Deliverable 5.6: Universal Node Benchmarking

Dissemination level	PU
Version	0.9
Due date	31.05.2016
Version date	29.07.2016

This project is co-funded
by the European Union

Document information

Editor

Hagen Woesner (BISDN)

Contributors

Ivano Cerrato (POLITO), Fulvio Risso (POLITO), Mauricio Vasquez Bernal (POLITO), David Verbeiren (INTEL), Gergely Pongrácz (ETH), Vinicio Vercellone (TI), Jon Matias (EHU), Willy Failla (TP), Tobias Hintze (TP), Kostas Pentikousis (TP), Michael Schlosser (BISDN)

Reviewers

t.b.d.

Coordinator

Dr. András Császár
Ericsson Magyarország Kommunikációs Rendszerek Kft. (ETH)
Könyves Kálmán körút 11/B épület
1097 Budapest, Hungary
Fax: +36 (1) 437-7467
Email: andras.csaszar@ericsson.com

Project funding

7th Framework Programme
FP7-ICT-2013-11
Collaborative project
Grant Agreement No. 619609

Executive Summary

A focal point of the research and development work in UNIFY is the implementation of a proof-of-concept high-performance data plane implementation for the Universal Node (UN). In Network Function Virtualization (NFV) terms, the UN is a hyper-converged Virtualized Network Function (VNF) infrastructure that exposes a big-switch/big-software abstraction to an overarching orchestrator.

This deliverable presents the empirical performance evaluation results of our research and development effort on the select, individual UN components that have been implemented. Note that this deliverable focuses on individual UN components while the details of the UN architecture and the evaluation of entire use case pipelines are presented in the subsequent Deliverable D5.7.

Overall this deliverable covers three distinct aspects. First, we review the architectural options for implementing UN. We present the different types of VNFs that were considered and developed during the project and refer back to the original project work plan and review the implementation options. It is worth noting that the project has advanced the notion of domain-oriented orchestration and therefore WP5 ensured that all implementation options can be supported by the respective UN orchestrator. Moreover, we present three options for the underlying softswitch along with a quick overview of the host/VNF interfaces. A brief description of platforms where the UN has been ported to is also provided.

The second aspect covered in this deliverable is the benchmarking methodology adopted. As we explain, the sheer size of the design space with multiple possible combinations of hardware and software, as well the target set for being able to deploy the Universal Node in a wide range of network segments ranging from customer premises to aggregation and core networks to carrier data centers does not lend itself to a simple, straightforward methodology that can be applied across the board for the UN. Note that this is also the case with today's equipment: for instance, Customer Premises Equipment (CPE) and core network elements are evaluated following different benchmarking methodologies. Therefore, we too use multiple test setups to evaluate the performance of the UN components. Given also the ongoing research efforts, quite naturally, there cannot be the same test setup at all partners, due to restrictions in the availability of hardware as well as different measurement objectives, as explained above.

This deliverable concludes with the presentation, for the first time, of a number of empirical performance evaluation results of all the options and UN components considered. Despite the wide range of options, we are able to reach certain conclusions which hold across the individual platforms. Essentially, the results obtained paint a picture of an architecture that is implementable on various platforms, exploits local hardware acceleration options where possible, is adaptable to the available hypervisors, and still is sufficiently small to work on devices that have limited resources, such as a CPE.

Contents

1 Introduction **1**

1.1 VNF Types . . . 1

1.1.1 QEMU/KVM Virtual Machines . . . 1

1.1.2 Lightweight Containers . . . 2

1.1.3 DPDK Processes . . . 3

1.1.4 Native Functions . . . 4

1.2 Soft Switches: ERFS, OVS, and xDPd . . . 4

1.2.1 Ericsson Research Flow Switch (ERFS) . . . 4

1.2.2 eXtensible DataPath Daemon (xDPd) . . . 5

1.3 Host/VNF Interfaces . . . 6

1.3.1 virtio + vhost . . . 6

1.3.2 Dpdkr ports (IVSHMEM) . . . 8

1.3.3 Direct VM2VM . . . 9

1.3.4 PCI-PT . . . 11

1.3.5 SR-IOV . . . 11

1.3.6 Conclusion . . . 12

1.4 Compute Platforms . . . 12

1.4.1 Intel x86 . . . 12

1.4.2 Power PC . . . 12

1.4.3 ARM . . . 13

1.4.4 MIPS . . . 13

2 Methodology **13**

2.1 Introduction . . . 13

2.2 Telecom Italia Test Suite . . . 14

2.3 DPDK pktgen-based VNF Benchmarking Toolchain . . . 14

2.4 Inter-VNF Communication Throughput . . . 15

2.4.1 KVM and DPDK . . . 15

2.4.2 Docker vs. KVM . . . 17

2.5 Inter-VNF Communication Latency . . . 17

2.6 Hardware-accelerated I/O Throughput . . . 18

2.6.1 Improvements on PktGen for Throughput Calculation . . . 19

2.6.2 Setup for PCI-PT and vhost-user connected DPDK BNG . . . 19

2.7 Banana-Pi (ARM) Test Description . . . 20

3 Results **20**

3.1 Type 1 to Type 5 VNFs . . . 20

3.1.1 QEMU/KVM virtual machine vs. containers (type 1-2) vs. bare metal DPDK . . . 20

3.1.2 Type 4: Virtual Switch Plugin . . . 21

3.1.3 Type 5: Native Function . . . 22

3.2 Host/VNF Interfaces . . . 23
3.3 Hardware-accelerated I/O: PCI-PT, SR-IOV . . . 24
3.3.1 Intel DPDK BNG within guest machine vs. bare metal . . . 25
3.4 Inter-VNF Communication Throughput . . . 27
3.5 Inter-VNF communication Latency . . . 28
3.5.1 Measurement Results . . . 29

4 Conclusion 30

References 33

A Annex A - Conformance Tests 34
A.1 Test 1 - RMI: Node information and capabilities . . . 34
A.2 Test 2 - RMI: Node available resources . . . 35
A.3 Test 3 - NMI: NF-FG deployment . . . 35
A.4 Test 4 - NMI: NF-FG modification . . . 35
A.5 Test 5 - NMI: NF-FG deletion . . . 36
A.6 Test 6 - NMI: NF-FGs list . . . 36
A.7 Test 7 - NMI: NF-FG data . . . 36
A.8 Test 8 - VSIRI: VNF specifications by type . . . 37
A.9 Test 9 - VSIRI: VNF specifications by ID . . . 37
A.10 Test 10 - VSIRI: VNF images . . . 37

B Annex B - Functional Tests 37
B.1 Test 11 - URM: Discover and report available resources . . . 38
B.2 Test 12 - URM: Update available resources . . . 38
B.3 Test 13 - URM: Local scaling . . . 39
B.4 Test 14 - URM: NF-FG lifecycle management . . . 39
B.5 Test 15 - URM/LO: Optimized placement . . . 39
B.6 Test 16 - URM: Support to multiple VEE solutions . . . 40
B.7 Test 17 - URM: Support to multiple VSE solutions . . . 40
B.8 Test 18 - URM: Configuration of external connection to other nodes . . . 40
B.9 Test 19 - URM: Monitoring support and data report . . . 41
B.10 Test 20 - VSE: Dynamic deployment of LSI . . . 41
B.11 Test 21 - VSE: External traffic steering . . . 41

C Annex C - Performance Tests 42
C.1 Test 22 - Throughput . . . 42
C.2 Test 23 - CPU . . . 43
C.3 Test 24 - Memory . . . 44
C.4 Test 25 - Switching . . . 44
C.5 Test 26 - Control Interfaces . . . 45

List of Figures

1 Containers vs classic virtualization . . . 3
2 OVS internal modules . . . 5
3 Virtio architecture . . . 6
4 Virtio frontend drivers: a) standard Drivers, b) DPDK virtio-PMD. . . . 7
5 Virtio backend drivers: a) inside QEMU, b) vhost, c) vhost-user . . . 8
6 Sharing structures between OvS and DPDK applications. . . . 9
7 `dpdkr` port . . . 9
8 Example of service with a direct connection between two VNFs . . . 10
9 Direct connection between VMs through a new implementation of the `dpdkr` port . . . 10
10 PCI-PT(a) and SR-IOV(b) technology . . . 11
11 The architecture of the used measurement tool . . . 15
12 Principle setup of the Inter-VNF tests . . . 15
13 (a) Virtio + vhost and standard Open vSwitch. (b) Virtio + vhost-user and Open vSwitch-DPDK enabled. . . . 16
14 (a) Ivshmem. (b) Direct VM2VM. . . . 17
15 (a) netperf in Docker. (n) netperf in KVM . . . 17
16 (a)Latency Testing Architecture. (b) Base Latency Measurement Architecture . . . 18
17 Setup for testing different I/O variants with the Intel DPDK BNG prototype. . . . 19
18 CarOS UN test configuration. . . . 20
19 Performance of bare metal vs. containers vs classic virtualization . . . 21
20 Performance of simple port forwarder in KVM using ivshmem . . . 21
21 Performance of L2 type 4 VNF (bare metal) . . . 22
22 The setup of the L3 pipeline . . . 23
23 the setup of the datacenter gateway pipeline . . . 24
24 Throughput of the L2FWD/DPDK VM process using SR-IOV technology (Gbps) . . . 26
25 Throughput of the L2FWD/DPDK VM process using SR-IOV technology (Gbps) . . . 26
26 Power consumption of the Intel DPDK BNG in bare metal implementation (left) and PCI-PT (right) for total offered load up to 20 Gbit/s. . . . 27
27 Inter-VNF communication throughput . . . 28
28 Docker vs KVM throughput results . . . 28
29 InterVNF latency using 50% load . . . 29
30 InterVNF latency using different data rates . . . 30
31 CarOS UN delay measurements for various packet sizes and number of nodes in a chain . . . 31
32 CarOS UN throughput measurements for a single device; various packet sizes. Left axis indicates the scale for the bar plot (packets/s), right axis indicates the scale for the line plot (b/s) . . . 31
33 CarOS UN throughput measurements for a chain of two devices; various packet sizes. Left axis indicates the scale for the bar plot (packets/s), right axis indicates the scale for the line plot (b/s) . . . 31

34 CarOS UN throughput measurements for a chain of three devices; various packet sizes. Left axis indicates the scale for the bar plot (packets/s), right axis indicates the scale for the line plot (b/s) 31
35 A chain topology with three CarOS UNs used for the measurement tests presented in Figure 34. 32

List of Tables

1 VNF Types 1
2 Combinations of frontend and backend drivers. 8
3 Inter-VNF and VNF-host communication technologies 12
4 TI test framework test list. For detailed description, of each test, check Annex A to C. . . 14
5 Results of the router pipeline (kpps per core, 1-100-10k-500k IP flows, Xeon) 22
6 Results of the router pipeline (kpps per core, 1-100-10k-500k IP flows, Xeon) 23
7 Results of the DC GW pipeline 24
8 Results of the DC GW pipeline 25
9 Throughput of the SR-IOV with L2FWD/DPDK benchmark 25
10 Packet Loss of the SR-IOV with L2FWD/DPDK with 64B (100%) 27
11 Throughput and power consumption for different I/O techniques and the Intel DPDK prototype. 27

Introduction

One of our empirical performance evaluation objectives is to relate the use of different types of resources (mainly compute and networking) in the Universal Node (UN) and determine if it matches the expected performance. This benchmark is also relevant for both external embedding algorithms and internal optimal placement of Virtualized Network Functions (VNFs), since the obtained results can be used as a reference to select between the available alternatives in order to meet the requested Service Level Agreements (SLAs) and Key Performance Indicator (KPI) parameters. As a result, the benchmarking results are critical for optimal orchestration of resources to obtain the best possible performance while optimally configuring the resources and minimizing the involved resources (if applicable).
From the Universal Node (UN) perspective, our empirical performance evaluation focuses on three key aspects that affect the final performance of a VNF deployed on the Universal Node (UN): i) the compute resource that executes the VNF, ii) the underlying Logical Switch Instance (LSI), or LSI-0 (currently only softswitches are considered), and iii) the interfaces. The benchmarking methodology defines how this activity must be performed to extract, when possible (due to interrelation between them) the individual impact of each of these pieces on the overall performance of the Universal Node (UN).
In order to compare the results from different platforms/architectures and technologies, the results reported in this deliverable reflect performance in terms of Million pakcets per second (Mpps)/core or even Mpps/s/W when energy is considered as an optimization parameter to be considered either by the external orchestrator or the local orchestrator in the Universal Node (UN).

VNF Types

Deliverable D5.2 [D5.2] introduced five types of VNFs, named from `Type 1` to `Type 5`. During the course of the project we recognized that this terminology may be confusing to some readers. Hence, we replaced these terms with more descriptive names. For consistency, we list in Table 1 the mapping between the terms used in D5.2 and those used in this deliverable.

Table 1: VNF Types

Old name	New name
Type 1	QEMU/KVM virtual machine
Type 2	Lightweight container
Type 3	DPDK process
Type 4	Virtual switch plugin
Type 5	Native function

Each VNF type is described in the remainder of this section, with the exception of virtual switch plugins which are not supported by the Universal Node (UN).

QEMU/KVM Virtual Machines

VNFs can be executed as processes running in a Virtual Machine (VM). Practically, each VM is dedicated to a single process in order to guarantee isolation between different VNFs. The Universal Node platform can execute VMs in the QEMU/KVM virtualization environment [15e], an open source virtualization solution that supports both machine emulation and hypervisor operation. QEMU is implemented by KVM, a Linux

kernel module developed as part of the QEMU project. When guest and host refer to different processor architectures, dynamic binary translation is used. When they refer to the same CPU architecture, the executable code of the guest can be directly executed on the host processor with the hypervisor only intervening when needed for emulating the rest of the guest platform.
KVM takes advantage of hardware-assisted virtualization features present in modern x86 processors to further reduce the overhead incurred by running applications in such VM thereby providing close to native performance in many cases. These features attempt to reduce the frequency or the cost of processor control transitions from the application running in the guest to the hypervisor ("`VM Exits`") that can happen for various reasons. Example of such features (in the case of the Intel Xeon processors) include:

- *specific VM control structures (VMCS)* with storage for processor state for VM decrease the latency of transitions;
- *Extended Page Tables (EPT)* which remove the need for the hypervisor to maintain shadow page tables and avoid hypervisor intervention when the guest OS modifies its page tables;
- *tagged Translation Lookaside Buffer (TLB)* entries that avoid having to flush the whole TLB at VM transitions.

QEMU is a *full* machine virtualization solution that allows running unmodified operating systems, drivers and applications in the guest systems. This is in contrast with solutions referred to as *para-virtualization* (e.g., Xen [15i]), where the guest operating system must be specifically adapted in some areas to interface to the hypervisor instead of the real hardware. Para-virtualization removes the need for emulating hardware and can therefore provide a significant performance advantage when accessing devices such as network adapters.
Various evolutions have blurred the lines between these different approaches:

- QEMU/KVM now also uses para-virtualization when available, more recently in the form of the Virtio (Section 1.3.1) infrastructure in the Linux kernel;
- the presence of a input–output memory management unit (IOMMU) in modern hardware platforms allows giving direct access to virtual machines to designated devices, or portions of devices (e.g., PCI passthrough - Section 1.3.4);
- some devices include dedicated support for being efficiently shared among multiple virtual machines; this is the case of the PCI-SIG Single Root I/O Virtualization (SR-IOV) (Section 1.3.5) technology.

Lightweight Containers

Lightweight containers can also be exploited to execute VNFs. For example, Linux containers (LXC) are a lightweight virtualization mechanism that, unlike VMs, does not run a complete operating system: all containers share the host kernel, i.e. on the physical machine there is a single kernel, with a single scheduler, a single memory manager and so on. Containers do not emulate hardware. In practice, LXC limits the resources (e.g., CPU, memory) visible to a running userland process. Such a limitation of resources is achieved through the `cgroups` feature of the Linux kernel, while isolation is provided through Linux namespaces, which give to the process running in the container a limited view of the process trees, networking, file system etc.

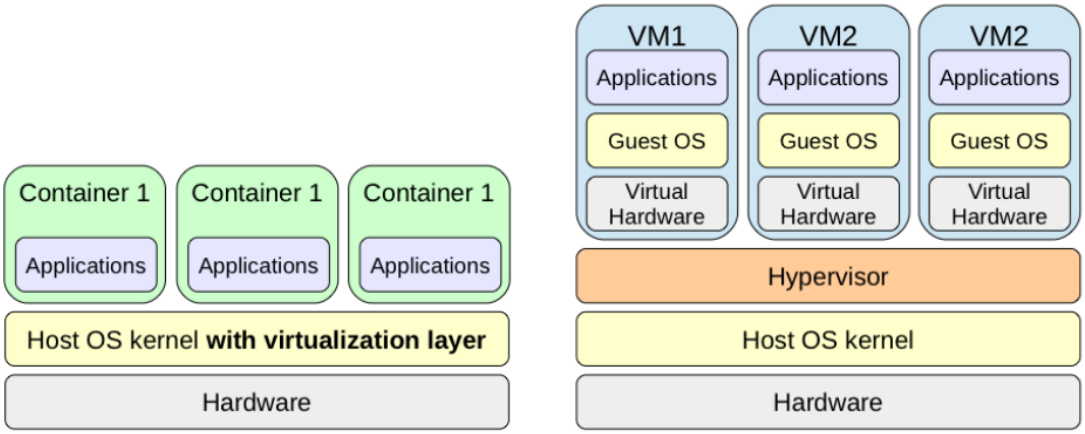

Figure 1: Containers vs classic virtualization

The main differences between classic virtualization and containers are depicted in Figure 1, which shows that there is no hypervisor when using LXC: the hypervisor is the host kernel itself, with its integrated virtualization features.
Docker containers enable the creation of lightweight, portable, self-sufficient containers from any application. A Docker container comprises an application and its dependencies (for example binaries and libraries) and can be used to isolate processes running in user space. Using Docker, containers file systems are created using copy-on-write, which makes deployment extremely fast, memory-cheap and disk-cheap. Changes to a container's file system can be committed into a new image and reused to create more containers. No templating or manual configuration is required. With traditional VMs, each application, each copy of an application, and each slight modification of an application requires creating an entirely new VM. In contrast, with Docker, running several copies of the same application on a host does not even require copying shared binaries and, if the application is modified, only the differences need to be copied, making it very efficient to store and run containers and updating applications easier.
Docker adds to plain LXC the ability to build once a container with an application inside and to run and migrate it across different Linux systems (with the same architecture), without the need to worry about configuration and dependencies typical of each platform. Docker offers resource management and isolation the same way as LXC and eases applications versioning management.

DPDK Processes

VNFs can be implemented as processes based on the Intel Data Plane Development Kit (DPDK) [15a], a framework optimized for the development of data plane applications targeting high-speed packet processing. Each VNF implemented as a DPDK process is runs on the host operating system alongside other components of the Universal Node (e.g., the virtual switch). The only difference between this type of VNF and those executed in lightweight containers is that, in the former case, no virtualization is used and no additional process isolation is provided on top of what the host operating system provides.

Native Functions

Some VNFs can be implemented as LSIs since they rely on normal operations of the switching engine. An example of such a VNF is an encapsulation/decapsulation function provided by the virtual switch and invoked as an explicit element of the service being implemented.

Soft Switches: ERFS, OVS, and xDPd

It is always possible to implement a VNF as a separate entity and run it in a virtual machine or container. This is the basic idea of NFV. The UN concept is a bit different however from NFV because of the concept of "switch internal" network functions. This basically means that if some function (e.g. firewall, routing) can be done in the switch, the upper layer orchestrators should use this possibility, optimizing the entire pipeline. So in Unify the vSwitches play a much more crucial role than in the traditional NFV architecture. Because of this in the project we investigated 3 virtual switches: ERFS, OVS, and xDPd. While the latter two have been discussed in previous deliverables, the Ericsson Research Flow Switch (ERFS) is a new development based on lessons learned within WP5, optimizing the pipeline even further and consequently gaining higher performance.

Ericsson Research Flow Switch (ERFS)

ERFS is an optimized OpenFlow pipeline built directly for supporting the OF 1.3 specification without any legacy for below 1.3. High performance was the main focus, which means that two main areas were selected for optimization:

- *lookup / classification*: in ERFS the lookup algorithm is selected optimally on a per table basis. This means that based on the incoming rules to a given
- *tunneling*: tunnels are supported inside the pipeline which is more efficient, especially with the JIT code generator
- *JIT*: ERFS uses just-in-time linked code for executing parsing, (some) lookup algorithms and the actions that are to be executed in the given table or group

Functionality-wise ERFS supports the mandatory OF 1.3 structures and commands, some optional and also some experimental features and extensions. For an up-to-date feature list please refer to the README file. Some features can be seen below:

- stateful and stateless load balancing with select groups, indirect groups
- rate limiting with meters (1 band, drop-only)
- matches and set-field on all fields, including metadata and tunnel-id
- built-in tunneling support: QinQ, GRE, VxLAN, MPLS, GTP (push/pop)
- NFV support with high-performance, dynamically created virtual ports: ivshmem, vhost, kni
- support for logical switch instances (LSIs) and zero-overhead inter-switch connections (each LSI has its own OpenFlow port allowing multiple controllers)

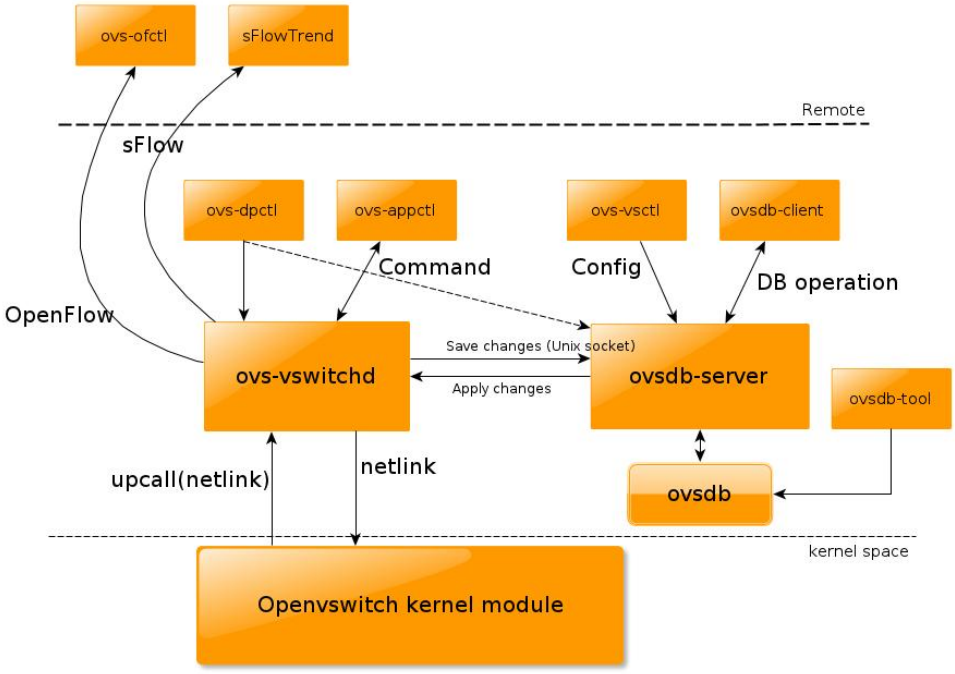

Figure 2: OVS internal modules

- defragmentation handling with special virtual port
- 3rd party plugin (shared library) support for experimental actions / instructions / messeges

subsubsectionOpen vSwitch (OVS) Open vSwitch (http://openvswitch.org/) is the de-facto standard virtual switch in Linux environment. It is a multilayer software switch, meaning it supports virtual bridges and dynamic connections. It is also multi-platform, supports standard management interfaces and opens the forwarding functions to programmatic extension and control. Its main function is virtual switching in VM environments, e.g. connectivity between the external world and the VMs or containers.
OVS has a Linux-kernel based flavour and a DPDK based flavour as well, with extensive work on supporting more backends, like Open Data Plane (ODP) and this way several ARM platforms. The internal architecture can be seen on the next figure.
Open vSwitch supports multiple Linux-based virtualization technologies including Xen/XenServer, KVM, and VirtualBox and multiple interfaces, like ivshmem, vhost-user, virteth and kni.
In contrast to its huge amount of extra features it lacks a few basic ones, like:

- *rate limiting*: generic OpenFlow meters are supported only by the control plane, but not implemented by the pipeline
- *tunneling*: instead of supporting tunneling inside the pipeline (e.g., push/pop VxLAN header), OVS supports tunneling via virtual ports, which has a certain performance penalty

eXtensible DataPath Daemon (xDPd)

xDPd [16b] is a framework for the development of multi-platform OpenFlow switches. It was specifically designed for portability across different compute and network processor platforms, partially sacrifying perfor-

mance for portability. A pipeline written in ANSI-C is the core packet handling element, while the OpenFlow endpoint(s) and management interfaces are built using the Revised OpenFlow Library (rofl), a set of C++ libraries that was developed to support in writing OpenFlow datapaths and controllers. xDPd has undergone some work to improve stability, and a trie-based matching algorithm has been implemented, but not yet merged to the main repository at the time of writing. This is apparent in some of the measurement results, especially for IPv4 traffic.

Host/VNF Interfaces

With the recent NFV paradigm, the communication between VNFs (usually executed inside virtual machines or lightweight containers) and the external world, and among different VNFs, has become a critical issue. This is due to several factors, such as the huge amount of packets that a VNF usually has to process, and the fact that some of the existing technologies used to implement the NFV paradigm were originally designed to work in a data center environment, which has weaker requirements than NFV in terms of I/O. This section presents an overview of the different inter-VNF and VNF-host communication technologies; particularly, both software and hardware-based technologies are analyzed.

virtio + vhost

Virtio [Rus08; IBM15] is a standard that defines a common framework for device para-virtualization in different hypervisors. As shown in the Figure 3, the virtio architecture is split in two parts: frontend drivers that are executed in the guest (i.e., the operating system executed inside the virtual machine), and backend drivers implemented in the host (i.e., the hypervisor). A more detailed description of the frontend and backend drivers is the following.

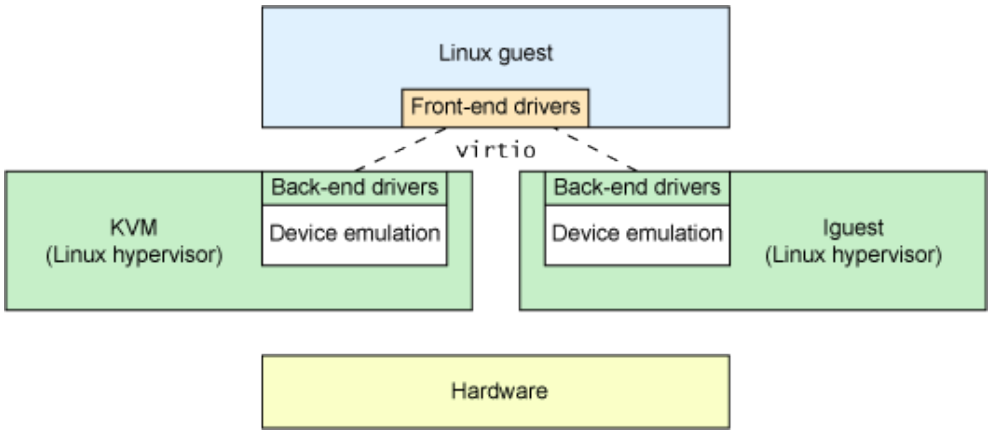

Figure 3: Virtio architecture

Frontend drivers

As shown in Figure 4, the **frontend drivers** can be subdivided into two different implementations categories:

- *standard drivers*: these drivers expose the virtio device to the guest as a standard network device. The biggest advantage of this approach is that standard applications can use it; on the other hand, it has as a drawback the overhead of the network stack (interrupts and data copies between user-space and kernel-space). Different implementations of virtio drivers are available for different operating systems as Linux, Windows, and FreeBSD.

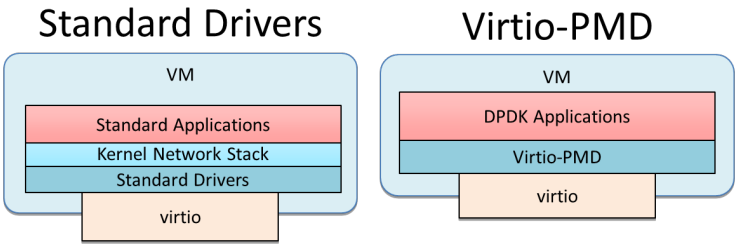

Figure 4: Virtio frontend drivers: a) standard Drivers, b) DPDK virtio-PMD.

- *DPDK virtio-PMD*: DPDK implements a Poll Mode Driver for virtio network interfaces [15d]. As all the PMD drivers, it presents huge performance advantages over traditional network drivers, because it bypasses the network kernel stack and uses polling instead of interrupts. The main limitation of it is that it can be used only by DPDK applications and that it works in polling mode, hence putting under pressure the CPU consumption.

Backend drivers

The **backend driver**, i.e., the part of code that runs in the hypervisor, is usually called **vhost**. There are different implementations of vhost, which are described in the following.

In the past, the virtio backend driver was implemented inside QEMU, which was in charge of receiving the packets from the virtio device and then sending them through a TAP interface. This solution required a high number of context switches for each packet. In fact, if we consider a packet transmission, the packet is first processed by the frontend drivers in the kernel of the guest; then it is processed by QEMU in host user-space; finally, through the TAP, it is provided to a vSwitch that forwards the packet in the kernel of the host.

Due to this overhead, recent implementations introduce the vhost module, which comes in two variants:

- vhost [15h]: vhost is a Linux kernel module that acts as backend driver for virtio devices. In this architecture the control and data-plane are separated; QEMU is still on charge of the control-plane, while the data-plane goes directly from virtio in the Guest to vhost in the host. Vhost basically receives the packets from the virtio device in the Guest and then sends them using a TAP interface that is usually connected to a vSwitch running in kernel-space (e.g, Linux Bridge or Open vSwitch). Its main drawbacks are the interrupt mechanism and the cost of sending packets through the TAP interface.

- vhost-user [15g]. In recent years there is a trend to develop data processing applications in user-space instead of kernel-space. Since vhost is executed within the kernel, it is not the best solution to connect those applications to VMs; then, in order to overcome this limitation, vhost-user has been developed. It is a new QEMU feature that allows user-space applications to act as backend for virtio-net devices; in this case the data traffic passes directly from the virtio device to the user-space application without passing through the host's kernel. Currently, DPDK implements vhost-user in the `vhost_library` [15f], which is used by Open vSwitch DPDK-enabled to support the so called `dpdkvhostuser ports`; such ports have, as a main characteristic, the fact that they work on polling mode.

The differences between the backend drivers just described are summarized in Figure 5.

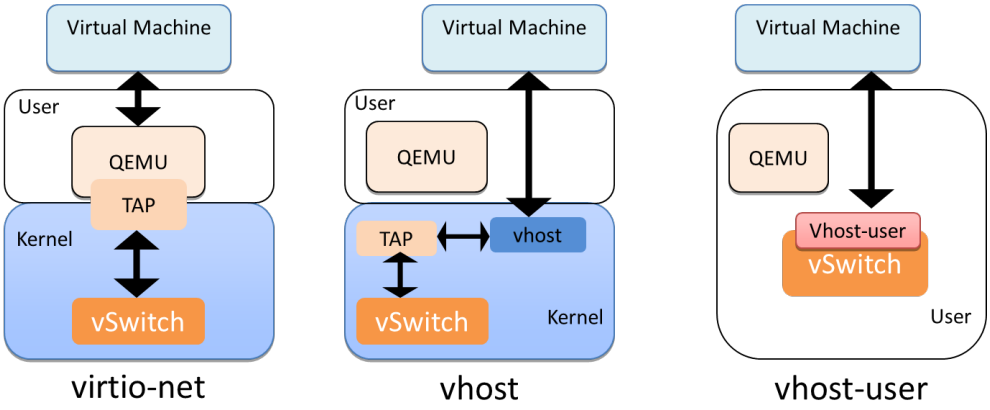

Figure 5: Virtio backend drivers: a) inside QEMU, b) vhost, c) vhost-user

Frontend driver	Backend driver	Comments
virtio-net (standard drivers)	vhost	Standard guest applications
		Host packet processing is performed in kernel-space
	vhost-user	Standard guest applications
		Host packet processing performed in user-space
DPDK virtio-PMD	vhost	Only DPDK guest applications
		Host packet processing performed in kernel-space
	vhost-user	Only DPDK guest applications
		Host packet processing performed in user-space

Table 2: Combinations of frontend and backend drivers.

Please notice that the mentioned frontend and backend drivers can be combined in any way, as shown in Table 2.

Dpdkr ports (IVSHMEM)

Inter-VM Shared Memory (`ivshmem`) [Mac15] is a mechanism to share (part of) the host memory with virtual machines running on KVM/QEMU. The memory is exposed to the guest as a Base Address Register (BAR) of a PCI device; applications can `mmap` the shared memory into its virtual address space. DPDK provides the `rte_ivshmem` [15b] library that allows the creation of `ivshmem` devices; as shown in Figure 6, such a library maps DPDK data structures into the device and adds some further information such as their virtual addresses in a metadata file. The information contained in this metadata file is then used by the Environmental Abstraction Layer (EAL) of DPDK (inside the guest) to map the data structures into the same virtual address as in the host; this allows the guest and host applications to exchange pointers without needing to perform any translation. Due to the presence of this meta-data file, the DPDK implementation of ivshmem is different from the standard implementation; it implies that a modified version of QEMU has to be used.

Open vSwitch DPDK-enabled implements a port type called `dpdkr` (Figure 7), which is composed of two DPDK rte_rings that are shared with the guest virtual machine using ivshmem. rte_rings are the DPDK implementation of a FIFO queue using a lock-less approach. `dpdkr` ports use zero copy (only pointers to packets are exchanged) and does not have any notification mechanism, then both Open vSwitch and DPDK applications have to work on polling mode. Finally, such a ports are not exposed as network interfaces to the

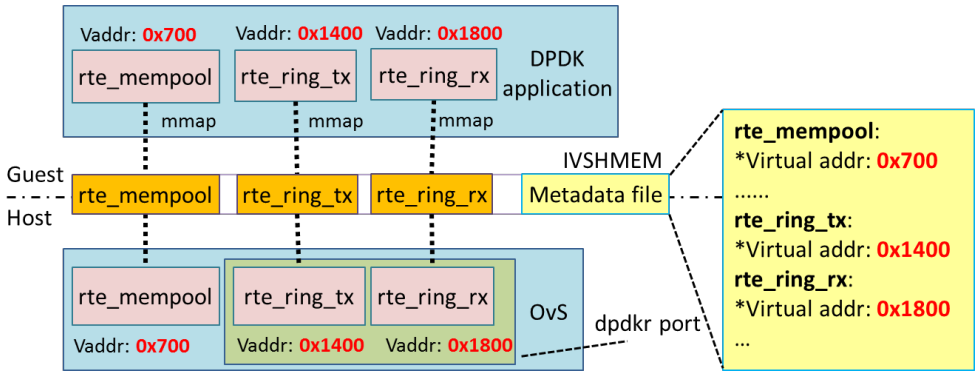

Figure 6: Sharing structures between OvS and DPDK applications.

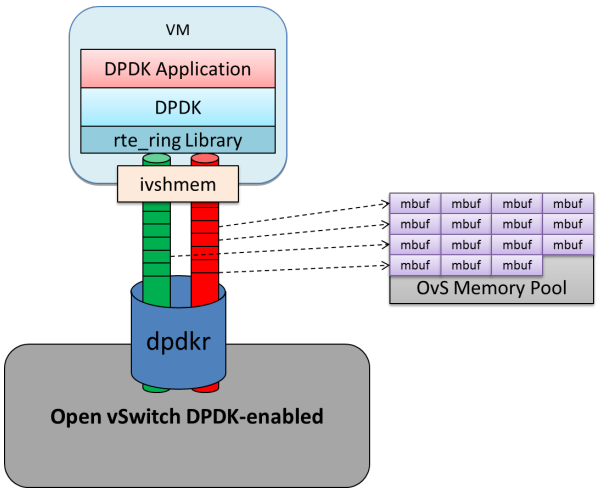

Figure 7: `dpdkr` port

guest: they are in fact exposed as a pair of rte_rings, hence applications have to use the DPDK rte_ring library to access to them.

Direct VM2VM

In NFV, it is very common that all the packets that output from one VNF have to go to the same next VNF, as in the case of the connection between the network monitor and the firewall in Figure 8. For this reason, in the context of Unify it has been implemented an extension of Open vSwitch [Pfa+09] that allows to directly connect two (DPDK-based) VNFs using `dpdkr` ports, in order to avoid the cost of passing through the vSwitch forwarding plane when moving packets among them. Notably, such a solution maintains compatibility with the applications executed in the VMs and with the other standard components of the (Universal Node) architecture (e.g., Openflow controller).

As already detailed in Section 1.3.2, the vanilla implementation of a `dpdkr` port (e.g., `dpdkr1` and `dpdkr4` in Figure 9) consists of a memory region that is shared between Open vSwitch and the guest DPDK-application. This idea was extended to create a special implementation of the `dpdkr` port, which is actually a direct connection between two VMs. As an example, consider the ports `dpdkr2` and `dpdkr3` in Figure 9, which are mapped on the same memory region so that packets can flow from one VM to the other without

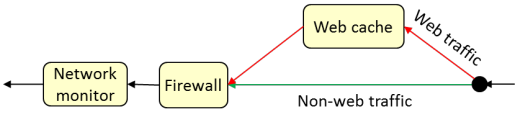

Figure 8: Example of service with a direct connection between two VNFs

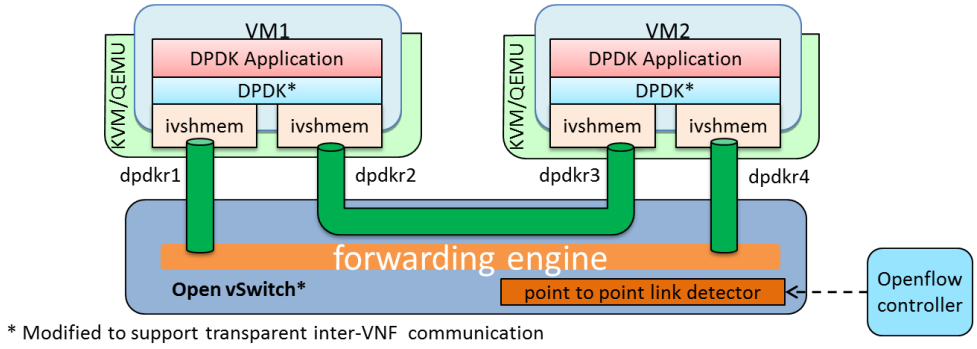

Figure 9: Direct connection between VMs through a new implementation of the `dpdkr` port

requiring the intervention of the vSwitch forwarding plane. In order to keep compatibility with external elements, these two ports directly connected are still managed by Open vSwitch and exported as standard `dpdkr` ports.

When Open vSwitch realizes that there is a direct path between two (standard) `dpdkr` ports (by means of our new *point to point link detector* OvS module, which analyzes the forwarding rules inserted by means, e.g., of the Openflow protocol) and it creates a new pair of `dpdkr` port mapped on the same memory region. At this point, the two VNFs involved in the direct connection must start to use the new `dpdkr` ports. Hence, it was defined a procedure that basically maps the new ports in an `ivshmem` device, which is then hot-plugged into the two communicating VMs; at this point DPDK running in the guest reads that device and overwrites the old `dpdkr` port with the new one.

Our solution allows changing the implementation of a `dpdkr` port between the standard and the optimized version *dynamically*. The process is also *transparent to applications* and most of the components in a typical SDN/NFV environment. In fact, applications do not need to be modified; they just require to be recompiled using a modified version of DPDK. Furthermore, neither the hypervisor, nor the Open Flow controllers, nor the host operating systems have to be modified at all. In fact, the modifications were only done in Open vSwitch and DPDK.

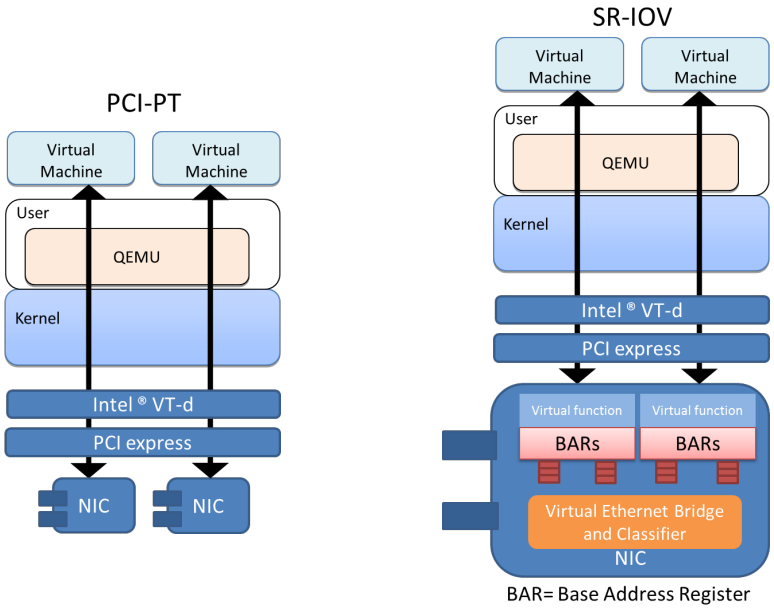

Figure 10: PCI-PT(a) and SR-IOV(b) technology

PCI-PT

PCI pass-through [15c] (or direct device assignment) consists in directly exposing a NIC to the guest operating system, which then runs the native driver of the device (there is no para-virtualization in this case). In order to support the direct assignment to a specific virtual machine, the hardware has to provide some special capabilities; particularly, it has to perform memory translations in order to let the guest operating system to access directly the NIC, as well as it has to provide a mechanism to enable the DMA transfer between the NIC and the guest memory. This technology allows reaching almost the same performance that in bare metal, but it has the big drawback that a whole NIC has to be assigned to a single guest. Due to the current processing and memory capacity of the servers, it is normal to have a rather large number of VMs running on them, then an unreasonable number of physical NICs will be necessary.

SR-IOV

The Single Root I/O Virtualization [15c] technology aims at getting the same PCI-PT performance while avoiding the huge limitation in the number of NICs. In order to reach it, SR-IOV allows a single physical NIC to be shown as a multiple physical NICs. This technology has the concept of physical and virtual functions: a physical function (PF) provides the full set of capabilities, configuration and data transfer, while a virtual function (VF) is lightweight and has restricted configuration options. Furthermore, VFs can be directly exposed to the virtual machines while PFs are used by the hypervisor to perform the configuration of the NIC.

Each VF has its own PCI configuration space and TX/RX queues in order to have insulation among them. Queues from different VFs are interconnected using a Virtual Ethernet Switch that is in charge of forwarding inter-VF traffic as well as forwarding external network traffic to the correct VF.

The support of the SR-IOV technology requires a SR-IOV-capable NIC, memory translation technologies and a hypervisor that performs the configuration of the NIC.

Conclusion

The different inter-VNF and VNF-host communication technologies introduced in this section are summarized in Table 3.

Technology	Zero copy	Interrupt support	Isolation	Implementation	Supported apps
virtio + vhost		Y	Y	Software	All
virtio + vhost-user		Y	Y	Software	All
ivshmem	Y			Software	DPDK
VM2VM	Y		Y	Software	DPDK
PCI-PT		Y	Y	Hardware	ALL
SR-IOV		Y	Y	Hardware	ALL

Table 3: Inter-VNF and VNF-host communication technologies

Compute Platforms

The portability of the Universal Node have been proved by installing the software on multiple hardware platforms with very different characteristics. Naturally, Intel's x86 were the first platforms, most of the performance measurements presented in this deliverable have been executed on Intel Xeon servers or, for what concerns the CPE-like platforms, Intel Atom. Recently, several other CPU architectures have been added; the following paragraphs give some detail on the individual platforms.

Intel x86

Firstly, the Universal Node was tested on several standard Intel servers, ranging from single CPU i7 machines to dual-processors Xeon platforms; in this case we used the standard compiling chain documented in the Universal Node repository and based on the Ubuntu 14.04 LTS operating system. In this case all the features (including the several supported softswitches) were turned on and tested. Most of the measurement results shown here for Intel platforms were drawn from a lower-end server with one or two sockets running Xeon 1630 CPUs and Intel X710 or XL710 network cards.

Power PC

One of two enterprise-grade CPE that the UN was deployed upon was a Freescale Hawkeye HK-0910[1], featuring also IPSec and L2 switch hardware acceleration. The software environment was based on the Linux Yocto project, which uses *recipes* to assemble together the required packages and create the software image that will be executed on the hardware platform. The overall software setup was very similar to the previous box, hence only NNF are enabled, although the platform should support also virtualization. In particular, the IPSec NNF was able to control the hardware accelerator, hence it was able to provide IPSec transfers at wire speed.

[1] Freescale QorIQ T1040, 1.2GHz (four e5500 cores), 64MB NOR Flash, 2GB RAM DDR3L-1600

ARM

In this deliverable we report the performance of "Banana Pi R1" switches running CarOS and being orchestrated by the UN-orchestrator. All source code is available at the CarOS public repository on GitHub[16a] [2].

Banana Pi R1 is an open hardware router, with a A20 ARM Cortex-A7 dual-core CPU, 1 GB RAM and five Gigabit Ethernet ports, four of which are connected to an onboard switch. The configuration includes the UN-orchestrator which manages the installed Open vSwitch. In the remainder of this document we will refer to this software and hardware configuration as *CarOS UN* as shorthand for the abovementioned system under test.

A domestic home gateway, namely a Netgear R6300v2[3] executes the Universal Node compiled for the OpenWrt architecture. Given the limited hardware capabilities of this box, the Universal Node was able to launch only native network functions, while the standard OVS was used as softswitch. In addition, service access points such as tunnels (e.g., GRE) and VLANs were supported, hence enabling to connect the Universal Node to external domains and to create complex services requiring the stitching of multiple sub-graphs, spanning across multiple domains.

MIPS

The second professional CPE was a Tiesse Imola 5, a professional CPE manufactured by an Italian SME based in Ivrea. The hardware platform was based on a Ikanos single core CPU[4], and also in this case the OpenWrt platform was used to create the running software image. However we had to use an old version of the Linux kernel (3.10.49) because it was the latest version supported by the Ikanos drivers that are needed to control the xDSL interface. However, this prevented us from creating GRE tunnels in OpenvSwitch due to a known incompatibility of that kernel version; in this case the Universal Node was able only to create VLANs toward external domains.

Methodology

Introduction

This section describes the benchmarking methodology that was applied to characterize the different Universal Node prototypes and development activities. Main focus of this deliverable is the performance evaluation of the individual components and platforms. As a number of partners were working on performance evaluation of different aspects, on different platforms, and with different computational capabilities, the results are not always comparable to each other. This section explains the different test suites used at premises, and outlines their commonalities and differences.

[2] Travelping contributes to the development of CarOS, a Linux-based OS distribution targeting carrier networks. To date, CarOS has been used in large production networks, for instance, on specialized hardware for session control. In the context of UNIFY, Travelping researches and evaluates the feasibility of, on the one hand, softwarization and orchestration approaches and, on the other, using very lightweight nodes running CarOS as UNs.

[3] 800MHz dual code ARM Cortex A9 CPU, 128 MB flash 256 MB RAM, 128 MB flash and 256 MB RAM, network connectivity provided by 4 GbE LAN ports, 802.11 b/g/n 2.4GHz and 802.11 a/n/ac 5.0GHz, and 1 GbE WAN port.

[4] Ikanos Fusiv Core Vx185, single core MIPS 34Kc V5.4 CPU running at 500MHz, 256MB RAM, 256MB flash, with xDSL acceleration.

Conformance Tests	Test 1 - RMI: Node information and capabilities Test 2 - RMI: Node available resources Test 3 - NMI: NF-FG deployment Test 4 - NMI: NF-FG modification Test 5 - NMI: NF-FG deletion Test 6 - NMI: NF-FGs list Test 7 - NMI: NF-FG data Test 8 - VSIRI: VNF specifications by type Test 9 - VSIRI: VNF specifications by ID Test 10 - VSIRI: VNF images
Functional Tests	Test 11 - URM: Discover and report available resources Test 12 - URM: Update available resources Test 13 - URM: Local scaling Test 14 - URM: NF-FG lifecycle management Test 15 - URM/LO: Optimized placement Test 16 - URM: Support to multiple VEE solutions Test 17 - URM: Support to multiple VSE solutions Test 18 - URM: Configuration of external connection to other nodes Test 19 - URM: Monitoring support and data report Test 20 - VSE: Dynamic deployment of LSI Test 21 - VSE: External traffic steering
Performance Tests	Test 22 - Throughput Test 23 - CPU Test 24 - Memory Test 25 - Switching Test 26 - Control Interfaces

Table 4: TI test framework test list. For detailed description, of each test, check Annex A to C.

Telecom Italia Test Suite

TILabs designed a test suite including three different groups of tests listed in 4. The detailed description of the configuration, procedure and expected results is reported in Annex A to C for each single test. The tests have been designed considering the node from three main viewpoints:

- UN interfaces compliance with respect to the primitives defined within WP2 for the corresponding reference points
- UN supported features with respect to the list of UN components and interfaces capabilities provided by WP5
- UN implementation and optimization by the evaluation of its performance

DPDK `pktgen`-based VNF Benchmarking Toolchain

As can be seen in Figure11, the NF node was treated as a black box by the Network Function Performance Analyzer (NFPA) node which was running DPDK PktGen via Lua scripts generated by the NFPA engine. The machines used were a Xeon 2630 (45nm, Sandy Bridge) and an Atom C2758 (22 nm).
Results were obtained, saved and analyzed. The following parameters are available in the NFPA database:

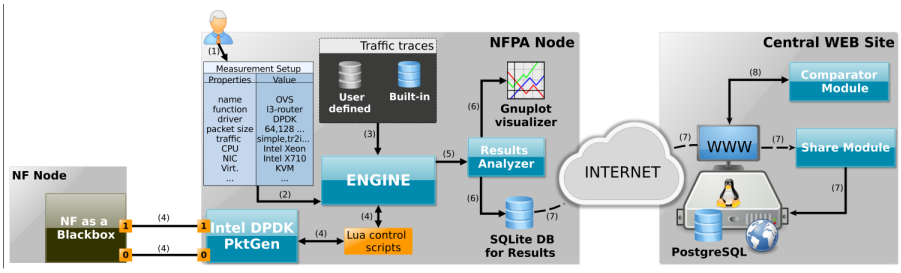

Figure 11: The architecture of the used measurement tool

- average and variance of bandwidth in bps
- average and variance of packet rate in pps
- average and variance of packet loss

Inter-VNF Communication Throughput

The goal of this section is to present the throughput testing of the different inter-VNF communication technologies available nowadays. Tests are executed on the architecture presented in Figure 12; as evident, it consists of a VNF (the Producer) that generates traffic, which is connected to a second VNF (the Consumer) acting as a traffic sink by means of a communication channel. The communication channel is the set of elements that are required to transfer data traffic from the Producer to the Consumer; it is usually composed of (Virtual) Network Interfaces and a (Virtual) Switch.

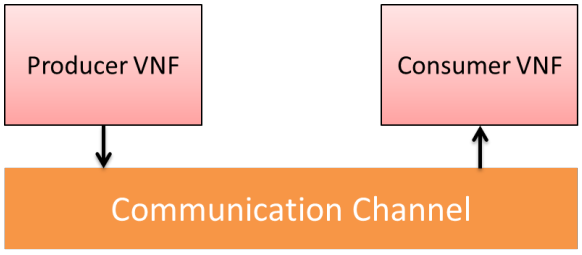

Figure 12: Principle setup of the Inter-VNF tests

KVM and DPDK

In this case, both VNFs run inside virtual machines, KVM/QEMU is used as hypervisor, and both Producer and Consumer applications are implemented using DPDK; in fact, DPDK is the only technology that supports all the interface types under test. The Producer sends packets to the Consumer as fast as possible; packets are sent and received using a burst of 32. Tests are executed using three different size of packets, 64, 700 and 1500 bytes. Due to the different kind of interfaces, the implementation of the Consumer and Producer applications changes among them; in all the cases but ivshmem and directVM2VM, the Producer application allocates and copies data to the packets, and the Consumer application frees[5] the packets.

[5]Note that *allocate* and *free* packets in DPDK terminology just implies to get/put buffers from/to the DPDK memory pool.

The following list shows the different technologies that are used in our tests in order to implement the communication channel:

- Virtio + vhost and standard Open vSwitch;
- Virtio + vhost-user and Open vSwitch-DPDK enabled;
- Ivshmem and Open vSwitch-DPDK enabled;
- direct VM2VM.

In order to give to the reader a clear vision of the different configurations, they are described in the following.

Virtio + vhost and standard Open vSwitch

As shown in Figure 13 (a), in this case the virtual NICs terminate in the vhost kernel module, which is in turn connected to Open vSwitch by means of TAP interfaces. Note that this implies that Open vSwitch has to be used without DPDK support, which also implies that Open vSwitch does not work on polling mode.

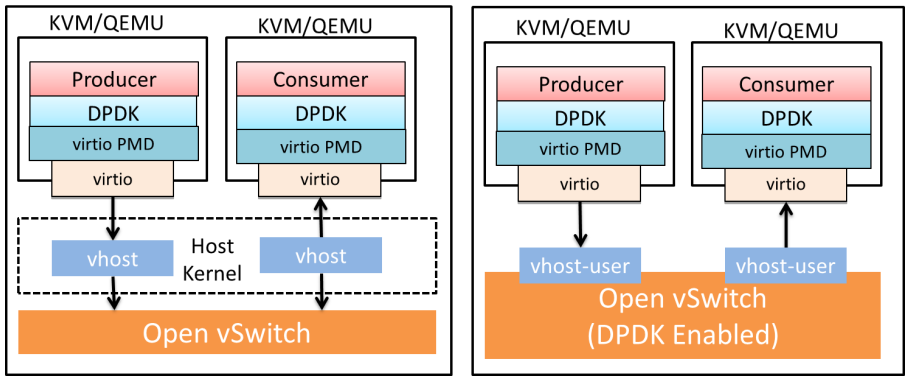

Figure 13: (a) Virtio + vhost and standard Open vSwitch. (b) Virtio + vhost-user and Open vSwitch-DPDK enabled.

Virtio + vhost-user and Open vSwitch-DPDK enabled

In this case, depicted in Figure 13(b), the vhost module is executed in user space.

Ivshmem + Open vSwitch-DPDK enabled

This configuration is shown in Figure 14(a).

Direct VM2VM

This is the optimization proposal that has been created by POLITO in the context of Unify. As shown in Figure 14 (b), in this case the virtual switch is bypassed.

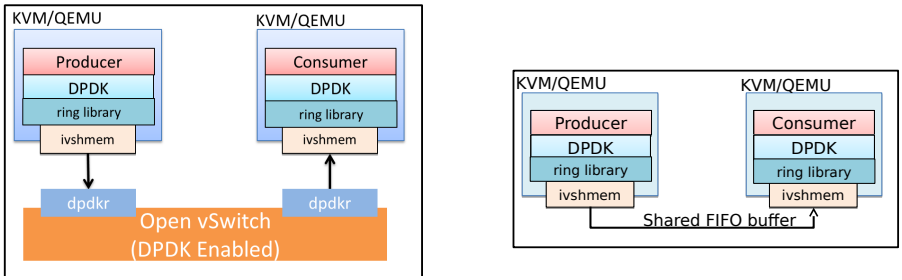

Figure 14: (a) Ivshmem. (b) Direct VM2VM.

Docker vs. KVM

In this case, the netperf application is used to generate and consume traffic. Two different scenarios are considered:

QEMU/KVM

netperf is executed in two different virtual machines, they are connected to a standard Open vSwitch using virtio - vhost interfaces.

Docker

netperf is executed in Docker container, they are connected to Open vSwitch using veth interfaces.

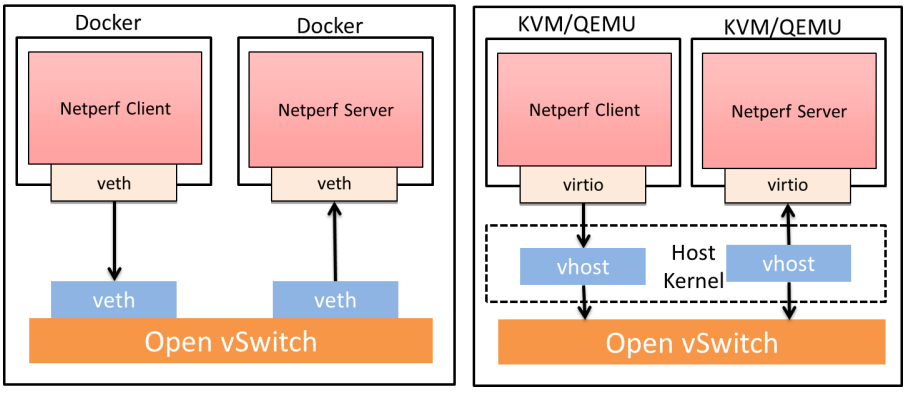

Figure 15: (a) netperf in Docker. (n) netperf in KVM

Inter-VNF Communication Latency

This subsection describes the setup used to measure the latency that an inter-VNF channel presents. The test setup presented in Section 2.4 does not allow to measure latency due to lack of tools capable of

measuring it accurately when using virtual interfaces. As show in Figure 16 (a) a new testing architecture was defined, packets are generated by an external server that is connected to the test server using 10GBs NICs. Once the packets enter the test server they go through two virtual machines and finally are sent back to the external server. To measure the time a packet requires to go from the first physical interface to the second, MoonGen (the traffic generation tool used in this test) uses the timestamping capabilities of the NICs. The VMs execute an ad-hoc application that takes packets from one port and puts them into another, without accessing the content.

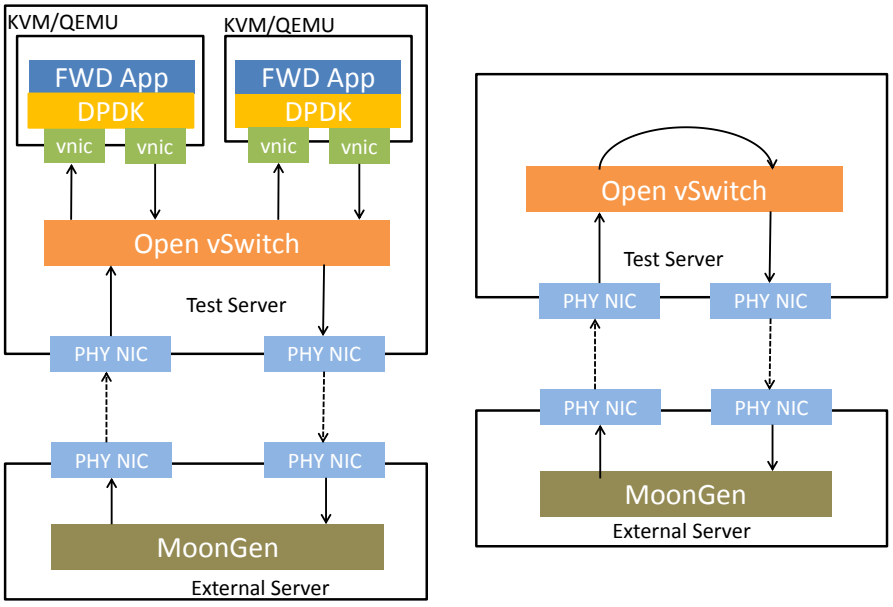

Figure 16: (a)Latency Testing Architecture. (b) Base Latency Measurement Architecture

It is clear that the latency related to the wires and to the OvS-to-NIC communication should not be included in the final results. In order to determine this "baseline measurement", an extra case is presented in Figure 16 (b), where OvS is configured to forward the packets between the physical NICs. Finally, the reported latency has been calculated as the difference between the one measured when the packets go through the VMs and the one when the packets simply go through OvS.
Finally, traffic is generated using different data-rates and only 64 byte long packets.

Hardware-accelerated I/O Throughput

The test environment used to benchmark the SR-IOV technology is detailed below. Two identical machines have been used, one as traffic generator and the other one as the Device Under Test (DUT) with SR-IOV support. The DUT is a bi-processor machine, 2x Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz (12 physical cores/24 threads) with 128 GB RAM (8x16 GB 2133 MHz). Two different network adapters have been tested on the DUT: a Dual-Port 10 Gigabit SFP+ Intel (R) Ethernet Converged Network Adapter X520 and a Dual-Port 10 Gigabit SFP+ Intel (R) Ethernet Converged Network Adapter X710.
PktGen v2.9.12 (RTE 2.2.0) has been used for traffic generation on one machine, and SR-IOV has been configured on the DUT with two Virtual Funtions (VFs) for I/O on two different Physical Function (PFs). Moreover, a QEMU/KVM VM is running on the DUT, on which the L2FWD process (DPDK 2.2.0) is

running. Additionally, an Endace DAG 9.2X2 card has been used (with an optical splitter) to monitor and capture the traffic between the traffic generator and the DUT, in order to obtain the performance numbers summarized in Table 9. In order to obtain the 95% Confidence Interval, the throughput has been calculated over 30 repetitions for each packet size value. It must be highlighted that for 1024B and 1500B packet size the throughput value has been the same for all the repetitions (same value for the 100% of the measurements).

Improvements on PktGen for Throughput Calculation

Several improvements have been developed as extensions to the DPDK lua scripts to automate the throughput measurement and confidence interval calculation. Moreover, the actual value of throughput is obtained from the DAG card, so the correlation between the PktGen results and values obtained from the DAG have been also automated.
Moreover, a patch has been submitted to DPDK (through the DPDK-dev mailing list) to increase the PktGen granularity. This patch allows to improve the resolution of the traffic generator to admit decimal values for rate definition. In the original code, the packet rate can only be an integer value, which has an impact on bigger packet size measurements, since 1% rate hides a bunch of possible throughput values (i.e. the tool does not have enough resolution).

Setup for PCI-PT and vhost-user connected DPDK BNG

The setup for the PCI-PT measurements was similar to the one described in for SR-IOV, however the available hardware was less powerful. The DUT was a SuperMicro X10SRi-F, 64G RAM with Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz (6 cores) and an Intel (R) X710 network card, partially also a XL710. As the use case tested here was more complex than L2 forwarding, the traffic source and destination had to be different as well. Therefore, an Intel-developed traffic generator named prox was used. This one was capable of generating GRE and QinQ traffic.

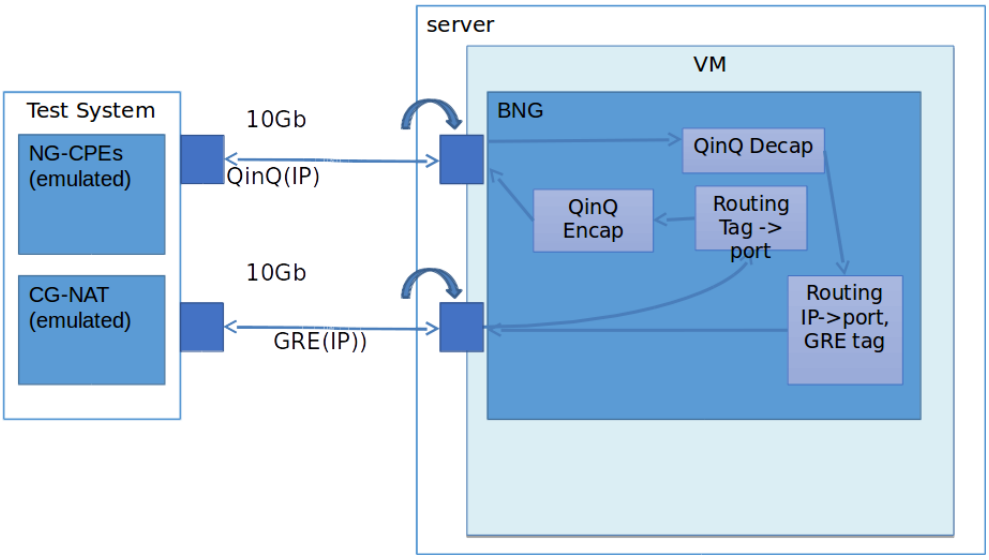

Figure 17: Setup for testing different I/O variants with the Intel DPDK BNG prototype.

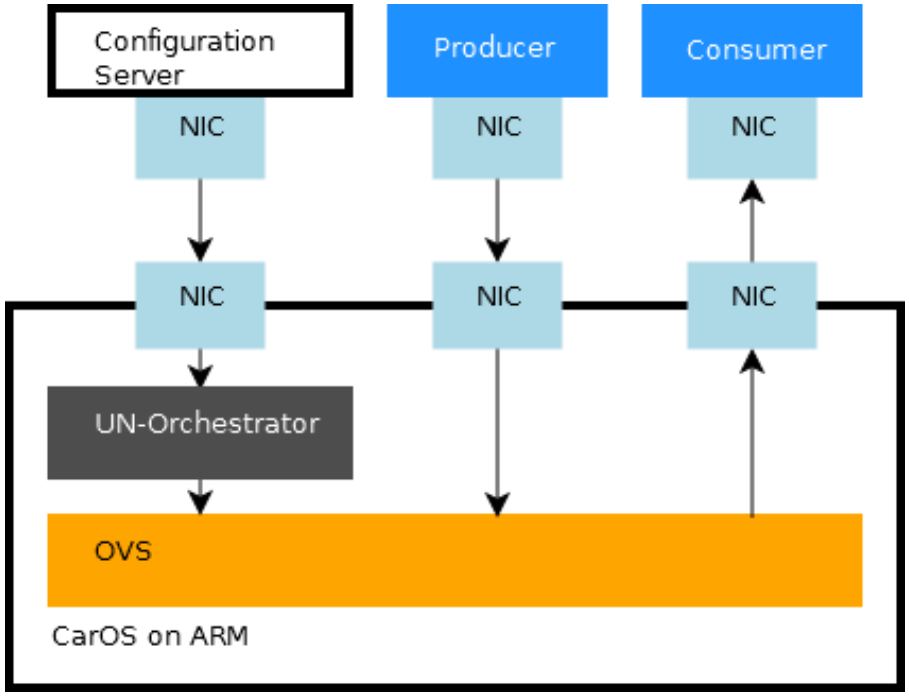

Figure 18: CarOS UN test configuration.

Banana-Pi (ARM) Test Description

Figure 18 illustrates the high-level test configuration used in evaluating the CarOS UN. We use dedicated traffic producer and consumer machines that can generate and receive traffic in excess of 1 Gb/s and employ the UN-orchestrator prior to each test reported below to configure the CarOS UN.

Results

This section contains the benchmarking results of the above shown different aspects.

Type 1 to Type 5 VNFs

Performance measurements were done with a DPDK Pktgen based toolchain described in Section 2.3 and depicted on Figure 11.
The execution environment (bare metal, container or VM) has an effect on the achievable performance. As a rough summary it can be stated that the virtualization environment has relatively small impact on performance as long as zero-copy interfaces are in use between the host and the guest and architecture emulation is not required (meaning that the host architecture is sufficient to run the VNF).

QEMU/KVM virtual machine vs. containers (type 1-2) vs. bare metal DPDK

The computational overhead of a virtual machine is relatively small (around 10%). As stated above, the results depend mostly on the used interface between the host and the guest process (see below). The overhead of containers is slightly even lower than a full virtual machine (around 5%).
The performance difference between classic virtualization and containers are depicted in Figure 19. The

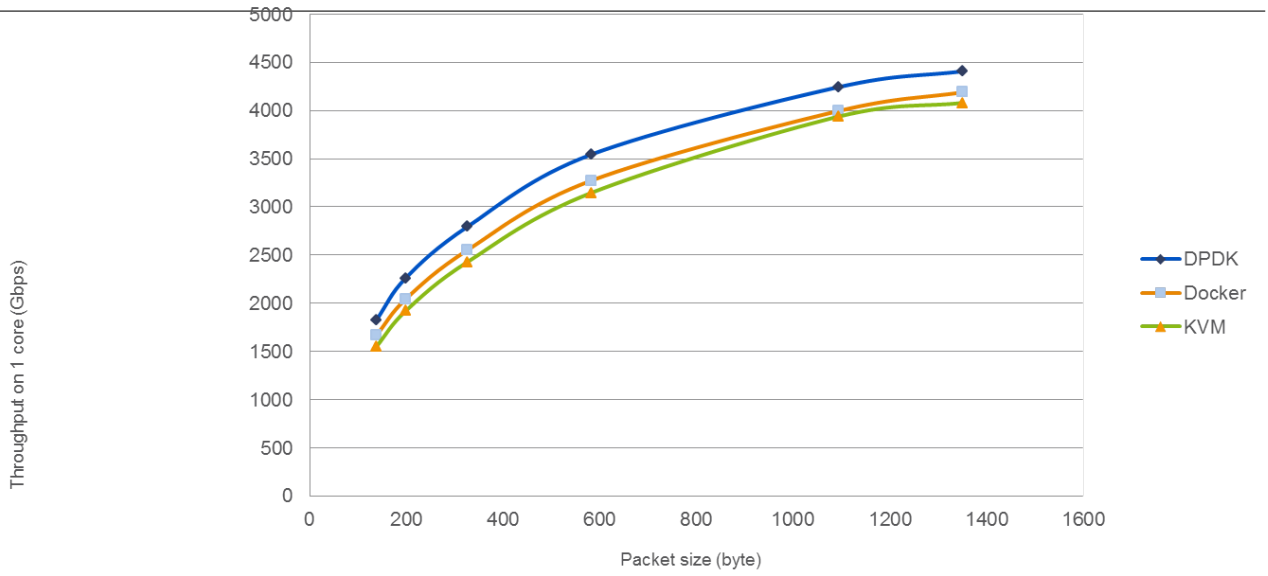

Figure 19: Performance of bare metal vs. containers vs classic virtualization

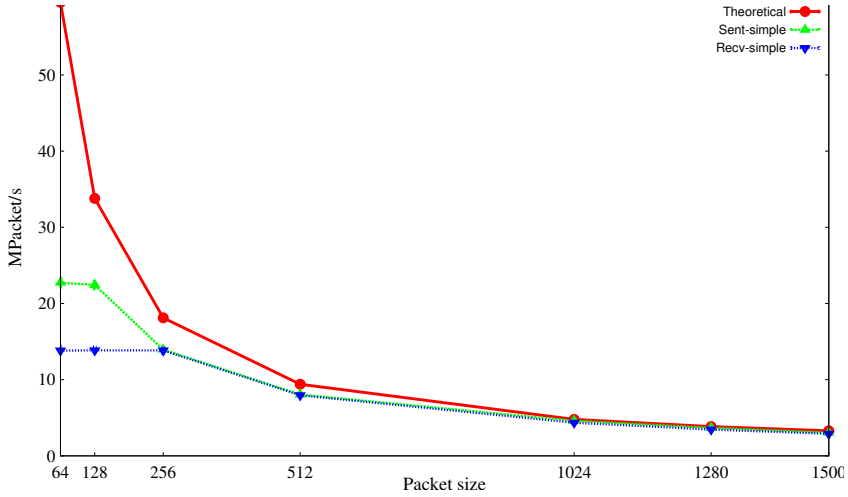

Figure 20: Performance of simple port forwarder in KVM using ivshmem

measured setup used a complex "base station VNF"d that implements IPSec (AES) decoding, GTP decapsulation, IPSec encoding and GTP encapsulation.

In simpler setups the difference between the environments can be bigger due to the fixed cost type elements introduced by the virtualization layer. For example on Figure 20 it is shown that a simple port-forwarder can do 13 Mpps with KVM using the shared memory interface (ivshmem), while the bare metal performance of the given application was around 15 Mpps.

Type 4: Virtual Switch Plugin

There are two examples for this so far: VxLAN and GTP-U processing was integrated to ERFS as a switch plugin (extension). Since these plugins were integrated into the pipeline the performance we can get is relatively high. Just the tunneling can be done with 16 Mpps / core, while a more complicated pipeline with VxLAN is shown below at the native function section, where the entire pipeline has 10 Mpps per core performance.

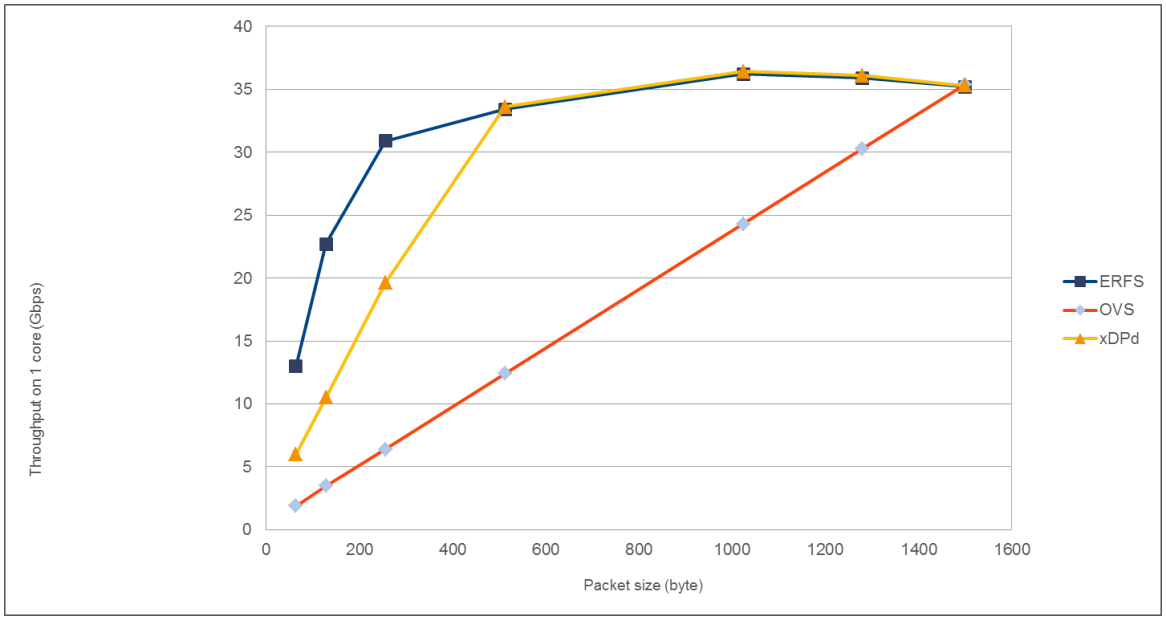

Figure 21: Performance of L2 type 4 VNF (bare metal)

Type 5: Native Function

Native function in our case can mean two things:

- either a native Linux function that is supported by the host OS (e.g. iptables) or
- a VNF function that is implemented by the virtual switch

In this section the latter is discussed, since kernel-based VNFs have been well studied for years now.
Figure 21 shows the difference between different OF switches acting as type 5 Ethernet switches on bare metal environment. It is clearly visible that with an ERFS based type 5 switch it is possible to beat even the optimized DPDK l2fwd VNF. In this scenario a 40GbE NIC was used to forward the packets. The switches were loaded with 100 MAC rules and 100 different MAC addresses were used in the packet trace to use all rules. It is visible that OVS 2.4 had some issues with DMAC forwarding, while xDPd and ERFS behaved as expected.
Another important type 4 use case was L3 routing. The pipeline that was configured for this use case is visible on Figure 22.
The results (packet per second vs. number of IP flows) of the L3 use case are summarized in Table 5. The packet size was 64 byte.

Flows	ERFS	OVS 2.4	OVS 2.4 - DPDK
1 flow	16021	n/a	5567
100 flow	12856	927	4156
10k flow	12624	783	3318
500k flow	12675	3	311

Table 5: Results of the router pipeline (kpps per core, 1-100-10k-500k IP flows, Xeon)

The Atom results are summarized in Table 6. The packet size was 64 byte. Overall it seems that when we normalize the results with power consumption the Atom machine is ahead with 50%, which most probably

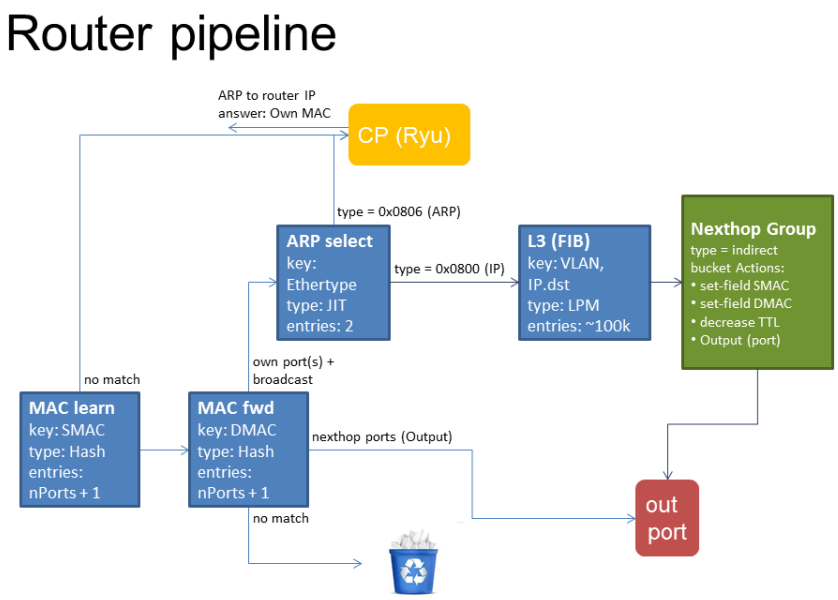

Figure 22: The setup of the L3 pipeline

the effect of the newer architecture and the 22 nm design.

Flows	ERFS	OVS 2.4	OVS 2.4 - DPDK
1 flow	3742	n/a	2565
100 flow	3580	174	2036
10k flow	3545	141	1001
500k flow	3543	n/a	n/a

Table 6: Results of the router pipeline (kpps per core, 1-100-10k-500k IP flows, Xeon)

Tha last native function was a bit more complex pipeline that implemented a datacenter gateway. Here inside the datacenter we assumed to have L3 connectivity, so the tenants / services are differentiated by using VxLAN. That means that incoming packets are classified against the configured virtual IP addresses and if a valid service is found, the DCGW load balances between the blades where the selected service is deployed. The required OpenFlow pipeline can be seen on Figure 23.
Since OVS and xDPd don't support VxLAN inside the pipeline, we tried to use MPLS tunnels instead of VxLAN. This is slightly more lightweight, but still gives us an assumption of what OVS and xDPd are capable in a tunneling based scenario.
The results (packet per second vs. number of IP flows) of the L3 use case are summarized in Table 7. The packet size was 64 byte. The -X sign meand Xeon results, while -A are atom results.

Host/VNF Interfaces

We investigated ivshmem, userspace vhost and virteth interfaces. The simplest use case is that in the guest we have a simple port forwarder, while in the host we run a DPDK-based switch. For this measurement we used OVS 2.4 DPDK in the host. The results of this use case are summarized in Table 8. The packet size was also 64 byte.

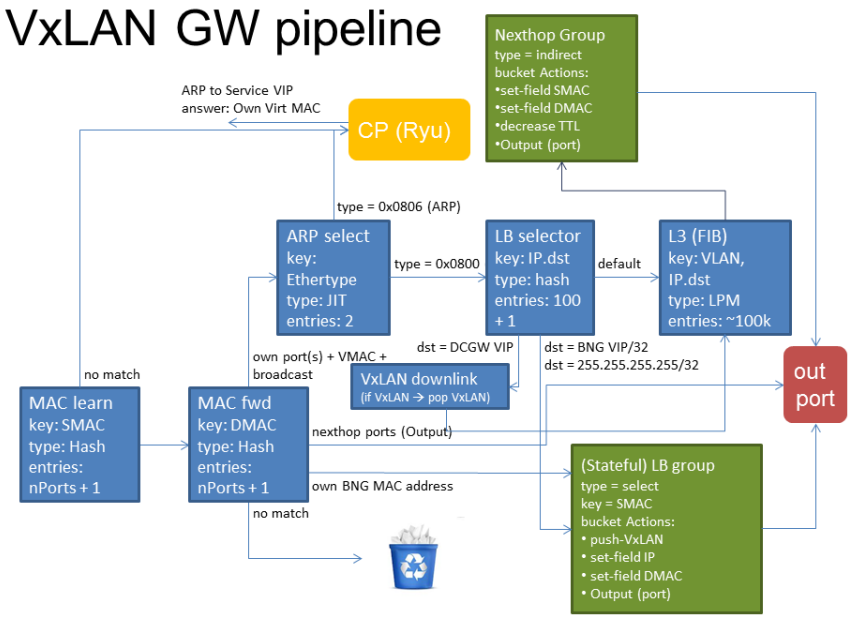

Figure 23: the setup of the datacenter gateway pipeline

Flows	ERFS	xDPd-DPDK	OVS 2.4-DPDK
uplink-X	8477	798	3288
downlink-X	10807	706	2914
uplink-A	2193	168	578
downlink-A	2760	164	572

Table 7: Results of the DC GW pipeline

Hardware-accelerated I/O: PCI-PT, SR-IOV

This table shows the throughput of the complete setup described above in section 2.6. This means that this benchmark does not present results of the SR-IOV technology, but its use in a realistic setup with a VM running the L2FWD/DPDK process. The main conclusion is that the bottleneck is the L2FWD process, since with this performance numbers the SR-IOV I/O does not lose any frame. The throughput of the L2FWD/DPDK VM process using SR-IOV technology can be seen on Figure 24 (Gbps) and Figure 25 (Mpps).

The analysis of packet loss at 100% in the same setup is also interesting to understand which could be the impact and where is the real bottleneck. When considering the worst case with 64B, the results are the following shown in Table 10.

This table shows that when line rate (100%) is configured on PktGen (10G) with 64B frames, the packet lost is 0.0036% at the SR-IOV and 4.55% at the L2FWD/DPDK process, which actually becomes the bottleneck of the system. There is a negligible packet loss at SR-IOV, while the L2FWD/DPDK experiments a higher packet loss. Anyway, the SR-IOV does not have an impact on throughput calculation, since the limiting factor is the forwarding process at the VM.

Pktsize	ivshmem	uvhost	veth
64B	5223	2700	212
128B	5221	2611	211
512B	5211	*LR-10GbE	213
1500B	*LR-40GbE	*LR-10GbE	211

Table 8: Results of the DC GW pipeline

Pktsize	Theoretical Max (Gbps)	Throughput (Gbps)	95% Confidence Interval (Min-Max))
64B	7.619	7.054	7.031 - 7.077
128B	8.649	8.581	8.571 - 8.592
256B	9.275	9.264	9.259 - 9.268
512B	9.624	9.608	9.607 - 9.610
1024B	9.808	9.8	9.8 (100%)
1500B	9.868	9.86	9.86 (100%)

Table 9: Throughput of the SR-IOV with L2FWD/DPDK benchmark

Intel DPDK BNG within guest machine vs. bare metal

Using DPDK's poll mode driver is drastically improving I/O performance, however, this comes at a price. In the setup of the BNG described in 2.6.2, four different variants of I/O have been compared: virtio as base line, OvS with vhost-user, PCI-PT, and finally a bare metal implementation of a BNG as a DPDK process. In terms of throughput, virtio showed a similar low performance as other measurements in this series, confirming the base line. Using vhost-user to feed the two interfaces going in and out of the BNG VM, a remarkable increase to about 4.2 Mpps could be measured. Not surprisingly, though, a direct pass-through of the PCI interfaces into the VM yielded around 19 Mpps. By this time, the running application became the bottleneck rather than the I/O. This could be observed from frame drops within the application. In order to further remove limitations of virtualization from the setup, the DPDK application was put directly on the server, consuming all available cores, and yielding a further increase to roughly 22 Mpps. We noted frame losses within the application from around 73% offered load, however we attribute these to the suboptimal reduction in size for the server platform at hand (having 'only' six physical CPU cores appears way too small for DPDK application designed for performance).

Power Consumption of DPDK BNG within guest machine vs. bare metal

Measurements of the total power consumption of the server were performed along the variation of offered load from the external source. During the boot-up of the server, a pattern appeared that could not be depicted in a diagram, but instead needs to be described textually. Power consumption of the system under test started at 8W (When the server is not running, but still consumes power for network cards, clocks etc.) When starting the server, power consumption briefly goes up to around 90W, before the power saving features of the operating system kick in and bring the total consumption below 60W. Normal operation of the server would now not bring it above 70-80W, even with KVM starting virtual machines.
The moment that the BNG DPDK process is switched on (Figure 26 starts at an offered load of 0), the CPU practically draws more than twice the current, pushing the total power consumption up to 118W. For bare metal, the increased offered load now leads to a steady increase of power consumption, albeit the total

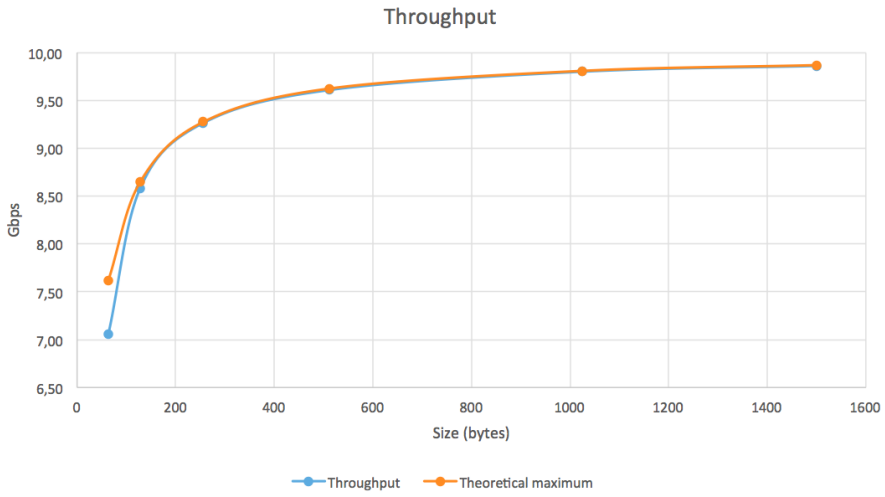

Figure 24: Throughput of the L2FWD/DPDK VM process using SR-IOV technology (Gbps)

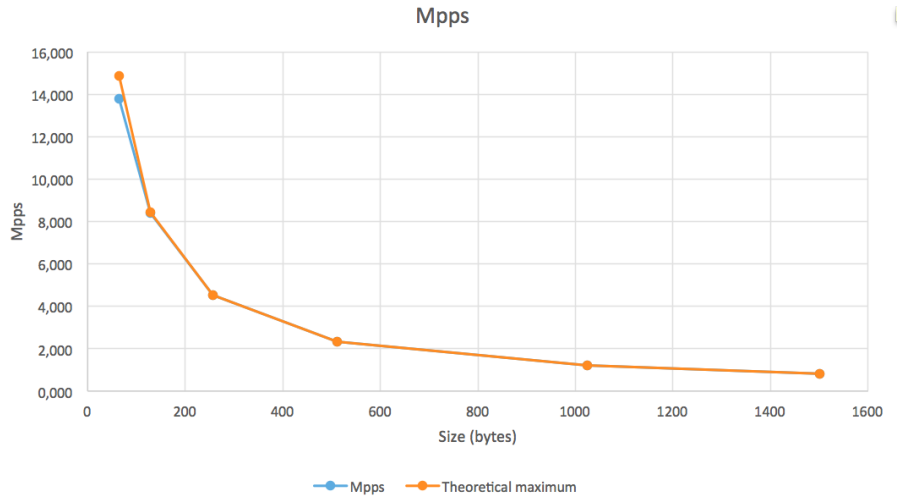

Figure 25: Throughput of the L2FWD/DPDK VM process using SR-IOV technology (Gbps)

difference is 7W between no offered load and full line rate.

The situation is different for PCI-PT and the DPDK process running in a VM. While the mere fact that the VM was brought up barely increased power, invoking the DPDK process within the VM pushes up the consumption to 131.5W, and the first packets arriving caused a peak in power consumption, while a further increase in offered load actually reduces power. Lacking deeper insight into the details of Xeon CPUs, we can only assess this behaviour. A speculation would be that the initiation of PCI bus transfer from the NIC to the CPU core has an overhead that shrinks relative to the amount of data transmitted per PCI access. The worst performance (both lowest I/O and highest power consumption at 145W) could be observed with virtio. Results are being collected in Table 11.

One conclusion that may be drawn from these measurements is that when normalizing the power consumption by the achieved throughput, a benchmark of Joule/packet appears useful, and that for the particular application these values differ from 5.17 $\mu J/packet$(bare metal) to 217 $\mu J/packet$ (virtio). These values may become particularly of interest when considering energy-limited nodes. These values are application specific, and several different applications should be benchmarked when characterising a platform.

Device (Tx/Rx)	Number of Packets	rate(%)
Traffic Generation (Tx)	1000000000	100
DUT SR-IOV (Rx)	-36069	-0.0036
DUT L2FWD/DPDK (Tx)	-45449969	-4.55

Table 10: Packet Loss of the SR-IOV with L2FWD/DPDK with 64B (100%)

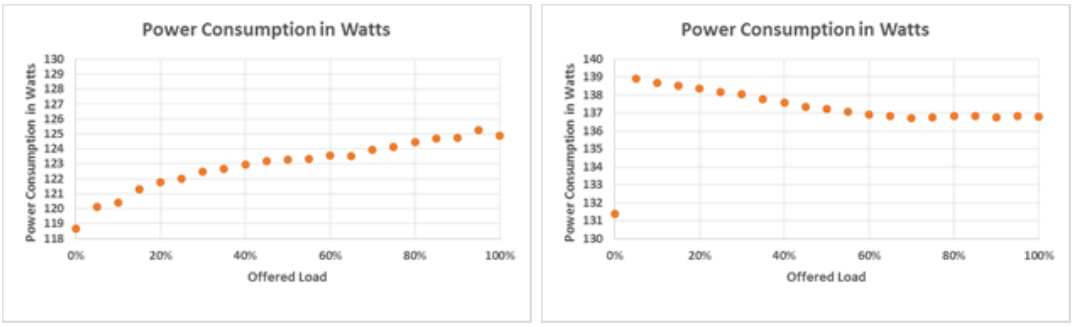

Figure 26: Power consumption of the Intel DPDK BNG in bare metal implementation (left) and PCI-PT (right) for total offered load up to 20 Gbit/s.

Inter-VNF Communication Throughput

This section contains the results of the tests whose methodology presented in Section 2.4. DPDK 2.0.0 and Open vSwitch 2.4.0 were used.
The test server has two Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz CPUs, 10 physical cores / 20 logical cores, 64kB per core L1 cache, 256kB per core cache L2 and 25600 kB per socket L3 cache, 64 GB of ram (8*8GB DDR3 1600 MHz). 2048 2MB huge pages were allocated, all the cores but the first were isolated, (i.e, core 0 was dedicated to the operating system). In all the cases the cores of the sending and receiving processes (KVM or Docker) were pinned to different physical cores (all of them isolated to avoid task pre-emption). In the case of OVS-DPDK it was configured to use a single core in all the cases.

KVM and DPDK

Figure 27 shows the results for the KVM and DPDK testbed. It is worth mentioning that, as it will be stated in Section 2.4, ivshmem and directVM2VM do not include neither allocation nor data copy. For this reason the results do not depend on the size of the packets.
The performance of standard virtio is very low, just 0.4 Mpps, while the DPDK-based vhost-user has a performance that is one order of magnitude bigger than the standard one, 5 Mpps.
The ivshmem and directVM2VM channels show impressive throughput thanks to their zero copy characteristic, specially the case of directVM2VM that reaches about 350 Mpps.

	bare metal	PCI-PT	vhost-user	virtio
max. throughput (Mpps)	21.95	18.98	4.2	0.67
max energy consumption (W)	125.2	138.9	n/a	145.6

Table 11: Throughput and power consumption for different I/O techniques and the Intel DPDK prototype.

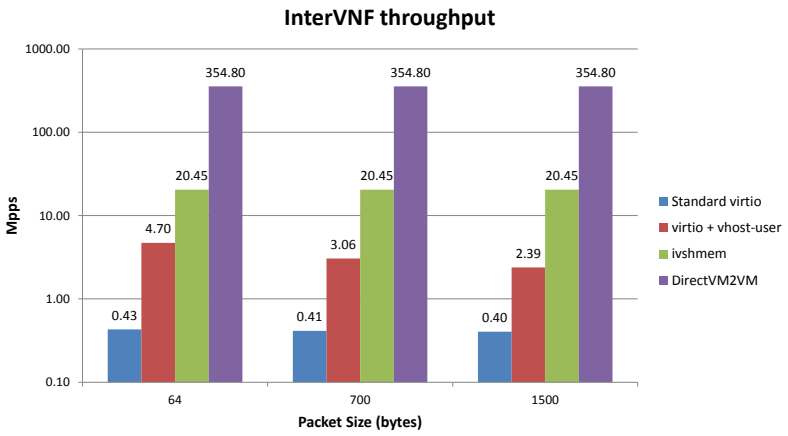

Figure 27: Inter-VNF communication throughput

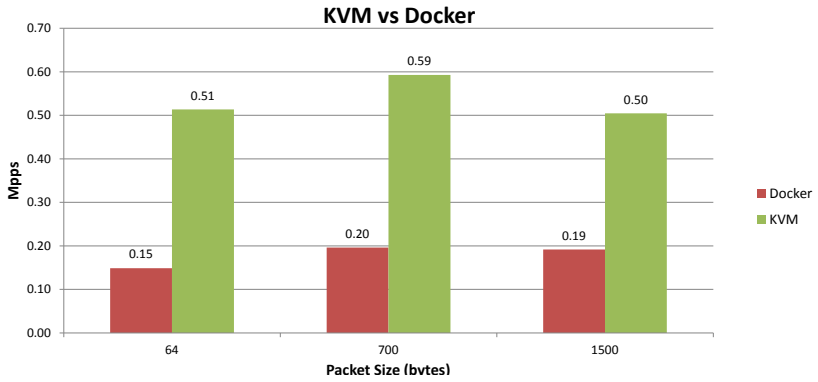

Figure 28: Docker vs KVM throughput results

KVM and Docker

Figure 28 show the comparison of network performance between KVM and Docker. As evident, even vanilla OVS (not to mention optimized softswitches, based on DPDK) is much faster than Docker. This clearly shows how Docker is a risky choice when high throughput VNFs have to be instantiated.

Inter-VNF communication Latency

These test were carried out according to the methodology presented in Section 2.5, while the servers and the core pinning configuration were the same that in 3.4. The numbers that are presented should be taken just a relative measure, because the latency includes also the time required for the forwarder application that is executed as VNF.

Figure 29 shows the latency when a data rate of about 50% of the reported in Section 3.4 was used; this number was chosen in order to avoid filling up the queues in the system, which will severely impact the latency numbers. Instead, in this test we wanted to measure the baseline delay introduced by each technology, hence (possibly) with empty queues.

Results show clearly that the standard OvS presents a huge latency compared to the other cases, due to the data copy performed by vhost-user it presents also a latency bigger than the ivshmem case, finally direct VM2VM shows the lowest latency.

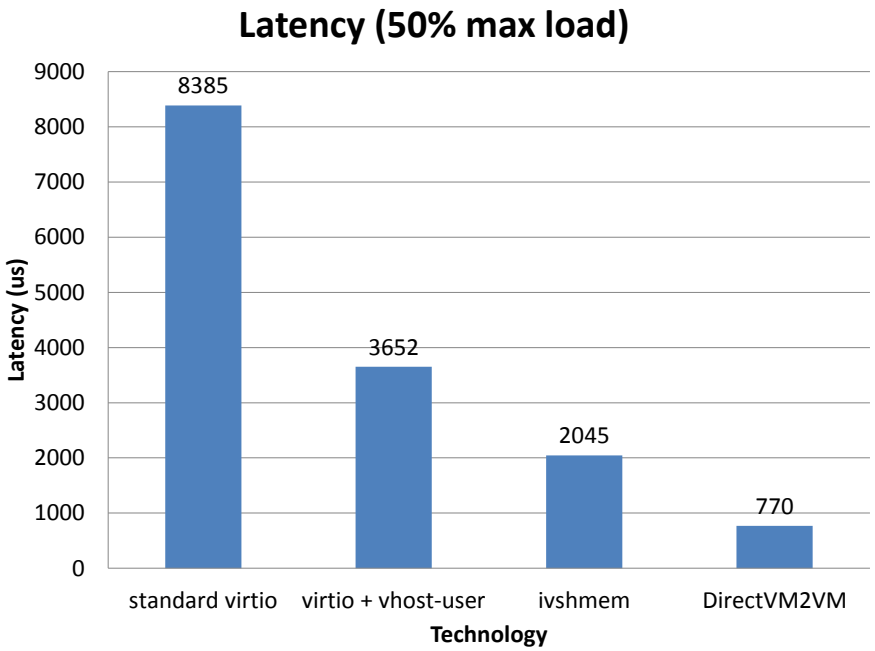

Figure 29: InterVNF latency using 50% load

Figure 30 shows the measured latency with different data rates. It demostrates that the latency increases with the data rate, after the 50% it becomes huge compared to the results with low data rates, this behavior is due to the fact that the queues of the vSwitch become full and there are packet losses. It also shows that OVS-DPDK is able to achieve excellent latency numbers with low data rates, in fact this technology, which makes use of a polling mechanism, is very sensible to the data rate when considering the delay introduced when processing traffic.
Finally, please note that the max throughput that can be reached using directVM2VM is 350Mpps, which is impossible to reach using 10GBps NICs. For this reason the scale of the directVM2VM line has been changed, the maximum data rate in this case was only 3%.

Measurement Results

We measure delay using `ping`, and throughput in terms of capacity and packet switching speed using `iperf`. We consider the Ethernet packet sizes suggested in RFC 2544 in our measurement campaign, which involves one, two, three or four CarOS UNs connected back to back, forming a chain topology. Specifically, for the delay measurements illustrated as boxplots in 31, we experiment with packet sizes of 64, 512, and 1024 bytes. As can be surmised from the figure, adding one CarOS UN adds approximately 0.2 ms of delay on median, irrespective of the packet size.
Similarly, we measure throughput in terms of packet switching speed and capacity for various packet sizes, employing one or more CarOS UNs. 32 illustrates the measured throughput when one CarOS UN is placed between the traffic producer and consumer machines. For small packet sizes (e.g. 256 bytes), throughput in terms of packets switching speed (left axis, bar plot) exceeds 55000 packets per second, while delivering slightly over 70 Mb/s. For Ethernet MTU packet sizes, we measure more than 35000 packets/s, and almost 95 Mb/s.
33 illustrates that when we use a chain topology with two CarOS UNs we do not measure siginificant changes

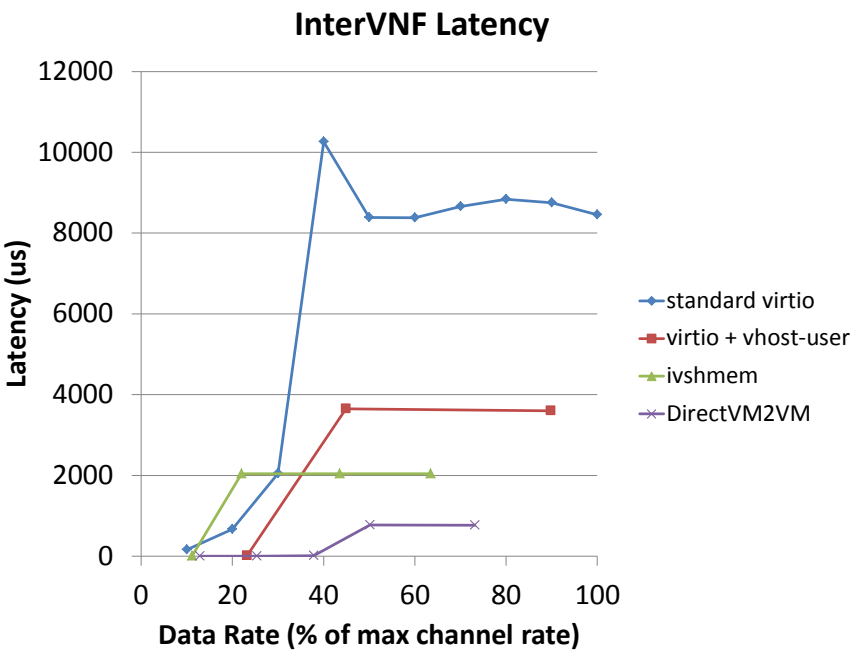

Figure 30: InterVNF latency using different data rates

in terms of packet switching speed. We do measure significantly higher throughput, saturating at nearly 450 Mb/s for Ethernet MTU-sized packets. This is also reported in 34 where we present results from the lab setup illustrated in Figure 35 with a chain topology of three CarOS UNs between the traffic generator and sink.

Overall, our component measurement results illustrate the feasibility of using lightweight resource nodes orchestrated via the UN-orchestrator while at the same time delivering performance in terms of delay and throughput that makes them suitable for a range of applications at the edge or home networks.

Conclusion

We have seen in this deliverable that the UN has been designed as a concept that allows joint scheduling of compute and network services, a hyper-converged VNF infrastructure in Network Function Virtualization (NFV) terms. Our research and development work points out that the UN makes a quite unique proposition, particularly useful in smaller setups, and able to scale up to multi-node or multi-rack settings. A key advantage of the UN on smaller devices is that an entire OpenStack is simply too heavy weight, for instance, in a CPE, or practically of little to no value as network optimizations cannot be attained as would be expected in a carrier-grade network. Through our work, on the contrary, extending the reach of the UN across the entire network in the framework of multi-domain orchestration with the UN Orchestrator, all UNIFY technologies, including the overarching orchestrator as well as the SP-DevOps toolset, apply elegantly and efficiently.

VNF chaining via a softswitch is a straightforward implementation in single-node settings. Here, a flow between multiple network functions crosss the border between the - typically kernel/DPDK/ODP-based - software switch and the VNF container/VM every time traffic is moved between VNFs. The fine-grained decomposition of network functions, as it is suggested by UNIFY, calls for high-performance implementation

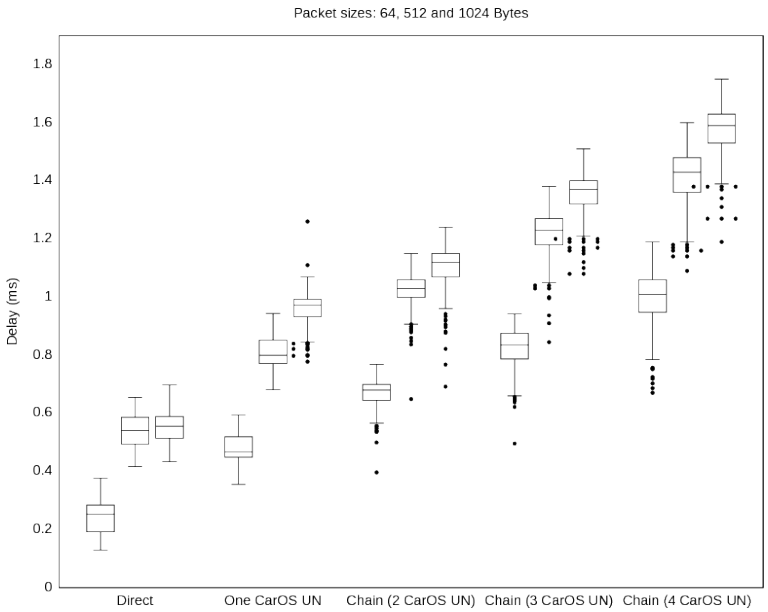

Figure 31: CarOS UN delay measurements for various packet sizes and number of nodes in a chain

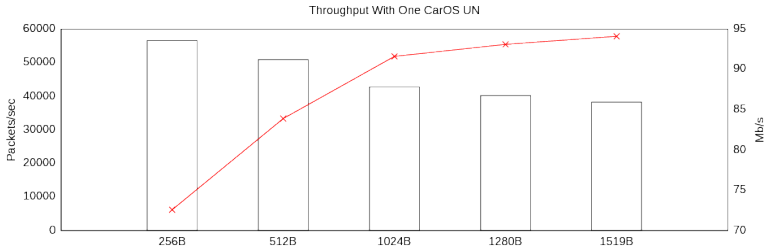

Figure 32: CarOS UN throughput measurements for a single device; various packet sizes. Left axis indicates the scale for the bar plot (packets/s), right axis indicates the scale for the line plot (b/s)

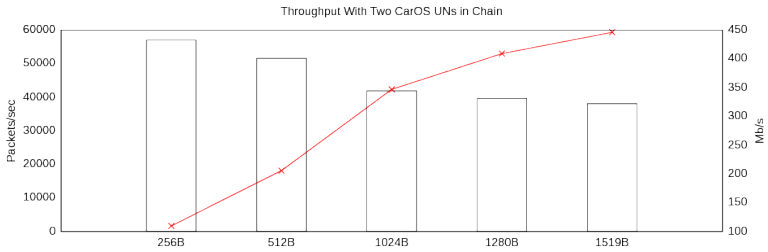

Figure 33: CarOS UN throughput measurements for a chain of two devices; various packet sizes. Left axis indicates the scale for the bar plot (packets/s), right axis indicates the scale for the line plot (b/s)

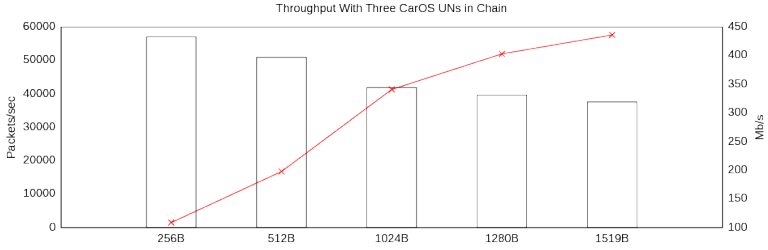

Figure 34: CarOS UN throughput measurements for a chain of three devices; various packet sizes. Left axis indicates the scale for the bar plot (packets/s), right axis indicates the scale for the line plot (b/s)

Figure 35: A chain topology with three CarOS UNs used for the measurement tests presented in Figure 34.

of this particular aspect.

A first order evaluation result is that straightforward implementations of VNF chaining which simply employ OVS and KVM or Docker containers are practically limited from a I/O performance standpoint. As the compute-side degradations of virtualization in either KVM or Linux containers are almost negligible, this would call for implementation of multiple network functions in one VM, contradicting the idea of fine-grained decomposition.

The potential of hardware accelerated I/O has been evaluated within WP5, and a handful of improvements have been either developed or studied. Our empirical evaluations confirm that implementations using DPDK accelerate I/O by an order of magnitude or more when compared to using virtio. Inter-VNF can further be accelerated when locality of VNFs is taken into account in the placement decisions met by the orchestrator. Shared memory communication between VNFs decreases latency and essentially removes any constraints in throughput.

Of the available hardware accelerations, DPDK-accelerated vSwitches apparently are the means of choice as long as no direct VM-to-VM communication is possible. This is the case for all node-external communication, but also when traffic leaving a VNF needs to be de-multiplexed or tagged depending on its header information. A particular use case for this could be service function chaining, when a service function header would need to be applied to the traffic by means of a (type-5) native function.

Finally, our empirical performance evaluation indicates that, in general, the latest options for hardware acceleration (like SR-IOV) have turned out to be surprisingly fragile and hardware-dependent. DPDK per se has undergone many changes during the last two years, and as if it needs further illustration of this statement, very recently an announcement was made that ivshmem, which has been used to achieve some of the results presented earlier, would soon be removed from the DPDK API.

References

[15a] *DPDK*. en. 2015. URL: http://dpdk.org/.

[15b] *IVSHMEM library*. en. 2015. URL: http://dpdk.org/doc/guides/prog%5C_guide/ivshmem%5C_lib.html.

[15c] *PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology*. en. 2015. URL: http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html.

[15d] *Poll Mode Driver for Emulated Virtio NIC*. en. 2015. URL: http://dpdk.org/doc/guides/nics/virtio.html.

[15e] *QEMU*. en. 2015. URL: http://wiki.qemu.org.

[15f] *Vhost Library*. en. 2015. URL: http://dpdk.org/doc/guides/prog_guide/vhost_lib.html.

[15g] *Vhost-User Feature for QEMU. Vhost-User Applied to Snabbswitch Ethernet Switch*. en. 2015. URL: http://www.virtualopensystems.com/en/solutions/guides/snabbswitch-qemu/.

[15h] *Virtio and vhost_net*. en. 2015. URL: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Virtualization_Tuning_and_Optimization_Guide/sect-Virtualization_Tuning_Optimization_Guide-Networking-Virtio_and_vhostnet.html.

[15i] *Xen*. en. 2015. URL: http://www.xen.org/.

[16a] *CarOS Release GitHub Repository*, travelping GmbH, 2016. URL: https://github.com/carosio/caros-release.

[16b] *extensible DataPath daemon - xDPd*. en. 2016. URL: http://github.com/bisdn/xdpd.

[D5.2] Hagen Woesner et al. *D5.2 Universal Node Interfaces and Software Architecture*. Tech. rep. UNIFY Project, Aug. 2014. URL: http://fp7-unify.eu/files/fp7-unify-eu-docs/Results/Deliverables/UNIFY-WP5-D5.2-Universal_node_interfaces_and_software_architecture.pdf.

[IBM15] IBM. *Virtio: An IO virtualization framework for Linux*. 2015. URL: http://www.ibm.com/developerworks/library/l-virtio/.

[Mac15] Cam Macdonell. *Nahanni, a shared memory interface for KVM*. en. 2015. URL: http://www.linux-kvm.org/images/e/e8/0.11.Nahanni-CamMacdonell.pdf.

[Pfa+09] Ben Pfaff, Justin Pettit, Teemu Koponen, Keith Amidon, Martin Casado, and Scott Shenker. "Extending networking into the virtualization layer". In: *Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*. New York City,NY, Oct. 2009.

[Rus08] Rusty Russell. "Virtio: Towards a De-facto Standard for Virtual I/O Devices". In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 95–103. ISSN: 0163-5980. DOI: 10.1145/1400097.1400108. URL: http://doi.acm.org/10.1145/1400097.1400108.

Annex A - Conformance Tests

Conformance tests are focused on the primitives defined within WP2: in particular, the UN interfaces reflect the SI-Or reference point. Therefore, the following primitives shall be tested:

- Resource Management Interface/Get node information and capabilities
- Resource Management Interface/Get available resources
- NF-NG Management Interface/Deploy NF-FG
- NF-NG Management Interface/Modify NF-FG
- NF-NG Management Interface/Delete NF-FG
- NF-NG Management Interface/Get NF-FG list
- NF-NG Management Interface/Get NF-FG data
- VNF Specs and Images Repository Interface/Fetch VNF specification by type
- VNF Specs and Images Repository Interface/Fetch VNF specification by ID
- VNF Specs and Images Repository Interface/Fetch VNF image

The tests are performed by using the proper APIs, provided by the node to retrieve the information or to manage the graphs. Each interface is compliant to the protocol whether it returns an output to the provided primitive; moreover, the output content and its correctness shall be verified as well, taking into account also the results of the corresponding functional tests for the URM. The complete description for all the conformance tests is reported below.

Test 1 - RMI: Node information and capabilities

Description	The interface is tested to be compliant with the primitive "Get node info and capabilities", used to retrieve the node information and main capabilities.
Expected Results	The interface is compliant if no error occurs when the primitive is sent. The RMI should report some node capabilities, as the total processing capacity, the total memory local disk capacity, CPU information, platform tag, ports list, flow space specification capabilities and the supported VNF types.
Configuration	The node is powered on and bootstrapped
Test Procedure	1. The API provided by the UN in order to get the node information and capabilities is sent to the RMI 2. The RMI conformance is verified if the interface replies 3. The retrieved information and capabilities are compared to the already known node information, provided by the developer or obtained through some other low level commands (e.g. Linux tools)

Test 2 - RMI: Node available resources

Description	The interface is tested to be compliant with the primitive "Get available resources", used to discover and report the available resources of the node to the upper layers. It should also be able to update the list of resources when a NF-FG is deployed.
Expected Results	The interface is compliant if no error occurs when the primitive is sent. The RMI should report the available resources, as the available processing capacity, the available memory, the available local disk capacity and the available capacity on ports.
Configuration	The node is powered on and bootstrapped
Test Procedure	1. The API provided by the UN in order to get the available resources is sent to the RMI 2. The RMI conformance is verified if the interface replies 3. Some information on the available resources may be retrieved with some low level commands on the host environment (e.g. top for Linux) and the two resources lists may be compared

Test 3 - NMI: NF-FG deployment

Description	• The interface is tested to be compliant with the primitive "Deploy NF-FG", used to deploy a NF-FG. The request should include the graph ID or the graph data as parameter to describe the required graph.
Expected Results	The interface is compliant if no error occurs when the primitive is sent. The NMI should reply to the request with the graph ID or with a result code.
Configuration	The node is powered on and bootstrapped
Test Procedure	1. The API provided by the UN to deploy a graph is sent to the NMI 2. The NMI conformance is verified if the interface replies 3. The NMI should reply with the graph ID or with a result code

Test 4 - NMI: NF-FG modification

Description	The interface is tested to be compliant with the primitive "Modify NF-FG", used to update an already deployed NF-FG. The graph to be updated is identified by an ID or its data are included in the request.
Expected Results	The interface is compliant if no error occurs when the primitive is sent. The NMI should reply to the request with the graph ID or with a result code.
Configuration	The node is powered on and bootstrapped A NF-FG is deployed on the node
Test Procedure	1. The API provided by the UN to update the existing graph is sent to the NMI 2. The NMI conformance is verified if the interface replies 3. The NMI should reply with the graph ID or with a result code

Test 5 - NMI: NF-FG deletion

Description	The interface is tested to be compliant with the primitive "Delete NF-FG", used to tear down a currently deployed NF-FG. The graph ID or its data are passed as parameter of the request.
Expected Results	The interface is conformant if no error occurs when the primitive is sent. The NMI should reply to the request with the graph ID or with a result code.
Configuration	The node is powered on and bootstrapped A NF-FG is deployed on the node
Test Procedure	1. The API provided by the UN to tear down the existing graph is sent to the NMI 2. The NMI conformance is verified if the interface replies 3. The NMI should reply with the graph ID or with a result code

Test 6 - NMI: NF-FGs list

Description	The interface is tested to be compliant with the primitive "Get NF-FG list", used to retrieve the list of all the currently deployed NF-FGs.
Expected Results	The interface is compliant if no error occurs when the primitive is sent. The NMI should reply to the request with the list of the IDs of all the graphs currently deployed on the node.
Configuration	The node is powered on and bootstrapped Some NF-FGs are deployed on the node
Test Procedure	1. The API provided by the UN to get the full list of the NF-FGs deployed on the node is sent to the NMI 2. The NMI conformance is verified if the interface replies 3. The NMI should reply with the list of the graphs' IDs currently deployed on the node

Test 7 - NMI: NF-FG data

Description	The interface is tested to be compliant with the primitive "Get NF-FG data", used to retrieve the data of a specific deployed NF-FG, identified by its ID.
Expected Results	The interface is compliant if no error occurs when the primitive is sent. The NMI should reply with the data related to the graph corresponding to the ID reported in the request
Configuration	The node is powered on and bootstrapped; A NF-FG is deployed on the node
Test Procedure	1. The API provided by the UN to get the data of a specific NF-FG deployed on the node is sent to the NMI 2. The NMI conformance is verified if the interface replies 3. The NMI should reply with the data related to the requested graph

Test 8 - VSIRI: VNF specifications by type

Description	The interface is tested to be compliant with the primitive "Fetch VNF specifications by type", used to retrieve the list of possible VNF specifications for a given NF abstract type or template.
Expected Results	The interface is compliant if no error occurs when the primitive is sent. The VSIRI should reply with a list of all the possible VNF specifications for the required NF type, as they are stored in the repository.
Configuration	The node is powered on and bootstrapped
Test Procedure	1. The API provided by the UN to fetch VNF specifications by type is sent to the VSIRI for a particular NF type 2. The VSIRI conformance is verified if the interface replies 3. The VSIRI should reply with a list of all the possible VNF specifications for the required NF type

Test 9 - VSIRI: VNF specifications by ID

Description	The interface is tested to be conformant to the primitive "Fetch VNF specifications by ID", used to retrieve the list of possible VNF specifications for a NF specified by its ID.
Expected Results	The interface is conformant if no error occurs when the primitive is sent. The VSIRI should reply with a list of all the possible VNF specifications for the required NF specified by the ID, as they are stored in the repository.
Configuration	The node is powered on and bootstrapped
Test Procedure	1. The API provided by the UN to fetch VNF specifications by ID is sent to the VSIRI for a specific NF's ID 2. The VSIRI conformance is verified if the interface replies 3. The VSIRI should reply with a list of all the possible VNF specifications for the NF corresponding to the required ID

Test 10 - VSIRI: VNF images

Description	The interface is tested to be conformant to the primitive "Fetch VNF image", used to retrieve the VM binary referenced in a VNF specification.
Expected Results	The interface is conformant if no error occurs when the primitive is sent. The VSIRI should reply with the VM image corresponding to the required VNF specification, as it is stored in the repository
Configuration	The node is powered on and bootstrapped
Test Procedure	1. The API provided by the UN to fetch VNF image is sent to the VSIRI for a specific VNF 2. The VSIRI conformance is verified if the interface replies 3. The VSIRI should reply with the binary corresponding to the requested VNF specification

Annex B - Functional Tests

Functional tests are based on the detailed list of functionalities of the UN components and interfaces, that has been provided in D5.3. Tables in B describe the tests configurations and procedures. Only the

functionalities that can be evaluated (or measured) from the outside have been considered; no test is defined for evaluating features related to the implementation and the internal functioning of the UN.

Test 11 - URM: Discover and report available resources

Description	Test focused on the URM functionality to discover the resources available on the node at the bootstrapping of the UN, in terms of number of available CPU/cores, available memory, aspects related to the virtualization engine and to the switching engine. Then, the URM has to propagate this information to the upper layer.
Expected Results	By the end of Phase III, the UN should be able to discover the full list of hardware resources, including the actual memory usage and the possible list of hardware coprocessors (if any), and to report it.
Configuration	The UN is powered on and bootstrapped
Test Procedure	1. The list of resources is displayed, using the proper API provided by the UN 2. The list of resources is displayed, using low level show commands provided by the Host environment (e.g. Linux tools as top, ps) 3. Compare the obtained lists of resources 4. Verify the report of the resource list to the upper layer, by capturing traffic on the RMI

Test 12 - URM: Update available resources

Description	Test focused on the UN functionality of updating the available resources: as a consequence of errors and/or updates of the node, or simply after the deployment, the modification or the removal of a NF-FG, the available resources change. When this happens, the URM has to notify the upper layer so it can react properly.
Expected Results	By the end of Phase III, the UN should be able to dynamically discover and report the full list of hardware resources. The UN should also be able to report the usage of any existing hardware coprocessor that is installed within the UN itself.
Configuration	The UN is powered on and bootstrapped A NF-FG is deployed
Test Procedure	1. The list of resources is displayed, using the proper API provided by the UN 2. The NF-FG is modified/updated/removed 3. The report of the updated list of resources to the upper layer is verified, by capturing traffic on the RMI

Test 13 - URM: Local scaling

Description	Test focused on the UN functionality of local scaling, which means the URM modulates the compute resources allocated to a VNF instance according to the number of parallel threads for VNF implementations.
Expected Results	By the end of Phase III, the URM should be able to interact with the VNF EE when changing the resource allocation, to effectively implement the change and to notify the VNF of the resource allocation change.
Configuration	The UN is powered on and bootstrapped A VNF that can scale by modifying the number of parallel threads is instantiated
Test Procedure	1. The traffic load is gradually increased 2. The scaling feature is verified: if the output traffic is not dropped if the resources in use decrease

Test 14 - URM: NF-FG lifecycle management

Description	Test focused on the UN functionality to manage the NF-FG lifecycle: the URM maps the VNFs to physical resources, relying on the information gathered during the resources discovery procedure.
Expected Results	By the end of Phase II, the URM should be able to manage the NF-FGs lifecycle phases, as the NF-FG deployment, update and removal requests from the upper level orchestrator.
Configuration	The UN is powered on and bootstrapped
Test Procedure	1. A NF-FG is deployed 2. The deployment of the NF-FG is verified, using the proper API provided by the UN 3. The NF-FG is modified 4. The updating of the NF-FG is verified, using the proper API 5. The NF-FG is removed 6. The deletion of the NF-FG is verified, using the proper API

Test 15 - URM/LO: Optimized placement

Description	Test focused on the LO functionality to perform optimized placement of the VNF threads into the UN topology, considering both the compute and networking resources required for the deployment of the NF-FG.
Expected Results	By the end of Phase III, the optimization should be based on the embedding algorithms developed in the WP3 orchestrator.
Configuration	The UN is powered on and bootstrapped
Test Procedure	1. The algorithm implementing the optimization is disabled 2. A NF-FG is deployed 3. The list of resources in use is retrieved, using the proper API 4. The NF-FG is deleted 5. The algorithm is enabled 6. The same NF-FG as before is deployed 7. The list of resources in use is displayed again and compared to the previous one, in order to verify the optimization

Test 16 - URM: Support to multiple VEE solutions

Description	Test focused on the UN functionality to support different VEEs, as KVM virtual machines, Docker containers or DPDK processes, running in the host environment, together with their interfaces.
Expected Results	By the end of Phase II, the prototype should provide support for Docker as VEE, for running VNFs as separate DPDK processes alongside the VSE and also for VEEs to traditional virtual machines (KVM).
Configuration	The UN is powered on and bootstrapped A NF-FG is deployed on the UN
Test Procedure	1. The URM is instructed to map the NF-FG to each one of the different kinds of VEEs at a time, to verify they are supported 2. Using available APIs, the VEE involved in the NF-FG deployment is verified to be one of the required types

Test 17 - URM: Support to multiple VSE solutions

Description	Test focused on the UN functionality to provide information to the URM about the physical ports as well as the supported capabilities of the VSE.
Expected Results	By the end of Phase II, some abstractions in the design of the interface should be provided and also a VSE solution should be identified among the alternative ones.
Configuration	The UN is powered on and bootstrapped
Test Procedure	1. Information about the physical ports are retrieved, using the proper API provided by the UN 2. Information about the supported VSE (e.g. OpenFlow version) should be retrieved as well, using the proper API

Test 18 - URM: Configuration of external connection to other nodes

Description	Test focused on the VSE Management functionality to configure the external connectivity of the NF-FGs, which is implemented in a special LSI-0 and has to be instantiated during the bootstrapping process of the UN.
Expected Results	By the end of Phase II, the LSI-0 should be properly configured by a local OpenFlow controller inside the UN, based on the information provided by the NF-FG. The implementation should be based on different soft switches (xDPd, OVS DPDK) and some traffic steering mechanisms should be supported to provide isolation in LSI-0 (e.g. tunneling, tagging, etc.).
Configuration	The node is powered on and bootstrapped
Test Procedure	1. The LSI-0 instantiation and configuration are verified, by using the proper API(s) provided by the UN itself

Test 19 - URM: Monitoring support and data report

Description	Test focused on the URM functionality to support the communication between local Observability Points (OPs) and Observability Points on other UNs or other nodes. Also the monitoring functionality of the VSE in the form of standard OpenFlow counters is tested.
Expected Results	By the end of Phase I, the OPs should be handled as normal VNFs. By the end of Phase III, the VSE should support the basic standard OpenFlow counters plus additional pluggable monitoring extensions and optimization.
Configuration	The node is powered on and bootstrapped
Test Procedure	1. A monitoring point is set, according to the specifications proper of the UN 2. The functionality of the monitoring point is verified, using the available APIs 3. In the case of OpenFlow counters, some procedures can be applied to verify their correct functioning; for example: Read the counter for the received/transmitted packets Send some more packets See whether the counter has been coherently updated

Test 20 - VSE: Dynamic deployment of LSI

Description	Test focused on the soft switch functionality to create an LSI, considering the two WP5 candidates soft switches xDPd and OVS DPDK.
Expected Results	By the end of Phase III, the dynamic deployment of LSI should be completed, also considering the integration of QoS in the slicing.
Configuration	The node is powered on and bootstrapped
Test Procedure	1. A NF-FG is deployed 2. The dynamic instantiation of the LSIs is verified, using the available show commands (e.g. xcli show lsis for xDPd) 3. The NF-FG is modified/deleted 4. The update/removal of the LSIs instantiated for the NF-FG is verified, using the same show commands as before 5. The same procedure (1-4) is applied considering all the soft switches supported by the UN

Test 21 - VSE: External traffic steering

Description	Test focused on the UN ability to dynamically create and remove virtual interfaces on the LSI-0 towards the deployed NF-FG resources, thus providing the external connectivity for the NF-FG.
Expected Results	By the end of Phase II, a general approach for NF-FG isolation should be proposed and different technologies and encapsulation mechanisms should be considered for their implementations. The main goal is to provide a set of basic resources to the upper layer to define the most adequate mechanisms to isolate the NF-FGs on a network-wide view. The optimization of the steering mechanism is expected by the end of Phase III.
Configuration	The node is powered on and bootstrapped
Test Procedure	1. The traffic is captured on the OpenFlow interface, in order to get the isolation mechanism information 2. When a NF-FG is deployed, the packets are captured in order to verify the correct external steering

Annex C - Performance Tests

The configurations of interest for the Performance tests have been conceived as combinations of the number of NFs, NF types, number of traffic flows and type of packets. All these single elements, which are implied in the test procedures, are detailed below.

Test 22 - Throughput

Description	Test focused on throughput and latency performances of the DUT.
Expected Results	The goal is to characterize the UN from the throughput and latency viewpoint, by considering variable input bit rate and packet sizes, applied to different test scenarios. The upper limit of the node, in terms of bits per second and packets per second, that can flow through the node without losses, should be determined. Two charts describing the trend of the packet loss and the trend of the latency introduced by the node versus the input throughput should be provided, for each test configuration.
Configuration	The Ethernet interfaces of the node are connected to a traffic generator/analyzer One or more NFs are instantiated on the node, according to the scalability scenarios The node is configured to switch packets coming in the subscriber interface to the pass-through NF and then to the internet interface The node is configured to switch packets coming in the internet interface to the pass-through NF and then to the subscriber interface
Scalability Scenarios	Type of Employed NF Pass-through Test Procedure 1. IP traffic is sent to the DUT from the traffic generator, defining a single flow with different packet sizes (64, 128, 256, 500, 1000, 1500 byte) 2. Using monitoring features provided by the analyzer, statistics related to received/dropped packets and latency are collected 3. Throughput and latency performances are considered for: Increasing bit rate Different scalability scenarios 4. The procedure (1 – 3) is applied again considering a multi-flow traffic 5. The procedure (1 – 4) is applied again considering a realistic flow traffic

Test 23 - CPU

Description	Test focused on CPU performances of the DUT.
Expected Results	The goal is to characterize the UN from the CPU resource consumption viewpoint, as it employs a CPU intensive NF type to process the incoming packets. The upper limit of the node, in terms of bits per second and packets per second that can flow through the node without losses, should be determined. Two charts describing the trend of the packet loss and the trend of the latency introduced by the node versus the input throughput should be provided, for each test configuration.
Configuration	The Ethernet interfaces of the node are connected to a traffic generator/analyzer One or more NFs are instantiated on the node, according to the scalability scenarios The node is configured to switch packets coming in the subscriber interface to the ad-hoc NF and then to the internet interface The node is configured to switch packets coming in the internet interface to the ad-hoc NF and then to the subscriber interface
Scalability Scenarios	Type of Employed NF Ad-hoc function CPU Intensive: "stress –cpu N" Test Procedure 1. IP traffic is sent to the DUT from the traffic generator, defining a single flow with different packet sizes (64, 128, 256, 500, 1000, 1500 byte) 2. Using monitoring features provided by the analyzer, statistics related to received/dropped packets and latency are collected 3. Throughput and latency performances are considered for: Increasing CPU load by increasing the bit rate Different scalability scenarios 4. The procedure (1 – 3) is applied again considering a multi-flow traffic 5. The procedure (1 – 4) is applied again considering a realistic flow traffic

Test 24 - Memory

Description	Test focused on Memory performances of the DUT.
Expected Results	The goal is to characterize the UN from the Memory resource consumption viewpoint, as it employs a memory intensive NF type to process the incoming packets. The upper limit of the node, in terms of bits per second and packets per second that can flow through the node without losses, should be determined. Two charts describing the trend of the packet loss and of the latency introduced by the node versus the input throughput should be provided, for each test configuration.
Configuration	The Ethernet interfaces of the node are connected to a traffic generator/analyzer One or more NFs are instantiated on the node, according to the scalability scenarios The node is configured to switch packets coming in the subscriber interface to the ad-hoc NF and then to the internet interface The node is configured to switch packets coming in the internet interface to the ad-hoc NF and then to the subscriber interface
Scalability Scenarios	Type of Employed NF Ad-hoc function memory intensive: "stress –vm N –vm-bytes B" Test Procedure 1. IP traffic is sent to the DUT from the traffic generator, defining a single flow with different packet sizes (64, 128, 256, 500, 1000, 1500 byte) 2. Using monitoring features provided by the analyzer, statistics related to received/dropped packets and latency are collected 3. Throughput and latency performances are considered for: Increasing memory load by increasing the bit rate Different scalability scenarios 4. The procedure (1 – 3) is applied again considering a multi-flow traffic 5. The procedure (1 – 4) is applied again considering a realistic flow traffic

Test 25 - Switching

Description	Test focused on Switching performances of the DUT.
Expected Results	The goal is to characterize the UN from its Switching capability viewpoint. The setup rate, as the number of setup flow rules per second, should be determined for different input bit rate. Moreover, the latency in the rules implementation should be measured as well.
Configuration	The Ethernet interfaces of the node are connected to a traffic generator/analyzer The node is configured to switch packets coming in the subscriber interface to the internet interface The node is configured to switch packets coming in the internet interface to the subscriber interface
Scalability Scenario	Test Procedure 1. IP traffic is sent to the DUT from the traffic generator, defining a single flow with different packet sizes (64, 128, 256, 500, 1000, 1500 byte) 2. Using monitoring features provided by the analyzer, statistics related to number of updated flow rules and latency are collected 3. Setup Rate and latency performances are considered for: Maximum number of supported switching rules Equally spaced switching rule setup requests (taking into account also a decreasing time interval among the requests) Bursts of switching rule setup requests 4. The procedure (1 – 3) is applied again considering a multi-flow traffic 5. The procedure (1 – 4) is applied again considering a realistic flow traffic

Test 26 - Control Interfaces

Description	Test focused on Control Interfaces performances of the DUT.
Expected Results	The goal is to characterize the UN from its Control Interfaces viewpoint. The expected output are two charts that describe the setup rate (number of flow rules updated per second) and the trend of latency introduced by the node versus the input NF-FG requests respectively, for each test configuration.
Configuration	The Ethernet interfaces of the node are connected to a traffic generator/analyzer One NF is instantiated on the node, according to the scalability scenario The node is configured to switch packets coming in the subscriber interface to the pass-through NF and then to the internet interface The node is configured to switch packets coming in the internet interface to the pass-through NF and then to the subscriber interface
Scalability Scenario	Type of Employed NF Pass-through Test Procedure 1. IP traffic is sent to the DUT from the traffic generator, defining a single flow with different packet sizes (64, 128, 256, 500, 1000, 1500 byte) 2. Setup Rate and latency performances are considered for: Maximum number of supported NF-FGs Equally spaced NF-FG requests (taking into account also a decreasing time interval among the requests) Bursts of NF-FG requests 3. The procedure (1 – 2) is applied again considering a multi-flow traffic 4. The procedure (1 – 3) is applied again considering a realistic flow traffic