequently, the amount of total traffic generated).

The following sections describe each of the applications included in the *BusyBox* and their main usage. Each application kind (i.e. sender, receiver, client or server) is present in two forms, one that uses socket syscalls and one that makes use of DPDK API. The two implementations are strictly related as they can be used to compare the performance of the same programming logic when applied respectively with traditional Linux networking API or with DPDK.

It is important to notice that there is no need to learn how to start these applications manually, as more high-level tools capable of automatically start and stop these applications with the correct parameters in the desired containers are also provided by the framework.

4.2.1 Syscall-Based Sender Application

This application uses sockets (either simple UDP sockets or raw sockets) to send packets to a corresponding receiver application and it is started by giving `send` as very first parameter to the *BusyBox*. By default it uses `send()` to send all the packets in each burst one at a time; if the `-m` parameter is provided it uses `sendmmsg()` instead to a whole burst of packets in one single syscall. In both cases, the use of `-R` parameter forces the application to use raw sockets, building Ethernet, IP and UDP packet headers in user space.

Since packets are sent in bursts, this application will attempt to equally divide the desired sending rate (`-r`) by the burst size (`-b`) to obtain the number of times it must wake up to generate new traffic. It must be noted that to achieve the highest performance possible, all applications included in the *BusyBox* do not use `sleep` or interrupt-based timers to be woken up at certain intervals, but instead continuously poll the content of the Time Stamp Counter (TSC) register of the CPU until the next expected value is reached.

This application is used to generate packets for throughput evaluation; as such, each second a log is printed containing the number of total packets that the application attempted to send, the number of actually sent packets and (by difference) the number of packets dropped by the interface. Using the `-s` option postpones the print of all the statistics at the end of the execution.

4.2.2 Syscall-Based Receiver Application

This application is the dual of the sender application and it is started by using `recv` as very first parameter of the *BusyBox*. As such, its description is the same as the sender, except the usage of `recv()` / `recvmmsg()` syscalls. However, this application ignores the `-r` parameter, and thus it continuously polls the socket input stream to check for incoming packets, either one by one or in bursts, depending on the other parameters.

The only statistic logged by this application over time is the actual receiving rate in number of packets per second for throughput evaluations. Like with any other application, these statistics can be postponed to be printed at the end of the execution.

4.2.3 DPDK-Based Sender Application

This application has the same purpose of its syscall-based counterpart, with the difference that all syscall-related parameters are ignored and it expects EAL parameters to configure the port over which network traffic should be generated. This port can be either a *virtio/vhost-user* port or a SR-IOV VF (specified via its PCIe address). It can be launched by using `dpdk-send` as first parameter.

Each time a new burst must be generated, this application requests a set of buffers allocated from huge pages memory space and proceeds to fill them with packets up to the size of the burst. It then sends them all with a single call, which takes the control of the given buffers and returns how many messages have actually been sent. The dropped messages must then be freed from memory before working on the next burst.

The statistics generated by this application are exactly the same format as the syscall-based one.

4.2.4 DPDK-Based Receiver Application

This application is the dual of the DPDK-based sender application and it can be started by using `dpdk-recv` a first parameter. This application ignores the value of `-r` parameter too, as well as all syscall-related parameters, to continuously poll the given port for new incoming packets in bursts. Each packet received is checked to ensure that it was indeed meant for this instance of the receiver application, checking destination MAC address, IP address and UDP port. This is necessary as DPDK ports are configured in promiscuous mode, and thus some traffic not actually meant for this application shall be ignored.

The statistics generated by this application are exactly the same format as the syscall-based one.

4.2.5 Syscall-Based Client Application

This application uses sockets (either simple UDP sockets or raw sockets) to send packets to a corresponding server application, expecting to receive those messages back. It is started by using `client` as first parameter. In contrast with the other applications included in the *BusyBox*, this one uses multiple threads, one for sending bursts of packets and one to receive back those messages. In addition, to calculate the delay of each packet in the most accurate way across different CPU cores, a third thread is used to continuously update the last known value of the TSC. This way, both the sender and the receiver threads can share a common time base.

The basic structure of the sender and receiver threads is much similar to the body of the sender and receiver applications, with minimum changes. In particular, the payload of each packet will contain a timestamp (expressed as value of the TSC register), so that the receiver thread can correctly calculate the round-trip delay time for that packet. Other than that, this application shares the same characteristics of the others syscall-based applications.

This application is used to generate traffic for round-trip latency evaluation and it logs two values over time: the number of packets received back each second and the average delay among of those packets. Notice that only packets received back contribute to the calculation of the average delay for each second. Using the `-s` option postpones the print of all the statistics at the end of the execution.

4.2.6 Syscall-Based Server Application

This application uses sockets (either simple UDP sockets or raw sockets) to send back packets received from a corresponding client application. It is started by using `server` as first parameter. This application uses a single thread, continuously checking for new incoming messages and sending back anything that is received as soon as possible. Data is not modified, except when using raw sockets: in that case, source and destination MAC addresses, IP addresses and UDP ports are swapped to match the inverted roles between sender and receiver application. For this application, the `-r` parameter is ignored, the opened socket is continuously checked for new incoming messages as soon as previous messages have been processed.

This application is used to send back traffic generated by clients for round-trip latency evaluation, but since all the statistics necessary for performance evaluation are produced by the client application it does not log anything.

4.2.7 DPDK-Based Client Application

This application has the same purpose of its syscall-based counterpart, with the difference that all syscall-related parameters are ignored and it expects EAL parameters to configure the port over which network traffic should be generated and then received back. This port can be either a *virtio/vhost-user* port or a SR-IOV VF (specified via its PCIe address). It can be launched by using `dpdk-client` as first parameter.

The basic structure of the threads of this application is very similar to the body pf the DPDK-based sender and receiver applications, with minimum changes. Like its syscall-based counterpart, a timestamp is added as first datum in each packet payload and it is later checked against the value of the TSC register upon reception.

The statistics generated by this application are exactly the same format as the syscall-based one.

4.2.8 DPDK-Based Server Application

This application is the dual of the DPDK-based client application and it can be started by using `dpdk-server` a first parameter. This application ignores the value of `-r` parameter too, as well as all syscall-related parameters, to continuously poll the given port for new incoming packets in bursts. Each packet received is checked to ensure that it was indeed meant for this instance of the receiver application, checking destination MAC address, IP address and UDP port. This is necessary as DPDK ports are configured in promiscuous mode, and thus some traffic not actually meant for this application shall be ignored. Then addresses and port numbers are swapped before sending back the received message. Messages are sent back in bursts with a number of packet per burst less or equal the number of received packets (depending whether some packets are dropped during header processing).

Like its syscall-based counterpart, this application does not log any statistics.

4.3 System Setup and Architecture

The framework includes some tools to setup an entire NFV use-case scenario by deploying a set of applications from the available list into a corresponding number of containers, taking care of all the setup necessary to interconnect these applications with the desired network technology. The latter may be any among *linux-bridge*, a software-based virtual switch (making use of *virtio* and *vhost-user*) or an SR-IOV Ethernet adapter. A basic representation of each of these scenarios is depicted in Fig. 3.1.

More in details, a set of Bash scripts has been developed to do a number of operations that automatize the deployment and the setup of the system for a specific evaluation:

- One to setup and start a single test on the local host. This script accepts a configuration file as input to specify which applications to deploy in containers on the local host and a set of parameters to be forwarded to each application.
- One to setup and start a single test on a remote host; this one simply copies onto the other host the configuration files and parameters needed, starts the script for local tests on that host and waits for test termination.
- One to iterate through a series of tests and start them one by one on the local host, on another host or on multiple hosts at the same time (usually when testing network performance in multi-host scenarios).

This section describes more in details all the actions performed by each script to bring each host into a suitable state for testing and perform the actual tests.

4.3.1 Local Test Configuration Script

This section describes the parameters and the actions taken by the Local Test Configuration Script to perform a single test execution on the local host. The script can be started by executing the command `./run_local_test` from the `scripts` directory. Table 4.2 contains the list of parameters available for this script.

When the script is started, the system is first brought into a suitable state for reproducible performance evaluations. To maximize reproducibility of test results, it disables CPU frequency scaling by setting the frequency governor of each core to "performance" and by disabling Turbo Boost on

Table 4.2: List of parameters accepted by the local test configuration script.

Parameter	Has Argument	Description
`-r` `--rate`	YES	The sending rate in packets per second
`-p` `--pkt-size`	YES	The size of each packet in bytes
`-b` `--bst-size`	YES	The number of packets in each burst
`-R` `--use-raw-sock`	NO	If using syscall-based API, use raw sockets instead of normal sockets
`-B` `--use-block-sock`	NO	If using syscall-based API, use blocking sockets instead of non-blocking API; ignored otherwise
`-c` `--consume-data`	NO	If provided, the applications will generate the content of the packet and it will touch each byte; if not, only packet headers are actually manipulated
`-m` `--use-multi`	NO	If using syscall-based API, enable the use `sendmmsg()`/`recvmmsg()` API; ignored otherwise
`-t` `--timeout`	YES	The duration of the test in seconds; default value is 60 s
`-v` `--vswitch`	YES	The name of the virtual switch to use; possible parameters are `basicfwd`, `linux-bridge`, `ovs`, `snabb`, `sriov`, `vpp`. If `linux-bridge` is used, each application started will use kernel-based networking, otherwise the DPDK-variants of each application will be deployed
`-f` `--conf-filename`	YES	The path of the containers' configuration file (see Table 4.3 and Listing 4.1)
`-D` `--debug`	NO	Force applications to log statistics during execution. The default behavior is to not log statistics to avoid interference with test results

the local host. It then proceeds to load a few kernel modules necessary to manage SR-IOV VFs using DPDK tools.

The script then proceeds to start the designated virtual switch for the current evaluation and to configure it for local communications. Each switch is configured with a number of ports equal to the number of applications that shall be deployed on the local host (either virtual ports or SR-IOV VFs) and then started. Each virtual switch is configured to act as a simple learning L2 switch, with the only exception represented by DPDK Basic Forwarding Sample Application, which does not have this functionality.

In addition, for VPP and OVS, which both support Intel SR-IOV-enabled Ethernet controllers, a physical Ethernet device is also connected to the virtual switch. This way, traffic to or from outside the local host can be exchanged via the physical NIC. Among the other virtual switches, using the one embedded within the SR-IOV device enables multi-host communications out-of-the-box, without any further configuration needed, Snabb supports only a few Ethernet devices, which does not include the ones available to us during the development of the framework, since each different driver must be implemented from scratch in Lua language, and the Basic Forwarding Sample Application. The latter does have support for the Ethernet devices we used during development, but it would be of little use: since only one port (virtual or physical) can be connected to another one using this software, so only one container could be connected to the outside using this configuration. Of course, if *linux-bridge* is used instead of *kernel bypassing* solutions, no operation is needed.

At this point, the script starts a set of containers, each with a specific application running inside. The list of all applications and containers to start, as well as how such applications should be connected with each other is provided through a configuration file, in the form of a "run commands" configuration file. The list of all available parameters to configure the containers to deploy for testing is available in Table 4.3, while an example of a configuration file is shown in Listing 4.1, in which three containers are started in each test: the first two communicate with each other, while the third one is configured to communicate to another container deployed on a separate host (more on that in the following section). On startup, each application will be pinned to the specific cores assigned to the respective container during installation and system configuration.

Once all containers are up and running, they will connect to the designated virtual switch automatically and thus the script waits for a certain amount of time (depending on the `--timeout` parameter) before signaling

them all to terminate execution. When all applications have terminated, the system is cleaned up and restored to its original state.

4.3.2 Remote Test Configuration Script

This script sets up for execution a single test on a remote host, specified via the `--hostname` parameter, and collects the statistics produced during the evaluation on the local host. The script can be started by executing the command `./run_remote_test` from the `scripts` directory. Its parameters are the same as the Local Test Configuration Script, with the addition of the ones listed in Table 4.4.

This script is very simple, as it simply copies the specified configuration file from the local host to the remote one and starts the Local Test Configuration Script on the remote host. It then copies back all the test results upon test termination.

4.3.3 Multiple Tests Configuration Script

This script is the most high-level tool that has been developed for the framework and it is intended for a more general use than the two previous scripts. Its role is to automate the execution of multiple tests with different configuration parameters (e.g. sending rate, packet size, virtual switch in use, etc.) either on a single or a multiple host environment. The script can be started by executing the command `./run_testset` from the `scripts` directory.

While some parameters resemble the ones used for the other two scripts, some of them have been adjusted to accept sets of values over which iterate to perform each different testing scenario. The complete list of parameters accepted by the script is shown in Table 4.5. This way, multiple tests can be performed varying the following parameters: packet size, burst size, desired sending rate, and virtual switch used. To achieve this goal, the parameter `-v` now accepts a string containing a list of different virtual switches that can be used in different tests (e.g. `"ovs snabb sriov"`) , while the parameters `-p` , `-b` , and `-r` can have two different kinds of parameters:

- The first kind of parameter is represented by a simple list of values to iterate enclosed within double quotes and separated by spaces, e.g. `"32 64 128 512"`.
- The other kind is represented by a description of three parameters that shall be used to generate a list of values to iterate, again enclosed

Table 4.3: List of parameters that can be provided via the local test configuration file.

Parameter Name	**Description**
`LXC_CONT_NAMES`	The names of each container to be deployed
`LXC_CONT_VFS`	The device names of each SR-IOV VF associated to each container (if SR-IOV is used)
`LXC_CONT_MACS`	The MAC addresses of each local container to be deployed
`LXC_CONT_IPS`	The IP addresses of each container
`LXC_CONT_NMASKS`	The network masks associated with each container
`LXC_CONT_VFPCIS`	The PCIe addresses of each VF
`LXC_CONT_SOCKS`	The names of the *virtio* sockets to be used in case *vhost-user* is adopted for network communications
`LXC_CONT_OTHER_IPS`	The IP addresses of the other container in the application pair
`LXC_CONT_OTHER_MACS`	The MAC addresses of the other container in the application pair
`LXC_CONT_CMDNAMES`	The names of the application to be started in the container; accepted values are `send`, `recv`, `client` and `server`. In this case there is no distinction between syscall-based or DPDK-based applications, the actual started application is determined by the networking mechanism specified with the script parameter `-v` (see Table 4.2)

```
LXC_CONT_NAMES=(                          \
    dpdk_c0 dpdk_c1 dpdk_c2               \
)

LXC_CONT_VFS=(                            \
    enp4s2 enp4s2f1 enp4s2f2              \
)

LXC_CONT_MACS=(                           \
    02:00:00:00:00:10 02:00:00:00:00:11 \
    02:00:00:00:00:12                     \
)

LXC_CONT_IPS=(                            \
    10.0.3.10 10.0.3.11 10.0.3.12         \
)

LXC_CONT_VFPCIS=(                         \
    04:02.0 04:02.1 04:02.2               \
)

LXC_CONT_SOCKS=(                          \
    sock0 sock1 sock2                     \
)

LXC_CONT_OTHER_IPS=(                      \
    10.0.3.11 10.0.3.10 10.0.3.20         \
)

LXC_CONT_OTHER_MACS=(                     \
    02:00:00:00:00:11 02:00:00:00:00:10 \
    02:00:00:00:00:20                     \
)

# Can be either send, recv, client, server
LXC_CONT_CMDNAMES=(                       \
    send recv send                        \
)
```

Listing 4.1: Example of a simple local test configuration file. Each container is assigned an index i and its parameters are supplied as arrays of values, where the i-th value is the attribute for the i-th container. In this case, `dpdk_c0` and `dpdk_c1` are connected in a pair, while `dpdk_c2` should connect to a container deployed on another host.

Table 4.4: List of parameters accepted by the remote test configuration script. In addition, the script accepts all parameters listed in Table 4.2, with the exception of the `--conf-filename` parameter, which is ignored.

Parameter	Has Argument	Description
`-H` `--hostname`	YES	The name of the host on which the test should be performed
`-F` `--remote-filename`	YES	The path of the containers configuration file for the remote host (see Table 4.3 and Listing 4.1); the file must be on the local host, it will be copied on the remote one

within double quotes and separated by two dots (`..`): the three parameters are in order the minimum value, the increment between the values of the generated list and the maximum value (included); for example, the value `"64..32..256"` for the burst size parameter instructs the script to iterate over the values `"64 96 128 160 192 224 256"`. This mechanism can be used as a shorthand when there is a need to iterate over many values separated by a fixed increment.

Once all parameters are parsed, the script will generate a sequence of test cases corresponding to all the possible combinations among all the possible values for each parameters. It will then perform each test one by one, organizing test results in a directory tree resembling the following structure:

```
bst-size/vswitch/pkt-size/rate/
```

Table 4.5: List of parameters accepted by the multiple tests configuration script. A detailed description of the difference between list and sequence parameters can be found in Section 4.3.3.

Parameter	Has Argument	Description
`-r` `--rate-params`	YES	The desired sending rate; as parameter it takes either a list or a specification of a sequence of values
`-p` `--pkt-size-params`	YES	The packet size; as parameter it takes either a list or a specification of a sequence of values
`-b` `--bst-size-params`	YES	The burst size; as parameter it takes either a list or a specification of a sequence of values
`-R` `--use-raw-sock`	NO	Same as Table 4.2
`-B` `--use-block-sock`	NO	Same as Table 4.2
`-c` `--consume-data`	NO	Same as Table 4.2
`-m` `--use-multi`	NO	Same as Table 4.2
`-t` `--timeout`	YES	Same as Table 4.2
`-v` `--vswitch-list`	YES	A list of virtual switches to use, see Table 4.2 for the possible values
`-f` `--conf-filename`	YES	Same as Table 4.2
`-D` `--debug`	NO	Same as Table 4.2
`-H` `--hostname`	YES	Same as Table 4.4; if not provided the test will be performed only on the local host
`-F` `--remote-filename`	YES	Same as Table 4.4

Chapter 5

Experimental Results

To test the functionality of the developed framework we performed a set of experiments in a real use-case scenario. The experiments were also carried out as a way to compare the various available technologies and formulate some general considerations.

5.1 Platform Description and Test Set-Up

Experiments were performed on two similar hosts. The first one has been used for all local inter-container communication tests, while the latter has been used in combination with the first for multi-host communication tests (using containers as well).

The two hosts are two Dell PowerEdge R630 V4 servers, each equipped with two Intel® Xeon® E5-2640 v4 CPUs at 2.40 GHz, respectively 64 and 128 GB of RAM, and an Intel® X710 DA2 Ethernet Controller for 10 GbE SFP+ each (used in SR-IOV experiments and multi-host scenarios). The two Ethernet controllers have been connected directly with a 10 Gigabit Ethernet cable. Both hosts are configured with Ubuntu 18.04.3 LTS, Linux kernel version 4.15.0-54, DPDK version 19.05, OVS version 2.11.1, Snabb version 2019.01, and VPP version 19.08.

To maximize results reproducibility, the framework carries out each test disabling CPU frequency scaling (governor set to performance and Turbo Boost disabled) and it has been configured to avoid using hyperthreads to deploy each testing application.

5.1.1 Testing Parameters

Evaluation of system performance is carried out by varying a set of parameters accepted by the framework via its command line tools. In particular, the multiple tests configuration script (`run_testset`) has been used multiple

times to iterate over a certain set of parameters for each fixed configuration. Each configuration is represented by a set of containers and testing applications deployed on one or two hosts and grouped in application pairs (e.g. sender/receiver or client/server).

To easily identify which test is referred in this dissertation, to each different parameters is assigned a symbol and a set of possible values. Table 5.1 contains the description of each parameters, as well as what is the meaning of the values that may be used. Using that notation, a test between a single sender and a receiver application deployed on the same host and exchanging 64 bytes packets at 5 Mpps in bursts of 32 packets using VPP as virtual switch can be expressed for example with the following tuple:

$$(D = throughput, L = local, S = 1vs1, V = vpp, P = 64, R = 5M, B = 32)$$

When showing test results, some of these parameters may be left free to vary within a range of values. In that case, any non-explicitly set value in a tuple means that that parameters is a free parameter of a certain set of tests. For example, when showing the throughput achieved when varying the sending rate, the R parameter will not be included within the tuple.

5.2 Kernel-Based Networking Performance

The first result we want to show is that performance achieved without using *kernel bypass* techniques are less than satisfactory to achieve the requirements of many NFV scenarios, while techniques like *vhost-user*, DPDK and SR-IOV offloading are able to achieve much higher performance on similar set-ups. For example, when exchanging 64 bytes packets between a single pair of sender and a receiver applications all DPDK-based solutions are able to achieve at least a throughput of 2 Mpps (see Fig. 5.1a), while traditional sockets are not able to reach even 1 Mpps.

Table 5.2 reports the maximum throughput achieved using the traditional socket APIs. As shown, the maximum performance obtained using sockets is achieved bypassing part of the network stack using raw sockets and sending packets in bursts; the burst size is also a significant factor when determining maximum performance, as smaller burst sizes are negatively affected by the higher number of system calls performed to exchange data. In general, achieved performance is significantly impaired when a virtual Ethernet port is used to connect two containers via *linux-bridge* with respect to techniques that can bypass the Linux kernel. That is why all the

Table 5.1: List of parameters used to run performance tests with the framework.

Parameter	**Symbol**	**Description**
Test Dimension	D	Whether the test is done to evaluate *throughput* or *latency* performance.
Hosts Used	L	Whether tests are limited to only "*local*" communications or multi-host communications are tested (indicated as "*remote*").
Containers Set	S	The number of container pairs deployed on the host for the test duration; the values are expressed as "$NvsN$", for example "$1vs1$" means that there will be two containers in a pair, while "$4vs4$" means four pairs of containers are deployed.
Virtual Switch	V	The virtual switch used to connect the containers; can be one among *linux-bridge*, $basicfwd$ (for the Basic Forwarding Sample Application), *ovs*, *snabb*, *sriov*, *vpp*.
Packet Size	P	The size of each packet in bytes.
Sending Rate	R	The desired sending rate from either sender or client applications, expressed in packets per second.
Burst Size	B	The number of packets per burst.

Table 5.2: Maximum throughput achieved for various socket-based solutions.
($D = throughput, L = local, S = 1vs1, V = linux\text{-}bridge, P = 64, R = 1M, B = 64$)

Technique	**Max Throughput** (kpps)
UDP sockets using `send`/`recv`	338
UDP sockets using `sendmmsg`/`recvmmsg`	409
Raw sockets using `send`/`recv`	360
Raw sockets using `sendmmsg`/`recvmmsg`	440

results presented in the following sections refer to *kernel bypass* technologies only.

5.3 Throughput Performance Evaluation

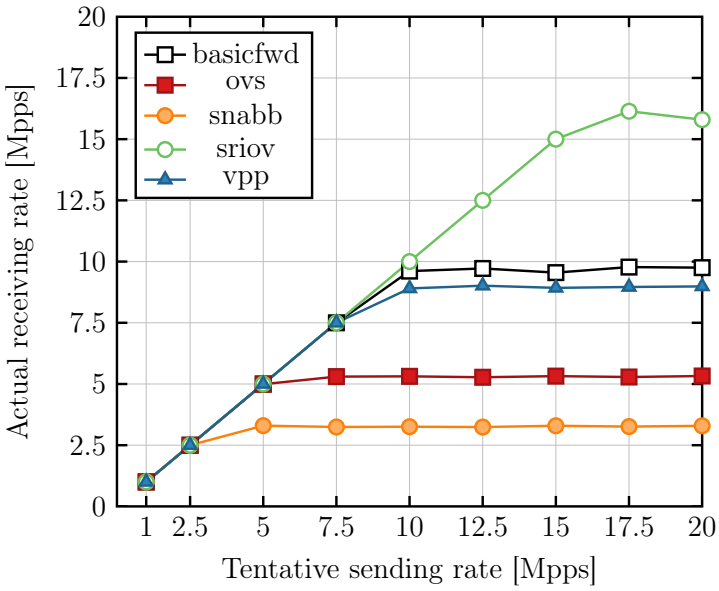
Switching the to networking technologies based on *kernel bypass*, we first evaluated how each of these technologies perform when increasing the desired sending rate or the packet size between different tests with a fixed burst size. We performed these tests both for single host inter-container communications and host-to-host communications. Each test uses only a fixed set of parameters and runs for 1 minute, then an average of the measured throughput is calculated after discarding initial and final values, to ensure skipping initial warm-up and shutdown phases. Note that the standard deviation among the averaged values was below 2.5% (and around 0.5% on average) for all the runs.

Given the exponential explosion of test cases, only significative and representative results are shown in this section and following sections.

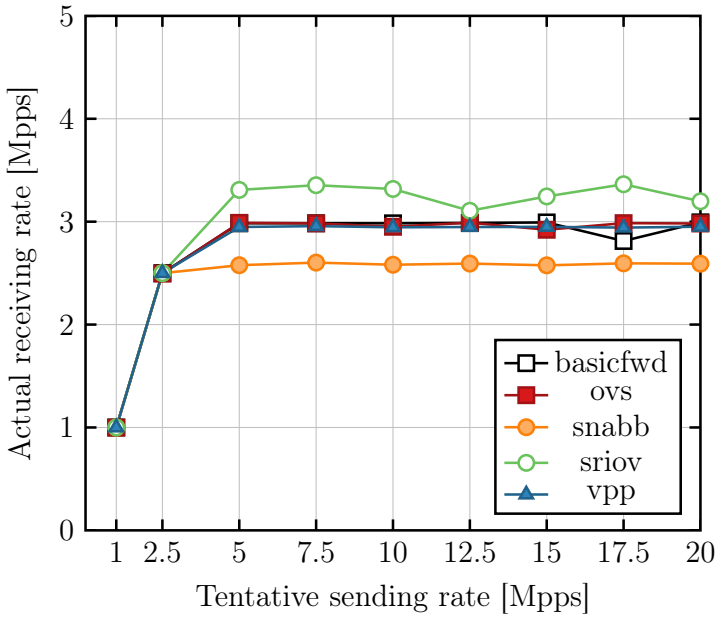
5.3.1 Single Host Throughput Performance

First we will show results achieved with a single pair of containers deployed on a single host. In these tests we varied the desired sending rate from 1 Mpps to 20 Mpps, the packet size from 64 bytes to 1500 bytes and we chose between two typical values for the burst size, 32 and 256 packets per burst:

$$(D = throughput, L = local, S = 1vs1)$$



(a) $(D = throughput, L = local, S = 1vs1, B = 32, P = 64)$



(b) $(D = throughput, L = local, S = 1vs1, B = 32, P = 512)$

Figure 5.1: Receiving rates obtained varying the sending rate for fixed packet and burst sizes on a single host.

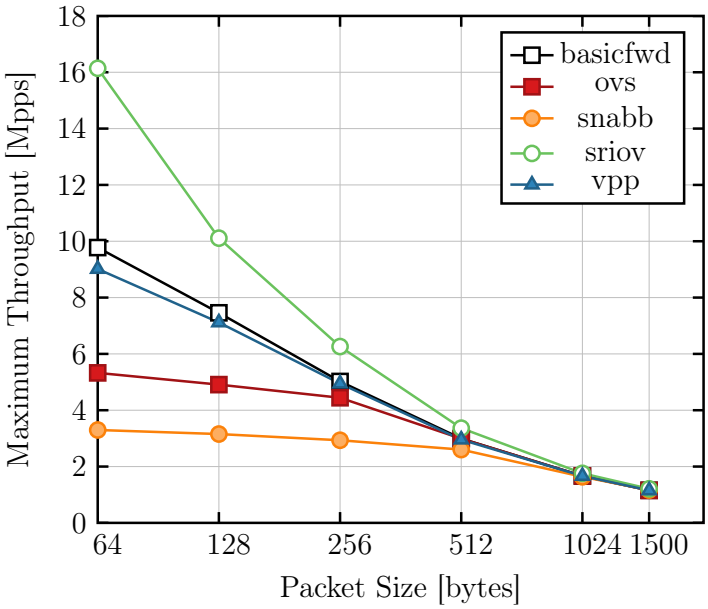
Figure 5.1 shows the achieved receiving rates in Mpps (on the Y axis) while varying the sending rate (on the X axis) for a fixed burst size of 32 packets. In the figure, each plot corresponds to a certain value of the packet size, as indicated in the respective captions. In all our tests, each networking solution is able to achieve the desired throughput up to a certain maximum value, after which any additional packet sent is dropped before reaching its destination. This is a common characteristic among all our evaluations, thus we will move the discussion to the maximum achievable throughput for each networking solution, varying the other parameters.

Figure 5.2 shows the maximum receiving rates achieved in all our tests that employed only two containers on a single host for a fixed burst size of 32 packets. In each plot, the achieved rate is expressed on the Y axis (either in Mpps or in Gbps) as function of the size of each packet on the X axis, for which a logarithmic scale has been used.

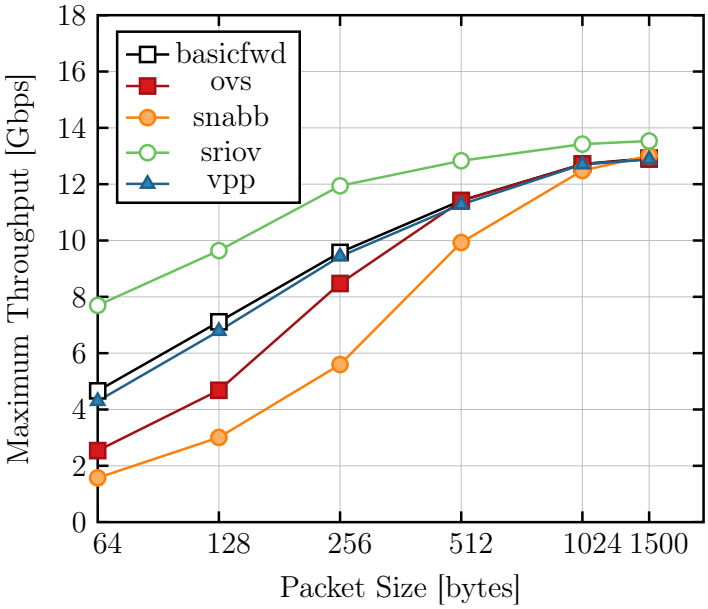
Among the solutions that use *kernel bypass* techniques, the maximum performance for these settings are achieved offloading network traffic to the SR-IOV enabled Ethernet controller, taking advantage of its embedded hardware switch. The Basic Forwarding Sample Application ranked second overall and, as expected, first among the *virtio*-based networking solutions, followed right after by VPP. It must be noted however that the former virtual switch cannot be used in real use-case scenarios, as it does not implement any real switching logic, it just forwards all the packets received from a given port to another statically assigned port. The very small performance gap between the Basic Forwarding Sample Application and VPP shows that the batch packet processing features implemented in the latter are able to amortize most of the overhead over the large amount of packets flowing through the switch, achieving very low per-packet overhead values. Finally, OVS and Snabb follow.

Comparing these performance with previous evaluations performed during the development of the framework [1], we were able to identify as major bottleneck for Snabb the virtual switch component included within our minimal Snabb configuration (which is also provided by Snabb). In fact, when the two *virtio* ports assigned to a Snabb instance are interconnected by a simple logical wire instead of a learning L2 switch (as in our previous evaluations [1]) the system is able to achieve much better performance, making Snabb the most efficient solution among *virtio*-based virtual switches, at least for relatively small packet sizes.

Notice that while the maximum throughput in terms of Mpps is achieved with the smallest of the packet sizes (64 bytes), when moving to Gbps as comparison unit the throughput increases almost logarithmically with the

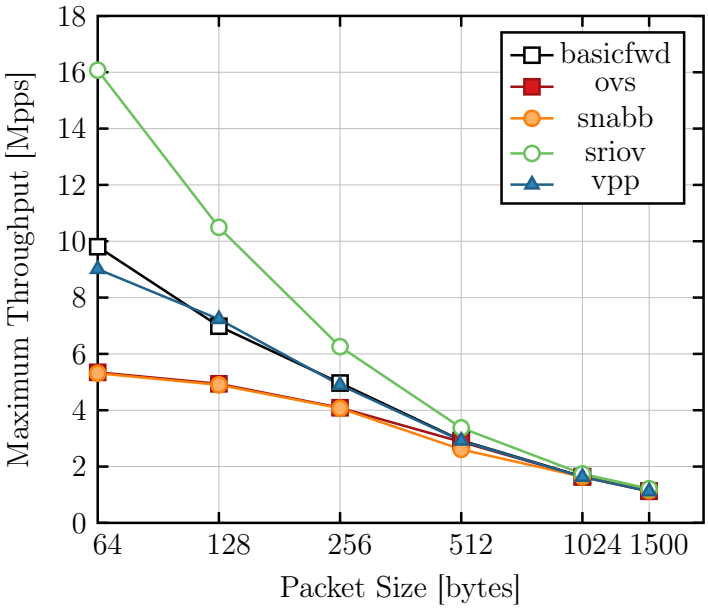


(a) $(D = throughput, L = local, S = 1vs1, B = 32)$

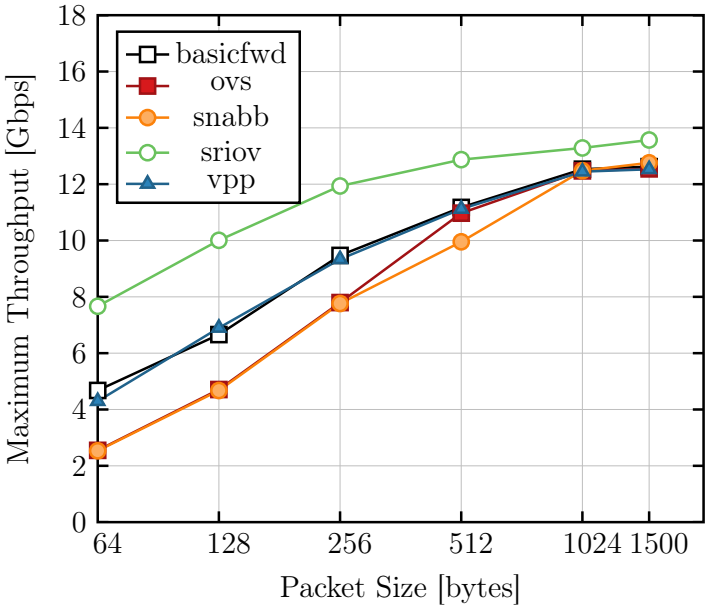


(b) $(D = throughput, L = local, S = 1vs1, B = 32)$

Figure 5.2: Maximum throughput achieved on a single host between two containers per packet size using 32 packets per burst. The two plots show the difference when evaluating throughput in Mpps instead of Gbps. Notice that for the X axis a logarithmic scale has been adopted.



(a) $(D = throughput, L = local, S = 1vs1, B = 256)$



(b) $(D = throughput, L = local, S = 1vs1, B = 256)$

Figure 5.3: Maximum throughput achieved on a single host between two containers per packet size using 256 packets per burst. The two plots show the difference when evaluating throughput in Mpps instead of Gbps. Notice that for the X axis a logarithmic scale has been adopted.

increase of the packet size for all testes solutions, as shown in Fig. 5.2b.

In both plots, it is clear how the achievable traffic by a single pair of sender/receiver applications deployed on the same host is limited by either the capability of the CPUs to move packets from a *virtio* port to another (for the software-based virtual switches) or by the limitations of the hardware SR-IOV device. Since all the *virtio*-based implementations are equivalent for bigger packet sizes in terms of throughput, we conclude that for big packet sizes the various software implementations of L2 switching functionalities are all equivalent and the only limitation is represented by how fast packets can be transferred between ports by the DPDK-based implementation of *vhost-user*. Given also the slightly superior performance achieved by SR-IOV, we also conclude that the hardware implementation of the virtual switch is more efficient at moving huge amounts of data between CPU cores than the equivalent software implementations that we tested, at least on our host.

Finally, Fig. 5.3 shows the achieved throughput when changing the burst size to 256 packets. Comparing these results with Fig. 5.2, we conclude that Snabb performance scale better with the burst size, improving up to performance very close to the what OVS achieves in the same scenario, but again VPP is the best solution among the software switches and SR-IOV is still the overall best. Their performance do not seem to be influenced much when varying the burst size from 32 packets to 256 packets.

5.3.2 Multiple Host Throughput Performance

To test inter-container communications between different hosts, we repeated the tests described in the previous section, this time deploying the sender application on one host and the receiver application on the other.:

$$(D = throughput, L = remote, S = 1vs1, V \in \{ovs, sriov, vpp\})$$

In all our tests involving multiple hosts, the two hosts communicate through a direct 10 Gigabit Ethernet cable connected to the SR-IOV Ethernet devices installed in each host. Notice that since the Basic Forwarding Sample Application application is not able to multiplex traffic from multiple input ports into a single output port, it has been excluded from out multiple host performance evaluations. Snabb has been excluded too, as it does not implement the correct drivers for our NIC device connecting the two hosts.

Our tests again show that each networking solution is able to achieve the desired throughput value up to a certain maximum, which depends on the

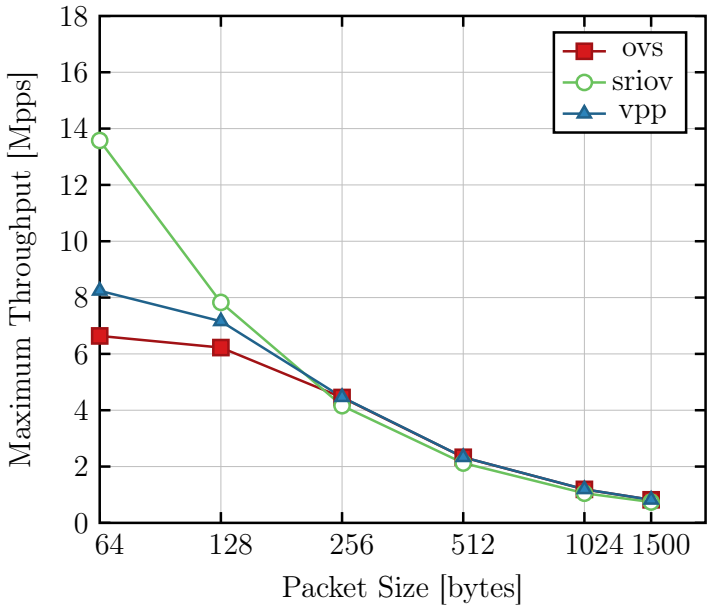
virtual switch adopted, as witnessed in our previous single host evaluations. Thus we will only consider the maximum achieved throughput in further exposition.

Figure 5.4 shows the maximum receiving rates achieved in all our tests that employed only two containers on a two hosts for a fixed burst size of 32 packets for various packet sizes. Again, in each plot a logarithmic scale has been used for the X axis.

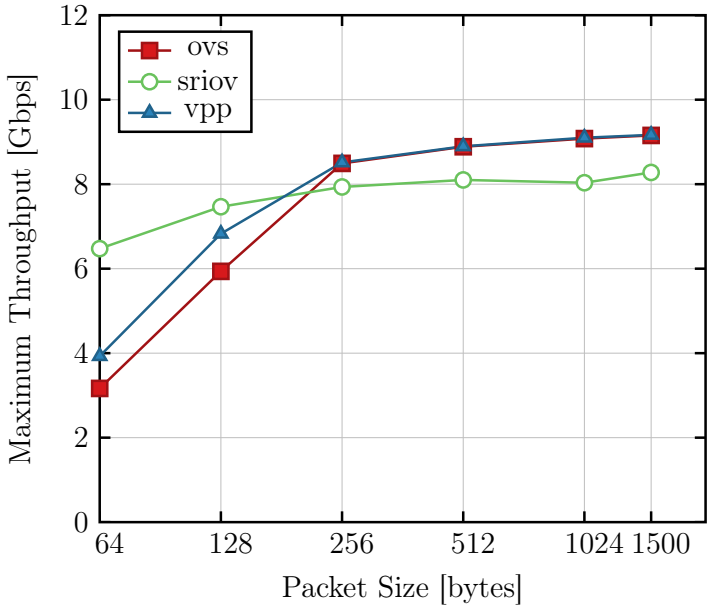
Comparing this figure to previous ones, it is clear that a big limiting factor for each of these technologies is the maximum throughput determined by the Ethernet standard; while before it was possible to exchange packets well over 10 Gbps, the limitations of the 10 Gigabit Ethernet standard now forbids this option. It is interesting to notice how this time there is no dominating solution over the others: while SR-IOV can reach higher throughput for small packet sizes, both OVS and VPP perform better for bigger ones. This could be due to the fact that when no SR-IOV VFs are instantiated the NIC device has more resources that can be used to actually send packets over the network. In this situation, packet processing operations are not performed because the next destination of each Ethernet frame is implicitly the device on the other end of the cable. Performance achieved by OVS and VPP for small packet sizes (up to 256 bytes) are exactly the same as the ones achieved in previous experiments, while for SR-IOV performance are still comparable with the previous ones; thus we conclude that when the expected traffic among a pair of components is composed by very small packets it is irrelevant whether the two components are deployed on the same host or on two (directly connected) different hosts from a throughput perspective.

5.4 Throughput Performance Scalability

While the previous tests gave us some insightful results, they only consider NFV systems composed by only a single pair of components. This is not a realistic situation for a complete NFV system because of the very low resource utilization: to fully utilize system resources a multitude components should be deployed on the same host or on multiple hosts. That's why in this section we consider some more realistic scenarios in which multiple pairs of components are deployed on one or multiple hosts.



(a) $(D = throughput, L = remote, S = 1vs1, B = 32)$



(b) $(D = throughput, L = remote, S = 1vs1, B = 32)$

Figure 5.4: Maximum throughput achieved between two containers deployed on two hosts per packet size using 32 packets per burst. The two plots show the difference when evaluating throughput in Mpps instead of Gbps. Notice that for the X axis a logarithmic scale has been adopted.

5.4.1 Single Host Throughput Performance Scalability

First we will show results achieved with a multiple pairs of containers deployed on a single host. Like in previous tests, we varied the desired sending rate from 1 Mpps to 20 Mpps, the packet size from 64 bytes to 1500 bytes and we chose between two typical values for the burst size, 32 and 256 packets per burst. However in this scenario multiple pairs of components will be deployed on the same host; in particular, we compared performance achieved in our previous $1vs1$ tests with the ones obtained with with 4 ($2vs2$) and then 8 containers ($4vs4$) running sender/receiver applications on the same host:

$$(D = throughput, L = local, S \in \{1vs1, 2vs2, 4vs4\})$$

In all test configurations, each of the sender and receiver applications will be started with the same set of parameters (e.g. using the same packet sending rate simultaneously for each sender application).

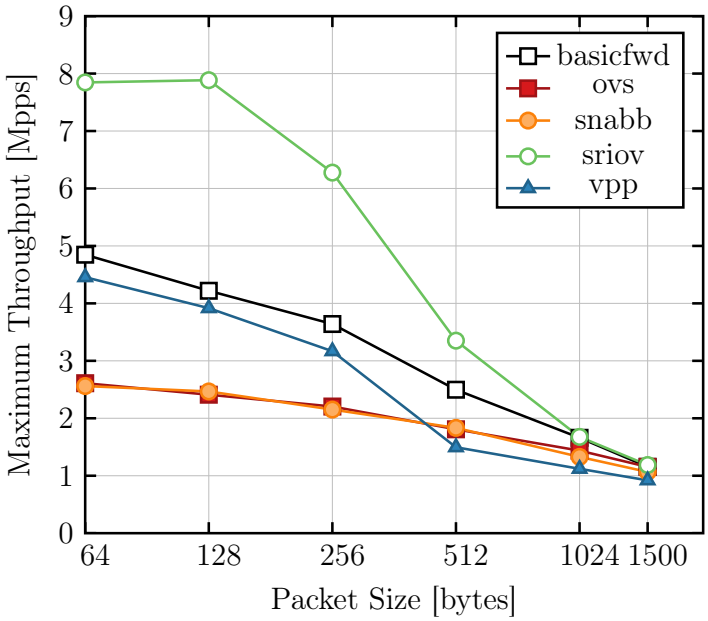
Results will be shown from two different perspectives:

- First we will consider the receiving applications point of view; in this evaluation we will consider the average of the receiving rates registered by the applications running simultaneously as our performance metric. This is the average amount of traffic that each application is able to sustain when deployed with a particular configuration.
- Then we will move on to the virtual switch point of view, in which the amount of traffic correctly handled is determined by the sum of all the packets correctly delivered to receiver applications at the same time.

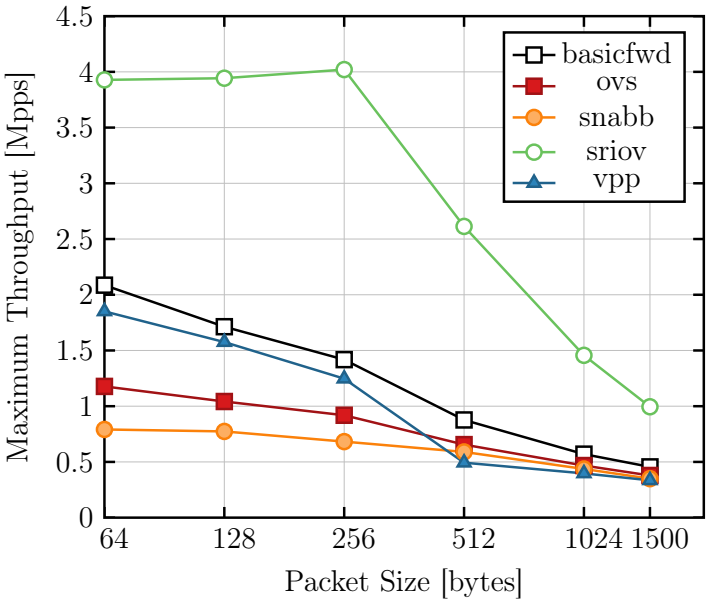
The relationship between these two perspective is quite simple, as they are simply related by the following formula, in which M is the mean of the receiving rates declared by each application, N is the number of receiver applications deployed on the same host, and T is the total throughput sustainable by the virtual switch:

$$T = M \cdot N$$

Again, in further exposition we will show only the maximum achievable throughput in each of our different test configurations as a similar behavior to previous tests has been observed when varying the sending rate.



(a) $(D = throughput, L = local, S = 2vs2, B = 32)$



(b) $(D = throughput, L = local, S = 4vs4, B = 32)$

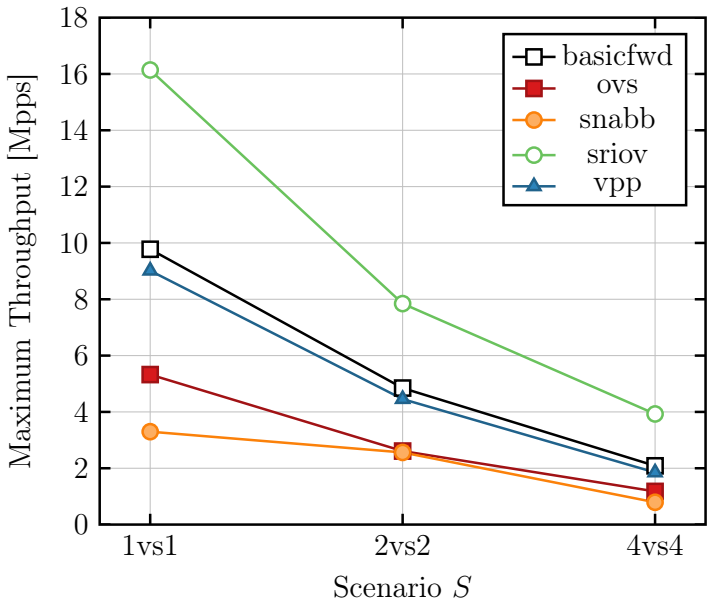
Figure 5.5: Maximum throughput achieved on a single host between two containers per packet size using 32 packets per burst. The two plots show the difference between $2vs2$ and $4vs4$ scenarios. Notice that for the X axis a logarithmic scale has been adopted.

Figure 5.5 shows the maximum average receiving rates M achieved in our tests with 4 and 8 containers respectively, expressed in Mpps, as registered by each receiving application. First we notice that the overall ranking among virtual switches remains slightly unchanged, at least for small packets. In particular, when the packet size is relatively small the total throughput of the system seems to be partitioned among all containers participating when compared to the $1vs1$ tests shown in Fig. 5.2. With the increase of the packet size however performance of each virtual switch get slightly better, although this behavior depends on the number of active participant in each test.

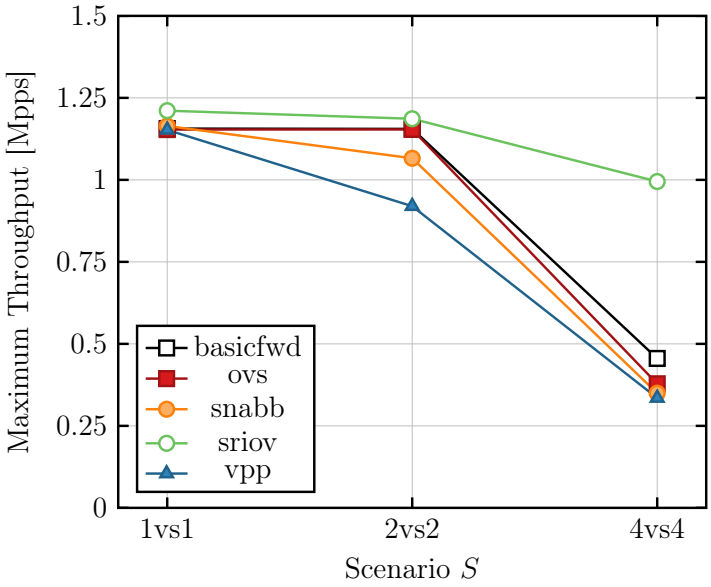
To better understand the phenomenon, Fig. 5.6 shows how the average per-application throughput (in Mpps) scales over the number of active participants in the system for each virtual switch, first with 64 bytes and then with 1500 bytes per packet. If we consider the case of 1500 bytes packets, *virtio*-based virtual switches are able to sustain 2 senders with an average performance drop for each receiver application under 20% with respect to the single sender scenario, while with SR-IOV the per-application performance drop for 2 senders is around 2%. However, when moving to 4 senders and 4 receivers, the per-application performance drop for *virtio*-based switches is 67% on average, while per-application performance with SR-IOV drops only by 18% in the same scenario.

If we consider the total throughput of the system instead of the per-application performance, it is clear how SR-IOV can scale much better with respect to the other solutions (at least for bigger packet sizes), as depicted in Fig. 5.7. While for smaller packet sizes the total throughput that all virtual switching solutions are able to process remains unchanged with the number of participants, for big packets SR-IOV is able to sustain the new traffic almost linearly with the number of participants. From these results we conclude that while for *virtio*-based solutions the major bottleneck of the system is the throughput in terms of Gbps delivered among CPU cores, for a SR-IOV device the major limitation is represented by the actual number of packets exchanged.

Finally, notice that we did not show a similar comparison for different burst sizes, because the same reasoning shown in this section for a burst size of 32 packets can be applied to 256 packets per burst, since both scenarios present very similar results.

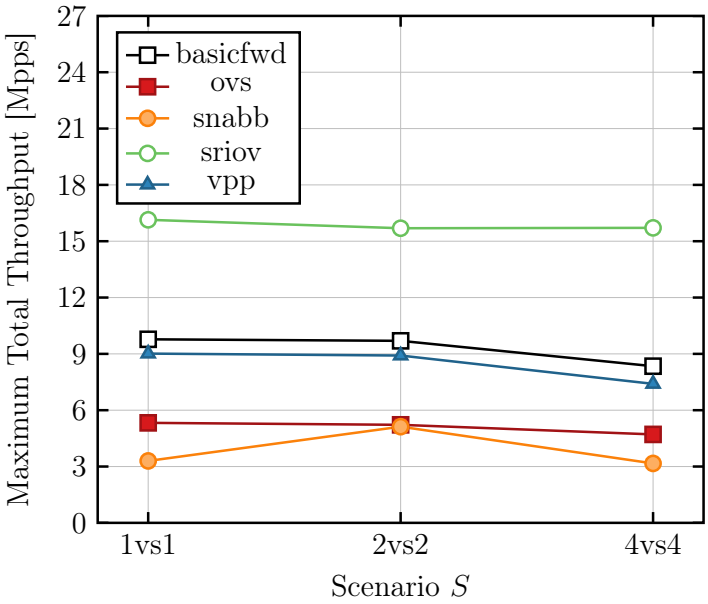


(a) $(D = throughput, L = local, P = 64, B = 32)$

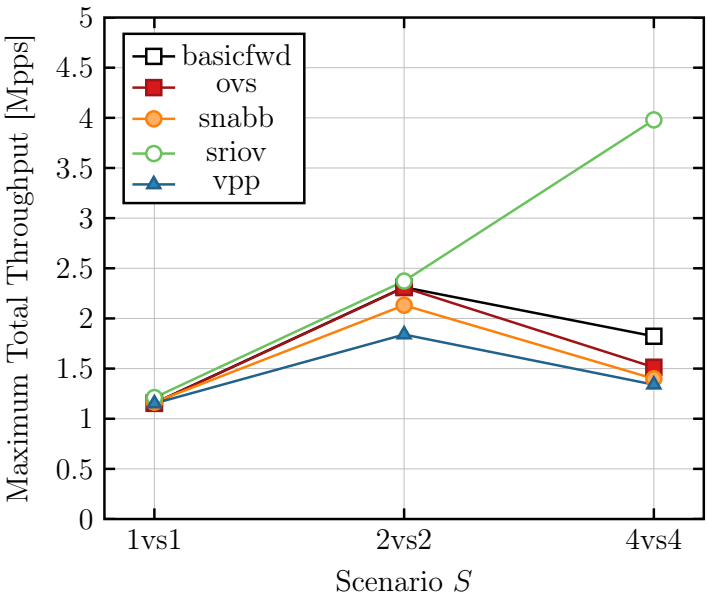


(b) $(D = throughput, L = local, P = 1500, B = 32)$

Figure 5.6: Maximum throughput achieved on a single host varying the number of simultaneous participants using packets of 64 and 1500 bytes respectively.



(a) $(D = throughput, L = local, P = 64, B = 32)$



(b) $(D = throughput, L = local, P = 1500, B = 32)$

Figure 5.7: Maximum total throughput achieved on a single host varying the number of simultaneous participants using packets of 64 and 1500 bytes respectively.

5.4.2 Multiple Hosts Throughput Performance Scalability

When deploying applications on multiple containers, we observed again slightly different behaviors depending on the packet size. For these tests we decided to increase the number of participants up to 4 containers per host (for a total of 8 containers), since the amount of resources at our disposal doubled with respect to single host evaluations:

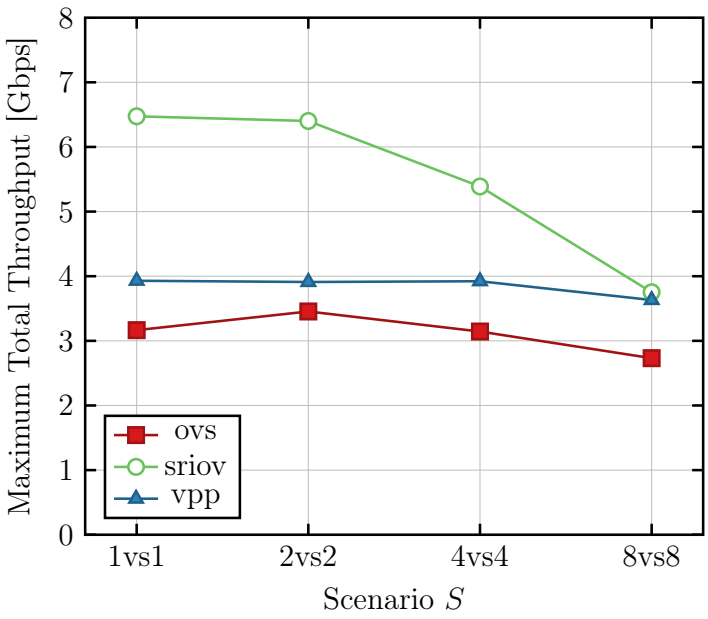
$$(D = throughput, L = remote, S \in \{1vs1, 2vs2, 4vs4, 8vs8\})$$

In this situation, the most relevant statistic is represented by the maximum total throughput of the system depending on the number of participants, depicted in Fig. 5.8. In the figure we can observe again different behaviors depending on the packet size. For bigger packets (Fig. 5.8a), the major bottleneck of the system remains the limitation of the Ethernet standard, as maximum performance achieved are not affected by the number containers deployed on both hosts. For smaller ones (Fig. 5.8a) the bottleneck depends on the virtual switch in use: while SR-IOV is still mostly subject to the limitations of the Ethernet cable, both OVS and VPP are still limited by the CPU. In fact, OVS and VPP performance remain unchanged between single host and multiple host test configurations for 64 bytes packets.

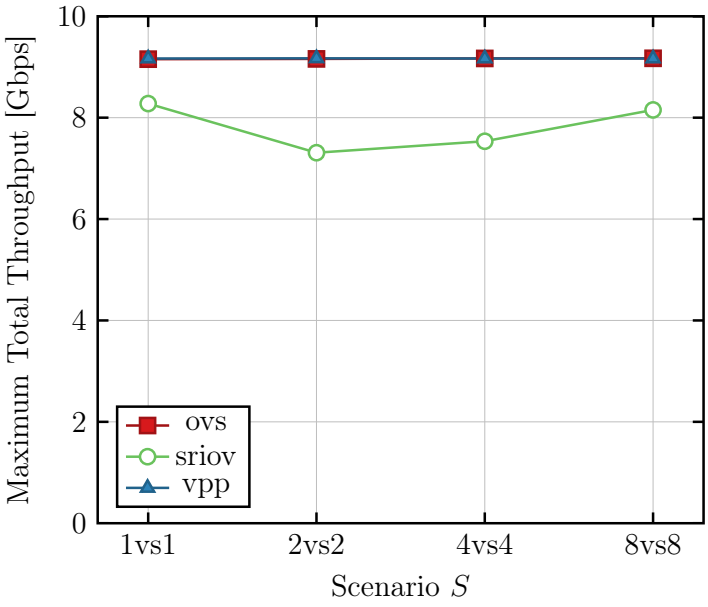
5.5 Latency Performance Evaluation

After evaluating how the various options available influence the achievable throughput we moved on to check what is the round-trip latency between two applications deployed in an NFV scenario using similar scenarios. For this kind of tests we were interested to find out the minimum latency achievable with each virtual switch to evaluate the per-packet processing overhead and how that scales when multiple packets are received in a very short span of time, by increasing the burst size.

For this purpose, a single pair of client/server applications has been deployed on a single or multiple hosts to perform each evaluation. Each test is configured to exchange a very low number of packets per second, enough so that there is no interference between the processing of a burst of packets and the following one. After some initial evaluations, we concluded that a sending rate of 1000 pps is small enough to avoid any interference while still providing enough data for each test to be statistically relevant. For example, for the minimum burst size used in our evaluations of 4 packets

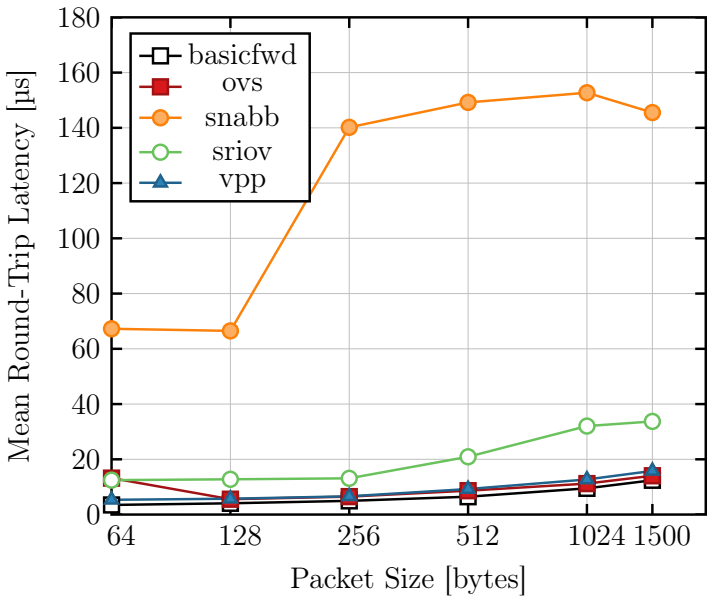


(a) $(D = throughput, L = remote, P = 64, B = 32)$



(b) $(D = throughput, L = remote, P = 1500, B = 32)$

Figure 5.8: Maximum total throughput achieved between containers deployed on different hosts varying the number of simultaneous participants. The throughput is here expressed in Gbps to compare it with the physical limit of 10 Gbps of the Ethernet standard.



$(D = latency, L = local, S = 1vs1, B = 4, R = 1000)$

Figure 5.9: Average round-trip latency achieved on a single host between two containers per packet size (including Snabb).

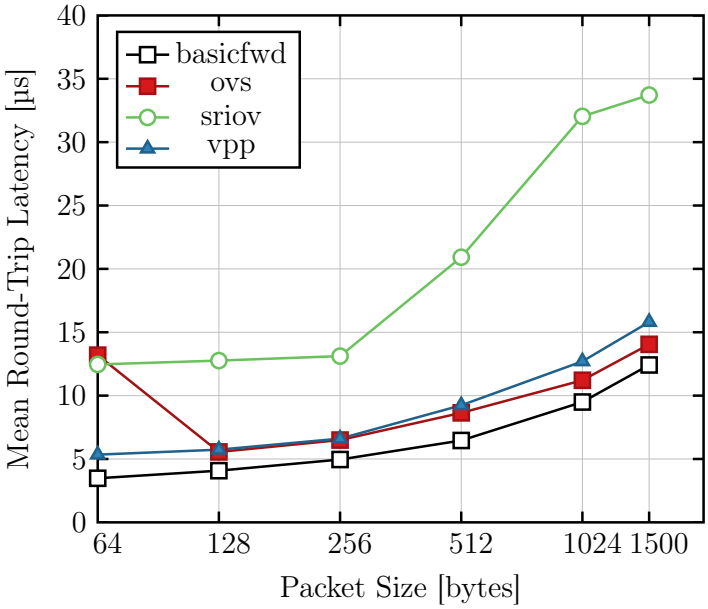
per burst, the interval between the generation of a burst of packets and the following one is 4 ms, which is much larger than the highest value of round-trip latency that we measured for that burst size. Similar results are true for bigger burst sizes.

5.5.1 Single Host Latency Performance

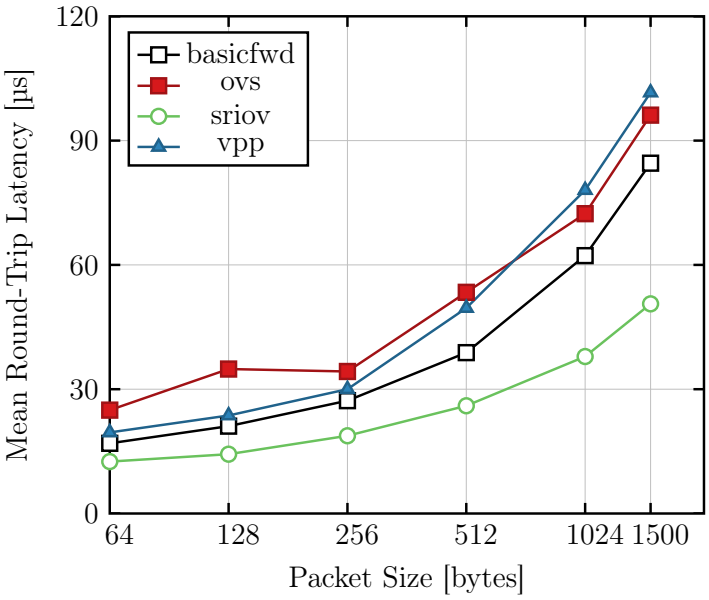
This test configuration is meant to evaluate the per-packet overhead of each virtual switch option to forward packets only on the local host; in these tests we varied both burst size from 4 packets to 128 packets per burst and the packet size from 64 bytes to 1500 bytes:

$$(D = latency, L = local, S = 1vs1, R = 1000)$$

Figure 5.9 shows the measured round-trip latency in microseconds as function of the packet size with 4 packets per burst. From the figure it is clear that Snabb performs poorly with respect to all the other virtual switches: while all the other virtual switches maintain the average delay well below 40 µs, Snabb's delay is always greater than 60 µs, even for smaller packet sizes. This is a behavior that we witnessed in all of our tests,

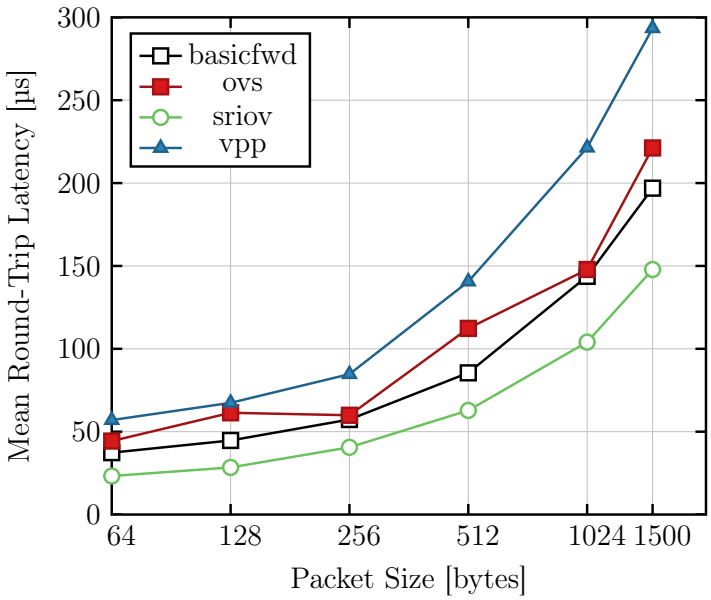


(a) $(D = latency, L = local, S = 1vs1, B = 4, R = 1000)$



(b) $(D = latency, L = local, S = 1vs1, B = 32, R = 1000)$

Figure 5.10: Average round-trip latency achieved on a single host between two containers per packet size (continues next page).



(c) $(D = latency, L = local, S = 1vs1, B = 128, R = 1000)$

Figure 5.10 (cont.): Average round-trip latency achieved on a single host between two containers per packet size.

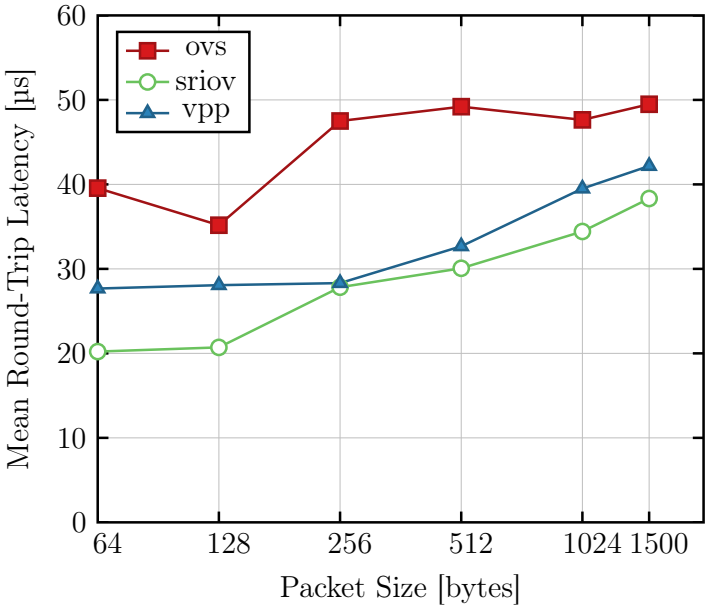
regardless of the applied parameters; that's why in all our other plots Snabb has been excluded for the sake of clarity when examining the other options.

Figure 5.10 shows the same results of Fig. 5.9 without including Snabb. The figure shows that only *virtio*-based solutions are able to achieve single-digit microsecond round-trip latency on the local host, while SR-IOV has a higher latency for small packet and burst sizes. When the burst size increases the roles are reversed: starting from 32 packets per burst, SR-IOV achieves smaller round-trip latency than software virtual switches, although it is never able to achieve a latency smaller than 10 µs. From this we infer that SR-IOV performance are less influenced by the variation of the burst size with respect to the other options available and thus more suitable when the traffic on the local machine is grouped into bigger bursts.

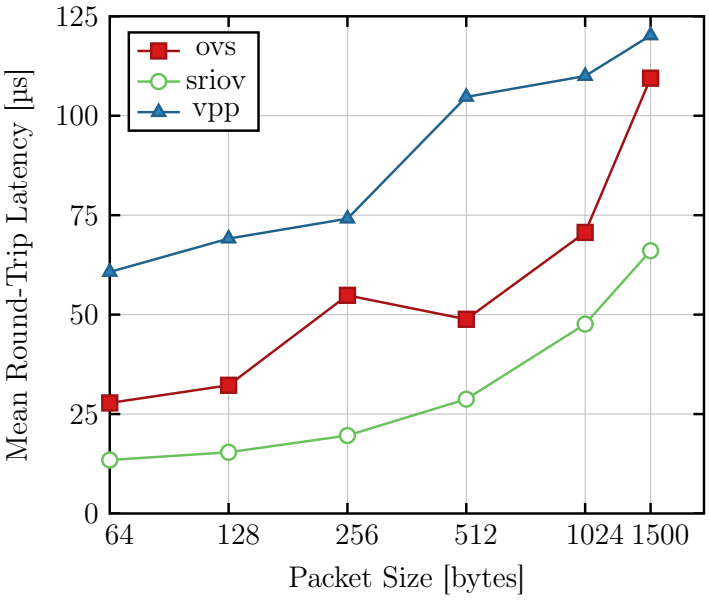
5.5.2 Multiple Host Latency Performance

The last tests we performed are meant to evaluate the differences among OVS, VPP and SR-IOV in terms of round-trip latency between two hosts, using the same parameters of local latency tests:

$$(D = latency, L = remote, S = 1vs1, R = 1000)$$

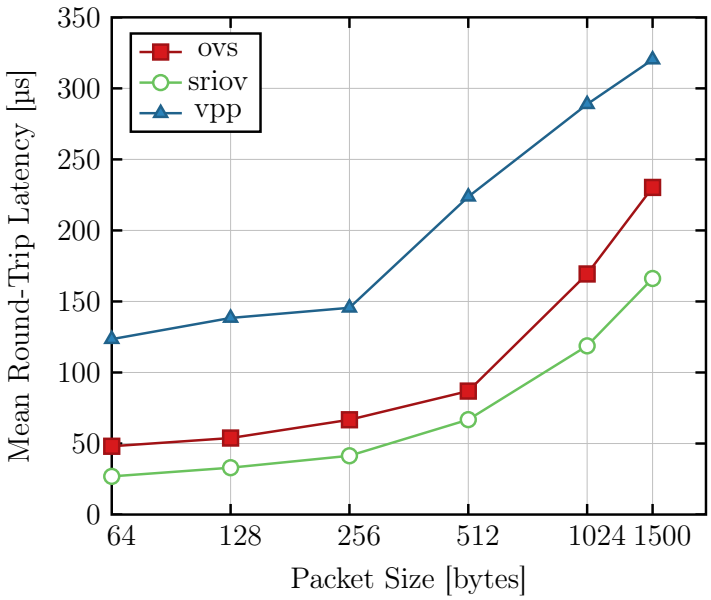


(a) $(D = latency, L = remote, S = 1vs1, B = 4, R = 1000)$



(b) $(D = latency, L = remote, S = 1vs1, B = 32, R = 1000)$

Figure 5.11: Average round-trip latency achieved on between between two containers deployed on different hosts per packet size (continues next page).



(c) $(D = latency, L = remote, S = 1vs1, B = 128, R = 1000)$

Figure 5.11 (cont.): Average round-trip latency achieved on between between two containers deployed on different hosts per packet size.

Figure 5.11 shows the measured round-trip latency in microseconds as function of the packet size between two applications deployed on separate hosts. In the figure we can notice that SR-IOV achieves lower latency with respect to OVS and VPP. While for local communications the performance of the two software switches are mostly comparable with each other, for remote communications the performance of the two switches vary depending on the burst size: while for smaller burst sizes (Fig. 5.11a) VPP has better performance, comparable with the ones achieved by SR-IOV, as the burst size increases OVS becomes the more efficient of the two from a latency point of view. In particular, for burst sizes bigger than 32 packets per burst VPP performance drop significantly with respect to the other two solutions.

Chapter 6

Conclusions

Currently, the complexity of network infrastructure management is limited by traditional approaches to implement services and network functions as highly specialized middle-boxes and other physical appliances. To overcome the ossification affecting Internet infrastructure, new approaches that make use of virtualization techniques typically adopted within cloud infrastructures are being developed to deploy more easily new functions and increase the dynamicity of network infrastructures as a whole. This transition from a hardware-oriented approach to software-based network functions using virtualization techniques has been subject of many studies over the past few years and it has been demonstrated to be effective to reduce costs of operations and management of network infrastructures.

The new opportunities and challenges presented by this transition towards NFV based approaches demonstrated that full virtualization techniques, relying on VMs, introduce too much overheads on network communications to actually allow for a wide adoption of NFV. In particular, as long as both PNFs and VNFs will coexist within the same infrastructures overheads introduced by VMs will limit the effectiveness of these new approaches in reducing costs while maintaining similar service levels.

To tackle these problems, most systems that adopt NFV as networking paradigm use OS containers as virtualization providers, which are far more lightweight than VMs and thus allow VNFs to match the desired performance. In addition, user-space implementations of network protocols and device drivers enhance network performance bypassing the kernel for both local and remote communications. The diffusion of open-source frameworks like DPDK allowed developers to write portable applications that can achieve very high levels of performance on a multitude of supported platforms, paving the way for a growing number of network functions to be implemented as software network functions.

The introduction of a number of software network switches implementations (e.g. Open vSwitch), as well as new devices with hardware sup-

port for virtual switching, like SR-IOV Ethernet controllers, allow multiple VNF instances to be deployed on the same host, enabling greater resource utilization. Given the central role of these virtual switches in NFV infrastructures, it is essential to analyze the performance of these solutions with respect to each other when transitioning to the new NFV approach.

This thesis focused on the design and implementation of a software framework aimed to evaluate and compare network performance of various virtual networking solutions that can be adopted to connect various VNFs deployed as Linux containers in one or multiple hosts. This new tool can be easily deployed on private cloud infrastructures to evaluate which solution is the most suitable to be used in a real production environment.

To illustrate the functionalities of this framework, a number of tests and performance evaluations of commonly used virtual switches have been performed with synthetic workloads emulating real use-case scenarios. From these tests, we concluded that the internal switch of SR-IOV network devices is more efficient than other solutions when communication between multiple applications deployed on the same host is a key priority; it is both able to sustain higher throughput per applications, especially for bigger packet sizes, and it scales very efficiently with the number of applications deployed on the same system. In addition, SR-IOV devices have the advantage of not requiring additional CPU resources to effectively support multiple network streams, needing only one CPU core for operations and management of VFs. In similar scenarios, VPP is the most efficient among software-based virtual switches, but it needs more computational power than SR-IOV (since packet processing operations are implemented in software) and it cannot scale along with the number of streams on the local host unless additional CPUs are dedicated for its packet processing operations. Same reasoning applies to other software-based virtual switches.

On the other hand, when traffic is forwarded from one host to another the efficiency of any virtual switch is mostly limited by throughput defined by the Ethernet standard. In this case, both SR-IOV and software-based virtual switches show similar performance, although the latter seem to attain better results within the limits imposed by the Ethernet standard.

Finally, from a latency perspective we demonstrated that both for local and remote communications SR-IOV can attain smaller round-trip latency with respect to the other solutions with bigger burst sizes. Among software virtual switch implementations, we show that Snabb is highly inadequate with respect to the other solutions available for local communications, but there is no dominating solution with respect to the others, as their performance depend strongly on the burst size and the other parameters used to

generate synthetic network traffic.

6.1 Future Work

Despite the customizability of this framework, thanks to its many parameters that can be used for both installation and performance testing, one key functionality that lacks implementation is the capability to change the amount of resources allocated to each virtual switch per test run. Right now, each virtual switch is configured to use a static number of cores. This is particularly important for all the software-based implementations (such as OVS or VPP) which can spawn multiple worker threads on more than one CPU core; each of these worker threads can contribute to the processing of the traffic flowing through the switch, thus the total throughput of each switch may vary depending on the amount of processing resources allocated. It is important to notice however that while allocating more resources to each switch can potentially improve network performance, it also steals processing power from applications, reducing the number of VNFs that can be deployed on the same machine. An evaluation of this performance/scalability trade-off is necessary to efficiently design a networking infrastructure.

During the development of the framework we repeated performance evaluations multiple times, each time we upgraded one or multiple components to their most recent stable versions. From these evaluations we noticed that these technologies are steadily evolving over time, with each new version being more efficient than the previous one. That is also why it could be interesting to repeat again these performance evaluations in the future for new versions of the software switches, checking whether they can reach the superior performance achieved only by SR-IOV in our evaluations.

Finally, while the framework developed for this thesis can be already used to effectively compare and evaluate some of the most commonly adopted switching solutions in the industry, some other technologies have been postponed for future implementation and analysis. In fact, the framework is only compatible with *virtio*-based software switches or NICs that implement SR-IOV functionalities, but not all options available on the market make use of these two technologies. Examples of technologies that the framework is planned to support include NetVM [79] and the Netmap framework [55] (along with its associated virtual switch, VALE [80]).

Acronyms

Acronyms	Full names	Acronyms	Full names
CAPEX	Capital Expenditure	COTS	Commercial Off-the-Shelf
DPDK	Data Plane Development Kit	EAL	Environment Abstraction Layer
ETSI	European Telecommunications Standards Institute	FD.io	Fast Data Project
HPC	High Performance Computing	IaaS	Infrastructure as a Service
IDS	Intrusion Detection System	IP	Internet Protocol
ISG	Industry Specification Group	LXC	Linux Containers
MAC	Media Access Control	MANO	Management and Orchestration
N-PoP	Network Point of Presence	NAPI	Linux New API
NAS	Network Attached Storage	NAT	Network Address Translator
NFV	Network Function Virtualization	NFVI	Network Function Virtualization Infrastructure
NFVO	NFV Orchestrator	NIC	Network Interface Controller
NUMA	Non-Uniform Memory Access	OPEX	Operating Expense
OS	Operating System	OVS	Open vSwitch
PaaS	Platform as a Service	PF	Physical Function

(Continues on next page)

Acronyms	Full names	Acronyms	Full names
PMD	Poll Mode Driver	PNF	Physical Network Function
QoS	Quality of Service	RDMA	Remote Direct Memory Access
SAN	Storage Area Network	SDN	Software-Defined Networking
SR-IOV	Single-Root I/O Virtualization	TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer	TSC	Time Stamp Counter
UDP	User Datagram Protocol	VF	Virtual Function
VIM	Virtual Infrastructure Manager	VM	Virtual Machine
VMM	Virtual Machine Manager	VNF	Virtualized Network Function
VNFC	Virtual Network Function Component	VNFM	VNF Manager
VPP	Vector Packet Processing		

Bibliography

[1] Gabriele Ara, Luca Abeni, Tommaso Cucinotta, and Carlo Vitucci. On the use of kernel bypass mechanisms for high-performance inter-container communications. In *Lecture Notes in Computer Science*. Springer International Publishing, June 2019.

[2] Giorgos Papastergiou, Gorry Fairhurst, David Ros, Anna Brunstrom, Karl-Johan Grinnemo, Per Hurtig, Naeem Khademi, Michael Tuxen, Michael Welzl, Dragana Damjanovic, and Simone Mangiante. De-ossifying the internet transport layer: A survey and future perspectives. *IEEE Communications Surveys & Tutorials*, 19(1):619–639, 2017. doi:10.1109/comst.2016.2626780.

[3] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud. *ACM SIGCOMM Computer Communication Review*, 39(1):68, December 2008. doi:10.1145/1496091.1496103.

[4] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network Function Virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, February 2015. doi:10.1109/mcom.2015.7045396.

[5] Docker. `https://www.docker.com/`. [Online] Accessed September 5, 2019.

[6] Linux Containers (LXC). `https://linuxcontainers.org/`. [Online] Accessed September 5, 2019.

[7] Nane Kratzke. Smuggling multi-cloud support into cloud-native applications using elastic container platforms. In *Proceedings of the 7th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2017. doi:10.5220/0006230700570070.

[8] DPDK. `https://www.dpdk.org/`. [Online] Accessed September 5, 2019.

[9] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with SR-IOV. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, November 2012. doi:10.1016/j.jpdc.2012.01.020.

[10] Korian Edeline and Benoit Donnet. Towards a middlebox policy taxonomy: Path impairments. In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, April 2015. doi:10.1109/infcomw.2015.7179418.

[11] Brian Carpenter and Scott Brim. Middleboxes: Taxonomy and issues. Technical report, RFC 3234, February, 2002.

[12] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem. *ACM SIGCOMM Computer Communication Review*, 42(4):13, September 2012. doi:10.1145/2377677.2377680.

[13] Bo Yi, Xingwei Wang, Keqin Li, Sajal k. Das, and Min Huang. A comprehensive survey of Network Function Virtualization. *Computer Networks*, 133:212–262, March 2018. doi:10.1016/j.comnet.2018.01.021.

[14] Antonio Manzalini, Roberto Minerva, Franco Callegati, Walter Cerroni, and Aldo Campi. Clouds of virtual machines in edge networks. *IEEE Communications Magazine*, 51(7):63–70, July 2013. doi:10.1109/mcom.2013.6553679.

[15] ETSI. Network Functions Virtualisation. White Paper 1, SDN and Openflow World Congress, Darmstadt, Germany, 2012. URL `https://portal.etsi.org/NFV/NFV_White_Paper.pdf`.

[16] Haneul Ko, Giwon Lee, Insun Jang, and Sangheon Pack. Optimal middlebox function placement in virtualized evolved packet core systems. In *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, August 2015. doi:10.1109/apnoms.2015.7275380.

[17] China Mobile Research Institute. C-RAN: the road towards green RAN. White paper, ver 2, 2011. URL `https://pdfs.semanticscholar.org/eaa3/ca62c9d5653e4f2318aed9ddb8992a505d3c.pdf`.

[18] Kostas Pentikousis, Yan Wang, and Weihua Hu. Mobileflow: Toward software-defined mobile networks. *IEEE Communications Magazine*, 51(7):44–53, July 2013. doi:10.1109/mcom.2013.6553677.

[19] ETSI. NFV: Use Cases. White Paper, ETSI, 2017. URL http://www.etsi.org/deliver/etsi_gr/NFV/001_099/001/01.02.01_60/gr_NFV001v010201p.pdf.

[20] ETSI. Network Functions Virtualisation. White Paper 2, SDN and Openflow World Congress, Frankfurt, Germany, 2013. URL http://portal.etsi.org/NFV/NFV_White_Paper2.pdf.

[21] ETSI. Network Functions Virtualisation. White Paper 3, SDN and Openflow World Congress, Dusseldorf, Germany, 2014. URL http://portal.etsi.org/NFV/NFV_White_Paper3.pdf.

[22] ETSI. Network Function Virtualization (NFV): Management and Orchestration. White Paper, ETSI, 2014. URL https://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf.

[23] ETSI. NFV: Architecture Framework. White Paper, ETSI, 2014. URL http://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.02.01_60/gs_nfv002v010201p.pdf.

[24] ETSI. Network Function Virtualization (NFV): Infrastructure Overview. White Paper, ETSI, 2014. URL https://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/001/01.01.01_60/gs_NFV-INF001v010101p.pdf.

[25] ETSI. Network Function Virtualization (NFV): Infrastructure – Compute Domain. White Paper, ETSI, 2014. URL https://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/003/01.01.01_60/gs_NFV-INF003v010101p.pdf.

[26] ETSI. Network Function Virtualization (NFV): Infrastructure – Hypervisor Domain. White Paper, ETSI, 2014. URL https://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/004/01.01.01_60/gs_NFV-INF004v010101p.pdf.

[27] ETSI. Network Function Virtualization (NFV): Infrastructure – Network Domain. White Paper, ETSI, 2014. URL https://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/005/01.01.01_60/gs_NFV-INF005v010101p.pdf.

[28] ETSI. Network Function Virtualization (NFV): Service Quality Metrics. White Paper, ETSI, 2014. URL https://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/010/01.01.01_60/gs_NFV-INF010v010101p.pdf.

[29] ETSI. Network Function Virtualization (NFV): Resiliency Requirements. White Paper, ETSI, 2014. URL http://www.etsi.org/deliver/etsi_gs/NFV-REL/001_099/001/01.01.01_60/gs_nfv-rel001v010101p.pdf.

[30] ETSI. Network Function Virtualization (NFV): Security and Trust Guidance. White Paper, ETSI, 2014. URL https://www.etsi.org/deliver/etsi_gs/NFV-SEC/001_099/003/01.01.01_60/gs_NFV-SEC003v010101p.pdf.

[31] Open Virtualization Alliance. KVM. http://www.linux-kvm.org. [Online] Accessed September 5, 2019.

[32] Citrix. Citrix Hypervisor (formerly XenServer. https://www.citrix.com/products/citrix-hypervisor. [Online] Accessed September 5, 2019.

[33] VMware. vSphere. https://www.vmware.com/products/vsphere. [Online] Accessed September 5, 2019.

[34] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, March 2015. doi:10.1109/ispass.2015.7095802.

[35] David Beserra, Edward David Moreno, Patricia Takako Endo, Jymmy Barreto, Djamel Sadok, and Stenio Fernandes. Performance analysis of LXC for HPC environments. In *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*. IEEE, July 2015. doi:10.1109/cisis.2015.53.

[36] Miguel G. Xavier, Israel C. De Oliveira, Fabio D. Rossi, Robson D. Dos Passos, Kassiano J. Matteussi, and Cesar A.F. De Rose. A performance isolation analysis of disk-intensive workloads on container-based clouds. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, March 2015. doi:10.1109/pdp.2015.67.

[37] Rabindra K. Barik, Rakesh K. Lenka, K. Rahul Rao, and Devam Ghose. Performance analysis of virtual machines and containers in cloud computing. In *2016 International Conference on Computing, Communication and Automation (ICCCA)*. IEEE, April 2016. doi:10.1109/ccaa.2016.7813925.

[38] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support PaaS. In *2014 IEEE International Conference on Cloud Engineering*. IEEE, March 2014. doi:10.1109/ic2e.2014.41.

[39] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of amazon EC2 data center. In *2010 Proceedings IEEE INFOCOM*. IEEE, March 2010. doi:10.1109/infcom.2010.5461931.

[40] Hagen Woesner and David Verbeiren. SDN and NFV in telecommunication network migration. In *2015 Fourth European Workshop on Software Defined Networks*. IEEE, September 2015. doi:10.1109/ewsdn.2015.80.

[41] Reece Johnston, Sun il Kim, David Coe, Letha Etzkorn, Jeffrey Kulick, and Aleksandar Milenkovic. Xen network flow analysis for intrusion detection. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference on - CISRC '16*. ACM Press, 2016. doi:10.1145/2897795.2897802.

[42] Naylor G. Bachiega, Paulo S. L. Souza, Sarita M. Bruschi, and Simone do R. S. de Souza. Container-based performance evaluation: A survey and challenges. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, April 2018. doi:10.1109/ic2e.2018.00075.

[43] OpenVZ. https://openvz.org/. [Online] Accessed September 5, 2019.

[44] Amazon Elastic Container Service (ECS). https://aws.amazon.com/ecs. [Online] Accessed September 5, 2019.

[45] Google Kubernetes Engine. https://cloud.google.com/kubernetes-engine. [Online] Accessed September 5, 2019.

[46] Kata containers. https://katacontainers.io/. [Online] Accessed September 5, 2019.

[47] Anil Madhavapeddy and David J Scott. Unikernels: Rise of the virtual library operating system. *ACM Queue*, 11(11):30, 2013. URL https://queue.acm.org/detail.cfm?id=2566628.

[48] Unikernel. http://unikernel.org/. [Online] Accessed September 5, 2019.

[49] L. Deri. nCap: wire-speed packet capture and transmission. In *Workshop on End-to-End Monitoring Techniques and Services*. IEEE, 2005. doi:10.1109/e2emon.2005.1564468.

[50] Luigi Rizzo. Revisiting network I/O APIs: The Netmap framework. *Queue*, 10(1):30, January 2012. doi:10.1145/2090147.2103536.

[51] Rusty Russell. VIRTIO: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008. doi:10.1145/1400097.1400108.

[52] Rusty Russell, Michael S. Tsirkin, Cornelia Huck, and Pawel Moll. Virtual I/O Device (VIRTIO) Version 1.0. Standard, OASIS Specification Committee, 2015. URL http://docs.oasis-open.org/virtio/virtio/v1.0/cs04/virtio-v1.0-cs04.html.

[53] Michael S. Tsirkin. vhost-net and virtio-net: Need for Speed. In *Proc. of the KVM Forum*, May 2010.

[54] Michele Paolino, Nikolay Nikolaev, Jeremy Fanguede, and Daniel Raho. SnabbSwitch user space virtual switch benchmark and performance optimization for NFV. In *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE, November 2015. doi:10.1109/nfv-sdn.2015.7387411.

[55] Luigi Rizzo. Netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, 2012. USENIX Association. ISBN 978-931971-93-5. URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo.

[56] Luigi Rizzo. OVS: Accelerating the datapath through Netmap/VALE. In *Open vSwitch 2014 Fall Conference*, 2014. URL http://openvswitch.org/support/ovscon2014/18/1630-ovs-rizzo-talk.pdf.

[57] Intel. Impressive packet processing performance enables greater workload consolidation. White Paper, Intel, 2012. URL https://media15.connectedsocialmedia.com/intel/06/13251/Intel_DPDK_Packet_Processing_Workload_Consolidation.pdf.

[58] DPDK Environment Abstraction Layer. https://doc.dpdk.org/guides/prog_guide/env_abstraction_layer.html. [Online] Accessed September 5, 2019.

[59] DPDK Testpmd Application User Guide. https://doc.dpdk.org/guides/testpmd_app_ug/index.html. [Online] Accessed September 5, 2019.

[60] DPDK Basic Forwarding Sample Application. https://doc.dpdk.org/guides/sample_app_ug/skeleton.html. [Online] Accessed September 5, 2019.

[61] Linux Foundation Collaborative Project: Open vSwitch (OVS). https://www.openvswitch.org. [Online] Accessed September 5, 2019.

[62] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, 2015. URL https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff.

[63] Open Network Foundation (ONF). ONF SDN Evolution. White Paper, ONF, 2016. URL http://www.opennetworking.org/wp-content/uploads/2013/05/TR-535_ONF_SDN_Evolution.pdf.

[64] Intel. Open vSwitch enables SDN and NFV transformation. White Paper, Intel, 2015. URL https://networkbuilders.intel.com/docs/open-vswitch-enables-sdn-and-nfv-transformation-paper.pdf.

[65] Vector Packet Processing (VPP). https://fd.io/technology/. [Online] Accessed September 5, 2019.

[66] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM*

Transactions on Computer Systems, 18(3):263–297, August 2000. doi:10.1145/354871.354874.

[67] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, May 2015. doi:10.1109/ancs.2015.7110116.

[68] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. High-speed software data plane via vectorized packet processing. *IEEE Communications Magazine*, 56 (12):97–103, December 2018. doi:10.1109/mcom.2018.1800069.

[69] Snabb. https://github.com/snabbco/snabb. [Online] Accessed September 5, 2019.

[70] Luca Abeni, Csaba Kiraly, Nanfang Li, and Andrea Bianco. On the performanc of KVM-based virtual routers. *Computer Communications*, 70:40–53, October 2015. doi:10.1016/j.comcom.2015.05.005.

[71] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond Softnet. In *Annual Linux Showcase & Conference*, volume 5, pages 18–18, 2001. URL https://www.usenix.org/legacy/publications/library/proceedings/als01/full_papers/jamal/jamal.pdf.

[72] Khaled Salah and Abdulhakim Ali Qahtan. Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme. *Computer Communications*, 32(1):179–188, January 2009. doi:10.1016/j.comcom.2008.10.001.

[73] RDMA Consortium. http://www.rdmaconsortium.org. [Online] Accessed September 5, 2019.

[74] Daniel Géhberger, David Balla, Markosz Maliosz, and Csaba Simon. Performance evaluation of low latency communication alternatives in a containerized cloud environment. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, July 2018. doi:10.1109/cloud.2018.00009.

[75] Giuseppe Lettieri, Vincenzo Maffione, and Luigi Rizzo. A survey of fast packet I/O technologies for Network Function Virtualization. In *Lecture Notes in Computer Science*, pages 579–590. Springer International Publishing, 2017. doi:10.1007/978-3-319-67630-2_40.

[76] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. Comparison of frameworks for high-performance packet IO. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, May 2015. doi:10.1109/ancs.2015.7110118.

[77] PF_RING. https://www.ntop.org/products/packet-capture/pf_ring/. [Online] Accessed September 5, 2019.

[78] Nikolai Pitaev, Matthias Falkner, Aris Leivadeas, and Ioannis Lambadaris. Characterizing the performance of concurrent virtualized network functions with OVS-DPDK, FD.IO VPP and SR-IOV. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering - ICPE '18*. ACM Press, 2018. doi:10.1145/3184407.3184437.

[79] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management*, 12(1):34–47, March 2015. doi:10.1109/tnsm.2015.2401568.

[80] Luigi Rizzo and Giuseppe Lettieri. VALE, a switched ethernet for virtual machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies - CoNEXT '12*. ACM Press, 2012. doi:10.1145/2413176.2413185.