unifying cloud
and carrier networks

Deliverable 5.5: Universal Node Prototype

Dissemination level	PU
Version	0.1
Due date	31.03.2015
Version date	09.11.2015

This project is co-funded
by the European Union

Document information

Editors

Ivano Cerrato (POLITO)

Contributors

Ivano Cerrato (POLITO), Fulvio Risso (POLITO), David Verbeiren (INTEL), Gergely Pongrácz (ETH), Hagen Woesner (BISDN)

Reviewers

András Császár (ETH), Tobias Steinicke (TP)

Coordinator

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

Project funding

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

Legal Disclaimer

Summary

This document illustrates the Universal Node prototype, as defined in Deliverable D5.5, which implements the architecture defined in Deliverable D5.2 and includes, as main components, the local orchestrator with different computing and network plugins, the softswitch, and the northbound interface implementing the NF-FG as defined in WP3.
The UN architecture is modular and supports different execution environments and different softswitches. This enables to exploit different technologies and to adapt to the necessity of different use cases.
The UN will provide a foundation platform for the development of the other components foreseen in the whole UNIFY architecture.

Contents

List of Figures

1 Introduction

The Universal Node is a physical machine (e.g., an high volume standard server, a CPE) that executes several software components: a Universal Node orchestrator, a virtual switching engine, one or more virtual execution environments, and a variable number of Virtual Network Functions executed within the virtual execution environments. The prototype described in this deliverable implements the Universal Node orchestrator and the so called Name Resolver, a simple module defined in WP5 and implementing a simplified version of the Network Function Information Base (NFIB) used for validating the prototype. Moreover, in order to test the functionalities of the orchestrator, we also implemented some simple Virtual Network Functions for the different virtual execution environments.

1.1 Universal Node Orchestrator

The **Universal Node orchestrator (un-orchestrator)** is the main component of the Universal Node (UN). It handles the orchestration of compute and network resources within a UN, hence managing the complete lifecycle of computing containers (e.g., VMs, Docker, DPDK processes) and networking primitives (e.g., OpenFlow rules, logical switching instances, etc). It receives commands through a REST API according to the Network Functions Forwarding Graph (NF-FG) formalism, and takes care of implementing them on the physical node.
More in detail, when it receives a command to deploy a new NF-FG, it does all the operations required to actually implement the forwarding graph:

- retrieve the most appropriate image for the selected Virtual Network Function (VNF);
- configure the virtual switch (vSwitch) to create a new logical switching instance (LSI) and the ports required to connect it to the VNF to be deployed;
- deploy and start the VNF;
- translate the rules to steer the traffic into OpenFlow flowmod messages to be sent to the vSwitch (some flowmod are sent to the new LSI, others to the `LSI-0`, i.e. an LSI that steers the traffic into the proper graph.)

Similarly, the un-orchestrator takes care of updating or destroying a graph when the proper messages are received.
As depicted in Figure 1, which provides an overall view of the UN, the un-orchestrator includes several modules; the most important ones are the network controller and the compute controller, which are exploited to interact respectively with the vSwitch and the hypervisor(s). These two modules are detailed in the following sections.

1.1.1 The network controller

The network controller is the sub-module that interacts with the vSwitch. It consists of two parts:

- the *Openflow controller(s)*: a new Openflow controller is created for each new LSI, which is used to steer the traffic among the ports of the LSI itself;

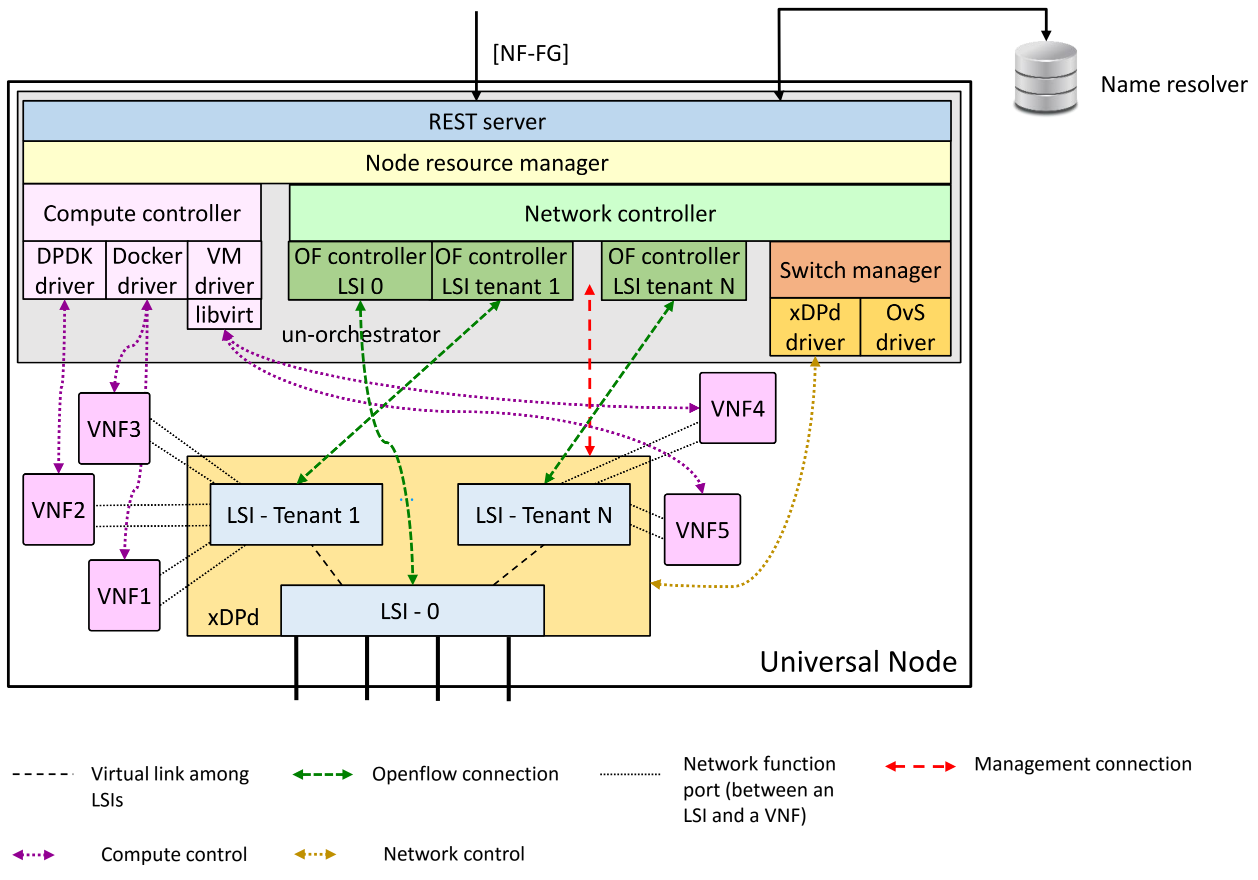


Figure 1: Universal Node architecture overview.

- the *switch manager*: it is used to create/destroy LSIs and virtual ports. In practice, it allows the un-orchestrator to interact with the vSwitch in order to perform management operations. Each virtual switch implementation (e.g., xDPd, OvS) may require a different implementation for the switch manager, according to the API exported by the vSwitch itself.

Currently, the un-orchestrator supports Open vSwitch (OvS), the extensible DataPath daemon (xDPd) and the Ericsson research flow switch (ERFS) as vSwitches, although further vSwitches can be supported by writing a module implementing a proper API.
Note that, according to Figure 1, in the boot phase the network controller creates a first LSI (called `LSI-0`) that is connected to the physical interfaces and that will be connected to several other LSIs. Each one of these additional LSIs corresponds to a different NF-FG; hence, it is connected to the VNFs of such a NF-FG, and takes care of steering the traffic among them as required by the graph description. Instead the `LSI-0`, being the only one connected to the physical interfaces of the UN and to all the other graphs, dispatches the traffic entering into the node to the proper graph, and properly handles the packets already processed in a graph.

1.1.2 The compute controller

The compute controller is the sub-module that interacts with the virtual execution environment(s) (i.e., the hypervisor) and handles the lifecycle of a Virtual Network Function (i.e., creating, updating, destroying a VNF), including the operations needed to attach VNF ports already created on the vSwitch, to the VNF itself. Each execution environment may require a different implementation for the compute controller, according to the commands supported by the hypervisor itself.
Currently the prototype supports virtual network functions as (KVM) VMs, Docker and DPDK processes, although only a subset of them can be available depending on the chosen vSwitch. Also in this case, further execution environments can be supported through the implementation of a proper API.

1.1.3 Compute and network controllers: supported combinations

Table 1 shows the execution environments that are supported with the different vSwitches.

Table 1: Supported combination of compute and network controllers

	Docker	**KVM**	**KVM-DPDK (IVSHMEM)**	**DPDK processes**
xDPd-DPDK	Y*	Y*	N	Y
OvS	Y*	Y	No (requires OvS-DPDK)	(requires OvS-DPDK)
OvS-DPDK	Y	Y	Y	Y
ERFS	Y*	Y*	Y	Y

* In this case the packet exchange between the virtual switch and the execution environment is not optimized.

1.2 Name Resolver

The Name Resolver is a module that returns a set of implementations for a given NF. It is exploited by the un-orchestrator each time that a NF must be started in order to translate the “abstract” name (e.g., `firewall`) into the proper suitable software image (e.g., `firewal_vmimage_abc`).

1.3 Repository

The most updated code of the Universal Node orchestrator and the Name Resolver, together with some example VNFs, can be downloaded from `https://github.com/netgroup-polito/un-orchestrator/`.

2 How to compile the Universal Node components

In order to execute the un-orchestrator, we need to setup different components, namely:

- a set of libraries needed to compile the un-orchestrator code;
- a virtual switch (either xDPd, ERFS or OpenvSwitch) as a base switch for the platform;
- one or more execution environments for Virtual Network Functions, e.g., KVM for executing VM, Docker, or other;
- the Name Resolver.

2.1 Required libraries

Several libraries are required to compile the un-orchestrator. In the following we list the steps required on an **Ubuntu 14.04**[1].

```
; Install required libraries
; - build-essential: it includes GCC, basic libraries, etc
; - cmake: to create cross-platform makefiles
; - cmake-curses-gui: nice 'gui' to edit cmake files
; - libboost-all-dev: nice c++ library with tons of useful functions
; - libmicrohttpd-dev: embedded micro http server
; - libxml2-dev: nice library to parse and create xml
$ sudo apt-get install build-essential cmake cmake-curses-gui \
      libboost-all-dev libmicrohttpd-dev libxml2-dev

; Download JSON Spirit (nice library to parse JSON files)
$ git clone https://github.com/sirikata/json-spirit
$ cd json-spirit/
; Now install the above library according to the description provided
; in the cloned folder

; Download ROFL-common (library to parse OpenFlow messages)
$ git clone https://github.com/bisdn/rofl-common
$ cd rofl-common/
$ git checkout stable-0.6
; Now install the above library according to the description provided
; in the cloned folder
```

2.2 Install the proper virtual switch

According to Section 1.1.3, the un-orchestrator supports different types of virtual switches. You have to install the one that you want to use, choosing from the possibilities listed in this section.

[1]Note that we never tested the Universal Node components on a different operating system.

2.2.1 xDPd with DPDK support

In order to install xDPd with DPDK support, you have to follow the steps below.

```
$ git clone https://github.com/bisdn/xdpd
$ cd xdpd/

; Install all the libraries required by the README provided in this folder
$ bash autogen
$ cd build
$ ../configure --with-hw-support=gnu-linux-dpdk \
      --with-plugins="node_orchestrator rest"
$ make
$ sudo make install
```

It is worth noting that xDPd is not compiling on Linux kernels newer than `3.16.0-30`.

2.2.2 ERFS with DPDK support

In order to install ERFS with DPDK support, you have to follow the steps below.

```
$ svn checkout <Unify SVN repo>
$ cd <WP5 / Task 5.2 / code / erfs>

; Make sure you have DPDK with ivshmem target installed
; Also make sure you have a proper C++ build environment with boost
$ ./makeconf
$ ./configure --enable-debug=[yes/no]
$ make
```

It is worth noting that performance-wise there is some drop on Xeon between DPDK 1.7 and later versions due to the change in the mbuf structure.

2.2.3 OpenvSwitch (of-config) [DEPRECATED]

OpenvSwitch can be installed with either the OVSDB or OF-CONFIG plugin. Although both protocols allow to control the vSwitch (e.g., create/delete new bridging instances, create/delete ports, etc.), we found out that OF-CONFIG, although being standard, is rather limited in terms of capabilities. For instance, it cannot set the type of port configured on the switch (e.g., `virtio` or `ivshmem`), requiring the orchestrator to rely on a combination of OF-CONFIG commands and bash scripts to perform its job. For this reason we suggest to install OpenvSwitch with its native OSVDB support (next section); although OVSDB is not standard, it seems that it does its job better than OF-CONFIG. In any case, the compilation instruction for setting up OpenvSwitch with OF-CONFIG are the following:

```
$ sudo apt-get install autoconf automake gcc libtool libxml2 \
      libxml2-dev m4 make openssl dbus
```

```
; Download LIBSSH from:
; https://red.libssh.org/projects/libssh/files
; Now install the library following the INSTALL file provided
; in the root directory

; Clone the libnetconf repository
$ git clone https://github.com/cesnet/libnetconf
$ cd libnetconf/
$ git checkout -b 0.9.x origin/0.9.x

; Install the libnetconf library according to the descrition provided
; in the cloned folder.

; Download OpenvSwitch from
; http://openvswitch.org/releases/openvswitch-2.4.0.tar.gz
$ tar -xf openvswitch-2.4.0.tar.gz
$ cd openvswitch-2.4.0
$ ./configure --prefix=/ --datarootdir=/usr/share \
      --with-linux=/lib/modules/$(uname -r)/build
$ make
$ sudo make install

; Clone the of-config repository
$ git clone https://github.com/openvswitch/of-config
; Install of-config according to the description provided
; in the cloned folder.
```

2.2.4 OpenvSwitch (OVSDB)

At first, download the OpenvSwitch source code from:

```
http://openvswitch.org/releases/openvswitch-2.4.0.tar.gz
```

Then execute the following commands:

```
$ tar -xf openvswitch-2.4.0.tar.gz
$ cd openvswitch-2.4.0
$ ./configure --prefix=/ --datarootdir=/usr/share \
      --with-linux=/lib/modules/$(uname -r)/build
$ make
$ sudo make install
```

2.2.5 OpenvSwitch (OVSDB) with DPDK support

Before installing OvS with DPDK, you must download and compile the DPDK library. At first, download the source code from:

```
http://dpdk.org/browse/dpdk/snapshot/dpdk-2.1.0.tar.gz
```

Then execute the following commands:

```
$ tar -xf dpdk-2.1.0.tar.gz
$ cd dpdk-2.1.0
$ export DPDK_DIR=‘pwd‘
; modify the file ‘$DPDK_DIR/config/common_linuxapp‘ so that
; ‘CONFIG_RTE_BUILD_COMBINE_LIBS=y‘
; ‘CONFIG_RTE_LIBRTE_VHOST=y‘
```

To compile OvS with the DPDK support, execute:

```
$ make install T=x86_64-ivshmem-linuxapp-gcc
$ export DPDK_BUILD=$DPDK_DIR/x86_64-ivshmem-linuxapp-gcc/
```

Now, download the OpenvSwitch source code:

```
$ git clone https://github.com/openvswitch/ovs
```

Then execute the following commands:

```
$ cd ovs
$ ./boot.sh
$ ./configure --with-dpdk=$DPDK_BUILD
$ make
$ sudo make install
```

Now create the ovsdb database:

```
$ mkdir -p /usr/local/etc/openvswitch
$ mkdir -p /usr/local/var/run/openvswitch
$ rm /usr/local/etc/openvswitch/conf.db
$ sudo ovsdb-tool create /usr/local/etc/openvswitch/conf.db \
      /usr/local/share/openvswitch/vswitch.ovsschema
```

2.3 Install the Virtual Execution Environment(s) for Virtual Network Functions

The current un-orchestrator supports different types of execution environments. You have to install the ones that you want to use, selecting one or more possibilities from the ones listed in this section.

2.3.1 Docker

In order to support the Docker execution environment, follow the instructions provided here:

```
http://docs.docker.com/installation/
```

2.3.2 QEMU/KVM/Libvirt

This is needed in order to run VNFs in KVM-based virtual machines. Two flavors of virtual machines are supported:

- virtual machines that exchange packets with the vSwitch through the `virtio` driver. This configuration allows you to run both traditional processes and DPDK-based processes within the virtual machines. In this case, the host backend for the virtual NICs is implemented through `vhost` in case OvS and xDPd as vSwitches, and through `vhost-user` when OvS-DPDK is used as vSwitch;
- virtual machines that exchange packets with the vSwitch through shared memory (`ivshmem`). This configuration is oriented to performance, and only supports DPDK-based processes within the virtual machine.

Standard QEMU/KVM (without `ivshmem` support)
To install the standard QEMU/KVM execution environment, execute the following command:

```
$ sudo apt-get install libvirt-dev qemu-kvm libvirt-bin \
      bridge-utils qemu-system
```

If you intend to use (DPDK) `vhost-user` ports, a recent version of Libvirt must be used that supports configuration of this type of ports. You can build it from sources using the following commands:

```
$ sudo apt-get install libxml-xpath-perl libyajl-dev libdevmapper-dev \
      libpciaccess-dev libnl-dev
$ git clone git://libvirt.org/libvirt.git
; select the commit that is known to work and have the necessary support
$ cd libvirt
$ git checkout f57842ecfda1ece8c59718e62464e17f75a27062
$ ./autogen.sh
$ make
$ sudo make install
```

In case you already had libvirt installed on the system, this will install an alternative version which must then be used instead of the default one:

```
; Stop any running Libvirtd instance and run the alternative version
; just installed:
$ sudo service libvirt-bin stop
$ sudo /usr/local/sbin/libvirtd --daemon
```

Similarly, if you use virsh, you'd have to use the version from `/usr/local/bin`.

QEMU with `ivshmem` support

To compile and install the QEMU/KVM execution environment with the support to `ivshmem`, further steps are required (in addition to the command listed above):

```
$ git clone https://github.com/01org/dpdk-ovs
$ cd dpdk-ovs/qemu
$ mkdir -p bin/
$ cd bin
$ sudo apt-get install git libglib2.0-dev libfdt-dev \
      libpixman-1-dev zlib1g-dev
$ ../configure
$ make
$ sudo make install
```

2.3.3 DPDK processes

In order to run VNFs implemented as DPDK processes, no further operations are required, since the DPDK library has already been installed together with the vSwitch.

2.4 Compile the Name Resolver

In order to compile the name resolver, execute the following commands:

```
$ cd [un-orchestrator]/name-resolver

; Choose among possible compilation options
$ ccmake .
```

The previous command allows you to select some configuration parameters for the name-resolver, such as the logging level. When you're finished, exit from the `ccmake` interface by generating the configuration files (press '`c`' and '`g`') and type the following commands:

```
; Create makefile scripts based on the previously selected options
$ cmake .

; Compile and create the executable
$ make
```

2.5 NF-FG library

This step is required to compile the Network Functions - Forwarding Graph (NF-FG) (defined in WP3), which is based on the concept of *virtualizer*. In addition, the Universal Node supports also an U-SI interface that was defined before the official NF-FG specification, and that is kept to allow further experimentations in the UN. More details about this additional interface will be provided in Section 4.

```
; Retrieve the NF-FG library.
$ cd [un-orchestrator]
$ git submodule update --init --recursive
```

2.6 Compile the un-orchestrator

We are now ready to compile the un-orchestrator.

```
$ cd orchestrator

; Choose among possible compilation options
$ ccmake .
```

The previous command allows you to select some configuration parameters for the un-orchestrator, such as the virtual switch used, which kind of execution environment(s) you want to enable, the NF-FG format to use (the default WP5 one or the one defined in WP3), etc. When you are finished, exit from the `ccmake` interface by *generating the configuration files* (press '`c`' and '`g`') and type the following commands:

```
; Create makefile scripts based on the previously selected options
$ cmake .

; Compile and create the executable
$ make
```

3 How to execute the un-orchestrator

The full list of command line parameters for the un-orchestrator can be retrieved by the following command:

```
$ sudo ./node-orchestrator --h
```

Please refer to the help provided by the un-orchestrator itself in order to understand how to use the different options.
The un-orchestrator requires a virtual switch up and running in the server, which is completely independent from this software. Therefore you need to start your preferred vSwitch first, before running the un-orchestrator. Proper instructions for xDPd and OpenvSwich are provided below. Similarly, the un-orchestrator requires that the Name Resolver is already running; the instructions to execute this component are provided in the following.

3.0.1 Configuration file examples

Folder `[un-orchestrator]/orchestrator/config` contains some configuration file examples that can be used to configure/test the un-orchestrator.

- `config/universal-node-example.xml`: configuration file describing the physical ports to be handled by the un-orchestrator, as well as the amount of CPU, memory and storage provided to the Universal Node;
- `config/simple_passthrough_nffg.json`: simple graph that implements a passthrough function, i.e., traffic is received from a first physical port and sent out from a second physical port, after having been handled to the vswitch. This graph is written according to the NF-FG definition supported through the U-Sl interface;
- `config/passthrough_with_vnf_nffg.json`: graph that includes a VNF. Traffic is received from a first physical port, provided to a network function, and then sent out from a second physical port. This graph is written according to the NF-FG definition supported through the U-Sl interface.

The same graphs of `config/simple_passthrough_nffg.json` and `config/passthrough_-with_vnf_nffg.json` are described using the WP3 definition based on the *virtualizer* in files `config/virtualizer/simple_passthrough_nffg.xml` and `config/virtualizer/passthrough_with_vnf_nffg.xml`.

3.1 How to start the proper virtual switch

As stated above, the proper vSwitch must be started on the Universal Node before the boot of the un-orchestrator; in the following the instructions to run the supported vSwitches are provided.

3.1.1 How to start xDPd with DPDK support to work with the un-orchestrator

Set up DPDK (after each reboot of the physical machine), in order to:

- Build the environment `x86_64-native-linuxapp-gcc`;
- Insert `IGB UIO` module;
- Insert `KNI` module;
- Setup hugepage mappings for non-NUMA systems (1000 should be a reasonable number);
- Bind Ethernet devices to IGB UIO module (bind all the Ethernet interfaces that you want to use).

```
$ cd [xdpd]/libs/dpdk/tools
$ sudo ./setup.sh
; Follow the instructions provided in the script
```

Start xDPd:

```
$ cd [xdpd]/build/src/xdpd
$ sudo ./xdpd
```

xDPd comes with a command line tool called `xcli`, that can be used to check the flows installed in the LSIs, which are the LSIs deployed, see statistics on flows matched, and so on. The `xcli` can be run by just typing:

```
$ xcli
```

3.1.2 How to start ERFS with DPDK support to work with the un-orchestrator

Set up DPDK (after each reboot of the physical machine), in order to:

- Insert `IGB UIO` kernel module;
- Set up huge page filesystem;
- Bind Ethernet devices to IGB UIO module (bind all the Ethernet interfaces that you want to use).

Start ERFS:

```
$ cd [erfs]
$ sudo ./dof [dpdk parameters] -- [erfs parameters]
; example: ./dof -c 0xe -n 2 -- -p 6633 -c example.cfg -C 1
```

Here dpdk parameters are the usual: core mask or list, memory channels, socket memories, etc. ERFS parameters can be seen in the README file. There are no mandatory parameters.
ERFS comes with a run-time configuration interface and also with configuration file support. Configuration commands can be sent to the switch either on this TCP port (default control port number - 1 = 16632), or by using a file, and adding the "-c <filename>" to the command line.

```
add-switch dpid=<dpid>

      Create a new switch with the specified data path id. A new control
      port is also opened for it, starting from port number specified on
      the command line (or 16633).

add-port dpid=<dpid> port-num=<OF pnum> PCI:x:y.z [rx-queues=<q>]

      Add a physical port with the specified PCI id to a switch.
      Specifying the number of RX queues is optional (default = 1).

add-port dpid=<dpid> port-num=<OF pnum> XSWITCH

      Add an "inter-switch" port to a switch. Using this port
      packets can be sent between two switches. See the "connect" command.
      The name of the port will be: XSWITCH:<dpid>-<OF pnum>

add-port dpid=<dpid> port-num=<OF pnum> KNI:<iface_id> socket=<s>

      Add a KNI port to a switch. <iface_id> should be a unique number,
      as the interface name the kernel will see is: "kni<iface_id>"
      <s> is the NUMA socket number to be used for the queues of the port.

add-port dpid=<dpid> port-num=<OF pnum> IVSHMEM socket=<s>

      Add an IVSHMEM port to a switch. The port name will be:
      IVSHMEM:<dpid>-<OF pnum>. IVSHMEM groups should be grouped, see
      "group-ivshmems".
      <s> is the NUMA socket number to be used for the queues of the port.

add-port dpid=<dpid> port-num=<OF pnum> DEFRAG buckets=<b>

      Add an IP defragmenter virtual port to the switch.
      The port name will be: DEFRAG:<dpid>-<OF pnum>
      Packets sent to the port will reappear in table 0 with their
      original inport, and metadata set to 1.

connect XSWITCH:<dpid1>-<num1> XSWITCH:<dpid2>-<num2>

      Connect two inter switch ports. Packets sent to XSWITCH:<dpid1>-<num1>
      will appear as input on XSWITCH:<dpid2>-<num2>. The lcore which
      processed the packet in the dpid1 switch continues processing in the
      dpid2 switch too.

group-ivshmems <metadata_name> IVSHMEM:<dpid>-<num1> IVSHMEM:<dpid>-<num2> ...

      Group a set of IVSHMEM ports to be used by a single VM. This command
      generates the necessary command line for Qemu in a file:
      /tmp/ivshmem_qemu_cmdline_<metadata_name>

lcore <lcore> PCI:x:y.z[/queue] | IVSHMEM:<dpid>-<num> | KNI:<num>

      Specify which ports/queues an lcore should read.

defrag-stat dpid=<dpid> port-num=<OF pnum>

      Returns detailed statistics from the specified defragmenter port.

plugin <path_of_shared_library>

      Loads a shared library implementing experimental actions/instructions.
```

3.1.3 How to start OvS (managed through OFCONFIG) to work with the un-orchestrator [DEPRECATED]

Start OvS:

```
$ sudo /usr/share/openvswitch/scripts/ovs-ctl start
```

In addition, you have to start the OF-CONFIG server, which represents the daemon the implements the protocol used to configure the switch.
OF-CONFIG server can be started by:

```
$ sudo ofc-server
```

By default, `ofc-server` starts in daemon mode. To avoid daemon mode, use the `-f` parameter. For the full list of the supported parameters, type:

```
$ ofc-server -h
```

3.1.4 How to start OvS (managed through OVSDB) to work with the un-orchestrator

Start OVS:

```
$ sudo /usr/share/openvswitch/scripts/ovs-ctl start
```

Start ovsdb-server:

```
$ sudo ovs-appctl -t ovsdb-server ovsdb-server/add-remote ptcp:6632
```

3.1.5 How to start OvS (managed through OVSDB) with DPDK support to work with the un-orchestrator

Configure the system (after each reboot of the physical machine):

```
$ sudo su
; Set the huge pages of 2MB; 4096 huge pages should be reasonable.
$ echo 4096 > /proc/sys/vm/nr_hugepages

; Umount previous hugepages dir
$ umount /dev/hugepages
$ rm -r /dev/hugepages

; Mount huge pages directory
$ mkdir /dev/hugepages
$ mount -t hugetlbfs nodev /dev/hugepages
```

Set up DPDK (after each reboot of the physical machine):

```
$ sudo modprobe uio
$ sudo insmod [dpdk-folder]/x86_64-ivshmem-linuxapp-gcc/kmod/igb_uio.ko
; Bind the physical network device to ‘igb_uio‘. The following row
; shows how to bind eth1. Repeat the command for each network interface
; you want to bind.
$ [dpdk-folder]/tools/dpdk_nic_bind.py --bind=igb_uio eth1
```

Start ovsdb-server:

```
$ sudo ovsdb-server --remote=punix:/usr/local/var/run/openvswitch/db.sock \
      --remote=db:Open_vSwitch,Open_vSwitch,manager_options \
      --pidfile --detach
```

The first time after the OVSDB database creation, initialize it:

```
$ sudo ovs-vsctl --no-wait init
```

Start the switching daemon:

```
$ export DB_SOCK=/usr/local/var/run/openvswitch/db.sock
$ sudo ovs-vswitchd --dpdk -c 0x1 -n 4 --socket-mem 1024,0 \
      -- unix:$DB_SOCK --pidfile --detach
```

3.2 How to start the Name Resolver

The full list of command line parameters for the name-resolver can be retrieved by the following command:

```
$ sudo ./name-resolver --h
```

Please refer to the help provided by the Name Resolver itself in order to understand how to use the different options.
Please check `[un-orchestrator]/name-resolver/config/example.xml` to understand the configuration file required by the Name Resolver. This file represents a database containing information on all the possible implementations for each available Virtual Network Function.

4 NF-FG supported through the U-SI interface

This section provides some examples of NF-FGs that can be deployed on the Universal Node through the un-orchestrator. We omit additional details of the official NF-FG definition as it has already documented in WP3 [D3.2a].

4.1 Example 1

This example is very simple: configures a graph that receives all the traffic from interface `eth0` and sends it to interface `eth1`, without traversing any VNF.

```
{
	"flow-graph":
	{
		"flow-rules": [
		{
			"id": "00000001",
			"match":
			{
				"port" :"eth0"
			},
			"action":
			{
				"port": "eth1"
			}
		}
		]
	}
}
```

4.2 Example 2

This example is more complex, and it includes a network function called `firewall`. Packets coming from the interface `eth0` are sent to the first port of the network function (`firewall:1`), while packets coming from the second port of the network function (`firewall:2`) are sent on the network interface `eth1`.

```
{
	"flow-graph":
	{
		"VNFs": [
		{
			"id": "firewall"
		}
		],
			"flow-rules": [
			{
				"id": "00000001",
				"match":
				{
					"port" :"eth0"
				},
				"action":
				{
					"VNF_id": "firewall:1"
				}
			},
			{
				"id": "00000002",
				"match":
				{
					"VNF_id" :"firewall:2"
				},
				"action":
				{
					"port": "eth1"
				}
			}
			]
	}
}
```

4.3 Example 3

In this example, traffic coming from eth0 is forwarded to the firewall through the port firewall:1. Then, traffic coming from the firewall (firewall:2) is split based on the destination TCP port. Packets directed to the TCP port 80 is provided to the web cache then to the NAT, while all the other traffic is directly provided to the NAT. Finally, packets from NAT:2 leaves the graph through the port eth2.

```
{
	"flow-graph":
	{
		"VNFs": [
		{
			"id": "firewall"
		},
		{
```

```
                        "id": "NAT"
                },
                {
                        "id": "web-cache"
                }
                ],
                "flow-rules": [
                {
                        "id": "00000001",
                        "match":
                        {
                                "port" :"eth0"
                        },
                        "action":
                        {
                                "VNF_id": "firewall:1"
                        }
                },
                {
                        "id": "00000002",
                        "priority" :"10",
                        "match":
                        {
                                "VNF_id" :"firewall:2",
                                "tcp_dst" :"80"
                        },
                        "action":
                        {
                                "VNF_id": "web-cache:1"
                        }
                },
                {
                        "id": "00000003",
                        "priority" :"1",
                        "match":
                        {
                                "VNF_id" :"firewall:2"
                        },
                        "action":
                        {
                                "VNF_id": "NAT:1"
                        }
```

```
            },
            {
                  "id": "00000004",
                  "match":
                  {
                        "VNF_id" :"web-cache:2"
                  },
                  "action":
                  {
                        "VNF_id": "NAT:1"
                  }
            },
            {
                  "id": "00000005",
                  "match":
                  {
                        "VNF_id" :"NAT:2"
                  },
                  "action":
                  {
                        "port": "eth1"
                  }
            }
            ]
      }
}
```

4.4 Supported matches

Within the match element of the NF-FG description, the following fields are allowed (all the values must be specified as strings):

```
"port"   //only if "VNF_id" is not specified
"VNF_id"  //only if "port" is not specified
"eth_src"
"eth_src_mask"
"eth_dst"
"eth_dst_mask"
"ethertype"
"vlan_id"  //can be a number, "ANY", or "NO_VLAN"
"vlan_pcp"
"ip_dscp"
"ip_ecn"
```

```
"ip_proto"
"ipv4_src"
"ipv4_src_mask"
"ipv4_dst"
"ipv4_dst_mask"
"tcp_src"
"tcp_dst"
"udp_src"
"udp_dst"
"sctp_src"
"sctp_dst"
"icmpv4_type"
"icmpv4_code"
"arp_opcode"
"arp_spa"
"arp_spa_mask"
"arp_tpa"
"arp_tpa_mask"
"arp_sha"
"arp_tha"
"ipv6_src"
"ipv6_src_mask"
"ipv6_dst"
"ipv6_dst_mask"
"ipv6_flabel"
"ipv6_nd_target"
"ipv6_nd_sll"
"ipv6_nd_tll"
"icmpv6_type"
"icmpv6_code"
"mpls_label"
"mpls_tc"
```

4.5 Supported actions

Within the `action` element of the NF-FG description, one and only one of the following fields **MUST** be specified:

```
"port" //only if "VNF_id" is not specified
"VNF_id" //only if "port" is not specified
```

The previous fields indicates an output port through which packets can be sent. Other actions can be specified together with the previous ones:

- *vlan push*, which adds a specific vlan label to the packet;
- *vlan pop*, which removes the more external vlan label to the packet.

The syntax to be used for these operations is the following:

```
"action":
{
      "vlan":
      {
            "operation":"push",
            "vlan_id":"25"
      }
}
```

```
"action":
{
      "vlan":
      {
            "operation":"pop"
      }
}
```

As an example, the following NF-FG tags all the packets coming from interface `eth0` and forwards them on interface `eth1`.

```
{
      "flow-graph":
      {
            "flow-rules": [
            {
                  "id": "00000001",
                  "match":
                  {
                        "port" :"eth0"
                  },
                  "action":
                  {
                        "port": "eth1",
                        "vlan":
                        {
                              "operation":"push",
                              "vlan_id":"25"
                        }
                  }
```

```
                }
                ]
        }
}
```

5 U-SI REST API

The un-orchestrator can either accept a NF-FG from a file, or from its REST interface, that is available thanks to a small HTTP server embedded in the UN. The main REST commands to be used to interact with the un-orchestrator (e.g., deploy a new graph, update an existing graph, etc.) are detailed in the remainder of this section.
We omit additional details of the API implementing the SI-Or interface as it has already documented in WP3.

5.1 Main REST commands accepted by the un-orchestrator

Deploy an NF-FG called "myGraph" (the NF-FG description must be based on the formalism presented in Section 4):

```
PUT /graph/myGraph HTTP/1.1
Content-Type :application/json

{
      "flow-graph":
      {
            "VNFs": [
            {
                  "id": "firewall"
            }
            ],
            "flow-rules": [
            {
                  "id": "00000001",
                  "match":
                  {
                        "port" :"eth0"
                  },
                  "action":
                  {
                        "VNF_id": "firewall:1"
                  }
            },
            ]
      }
}
```

The same message used to create a new graph can be used to add "parts" (i.e., network functions and flows) to an existing graph. For instance, it is possible to add a new flow to the NF-FG called "myGraph" as follows

```
PUT /graph/myGraph HTTP/1.1
Content-Type :application/json

{
      "flow-graph":
      {
            "flow-rules": [
            {
                  "id": "00000002",
                  "match":
                  {
                        "VNF_id": "firewall:2"
                  },
                  "action":
                  {
                        "port": "eth1"
                  }
            },
            ]
      }
}
```

Retrieve the description of the graph with name “myGraph”:

```
      GET /graph/myGraph HTTP/1.1
```

Delete the graph with name “myGraph”

```
      DELETE /graph/myGraph HTTP/1.1
```

Delete the flow with ID “flow_id” from the graph with name “myGraph”:

```
      DELETE /graph/myGraph/flow_id HTTP/1.1
```

Retrieve information on the available physical interfaces:

```
      GET /interfaces HTTP/1.1
```

As a final remark, note that the REST commands just described are not valid when using the NF-FG formalism/library defined in WP3.

5.2 Send commands to the un-orchestrator

In order to interact with the un-orchestrator through its REST API, you can use your favorite REST tool (e.g., some nice plugins for Mozilla Firefox). Just in case, you can also use the cURL command line tool, such as in the following example (where the NF-FG to be instantiated is stored in the file “`myGraph.json`”):

```
$ curl -i -H "Content-Type: application/json" -d "@myGraph.json" -X PUT \
      http://un-orchestrator-address:port/grap/myGraph
```

6 How to deploy a VNF on the Universal Node

This section details how to deploy and run a VNF on the Universal Node. This requires the execution of the following main steps (detailed in the remainder of this section):

- create the desired VNF image;
- register such a VNF in the Name Resolver database;
- send to the un-orchestrator an NF-FG including the VNF to be instantiated.

6.1 Create the VNF image

The UN currently supports three types of VNFs: VNFs executed as Docker containers, VNFs executed inside KVM-based virtual machines, and VNFs based on the DPDK library (i.e., DPDK processes). The folder `[un-orchestrator]/NFs` contains several VNFs to be used as examples for the un-orchestrator.

6.2 Register the VNF in the Name Resolver

Once the VNF image is created, such a VNF must be registered in the Name Resolver database. This operation requires to edit the configuration file of the Name Resolver and reboot the Name Resolver itself; an example of such a file is available at `[un-orchestrator]/name-resolver/config/example.xml`.

6.3 Provide the graph description to the un-orchestrator

In order to deploy your VNF on the UN, you must provide to the un-orchestration a NF-FG including such a VNF.

6.4 An example

This section shows an example in which a VNF called `dummy` and executed as a Docker container is deployed and then run on the Universal Node. This VNF is deployed as part of the service shown in Figure 2.

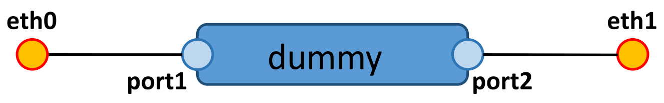


Figure 2: Example of NF-FG to be deployed on the UN.

To create the VNF image and store it in the local file system of the Universal Node, execute the following command in the folder containing the Docker file describing the VNF:

```
   sudo docker build --tag="dummy" .
```

Then, register the new VNF in the Name Resolver by adding the following piece of XML to the configuration file of the Name Resolver itself:

```
<network-function name="dummy" num-ports="2" description="dummy VNF used to
      show the usage of the Universal Node">
      <implementation type="docker" uri="dummy"/>
</network-function>
```

At this point, prepare a NF-FG and pass it to the un-orchestator, which will take care of executing all the operations required to implement the graph. The graph shown in Figure 2 can be described in the JSON syntax defined in Section 4 as follow:

```
{
   "flow-graph": {
      "VNFs": [
      {
         "id": "dummy"
      }
      ],
      "flow-rules": [
      {
         "id": "00000001",
         "match":
         {
            "port": "eth0"
         },
         "action":
         {
            "VNF_id": "dummy:1"
         }
      },
      {
         "id": "00000002",
         "match":
         {
            "VNF_id": "dummy:2"
         },
         "action":
         {
            "port": "eth1"
         }
      },
      {
         "id": "00000003",
         "match":
         {
            "port": "eth1"
         },
         "action":
         {
            "VNF_id": "dummy:2"
         }
      },
      {
         "id": "00000004",
         "match":
         {
            "VNF_id": "dummy:1"
         },
         "action":
         {
            "port": "eth0"
         }
      }
      ]
   }
}
```

This json can be stored in a file (e.g., `nffg.json`) and provided to the un-orchestrator either through the command line at the boot of the un-orchestrator, or through its REST API. In the latter case, the command to be used is the following:

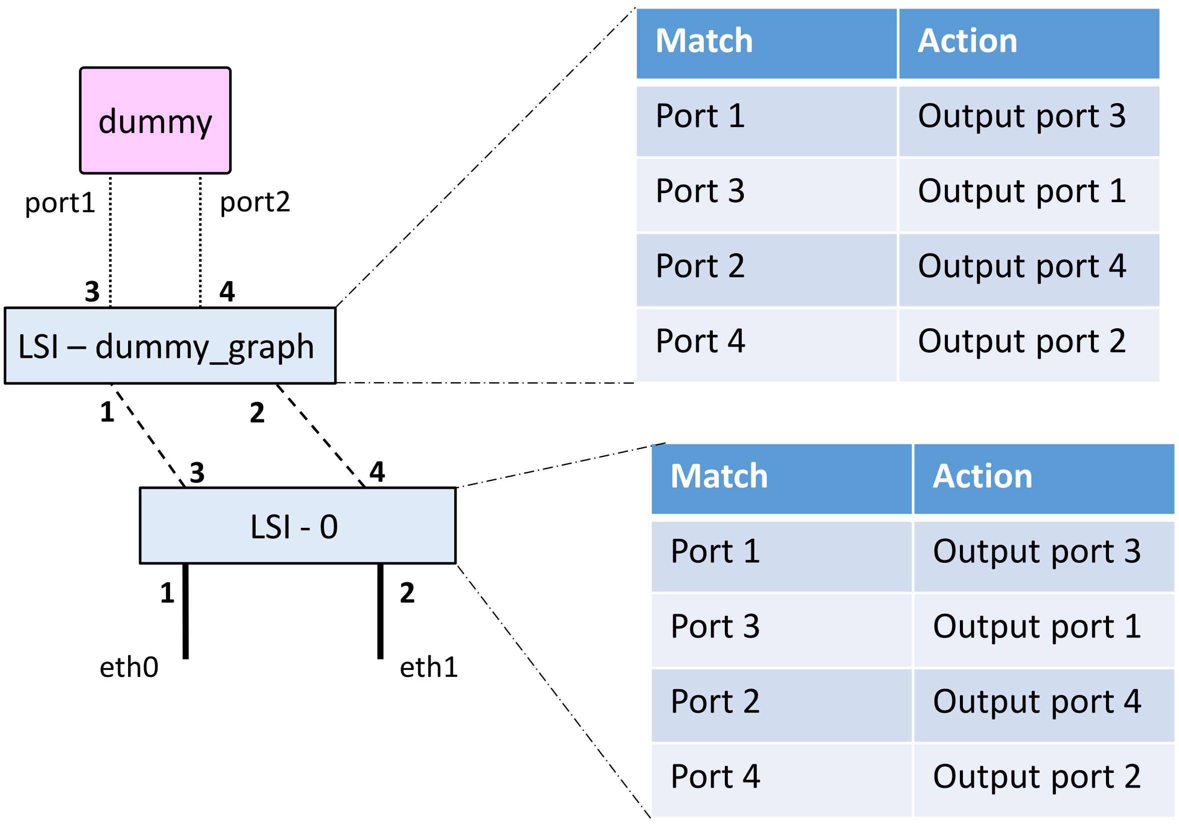
```
curl -i -H "Content-Type: application/json" -d "@nffg.json" -X PUT \
      http://un-orchestrator-address:port/graph/graphid
```

where the `graphid` is an alphanumeric string that will uniquely identify your graph in the orchestrator.
At this point the un-orchestrator:

- creates a new LSI through the *network controller*, inserts the proper Openflow rules in such an LSI in order to steer the traffic among the VNFs of the graph, and inserts the proper Openflow rules in the `LSI-0` (which is the only LSI connected to the physical interfaces) in order to inject the proper traffic in the graph, and properly handle the network packets exiting from such a graph;

- starts the Docker image implementing the VNF with name dummy (the image associated with the name of the VNF is obtained through the Name Resolver) through the *compute controller*.

Figure 3 shows how the NF-FG of the example is actually implemented on the UN; in particular, it depicts the connections among LSIs and the VNF, and the rules in the flow tables of the involved LSIs.



Match	Action
Port 1	Output port 3
Port 3	Output port 1
Port 2	Output port 4
Port 4	Output port 2

Match	Action
Port 1	Output port 3
Port 3	Output port 1
Port 2	Output port 4
Port 4	Output port 2

Figure 3: NF-FG deployed on the UN.

To conclude, the deployment of a second graph will trigger the creation of a new LSI, again connected with the `LSI-0`; the `LSI-0` will then be instructed to properly dispatch the traffic coming from the physical ports among the deployed NF-FGs, according the the NF-FGs themselves.

References

[D3.2a] Pontus Skoldstrom et al. *D3.2a Network Function Forwarding Graph specification (Supplement to D3.2)*. Tech. rep. D3.2sup. UNIFY Project, July 2015.