Comparative Evaluation of Kernel Bypass Mechanisms for High-performance Inter-container Communications

Gabriele Ara[1][a], Tommaso Cucinotta[1][b], Luca Abeni[1][c] and Carlo Vitucci[2]

[1] *Scuola Superiore Sant'Anna, Pisa, Italy*

[2] *Ericsson, Stockholm, Sweden*

{gabriele.ara, tommaso.cucinotta, luca.abeni}@santannapisa.it, carlo.vitucci@ericsson.com

Keywords: Kernel Bypass, DPDK, NFV, Containers, Cloud Computing.

Abstract: This work presents a framework for evaluating the performance of various virtual switching solutions, each widely adopted on Linux to provide virtual network connectivity to containers in high-performance scenarios, like in Network Function Virtualization (NFV). We present results from the use of this framework for the quantitative comparison of the performance of software-based and hardware-accelerated virtual switches on a real platform with respect to a number of key metrics, namely network throughput, latency and scalability.

1 INTRODUCTION

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.

Recently, network operators started replacing traditional physical network 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). These are equivalent implementations of the same services provided in software, often enriched with the typical elasticity of cloud applications, i.e., the ability to scale out and back in the service as needed. This brings new flexibility in physical resources management and allows the realization of more dynamic networking infrastructures.

Given the nature of the services usually deployed in NFV infrastructures, these systems must be characterized by high performance both in terms of throughput and latency among VNFs. Maintaining low communication overheads among interacting software components has become a critical issue for these systems. Requirements of NFV applications are so tight that the industry has already shifted its focus from traditional Virtual Machines (VMs) to Operating System (OS) containers to deploy VNFs, given the reduced overheads characterizing container solutions like LXC or Docker, that exhibit basically the same performance as bare-metal deployments (Felter et al., 2015; Barik et al., 2016).

Given the superior performance provided by containers when used to encapsulate VNFs, primary research focus is now into further reducing communication overheads by adopting user-space networking techniques, in combination with OS containers, to bypass the kernel when exchanging data among containers on the same or multiple hosts. These techniques are generally indicated as *kernel bypass* mechanisms.

1.1 Contributions

In this work, we propose a novel benchmarking framework that can be used to compare the performance of various networking solutions that leverage *kernel bypass* to exchange packets among VNFs deployed on a private cloud infrastructure using OS containers. Among the available solutions, this work focuses on the comparison of virtual switches based on the Data Plane Development Kit (DPDK) framework. This occupies a prominent position in industry and it

[a] https://orcid.org/0000-0001-5663-4713

[b] https://orcid.org/0000-0002-0362-0657

[c] https://orcid.org/0000-0002-7080-9601

can be used to efficiently exchange packets locally on a single machine or among multiple hosts accessing Network Interface Controller (NIC) adapters directly from the user-space bypassing the OS kernel. We present results from a number of experiments comparing the performance of these virtual switches (either software-based or hardware-accelerated) when subject to synthetic workloads that resemble the behavior of real interacting VNF components.

2 BACKGROUND

There are a number of different options when multiple applications, each encapsulated in its own OS container or other virtualized environment, need to communicate through network primitives. Usually, they involve some form of network virtualization to provide each application a set of gateways to exchange data over a virtual network infrastructure.

For the purposes of this work, we will focus on the following: **(i)** *kernel-based networking*, **(ii)** *software-based user-space networking*, **(iii)** *hardware-accelerated user-space networking*.

In the following, we briefly summarize the main characteristics of each of these techniques when adopted in NFV scenarios to interconnect OS containers within a private cloud infrastructure. Given the demanding requirements of NFV applications in terms of performance, both with respect to throughput and latency, we will focus on the performance that can be attained when adopting each solution on general-purpose computing machines running a Linux OS.

2.1 Containers Networking through the Linux Kernel

A first way to interconnect OS containers is to place virtual Ethernet ports as gateways in each container and to connect them all to a virtual switch embedded within the Linux kernel, called "*linux-bridge*". Through this switch, VNFs can communicate on the same host with other containerized VNFs or with other hosts via forwarding through actual Ethernet ports present on the machine, as shown in Figure 1a.

The virtual ports assigned to each container are implemented in the Linux kernel as well and they emulate the behavior of actual Ethernet ports. They can be accessed via blocking or nonblocking system calls, for example using the traditional POSIX Socket API, exchanging packets via `send()` and `recv()` (or their more general forms `sendmsg()` and `recvmsg()`); as a result, at least two system calls are required to exchange each UDP datagram over the virtual network, therefore networking overheads grow proportionally with the number of packets exchanged.

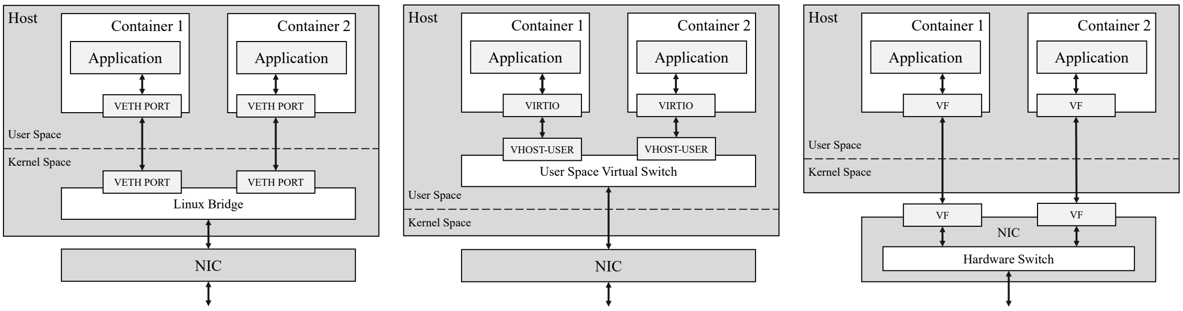
These overheads can be partially amortized resorting to batch APIs to exchange multiple packets using a single system call (either `sendmmsg()` or `recvmmsg()`), grouping packets in *bursts*, amortizing the cost of each system call over the packets in each burst. However, packets still need to traverse the whole in-kernel networking stack, going through additional data copies, to be sent or received.

This last problem can be tackled bypassing partially the kernel networking stack using raw sockets instead of regular UDP sockets and implementing data encapsulation in user-space. This is often done in combination with zero-copy APIs and memory-mapped I/O to transfer data quickly between the application and the virtual Ethernet port, greatly reducing the time needed to send a packet (Rizzo, 2012b).

Despite the various techniques available to reduce the impact of the kernel on network performance, their use is often not sufficient to match the demanding requirements of typical NFV applications (Ara et al., 2019). In typical scenarios, the only solution to these problems is to resort to techniques that can bypass completely the kernel when exchanging packets both locally and between multiple hosts.

2.2 Inter-container Communications with Kernel Bypass

Significant performance improvements for inter-container networking can be achieved by avoiding system calls, context switches and unneeded data copies as much as possible. Various I/O frameworks undertake such an approach, recurring to *kernel bypassing* techniques to exchange batches of raw Ethernet frames among applications without a single system call. For example, frameworks based on the *virtio* standard (Russell et al., 2015) use para-virtualized network interfaces that rely on shared memory to achieve high-performance communication among containers located on the same host, a situation which is fairly common in NFV scenarios. While *virtio* network devices are typically implemented for hypervisors (e.g. QEMU, KVM), recently a complete user-space implementation of the *virtio* specification called *vhost-user* can be used for OS containers to completely bypass the kernel. However, *virtio* cannot be used to access the physical network without any user-space implementation of a virtual switch. This means that it cannot be used alone to achieve both dynamic and flexible communications among independently deployed containers.



(a) Kernel-based solution. (b) Using DPDK with *vhost-user*. (c) Using SR-IOV support.

Figure 1: Different approaches to inter-container networking.

2.2.1 Data Plane Development Kit (DPDK)

Data Plane Development Kit (DPDK)[1] is an open source framework for fast packet processing implemented entirely in user-space and characterized by a high portability across multiple platforms. Initially developed by Intel, it now provides a high-level programming abstraction that applications can use to gain access to low-level resources in user-space without depending on specific hardware devices.

In addition, DPDK supports *virtio*-based networking via its implementation of *vhost-user* interfaces. Applications can hence exchange data efficiently using DPDK to communicate locally using *vhost-user* ports and remotely via DPDK user-space implementations of actual Ethernet device drivers: the framework will simply initialize the specified ports accordingly in complete transparency from the application point of view. For this reason, DPDK has become extremely popular over the past few years when it is necessary to implement fast data plane packet processing.

2.3 High-performance Switching among Containers

After having described *virtio* in Section 2.2, now we can explain better how *virtio*, in combination with other tools, can be used to realize a virtual networking infrastructure in user-space. There are essentially two ways to achieve this goal: 1) by assigning each container a *virtio* port, using *vhost-user* to bypass the kernel, and then connect each port to a virtual switch application running on the host (Figure 1b); 2) by leveraging special capabilities of certain NIC devices that allow concurrent access from multiple applications and that can be accessed in user-space by using DPDK drivers (Figure 1c). The virtual switch instance, either software or hardware, is then connected to the physical network via the actual NIC interface present on the host.

Many software implementations of L2/L3 switches are based on DPDK or other *kernel bypass* frameworks, each implementing their own packet processing logic responsible for packet forwarding. For this reason, performance among various implementation may differ greatly from one implementation to another. In addition, these switches consume a non-negligible amount of processing power on the host they are deployed onto, to achieve very high network performance. On the other hand, special NIC devices that support the Single-Root I/O Virtualization (SR-IOV) specification allow traffic offloading to the hardware switch implementation that they embed, which applications can access concurrently without interfering with each other.

Below, we briefly describe the most common software virtual switches in the NFV industrial practice, and the characteristics of SR-IOV compliant devices.

DPDK Basic Forwarding Sample Application[2] is a sample application provided by DPDK that can be used to connect DPDK-compatible ports, either virtual or physical, in pairs: this means that each application using a given port can only exchange packets with a corresponding port chosen during system initialization. For this reason, this software does not perform any packet processing operation, hence it cannot be used in real use-case scenarios.

Open vSwitch (OVS)[3] is an open source virtual switch for general-purpose usage with enhanced flexibility thanks to its compatibility with the *OpenFlow* protocol (Pfaff et al., 2015). Recently, Open vSwitch (OVS) has been updated to support DPDK and *virtio*-based ports, which accelerated considerably packet forwarding operations by performing them in user-space rather than within a kernel module (Intel, 2015).

[1] https://www.dpdk.org/

[2] https://doc.dpdk.org/guides/sample_app_ug/skeleton.html

[3] https://www.openvswitch.org

FD.io Vector Packet Processing (VPP)[4] is an extensible framework for virtual switching 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 uses Cisco Vector Packet Processing (VPP) that processes packets in batches, improving the performance thanks to the better exploitation of instruction and data cache locality (Barach et al., 2018).

Snabb[5] is a packet processing framework that can be used to provide networking functionality in user-space. It allows for programming arbitrary packet processing flows (Paolino et al., 2015) by connecting functional blocks in a Directed Acyclic Graph (DAG). While not being based on DPDK, it has its own implementation of *virtio* and some NIC drivers in user-space, which can be included in the DAG.

Single-Root I/O Virtualization (SR-IOV) (Dong et al., 2012) is a specification that allows a single NIC device to appear as multiple PCIe devices, called Virtual Functions (VFs), that can be independently assigned to VMs or containers and move data through dedicated buffers within the device. VMs and containers can directly access dedicated VFs and leverage the L2 hardware switch embedded in the NIC for either local or remote communications (Figure 1c). Using DPDK APIs, applications within containers can access the dedicated VFs bypassing the Linux kernel, removing the need of any software switch running on the host; however, a DPDK daemon is needed on the host to manage the VFs.

3 RELATED WORK

The proliferation of different technologies to exchange packets among containers has created the need for new tools to evaluate the performance of virtual switching solutions with respect to throughput, latency and scalability. Various works in the research literature addressed the problem of network performance optimization for VMs and containers.

A recent work compared various *kernel bypass* frameworks like DPDK and Remote Direct Memory Access (RDMA)[6] against traditional sockets, focusing on the round-trip latency measured between two directly connected hosts (Géhberger et al., 2018). This work showed that both DPDK and RDMA outperform POSIX UDP sockets, as only with *kernel bypass* mechanisms it is possible to achieve single-digit microsecond latency, with the only drawback that applications must continuously poll the physical devices for incoming packets, leading to high CPU utilization.

Another work (Lettieri et al., 2017) compared both qualitatively and quantitatively common high-performance networking setups. It measured CPU utilization and throughput achievable using either SR-IOV, Snabb, OVS and Netmap (Rizzo, 2012b), which is another networking framework for high-performance I/O. Their focus was on VM to VM communications, either on a single host or between multiple hosts, and they concluded that in their setups Netmap is capable of reaching up to 27 Mpps (when running on a 4 GHz CPU), outperforming SR-IOV due to the limited hardware switch bandwidth.

Another similar comparison of various *kernel bypass* technologies (Gallenmüller et al., 2015) evaluates throughput performance of three networking frameworks: DPDK, Netmap and PF_RING[7]. They concluded that for each of these solutions there are two major bottlenecks that can potentially limit networking performance: CPU capacity and NIC maximum transfer rate. Among them, the latter is the dominating bottleneck when per-packet processing cost is kept low, while the former has a bigger impact as soon as this cost increases and the CPU becomes fully loaded. Their evaluations also showed that DPDK is able to achieve the highest throughput in terms of packets per second, independently from the burst size; on the contrary, Netmap can reach its highest throughput only for a burst size greater than 128 packets per burst, and even then it cannot reach the same performance of DPDK or PF_RING.

The scalability of various virtual switching solutions with respect to the number of VMs deployed on the same host has been addressed in another recent work (Pitaev et al., 2018), in which VPP and OVS have been compared against SR-IOV. From their evaluations, they concluded that SR-IOV can sustain a greater number of VMs with respect to software-based solutions, since the total throughput of the system scales almost linearly with the number of VMs. On the contrary, both OVS and VPP are only able to scale the total throughput up to a certain plateau, which is influenced by the amount of allocated CPU resources: the more VMs are added, the more the system performance degrades if no more resources are allocated for the each software virtual switch.

Finally, a preliminary work (Ara et al., 2019) was presented by the authors of this paper, performing a basic comparison of various virtual switching techniques for inter-container communications. That evalu-

[4] https://fd.io/

[5] https://github.com/snabbco/snabb

[6] http://www.rdmaconsortium.org

[7] https://www.ntop.org/products/packet-capture/pf_ring/

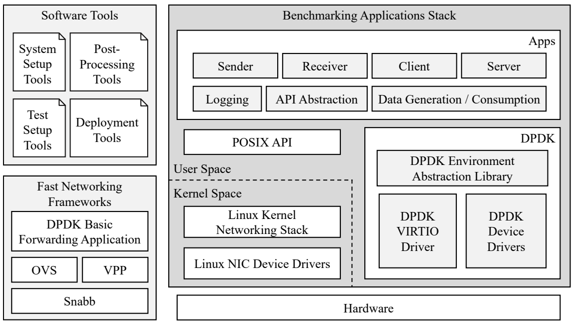


Figure 2: Main elements of the proposed framework.

ation however was limited to only a single transmission flow between a sender and a receiver application deployed on the same machine, for which SR-IOV was the most suitable among the tested solutions. This work extends that preliminary study to a much broader number of test cases and working conditions, on either one or multiple machines, and presents a new set of tools that can be used to conveniently repeat the experiments in other scenarios.

Compared to the above mentioned works, in this paper we present for the first time an open-source framework that can be used to evaluate the performance of various widely adopted switching solutions among Linux containers in the common NFV industrial practice. This framework can be used to carry out experiments from multiple points of view, depending on the investigation focus, while varying testing parameters and the system configuration. The proposed framework eases the task of evaluating the performance of a system under some desired working conditions. With this framework, we evaluated system performance in a variety of application scenarios, by deploying sender/receiver and client/server applications on one or multiple hosts. This way we were able to draw some general conclusions about the characteristics of multiple networking solutions when varying the evaluation point of view, either throughput, latency or scalability of the system. As being open-source, the framework can be conveniently extended by researchers or practicioners, should they need to write further customized testing applications.

4 PROPOSED FRAMEWORK

In this section we present the framework we realized to evaluate and compare the performance of different virtual networking solutions, focusing on Linux containers. The framework we developed can be easily installed and configured on one or multiple general-purpose servers to instantiate a number of OS containers and deploy in each of them a selected application that generates or consumes synthetic network traffic. These applications, also developed for this framework, serve the dual purpose to generate/consume network traffic and to collect statistics to evaluate system performance in the given configuration.

In particular, this framework can be used to configure a number of tests, each running for a specific amount of time with a given system configuration; after each test is done, the framework collects performance results from each running application and provides the desired statistics to the user. In addition, multiple tests can be performed consecutively by specifying multiple values for any test parameter.

The software is freely available on GitHub, under a GPLv3 license, at: https://github.com/gabrieleara/nfv-testperf . The architecture of the framework is depicted in Figure 2, which highlights the various tools it includes. First there are a number of software tools and Bash scripts that are used to install system dependencies, configure and customize installation, run performance tests and collect their results. Among the dependencies installed by the framework there are the DPDK framework (which includes also the Basic Forwarding Sample Application), and the other software-based virtual switches that will be used during testing to connect applications in LXC containers: OVS (compiled with DPDK support), VPP, and Snabb (configured as a simple learning switch). Each virtual switch is configured to act as a simple learning L2 switch, with the only exception represented by the DPDK Basic Forwarding Sample Application, which does not have this functionality. In addition, VPP and OVS can be connected to physical Ethernet ports to perform tests for inter-machine communications.

Finally, a set of benchmarking applications are included in the framework, serving the dual purpose to generate/consume network traffic and to collect statistics that can be used to evaluate the actual system performance in the provided configuration. These can measure the performance from various view points:

Throughput. As many VNFs deal with huge amounts of packets per second, it is important to evaluate the maximum performance of each networking solution with respect to either the number of packets or the amount of data per second that it is able to effectively process and deliver to destination. To do this, the sender/receiver application pair is provided to generate/consume unidirectional traffic (from the sender to the receiver) according to the given parameters.

Latency. Many VNFs operate with networking protocols designed for hardware implementations of certain network functions; for this reason, these protocols expect very low round-trip latency between interacting components, in the order of single-digit microsecond latency. In addition, in NFV infrastructures it is crucial to keep the latency of individual interactions as little as possible to reduce the end-to-end latency between components across long service chains. For this purpose, the client/server application pair is used to generate bidirectional traffic to evaluate the average round-trip latency for each packet when multiple packets are sent and received in bursts through a certain virtual switch. To do so, the server application will send back each packet it receives to its corresponding client.

Scalability. Evaluations from this point of view are orthogonal with respect of the two previous dimensions, in particular with respect to throughput: since full utilization of a system is achieved only when multiple VNFs are deployed on each host that is present within infrastructure, it is extremely important to evaluate how this affects performance in the mentioned dimensions. For this purpose there are no dedicated applications: multiple instances of each designated application can be deployed concurrently to evaluate how that affects global system performance.

The benchmarking applications are implemented in C and they are built over a custom API that masks the differences between POSIX and DPDK; this way, they can be used to evaluate system performance using *linux-bridge* or other virtual switches that bypass the kernel to exchange data. When POSIX APIs are used to exchange packets, raw sockets can also be used rather than of regular UDP sockets to bypass partially the Linux networking stack, building Ethernet, IP and UDP packet headers in user-space. The traffic generated/consumed by these applications can be configured varying a set of parameters that include the sending rate, packet size, burst size, etc. To maximize application performance, packets are always exchanged using polling and measurements of elapsed time are performed by checking the TSC register instead of less precise timers provided by Linux APIs.

During each test, each application is deployed within a LXC container on one or multiple hosts and it is automatically connected to the other designated application in the pair. The Linux distribution that is used to realize each container is based on a simple *rootfs* built from a basic *BusyBox* and it contains only the necessary resources to run the benchmarking applications. The framework then takes care of all the setup necessary to interconnect these applications with the desired networking technology. The latter may be any among *linux-bridge*, another software-based virtual switch (using *virtio* and *vhost-user* ports) or a SR-IOV Ethernet adapter; each scenario is depicted in Figure 1. In any case, deployed applications use polling to exchange network traffic over the selected ports. For tests involving multiple hosts, only OVS or VPP can be used among software-based virtual switches to interconnect the benchmarking applications; otherwise, it is possible to assign to each container a dedicated VF and leverage the embedded hardware switch in the SR-IOV network card to forward traffic from one host to another.

The proposed framework can be easily extended to include more virtual switching solutions among containerized applications or to develop different testing applications that generate/consume other types of synthetic workload. From this perspective, the inclusion of other *virtio*-based virtual switches is straightforward, and it does not require any modification of the existing test applications. In contrast, other networking frameworks that rely on custom port types/programming paradigms (e.g., Netmap) may require the extension of the API abstraction layer or the inclusion of other custom test applications. Further details about the framework's extensibility can be found at https://github.com/gabrieleara/nfv-testperf/wiki/Extending .

5 EXPERIMENTAL RESULTS

To test the functionality of the proposed framework, we performed a set of experiments in a synthetic use-case scenario, comparing various virtual switches.

Experiments were performed on two similar hosts: the first has been used for all local inter-container communications tests, while both hosts have been used 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, 64 GB of RAM, and an Intel® X710 DA2 Ethernet Controller for 10 GbE SFP+ (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.

Table 1: List of parameters used to run performance tests with the framework.

Parameter	Symbol	Values
Test Dimension	D	*throughput* or *latency*
Hosts Used	L	*local* or *remote* (i.e. single or multi-host)
Containers Set	S	*NvsN* (where N is the number of pairs)
Virtual Switch	V	*linux-bridge*, *basicfwd*, *ovs*,*snabb*, *sriov*, *vpp*
Packet Size	P	Expressed in bytes
Sending Rate	R	Expressed in pps
Burst Size	B	Expressed in number of packets per burst

5.1 Testing Parameters

The framework can be configured to vary certain parameters before running each test. Each configuration is represented by a set of applications (each running within a container), deployed either on one or both hosts and grouped in pairs (i.e. sender/receiver or client/server) and a set of parameters that specify the synthetic traffic to be generated. Given the number of evaluated test cases, for each different perspective we will show only relevant results.

To identify each test, we use the notation shown in Table 1: each test is uniquely identified using a tuple in following form:

$$(D,\ L,\ S,\ V,\ P,\ R,\ B)$$

When multiple tests are referred, the notation will omit those parameters that are free to vary within a predefined set of values. For example, to show some tests performed while varying the sending rate the R parameter will not be included in the tuple.

Each test uses only a fixed set of parameters and runs for 1 minute; once the test is finished, an average value of the desired metric (either throughput or latency) is obtained after discarding values related to initial warm-up and shutdown phases, thus considering only values related to steady state conditions of the system. The scenarios that we considered in our experiments are summarized in Figure 3.

5.2 Kernel-based Networking

First we will show that performance achieved without using *kernel bypass* techniques are much worse than the ones achieved using techniques like *vhost-user* in similar set-ups. Table 2 reports the maximum throughput achieved using socket APIs and *linux-bridge* to interconnect a pair of sender and receiver applications on a single host (Figure 3a). In this scenario, it is not possible to reach even 1 Mpps using kernel-based techniques, while using DPDK any virtual switch can easily support at least 2 Mpps in similar set-ups. That is why all the results that will follow will consider *kernel bypass* technologies only.

Table 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

5.3 Throughput Evaluations

Moving on to techniques that use DPDK, we first evaluated throughput performance between two applications in a single pair, deployed both on the same host or on multiple hosts, varying desired sending rate, packet and burst sizes. In all our experiments, we noticed that varying the burst size from 32 to 256 packets per burst did not affect throughput performance, thus we will always refer to the case of 32 packets per burst in further reasoning.

5.3.1 Single Host

We first deployed a single pair of sender/receiver applications on a single host (Figure 3a), varying the sending rate from 1 to 20 Mpps and the packet size from 64 to 1500 bytes:

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

In all our tests, each virtual switch achieves the desired throughput up to a certain maximum value, after which the virtual switch has saturated his capabilities, as shown in Figure 4a. This maximum throughput depends on the size of the packets exchanged through the virtual switch, thus from now on we will consider only the maximum achievable throughput for each virtual switch while varying the size of each packet.

Figures 4b and 4c show the maximum receiving rates achieved in all our tests that employed only two containers on a single host, expressed in Mpps and in Gbps respectively. In each plot, the achieved rate (Y axis) is expressed as function of the size of each packet (X axis), for which a logarithmic scale has been used.

From both figures it is clear that maximum performance are attained by offloading network traffic

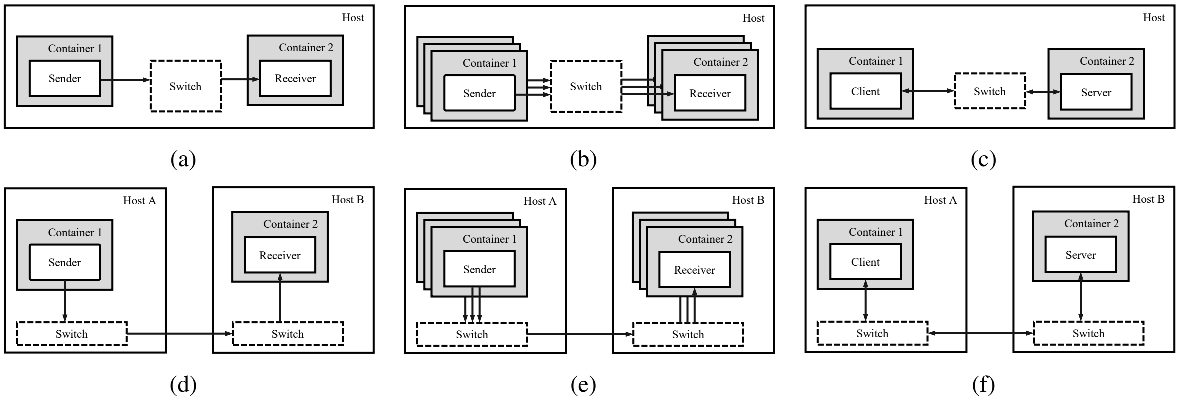


Figure 3: Different testing scenarios used for our evaluations. In particular, (a), (b), and (c) refer to single-host scenarios, while (d), (e), and (f) to scenarios that consider multiple hosts.

to the SR-IOV device, exploiting its embedded hardware switch. Second best ranked the Basic Forwarding Sample Application, which was expected since it does not implement any actual switching logic. The very small performance gap between VPP and the latter solution is also a clear indicator that the batch packet processing features included in VPP can distribute packet processing overhead effectively among incoming packets. Finally, OVS and Snabb follow. Comparing these performance with other evaluations present in literature (Ara et al., 2019), we were able to conclude that the major bottleneck for Snabb is represented by its internal L2 switching component.

In addition, notice that while the maximum throughput in terms of Mpps is achieved with the smallest of the packet sizes (64 bytes), the maximum throughput in terms of Gbps is actually achieved for the biggest packet size (1500 bytes). In fact, throughput in terms of Gbps is characterized by a logarithmic growth related to the increase of the packet size, as shown in Figure 4c.

From these plots we derived that for software switches the efficiency of each implementation impacts performance only for smaller packet sizes, while for bigger packets the major limitation becomes the ability of the software to actually move packets from one CPU core to another, which is equivalent for any software implementation. Given also the slightly superior performance achieved by SR-IOV, especially for smaller packet sizes, we also concluded that its hardware switch is more efficient at moving large number of packets between CPU cores than the software implementations that we tested.

5.3.2 Multiple Hosts

We repeated these evaluations deploying the receiver application on a separate host (Figure 3d), using the only virtual switches able to forward traffic between multiple hosts[8]:

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

Figure 4d shows the maximum receiving rates achieved for a burst size of 32 packets. In this scenario, results depend on the size of the exchanged packets: for smaller packet sizes the dominating bottleneck is still represented by the CPU for OVS and VPP, while for bigger packets the Ethernet standard limits the total throughput achievable by any virtual switch to only 10 Gbps. From these results we concluded that when the expected traffic is characterized by relatively small packet sizes (up to 256 bytes) deploying a component on a directly connected host does not impact negatively system performance when OVS or VPP are used. In addition, we noticed that in this scenario there is no clear best virtual switch with respect to the others: while for smaller packet sizes SR-IOV is more efficient, both OVS and VPP perform better for bigger ones.

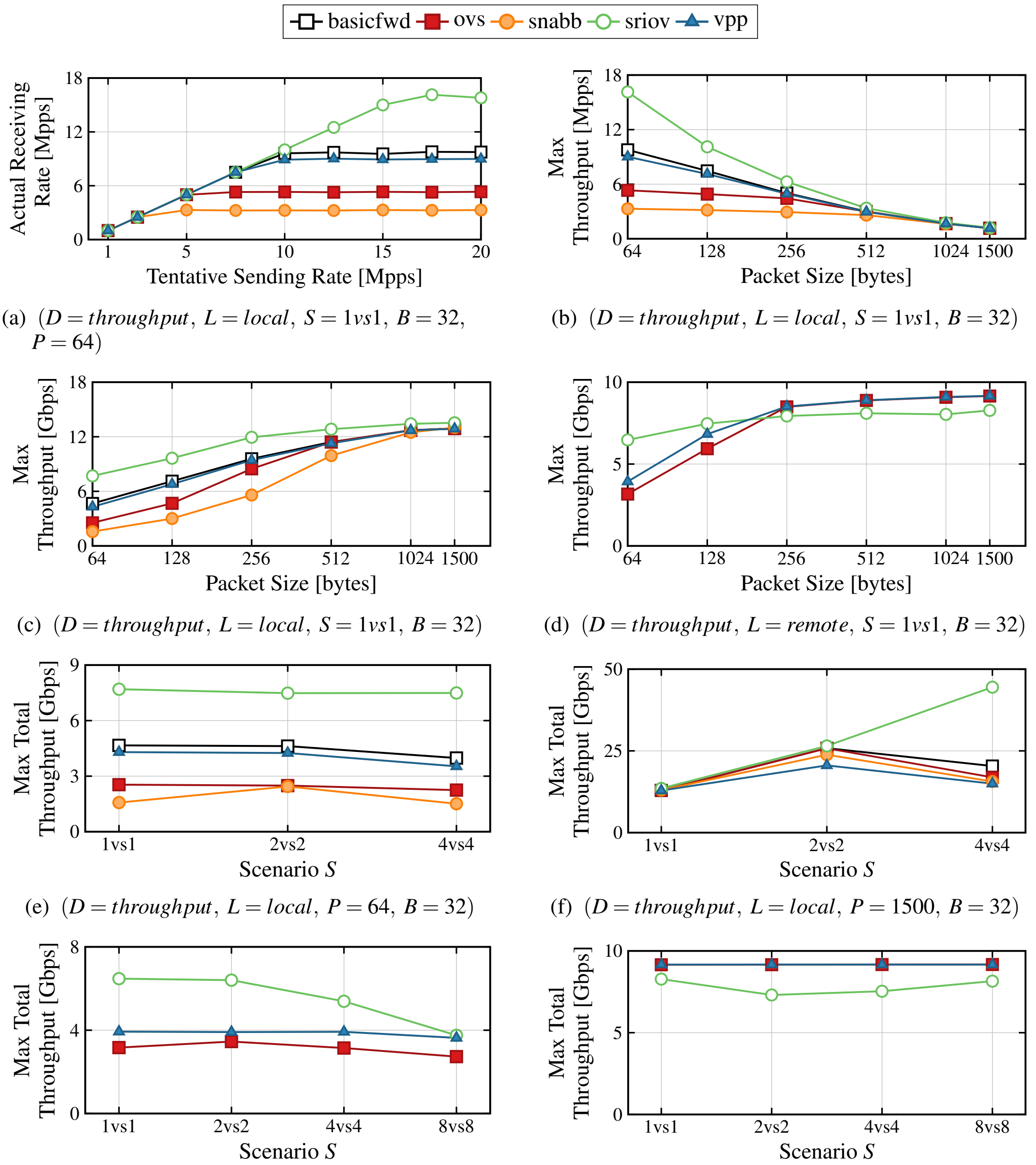
5.4 Throughput Scalability

To test how throughput performance scale when the number of applications on the system increases, we repeated all our evaluations deploying multiple application pairs on the same host (Figure 3b):

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

Figures 4e and 4f show the relationship between the maximum total throughput of the system achievable and the number of pairs deployed simultaneously, for packets of 64 and 1500 bytes respectively.

[8] The Basic Forwarding Sample Application does not implement any switching logic, while Snabb was not compatible with our selected SR-IOV Ethernet controller.



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

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

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

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

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

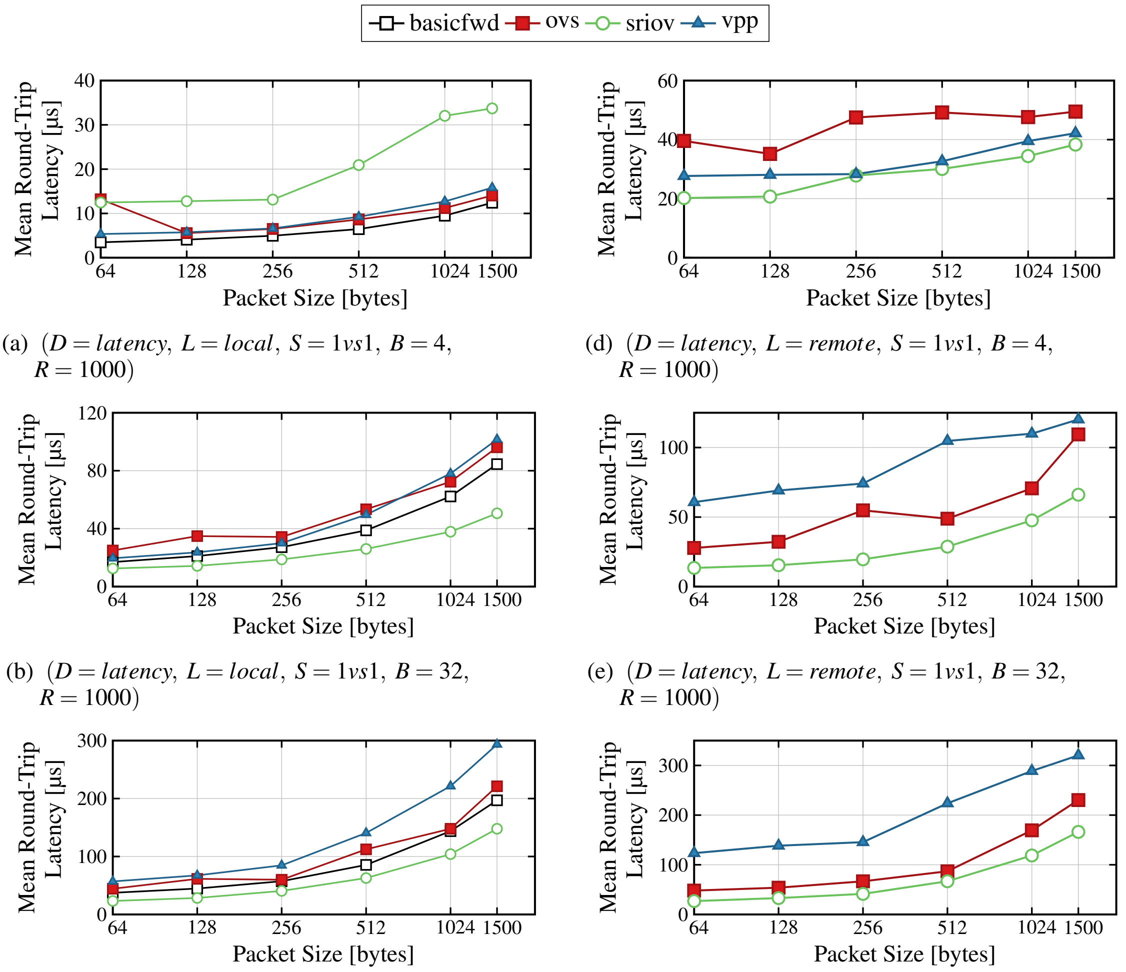
(f) $(D = throughput,\ L = local,\ P = 1500,\ B = 32)$

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

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

Figure 4: Throughput performance obtained varying system configuration and virtual switch used to connect sender/receiver applications deployed in LXC containers.

While the maximum total throughput does not change for relatively small packet sizes when increasing the number of senders, for bigger packet sizes SR-IOV can sustain 4 senders with only a per-pair performance drop of about 18%, achieving almost linear performance improvements with the increase of the number of participants. On the contrary, *virtio*-based switches can only distribute the same amount of resources over a bigger number of network flows. From these results we concluded that while for SR-IOV the major limitation is mostly represented by the number of packets exchanged through the network, for *virtio*-based switches the major bottleneck is represented by the capability of the CPU to move data from one application to another, which depends on the overall amount of bytes exchanged.



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

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

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

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

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

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

Figure 5: Average round-trip latency obtained varying system configuration and virtual switch used to connect client/server applications deployed in LXC containers. Plots on the left refer to tests performed on a single-host, while plots on the right involve two separate hosts.

Repeating scalability evaluations on multiple hosts ($L = remote$), we were able to deploy up to 8 application pairs ($S = 8vs8$) transmitting data from one host to the other one (Figure 3e). Results of these evaluations, shown in Figures 4g and 4h, indicate that the outcome strongly depends on the size of the packets exchanged: for bigger packets the major bottleneck is represented by the limited throughput of the Ethernet standard; for smaller ones the inability of each virtual switch to scale adequately with the number of packet flows negatively impacts even more system performance.

5.5 Latency Performance

We finally evaluated the round-trip latency between two applications when adopting each virtual switch to estimate the per-packet processing overhead introduced by each different solution. In particular we focused on the ability of each solution to distribute packet processing costs over multiple packets when 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, so that there is no interference between the processing of a burst of packets and the following one.

First we considered a single pair of client/server applications deployed on a single host (Figure 3c). In these tests we varied the burst size from 4 to 128 packets and the packet size from 64 to 1500 bytes:

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

In all our tests, Snabb was not able to achieve comparable latency with respect to other solutions; for example, while all other virtual switches could maintain their average latency under 40 µs for $B = 4$, Snabb's delay was always greater than 60 µs, even for very small packet sizes. Since this overall behavior is repeated in all our tests, regardless of which parameters are applied, Snabb will not be discussed further.

Figures 5a to 5c show that only *virtio*-based solutions were able to achieve single-digit microsecond round-trip latency on a single host, while SR-IOV has a higher latency for small packet and burst sizes. Increasing the burst size over 32 packets, these roles are reversed, with SR-IOV becoming the most lightweight solution, although it was never able to achieve single-digit microsecond latency. From this we inferred that SR-IOV performance are less influenced by the variation of the burst size with respect to the other options available and thus it is more suitable when traffic on a single host is grouped into bigger bursts.

We then repeated the same evaluations by deploying the server application on a separate host ($L = remote$, Figure 3f). In this new scenario, SR-IOV always outperforms OVS and VPP, as shown in Figures 5d to 5f. This can be easily explained, since when using a *virtio*-based switch to access the physical network two levels of forwarding are introduced with respect to the case using only SR-IOV: the two software instances, each running in their respective hosts, introduce additional computations with respect to the one performed in hardware by the SR-IOV device.

6 CONCLUSIONS AND FUTURE WORK

In this work, we presented a novel framework aimed to evaluate and compare the performance of various virtual networking solutions based on kernel bypass. We focused on the interconnection of VNFs deployed in Linux containers in one or multiple hosts.

We performed a number of performance evaluations on some virtual switches commonly used in the NFV industrial practice. Test results show that SR-IOV has superior performance, both in terms of throughput and scalability, when traffic is limited to a single host. However, in scenarios that consider inter-host communications, each solution is mostly constrained by the limits imposed by the Ethernet standard. Finally, from a latency perspective we showed that both for local and remote communications SR-IOV can attain smaller round-trip latency with respect to its competitors when bigger burst sizes are adopted.

In the future, we plan to support additional virtual networking solutions, like NetVM (Hwang et al., 2015), Netmap (Rizzo, 2012a), and VALE (Rizzo and Lettieri, 2012). We also plan to extend the functionality of the proposed framework, for example by allowing for customizing the number of CPU cores allocated to the virtual switch during the runs. Finally, we would like to compare various versions of the considered virtual switching solutions, as we noticed some variability in the performance figures after upgrading some of the components, during the development of the framework.

REFERENCES

Ara, G., Abeni, L., Cucinotta, T., and Vitucci, C. (2019). On the use of kernel bypass mechanisms for high-performance inter-container communications. In *High Performance Computing*, pages 1–12. Springer International Publishing.

Barach, D., Linguaglossa, L., Marion, D., Pfister, P., Pontarelli, S., and Rossi, D. (2018). High-speed software data plane via vectorized packet processing. *IEEE Communications Magazine*, 56(12):97–103.

Barik, R. K., Lenka, R. K., Rao, K. R., and Ghose, D. (2016). Performance analysis of virtual machines and containers in cloud computing. In *2016 International Conference on Computing, Communication and Automation (ICCCA)*. IEEE.

Dong, Y., Yang, X., Li, J., Liao, G., Tian, K., and Guan, H. (2012). High performance network virtualization with SR-IOV. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480.

Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE.

Gallenmüller, S., Emmerich, P., Wohlfart, F., Raumer, D., and Carle, G. (2015). Comparison of frameworks for high-performance packet IO. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE.

Géhberger, D., Balla, D., Maliosz, M., and Simon, C. (2018). Performance evaluation of low latency communication alternatives in a containerized cloud environment. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE.

Hwang, J., Ramakrishnan, K. K., and Wood, T. (2015). NetVM: High performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management*, 12(1):34–47.

Intel (2015). Open vSwitch enables SDN and NFV transformation. White Paper, Intel.

Lettieri, G., Maffione, V., and Rizzo, L. (2017). A survey of fast packet I/O technologies for Network Function Virtualization. In *Lecture Notes in Computer Science*, pages 579–590. Springer International Publishing.

Paolino, M., Nikolaev, N., Fanguede, J., and Raho, D. (2015). 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.

Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., et al. (2015). The design and implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130.

Pitaev, N., Falkner, M., Leivadeas, A., and Lambadaris, I. (2018). 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.

Rizzo, L. (2012a). Netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA. USENIX Association.

Rizzo, L. (2012b). Revisiting network I/O APIs: The Netmap framework. *Queue*, 10(1):30.

Rizzo, L. and Lettieri, G. (2012). VALE, a switched ethernet for virtual machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies - CoNEXT '12*. ACM Press.

Russell, R., Tsirkin, M. S., Huck, C., and Moll, P. (2015). Virtual I/O Device (VIRTIO) Version 1.0. Standard, OASIS Specification Committee.