unifying cloud
and carrier networks

D5.2 Universal Node Interfaces and Software Architecture

Dissemination level	PU
Version	1.0 Reviewed
Due date	31.03.2014
Version date	26.08.2014

This project is co-funded
by the European Union

Document information

Authors

Editor: INTEL

Contributors:

BISDN - Hagen Woesner

EHU - Jokin Garay, Jon Matias

ETH - Gergely Pongracz, Robert Szabo

INTEL - David Verbeiren

OTE - George Agapiou

POLITO – Fulvio Risso, Mario Baldi, Alex Palesandro, Ivano Cerrato

Reviewers: Balazs Sonkoly (BME), Holger Winkelmann (TP)

Coordinator

Dr. András Császár

Ericsson Magyarország Kommunikációs Rendszerek Kft. (ETH) AB

Email: andras.csaszar@ericsson.com

Project funding

7th Framework Programme

FP7-ICT-2013-11

Collaborative project

Grant Agreement No. 619609

Legal Disclaimer

Revision and history chart

Version	Date	Comment
0.1	30.06.2014	Initial version based on MS5.1
0.2	14.07.2014	Updated section on flow space specification, closed most items for review
0.3	25.07.2014	Added application control interfaces considerations, some tidying up. Version released for review.
0.4	25.08.2014	Addressed comments from reviewers.
1.0	26.08.2014	Final version

Table of contents

Executive summary

This document establishes the software architecture of the Universal Node and defines its external interfaces.

An overview of existing related technologies and interfaces is first given (Section 2). It covers compute platform virtualization solutions, the cloud computing platform OpenStack, high-performance packet processing as well as control and management interfaces.

Next, an overview of the UNIFY project is provided with its layers and reference points and a mapping is established with the Universal Node (UN) in order to locate it in the programmability framework being defined in WP3. The concept of Network Function Forwarding Graph (NF-FG) is introduced and some of its aspects that are critical for the architecture of the UN are detailed.

Based on the positioning of the UN within the UNIFY architecture, a functional view of the UN is derived. This represents an evolution from the high-level UN architecture that was introduced in D5.1 in that the interface with the upper orchestration layer has been updated to have the NF-FG as core element.

The proposed UN architecture is then described introducing more concrete architectural blocks and how they interact with each other. One key element of the UN architecture is the Unified Resource Manager where compute and networking resources are managed by a common entity, reflecting the fact that computing and networking resources are not independent on the UN since packet switching and packet processing, even when happening outside of Virtual Network Functions, may require dedicated compute resources. Additionally, this common resource manager also plays the role of a local orchestrator that is responsible for optimized assignment of the node resources to the deployed NF-FGs and their Virtual Network Functions (VNF), taking into account platform topology aspects that can significantly affect performance of the platform and the Network Functions.

Finally the various interfaces of the UN are described with the semantics of their primitives or concrete interface proposals. This covers the interfaces with upper orchestration layer for deployment and management of NF-FGs and for resource discovery and resource usage reporting. It also includes an interface to fetch Virtual Network Function specifications and related binary images from a central repository.

With architecture in place and external interfaces defined, the UN prototyping work can progress in support of the benchmarking and integrated prototype efforts.

1 Introduction

In D5.1, the requirements on the Universal Node data plane were detailed and an initial high-level architecture was introduced. Since then, the work progressed within the overall UNIFY project with the overarching UNIFY architecture being defined (WP2) and the programmability framework being articulated (WP3).

This document describes how the Universal Node fits within the UNIFY architecture, provides an updated and more detailed architecture of the Universal Node and describes its external interfaces. It reflects the progress of these activities corresponding to project milestone M5.1 and provides a sound basis to support the UN prototyping activities.

It should be noted that some aspects are more detailed than others and that some topics remain open for further investigation as noted in the conclusion section of this document.

2 State of the Art and Related Work

2.1 OpenStack

OpenStack [1] is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacentre. Its components can be controlled through extensive REST based APIs and a dashboard gives administrators control while empowering their users to provision resources through a web interface.

OpenStack's great popularity is primarily due to the large amount of heterogeneous partners backing the project. According to this large support, the very wide range of different competence, inherited by different company cultures, is exploited in all different components development, resulting in a platform that pretends to cover every aspect of a cloud platform. The extremely modular architecture allows the framework to be scalable and flexibly adaptable in really different situations. Openstack provides components to easily handle different resource types (storage, compute, network) available in a data-centre environment. Nevertheless, its overall development is oriented to an extensible design, simplifying the writing of external plugins to implement further functionalities by third-party actors. This is concretely achieved in each component design, but also in the low involvement required to register and integrate a complete third-party component in the pool.

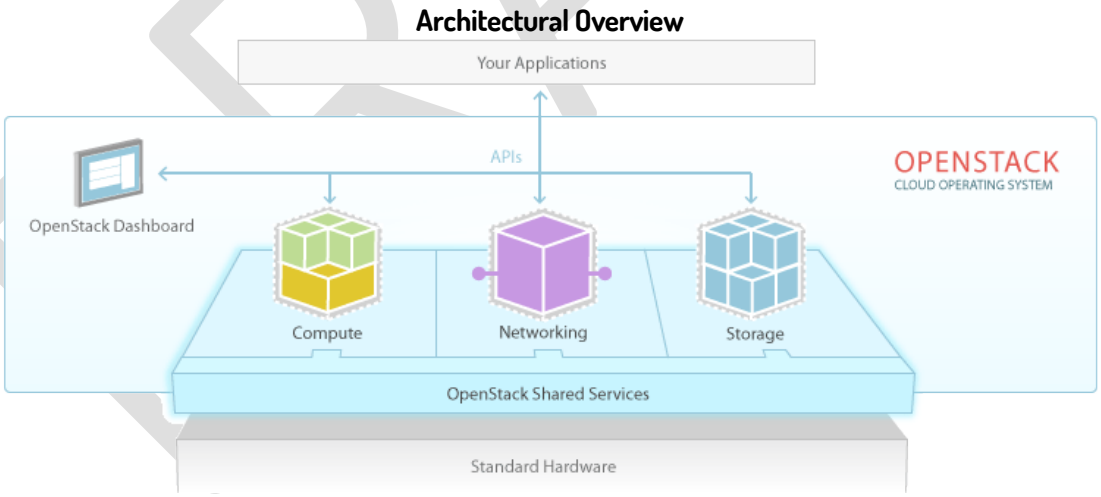

Figure 2.1 - OpenStack

OpenStack consists of the following main components (see Figure 2.1):

- Compute (codenamed "Nova") provides virtual servers upon demand. Rackspace and HP provide commercial compute services built on Nova and it is used internally at companies like Mercado Libre and NASA (where it originated).
- Network (codenamed "Neutron") provides "network connectivity as a service" between interface devices managed by other OpenStack services (most likely Nova). The service works by allowing users to create their own networks and then attach interfaces to them. OpenStack Network has a pluggable architecture to support many popular networking vendors and technologies.
- Image (codenamed "Glance") provides a catalogue and repository for virtual disk images. These disk images are most commonly used in OpenStack Compute. While this service is technically optional, any cloud of size will require it.
- Object Store (codenamed "Swift") provides object storage. It allows storing or retrieving files (but not mount directories like a fileserver). Several companies provide commercial storage services based on Swift. These include KT (formerly Korea Telecom), Rackspace (from which Swift originated) and Internap. Swift can be used as backend storage for Glance and also to store application data.
- Dashboard (codenamed "Horizon") provides a modular web-based user interface for all the OpenStack services. With this web GUI, most operations like launching an instance, assigning IP addresses and setting access controls can be performed.
- Identity (codenamed "Keystone") provides authentication and authorization for all the OpenStack services. It also provides support for other OpenStack components by offering a catalogue of services where the various components (Glance, Nova, Swift, Keystone itself...) and their respective API endpoints are listed and can be queried.
- Block Storage (codenamed "Cinder") provides persistent block storage to guest VMs.
- Orchestration (codenamed “Heat”) provides an orchestration service for all the OpenStack resources. It requires as input files a list of resources (e.g. Neutron networks, Nova virtual servers) with their relationships and parameters. As output, it will produce the correct sequence of API calls to the corresponding platform components (e.g. Neutron, Nova, Cinder) in order to deploy the desired resources.
- Ceilometer provides an infrastructure to collect measurements from other OpenStack components. Its primary targets are monitoring and metering, with an initial focus on customer billing purposes, but it should be able to share the collected data with any type of client component.

The interactions between the components are depicted in the following figure:

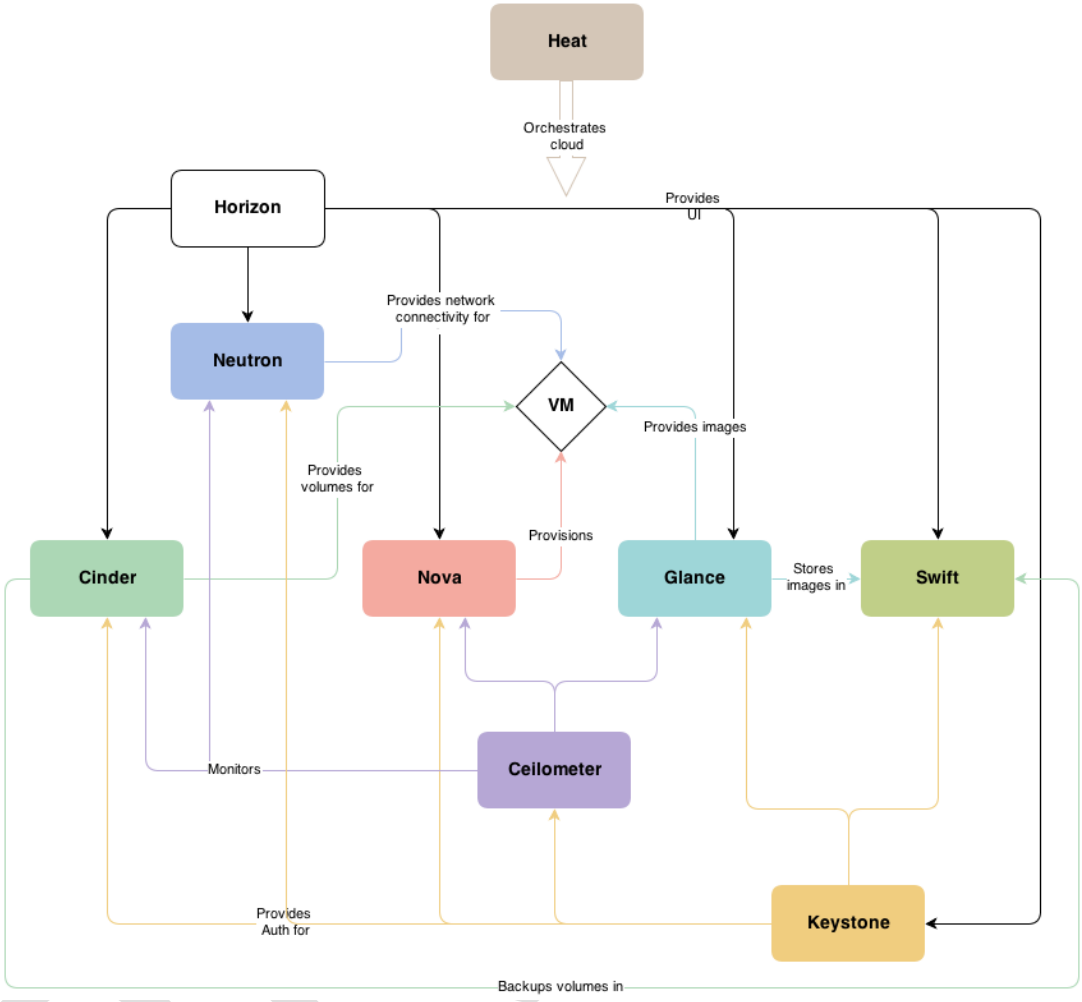

Figure 2.2 - OpenStack components

Each of the components provides an API to access it, which can be used as a REST API with the http protocol, or as CLI. The OpenStack APIs are documented at http://api.openstack.org.

The compute (Nova) and the networking (Neutron) components are the ones that are the most related to the UNIFY data plane.

The Nova API calls are used e.g. to launch a VM. Such a Nova API call is shown in Figure 2.3 below with the arrow #2. The VM itself will be started and managed directly by the Hypervisor. Various types of hypervisors are supported, the widest function set is available when using XenServer/XCP, KVM on x86 or QEMU on x86. The communication interface between the Nova Compute module and the Hypervisor is primarily libvirt.

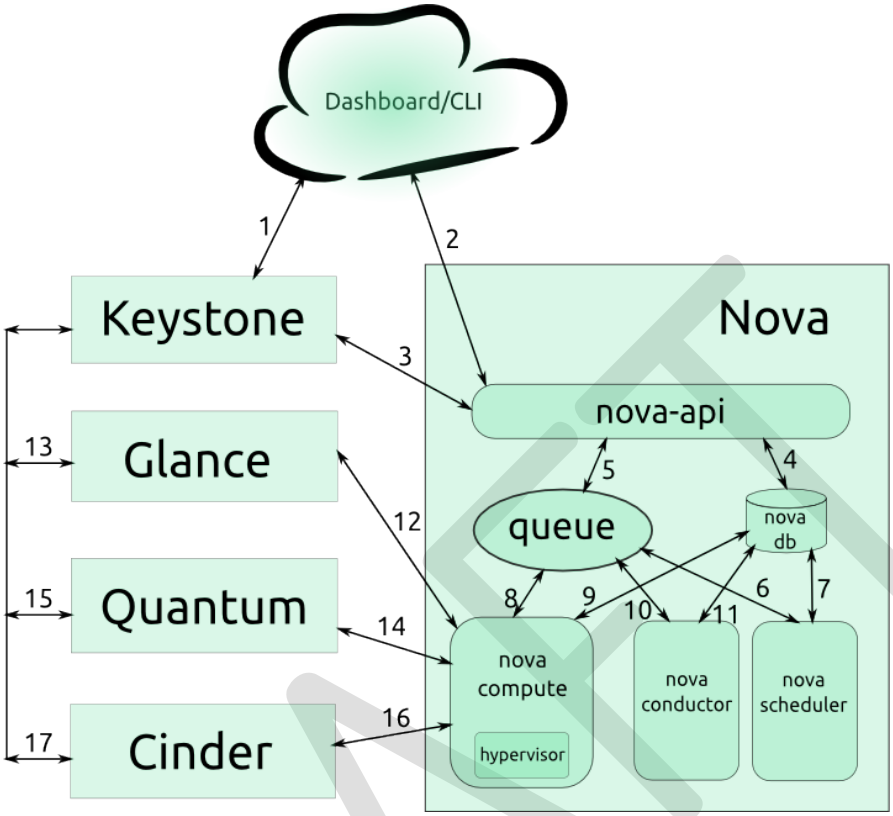

Figure 2.3 - Internals of Nova, steps to launch a VM

Note: "Quantum" is an earlier name of "Neutron"

2.1.1 Neutron

Neutron is the OpenStack component that provides flexible and on-demand access to network resources, establishing networking services between different devices (e.g. vNIC) created and managed by other OpenStack services.

Neutron exports to the user three classes of network abstractions: networks, subnets and ports. Those abstractions reflect the behaviour of their physical counterparts. At first, a network is a virtual per-tenant layer-2 broadcast domain. If not explicitly shared, each network is accessible only by its tenant. Each network has one or more IP addressing blocks called subnets. A subnet could have several network parameters associated with it (e.g. gateway, DNS server). A port represents a virtual switch port to which VMs attach their interfaces. Each port defines the MAC address and the IP address to be assigned to the VMs plugged interfaces. When an IP address, taken from a subnet pool, is selected, it implies that the port/interface is part of the selected subnet. When IP addresses are associated to a port, this also implies the port is associated with a subnet, as the IP address was taken from the allocation pool for a specific subnet. Moreover, standard APIs were extended in order to simplify user deployment, filling the conceptual gap between physical networks and

virtually defined ones. Therefore, in the extended REST API, several further abstractions are added. In particular, L3-network router extension is used to route packets between different L2 networks.

Neutron enforces the user-defined network topology leveraging different L2 technologies (GRE, VXLAN, VLAN) and supporting several backend L2-switch technologies (Open vSwitch, Arista, BigSwitch). Neutron L2 drivers usually consist of a component residing on the neutron server and an agent, replicated on each compute node, which enforces the required actions on the virtual switch present on the node. In order to provide a unified interface to different L2 virtual switch technology, starting from OpenStack Havana release, Neutron provides the Modular Layer 2 (ML2) component that abstracts L2 events (e.g. create/update/delete network), standardizing the interface for third-party technologies. In the default configuration, Neutron leverages the Open vSwitch (OVS) technology, requiring the creation of different instances in order to meet different network requirements. ML2 mechanism drivers are normally required to define their own hooks, which will be executed in presence of a certain event on Neutron-defined objects (network, subnet, port).

Neutron also supports connectivity to external networks, regulating access to the Internet. This can be provided by means of a Neutron router acting as default gateway, allowing VMs on the network to reach external hosts or, alternatively, specific VMs could be provided with a public address on the external network. Moreover, since internal addresses are not directly accessible by the Internet and external addresses are a scarce resource, Neutron provides the possibility to temporarily associate a "floating" IP address to specific VMs instances, allowing the traffic to be temporarily redirected to the correct VM. This association can be done dynamically after the VM provisioning.

Furthermore, Neutron supports per-tenant "security groups" definition in order to regulate ingress/egress traffic with firewall-like rules. Like other Neutron abstractions, these rules are not strictly related to a particular implementation but their instantiation will depend on the underlying plugin used. The operation of security groups is simple: traffic matched with security group rules is allowed and, without a match, packets are dropped. For instance, leveraging OVS plugin, iptables defined rules are used to implement security group policies.

In summary, Neutron available APIs include the following primitives:

- Create/Update/Delete Networks
- Create/Update/Delete Subnets
- Create/Update/Set default gateway/Delete Routers
- Create/Attach/Detach/Delete ports
- Create/Set-policy/Delete Security Groups

2.2 Platform Virtualization

2.2.1 QEMU and KVM

QEMU [2] [3] is an open source virtualization solution that supports both machine emulation and hypervisor operation. When the guest and host machines are of different processor architectures, dynamic binary translation allows QEMU to run the guest with reasonable performance. For compatible guest and host processors, 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. This is implemented by KVM, a Linux kernel module that is now developed as part of the QEMU project.

KVM takes advantage of hardware assisted virtualization features present in modern x86 processors to further reduce the overhead incurred by running applications in such virtual machines thereby providing close to native performance in many cases. These features attempt to reduce the frequency or the cost of the processor control transitions from the application running in the guest to the hypervisor ("VM Exits") that can happen for various reasons. Here are some examples of such features found on Intel Xeon processors:

- Specific VM control structures (VMCS) with storage for processor state for virtual machines decrease the latency of transitions.
- Extended Page Tables (EPT) 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 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 of course applications in the guest systems. This is in contrast with solutions referred to as para-virtualization, 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 however blurred the lines between these different approaches:

- QEMU/KVM now also uses para-virtualization when available, more recently in the form of the virtio infrastructure in the Linux kernel.
- The presence of an IOMMU on modern platforms allows giving direct access to virtual machines to designated devices, or portions of devices (PCI passthrough).
- Some devices also present 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) technology by which a

device presents multiple virtual functions (VF) to be used by virtual machines. Each VF has dedicated resources in the device (for example queues) but eventually makes use of the shared physical function (PF).

2.2.2 Xen

Xen [4] is an hypervisor that is originally based on the para-virtualization approach requiring modified versions of the guest operating systems that, running de-privileged, make so-called hypercalls (calls to the hypervisor) instead of issuing privileged instructions.

However, as discussed previously, hardware assisted virtualization features present in modern processor architectures also make it possible to let those guest operating systems execute the privileged instructions, once the appropriate configuration has been done by the hypervisor. Xen has of course embraced this approach as well which allows it to run unmodified operating systems in guests ("full virtualization").

In Xen terminology, virtual machines are called "domains" and the Xen architecture makes use of a privileged domain, called "domain 0", to manage the hypervisor and the devices. The hypervisor itself runs on the bare hardware ("Type 1" hypervisor) and launches the "domain 0". In contrast, QEMU runs on top of a host operating system, even though the usage of the KVM kernel module in the host OS makes both architectures more similar.

2.2.3 LXC

Linux containers (LXC) is an operating system-level virtualization technology that allows deploying and executing lightweight virtual machines (i.e., containers) on a Linux-based host system.

OS-level virtualization is a software virtualization method where different and isolated user-space instances run on the same kernel of the host system; with the exception of the commercial (and closed source) version of OpenVZ[1], also named Parallels Virtuozzo, OS-level virtualization technologies are integrated in the Linux kernel. In other words, different LXC containers appear such as distinct running instances of the "same" (in fact, some minimal differences are possible) operating system, with distinct bash shells, etc.

Linux containers are virtual environments that look like a classic virtual machine for the processes running inside them, but they are just processes for the kernel of the host system. Containers differ from classic virtual machines in that they do not emulate hardware and, because they share the same kernel as the host system, they cannot run different operating systems, for example Microsoft Windows. Note that different Linux distributions are fine as long as they share the same kernel as the host. Figure 2.4 below should clarify

[1] Another OS-level virtualization technology; most of its features are now integrated in the vanilla linux kernel and so included in LXC

the concept. Note that there is no hypervisor when using LXC: the hypervisor is the host kernel itself, with its integrated virtualization features.

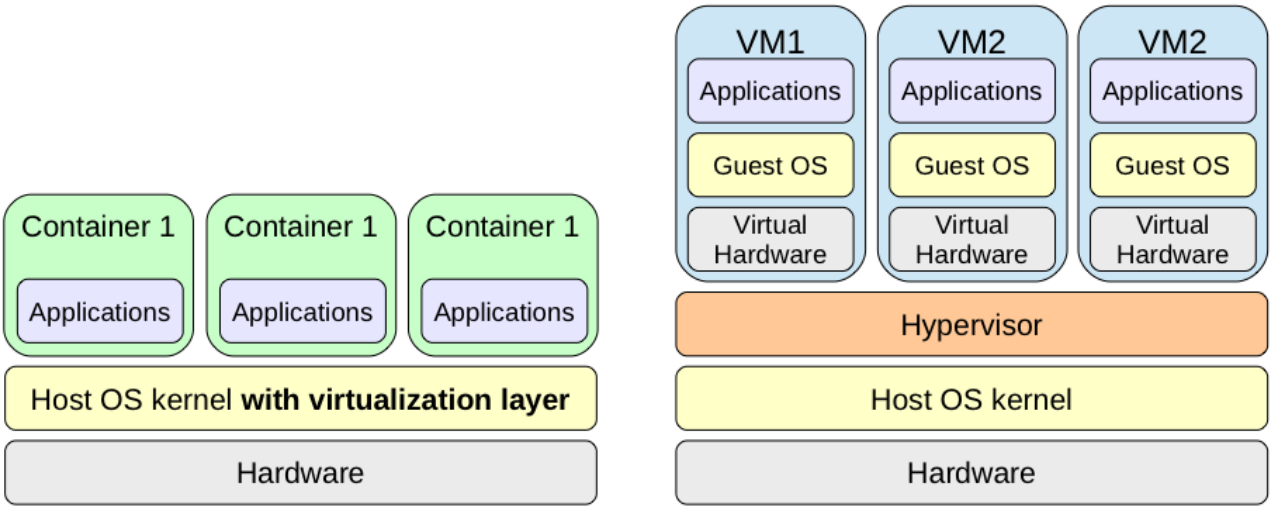

Figure 2.4 - Containers vs Hypervisors

Containers can practically achieve native execution performance because fundamentally they are based on isolation instead of classical virtualization: there is practically no overhead while running a process inside a container and the memory consumption is minimal (less than 1 MB); therefore containers are faster (to boot, freeze and dispose), lighter, denser and more scalable than classic virtual machines.

With containers, it is possible to isolate different processes or groups of processes and to allocate different resources to them without the complexity of full virtualization. Resources consist of CPU quotas, memory, I/O bandwidth, network isolation, disk quotas and file system isolation.

LXC is a collection of vanilla Linux kernel features (cgroups, namespaces, chroots, kernel capabilities etc.) that can be used to isolate processes in different ways and a user-space tool to use all of these features together to create full-fledged containers.

LXC is not a monolithic system, it is possible to configure containers with only the needed features: the only things that will be isolated are those specified in the configuration file for any container (for example it is possible to have only PID namespace isolation or only network isolation etc.); different resources can be assigned to different containers and different level of isolation between different containers (and the host) can be established.

Unfortunately the isolation is not 100% effective: tests prove that if a container demands more resources than it was assigned, it will impact other containers [5].

Moreover, LXC lacks the checkpointing and other features needed for the support of live migration, so other software tools such as CRIU [6] should to be used in order to (partially) achieve those functionalities.

Finally LXC support in OpenStack does not include all LXC features and, even more worrying, the only driver for using LXC with OpenStack will be deprecated by the Icehouse release: the libvirt driver for LXC as of April 2014 has no future plans for development [7]. This seems to suggest that LXC may not represent a good choice if the full support within OpenStack is a mandatory requirement.

2.2.4 Docker

Docker is an open source software that easily allows to create lightweight, portable, self-sufficient containers from any application. It does that by deploying applications inside Linux containers.

A container comprises an application and its dependencies (for example binaries and libraries). Containers serve to isolate processes running in user space on the host's operating system.

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 re-used to create more containers. No templating or manual configuration is required.

With traditional Virtual Machines, each application, each copy of an application, and each slight modification of an application requires creating an entirely new VM (to maintain all the versions). In contrast, when using Docker, running several copies of the same application on a host does not even require copying the shared binaries and, if the application is modified, only the differences need to be copied. This makes it very efficient to store and run containers and it also makes updating applications easier.

Docker adds to plain Linux containers the ability to build once a container with an application inside and to run and migrate it across different Linux systems (architecture must match), without the need to worry about configurations and dependencies typical of each platform.

Containers can either be created manually or, if a source code repository contains a DockerFile, automatically.

Docker offers resource management and isolation the same way as Linux containers do and, on top of that, it makes it easy to manage the versioning of applications and sharing of ready to run containers (in private or public clouds).

Docker provides its own container support (libcontainer, built on namespaces and cgroup OS features) but also supports other container drivers like LXC or libvirt. What Docker adds to that is the way it assembles a filesystem for the container by combining the host filesystem together with additional layers to "emulate" the complete environment. This is done using UnionFS (aufs = another union fs). The "emulated" environments are

isolated by the execution driver. The Docker image only contains the difference to a standard docker-exported environment.

Until version 0.6 Docker used standard LXC for containers, then it started using its own container toolkit named libcontainer. Version 1.0 of Docker will provide a stable core and stable API so it should be easier to extend Docker to support other container formats (e.g. BSD Jails, Solaris Zones, OpenVZ) and other features like Software-Defined Networking [8].

The main Docker binary supports 3 modes of operation: 1) launched as a daemon, it manages the containers on the Linux host, 2) as a CLI client which talks to the daemon over a REST API to control container creation and execution and 3) as a client of Docker Repositories that lets the user explore and fetch available images or share the images she creates.

OpenStack support is actively developed by Docker Inc. and the community. The initial idea was to develop a driver for Nova but there were disadvantages to this approach like the difficulty to expose some of the more useful Docker functionality (for example linking containers) or the fact that some VM-specific functionalities expected by the standard API extensions didn't make sense in a container context.

The new approach to orchestrating Docker in OpenStack is via Heat [9]. Using the Heat plugin, users may deploy and manage Docker containers on top of traditional OpenStack deployments, making it compatible with existing OpenStack clouds. The Docker plugin for Heat has been accepted into OpenStack and will be in the Icehouse release.

Despite the Heat plugin is now the official approach to support OpenStack, the Nova driver has seen continued development [10]: it now provides compatibility with Open vSwitch and the door has been opened to supporting other Neutron drivers.

2.3 Libvirt

Libvirt is a toolkit to interact with the virtualization capabilities of recent versions of Linux (and other OSes). As indicated, OpenStack Compute does not provide any virtualization capabilities by itself; instead, it uses other APIs to perform this task. These APIs are: (1) LibVirt API, (2) Xen API, (3) vSphere API, and (4) Windows Management Interface. The LibVirt API [11] is the most common used API in OpenStack because it is easy to use compared to other APIs and supports most of the virtualization technologies of the Linux platform. It is a toolkit (API, daemon, and command line utilities) implemented by Red Hat to install, run, and interact with the Linux virtualization technologies. The wide range of Linux virtualization technologies supported by libVirt includes KVM (Kernel virtual machine), Xen, UserMode Linux, QEMU (Quick EMUlator), and LXC (Linux Container).

The following are the advantages of the LibVirt API:

1. It allows setup of VMs (start/stop).
2. It provides efficient resource provisioning of VMs.
3. It can control VMs remotely though a secure interface.
4. It provides isolation from the frequent changes expected at the lower level of the virtualization framework.
5. It provides modules to enumerate, monitor and use the resources available on the managed node, including CPUs, memory, storage, networking, and NUMA partitions.
6. It provides domain specific monitoring such as the state monitoring APIs needed to implement management policies.

There are some shortcomings of the LibVirt API such as, (1) the Libvirt API often requires a caller to know the local path to a particular storage element, and (2) it does not support features like load balancing between VMs. In addition, not all the virtualization technologies are supported by the libVirt API. The unsupported technologies are Citrix Xen Server (CXP), Xen Cloud Platform (XCP), VMware ESX, and Microsoft Hyper-V. These technologies can be supported by using a different virtualization API in the OpenStack Compute. For example, the Xen API can support CXP and XCP, the vSphere API can support VMware ESX, and the Windows Management Interface API can support Microsoft Hyper-V.

2.4 Data Plane Processing on x86

Applications operating on the network traffic should be carefully engineered in order to be able to operate at line rate without any drop of the packet in transit. For instance, a packet processing application operating on a 10Gbps Ethernet network, must be able to process more than 14 Mpps of 64 bytes packets.

To achieve such high performance, current solutions for packet processing allow applications (e.g., network functions) to perform raw packet I/O, i.e., to directly access the network interface cards (NICs) without requiring the intervention of the operating system, which would introduce a non-negligible overhead. High performance is also achieved through the so called "zero-copy" mechanism, which allows an application to receive, process, and finally send a packet without any copy of the packet itself.

Other techniques used to achieve high-speed packet processing include: *(i)* interrupt mitigation, which limits the number of interrupt generated (and hence the number of context switches) by sending a single interrupt for a bucket of packets ready to be processed; *(ii)* data prefetching; *(iii)* batch processing; *(iv)* polling operating mode, in which the application continuously tries to receive packets from the NIC. It is worth noting that

polling, although improving performance (especially from a latency point of view), has the side effect of using an entire CPU core.

The remainder of this section briefly presents the three technologies that are mainly used today to implement high performance packet processing applications on general purpose compute platforms, namely: Intel DPDK, netmap, and pf_ring DNA.

2.4.1 Intel Data Plane Development Kit (DPDK)

Intel DPDK [12] is a framework optimized for the development of data plane applications. It provides to programmers many functions designed to create efficient network functions on x86 platforms, particularly with respect to those that involve high-speed packet processing. This section provides an overall description of the DPDK features and components (shown in Figure 2.5).

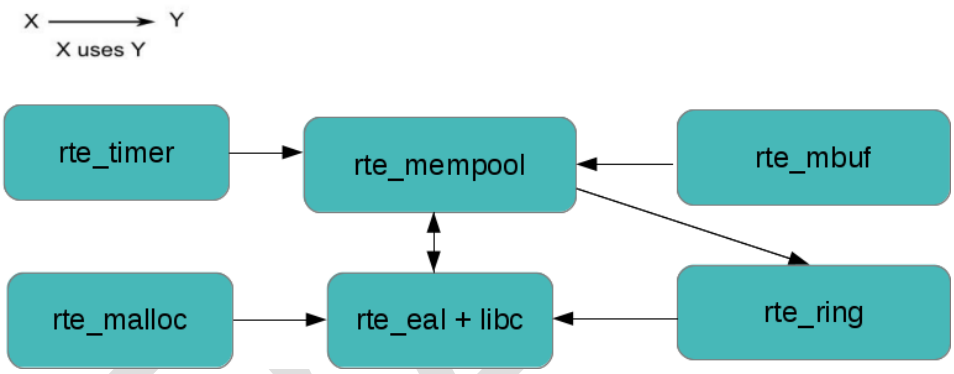

Figure 2.5 - Main DPDK components

2.4.1.1 Applications execution models

DPDK supports both the run to completion and the pipeline model. The former assumes that each CPU core executes the same application instance, so that several packets can be processed in parallel in the same way. Instead, in the latter each CPU core is responsible of a different stage of the packet processing; hence, in this case there may be several packets processed in parallel, each one in a different stage of the pipeline.

Furthermore, DPDK assumes that the processes being executed (e.g., network functions) operate in polling mode, in order to be more efficient and reduce the time spent by a packet travelling in the server. This requires each process to occupy one full core of the CPU (in fact, DPDK processes are usually pinned to a specific CPU core for optimization reasons), hence the number of processes running concurrently is limited by the CPU architecture and are usually on the order of a few dozens. Although this scheduling model is not mandatory, DPDK primitives are definitely more appropriate when applications are designed in that way; for example DPDK does not offer any interrupt-like mechanism to notify the application of the arrival of a packet on the NIC.

Hence, the programmer may need to adapt (or re-implement) some of the DPDK functions to support also processes operating in an interrupt-like fashion.

2.4.1.2 EAL, lcores, and multi-processes applications

The Environment Abstraction Layer (EAL) is one of the main components of DPDK. It is in charge of the initialization of the resources needed by DPDK applications and provides applications with an API to use these resources.

EAL defines the concept of logical core (lcore), which is an application instance running on a CPU core. In particular, a DPDK process consists of the master lcore, which can create other lcores in the initialization phase of the DPDK process, through the API offered by the EAL.

In addition, EAL supports multi-processes applications, which consist of a primary process and a number of secondary processes. In this case, shared memories and other DPDK resources can be only initialized by the primary process, while the secondary processes can just use pre-initialized resources. Moreover, all the processes that are part of the same DPDK application share all the resources allocated by the primary process through the EAL. Independent primary processes that do not share resources are also supported.

2.4.1.3 Memory management

DPDK offers two ways to manage the memory: the rte_malloc and the mempool.

The rte_malloc is a library that looks similar to the standard libc malloc. In fact, it can be used to allocate objects (of any size) during the execution of the program, using huge pages. The rte_malloc has the following nice features, aimed at improving the performance of applications: *(i)* objects are allocated aligned with the cache line, and *(ii)* applications can require that an object is allocated on a particular NUMA[2] socket.

The mempool, instead, is a set of pre-allocated objects; in particular, it contains a rte_ring of objects that can be acquired/released by lcores according to their needs. It is worth noting that several lcores can share the same mempool; in this case, to improve performance, it is possible to use a per-core cache of free objects. The mempool presents the following nice features: *(i)* it is created using huge pages, in order to reduce TLB misses; *(ii)* all objects within the mempool are aligned properly so that accesses to them are spread across all memory channels; *(iii)* it can be allocated on a particular NUMA socket. As a final remark, in a multi-processes DPDK application, a mempool can only be created by the primary process, through the EAL.

[2] In a Non-Uniform Memory Access (NUMA) multi-processor system, the memory is divided into multiple NUMA nodes. Processors have direct access to the local memory of the NUMA node they belong to but must go through an interconnect in order to access memory that is local to another NUMA node. As a result, the access time depends on the memory location relative to the processor. Optimized applications must therefore ensure that memory is allocated on the NUMA node that runs the code for which the lower access time is the most beneficial.

2.4.1.4 Data exchange mechanisms

lcores in the same DPDK application can exchange data among each other through the rte_ring, which is a lockless FIFO queue that allows burst/bulk-single/multi-enqueue/dequeue operations. Each slot of the rte_ring contains a pointer to the object to be exchanged; this way, data can be moved across lcores in a zero-copy fashion. If the rte_ring is used to exchange network packets, each slot of the buffer points to an rte_mbuf, which is an object in the mempool that contains a pointer to the packet, and some metadata associated with the packet itself (e.g., its length). It is worth noting that each rte_ring used in a DPDK application must be created, through the EAL, by the primary process.

2.4.1.5 Executing asynchronous operations

To enable the execution of callback functions asynchronously, DPDK provides some timers, which *(i)* can be periodic or single, and *(ii)* can be loaded in one core, and executed into another. The EAL provides an interface to add, delete and restart timers.

2.4.1.6 Accessing to the network

The Poll Mode Driver (PMD) library is the part of DPDK used by applications to access the network interface cards (NICs) without the intermediation (and the overhead) of the operating system. In addition, this library allows applications to exploit features offered by the Intel NIC controllers, such as RSS, FDIR, SR-IOV and VMDq. The PMD does not generate any interrupt when packets are available in the NIC, hence the lcores that receives packets from the network should implement a polling model.

As a final remark, packets received from the network are stored into a specific mempool, indicated during the initialization of the application, so that all the applications sharing the mempool can potentially access the data without incurring any copies.

2.4.2 netmap

netmap [13] is a library that provides, to packet processing applications such as network functions, fast access to the network interface cards. Hence, unlike DPDK, it does not offer an entire framework to build applications, but just a way to efficiently access to the packets that must be processed.

netmap uses some well-known performance-boosting techniques, such as memory-mapping the card's packet buffers, I/O batching, and the possibility to model send/receive queues as circular buffers to match what is implemented in the hardware. The programming model is simple (circular rings of fixed size buffers), and applications use only standard system calls: non-blocking `ioctl()` to synchronize with the hardware, and `poll()`-able file descriptors to wait for packet receptions or transmissions on individual queues.

It is worth noting that netmap implements a special device, `/dev/netmap`, which is the gateway to switch one or more network cards to netmap mode, where the NIC's datapath is disconnected from the operating

system. The file descriptor returned by opening this device can be used to `poll()` one or all the queues of a network interface.

2.4.2.1 Overall architecture

The key components in the netmap architecture are the data structures shown in Figure 2.6, which are designed to provide: *(i)* reduced/amortized per-packet overheads, *(ii)* efficient forwarding between interfaces, *(iii)* efficient communication between the NIC and the host stack, and *(iv)* support for multi-queue adapters and multi core systems. netmap supports these features by associating with each network interface three types of user-visible objects: packet buffers, netmap rings, and netmap_if descriptors. It is worth noting that the objects associated to all the netmap-enabled interfaces are placed in the same memory region, which is allocated by the kernel in a non-pageable area, and shared by all the user processes. Thanks to this single memory region, it is possible to implement zero-copy forwarding between interfaces.

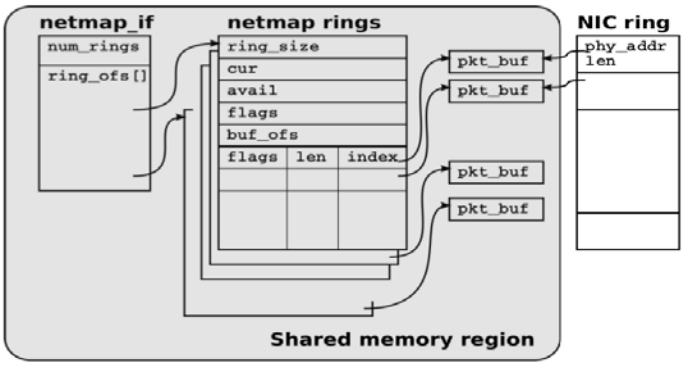

Figure 2.6 - Shared memory area exported by netmap

Packet buffers have a fixed size and are shared by the NICs and user processes. Buffers for all netmap rings are pre-allocated when the interface is put into netmap mode, so that during network I/O there is never the need to allocate them. The metadata describing the buffer (index, data length, some flags) are stored into slots that are part of the netmap rings described next. As evident from the figure, each buffer is referenced by a netmap ring and by the corresponding hardware ring.

A netmap ring is a device-independent replica of the circular queue implemented by the NIC, and includes information such as: *(i)* the number of slots in the ring, *(ii)* the current read or write position in the ring, *(iii)* the offset between the ring and the beginning of the array of packet buffers, and *(iv)* an array, in which each entry contains the index of the corresponding packet buffer, the length of the packet, and some flags used to request special operations on the buffer.

Finally, a netmap_if contains read-only information describing the interface, such as the number of rings and an array with the memory offsets between the netmap_if and each netmap ring associated with the interface.

2.4.3 Direct NIC Access (DNA)

Similar to netmap, DNA [14] is a library designed to provide to packet processing applications mainly a way to efficiently access to the network interface cards, both to receive and transmit packets.

In the remainder of this section, in addition to DNA, other two technologies will be briefly presented: PF_RING, which is the predecessor of DNA, and PF_RING ZC, which represents the evolution of DNA.

2.4.3.1 PF_RING

Briefly, PF_RING is a library that polls packets from NICs by means of Linux NAPI (NAPI is an interface to use interrupt mitigation techniques for networking devices in the Linux kernel, intended to reduce the overhead of packet receiving). This means that NAPI copies packets from the NIC to the PF_RING circular buffer, and then the user space application reads packets from ring. In this scenario, there are two pollers, both the application and NAPI and this results in CPU cycles used for this polling; the advantage, is that PF_RING can distribute incoming packets to multiple rings (hence multiple applications) simultaneously.

2.4.3.2 DNA

DNA is a way to map NIC memory and registers to user space, so that there is no additional packet copy besides the DMA transfer done by the NIC. This results in better performance as CPU cycles are used uniquely for consuming packets and not for moving them off the adapter. The drawback is that only one application at a time can open the DMA ring (note that modern NICs can have multiple RX/TX queues thus one application per queue can be simultaneously started) or, in other words, that user space applications need to talk each other in order to distribute packets.

As evident from Figure 2.7, to achieve better performance (in terms of throughput), in DNA the circular buffer containing the packets coming from the NIC used in FP_RING is replaced with a memory ring allocated by the driver, and which contains pointers to the input packets.

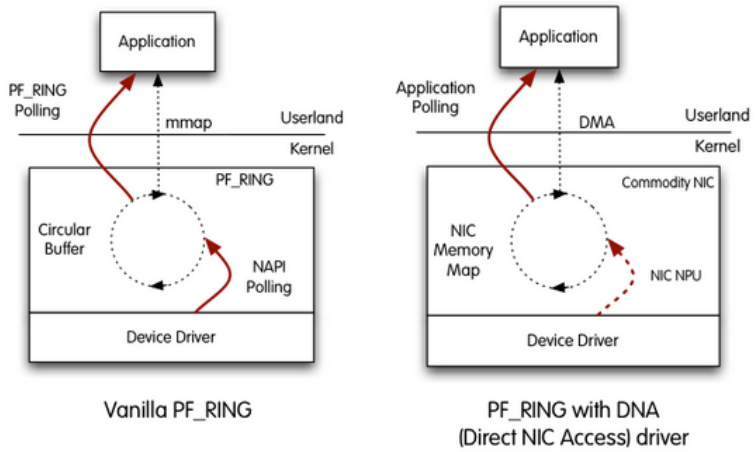

Figure 2.7 - Vanilla PF_RING vs PF_RING with DNA

2.4.3.3 PF_RING ZC

Recently, PF_RING ZC, a successor to DNA has been released. It implements zero-copy operations, in particular across threads, applications and for inter-process and inter-VM (KVM) communications.

In this last case, it is possible to forward packets in a zero-copy fashion to a KVM virtual machine without using techniques such as PCI passthrough, and to create a pipeline of applications communicating across VMs with zero copy, as shown in Figure 2.8. In other words, this technology puts together both the primitives to get fast (raw) access to the NIC, and the ones that allow transferring data between (userland) processes in a zero-copy fashion.

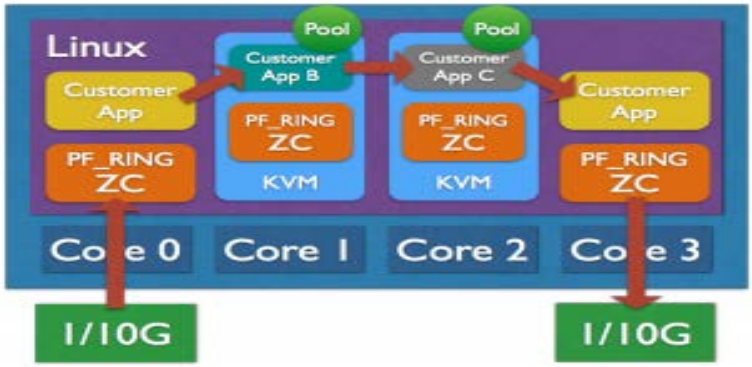

Figure 2.8 - Zero-copy communication between VMs

2.5 Virtual Switching Solutions

As was discussed in D5.1, the Universal Node includes a virtual switching element to support the local traffic steering. It provides switching of the packets between the physical ports of the node and the virtual ports of the Virtual Network Functions running on the node. This section presents several existing virtual switch implementations that can be considered as basis for the implementation of the virtual switching functionality of the UN.

2.5.1 Open vSwitch

Open vSwitch (OVS) [15] is an open source virtual switch implementation that supports the OpenFlow protocol (see 2.6.1) as well as various tunnelling and monitoring protocols. It provides a forwarding layer abstraction to enable support of various software and hardware platforms. On a standard server platform running Linux, it provides a kernel module that implements the "fast path" for already matched flows. Flows unknown to the kernel module are deferred to the main user space switch process, ovs-vswitchd, which applies the full set of

flow rules and downloads the resulting instructions into the kernel module so further packets of the same flow can stay in the fast path.

Quite naturally, there has been interest in bringing efficient data plane processing technologies, such as those described in section 2.4, into OVS. One such effort is the Intel DPDK vSwitch, a fork of OVS, which is covered in a later section.

But the latest standard OVS implementation now has experimental support for DPDK. When this is turned on, the kernel module is not used and the whole switch runs in userspace using only the ovs-vswitchd process. In this approach, DPDK support is integrated at the "netdev" OVS interface. This is in contrast to the DPDK vSwitch project which provides a different implementation starting higher up in the architecture, at the "dpif provider" interface. The "netdev" interface is simpler because it only deals with network interfaces, not with OpenFlow switching. With such an approach, all packets however go through the main matching logic in vswitchd which was originally not designed for the full throughput but only for packets that the underlying fast path could not match. Improvements in the performance of this part (for example inspired by what DPDK vSwitch does but applied to the standard ovs-vswitchd code) combined with the new DPDK based OVS netdev implementation could bring an interesting mix of features and performance.

2.5.2 Intel DPDK vSwitch

The Intel DPDK vSwitch [16] is a fork of the Open vSwitch project that aims at bringing the performance benefits of Intel DPDK into the OVS forwarding layer. It implements an alternative DPDK-based data path as a separate user space process running alongside ovs-vswitchd. This project started before DPDK support was added to the standard OVS code base.

Using DPDK, the user space data path of DPDK vSwitch is based on user IO and poll mode operation and hence completely bypasses all kernel processing and associated overhead (interrupt and softirq processing, device layer). OpenFlow switch processing of the packets can be applied immediately to the packets received from the Ethernet controller. Additionally, the first packets of a new flow don't have to cross the kernel to user space boundary anymore since they are already in user space. These advantages are common to this approach and the standard OVS with DPDK support mentioned earlier.

DPDK vSwitch was implemented as an alternative "dpif provider" in the OVS architecture. This is a lower interface of the OpenFlow switch layer of OVS where all wildcard rules are exploded as exact-match entries.

As Intel DPDK vSwitch re-implements a complete data path, it can make maximum usage of advanced optimizations enabled by DPDK (Huge Page usage, lock free and cache aligned structures, etc.) without affecting the other parts of OVS. The downside is however that Intel DPDK vSwitch must re-implement all data path features and, as a result, is currently not as feature rich as the standard OVS data path.

2.5.3 Extensible DataPath Deamon (xDPd)

An alternative to using Open vSwitch is xDPd [17] and the porting of the I/O subsystem to DPDK. This code has been published, but in its current version is comparable to the "netdev" option described for OVS in the previous chapter.

xDPd has been developed by BISDN as a user-space implementation of an OpenFlow v.1.0, v.1.2, v.1.3 [18] datapath. It was designed to run multiple hardware platforms and therefore shows a somewhat cleaner architecture than OVS. It has a documented internal interface, the Hardware Abstraction Layer (see Figure 2.9), which is the API between the hardware-independent Control and Management Module (CMM) and the hardware-dependent Platform Driver.

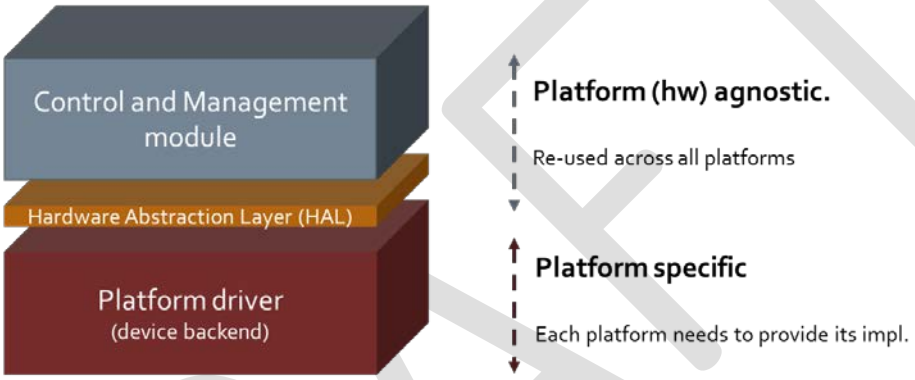

Figure 2.9 - xDPd general architecture

At the time of writing, the available hardware platforms are x86-gnu-linux, x86-dpdk, Cavium Octeon, Broadcom, EazyChip, and NetFPGA10G. The availability of the source code for each of the platforms is however subject to licenses of hardware vendors, and not all of the platform drivers could be made Open Source.

The implementation of the pipeline for different hardware platforms is eased by the availability of a set of libraries called ROFL (Revised OpenFlow Library). ROFL contains the code for the pipeline (packet classification, matching, actions) as well as the protocol machine for OpenFlow endpoints and management interfaces. For this reason, ROFL is also used for the implementation of controllers.

Figure 2.10 shows the functional decomposition of the Control and Management Module of xDPd.

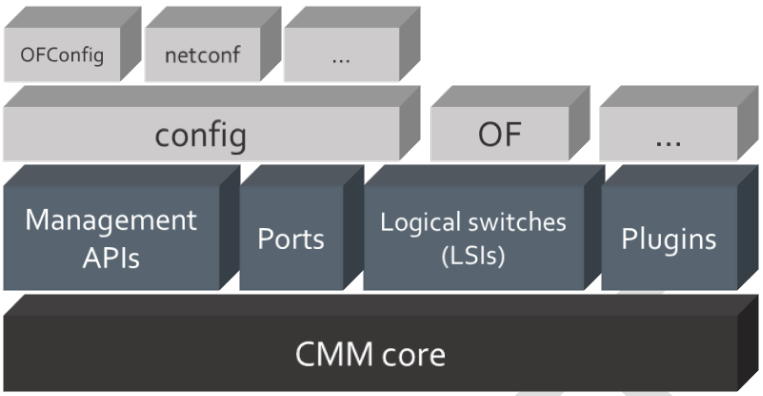

Figure 2.10 - Control and Management Module of xDPd

One of the features of xDPd is the support for multiple Logical Switch Instances (LSI). The LSIs are created either through a configuration file that is evaluated at startup of xDPd or dynamically through a configuration interface.

The LSI is bound to network interfaces, and, in the case of multiple LSIs, the network interfaces have to be exclusively assigned to one LSI. This is a simple way of slicing and a first realization of a virtualization.

2.6 Control Plane – Data Plane Protocols

In legacy "SDN-like" (centralized) networks CP-DP protocols traditionally implement function-specific approach, e.g. GTP-C.

In a Universal Node environment where functionality is a matter of configuration this approach is not feasible: if one wants to use a protocol describing a large variety of packet processing tasks, this protocol has to be generic and abstract enough to describe the many existing and envisioned dataplane tasks, yet it also must provide good-enough performance and hence should not be too abstract (e.g. a very generic protocol that carries only user-definable TLVs[3])

OpenFlow and ForCES were the two initiatives that were trying to find the sweet spot between high-enough abstraction level and good-enough performance.

2.6.1 OpenFlow

OpenFlow [18] (OF) was originally developed by researchers at Stanford to enable easy testing on real campus networks. It enables remote controllers to determine the path of network packets through the network of OF-capable switches. This separation of the control from the forwarding allows for more sophisticated and simpler-to-use traffic management than with legacy equipment. Also, OpenFlow aims to allow switches from different suppliers to be managed remotely using a single, open protocol.

[3] Type-Length-Value: the generic way of expressing parameters

Historically OpenFlow (version 1.0) used a large flow table that contained essentially extended ACL entries. This implied several limitations: e.g. when one wanted to configure non-overlapping L2 and L3 rules it ended up with a huge rule-set containing the Descartes-product of the eventually independent rules.

In later revisions this serious limitation was removed, now one can use up to 255 tables along with groups (common actions for many flows, or load balancing or multicast) and meters (traffic policing). This versatility causes serious problems for hardware switch vendors, while software switches still suffer from constraints that remained because of the hardware track: e.g. the limitation on the number of tables (255), the fixed order of tables and groups (no group to table jump is allowed) and the loop-free pipeline (it cannot go "back" to an already visited table). There are also several limitations on the supported packet manipulation actions, e.g. tunnelling support is very limited.

2.6.2 ForCES

Forwarding and Control Element Separation (ForCES, RFC 3746 [19]) tried to be an even more generic solution. It defines a standard framework and mechanism for the exchange of information between the logically separate functionality of the control and data forwarding planes of packet processing devices. It clearly separates control plane functionality such as routing protocols, signalling protocols, and admission control from per-packet activities of the DP such as packet forwarding, queuing, and header editing. It describes several basic building blocks and their control, but also allows easy extension.

Although it is well-defined in IETF and probably even better engineered than OpenFlow it lacks the momentum that can be seen around OF. Without entering into religious debates, one reason for that might be that the level of abstraction that ForCES provides is a bit too high for the data plane community.

2.7 Data Plane Management Protocols

2.7.1 OF-Config

The OpenFlow Management and Configuration Protocol (OF-Config) [20] is a special set of rules that defines a mechanism for OpenFlow controllers to access and modify configuration data on an OpenFlow switch.

Although controllers use OpenFlow to define how packets are forwarded between individual sources and destinations in the network, OF itself doesn't provide the configuration and management functions that are needed to allocate ports or assign IP addresses. OpenFlow configuration protocols, like OF-Config, help with this and give network engineers an overall view of every area of the network.

While the OpenFlow protocol generally operates on a time-scale of a flow (i.e. as flows are added and deleted), OF-Config operates on a slower time-scale. The OF-Config protocol is being developed by the Open Networking Foundation and is based on NETCONF [21] where the datamodel and RPCs are modelled in YANG [22].

2.7.2 OVSDB

The Open vSwitch Database management protocol (OVSDB, RFC 7047 [23]) is a protocol specifically designed to allow programmatic manipulation of the configuration of Open vSwitch. The tasks are relatively similar to that of OF-Config.

In an Open vSwitch implementation, a database server and a switch daemon are used. The OVSDB protocol is used in a control cluster, along with other managers and controllers, to supply configuration information to the switch database server. Controllers use OpenFlow to identify details of the packet flows through the switch. Each switch may receive directions from multiple managers and controllers, and each manager and controller can direct multiple switches.

3 UNIFY overview

3.1 Overall Architecture Overview

The UNIFY overarching architecture follows a three layered model with a narrow waist at the resource orchestration point, as depicted in Figure 3.1, which also includes the defined reference points. Additionally, a user or Application layer is considered on the top of the three layered UNIFY architecture.

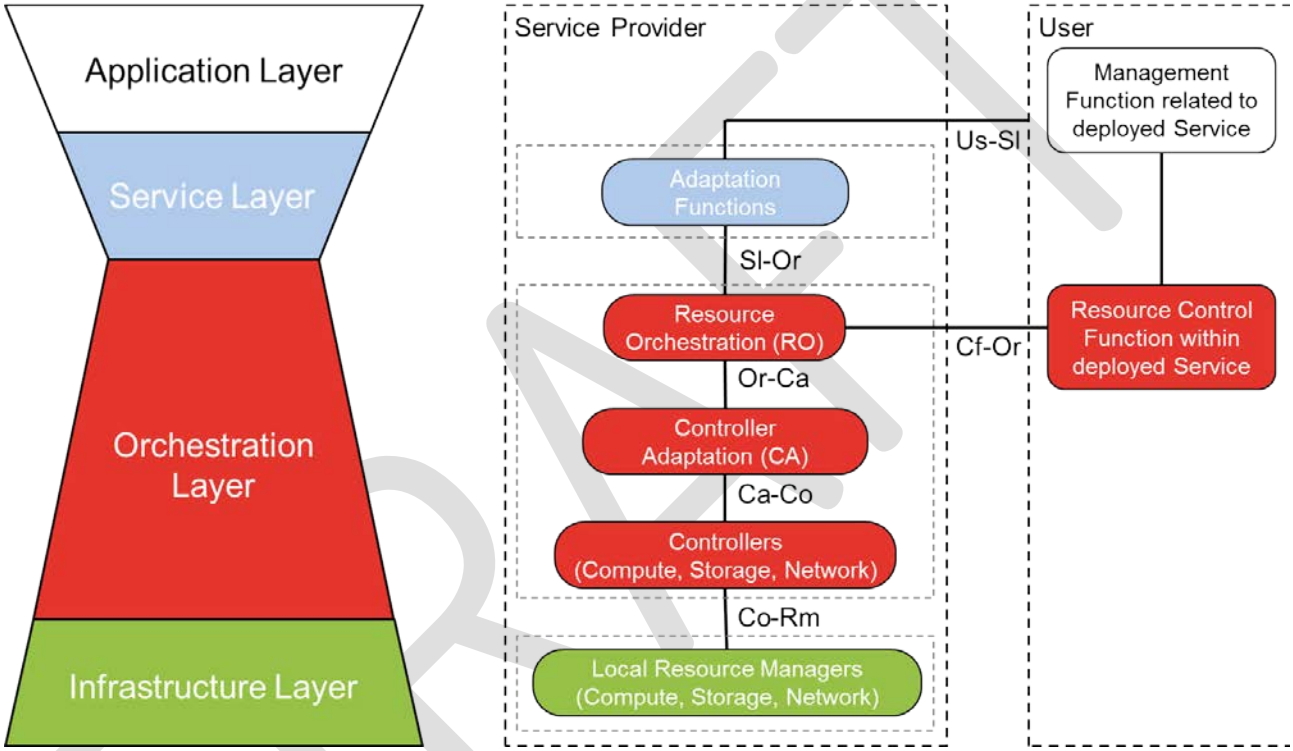

Figure 3.1 - UNIFY overarching architecture, layers and reference points

- The Service Layer (blue in Figure 3.1) owns the service logic for different kind of services which can be found as appliances, servers, or VMs, receives Service Graphs as input and translates them to Network Function Forwarding Graphs (NF-FG) before providing them to the Orchestration layer. This layer also provides other services related to user registration and user profile, as well as the AAA server for restricting access to a service to only registered users of the service, etc.

- The Orchestration Layer (red in Figure 3.1) is the core of the system; it is responsible for maintaining a global view of the network, computing and storage resources together with infrastructure capabilities as well as for the mapping of service requests to resources in the network. In the UNIFY view, service request are received as network function forwarding graphs (NF-FG), which are described in the next

subsection, with their associated deployment constraints. The Orchestration Layer represents the narrow waist in the UNIFY model because the abstractions on which it acts are restricted to compute, storage and network resources. The architecture assumes that orchestration can take place at multiple levels:

- High level (global and possibly regional) orchestrators: split the NF-FG into sub-graphs based on geographical constraints (e.g. link delays) and high-level resource map. It sees the underlying UNs or DCs as huge resource pools, the internal details are hidden on this level
- Local or hardware orchestrator: it covers a DC or a UN and does VNFs deployment based on actual hardware knowledge, e.g. capabilities, resources. It can deal with internal networking (e.g. backplane, inter-socket links), and can hide these details from the higher level. It is also the last control node to select an implementation for the given VNF type based on e.g. capability and performance constraints

Note that the orchestrator can have even more levels if the same abstraction is followed at each level, so the NF-FG description can be used on the interface and see the underlying nodes as abstract processing, storage and network resources.

In theory it is also possible to have one huge orchestrator only that would see all details inside DCs and UNs, but this would probably cause scalability issues – this orchestrator would see basically individual CPU cores

- The Infrastructure Layer (green in Figure 3.1) is the lowest layer of the overarching architecture and encompasses all the actual resources (i.e., compute, storage and networking) providing the physical means to actually deliver services. Besides the Universal Node, other types of resources are also considered in the scope of the UNIFY project, such as data centres, SDN Nodes and Legacy Nodes.

From this architecture, the service Adaptation Functions, Resource Orchestration, Controller Adaptation, Controllers and Local Resource Managers are identified as key components and their corresponding reference points have been defined (see Figure 3.1):

- Us-Sl: between the Users (services) and the Service Layer. The customer can specify its peering points with the UNIFY network, the requested network services and also the corresponding SLA requirements. These requests can be specified in multiple ways (e.g. bundled service that is composed of many VNFs and integrated SLA, service chain where the user specifies the types of VNFs, service chain with user-specified VNF binaries – and of course also the mix of these with possibly end-to-end SLA requirements, e.g. delay, bandwidth, "value").

- Sl-Or and Or-Or: between the Service Layer's Adaptation Functions and the Resource Orchestration and also in between the different levels of orchestrators. This interface relies on Network Function Forwarding Graphs (NF-FG) as are described in the following subsection. Using the service graph, the service layer defines VNF types, their connections and the KPIs on both the connections and VNFs.
- Or-Ca: southbound interface of the Resource Orchestration toward an adaptation logic scoping and interfacing with various controllers. This interface is also based on NF-FGs. The Controller Adaptation translates the NF-FG according to the northbound interfaces of the infrastructure elements it will distribute it to, splitting it according to the different concerns of the controllers (compute, networking) when applicable.
- Ca-Co: Northbound interfaces of controllers. This interface is controller specific and carries control messages towards the infrastructure (e.g. virtual machine instantiation or removal, switch control).
- Co-Rm: southbound interfaces of controllers. This is most probably some OpenFlow-like interface for switch control and controller specific for resource control.

3.1.1 Network Function Forwarding Graph

A Network Function Forwarding Graph describes the service requested by a User of the application layer as a graph where:

- The nodes represent the Network Functions composing the service chain and the Service Attachment Points (SAP) where the service is provided.
- The edges represent the logical connectivity between the NFs and to the SAPs.

The NF-FG also contains, attached to each of these elements, the Key Performance Indicators for the resources, specifying how the service has to be delivered. Example KPIs are: delay, bandwidth [bps], transaction rate [pps], and some value parameter (of course it can be more complex than a simple number) to express the importance of the given service.

The NF-FG is initially created by the Service layer as a translation of a Service Graph and SLAs requested by the user. Note that the Service Graph contains essentially the same information but at a higher abstraction level. As the NF-FG traverses the different layers, it is gradually enriched with the information required by each layer, based on the information provided by the VNF Repository (repository or catalogue containing the NF models and decompositions to be used by the different layers).

3.2 Universal Node in the UNIFY Architecture

In the layered model defined in UNIFY, the Universal Node implements functionalities of both the Infrastructure layer (completely) and the Orchestration layer (partially). The northbound interface of the UN

regarding the programmability framework corresponds to the Ca-Co reference point with the following considerations:

- The scope of the input provided by the Controller Adaptation layer to the UN is a sub-graph containing all the elements to be deployed in the UN.
- The input format will be a NF-FG as handled in the upper layers (Or-Ca) so that no adaptation is actually required.

So for the UN, the Controller Adaptation layer will not need to perform any adaptation but only the scoping necessary to provide the UN with the appropriate sub-graph.

Figure 3.2 shows the UN in relation to the global architecture and reference points defined.

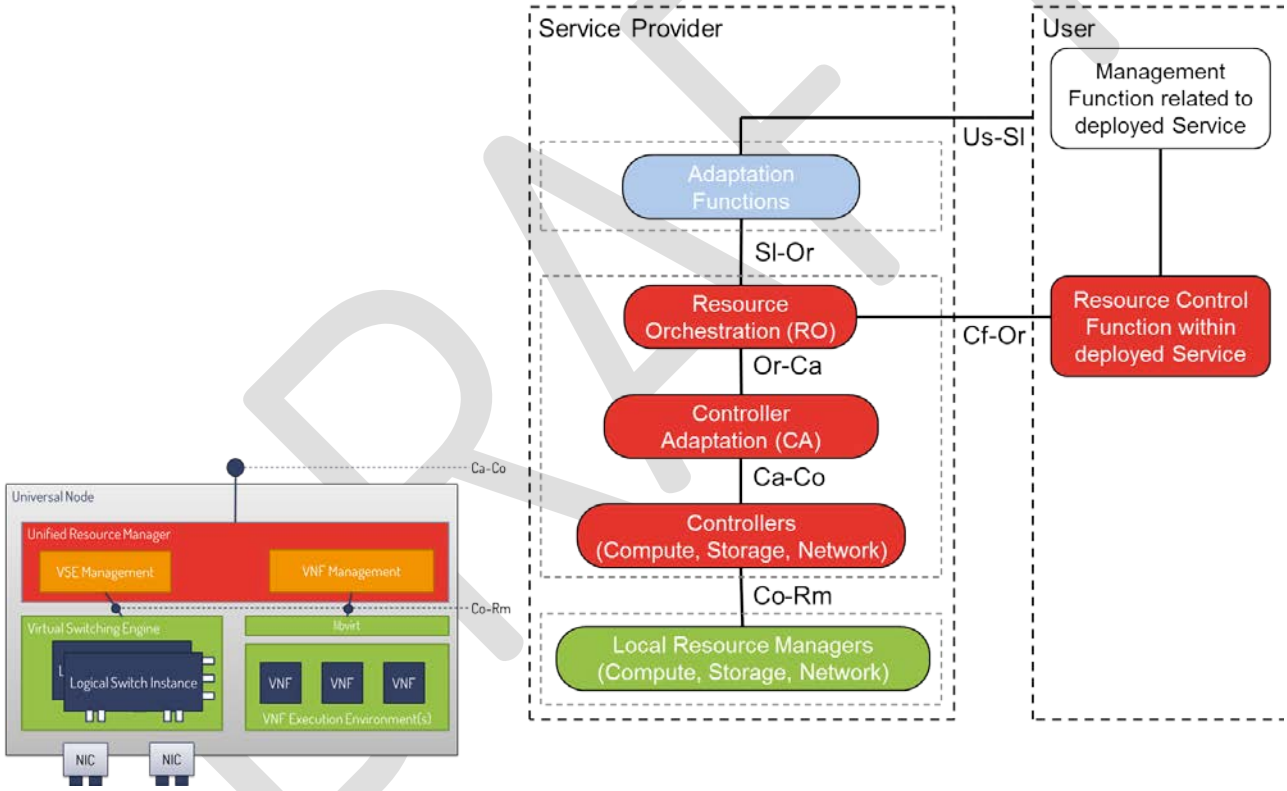

Figure 3.2 – UN architecture in relation to reference points

The current working approach in WP5 is a Unified Interface for all resources provided by the UN so it can optimize the placement of the requested NF-FG in its internal resources. This vision has evolved from the one presented in D5.1 where the interfaces were split by type of resource and the decomposition of the NF-FG in its components was done at higher layers.

Finally, regarding the integration of the UN in the overall programmability framework and the scopes of WP3 and WP5, the following decisions have been made:

1. The scope of WP5 is a single UN, any process involving more than one UN will be handled by the upper level orchestrator (WP3).
2. The scope of the NF-FG handed to the UN is a subset of the global NF-FG containing all elements to be deployed on that UN (also see 3.2.1):
    a. If several NFs of the same NF-FG are deployed in the same UN, the input will be a single NF-FG sub-graph with the information related to all of them (including internal links).
    b. If several NFs of different NF-FGs are deployed in the same UN, the input will be separate NF-FG sub-graphs with the information related to each of them.
3. Regarding the fulfilment of KPIs, two scenarios are possible:
    a. At deployment time, the UN can detect that it is not able to fulfil the requirements and would then reject the deployment (WP5 -> WP3). Note that this behaviour is only possible in a "resource reservation" based scheme.
    b. At runtime, multiple entities can potentially detect that the requirements are not fulfilled and that a scale up process should be triggered. These are detailed below.
4. The scaling of the NF-FG may be triggered from different sources:
    a. A NF instance may itself realize that its current capacity is being exceeded, or far exceeds the demand. If it is capable of scaling with support from the local orchestrator (see below), then no interaction outside of the UN is required. If it doesn't have this capability or there aren't enough local resources available, the application may notify the UNIFY architecture (local orchestrator) or its management plane, either of which would then proceed with the appropriate scaling actions.
    b. If the criteria for triggering the scaling process is monitored by the UNIFY architecture or reported to the UNIFY architecture, then, when the criteria is met, the Orchestrator can be signalled so additional resources or instances of NFs can be deployed/removed. In this case, the definition of the NF-FG must contain the criteria and thresholds that control the scaling process, as well as the action to be triggered when those are met.
    c. If the criteria is not monitored by (or reported to) the UNIFY architecture, then it must be monitored through the NF management plane (external to the UNIFY architecture) that would then request a modification of the NF-FG to include additional resources or instances of NFs.

5. The process of scaling the NF-FG is application (NF) dependant:

   a. If the application supports parallel processing with multiple threads, scaling can be done inside the application – the only thing the application needs is permission to use more resources. This can be achieved completely transparently to any external entities.

   b. If the load to the given application can be distributed without disrupting the application logic, it is possible to use a generic load balancer and scale the application even if the process that implements it doesn't support parallel processing / multithreading. In this case, scaling will be managed by the UN infrastructure (WP5).

   c. If the application load cannot be distributed due to reasons related to application logic, the scaling of such application will be managed by the upper layers (WP3).

3.2.1 NF-FG as input to the UN

The NF-FGs that are deployed on the UN only include elements that exist within the Universal Node itself. That is, they are sub-graphs of larger NF-FGs that the upper level orchestrator generates and that implement a complete end-to-end instantiation of a service throughout the complete network. The UN only receives the part of the graph that the upper level orchestrator decided to deploy on the UN.

In order to allow the UN to perform internal optimizations of the deployed NF-FG, the scope of the sub-graph must include both the elements related to the NFs to be deployed and the elements related to the traffic steering mechanism. That is, the scoping performed on the Controller Adaptation layer for the sub-graphs to be deployed on a UN must be done according to a domain criteria and not a functional criteria, as exemplified in Figure 3.3.

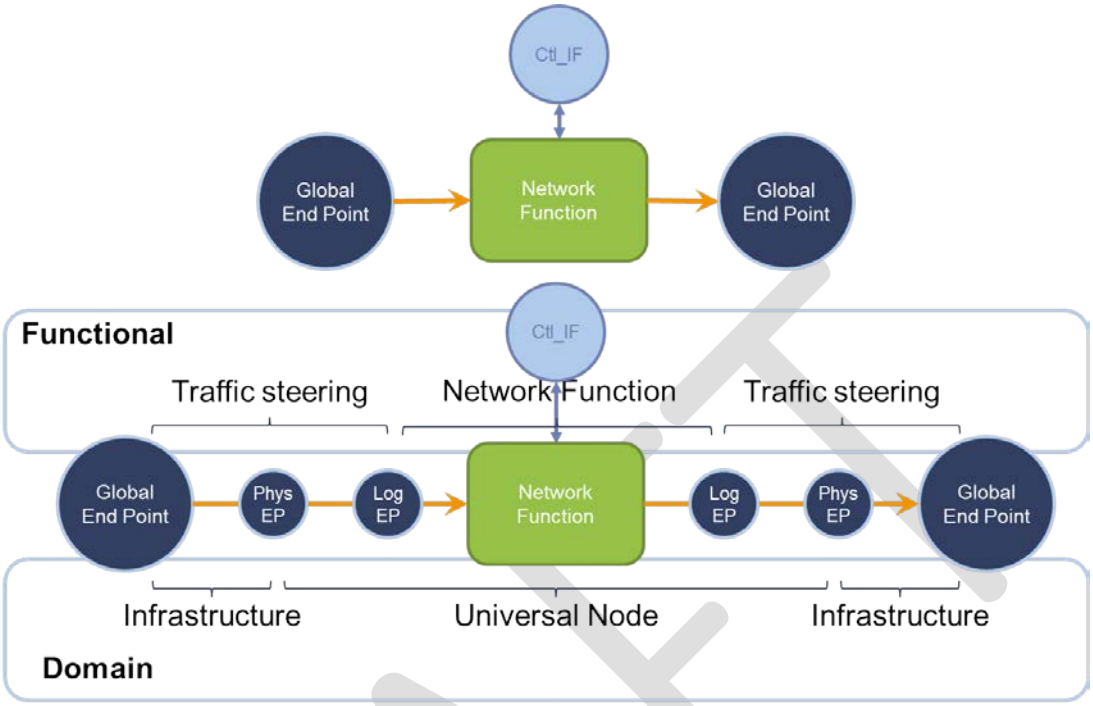

Figure 3.3 – Service graph, NF-FG graph and traffic steering

The Global End Points to which the NF must be connected are only known to the upper-level orchestrator while the UN only knows about its Physical End Points (its network interfaces) and the virtual ports of the deployed NF ("Logical End Points"). The NF is however part of what gets deployed on the UN. Hence the scoping done in the Controller Adaptation for the UN must include the elements between the physical End Points, including the NF(s), as shown by the "Universal Node" domain on the figure, whereas the connectivity between the Global End Points and the Physical End Points must be delegated by the upper-level orchestrator to different elements of the infrastructure (e.g. an SDN controller).

The NF-FGs to be deployed on the UN include the following key elements:

- One or more VNFs
- Links between the VNFs
- Endpoints at which traffic enters or leaves the graph
- Return address to connect the control interface ("Ctl_IF" on the figure) of the VNFs to

The VNFs that are part of the graph may be fully specified platform-specific ready to run configurations or, more often, abstract NF types for which the UN must select, from multiple candidate implementations, the one that is compatible with its platform and that most closely matches the requirements specified with the graph

(limits like the number of users or flows, as well as the KPIs must be taken into account). Note that this interface approach supports creating VNFs running user (3rd party) custom code by referencing a fully specified VNF instead of an abstract NF type.

Endpoints of the graph include flow space specifications that allow the UN to know which flows the VNFs of the graph will be processing. This enables the UN to perform optimized placement of the VNFs with regards to platform topology concerns (e.g. processor sockets). This is one of the main differentiator of the UNIFY UN approach with more typical cloud deployments where placement of application threads is not as critical.

Similarly, flow space specifications are also attached to the links between VNFs in the graph so that the UN can set up the desired traffic steering between the functions. These specifications can refer to the source VNF port and to header fields of the packets. Most VNFs will not be concerned by what happens to the packets that they send out after processing and will therefore usually output all traffic through a single virtual port. Other elements of the packets can then be used to split traffic towards multiple next-in-chain VNFs. However, some VNFs may also perform routing-type of functions, as shown in Figure 3.4. In this case, they will be configured with multiple virtual ports supporting outgoing traffic and will select the outgoing port on a per packet basis. Since a routing decision is already taken by the VNF itself, the flow rules specified for the traffic coming out of those ports will typically only match on the port, even though the usage of other fields is allowed for further splitting of the flows.

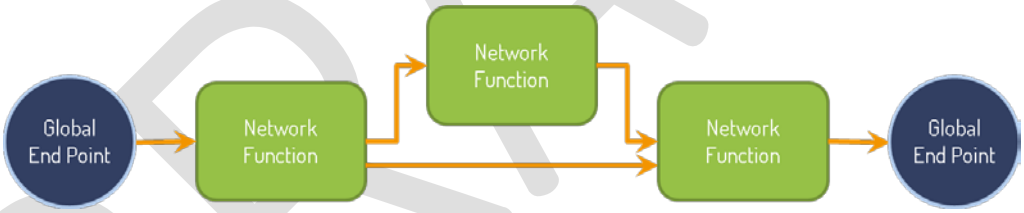

Figure 3.4 – NF-FG with a routing decision in the first NF

4 Universal Node Architecture

A high-level UN architecture was introduced in D5.1. However, this architecture has evolved, especially in terms of the interface with the upper orchestration layer with the introduction of Network Function Forwarding Graphs (NF-FG) at this interface. This section first provides an updated high-level overview of the UN functional architecture, and then refines the architecture and introduces more concrete architectural blocks and how they interact with each other.

4.1 Overview

Based on the positioning of the UN within the UNIFY architecture that was presented in section 0, a functional view of the UN is presented in Figure 4.1.

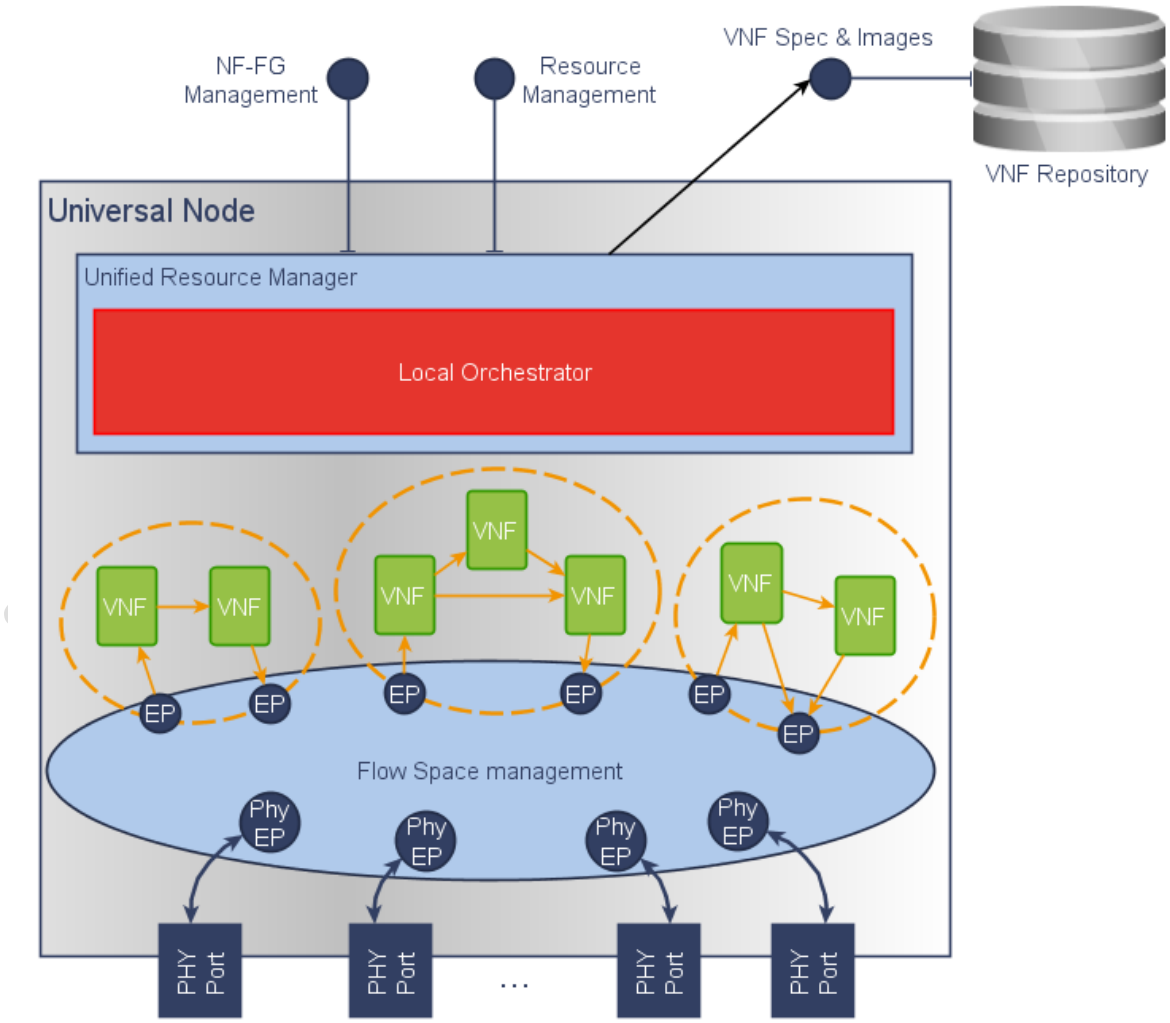

Figure 4.1 – UN Functional View

The figure presents a few NF-FGs deployed on the node, connected to the external physical network through a flow space management function that takes care of directing the flows as specified for the NF-FG endpoints. All of this is managed by a local orchestration function that is responsible for the complete deployment of the NF-FGs on the UN.

The figure also shows the three external interfaces of the UN:

- Resource management interface: This interface covers the discovery of resources exposed by the node as well as possible updates to those resources (such an update may be the result of actions performed through the other interfaces, or from a reconfiguration of the node) and also reporting to the upper layers the current availability of resources due to the NF-FGs already deployed in the UN.
- NF-FG management interface: The Universal Node management interface focuses on deploying and managing Network Function Forwarding Graphs.
- VNF Template and Images repository interface: When the UN is instructed to deploy a NF-FG, it needs to fetch the detailed specification and the related binaries of the involved VNFs. This constitutes an outbound interface of the UN towards a central VNF repository.

In the above functional view, the compute, storage and networking aspects are not separated as this is the intention of handling NF-FGs as primary entities in the orchestration interactions.

When moving to an architectural view however, the different aspects have to be distinguished as individual resources have to be managed using their own specific means. This leads to the UN high-level architecture of Figure 4.2.

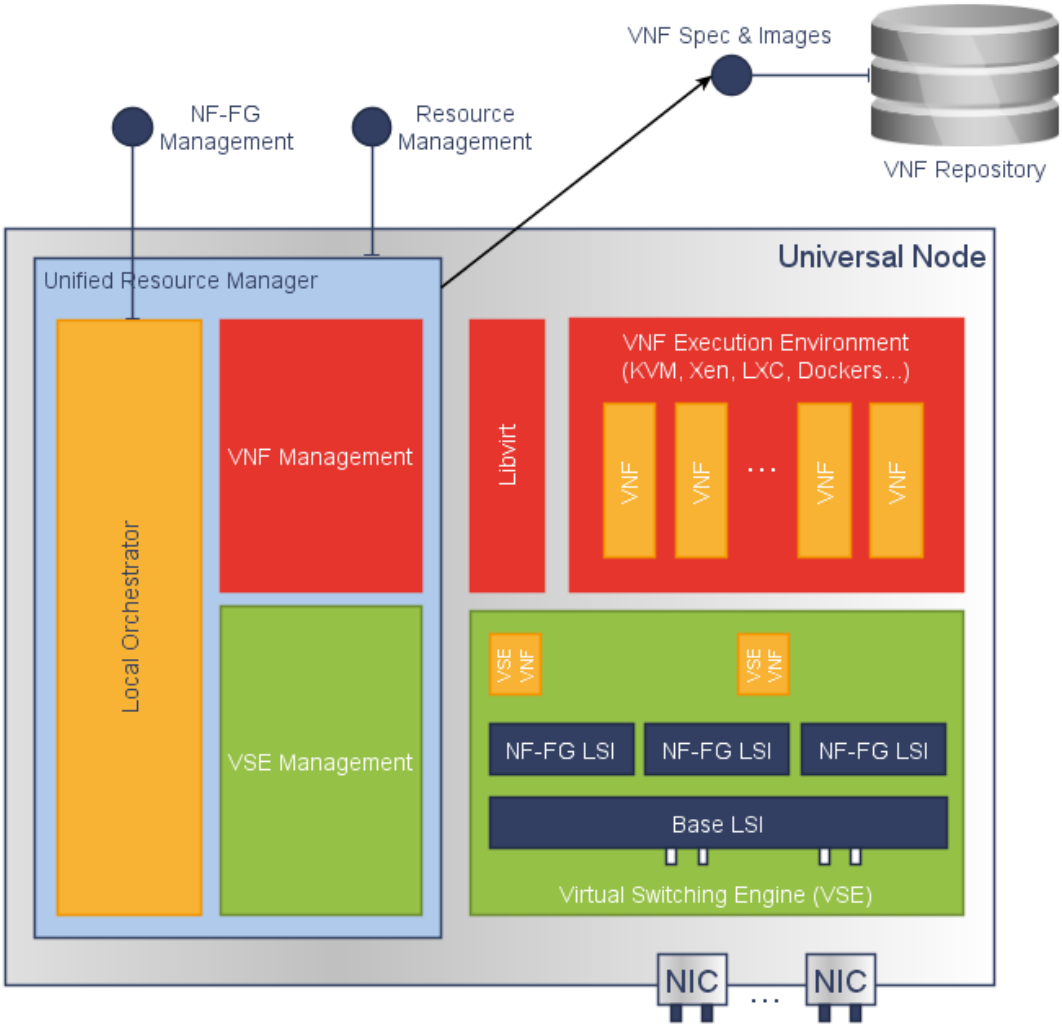

Figure 4.2 – UN Architecture Overview

The figure shows the three main functional blocks of the UN: the Unified Resource Manager (URM), the Virtual Switching Engine (VSE) and the VNF Execution Environment (VNF EE).

The VNF Execution Environment (VNF EE) consists in one or more compute platform virtualization solutions, including hypervisors and simpler container based approaches (Linux Containers, Docker...).

The Virtual Switching Engine (VSE) implements packet switching on the UN, managing the physical network interfaces (NICs) and the traffic steering with and within the deployed NF-FGs.

On top of the VNF EE and the VSE, the Unified Resource Manager (URM) plays the role of a local orchestrator that has complete and detailed view on the resources available on the node, their topology and related usage constraints and limitations. It provides the UN interfaces and controls the VNF EE and VSE to fulfil the NF-FG deployment and management requests.

Having the VNF EE and the VSE control as part of a single high-level resource manager component allows taking into account the compute resources required by the VSE when configuring certain topologies or functions.

4.2 UN Components

4.2.1 Host Environment

Typically, the systems on which the UN components execute will include an operating system providing support for managing and accessing the system resources, spawning and controlling processes, etc. This constitutes the UN Host Environment.

Depending on the virtualization solution chosen, the host environment may be a standard Linux operating system, or, in the case of Xen, it may consist of a special management domain ("domain 0").

In order to achieve high performance packet processing, using technologies such as those described in section 2.4, the host environment and its kernel will be kept out of the packets fast-path.

The host environment will support execution of the other UN components and will be used to control placement of processes and threads on specific CPU cores.

As many tasks of the host environment as possible as well as the non-performance-critical parts of the URM and other UN components will share a single reserved CPU core. This will avoid any interference with the packet processing tasks (on the host or in guest VMs) which will be assigned to other cores.

4.2.2 Unified Resource Manager

The Unified Resource Manager (URM) is the main component of the UN, responsible for controlling the local resources and interacting with the upper layers. It has three main functions:

- Resource management including discovery and control of the local resources in the VSE and VNF EE and communication with the upper orchestration layer for notification of the available resources (both the overall resources of the UN at start up and updates due to errors/upgrades and the instantaneous available resources due to deployment of NF-FG). For VNFs that are capable of scaling without deployment of additional instances, the URM will also be involved in this process. The exact interface between the VNF and the URM and the mechanism by which the URM will be able to allocate additional resources to the running VNF are future topics of interest to WP5.

- NF-FG management controlling the whole lifecycle of the NF-FG including deployment, operation, monitoring and removal. The URM translates the NF-FG received as input into a set of fully specified VNFs, including their mapping to specific CPU cores and other local resources, instructions for the VSE to connect the "ports" of the VNFs as specified by the graph and to steer traffic into and through the graph as provided by the flow space specification attached to the graph endpoints and links.
- Bootstrapping the node on the network after the bootstrapping of the Host environment has completed until the UN is available for deployment of NF-FGs (establishing basic connectivity and notifying of its presence so it can be further configured). The main steps are detailed below and its elements are shown in Figure 4.3:
  - Bootloader loads an initial URM and starts execution of the different components. It also loads the physical ID and credentials of the node (pre-configured in the UN) to support AAA processes.
  - Resource Discovery gets the information about the local resources in VSE and VNF EE through the VSE and VNF management modules.
  - Network Configuration provides the mechanism to communicate with the upper level orchestrator.
  - Agent communicates with upper level orchestrator to register the UN, detailing its available resources, retrieves an updated configuration (if required) and determines the communication channel with the upper layer (Controller Adaptation).
  - Once the UN is registered with the upper level orchestrator bootstrapping process is complete and the Local Orchestrator is ready for the deployment of NF-FGs

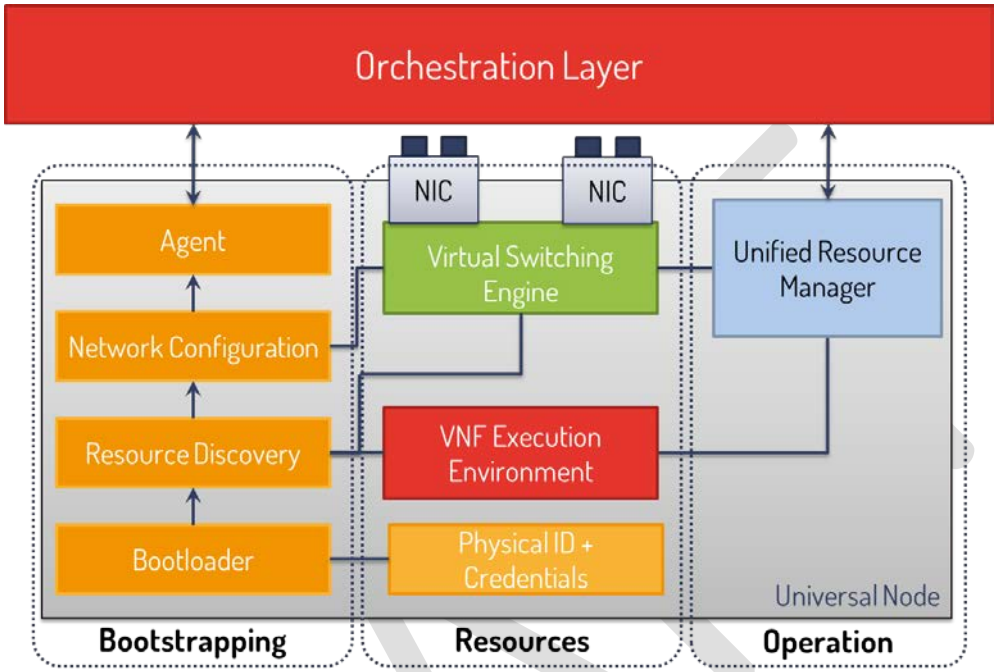

Figure 4.3 – Universal Node Bootstrapping Process Elements

Being in control of both the VNF EE and the VSE, the URM contains dedicated control logic for both of these with an overarching Local Orchestrator that oversees the complete deployment of the graphs:

- **The Local Orchestrator (LO) is the part of the URM that will contain most of the intelligence for performing optimized placement of the VNF threads onto the node CPU topology, taking into account platform specific constraints, communication costs, etc., as exposed by the VNF Management and VSE Management blocks. It is expected that the LO will be able to largely re-use the placement algorithm that will be developed in UNIFY for the upper level orchestrator (WP3). This should be made possible by expressing, from the VNF and VSE Management components, the compute platform topology in terms of CPU cores connected by communication links with specific characteristics (bandwidth, latency), similar to what is done for the external network.**
- **The VNF Management portion of the URM supports multiple virtualization solutions, most likely through the use of the Libvirt library. This module will be responsible for exposing the capabilities of the VNF EE**

towards the Local Orchestrator and for all the operations related to the VNFs depending on the VNF type as described in section 4.2.3.1.

- Similarly, the VSE Management will support multiple back-ends for different Virtual Switching Engine implementations through the definition of a UN internal interface. This module will be responsible for exposing the capabilities of the VSE towards the Local Orchestrator, for instructing the VSE to set up the interconnections between the VNFs according to the deployed NF-FGs and also for the traffic steering of the inbound and outbound flows of the NF-FGs.

Finally, the URM is also responsible for the run-time monitoring of low-level KPIs and to map those, when possible, to the KPI goals specified for the NF FG. It will also provide the required support to allow aggregation and manipulation of the monitoring data and to make the result available to the monitoring solutions being created in WP4, either locally on the node or to remote systems.

4.2.3 VNF Execution Environment

The VNF EE may include multiple types of compute platform virtualization, from simple containers (LXC) to complete hypervisors. This will be largely abstracted to the URM through the use of the Libvirt library. The latter will be enhanced or supplemented where needed to support precise usage of the CPU cores and assignment of virtual ports as decided by the URM and its Local Orchestrator.

4.2.3.1 Various Types of VNF

As already introduced in D5.1, we envision the need for deploying different types of VNFs on the UN. The following are the envisioned types, of which not all will necessarily be supported by the UN prototype, classified based on the execution environment of the VNF:

- VNF Type 1: VNF as a full virtual machine running with the support of a hypervisor. Such virtual machine includes a separate instance of an operating system and supporting libraries, etc.
- VNF Type 2: VNF as an isolated container running on the same operating system kernel as the host. Example: LXC, Docker. This type of VNF presents a reduced footprint but does not provide as much implementation freedom as the type 1.
- VNF Type 3: VNF as an additional process running on the host operating system alongside the other components of the UN such as the virtual switching engine. The only difference with the Type 2 is that no virtualization is used and no additional process isolation is provided on top of what the base operating system offers.
- VNF Type 4: VNF as a plugin to the Virtual Switching Engine. This VNF lives in the same process space as the switching engine and may be loaded using a plugin mechanism or as a permanent part of the engine.

- VNF Type 5: VNF implemented as a switch. This type only exists logically because such VNF relies on normal operation of the switching engine but is likely to be considered as VNF for some services when explicitly specified. An example of such VNF is an encapsulation/decapsulation function provided by the switching engine and invoked as an explicit element of the service being implemented. This type requires no explicit support from the UN but is listed her for completeness. It may however require allocation of additional processing resources to the VSE.

4.2.4 Virtual Switching Engine

The VSE supports the deployment of Logical Switch Instances which are independent switching domains providing traffic isolation. These are then used by the VSE to implement the required traffic steering and the isolation of the NF-FGs, which effectively provides tenant isolation as well. We foresee the following main types of LSIs:

- The Base LSI manages the physical ports and implements the traffic steering into the NF-FGs deployed on the node.
- Then, for each deployed NF-FG, there is an underlying LSI that supports the traffic steering between the VNFs of the graph (service chaining).
- Finally, there may be NFs that can be implemented by instantiating corresponding LSIs and using advanced features of the VSE. For example, some firewalls could be implemented by VSE features.

It is the intention of the UNIFY project to experiment with several implementations of the VSE core which will all plug under the VSE Management part of the Unified Resource Manager. Some of those implementations may also support various back-ends for different specific hardware platforms (x86 with commodity NICs, x86 with accelerators…).

On x86 hardware, the implementations will be based on – or use similar techniques as – the technologies that were presented in 2.4.

Only the Virtual Switching Engine backend can determine how much CPU and memory resources may be required for supporting a specific configuration or specific "advanced" flow mods. The VSE will therefore communicate with the Unified Resource Manager to allocate those resources and make sure the change is reflected in the available resources the node exposes.

5 Universal Node Interfaces

5.1 NF-FG Management Interface

The notations used in this subsection are illustrative only in order to make the description of the content of an NF-FG more concrete. Actual notation to be used in the UNIFY integrated prototype will have to be agreed upon with WP3 at a later stage. The one presented here could be considered as an initial proposal.

5.1.1 Network Function Forwarding Graphs

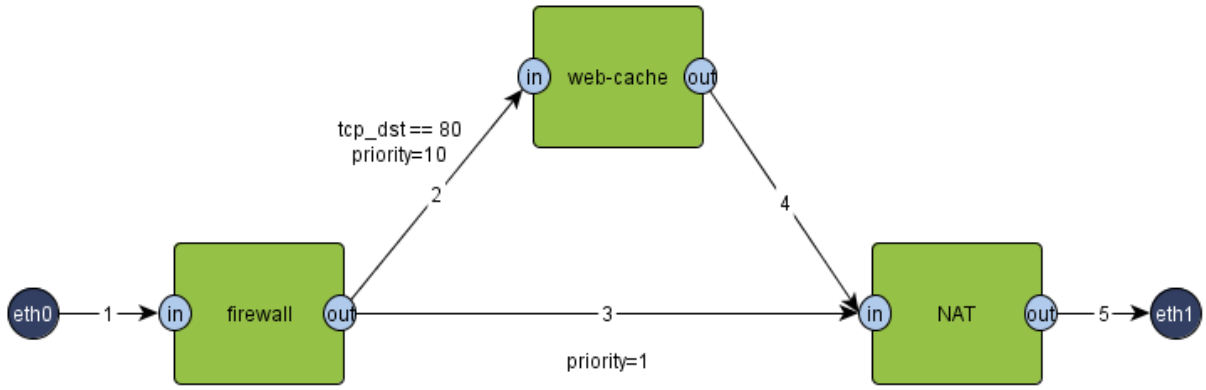

Figure 5.1 – Example NF-FG

An example NF-FG is depicted on Figure 5.1. A possible textual notation for this NF-FG with references to VNF abstract or concrete types and simple flow space specifications is presented below:

```
"flow_graph": {
  "VNFs": [
    {
      "id": "firewall",
"template":    "http://nf_repo.example.com/v1/vnf_specs/0263c411-a3f6-40e4-8e20-52fa238346e8",
"param_supported_by_all_implementations_matching_above_template": 50000
    },  // template defines ports "in" and "out"
    {
      "id": "NAT",
"template":    "http://nf_repo.example.com/v1/vnf_specs/d520c572-8125-4f49-a14f-06a0cc17cc2f",
    },
    {
```

```
      “id”: “web_cache”,
      “template”:  “http://nf_repo.example.com/v1/vnf_specs/bd76a775-4155-4b15-
b901-c5249031e1a3”,
    }
  ],
  "flow-rules": [
    {
      "id": "00000001",
      "match": {
        "port" : "eth0"
      },
      "action": {
        "type" : "output",
        "port": "firewall:in"
      }
    },
    {
      "id": "00000002",
      "priority" : "10",
      "match": {
        "port" : "firewall:out",
        "tcp_dst" : "80"
      },
      "action": {
        "type" : "output",
        "port": "web-cache:in"
      }
    },
    {
      "id": "00000003",
      "priority" : "1",
      "match": {
        "port" : "firewall:out"
      },
      "action": {
        "type" : "output",
```

```
        "port": "NAT:in"
      }
    },
    {
      "id": "00000004",
      "match": {
        "port" : "web-cache:out"
      },
      "action": {
        "type" : "output",
        “port": "NAT:in"
      }
    },
    {
      "id": "00000005",
      "match": {
        "port" : "NAT:out"
      },
      "action": {
        "type" : "output",
        "port": "eth1"
      }
    }
  ]
}
```

Subsequent modifications to the elements of the NF-FG can be achieved by referencing the specific NF-FG instance (see NF-FG ID in subsection 5.1.4) and the named VNFs (“firewall”, “web-cache”, “NAT”) or flow rules it contains.

5.1.2 VNF Specification

As mentioned in subsection 3.1.1, the VNF related information contained in the Network Function Forwarding Graph is enriched as it traverses the different layers of the UNIFY architecture, starting with an abstract definition of the application at the Service layer and ending with a fully characterized and deployable VNF image at the UN. As there could be several VNFs matching the initial abstract definition of the application, the available options must be gradually narrowed down until a specific VNF specification is selected.

The approach followed for detailing the VNF specification is for the upper layers to leave as much freedom as possible to the lower layers to decide the specific implementation, only narrowing it down to exclude those not meeting the defined constraints. Those constraints will mainly originate from the user requirements, translated to KPIs for the components of the NF-FG, and the characteristics of the placement selected by the Orchestrator (be it the one on the Orchestration layer or the Local Orchestrator inside the Unified Resource Manager). This process would be iterative if there are multiple Orchestration layers and will take place again in the UN.

In the UN, most probably all those VNF specifications still available would match the requirements, but as the UN has a finer view of the resource details maybe it could make additional discards. In any case, for the placement decision taken in the UN, a specific implementation will be selected in the Unified Resource Manager from those available options. The specific criteria and algorithm for this final selection will be defined later on the project.

Within the NF-FG, VNFs are described by a reference to the logical VNF template, or abstract VNF type, as well as some concrete values for the template parameters. This is illustrated on Figure 5.2.

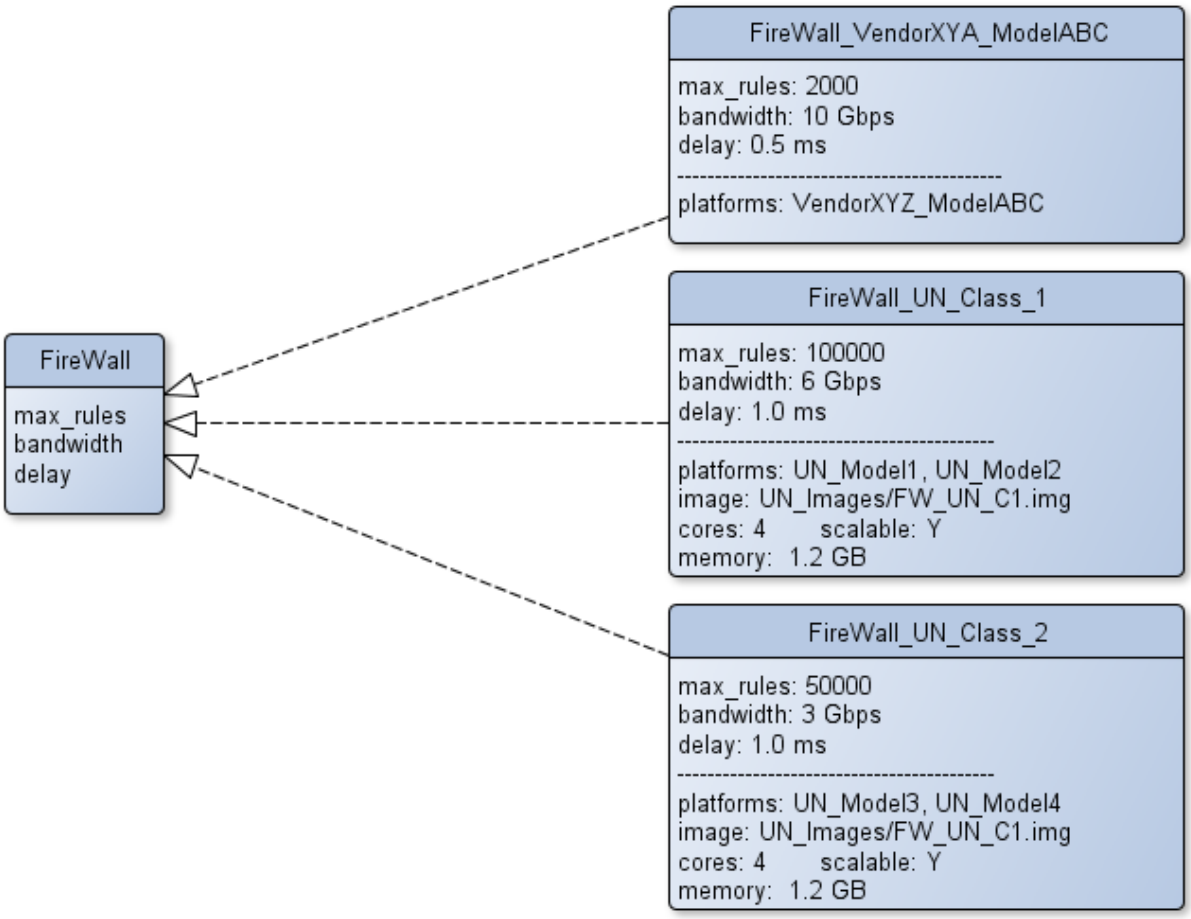

Figure 5.2 – VNF Templates with multiple concrete implementations

The VNF template has a list of template parameters that must be provided when instantiating the VNF. Concrete implementations reference the template that they implement. Note that a concrete VNF does not necessarily have to be an implementation of a template. It could instead be a standalone concrete VNF, in which case it would be referenced directly by the NF-FG without the additional indirection of a template. This is for example the case of a "user provided" VNF.

Each concrete implementation specifies the platforms that it is meant to support as well as potential limits it imposes on the template parameter values. These limits should be checked by the orchestrator before selecting and instantiating the Network Function.

The following elements are also part of the specification of a concrete VNF implementation:

- **VNF Type (See 5.2.3.1 Various Types of VNF)**
- **URI of the VM or code image (as applicable to the VNF Type)**

- Memory size
- Root filesystem size
- Ephemeral filesystem size
- CPU requirements – Different platforms may express this differently. For x86 platforms, the constraints on the CPU cores topology may be specified.
    - Example scheme [socket [(n cores, hyper-threaded=Y/N/NA) …] …] e.g. [[(2, HT=Y), (2, HT=N)], [4, HT=NA]] means: on one socket 2 HT cores and two non-HT cores and, possibly on another socket, 4 cores (HT or not).
- Ports – For each port, the following information is provided:
    - Direction: In, Out or both
    - Type – Defines how the VSE will exchange packets with the VNF. Not all types apply to all VNF types and port types are largely platform specific. The following are example port types when using Intel DPDK for the communication between the VSE and the VNF:
        - KNI – Packet go through the normal IP stack in a Linux VM.
        - IVSHMEM – The application running in the VM instance implementing the VNF uses DPDK to efficiently send and receive packets.
        - Additional port-types may be defined in order to support lightweight virtual machines (e.g., LXC), appropriately modified to implement an optimized communication channel with the virtual switch.
- Load balancing scheme – We envision that some VNFs could benefit from a platform provided load balancing capability that distributes the incoming traffic of a VNF over its multiple packet receiving threads. Whether an application can scale by increasing its number of threads depends on the exact workload and the application design. Some application may also have very specific load balancing schemes that they implement internally instead of making use of this shared platform feature. However, for applications that can use the platform provided mechanism, the VNF specification should indicate on which packet elements the load balancer should key (typically a 5-tuple hash but other schemes could be offered) in order to preserve correct application logic.

  Note: This assumes that the UN also supports a local VNF scaling mechanism whereby the local orchestrator and the application cooperate to dynamically adjust the processing resources assigned to

the VNF and its number of threads as the VNF load evolves, and this without involvement from upper level orchestration.

5.1.3 Flow Space Specification

The flow space specification attached to the endpoints and links of a NF-FG consists in match/action pairs similar to what OpenFlow supports as was illustrated in subsection 5.1.1.

In addition to the support of OpenFlow 1.3 like rules, some extensions would make the switch module a powerful and flexible packet processing engine. The currently envisioned extensions are described in the following subsections.

5.1.3.1 L3 Tunneling Support (PPP, VxLAN, GRE, GTP)

In the current OF specification tunnelling support is not part of the switch, but handled by tunnel ports, that are configured through OF-config. At first glance this seems quite generic, but this is not entirely the case: for L2 OF does have tunnelling support, which is implemented by push, pop and set primitives.

To support the above mentioned tunnels, a generic "push-pop-set bytes at offset" function could be provided. But, due to the limited number of tunnel types, providing a set of tunnel type specific ones could be sufficient. The following new push, pop and set primitives would be required:

PPP

- Push-PPP: Encapsulate the IP packet to a PPP frame and set PPP frame header values according to IP payload (Flag 0x7e, Address 0xff, Control 0x3, Protocol 0x21, Padding if necessary).
- Pop-PPP: Remove PPP framing from the IP packet (header, trailer).
- Set-PPP-FCS: calculates Frame Check Sequence.

PPPoE

- Push-PPPoE: Push a new PPPoE header to the packet and set PPPoE header values according to the PPPoE session phase (Version 0x1, Type 0x1, Code 0x0, Length based on payload). Also set the Ethernet type to PPPoE (0x8864). The PPPoE header is always pushed after the outmost Ethernet header.
- Pop-PPPoE: Pop the PPPoE header from the packet.
- Set-PPPoE-Session-ID: Set the session ID in the PPPoE header.

VxLAN

- Push-VxLAN: Push a new VxLAN header to the packet and set the flag to 0x8. The VxLAN header encapsulates the whole Ethernet frame (meaning that it is pushed before the outmost Ethernet frame).
- Pop-VxLAN: Pop the VxLAN header from the packet.
- Set-VxLAN-VNI: Set the VxLAN network identifier (VNI).

GTP

- Push-GTP: Push a new GTP-U header to the packet and set the header values accordingly (Version 1, PT 1, Type, Length). The GTP header is pushed before the outmost IP header.
- Pop-GTP: Pop the GTP header from the packet.
- Set-GTP-TEID: Set the GTP tunnel endpoint identifier (TEID).

GRE

- Push-GRE: Push a new GRE version 0 header to the packet and set the header values accordingly. GRE can encapsulate L2 and L3 packets as well. The best way is probably to always push the GRE header to the outmost existing header.
- Pop-GRE: Pop the GRE header from the packet.
- Set-GRE-key: Set the security/selector key of the session.

Note that some of the tunnels also require additional external network and transport headers (usually IP, UDP and Ethernet). Most of these are already known at the time of the Push action, so in theory all external headers could be inserted at the same time as the Push-<tunnel> action. But because in an "SDN-enabled" network we probably want to use some non-legacy way of networking, it is suggested to also add generic support for L2 and L3 headers.

Selection between these could be handled by flagging whether we need the "legacy" tunnel, and in this case push all headers to the packet in one step, while if we don't see this flag we go for the generic method.

The appropriate actions are:

IP

- Push-IP: Push a new IP header to the packet and set the header values accordingly (Version 0x4, IHL 0x5, Total length according to payload, copy other parameters – Flags, TTL, Fragment Offset – from internal IP header – if exists)
- Pop-IP: Pop the outmost IP header from the packet
- Set-IP-Dst: Set the destination IP address
- Set-IP-Src: Set the source IP address
- Set-IP-ToS: Set the Type of Service (QoS) field
- Set-IP-Type: Set the type of the IP packet

IPv6

- Push-IPv6: Push a new IPv6 header to the packet and sets the header values accordingly

- Pop-IPv6: Pop the outmost IPv6 header from the packet
- Set-IPv6-Dst: Set the destination IPv6 address
- Set-IPv6-Src: Set the source IPv6 address
- Set-IPv6-TC: Set the Traffic Class (QoS) field
- Set-IPv6-Next-Header: Set the type of the IPv6 packet

UDP

- Push-UDP: Push a new UDP header, sets Length, sets Checksum to zero
- Pop-UDP: Pop the outmost UDP header from the packet
- Set-UDP-SrcPort: Set the source UDP port
- Set-UDP-DstPort: Set the destination UDP port

For the Ethernet header there are already existing actions for setting its field values. For most of the use cases that is sufficient. However for some tunnelling setups (typically L2VPN use cases) we need to push external L2 header, which is not supported by the current standard. The following actions are needed:

- Push-Ethernet: Push a new Ethernet header to the packet, sets the type field
- Pop-Ethernet: Pop the outmost Ethernet header from the packet

In this case an example legacy GTP encapsulation would look like:

1. Push-GTP
2. Set-GTP-TEID (user's TEID on eNodeB)
3. Push-UDP
4. Set-UDP-DstPort (GTP-U well known port - 2152)
5. Push-IP
6. Set-IP-Src (GW Virtual IP address)
7. Set-IP-Dst (eNodeB address) ← this is updated during handover
8. Set-IP-Type (UDP)

Of course in this case if we had the "legacy-flag", we could simplify the action set to:

1. Push-GTP (legacy) = push GTP+UDP+IP, set UDP port and IP type
2. Set-GTP-TEID (user's TEID on eNodeB)
3. Set-IP-Src (GW Virtual IP address)

4. Set-IP-Dst (eNodeB address) ← this is updated during handover

5.1.3.2 Consistent Load Balancing Support

Current OF contains a special Group Table for load balancing support which is called "select" group. This group type executes one bucket in the group, based on a switch-computed selection algorithm (e.g. hash on some user-configured tuple or simple round robin). The actions in the buckets are typically output actions to the destination host or process.

Although the standard doesn't specify the algorithm that is used to select the destination bucket, most implementations use simple round robin mechanism. This is inadequate for most network functions where it is required that packets from the same IP flow follow the same path in the system (consistent load balancing). Implementing consistent load balancer functionality is currently only possible by involving the controller and installing table rules for each L4 flow. This solution has serious scalability problem. Implementing a simple hash + modulo based load balancer as a group seems to be feasible and could solve most of the issues with the current solution.

This is possible according to the current standard. The only missing element is a parameter for the group-mod command where the controller could specify the key that is used by the hash function. We recommend using the already specified ofp_group_mod structure by using 8 bits that is currently defined for padding. This way the structure would look like the following:

```
/* Group setup and teardown (controller -> datapath). */
struct ofp_group_mod {
      struct ofp_header header;
      uint16_t command; /* One of OFPGC_*. */
      uint8_t type; /* One of OFPGT_*. */
      uint8_t key; /* a bitmask instead of pad */
      uint32_t group_id; /* Group identifier. */
      struct ofp_bucket buckets[0];
};
```

The key would flag the different header values as described in the following enum:

```
enum ofp_group_type {
      OFPGK_MPLS_VPN    = 0x01, /* use the MPLS VPN label */
      OFPGK_ETH_ADDR    = 0x02, /* use the src and dst MAC */
      OFPGK_L3_TUN ID   = 0x04, /* use the outmost tunnel ID */
      OFPGK_EXT_IP      = 0x08, /* use the outmost src and dst IP */
      OFPGK_INT_IP      = 0x10, /* use the inner src and dst IP */
      OFPGK_5_TUPLE,    = 0x20, /* use the inner 5 tuple (default) */
      OFPGK_META        = 0x80, /* use the OF metadata */
};
```

This would allow us to define the behaviour of the selection algorithm in a flexible way. The last value would allow total flexibility, since there the hash function would use the metadata that is calculated by the switch during the different lookups. Of course this has additional performance costs, so we would not advise to make this mandatory.

The OFPGK_TUN_ID would use the outmost tunnel's ID that depends on the tunnelling technology. We suggest defining it this way as it is not very usual to use more than one L3 tunnel header on one packet. If this is the case the outmost tunnel header's ID is used. The following IDs must be used if the given tunnel header is present:

- PPPoE: Session ID
- VxLAN: VxLAN VNI
- GTP: GTP TEID
- GRE: GRE key

5.1.4 NF-FG Management Primitives

The UN provides the following primitives to deploy and manage NF FGs:

5.1.4.1 Deploy NF-FG

The request carries the following parameters:

[in]	Graph ID	Identifies the NF-FG instance. This is assigned by the caller (upper orchestrator). The UN must maintain internal mapping.
[in]	Graph Data	Actual NF-FG as described in 0.

The response indicates the outcome of the deployment. It should be noted that the UN may consider that it doesn't have the required resources to deploy the graph.

[out]	Graph ID	Identifies the NF-FG instance.
[out]	Result Code	SUCCESS FAILURE_NO_RESOURCE FAILURE_NO_COMPATIBLE_IMPLEMENTATION FAILURE_IMAGE_NOT_FOUND ...

5.1.4.2 Modify NF-FG

This primitive takes the exact same parameters and has the same response as the Deploy NF-FG primitive. The only difference is that the specified Graph ID must correspond to an already deployed NF-FG.

The modification may fail, for example if not enough resources are available.

5.1.4.3 Delete NF-FG

[in]	Graph ID	Identifies the NF-FG instance to destroy.

5.1.4.4 Get NF-FG List

[out]	Graph IDs List	List containing the Graph IDs of the NF-FGs deployed on the node.

5.1.4.5 Get NF-FG Data

By providing a graph identifier it is possible to retrieve the description of the graph.

[in]	Graph ID	Identifies the NF-FG instance whose description is to be retrieved.
[out]	Graph Data	Actual NF-FG as described in earlier section 5.1.1.

5.2 Resource Management

This interface covers the discovery of resources exposed by the node as well as possible updates to those resources. Such an update may be the result of actions performed through the other interfaces, or from a reconfiguration of the node. It also covers the periodic notification of the available resources due to deployment / undeployment of NF-FGs.

Finding the right level of abstraction for this interface will be one of the upcoming research areas in the UNIFY project. The goal is to avoid exposing too many low-level hardware details while still allowing the upper level orchestrator to make educated choices when placing functions on nodes. This will result in cases where expected capacity will not be met and the orchestrator will have to take corrective actions like deploying additional instances of the function on other nodes, possibly introducing a load-balancer if needed, or simply raising the amount of resources allocated to the function on the node.

5.2.1 Resource Management Primitives

The interface supports the following primitives:

5.2.1.1 Get Node Info and Capabilities

[out]	Total processing capacity	An abstract representation of the total processing capacity of the system taking into account the number of CPU cores, their frequencies, possibly their feature set, etc. This may mean that some sort of calibration for each type of system may be needed but this type of abstraction is required so that the upper orchestrator does not get exposed to details it wouldn't know how to map to application requirements.
[out]	Total memory	Total memory in MB
[out]	Local disk capacity	
[out]	CPU Info	• Architecture, model, vendor • Number of sockets, cores and topology • CPU features
[out]	Platform Tag	A tag that identifies the type of platform (different UN classes for CPE, network edge, etc.) and that can be used e.g. to filter out VNF implementations that don't support the platform.
[out]	Ports List	List of physical ports present on the node with their type or other relevant

		properties.
[out]	Flow space specification capabilities	Similar to OpenFlow capabilities but applies to the features that can be used when specifying the flow spaces within NF-FG.
[out]	Supported VNF Types	Identifies which VNF Types are supported on the node. See 4.2.3.1 for an overview of the types. Where relevant, this information may include additional details like the supported hypervisors, VM container formats, etc.

5.2.1.2 Get Available Resources

[out]	Available Processing Capacity	Remaining processing capacity (see 5.2.1.1) taking into account VNFs already running on the node and current VSE configuration.
[out]	Available Memory	In MB
[out]	Available Local disk capacity	
[out]	Available capacity on ports	Indication of remaining capacity on each physical port, as required by the upper level orchestrator to perform the orchestration task.

5.3 VNF Template and Images Repository Interface

When the UN is instructed to deploy a NF-FG, it needs to fetch the detailed specification and the related binaries of the involved VNFs. This constitutes an outbound interface of the UN towards a central VNF repository.

This interface supports at least the following operations:

- Retrieve the list of possible VNF specifications for a given NF abstract type or template as provided by the upper level orchestrator in the input NF-FG. This is likely in the form of a generic list operation with a filter on the "NF Abstract Type" attribute. The returned list contains an identifier for each VNF specification.
- Fetch a VNF specification given its identifier.
- Fetch a binary (e.g. a Virtual Machine image) that is referenced in a VNF specification.

This interface should provide APIs that allow a loose coupling between client and server, allowing retrieving, uploading and deleting images through simple API calls. An interesting approach is represented by an "Object Storage" component, where NF specifications and related images could be accessed through URIs and the proper set of REST API, leveraging the HTTP protocol and JSON format. Moreover, this APIs should be easily usable by the UN for retrieval of VNF data and by UNIFY management components to populate and manage the VNF repository. A candidate for such a management component is Glance, the OpenStack image manager. However it only knows about VM images and would need to be complemented or extended to cover the full scope of VNF repository management tasks.

In order to match different types of Glance APIs and to have the handiness of an object-like storage, OpenStack proposes its own object storage component, *Swift*. This component is supported by Glance as backend engine, but it also exposes RESTful APIs that can be directly accessed by the Unified Resource Manager, without any indirection. Furthermore, Swift is designed to be horizontally scalable with no-single point of failure.

Swift exposes its resources through a simple and easily discoverable URI scheme (http://docs.openstack.org/api/openstack-object-storage/1.0/content/). For instance, a VNF image URI could be built as https://nf_repo.domain.com/v1/<vnf_guid>/images/<image_guid>/ where nf_repo.domain.com is the URL of the repository, v1 refers to the API version, <vnf_guid> uniquely identifies the VNF unique ID (in existing Swift implementation, this URI component however maps to the "account" in the storage), "images" is a container that regroups different images for the VNF and <image_guid> uniquely identifies the specific image.

It is worth mentioning that OpenStack Swift supports the creation of additional objects that can be associated with the VNF, which enables to store also additional information such as VNF properties. For instance, this could support parameters such as the abstract VNF type, the VM image type (e.g., Xen, KVM), or additional

properties such as the ones needed to instantiate a lightweight virtual machine (e.g., Linux containers or Docker). Finally, the object model supported in Swift does not force programmers to define, *a-priori*, the list of properties that have to be associated to the VNF; any free-form object can be stored and retrieved later, hence offering the flexibility to add/modify the list of needed objects at any time.

It should also be noted that the VNF repository used by the UN is assumed to be shared with the upper level orchestrator(s) within the UNIFY architecture, even if it is clear that the data required by both may only partially overlap. As a result, the interface described here should be seen as an initial proposal that may require extensions or adaptations to serve the needs of the upper level orchestrator as developed in WP3.

The following paragraphs illustrate the operation of a RESTful API that supports a superset of the operations required over the interface and mentioned at the beginning of this subsection.

To retrieve the list of possible VNF specifications for a given abstract NF type, the UN issues a GET request to

> http://nf-repo.domain.com/v1/vnf_specs/
> ?nf_abstract_type=<abstract_nf_uuid_from_nffg>

Additional filtering in the query parameters could specify the target platform that the VNF must support. The UN would set the platform filter based on its own configured Platform Tag. This allows differentiating between NF specifications for UN and for legacy or fixed function devices but also between Universal Nodes of different classes, e.g. UNs that are destined as Customer Premise Equipment from those to be deployed in mini datacentres in the network which would typically have more resources.

In response to the GET request, the repository instance responds with a JSON-encoded list:

```
{"vnf_specs": [
  {"status": "active",
   "name": "Firewall XYZ for UN_CPE",
   "platforms": [ "UN_CPE" ],
   "uri":     "http://nf_repo.example.com/v1/vnf_specs/8a7e417c-a55f-cd38-386d-
f87a36b200de",
   ...},
  ...]}
```

The UN then selects and fetches a specific VNF specification from the list by issuing a GET request for the URI as indicated in the 'uri' field in the previously retrieved list:

> http://nf_repo.example.com/v1/vnf_specs/8a7e417c-a55f-cd38-386d-
> f87a36b200de

The content of a VNF specification was described in 5.1.2. When it references a Virtual Machine image, the VNF specification provides the URI at which the image can be fetched and from which additional meta-data can be retrieved (its size, date of creation, etc.). Example image URI:

```
http://glance.example.com/v1/images/71c675ab-d94f-49cd-a114-e12490b328d9
```

Image metadata is retrieved by issuing a HEAD request for this URI whereas the full image is fetched by issuing a GET request for the same URI.

5.4 Application Control Interface

Once deployed, NFs need to be configured and parameterized at run time. Consequently, there needs to be a control interface to the individual NFs, and this interface is inherently specific to the service provided.

Strictly speaking, this is not a UN interface but rather a NF interface and hence could equally apply to NFs deployed on other infrastructure than UNs. There may however be UN specific aspects to the concretization of such a control interface.

One may define a transport protocol between the NF and a Control Application that is essentially another NF in the NF-FG, but one that plays a different role in the workflow of NF-FG deployment. This transport protocol will be agnostic to the service it controls and may simply be TCP/IP.

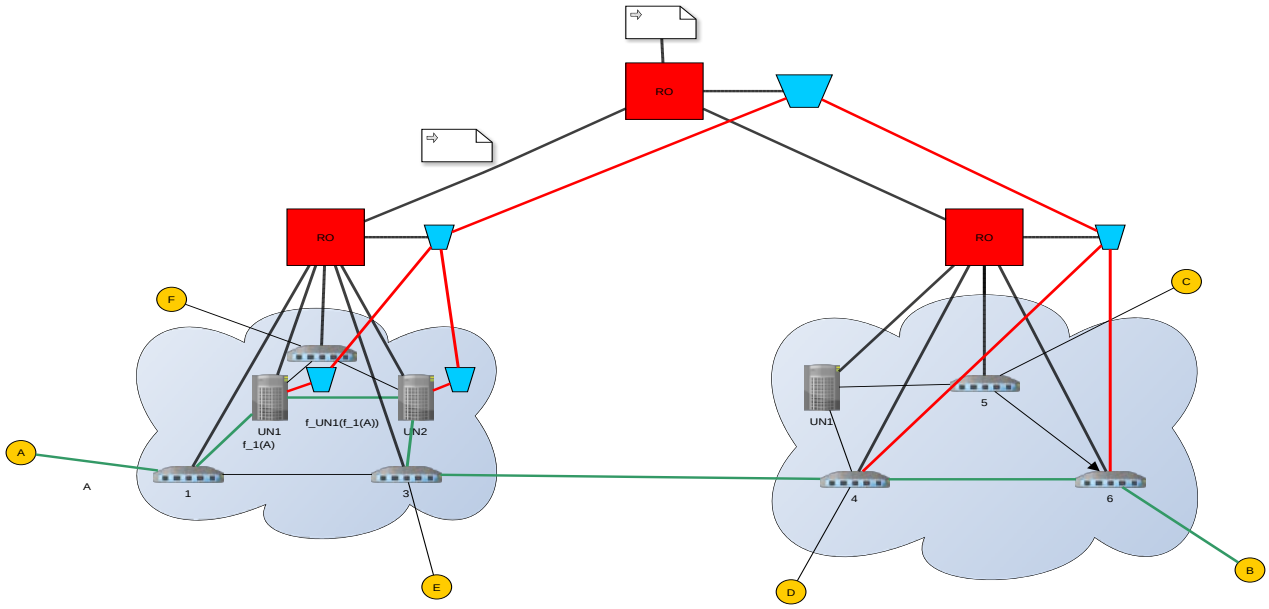

Figure 5.3 - Hierarchical rollout of CtlApps

Figure 5.3 shows a hierarchical rollout of CtlApps (blue trapezoid structures) where the lowest level CtlApps maintain the Application Control Interface (in red) to the VNFs inside the Universal Nodes.

A separation between the Orchestrator that embeds NF-FGs based on topological and resource information and the CtlApp that 'knows' about service specific parameters allows a service-agnostic orchestrator that does not have to be re-programmed for every new service.

6 Conclusion

Based on the requirements for the Universal Node data plane that were detailed in D5.1, an architecture and external interfaces of the UN have been proposed. Based on these interfaces, WP3 can implement the orchestration functions that are of interest to their research. The definition of the architecture puts WP5 in a position to start implementing prototypes, or testing existing solutions, for the various components that have been defined.

Some aspects will however require further attention as the work of the different work packages progresses:

- Resource usage reporting and the corresponding abstractions is a topic that requires further study. This document makes it clear which resources need to be reported and through which interface this reporting happens but, for some type of resources (networking), does not provide the specific abstractions that can be used to allow the upper level orchestrator to perform its placement tasks without exposing it to too many details or to measurements it wouldn't know how to interpret. This future work will also depend on the approach WP3 takes for representing similar resources in other infrastructure elements than Universal Nodes.
- Details of how control interfaces of the VNFs are returned or set up between VNFs was only briefly eschewed (see section 5.4) and requires additional investigations which will need to be coordinated with WP3 as these topics are possibly also relevant to the orchestration of VNFs in data centres. The concrete realization of these control interfaces, specifically on UNs, will also need to be further detailed. This will be covered in the future deliverable D5.3.
- Monitoring requirements (WP4) haven't really been addressed in any detail yet. This is mostly related to the fact that the work so far has focused on the fundamental elements of the UN platform and that we do not foresee particular problems in adding monitoring aspects to the external interfaces of the UN and its internal architecture. Performance implications will of course have to be evaluated during the prototyping phase.
- Sharing of VNFs between multiple NF-FGs for VNFs that support multiple independent "control" interfaces per VNF instance (multi-tenant capable applications) has not been addressed yet. It should however be fairly straightforward to extend the proposed NF-FG description to allow reference to VNFs already deployed as part of another graph.

List of abbreviations and acronyms

Abbreviation	Meaning
API	Application Programming Interface
CP	Control Plane
DP	Data Plane
GRE	Generic Routing Encapsulation
GTP	GPRS Tunneling Protocol
KPI	Key Performance Indicator
LSI	Logical Switch Instance
LXC	LinuX Containers
NF	Network Function
NF-FG	Network Function Forwarding graph
NIC	Network Interface Card (often refers to a physical network port regardless of its presence on a card)
NUMA	Non-Uniform Memory Access
OF	OpenFlow
OVS	Open vSwitch
PPP	Point to Point Protocol
REST	Representational State Transfer
SAP	Service Attachment Point
UN	Universal Node
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URM	Unified Resource Manager
VM	Virtual Machine
VNF	Virtual Network Function
VNF EE	VNF Execution Environment
VSE	Virtual Switching Engine
VLAN	Virtual LAN
VXLAN	Virtual Extensible LAN

References

[1] "OpenStack," [Online]. Available: https://www.openstack.org/. [Accessed 13 February 2014].

[2] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of USENIX Annual Technical Conference*, Jun. 2005.

[3] [Online]. Available: http://www.qemu.org/. [Accessed 30 April 2014].

[4] [Online]. Available: http://www.xenproject.org/. [Accessed 30 April 2014].

[5] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange and C. A. F. De Rose, "Performance Evaluation of Container-based Virtualization for High Performance Computing Environments," Pontifical Catholic University of Rio Grande.

[6] [Online]. Available: http://criu.org. [Accessed 5 May 2014].

[7] [Online]. Available: https://wiki.openstack.org/wiki/HypervisorSupportMatrix. [Accessed 5 May 2014].

[8] [Online]. Available: http://blog.docker.io/2013/08/getting-to-docker-1-0/. [Accessed 20 May 2014].

[9] [Online]. Available: http://blog.docker.io/2014/03/docker-will-be-in-openstack-icehouse/. [Accessed 20 May 2014].

[10] [Online]. Available: http://blog.docker.io/2014/04/openstack-update-icehouse-release-update. [Accessed 20 May 2014].

[11] "libvirt - The virtualization API," [Online]. Available: http://libvirt.org. [Accessed 24 06 2014].

[12] [Online]. Available: http://git.openvswitch.org/cgi-bin/gitweb.cgi?p=openvswitch;a=blob;f=INSTALL.DPDK;hb=HEAD. [Accessed 20 May 2014].

[13] L. Rizzo, "netmap - the fast packet I/O framework," [Online]. Available: http://info.iet.unipi.it/~luigi/netmap/. [Accessed 26 June 2014].

[14] "PF_RING DNA - Direct NIC Access," [Online]. Available: http://www.ntop.org/products/pf_ring/dna/. [Accessed 25 June 2014].

[15] "Open vSwitch," [Online]. Available: http://openvswitch.org/. [Accessed 22 April 2014].

[16] "Intel® DPDK vSwitch," [Online]. Available: https://github.com/01org/dpdk-ovs. [Accessed 22 April 2014].

[17] BISDN, "The eXtensible DataPath Daemon (xDPd)," [Online]. Available: http://www.xdpd.org/. [Accessed 23 May 2014].

[18] Open Networking Foundation, "OpenFlow Switch Specification v1.3.3," 13 September 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.3.pdf. [Accessed 10 February 2014].

[19] "RFC 3746," [Online]. Available: http://tools.ietf.org/html/rfc3746. [Accessed 25 June 2014].

[20] Open Networking Foundation, "OpenFlow Configuration and Management Protocol OF-CONFIG 1.0," [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config1dot0-final.pdf. [Accessed 20 May 2014].

[21] " RFC 6241," [Online]. Available: http://tools.ietf.org/html/rfc6241. [Accessed 25 August 2014].

[22] "RFC 6020," [Online]. Available: http://tools.ietf.org/html/rfc6020. [Accessed 25 August 2014].

[23] "RFC7047," [Online]. Available: http://tools.ietf.org/html/rfc7047. [Accessed 25 August 2014].

[24] "Open vSwitch," [Online]. Available: http://openvswitch.org/. [Accessed 13 February 2014].