NETWORK FUNCTIONS AS-A-SERVICE
OVER VIRTUALISED INFRASTRUCTURES

GRANT AGREEMENT NO. 619520

Deliverable D5.31

Network Functions Implementation and Testing - Interim

Editor P. Paglierani (ITALTEL)

Contributors P. Comi, M. Arnaboldi (ITALTEL), B. Parreira (PTINS), Y. Rebahi (FOKUS), A. Abujoda (LUH), F. Pedersini (UNIMI), N. Herbaut (VIOTECH), A. Kourtis, G. Xilouris (NCSRD), G. Dimosthenous (PTL).

Version 1.0

Date November 30th, 2015

Distribution PUBLIC (PU)

Executive Summary

This deliverable provides general guidelines and some specific information that can be used by Function Developers to develop Virtual Network Functions (VNFs) within the T-Nova framework. Also, it contains the description of six VNFs that are currently under development in T-Nova. The VNFs are the following:

- Security Appliance
- Session Border Controller
- Video Transcoding Unit
- Traffic Classifier
- Home gateway
- Proxy as a Service.

The VNFs developed in T-Nova span a very wide area of the Network Function domain, and can thus represent, from a developer's perspective, a set of highly significant implementation use cases, in which many problems related to network function virtualization have been faced and solved. Also, in the VNFs presented in this document different technologies have been adopted by T-Nova developers. Most VNFs, in fact, take advantage of various contributions coming from the open source community, such as [SNORT], or exploit recent technological advances, such as [DPDK], SR-IOV [Walters], or general purpose Graphical Processing Units [CUDA].

Some practical information that can be useful to function developers is also provided in the first part of the document, related to the most implementation issues encountered in the development phase. Finally, some preliminary results obtained in the tests that have been carried out are also reported and briefly discussed.

Table of Contents

Index of Figures

1. Introduction

This document contains a description of the Virtual Network Functions (VNFs) developed in the T-Nova project. Those VNFs have been developed in order to demonstrate, with real-life applications, the capabilities offered by the overall T-Nova framework. Moreover, other function providers who want to develop new VNFs within T-Nova can use them as an example. To this aim, general guidelines are briefly summarized and discussed in the first part of the document. In particular, some main concepts related to the VNF internal topology, the lifecycle management, the VNF Descriptor and the interaction with the monitoring framework are discussed, and information of practical interest to VNF developers is provided. Then, the specific VNFs currently being developed in T-Nova are described. In particular, six different VNFs are discussed, covering a wide range of applications, which are:

- Virtual Security Appliance;
- Virtual Session Border Controller;
- Virtual Transcoding Unit;
- Traffic Classifier;
- Virtual Home Gateway;
- Proxy as a Service;

For each VNF, architectural and functional descriptions are provided, along with the technologies used and the internal/external interfaces. In addition, some preliminary test results that were obtained are summarized and briefly discussed.

2. Network Functions Implementation

This section describes properties, rules, and best practices applicable to any VNF. Many of the VNF's properties should be described in the VNF Descriptor, or simply in the VNFD.

2.1. Common to all VNFs

2.1.1. VNF general architecture

In the T-NOVA framework, a VNF is defined as a group of Virtual Network Function Components (VNFC); also, one VNFC consists of one single Virtual Machine. Each VNF shall support the T-NOVA VNF lifecycle (i.e. start, stop, pause, scaling, etc.) under the control of the VNFM.

With the exception of some mandatory blocks (described in the following), internal implementations of a VNF are left to the VNF Developer. Nonetheless, it is suggested to adopt a common structure for the VNF internal components as depicted in Figure 1. VNF developers who aim to develop new VNFs should follow the common practices introduced in the T-NOVA framework.

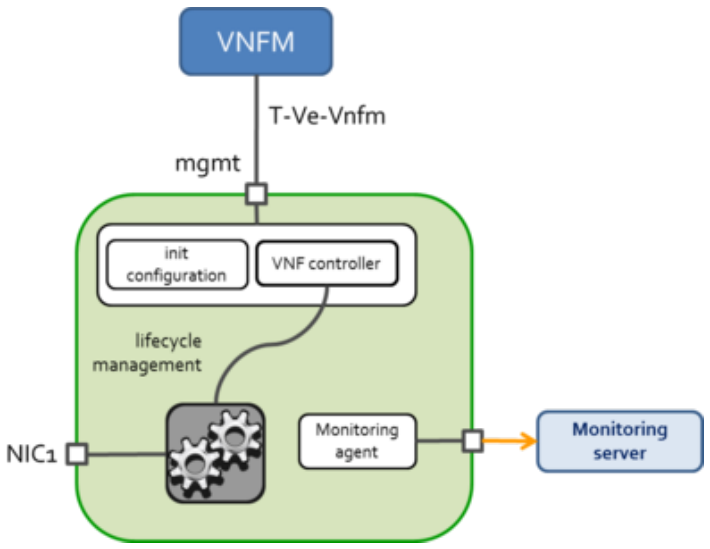


Figure 1. VNF internal components

In this architecture the **VNF Controller** is the internal component devoted to the support of the VNF lifecycle. The **Init Configuration** component is responsible for the initialization of the VNF that happens at the beginning of the VNF execution. The **Monitoring Agent** component transmits application-level monitoring data towards the Monitoring System. Extensive description of the VNF architecture and its specifications can be found in [D2.41].

All the VNF internal components are optional, except the VNF Controller. The VNF Controller often acts as VNF Master Function, and is responsible for the internal organisation of the VNFCs into a single VNF entity [NFVSWA]. It must be present in each VNF because it is in charge of supporting the T-Ve-Vnfm interface towards the VNFM. VNF developers are free to develop VNF internal components in any way they prefer as long as they comply with the VNF lifecycle management interface T-Ve-Vnfm, defined in [D2.21]. The previous case applies when the T-NOVA generic VNFM (VNFM-G) is used. Conversely, if the VNF is supplied with a specific VNFM (VNFM-S), then the implementation of the control plane within the VNF is entirely up to the developer.

In the case of a VNF composed by more than one VNFC, the VNF developer is free to accommodating the internal VNF components. The minimal mandatory requirement is that a unique VNF controller must be inserted in each VNF, i.e. the VNF controller shall be installed in just one VNFC. The remaining components, i.e. Init Configuration and Monitoring Agent, can be freely allocated in different VNFCs. In Figure 2 we provide an example of a VNF composed by two VNFCs. In this case the Init Configuration component is allocated in both the VNFCs, while there is only one Monitoring Agent. Of course different configurations are also possible depending on the particularities of the VNF.

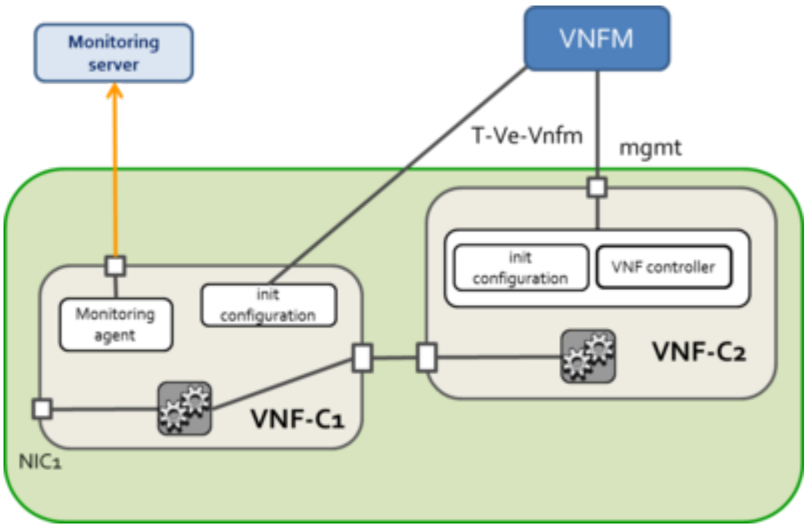


Figure 2. VNF composed by many VNFCs

2.1.2. VNF Descriptor

Aligned to the ETSI Specification for VNF Descriptors [NFVMAN], T-NOVA proposes a simplified version for the first sprint of the T-NOVA platform implementation. As defined in [NFVMAN], a VNF Descriptor (VNFD) is *"a deployment template which describes a VNF in terms of deployment and operational behaviour requirements"*.

As the VNFs are participating in the networking path, the VNFD also contains information on connectivity, interfaces and KPIs requirements. The latter is critical for the correct deployment of the VNF as it is used by the NFVO in order to establish appropriate Virtual Links within the NFVI between VNFC instances, or between a VNF instance and the endpoint interface to other Network Functions.

It should be noted that the 3rd party developers, implementing VNFs to be used in T-NOVA platform need to follow some basic guidelines in relation to the anticipated VNF internal architecture and deployment as well as for the VNFM – VNF communication as was illustrated in the previous subsection (Section 2.1.1), applicable in case the generic VNFM (VNFM-G) is being used.

Prior to presenting the VNFD template as it is introduced in T-NOVA, the following subsection elaborates on the structure and connectivity domains that are expected by T-NOVA.

2.1.3. VNFD and VNF Instantiation

The instantiation of a VNF is bound to the type and complexity of the VNF, a non-exhaustive list of factors that play role in this are:

- VNF complexity (e.g. VNF consisting of single VNFC)
- VNF internal networking topology (e.g. virtual networks required to interconnect the VNFCs)
- Restrictions on VNFC booting order
- Existence of VNF specific VNFM
- Use of Element Management (EM)

An example scenario for the instantiation of a VNF is explored below. A VNF comprised of 3 VNFCs is considered. Each VNFC is described in the VNFD, where the required resources for each VNFC are declared. The VNFM instantiates the three VNFCs by signalling to the NFVO and through that to the VIM the creation and instantiation of the three virtualisation containers that will host the three VNFCs. At this point the VNFC may be seen as independent entities not yet organised as a VNF. However, the VNFCs are interconnected using the stated in the VNFD internal networks and interfaces. The L2/L3 information, vital for the next steps is retrieved by the VNFM who in turn communicates with the NVF Controller in order to proceed with the configuration and organisation of those VNFCs into the VNF.

The next subsections provide more information on the T-NOVA pre-selected network domains foreseen for all the VNFs in order to simplify the deployment and instantiation process.

2.1.4. VNFC Networking

According to the T-NOVA architecture each VNFC should have four separate network interfaces, each one bound to a separate isolated network segment. The networks related to any VNFC are: management, datapath, monitoring and storage. The figure below (Figure 3) illustrates the above statement. In various cases the above rule might not be followed, especially in the case where there is no real requirement for a particular network/interface e.g a VNFC that is not using persistent storage on the storage array of the NFVI.

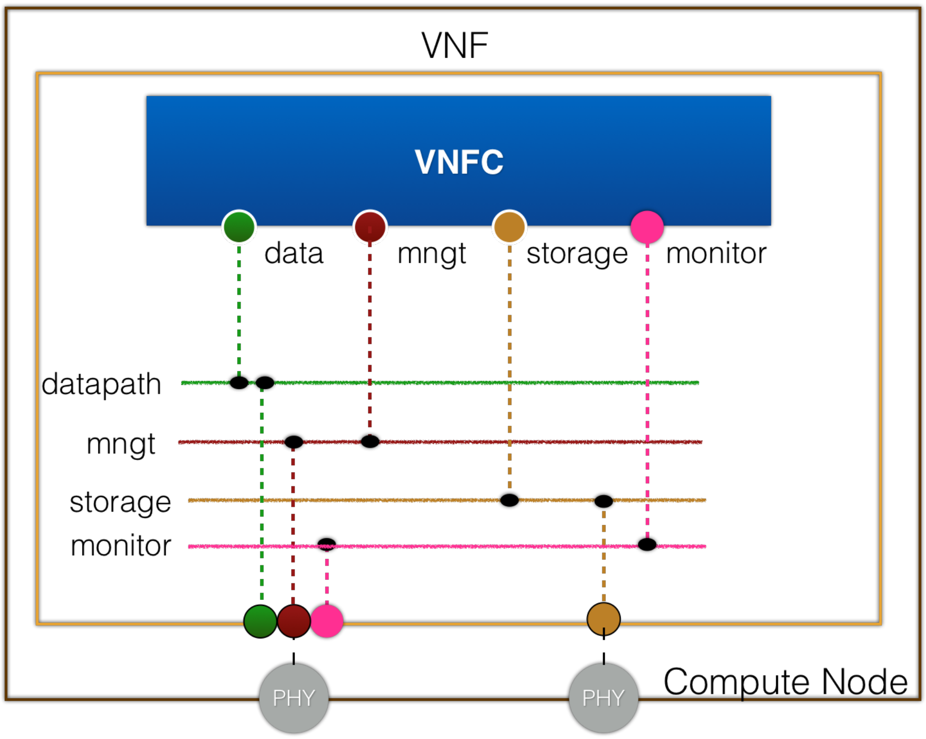


Figure 3 VNF/VNFC Virtual Links and Connection Points

The **management network** provides the connection and communication with the VNFM and passes lifecycle related events seamlessly to the VNF. The management network conveys the information passed over from the VNFM to the VNF Controller (T-Vnfm-Vnf interface). The VNFM can control each particular VNFC, through the management interface, and provide the overall stable functionality of the VNF.

The **datapath network** provides the networking for the VNF to accept and send data traffic related to the network function it supports. For example, a virtual Traffic Classifier would receive the traffic under classification from this interface. The datapath network can consist of more than one network interfaces, that can receive and send data traffic. For example, in the case where the function of the VNFC is a L2 function (e.g. a bridge), the anticipated interfaces are two, one for the ingress and another one for the egress. In some cases, those interfaces might be mapped on different physical interfaces too. The number of interfaces for the datapath and their particular use is decided by the VNF provider. Additionally, the data traffic that needs to be shared among different VNFCs uses the datapath network.

The **monitoring network** provides the communication with the monitoring framework. Each VNF has a monitoring agent installed on each VNFC, which collects VNF specific monitoring data and signals them to the Monitoring Framework (see [D4.01]) or to the VNFM/EM depending on the situation. The flow of the monitoring data is from the VNF service to the monitoring agent and finally to the monitoring framework, which collects data from all VNFs. A separate Monitoring network has been introduced in T-NOVA to cope with the amount of traffic generated in large-scale deployments. Though, the monitoring traffic can be easily aggregated with the management traffic into a single network instance, if this solution results adequate to the specific application.

The **storage network** is indented for supporting communication of the VNFCs with the storage infrastructure provided at each NFVI. This applies to the case where a VNFC will utilize persistent storage. In many NFVI deployment scenarios the physical interface that handles the storage signaling (e.g. iSCSI) on each compute node is separated from the other network segments. This network segment is considered optional and only applicable to the above use cases.

The resulting deployment in an Openstack environment is illustrated below in Figure 4. In this example a deployment of two VNFC that comprise a single VNF is illustrated.

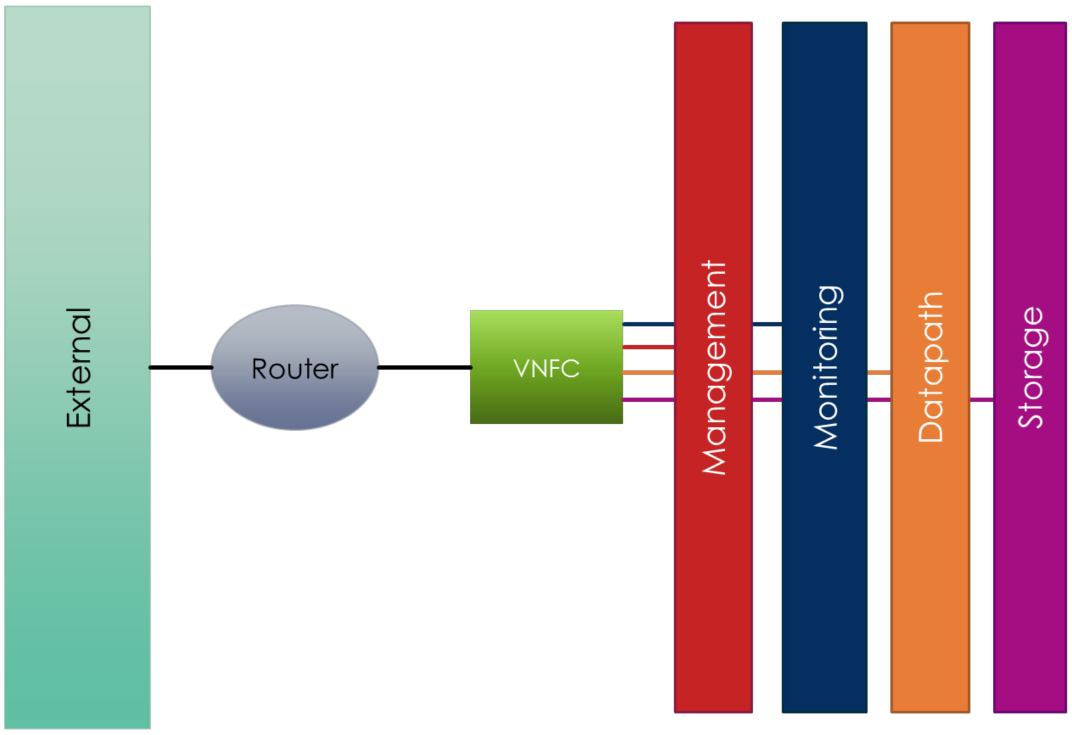


Figure 4. OpenStack overview of the T-NOVA VNF deployment networks.

2.1.5. VNF internal structure

A VNF may be composed by one or multiple components namely the VNFCs. In this perspective each VNFC is defined as a software entity deployed in a virtualization container. As discussed in the previous section, Figure 3 presents the case where a VNF is comprised of one VNFC, whereas in Figure 5, an example of a VNF comprised by 3 VNFC is given. The internal structure of the VNF is entirely up to the VNF developer and should be seamless to the T-NOVA SP. It should be also noted that for the same NF that is being virtualized, different developers might as well choose different implementation methods and provide the same NF with different number of components. The VNF internal structure is described in the VNFD as a graph. Inside the VNFD the nodes (vertices) of the graph are the interfaces of each VNFC, and the internal Virtual Links interconnecting the VNFC are the edges of the graph.

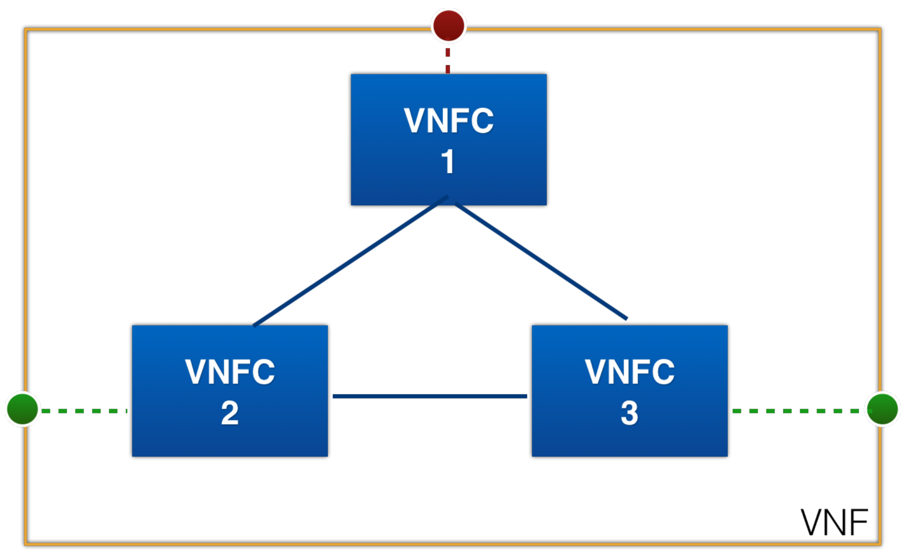


Figure 5 Multi VNFC composed VNF

The topology includes the connections between different network interfaces of the different network components. From a VNF perspective, this topology is translated into the different connections over the virtual links (VLs) that connect the different VNFCs. An example of inter-VNFC topology is shown in Figure 6. The monitoring network is considered as multiplexed with the management.

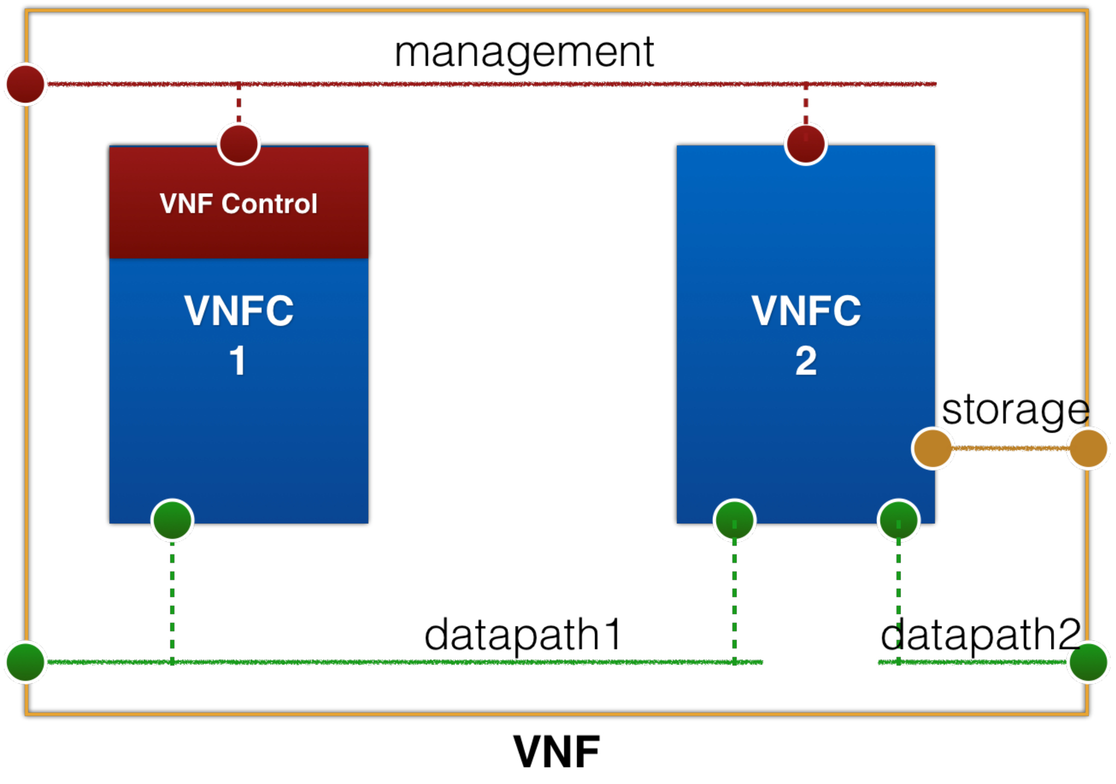


Figure 6. Virtual Link VNFC interconnection example.

In this example we allocate two datapath network segments. Datapath1 is used to allow incoming traffic to both VNFCs at the same time (using traffic mirroring). Datapath2 is used for the traffic exiting the VNF. The management is used for allowing the external communication of the VNF to the VNFM and also to allow VNF1 (controller) to control and coordinate VNFC2. It is apparent that other variants of the above networking example could be possible. The developer will use the VNFD in order to specify the exact internal structure and connectivity of the VNF.

2.1.6. VNF management interface

Each VNF shall have a management interface, named **T-Ve-Vnfm interface** that is used to support the VNF lifecycle. The VNF lifecycle is shown in Figure 7.

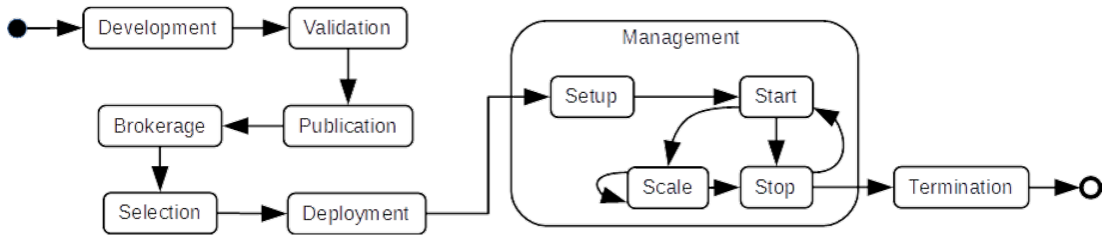


Figure 7. VNF lifecycle

A general discussion on the VNF lifecycle can be found in [NFVMAN]. Also, the implementation of the VNF lifecycle as envisioned in the T-Nova project is discussed in [D2.41], section 3.3. Some relevant aspects encountered by developers in the VNF development process are explained below. Some guidelines to clarify the activities that must be carried out in order to produce VNFs that can be used in the T-Nova framework are also provided.

In T-Nova, the VNF management interface is technically implemented by only one of the VNFCs constituting the VNF. Such VNFC is in charge of distributing the lifecycle-related information to all the others VNFCs constituting a VNF.

In accordance with the ETSI Mano, the interaction between VNFM and VNF, implementing the VNF lifecycle, is thoroughly described in the VNF Descriptor. In particular, the VNFD contains a section called "lifecycle_event", which provides all the details to allow the VNFM to interact with the VNF in a fully automated way.

To this aim, each VNF needs to be able to declare various lifecycle events (e.g. start, stop, pause...). For each of those events, the information needed to configure the VNF can be very different. Moreover, the command to trigger the re-configuration of the VNF can change between events. To support the different events each one needs to be detailed in the VNFD.

The information related to the VNF life-cycle is inserted in the “lifecycle_event” section of the VNFD. In particular, in such a section the following information is available:

- **Driver**: the protocol used for the connection between the VNF Manager and the controlling VNFC. In T-Nova, two protocol can be used, the Secure Shell (ssh) protocol and the http protocol.
- **Authentication**: such fields specify the type of authentication that must be used for the connection, and some specific data required by the authentication process (e.g. a link to the private key injected by the VNFM at startup).
- **Template File Format**: specifies the format of the file that contains the information about the specific lifecycle event, and that must be transferred once the command is run.
- **Template File**: includes the name and the location of the Template File.

- **VNF Container**: specifies the location of the Template File.
- **Command**: defines the command to be executed

2.1.7. VNF Descriptor (VNFD)

As explored in the above sections, the VNFD plays an important role in the proper deployment of a particular VNF in the NFVI by the NFVO, as well as in the portability of the VNF to NFVI variants. This section attempts a specification of the VNFD as used currently by T-NOVA. The refined final version will be published in D.5.32.

2.1.7.1. VNFD Preamble

The VNFD preamble provides the necessary information for the release, id, creation, provider etc. T-NOVA extents the information with Marketplace related information such as trading and billing.

Listing 1 VNFD Preample

```
  release: "T-NOVA v0.2" # Release information
  id: "52439e7c-c85c-4bae-88c4-8ee8da4c5485" # NFStore provides
the rules, not know prior to uploading and cannot be referenced on
time zero
  provider: "NCSRD" # Provider(T-NOVA) - Vendor (ETSI)
  provider_id: 23 # FP given by ID by the Marketplace
  description: "The function identifies, classifies and forwards
network traffic according to policies"
  descriptor_version: "0.1"
  version: "0.22"
  manifest_file_md5: "fa8773350c4c236268f0bd7807c8a3b2" #
Calculated by the NFStore during upload
  type: "TC"
  date_created: "2015-09-14T16:10:00Z" # Auto inserted at the
Marketplace
  date_modified: "2015-09-14T16:10:00Z" # Auto inserted at the
Marketplace (to bechecked)
  trade: true
  billing_model:
    model: "PAYG" # Valid Options are PAYG and Revenue Sharing
(RS)
    period: "P1W" # Per 1 week, e.g. P1D per 1 day
    price:
      unit: "EUR"
      min_per_period: 5
      max_per_period: 10
      setup: 0
```

2.1.8. Virtual Deployment Unit (VDU)

The vdu VNFD segment provides information about the required resources that will be utilised in order to instantiate the VNFC. The configuration of this part may be extremely detailed and complex depending on the platform specific options that are provided by the developer. However it should be noted that the more specific are the requirements stated here the less portable the VNF might be, depending on the NFVO policies and the SLA specifications. It is assumed that each vdu describes effectively the resources required for the virtualisation of one VNFC.

The listing below (Listing 2) presents a snippet of the vdu section of the VNFD focusing on the IT resources and platform related information. Other fields may also be noted such as: i) Lifecycle events – where the drivers for interfacing with the VNF controller are defined as well as the appropriate commands allocated to each lifecycle event; ii) scaling – defining the thresholds for scaling in-out; and iii) VNFC related subsection where the networking and inter-VNFC Virtual Links are defined.

Listing 2 VNFD vdu section

```
<snip>
vdu:
    - id: "vdu0" # VDUs numbered in sequence (TBD)
      vm_image: "http://store.t-nova.eu/NCSRDv/TC_ncsrd.v.022.qcow" # URL
that contains the
      vm_image_md5: "a5e4533d63f71395bdc7debd0724f433" # generated by the
NFStore

      # VDU instantiation resources
      resource_requirements:
        vcpus: 2
        cpu_support_accelerator: "AES-NI" # Opt. if accelarators are
required
        memory: 2 # default: GB
        memory_unit: "GB" # MB/GB (optional)
        memory_parameters: # Optional
          large_pages_required: ""
          numa_allocation_policy: ""
        storage:
          size: 20 # default: GB
          size_unit: "GB" # MB/GB/TB (optional)
          persistence: false # Storage persistence true/false
        hypervisor_parameters:
          type: "QEMU-KVM"
          version: "10002|12001|2.6.32-358.el6.x86_64"
        platform_pcie_parameters: # Opt. required if SR-IOV is used for
graphics accelaration
          device_pass_through: true
          SR-IOV: true
        network_interface_card_capabilities:
          mirroring: false
          SR-IOV: true
        network_interface_bandwidth: "10Gbps"
        data_processing_acceleration_library: "eg DPDK v1.0"
        vswitch_capabilities:
          type: "ovs"
          version: "2.0"
          overlay_tunnel: "GRE"
      networking_resources:
           average: "7Mbps"
           peak: "10Mbps"
           burst: "200KBytes"

      #VDU Lifecycle enents
      vdu_lifecycle_events:

      # VDU Scaling config
      scale_in_out: # number of instances for scaling allowed
        minimum: 1 # min number of instances
        maximum: 5 # max number of instances, if max=min then no scaling is
allowed

      # VNFC specific networking config
      vnfc:tantiation parameters
```

2.1.8.1. VNFC subsection

This section of VNFD is related to the internal structure of the VNF and describes the connection points of the vdu (name id, type) and the virtual links where they are connected. The above information is illustrated in the provide VNFD listing below.

Listing 3 VNFC section of the VNFD

```
vnfc:
        id: "vdu0:vnfc0" #incremental reference of VNFCs in the VDU

        networking:
          # Start of T-NOVA networking section
          - connection_point_id: "mngt0" # for monitoring and management purposes
            vitual_link_reference: "mngt"
            type: "floating" # The type of interface i.e floating_ip, vnic, tap
            bandwidth: "100Mbps"
          - connection_point_id: "data0" # datapath interface
            vitual_link_reference: "data"
            type: "vnic" # The type of interface i.e floating_ip, vnic, tap
            bandwidth: "100Mbps"
...snipped ...
      vlink:
        # T-NOVA VL Section : relative to resources:networks HEAT template,
subnets and dns information should be filled from the Orchestrator therefore are
not put here.
        - vl_id: "mngt"
          connectivity_type: "E-Line"
          connection_points_reference: ["vdu0:mngt0", "vnf0:mngt0"]
          root_requirement: "10Gbps"
          leaf_requirement: "10Gbps" # Used only for E-tree
          qos: "BE"
          test_access: "active" # Active: Use ext IP to be reached, Passive: L2
access via steering, None: no access
          net_segment: "192.168.1.0/24" #a-priori allocation by the system during
approval of the VNFD.
          dhcp: "TRUE"
...snipped...
```

The above information is parsed and translated to HEAT template that the NFVI VIM based on Openstack is able to parse and provide accordingly the required networks.

2.1.9. Instantiation parameters

The VNFD enables VNF developers to thoroughly describe the network function in terms of static and instantiation-specific parameters, or more easily, instantiation parameters. The latter, usually apply to the configuration of the VNF application and their values can only be known at runtime (e.g. IP addresses). Nevertheless, VNF developers still need to describe which of the runtime parameters are needed and where they are needed. For these cases, VNF developers can use one of two available mechanisms:

- **Get Attributes Function**: cloud orchestration templates usually provide a set of functions that enable users to programmatically define actions to be performed during instantiation. For example, in HOT, the "get_attr" function allows the retrieval of a resource attribute value at runtime;

- **Cloud Init**: cloud init is a library supported by multiple cloud providers that allows developers to configure services to be ran within VMs at boot time. In traditional IaaS environments it is commonly used to upload user-data and public keys to running instances.

Although both mechanisms can be used to upload deployment specific parameters to VNFs, they are intrinsically different and it's up to the VNF developers to decide which one best suits their VNFs. In case, developers can also decide to use both mechanisms. The main functional differences between these two are summarized in Table 2.

Table 2-A: Methods for instantiation parameter upload

Method	When	How
"get_attr"	In any momment between the Deployment and Termination VNF lifecycle events	The VIM provides the information to the Orchestrator, which forwards it to the VNFM, and this to the VNF by using the middleware API
Cloud Init + Configuration Management Software	At the VM boot time	The VIM provides the information through the Openstack Metadata API which is collected by the VM after Boot

Get Attributes

This method is based on the "get_attr" function defined in HOT. This function works by referencing in the template the logical name of the resource and the resource-specific attribute. The attribute value will be resolved at runtime. The syntax for the function can be found at [GetA] and is as follows:

```
 { get_attr: [<resource_name>, <attribute_name>, <key/index 1>,
<key/index 2>, …] }
```

In which:

- **resource_name**: The resource name for which the attribute needs to be resolved;
- **attribute_name**: The attribute name to be resolved. If the attribute returns a complex data structure such as a list or a map, then subsequent keys or indexes can be specified. These additional parameters are used to navigate the data structure to return the desired value;
- **key/index i**: …

2.1.9.1. Cloud Init + Configuration Management Software

Another complementary approach is to dissociate the actual initialization of the VM and its configuration through the middleware API. We delay the configuration of VNFC until a command is received at the VNF Controller level. Following an Infrastructure-as-code approach, the controller, when receiving a lifecycle event from the mAPI, is in charge of enforcing the compliance of the infrastructure with the state received from the Orchestrator through the mAPI.

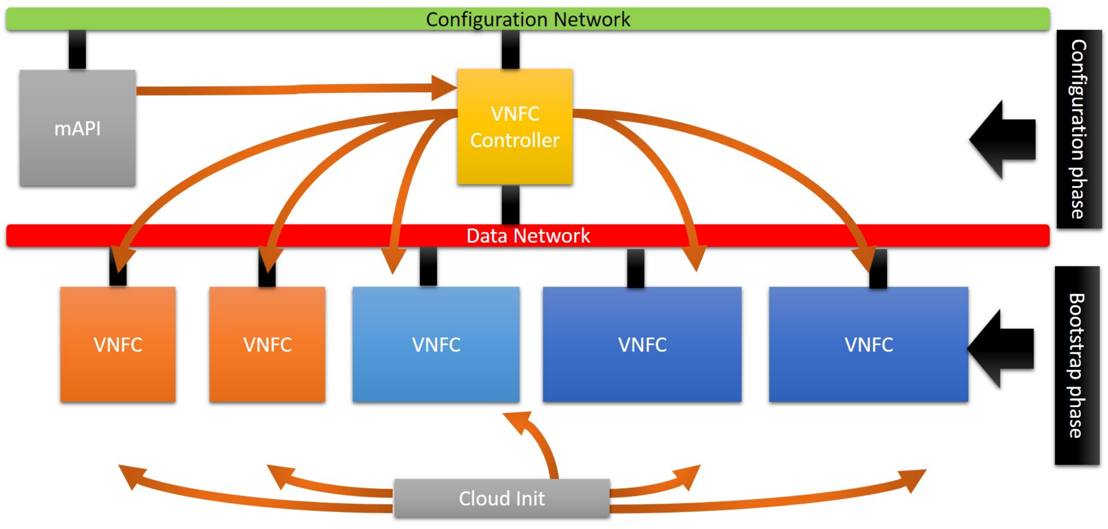


Figure 8. Two-phase VNF configuration

According to this technique, the allocation of a VNF is carried out in two different phases. The first one or Bootstrapping Phase, handled by cloud init, connects the VM together by sharing some very limited information revolving around networking (sharing of IP). The second one, the Configuration Phase, needs a configuration management software being installed on the various VNFCs that "glue" them together and allow configuration being propagated from the VNF Controller down to the other VNFCs.

When the VNF Controller receives a lifecycle event (for instance start) from the Middleware API, it will implement the order using the configuration management software on the other VNFC.

For an example of implementation for this method, please refer to 2.6.4.6.

This approach brings more complexity, as it imposes using configuration management software, and the VNF is responsible for applying the configuration. However, it becomes valuable when considering complex VNF with several VNFCs and scaling.

2.1.10. Interaction with configuration middleware

The main advantage of having a middleware to transmit commands to the VNFs is being able to express those commands in a technology-agnostic way in the VNFD.

The middleware API supports multiple technologies to send instantiation information and trigger state changes in VNFs. Currently two technologies are supported:

- SSH
- HTTP REST-based interface

VNF developers can use the VNFD to define the templates that hold the information (e.g. json files) and the commands that will trigger the configuration of the VNFs.

The specification for SSH is summarized below:

- **Authentication**:
    - Private-Key: a private-key is available in T-NOVA platform that enables authentication in virtual machines when using the SSH protocol. When deploying the virtual machines this private key is injected in the VM by the VIM.

- **Operations**:
    - Copy file to host: this operation allows the middleware API to send the configuration file to the VM. It enables the exchange of configuration parameters with the VNFs;
    - Run remote command: this operation is used to trigger a lifecycle event/reconfiguration of the VNF. With this operation VNF developers can specify not only commands that should be run inside the VM but also scripts.

The specification for HTTP is summarized below:

- **Authentication**:
    - Basic access authentication: for T-NOVA, when using HTTP only basic authentication will be supported. This authentication is realized by sending a username and password via HTTP header when making a request;
- **Operations**:
    - File upload: simple HTTP file upload operation using "multipart/form-data" encoding type. POST and PUT operations must be supported to enable a single step operation (upload file and trigger reconfiguration);
    - Send POST request: POST request only applies to the start operation following the RESTful architecture best practices;
    - Send PUT request: PUT requests are used in all lifecycle events with the exception of the "start" event;
    - Send DELETE request: this request will map in the "destroy" lifecycle event.

2.1.11. Monitoring

Collecting monitoring data between the VNF and the T-NOVA architecture is a major task in T-NOVA. Both System metrics that are non-VNF specific and software metrics that are VNF specific are collected by the same database in the T-NOVA platform. In this section, the functionality that each VNF should implement in order to push data monitoring metrics into the monitory database is described.

2.1.11.1. System Monitoring

Monitoring of the system information is currently collected using the collectd agent [Collectd] that is installed and run on each VNFC. Metrics such as CPU, RAM, I/O are collected on a VM-based level and pushed to the collectd server.

Other options including using Openstack Ceilometer to collect basic system information are currently considered by T4.4.

2.1.11.2. VNF Specific Monitoring

In this section, two methods for the interaction with the T-Nova monitoring framework are presented.

Simple Monitoring

The VNF developer should perform some monitoring specific developments during the implementation phase of the VNF in order to be T-NOVA compatible. Specifically, the following steps should be followed by the VNF developer:

- Installation of Python 2;

- Copy of the small Python script provided by T-NOVA to be called when monitoring data is to be sent;
- The monitoring data should be sent to the monitoring system regularly (for example, through the use of a cron job);
- Finally, the Python 2 script should be called to push the metrics in the T-NOVA database, in a key-value format.

The main advantages of this approach are:

1. It doesn't require any specific installation as long as Python 2 is already installed, which is usually installed by default on the most Linux distributions;
2. It allows VNF developers to implement their own way on how metrics are collected from their application. For example, in the currently developed VNFs, one developer queries XML files with XPath to output tuples (metric name, metric value) and the python script is called with those metrics. Another developer may parse logfiles and extract relevant data with some custom technology.

The main drawback is the necessity to support the Python script as an external asset, which is not acceptable from professional developers if updated regularly, as it must be audited before integration to the final solution.

2.1.11.3. SNMP Monitoring

The monitoring data can be exchanged with the Monitoring Manager via snmp protocol. This kind of interaction requires the installation of the snmp collectd agent in the Virtual Machine. The Monitoring Manager requests metrics by sending a snmp Get Request to the Monitoring Agent. Typically UDP is used as a transport protocol. In this case, the Agent receives Requests on UDP port 161, while the Manager sends Requests from any available source port. The Agent replies with a snmp GET Response to the source port of the Manager.

An example of these procedures is depicted in Figure 9, which refers to the vSBC. In this case the monitoring data are collected by the O&M component, acting as a snmp Monitoring Agent.

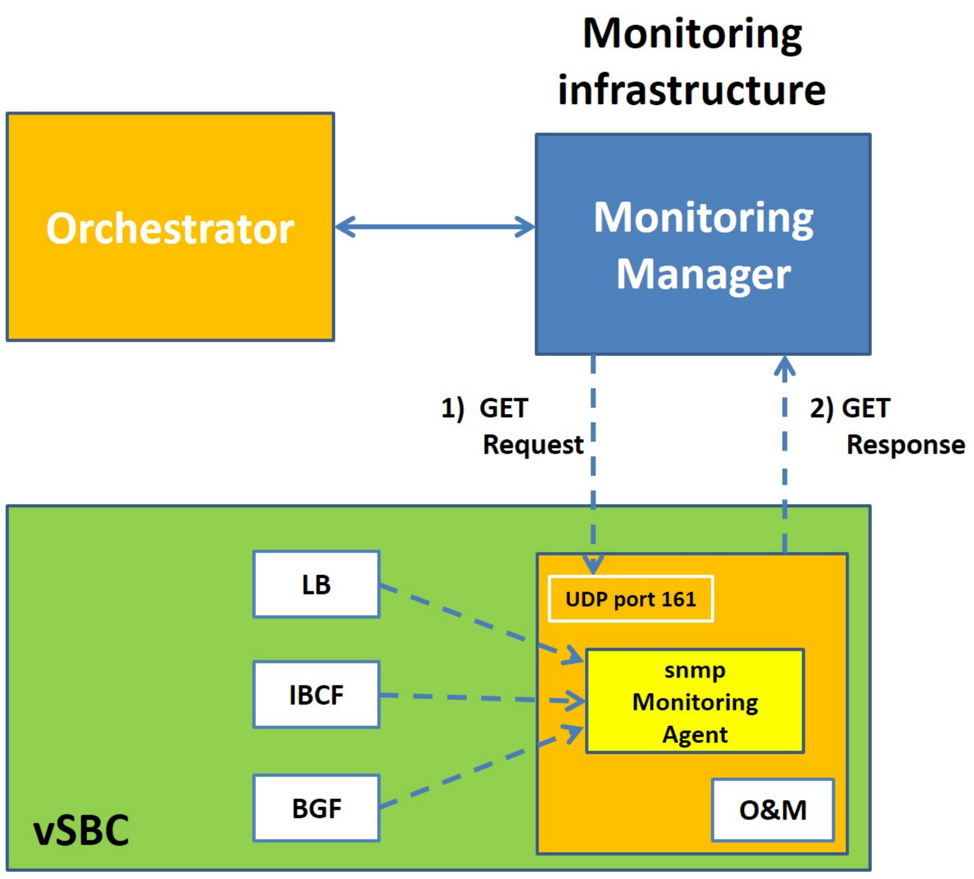


Figure 9. Monitoring data handling (via snmp) in case of vSBC

Monitoring data is described by a Management Information Base (MIB). This MIB includes the VNF/VDU identifiers and uses a hierarchical namespace containing object identifiers (OIDs). Each OID identifies a variable that can be read via snmp protocol (see RFC 2578). The snmp GET Request contains the OIDs to be sent to the Monitoring Manager and the snmp GET Response contains the values of the collected data.

Usually the time intervals between two consecutive collections of monitoring metrics are not synchronized with the snmp Request coming from the Manager. For this reason, unless the examined metric isn't an instantaneous value, the VNF returns to the Manager all data evaluated during the previous monitoring periods.

As an example, if the snmp GET Request is received by Monitoring agent at 17:27 and the VNF monitoring period is 5 minutes, the monitoring data sent to the Manager belong to the time interval "17:25 –> 17:30".

We also assumed that:

- The generation of the monitoring data is active by default inside the VNF;
- The monitoring period of each metric is configured inside VNF components (i.e., IBCF, BGCF in case of vSBC). If the frequency of the snmp GET Request is greater than the VNF monitoring period, the same value is sent more than once to the requesting Monitoring Manager.

2.2. Virtual Security Appliance

2.2.1. Introduction

A Security Appliance (SA) is a "device" designed to protect computer networks from unwanted traffic. This device can be active and block unwanted traffic. This is the case for instance of firewalls and content filters. A security Appliance can also be passive. Here, its role is simply detection and reporting. Intrusion Detection Systems are a good example. A virtual Security Appliance (vSA) is a SA that runs in a virtual environment.

In the context of T-NOVA, we have suggested a virtual Security Appliance (vSA) composed of a firewall, an Intrusion Detection System (IDS) and a controller that links the activities of the firewall and the IDS. The vSA high level architecture was discussed in details in [D5.01].

2.2.2. Architecture

The idea behind the vSA is to let the IDS Analyze the traffic targeting the service and if some traffic looks suspicious, the controller takes a decision by, for instance, revising the rules in the firewall and block this traffic.

The architecture of this appliance is depicted in Figure 10 and includes the following main components.

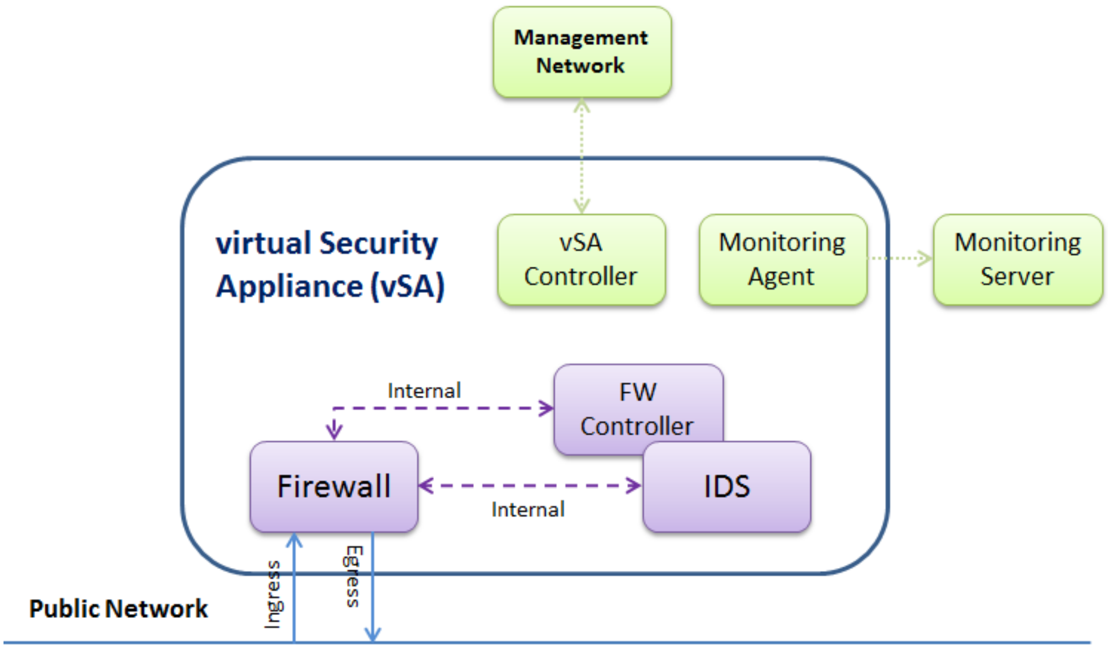


Figure 10. vSA high-level architecture.

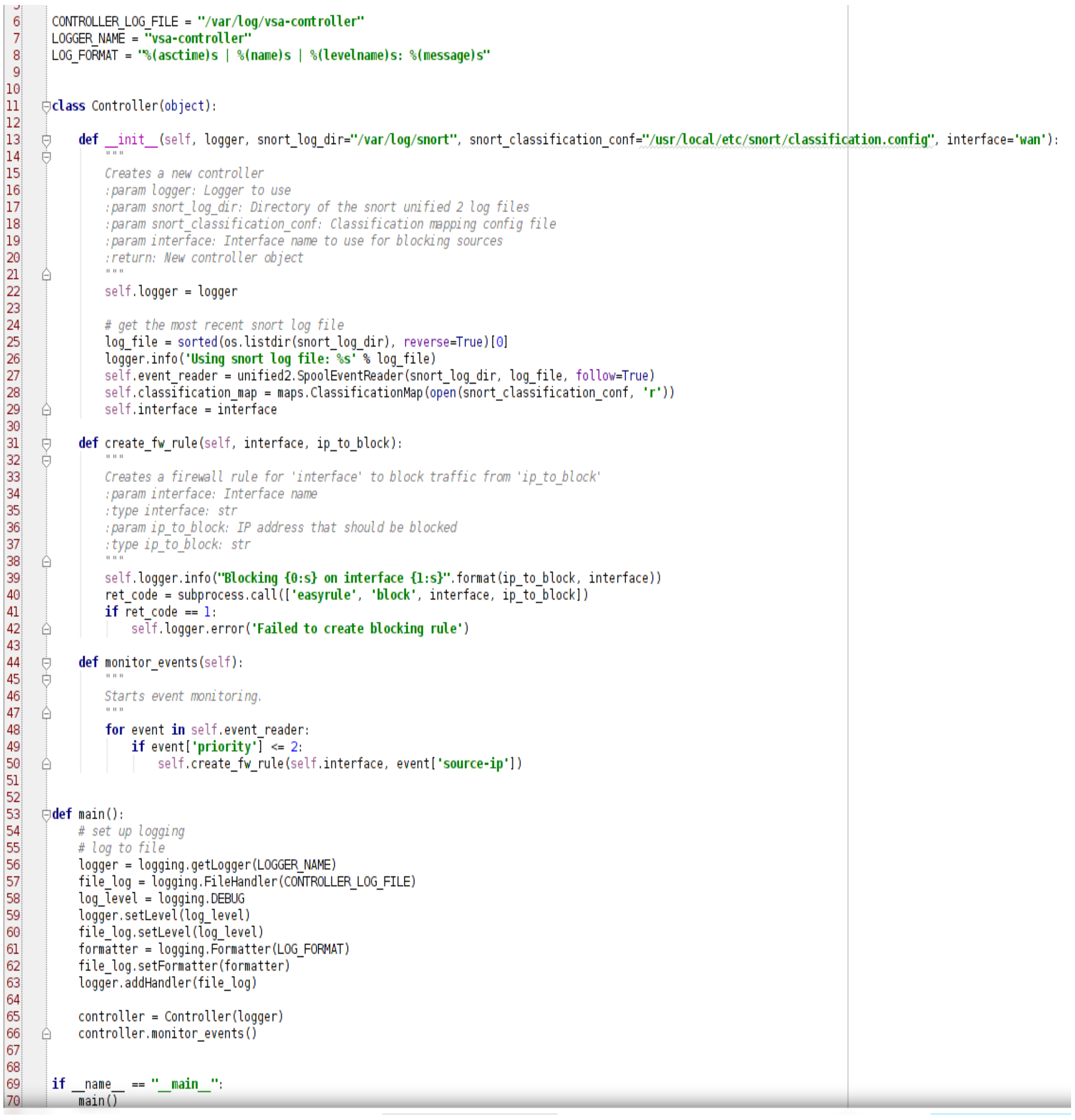
2.2.3. Functional description

The components of the architecture are the following:

- **Firewall:** this component is in charge of filtering the traffic towards the service.
- **Intrusion Detection System (IDS):** in order to improve attack detection, a combination of a packet filtering firewall and an intrusion detection system using both signatures and anomaly detection is considered. In fact, Anomaly detection IDS has the advantage over signature based IDS in detecting novel attacks for which

signatures do not exist. Unfortunately, anomaly detection IDS suffer from high false-positive detection rate. It is expected that combining both arts of detection will improve detection and reduce the number of false alarms. In T-NOVA, the open source signature based IDS [SNORT] is being used and will be extended to support anomaly detection as well. The mode of operation of the IDS component was also discussed in deliverable [D5.01];

- **FW Controller***:* this application looks into the IDS "alerts repository" and based on the related information, the rules of the firewall are revised. Figure 11 depicts a part of the FW Controller code.



```python
CONTROLLER_LOG_FILE = "/var/log/vsa-controller"
LOGGER_NAME = "vsa-controller"
LOG_FORMAT = "%(asctime)s | %(name)s | %(levelname)s: %(message)s"


class Controller(object):

    def __init__(self, logger, snort_log_dir="/var/log/snort", snort_classification_conf="/usr/local/etc/snort/classification.config", interface='wan'):
        """
        Creates a new controller
        :param logger: Logger to use
        :param snort_log_dir: Directory of the snort unified 2 log files
        :param snort_classification_conf: Classification mapping config file
        :param interface: Interface name to use for blocking sources
        :return: New controller object
        """
        self.logger = logger

        # get the most recent snort log file
        log_file = sorted(os.listdir(snort_log_dir), reverse=True)[0]
        logger.info('Using snort log file: %s' % log_file)
        self.event_reader = unified2.SpoolEventReader(snort_log_dir, log_file, follow=True)
        self.classification_map = maps.ClassificationMap(open(snort_classification_conf, 'r'))
        self.interface = interface

    def create_fw_rule(self, interface, ip_to_block):
        """
        Creates a firewall rule for 'interface' to block traffic from 'ip_to_block'
        :param interface: Interface name
        :type interface: str
        :param ip_to_block: IP address that should be blocked
        :type ip_to_block: str
        """
        self.logger.info("Blocking {0:s} on interface {1:s}".format(ip_to_block, interface))
        ret_code = subprocess.call(['easyrule', 'block', interface, ip_to_block])
        if ret_code == 1:
            self.logger.error('Failed to create blocking rule')

    def monitor_events(self):
        """
        Starts event monitoring.
        """
        for event in self.event_reader:
            if event['priority'] <= 2:
                self.create_fw_rule(self.interface, event['source-ip'])


def main():
    # set up logging
    # log to file
    logger = logging.getLogger(LOGGER_NAME)
    file_log = logging.FileHandler(CONTROLLER_LOG_FILE)
    log_level = logging.DEBUG
    logger.setLevel(log_level)
    file_log.setLevel(log_level)
    formatter = logging.Formatter(LOG_FORMAT)
    file_log.setFormatter(formatter)
    logger.addHandler(file_log)

    controller = Controller(logger)
    controller.monitor_events()


if __name__ == "__main__":
    main()
```

Figure 11. A sample of code of the FW Controller

- **Monitoring Agent:** this is a script that reports to the monitoring server the status of the VNF through some metrics such as (Number of errors coming in/ going out of the wan/lan interface of pfsense, Number of bytes coming in/ going out of the wan/lan interface of pfsense, CPU usage of snort, Percent of the dropped packets, generate by snort, etc);
- **vSA controller***:* this is the application in charge of the vSA lifecycle (for more details please refer to section 2.1.1.).

2.2.4. Interfaces

The different components of the architecture interact in the following way,

1. data packets are first of all filtered by the firewall (ingress interface) before being forwarded to the service (egress interface);
2. filtered data packets are sniffed by the IDS for further inspection (internal interface). The IDS will monitor and analyze all the services passing through the network;
3. data packets go through a signature based procedure. This will help in detecting efficiently well know attacks such as port scan attacks and TCP SYN flood attacks;
4. If an attack is detected at this stage, an alarm is generated and the firewall is instructed to revise its rules (internal interface);
5. If no attack is detected, no further action is required.

In addition to that, there are two extra interfaces: the first one is in charge of the vSA lifecycle management, and the second one monitors the status of the vSA and sends the related information to the monitoring server.

2.2.5. Technologies

As performance is one of the main issues when deploying software versions of security appliances, we started by providing a short evaluation of firewalls software that could run in virtual environments. The idea was not to go through all the relevant existing software but just the most popular ones that could be extended to fulfill the use case requirements. This evaluation was described in [D5.01]. It turns out that the open source firewalls that are richer and more complete are Vyatta VyOS and pfSense (please refer to [D5.01] for more details). In addition to that, VyOS seems to support REST APIs for configuration which are important in the integration with the rest of the T-NOVA framework.

These two options were also evaluated from the performance point of view and the results are discussed in section 2.3.6. Based on this assessment, the pfSense firewall seems to be the best option to be used within the vSA.

2.2.6. Dimensioning and Performance

To study the performance of firewalls, appropriate metrics are needed. Although the activities in this area are very scarce, we described in D5.01, potential metrics that could be used. This includes, throughput, latency, jitter, and goodput.

2.2.6.1. Testbed setup

For simplicity reasons, we have used Iperf [IPERF] for generating IP traffic in our tests. In fact, other IP traffic generators such as D-ITG [DITG], ostinato [Ostinato], and IPTraf [IPTR] could have also been utilized. Iperf mainly generates TCP and UDP traffic at different rates. Diverse loads (light, medium, heavy) and different packet sizes are also considered. For analyzing IP traffic, we used “tcpdump” for capturing it and “tcptrace” to analyse it and generate statistics. As for the virtualization, VirtualBox [VBOX] was used.

To run our tests, we decided to use two hosts. On the first one, we have installed the Iperf client and server and on the second one, we have setup the firewall under tests. This setup is in fact in line with the recommendations provided in RFC 2647.

2.2.6.2. Testing scenarios

The undertaken tests are based on three main scenarios,

1. **No firewall:** Here, we configure and check the connectivity between the Iperf client and server without a firewall in between. This enables us to test the capacity of the communication channel
2. **TCP traffic with firewall and no rules:** Here, we check whether the introduction of a firewall (running on a virtual machine in between) generates extra delay. We also test the capacity of the firewall in this context
3. **With firewall and increasing number of rules:** the objective of this scenario is to study the effect of introducing rules into the firewall. To achieve this scenario, some scripts for both pfsense and Vyos are implemented to generate rules in an automatic way. The scripts are shell scripts using specific API commands and generate blocking rules for random source IP addresses (excluding those used in the test setup) and the WAN interface. For pfsense, the easyrule function is extended and for VyOS, the “configure” environment (set of commands) is used. In this scenario, some tests are also performed using UDP instead of TCP

2.2.6.3. Tests results

When no firewall is used between the Iperf client and server, one can note that the throughput of the communication remains good (700 Mbit/s) as long as the number of parallel connections does not exceed 7 connections. When the number of connections goes beyond this value, the throughput decreases very fast to reach 0 when 20 connections are opened (Figure 12, Figure 14). One can also notice that the Round Trip Time (RTT) is severely affected when increasing the number of connections between the Iperf client and server (Figure 13, Figure 15).

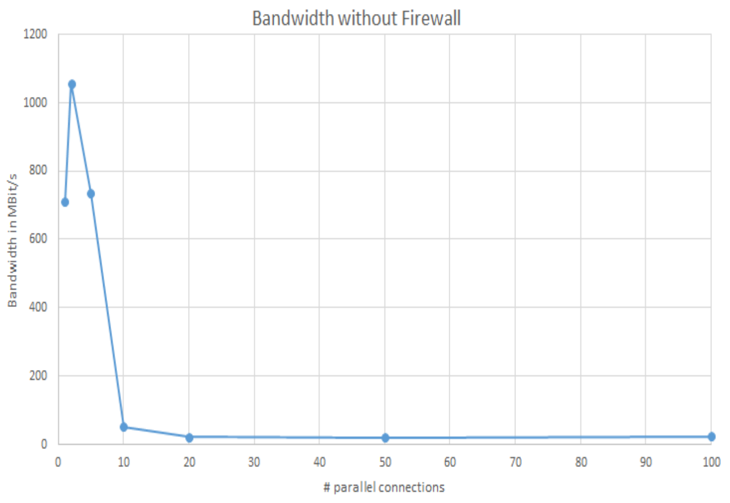


Figure 12. Throughput without firewall

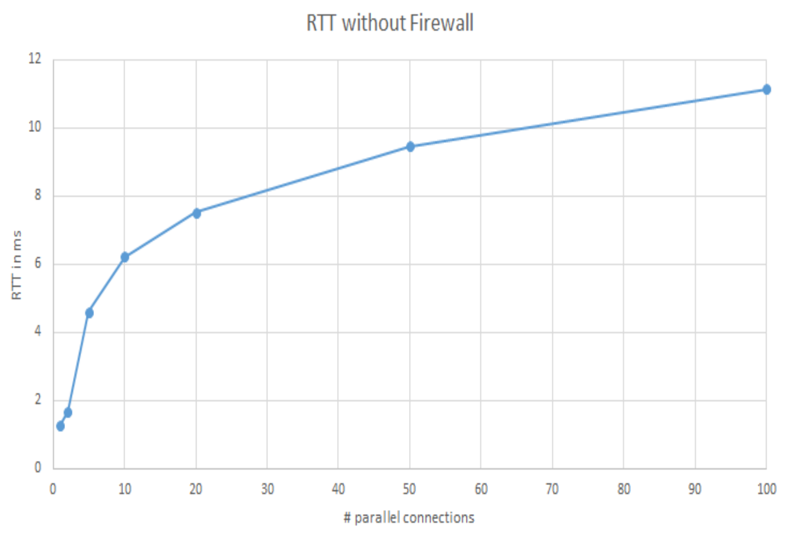


Figure 13. RTT without firewall

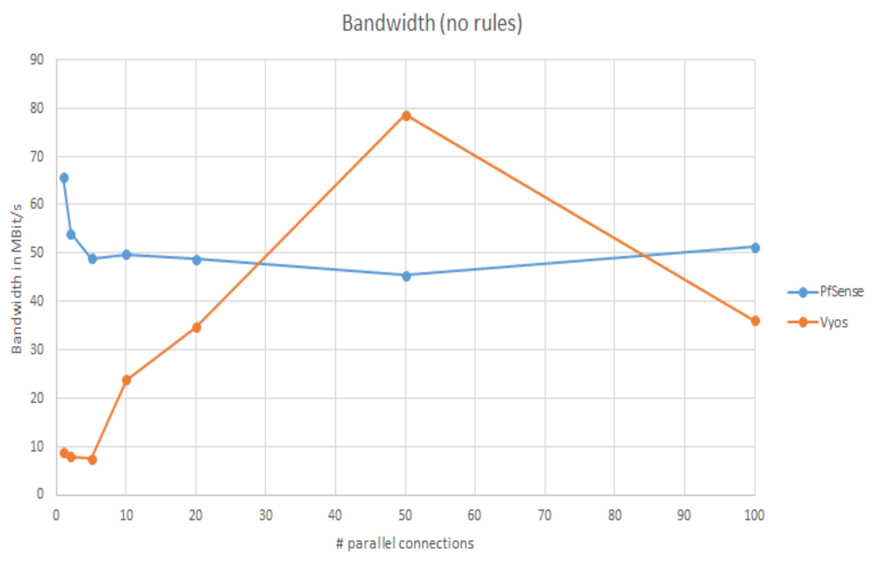


Figure 14. Firewall comparison without rules (Throughput)

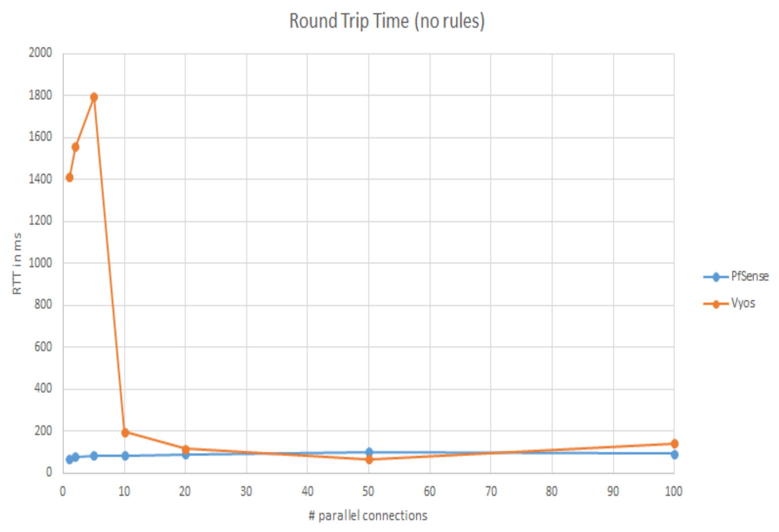


Figure 15. Firewall comparison with 10 rules (RTT)

The results obtained from a firewall (pfsense or Vyos) being settled between the Iperf client and server, the variation of the throughput and the RTT are depicted in Figure 14 and Figure 15, respectively. One can note that pfsense, in both cases, presents a more stable behavior when the number of connections increases.

2.2.7. Future Work

The next steps are the following,

- Move all components to one single VM, as using an OVS in a separate VM for IDS integration is not feasible, due to OpenStack limitations;
- Complete monitoring integration;
- Create Heat template;
- Create VNF descriptor.

2.3. Session Border Controller

2.3.1. Introduction

A Session Border Controller (SBC) is typically a standalone device providing network interconnection and security services between two IP networks. It operates at the edge of these networks and is used whenever a multimedia session involves two different IP domains. It performs:

- the session control on the “control” plane, adopting SIP as a signalling protocol;
- several functions on the “media” plane (i.e : transcoding, transrating, NAT,etc), adopting Real time Transport Protocol (RTP) for multimedia content delivery.

The vSBC is the VNF implementing the SBC service in T-NOVA virtualized environment, and it is a prototyped version of the commercial SBC that Italtel is developing for the NFV market. General requirements for vSBCs comprise both essential features (such as: IP to IP network interconnection, SIP signaling proxy, Media flow NAT, RTP media support) and also advanced requirements (such as : SIP signaling manipulation, real-time audio and/or video transcoding, Topology hiding, Security gateway, IPv4-IPv6 gateway, generation of metrics, etc.). For the objectives of the T-NOVA project, we focus on all essential features and a subset of advanced requirements (i.e: IPv4-IPv6 gateway; real-time audio and/or video transcoding for mobile and fixed network; metrics generation; etc).

2.3.2. Architecture

The basic architecture of the virtualized SBC is depicted in Figure 16.

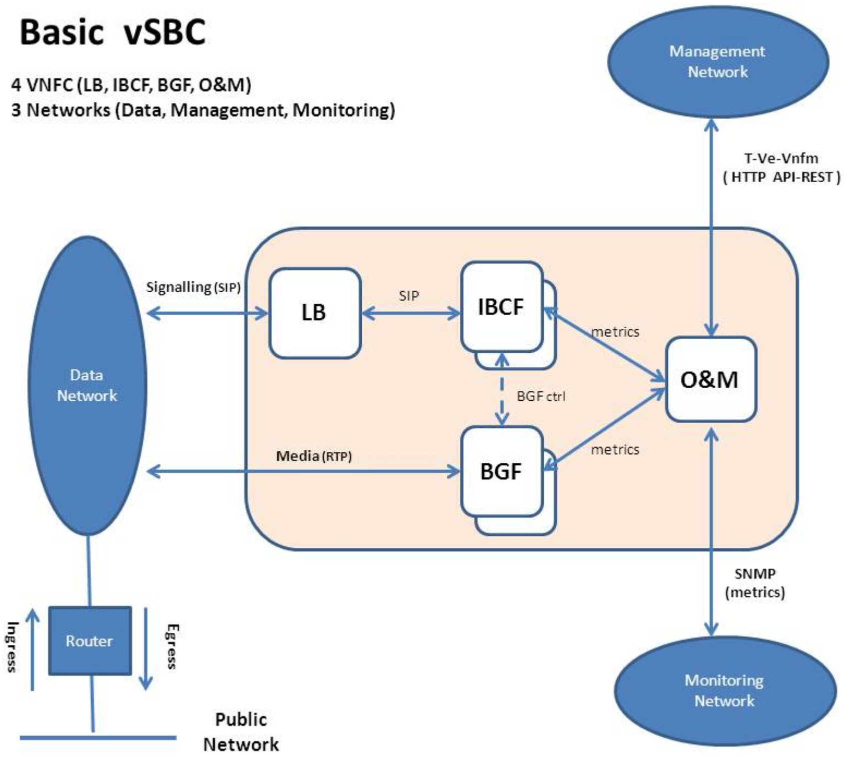


Figure 16. Basic vSBC internal architecture.

The basic vSBC consists mainly of:

- four Virtual Network Function Components (VNFCs) : LB, IBCF, BGF and O&M (described in detail below);
- four NICs;
- one Management interface (T-Ve-Vnfm) : it transports the HTTP commands of T-Nova lifecycle from the VNFM to the O&M component;
- one Monitoring Interface : the monitoring data, produced by the internal VNFCs (i.e: IBCF and BGF), are collected by the O&M and are cyclically sent (via snmp) to the T-NOVA Monitoring. See also Task 4.4 (Monitoring and Maintenance) for further details.

The enhanced architecture of the virtualized SBC comprises also the DPS component. This additional component, described in detail in the following paragraph, is based on DPDK acceleration technology and can provide high speed in processing the addressing information in the header of IP packets. The following Figure 17 depicts the enhanced architecture of the vSBC.

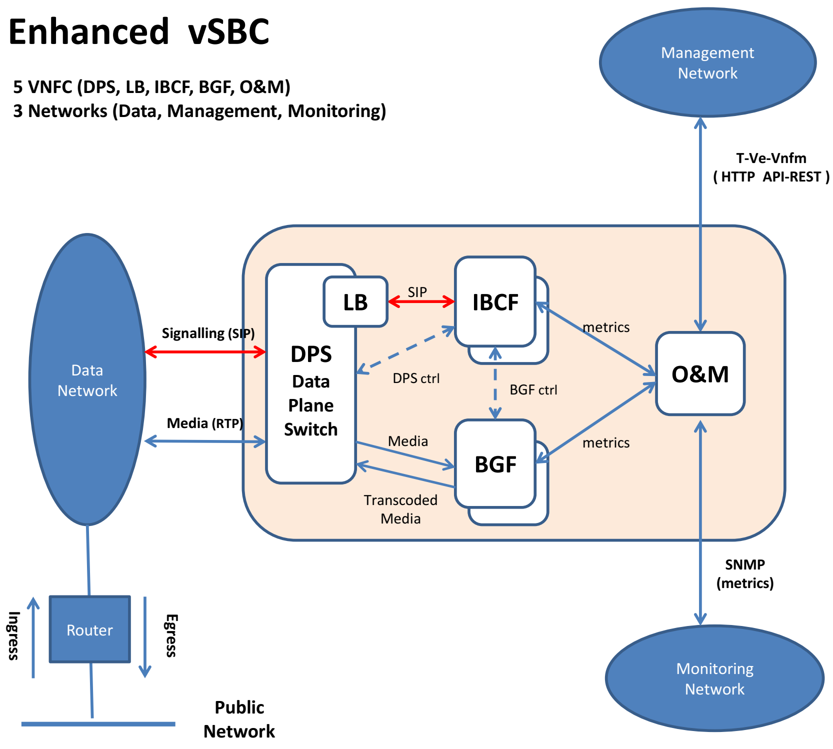


Figure 17. Enhanced vSBC internal architecture (with DPS component).

The enhanced vSBC architecture consists of:

- five Virtual Network Function Components (VNFCs) : DPS, LB, IBCF, BGF and O&M (described in detail below);
- two NICs in case the signalling and media, incoming and outgoing, are handled by DPS with the same NIC;
- one Management interface (T-Ve-Vnfm), as described in the basic architecture;
- one Monitoring Interface, as described in the basic architecture.

2.3.3. Functional description

- **Load Balancer (LB):** it balances the incoming SIP messages, forwarding them to the appropriate IBCF instance.
- **Interconnection Border Control Function (IBCF):** it implements the control function of the SBC. It analyzes the incoming SIP messages, and handles the communication between disparate SIP end-point applications. The IBCF extracts from incoming SIP messages the information about media streams associated to the SIP dialog, and instructs media plane components (DPS and/or BGF) to process them. It can also provide:
    - SIP message adaptation or modification, enabling in this way the interoperability between the interconnected domains
    - topology hiding (IBCF hides all incoming topological information to the remote network)
    - monitoring data (IBCF can send this information to the T-NOVA monitoring agent)

    - other security features.

- **Border Gateway Function (BGF):** it processes media streams, applying transcoding and transrating algorithms when needed (transcoding transforms the algorithm used for coding the media stream, while transrating changes the sending rate of IP packets carrying media content). This feature is used whenever the endpoints of the media connection support different codecs, and it is an ancillary function for an SBC because, in common network deployments, only a limited subset of media streams processed by the SBC need to be transcoded. The BGF is controlled by the IBCF using the internal BG ctrl interface (see Figure 16). If the transcoding /transrating function is implemented by a pure software transcoder its performance dramatically decreases, unless GPU (Graphical Processing Units) hardware acceleration is available in the virtual environment. Otherwise this issue could be mitigated by running more BGF instances. The BGF component can also provide metrics to the T-NOVA monitoring agent**;**
- **Operating and Maintenance (O&M):** it supervises the operating and maintenance functions of the VNF. In a cloud environment, the O&M module extends the traditional management operations handled by the Orchestrator (i.e.: scaling). The O&M component interacts (via HTTP) with the VNF manager, using the T-Ve-Vnfm interface depicted in Figure 16, for applying the T-NOVA lifecycle**;**
- **Data Plane Switch (DPS):** it is the (optional) front-end component of the vSBC based on DPDK acceleration technology available in Intel x86 architectures to reach a performance level comparable to the hardware based version adopting HW acceleration technologies. This component can use the same IP address, as ingress or egress point, both for signalling and media flows. Its goal is to provide high speed in processing the addressing information in the header of the IP packet, leaving untouched the payload. The DPS is instructed how to manage the IP packets by the IBCF component, acting as an external controller using an internal dedicated DPS ctrl interface (see Figure 17). The DPS component can:

    - either providing the packet forwarding to the BGF (in case of transcoding)
    - or applying a local Network Address Translation (NAT)/port translation

Note: DPS may be on optional component of the vSBC. In this case the LB component acts as the front-end of the VNF, of course with poorer performances.

2.3.4. Interfaces

The most relevant internal and external interfaces depicted in Figure 16 are:

1. **DSP ctrl:** this (enhanced) internal interface instructs the Data Plane Switch (DPS) to perform the incoming media packet (forwarding them to the BGF, or applying NAT). It is used only in case of the enhanced vSBC;
2. **BG ctrl:** this internal interface instructs the Border Gateway Function (BGF) to perform media stream transcoding/transrating between two codecs;
3. **Management Interface (T-Ve-Vnfm):** it is used to transport the HTTP commands of the T-NOVA lifecycle (i.e: start, stop, destroy, scale in/out, etc). It’s supported by the O&M component;

4. **Monitoring Interface:** the monitoring data, produced by the internal VNFCs (i.e: IBCF and BGF), are collected by the O&M and cyclically sent (via snmp) to the T-NOVA Monitoring Manager.

2.3.5. Technologies

The development of the vSBC is based on the use of:

- Linux operating system
- KVM hypervisor

Services provided by telecommunication networks greatly differ from standard IT applications running in the cloud in terms of carrier grade availability and high processing throughputs. For this reason:

- the pure software implementation, in some scenarios, shall be complemented by acceleration technologies available also in cloud environments. For example the DPS component uses the DPDK acceleration technology (available in Intel x86 architectures) to provide high speed in processing the addressing information in the header of IP packet;
- the transcoding/transrating feature, provided by the BGF component, benefits from GPU acceleration, so that a real time transcoding and a fixed video-rate can be guaranteed during the whole audio/video session. The GPU encoding algorithms are able to efficiently exploit the potential of all available cores. The overall system consists of a commercial GPU board [Nvidia], hosted in a PCIe bay. The transcoding will be performed and tested between domains using the most common codecs (i.e: Google's VP8 and ITU-T H.264).

2.3.6. Dimensioning and Performance

The vSBC performances can be monitored using the metrics generated by its internal VNFCs (see also [D4.41] for further information), for example:

1. monitoring data related to the control plane: total number of SIP sessions/transactions, number of failed SIP sessions/transactions due to vSBC internal problems;
2. monitoring data related to the media plane: incoming/outgoing RTP data throughput, RTP frame loss, latency, inter-arrival jitter, number of transcoding/transrating procedures, number of failed transcoding/transrating procedures due to vSBC internal problems;
3. base monitoring data: percentage of memory consumption, percentage of CPU utilization.

These monitoring data are strongly affected by:

- packet sizes;
- kinds of call (i.e: audio or video calls);
- audio/video codecs (i.e: H.264, VP8, …);
- transport protocols (i.e: UDP and TCP).

At the moment we don't have data related to the performance of the vSBC. Nevertheless we have target requirements about the expected behaviour of its internal VNFCs. For example IBCF and BGF components provide market sensitive performances: IBCF shall support up to a certain number of simultaneous SIP sessions (i.e: 10, 25, 50, 100, 500, 1000), while the BGF up to a certain number of simultaneous transcoding/transrating operations (i.e: 20). For this reason, in the commercial product, each component size will be associated to a license fee.

2.3.6.1. vSBC testing

These tests/implementations have been carried out so far:

- Creation of a JSON descriptor for the vSBC;
- Creation of a HEAT template (Hot) for the vSBC;
- Instantiation of the vSBC, using the HEAT template, in an Italtel test bed.

These further scenarios will be tested in the following development steps:

- Support of the following T-NOVA lifecycle events (using a HTTP REST-based interface and a basic access authentication):

    - Start (via http POST)
    - Stop (via http PUT), both immediate and graceful[1]
    - Destroy (via http DELETE)

- Basic audio sessions (without trascoding), using the most common audio codecs (i.e: G711, G729, …);
- Basic video sessions (without trascoding), using the most common video codecs (i.e: H248, VP8, …);
- Audio sessions with transcoding (for example G711 <-> G729);
- Video sessions with transcoding (for example H248 <-> VP8) :

    - without GPU
    - with GPU(NVIDIA family).

    The requested transcoding may be mono-directional (i.e: audio/video stream distribution) or bidirectional (i.e.: videoconferencing applications). The supported video resolution may be :

    - PAL (576x720)
    - 720p (1280x720)
    - HD (1920x1080)
    - 4k (3840x2160)

    We will check whether the introduction of a GPU accelerator increases the number of simultaneous video transcoding sessions offered by the vSBC. The encoding/decoding procedures must be handled in a real-time way (at least 30 fps) and with a fixed frame-rate during the whole video session.

- Insertion, inside the basic vSBC architecture, of the new DPS component in order to optimize NAT scenarios without transcoding. The aim is to test and compare performances of the basic architecture with those ones offered by the new DPS component;

[1] Note: this lifecycle event will be handled in an immediate or graceful mode according to a specific vSBC internal data.

- Test the proper evaluation of monitoring parameters, both for media and control plan, generated by the internal VNFCs in all scenarios previously described;
- Scale-in scenario;
- Scale-out scenario.

2.3.7. Future Work

Future work objectives:

1. to complete the harmonization about the vSBC interfaces with the other components of T-NOVA project (i.e: VNF manager);
2. to insert a GPU accelerator for trascoding/transrating procedures;
3. to test the two most innovative components of vSBC (DPS and BGF) in a deployment scenario with and without hardware/software accelerators available;
4. to complete the implementation of the monitoring parameters, used both for scaling procedures (in/out) and for SLA analysis;
5. to clarify the scale in/out scenarios;

2.4. Video Transcoding Unit

2.4.1. Introduction

The vTU provides the transcoding function for the benefit of many other VNFs for creating enhanced services.

2.4.2. Architecture

2.4.2.1. Functional description

The core task of the Video Transcoding Unit is to convert video streams from one video format to another. Depending on the applications, the source video stream could originate from a file within a storage facility, as well as coming in from of a packetized network stream from another VNF. Moreover, the requested transcoding could be mono-directional, as in applications like video stream distribution, or bi-directional, like in videoconferencing applications.

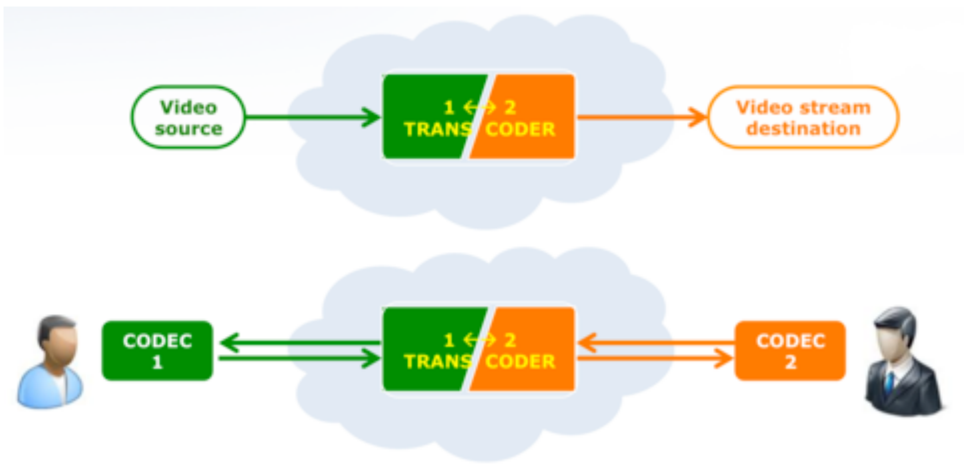


Figure 18. Functional description of the vTU

Having this kind of applications in mind, it is clear that the most convenient overall architecture for this Unit is a modular architecture, in which the necessary encoding and decoding functionalities are deployed as plug-in within a “container” Unit taking care for all the communication, synchronization and interfacing aspects. In order to find a convenient approach for the development of the vTU architecture, an investigation has been carried out, about the state of the art of any available software framework that could be usefully employed as starting point for this architecture. This investigation has identified **avconv**, the open-source audio-video library under Linux environments (https://libav.org/avconv.html), as the best choice for the basic platform for the vTU, as it is open-source, it is modular and customizable, and it contains most of the encoding/decoding plug-ins that this VNF could need.

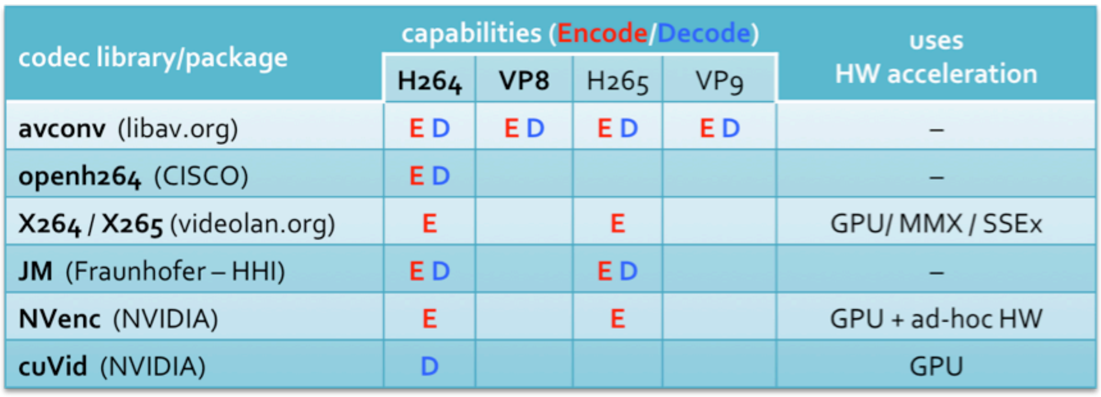
In order to define the functionalities that best fit to the needs of the target applications for the vTU, a survey has been carried out, searching for the most diffused video formats that should therefore be present as encoding/decoding facilities in the vTU. This analysis has shown that the following video formats should be primarily considered:

- ITU-T **H.264** (aka AVC)
- Google’s **VP8**

and the following ones would be also highly desirable, especially in the future:

- ITU-T **H.265** (aka HEVC)
- Google’s **VP9**.

Once the video formats of interest are defined, the whole panorama of the available codecs have been considered and evaluated, in order to identify tools which could be successfully employed in the Unit and those which could be possibly used as development basis. A synthesis of the available panorama is shown in the following table:

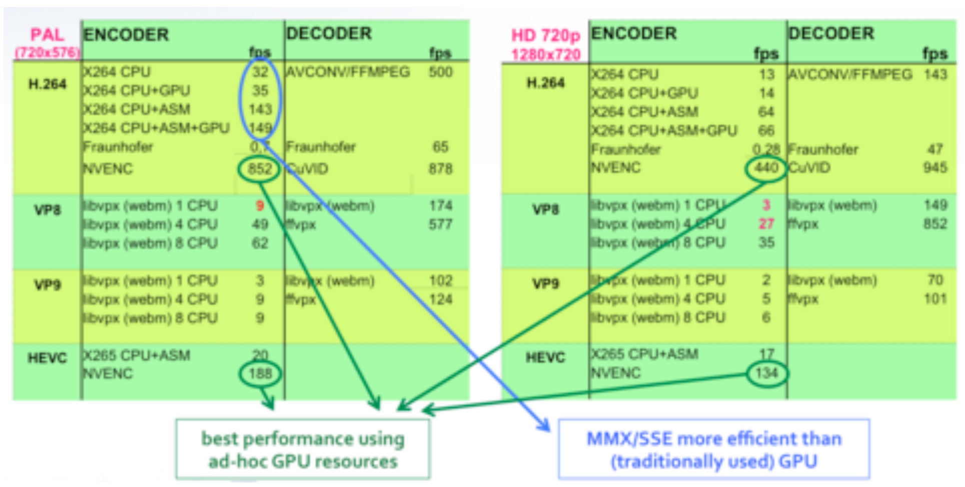


codec library/package	capabilities (Encode/Decode)				uses HW acceleration
	H264	VP8	H265	VP9	
avconv (libav.org)	E D	E D	E D	E D	–
openh264 (CISCO)	E D				–
X264 / X265 (videolan.org)	E		E		GPU/ MMX / SSEx
JM (Fraunhofer – HHI)	E D		E D		–
NVenc (NVIDIA)	E		E		GPU + ad-hoc HW
cuVid (NVIDIA)	D				GPU

The analysis evidenced that the choice of avconv as the starting development framework is most appropriate in terms of already-available codecs.

From the point of view of performance, however, avconv could be unable to fulfil the needs of the vTU, as all the endoders/decoders provided therein do not make any use of HW acceleration facilities. Performance, in terms of encoding/decoding speed, is actually of crucial importance in the vTU, as in all online applications, a fixed video frame-rate must be guaranteed during the whole video session.

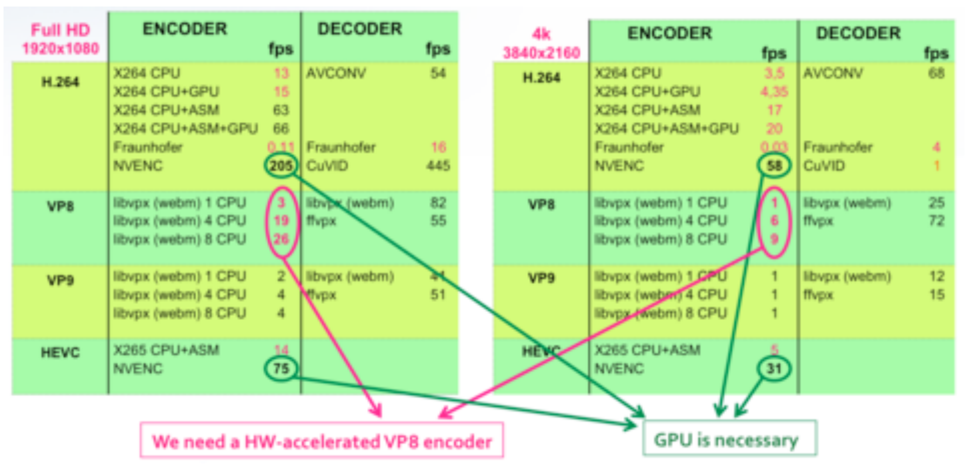
For this reason, a performance analysis of all available codecs has been carried out. Several of them, in fact, are able to exploit hardware acceleration facilities, like GPU’s or MMX/SSEx instructions, whenever they are available. The tests have been carried out considering a typical scenario for the underlying hardware infrastructure: a Xeon server with 8 cores (2 x 4-cores) Xeon E5-2620v3, equipped with GPU facility (1 NVIDIA GeForce GTX 980). Several video test sequences have been considered, at the most common resolutions. The obtained results, in terms of achieved encoded/decoded frames per second, are summarized in the following tables, for PAL (576x720 pixel), 720p (1280x720), HD (1920x1080) and 4k (3840x2160) resolutions:



PAL (720x576)	ENCODER	fps	DECODER	fps
H.264	X264 CPU	32	AVCONV/FFMPEG	500
	X264 CPU+GPU	35		
	X264 CPU+ASM	143		
	X264 CPU+ASM+GPU	149		
	Fraunhofer	0,7	Fraunhofer	65
	NVENC	852	CuVID	878
VP8	libvpx (webm) 1 CPU	9	libvpx (webm)	174
	libvpx (webm) 4 CPU	49	ffvpx	577
	libvpx (webm) 8 CPU	62		
VP9	libvpx (webm) 1 CPU	3	libvpx (webm)	102
	libvpx (webm) 4 CPU	9	ffvpx	124
	libvpx (webm) 8 CPU	9		
HEVC	X265 CPU+ASM	20		
	NVENC	188		

HD 720p 1280x720	ENCODER	fps	DECODER	fps
H.264	X264 CPU	13	AVCONV/FFMPEG	143
	X264 CPU+GPU	14		
	X264 CPU+ASM	64		
	X264 CPU+ASM+GPU	66		
	Fraunhofer	0.28	Fraunhofer	47
	NVENC	440	CuVID	945
VP8	libvpx (webm) 1 CPU	3	libvpx (webm)	149
	libvpx (webm) 4 CPU	27	ffvpx	852
	libvpx (webm) 8 CPU	35		
VP9	libvpx (webm) 1 CPU	2	libvpx (webm)	70
	libvpx (webm) 4 CPU	5	ffvpx	101
	libvpx (webm) 8 CPU	6		
HEVC	X265 CPU+ASM	17		
	NVENC	134		

best performance using ad-hoc GPU resources
MMX/SSE more efficient than (traditionally used) GPU



Full HD 1920x1080	ENCODER	fps	DECODER	fps
H.264	X264 CPU	13	AVCONV	54
	X264 CPU+GPU	15		
	X264 CPU+ASM	63		
	X264 CPU+ASM+GPU	66		
	Fraunhofer	0.11	Fraunhofer	16
	NVENC	205	CuVID	445
VP8	libvpx (webm) 1 CPU	3	libvpx (webm)	82
	libvpx (webm) 4 CPU	19	ffvpx	55
	libvpx (webm) 8 CPU	26		
VP9	libvpx (webm) 1 CPU	2	libvpx (webm)	41
	libvpx (webm) 4 CPU	4	ffvpx	51
	libvpx (webm) 8 CPU	4		
HEVC	X265 CPU+ASM	14		
	NVENC	75		

4k 3840x2160	ENCODER	fps	DECODER	fps
H.264	X264 CPU	3,5	AVCONV	68
	X264 CPU+GPU	4,35		
	X264 CPU+ASM	17		
	X264 CPU+ASM+GPU	20		
	Fraunhofer	0.03	Fraunhofer	4
	NVENC	58	CuVID	1
VP8	libvpx (webm) 1 CPU	1	libvpx (webm)	25
	libvpx (webm) 4 CPU	6	ffvpx	72
	libvpx (webm) 8 CPU	9		
VP9	libvpx (webm) 1 CPU	1	libvpx (webm)	12
	libvpx (webm) 4 CPU	1	ffvpx	15
	libvpx (webm) 8 CPU	1		
HEVC	X265 CPU+ASM	5		
	NVENC	31		

Based on the obtained results, the following observations can be drawn:

- As expected, encoding is much more time-consuming than decoding. On average, decoding is approximately 20 times faster than encoding, for the same format. The consequence is that encoders represent the bottleneck to performance in a vTU. For what decoding concerns, the tested tools have performed faster than 30 fps in all situations, at least for H264 and VP8 standards, which are those currently use;

- Hardware-accelerated tools are not only providing much better performance than CPU-based ones, as visible in the tables for all resolutions, but is necessary in some cases, e.g. for 4k resolutions, where standard algorithms are not able to reach 30 fps encoding speed and therefore could not support a real-time transcoding session;

- As highlighted in the first table, different hardware accelerators can be successfully exploited to speed-up the encoding process – not only GPU's. In particular, X264 performs significantly better using Assembly-level optimizations which exploit SIMD instructions (MMX/SSE), than delegating computation to GPU cores.

This is due to the fact that encoding/decoding algorithms cannot be massively parallelized, for two main reasons: a) there are strong sequential correlations and many spatial/temporal dependencies within the computation, and b) the limited extent of parallelism needed in all situations where data parallelism could be applied (e.g. computing DCT/IDCT for a macroblock). Nevertheless, the huge computing power of modern GPU's makes it reasonable to focus the research effort towards the development of GPU-accelerated encoding algorithms, able to efficiently exploit the potential of all available cores. Therefore, as shown in the table above, the first goal on which we focus is the development of a GPU-accelerated encoder for the VP8 standard video format.

2.4.3. Interfaces

As described in Section 2.4.2.1, the Virtual Transcoding Unit (vTU) is a VNF that, during its normal operation, receives an input video stream, transcodes it and generates an output video stream in the new format. For each transcoding job, the vTU also needs to receive a proper job description, in which all necessary information is provided, like, for instance, information on the video format of the input stream and on the desired video format for the output stream, the identification and definition of the input/output data channels (e.g. IP addresses and ports, in case of network streams, or file ID within a storage facility, for file-generated streams).

For these reasons, the vTU needs, at its inner level, to communicate through three interfaces, as Figure 19 shows:

- **Input** port, receiving the video stream to transcode;
- **Output** port, producing the transcoded video stream;
- **Control** port, receiving the job descriptor and, implicitly, the command to start the job.

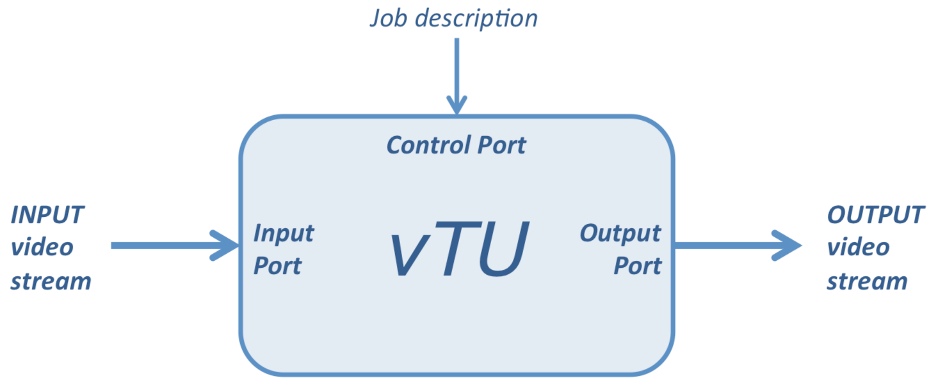


Figure 19. vTU low level interfaces

Through the Control interface, the vTU receives the job descriptor, which contains all necessary information to start the requested transcoding task. The starting command is implicit in the reception of the job description message: when such a message is received on the Control port, the vTU starts listening at the Input port and begins the transcoding task, according to the received description, as soon as the first stream packets are received.

The format of the job description message is XML based. The general structure of the message is shown in Figure 20. This format allows to define all necessary parameters, such as the desired input and output video formats and the I/O stream channels (files, in this case, but they could as well identify network channels sending/receiving RTP packets).

```
<vTU>
  <in>
    <local>
      <stream>test.y4m</stream>
    </local>
    <rstp>
      <ip/>
      <port/>
      <stream/>
      <timeout/>
    </rstp>
    <codec>
      <vcodec/>
      <acodec/>
    </codec>
  </in>
  <out>
    <local>
      <overwrite>y</overwrite>
      <stream>out_test.h264</stream>
    </local>
    <rstp>
      <ip/>
      <port/>
      <stream/>
      <timeout/>
    </rstp>
    <codec>
      <vcodec>h264</vcodec>
      <acodec/>
    </codec>
  </out>
</vTU>
```

Figure 20. XML structure of the vTU job description message.

2.4.4. Technologies

In the virtualization context, the problem of virtualizing a GPU is now well-known, and can be stated as follows: a guest Virtual Machine (VM), running on a hardware platform provided with GPU-based accelerators, must be able to concurrently and independently access the GPU's, without incurring in security issues [Walters],[Maurice]. Many techniques to achieve GPU virtualization have been presented. However, all the proposed methods can be divided in two main categories, which are usually referred to as API remoting [Walters] (also known as split driver model or driver paravirtualization) and PCI pass-through [Walters] (also known as direct device assignment [Maurice]), respectively. In the vTU, the passthrough approach has been adopted. For the sake of clarity, a brief review of this technology is shortly given in the next paragraph.

Pass-through techniques are based on the pass-through mode made available by the PCI-Express channel [Walters], [Maurice]. To perform PCI pass-through, an Input/Output Memory Management Unit (IOMMU) is used. The IOMMU acts like a traditional Memory Management Unit, i.e. it maps the I/O address space into the CPU virtual memory space, so enabling the access of the CPU to peripheral devices through Direct Memory Access channels. The IOMMU is a hardware device which provides, besides I/O address translations, also device isolation functionalities, thus guaranteeing secure access to the external devices [Walters]. Currently, two IOMMU implementations exist, one by Intel (VT-d) and one by AMD (AMD-Vi). To adopt the pass-through approach, this technology must also be supported by the adopted hypervisor. Nonetheless, Xenserver, open source Xen, VMWare

ESXi, KVM and also the Linux containers can support pass-through, thus allowing VMs accessing external devices such as accelerators in a secure way [Walter]. The performance that can be achieved by the pass-through approach are usually higher than the one offered by API-remoting [Walter], [Maurice]. Also, the pass-through method gives immediate access to the latest GPU drivers and development tools [Walter]. A comparison between the performance achievable using different hypervisors (including also Linux Containers) is given in [Walter], where it is shown that pass-through virtualization of GPU's can be achieved at low overhead, with the performance of KVM and of Linux containers very closed to the one achievable without virtualization. One major drawback of pass-through is that it can only assign the entire physical GPU accelerator to one single VM. Thus, the only way to share the GPU is to assign it to the different VMs one after the other, in a sort of "time sharing" approach [Walters]. This limitation can be overcome by a technique also known as Direct Device Assignment with SR-IOV (Single Root I/O Virtualization) [Walters]. A single SR-IOV capable device can expose itself as multiple, independent devices, thus allowing concurrent hardware multiplexing of the physical resources. This way, the hypervisor can assign an isolated portion of the physical device to a VM; thus, the physical GPU resources can be concurrently shared among different tenants. However, to the best of the author's knowledge, the only GPU enabled to this functionality belongs to the recently launched NVIDIA Grid family [Maurice], [Walters]; also, the only hypervisors which can currently support this type of hardware virtualization are VMWare Sphere and Citrix XenServer 6.2. However, since also KVM can now support SR-IOV, there is a path towards the use of GPU hardware virtualization also with this hypervisor [Walters].

2.4.5. Dimensioning and Performance

In order to obtain a realistic assessment of the performance of the vTU, as if it was embedded as a VNF in the T-NOVA framework, it is necessary to perform the tests on a virtualized platform resembling as much as possible the T-NOVA infrastructure.

The performance results presented in the table of Section 2.4.2.1. were obtained by the vTU running natively on physical computation resources. For a VNF like the vTU, however, the actual performance achievable in the T-NOVA environment could be quite different from those obtained running on the physical infrastructure. This is mainly due to the following reasons:

- CPU virtualization overheads (vCPU's switching over physical CPU's, at the hypervisor level);
- GPU virtualization strategies (e.g. multiple vGPU's associated to the same physical GPU);
- vCPU-vGPU communication overheads (switching overheads in managing time-sharing policies on the PCI-Express bus).

These reasons let one expect a possible performance loss, when running on a virtualized environment, especially in case of vTU running tasks which exploit the GPU resources.

For this reason, in order to obtain a realistic evaluation of the encoding/decoding computation speeds in the actual T-NOVA environment, the performance tests presented in Section 2.4.2.1. have been carried out on a virtualized environment. In order to get a significant comparison, the VM running the vTU has been equipped with the same amount of CPU and GPU cores as in the native tests. The following table presents the obtained results, in terms of computation speed (frames/sec), compared to the speed obtained on physical resources, for the same task.

Full HD (1920x1080)	ENCODER	Native	Virtualized	DECODER	Native	Virtualized
		time (s)	time (s)		time (s)	time (s)
H.264	X264 CPU	6,97	7,26	AVCONV/FFMPEG	0,39	0,6
	X264 CPU+GPU	7,78	10,08			
	X264 CPU+ASM	1,55	1,85			
	X264 CPU+ASM+GPU	5,81	4,23			
	Fraunhofer	846	869	Fraunhofer	6,85	6,7
	NVENC	1,64	2,57	CuVID	1,53	2,24
	NVENC (AvConv)	2,4	2,64			
VP8	libvpx (webm) 1 thread	26,64	27,79	libvpx (webm)	0,37	4,21
	libvpx (webm) 4 thread	10,14	13,56	ffvpx	1,15	0,42
	libvpx (webm) 8 thread	8,33	7,2			
VP9	libvpx (webm) 1 thread	46,23	50,77	libvpx (webm)	2,23	1,94
	libvpx (webm) 4 thread	23,35	26,64	ffvpx	1,75	2,22
	libvpx (webm) 8 thread	18,87	26,21			
HEVC	X265 CPU+ASM	6,29	6,52	AVCONV/FFMPEG		1,19
	NVENC	2,9	1,71			

4k (3840x2160)	ENCODER	Native	Virtualized	DECODER	Native	Virtualized
		time (s)	time (s)		time (s)	time (s)
H.264	X264 CPU	25,91	30,72	AVCONV/FFMPEG	1,38	1,74
	X264 CPU+GPU	22,21	24,36			
	X264 CPU+ASM	5,18	6,71			
	X264 CPU+ASM+GPU	5,89	9,81			
	Fraunhofer	3117	4714	Fraunhofer	22,37	24,49
	NVENC	2,72	8,63			
	NVENC (AvConv)	5,24	7,03			
VP8	libvpx (webm) 1 thread	78,63	76,4	libvpx (webm)	1,21	4,26
	libvpx (webm) 4 thread	32,44	36,78	ffvpx	4,37	1,54
	libvpx (webm) 8 thread	25,83	22,78			
VP9	libvpx (webm) 1 thread	146,19	155,2	libvpx (webm)	7,31	9,54
	libvpx (webm) 4 thread	60,7	75,23	ffvpx	5,43	9,16
	libvpx (webm) 8 thread	63,21	53,94			
HEVC	X265 CPU+ASM	6,29	17,88	AVCONV/FFMPEG	20,94	3,03
	NVENC	8,11	5,16			

As the table shows, the obtained results show that, for almost all the considered tasks, there is no significant performance loss with respect to the same task running on physical resources, even in the tasks running mainly on GPU (like H264 encoding using NVENC). This encouraging result is mainly due to the high efficiency of the adopted GPU virtualization strategy – GPU pass-through – which assigns a virtual GPU exclusively to a physical GPU, thus allowing to bypass any overhead in the GPU-CPU communication. The cost for this efficiency, however, is paid in terms of difficulty to share a physical GPU resource among multiple VMs.

2.4.6. Future Work

Two main steps are foreseen for the vTU. A first activity will focus on scaling mechanism for this VNF. Also, the vTU will be combined with other VNFs developed within T-Nova in order to create new service with a wider scope.

2.5. Traffic Classifier

2.5.1. Introduction

The Traffic Classifier (TC) VNF used comprises of two Virtual Network Function Components (VNFCs), namely the Traffic Inspection engine and Classification and Forwarding function. The two VNFCs are implemented in respective VMs. The proposed Traffic Classification solution is based upon a Deep Packet Inspection (DPI) approach, which is used to analyze a small number of initial packets from a flow in order to identify the flow type. After the flow identification step no further packets are inspected. The Traffic Classifier follows the Packet Based per Flow State (PBFS) in order to track the respective flows. This method uses a table to track each session based on the 5-tuples (source address, destination address, source port, destination port, and the transport protocol) that is maintained for each flow.

2.5.2. Architecture

Both VNFCs can run independently from one another, but in order for the VNF to have the expected behaviour and outcome, the 2 VNFCs are required to operate in a parallel manner.

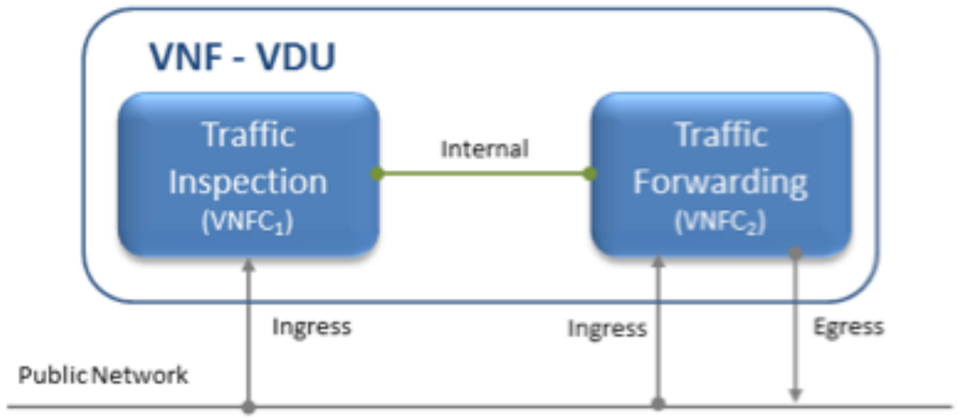


Figure 21. Virtual Traffic Classifier VNF internal VNFC topology

Furthermore, in order to achieve the parallel processing of the 2 VNFCs it is required for the traffic to be mirrored towards the 2 VNFCs, so the 2 VNFCs receive identical traffic. The 2 VNFCs are inter-connected internally with an internal virtual link, which transfers the information extracted by the Traffic Inspection VNFC, and transmits it to the Traffic Forwarding VNFC in order to apply the pre-defined rules.

2.5.3. Functional Description

The Traffic Inspection VNFC is the most processing intense component of the VNF. It implements the filtering and packet matching algorithms in order to support the enhanced traffic forwarding capability of the VNF. The component supports a flow table (exploiting hashing algorithms for fast indexing of flows) and an inspection engine for traffic classification.

The Traffic Forwarding VNFC component is responsible for routing and packet forwarding. It accepts incoming network traffic, consults the Flow Table for classification information for each incoming flow and then applies pre-defined policies (i.e. TOS/DSCP (Type of

Service/Differentiated Services Code Point) marking for prioritizing multimedia traffic) on the forwarded traffic. It is assumed that the traffic is forwarded using the default policy until it is identified and new policies are enforced. The expected response delay is considered to be negligible, as only a small number of packets are required to achieve the identification. In a scenario where the VNFCs are not deployed on the same compute node, traffic mirroring may introduce additional overhead.

2.5.4. Interfaces

The virtual Traffic classifier VNF is based upon the T-NOVA network architecture but from the advised set of network interfaces (management, datapath, monitoring and storage) uses the management, datapath and the monitoring. The storage interface is not particularly essential to the vTC, as all the computational and packet processing utilize mostly CPU and memory resources. The VNF requires intensive CPU tasks and a large number in memory I/Os for the traffic analysis, manipulation and forwarding. The storage interface would add an unnecessary overhead to the already intensive process, and it was decided to be excluded in favour of an optimal performance.

2.5.5. Technologies

The vTC utilizes various technologies in order to offer a stable and high performance VNF compliant to the high standards of legacy physical network functions. The implementation for the traffic inspection used for these experiments is based upon the open source nDPI library [REFnDPI]. The packet capturing mechanism is implemented using various technologies in order to investigate the trade-off between performance and modularity. The various packet handling/forwarding technologies are:

- **PF_RING**: PF_RING is a set of library drivers and kernel modules, which enable high-throughput, packet capture and sampling. For the needs of the vTC the PF_RING kernel module library is used, which is polling the packets through the LINUX NAPI. The packets are copied from the kernel to the PF_RING buffer and then they are analyzed using the nDPI library.
- **Docker**: Docker is a platform using container virtualization technology to run applications. In order to investigate the pros and cons of the container technology, the vTC is developed also as an independent container application. The forwarding and the inspecting of the traffic are also using PF_RING and nDPI as technologies, but they are modified to fit and function in a container environment.
- **DPDK**: DPDK comprises of a set of libraries that support efficient implementations of network functions through access to the system’s network interface card (NIC). DPDK offers to network function developers a set of tools to build high speed data plane applications. DPDK operates in polling mode for packet processing, instead of the default interrupt mode. The polling mode operation adopts the busy-wait technique, continuously checking for state changes in the network interface and libraries for packet manipulation across different cores. A novel DPDK-enabled vTC has been implemented in this test case in order to optimize the packet-handling and processing for the inspected and forwarded traffic, by bypassing the kernel space. The analyzing and forwarding functions are performed entirely on user-space which enhances the vTC performance.

The various technologies used generate a great variety of test case scenarios and exhibit a rich VNF test case. The PF_RING and Docker cases have the capability of keeping the NIC driver, and so the VNFC maintains connectivity with the OpenStack network connected. On the contrary, in the case of DPDK the NIC is unloaded of the Linux-kernel driver and loaded the DPDK one. However, the DPDK driver causes the VNFC to lose network connectivity with the network attached, the compensation is the significantly higher performance as shown in the next section.

2.5.6. Dimensioning and Performance

Results include comparison of the traffic inspection and forwarding performance of the vTC using PF_RING, Docker and DPDK.

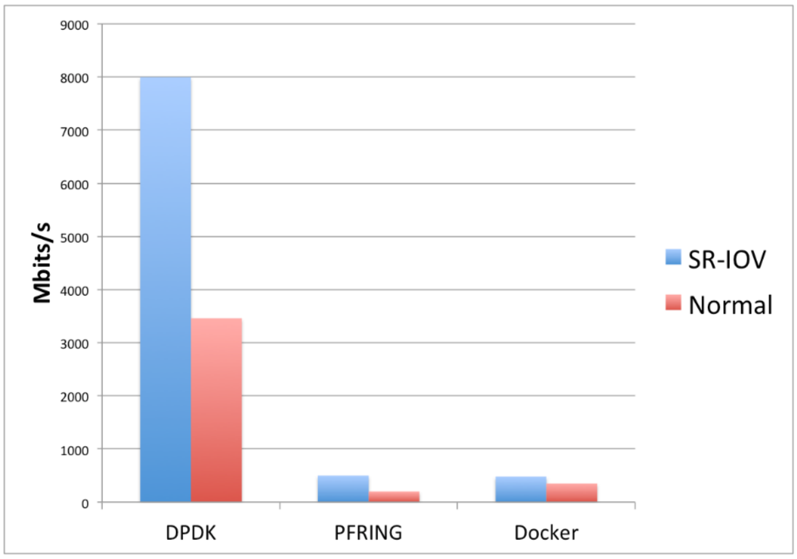


Figure 22. vTC Performance comparison between DPDK, PF_RING and Docker

As it can be seen from the evaluation results among the various approaches used for the vTC, the DPDK approach performs significantly better from the other 2 options. Especially in the case it is combined with SR-IOV connectivity it can achieve nearly 8Gbps/s of throughput. However, the DPDK version as already mentioned has an impact on connectivity with the OpenStack network, as the kernel stack is removed from the NIC. Although the PF_RING and Docker versions maintain connectivity with the network, their performance is clearly degraded compared to DPDK's.

The Dimensioning of the vTC due to its architecture is based on the infrastructure aspect. The vTC performance is dependent on whether there is SR-IOV available on the running infrastructure.

2.5.7. Deployment Details

The vTC was developed in order to be deployed and run in an OpenStack environment, the OS of the virtualized environment was Ubuntu 14.04 LTS. The selection of the OS version assures the maintenance and continuous development of the VNF. In order to conform to the T-NOVA framework a Rundeck job-oriented service functionality was implemented.

The vTC lifecycle is performed via the Rundeck framework in order to facilitate the seamless functionality of the VNF. In Rundeck, we have created different Jobs to describe the different lifecycle events. Each event has a description and is part of a Workflow.

An example workflow:

```
If a step fails: Stop at the failed step.
```

```
Strategy: Step-oriented
We add a step of type "Command". The command differs according to the
operation we want to implement. The operations we implemented are
described below:
* 1. VM Configuration - Command: "~/rundeck_jobs/build.sh"
* 2. Start Service - Command: "~/rundeck_jobs/start.sh"
* 3. Stop Service - Command: "~/rundeck_jobs/stop.sh"
```

In terms of the data traffic required to test the vTC, several changes and modifications had to be made in order to fit the desired traffic mirroring scenario it was tested. Detailed information about this subject is further discussed in the section below.

2.5.7.1. Traffic Mirroring – Normal Networking

In order to support direct traffic forwarding, meaning the virtual network interface of one Virtual Network Function Component (VNFC) be directly connected to another VNFC's virtual network interface, a modification on Neutron's OVS needs to be applied. Each virtual network interface of a VNFC is reflected upon one TAP-virtual network kernel device, a virtual port on Neutron's OVS, and a virtual bridge connecting them. This way, packets travel from the VNFC to Neutron's OVS through the Linux kernel. The virtual kernel bridges of the two VNFCs need to be shut down and removed, and then an OVSDB rule needs to be applied at the Neutron OVS, applying an all-forwarding policy between the OVS ports of the corresponding VNFCs. The OpenStack network detailed topology is shown in Fig. 15.

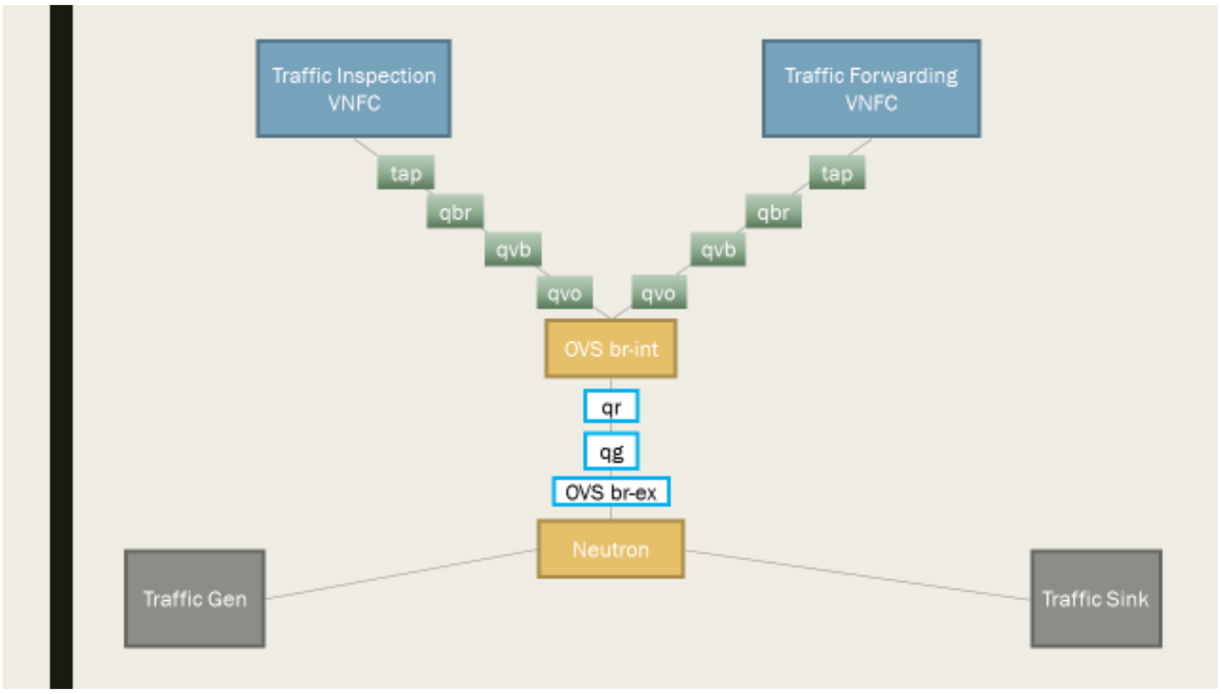


Figure 23. Example overview of the vTC OpenStack network topology.

First Option unbind interfaces from the Openstack networking and connect them directly via OVS

```
 * Remove from br-ex, the qvo virtual interfaces
 * Remove from the qbr linux bridge, the qvb and the tap virtual
interfaces
```

```
 * Add the tap-interfaces on the OVS directly and add a flow
forwarding the traffic to them.
```

This option has been tested and as shown in the results section for the cases of normal network setup.

2.5.7.2. Traffic Mirroring – SR-IOV

Single Root I/O virtualization (SR-IOV) in networking is a very useful and strong feature for virtualized network deployments. SRIOV is a specification that allows a PCI device, for example a NIC or a Graphic Card, to share access to its resources among various PCI hardware functions:

Physical Function (PF) (meaning the real physical device), from it a number of one or more Virtual Functions (VF) are generated. Supposedly we have one NIC and we want to share its resources among various Virtual Machines, or in terms of NFV various VNFCs of a VNF. We can split the PF into numerous VFs and distribute each one to a different VM. The routing and forwarding of the packets is done through L2 routing where the packets are forwarded to the matching MAC VF. In order to perform our mirroring and send all traffic both ways we need to change the MAC address both on the VM and on the VF and disable the spoof check.

2.5.8. Future Steps

Future steps include the implementation of automated way to apply the direct connection of the VNFCs. This step will be included in a HEAT deployment

- Benchmarking all options and comparing them.

Other options to be tested, is to add a TCP/IP stack on the DPDK and maintain the connectivity of the VNFCs. These alternatives include:

- A kernel with tcp/ip stack on the userspace DPDK rump kernel – https://github.com/rumpkernel/drv-netif-dpdk
- DPDK FreeBSD TCP/IP Stack porting https://github.com/opendp/dpdk-odp

2.6. Virtual Home Gateway (VIO)

2.6.1. Introduction

Another VNF that T-NOVA aims to produce is currently known in the research and the industry world under various names, notably Virtual Home Gateway (VGH), Virtual Residential Gateway, Virtual Set-Top Box or Virtual Customer Premise Equipment.

We will see how the initial need has been expanded to cover some aspects of the Content Delivery Network virtualization as well.

The following sections aim to provide a brief description of the proposed virtual function along with the requirements, the architecture design, functional description, and technology.

In T-NOVA, we will focus on the bottleneck points usually found in resource constrained physical gateway like media delivery, streaming and caching, media adaptation and contextawareness. In fact, some previous research proposals like [Nafaa2008] or [Chellouche2012] include the Home Gateways to assist the content distribution. By using a Peer-to-Peer approach, the idea is to offload the main networks and provide an “Assisted Content Delivery” by using a mix of Server Delivery and Peer delivery.

When virtualizing the Home Gateway, this approach can lead in some extent to the creation of a Virtual CDN or vCDN as a VNF.

Particular attention will be given to real world deployment issues, like coexistence with legacy hardware and infrastructure, compatibility with existing user premise equipment and security aspects.

2.6.2. Architecture

2.6.2.1. High level

Next, we present how the box and the server will connect to perform network operations. Figure 24 shows the high level architecture as presented by ETSI in [Netty] netty.io

[NFV001].

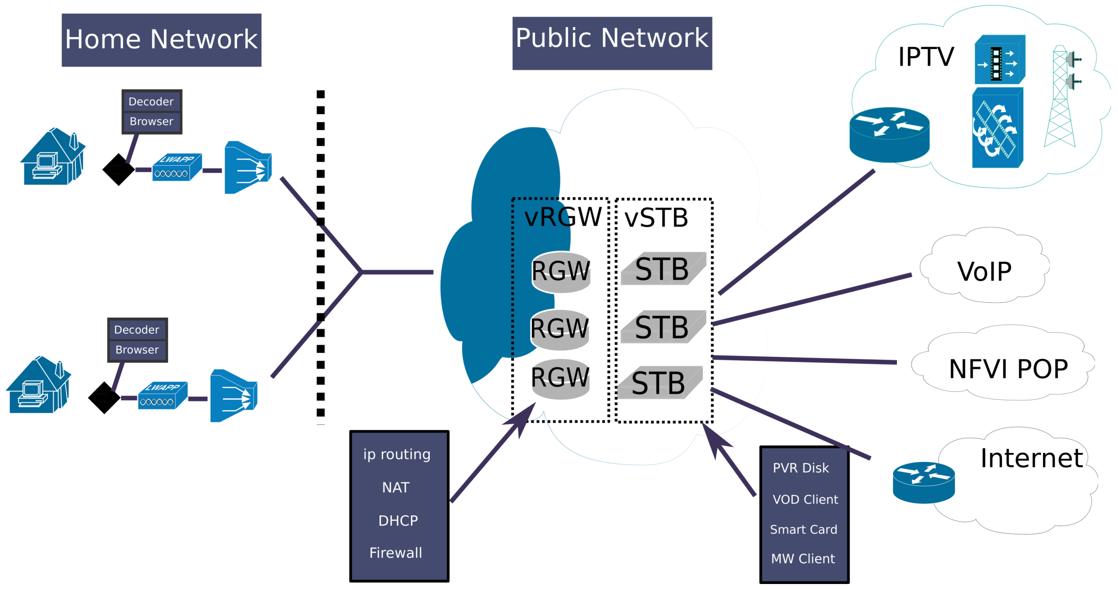


Figure 24 ETSI home virtualization functionality

We aim at supporting a subset of the vHG NF as well as some vCDN content caching and orchestration related NF.

2.6.2.2. Low Level

The vHG+vCDN vNF are composed from:

- **vHG:** 1 VM per user that acts as a transparent proxy that can redirect user requests to caches deployed in NFVI-POP;
- **vCDN/Streamer VNF:** an arbitrary amount of VMs (at least 3) that stores and delivery content to the end user;
- **vCDN/Transcoding vNF:** an arbitrary amount of VMs that ingest content from the content provider and provision a transcoded version to a Streamer vNF;
- **vCDN/Caching and Transcoding orchestrator vNF:** a VM that deploys redirect rules to the vHGs and triggers transcoding jobs.

To illustrate low level aspects of the VNFs that will be deployed, we will take a specific example of VNF which caches, transcodes and streams the most requested videos by gateway users.

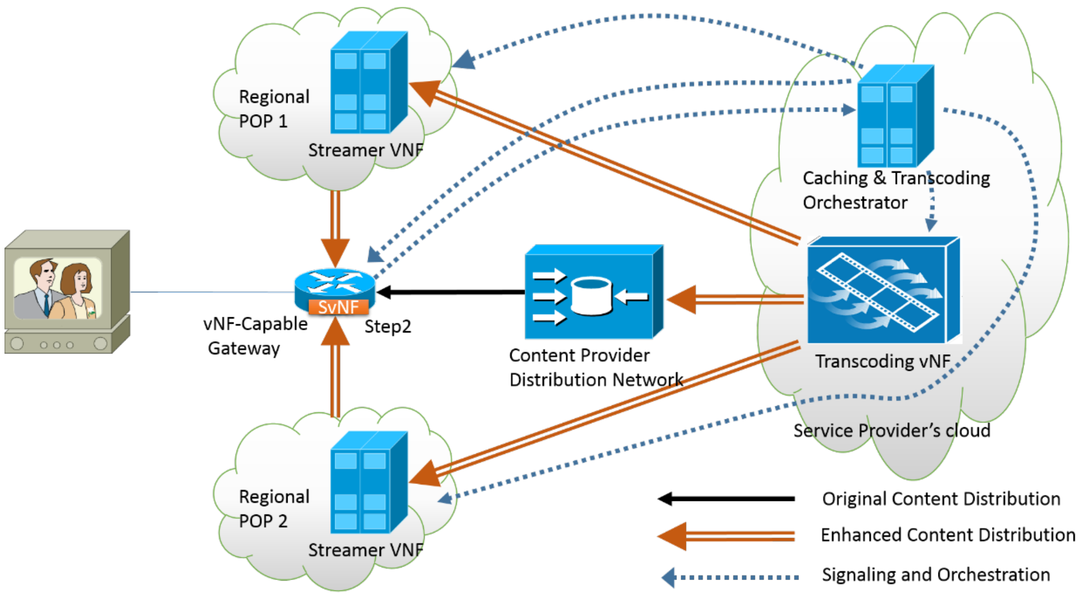

Regional POP 1
Streamer VNF
Caching & Transcoding Orchestrator
SvNF
vNF-Capable Gateway
Step2
Content Provider Distribution Network
Transcoding vNF
Service Provider's cloud
Regional POP 2
Streamer VNF
Original Content Distribution
Enhanced Content Distribution
Signaling and Orchestration

Figure 25 Low level architecture diagram for transcode/stream VNF

Figure 25 illustrates a modular gateway which acts as a HTTP proxy, notifying the content fronted when a video is consumed by the end user. Having this information allows the content frontend to trigger the download from the content provider's network to the VNF. Once the video is entered on the VNF, it's transcoded and moved to the storage shared by the streamers.

Once the video resource is available to the end user, the gateway redirects the user's request to the streamer, ensuring the best QoE possible.

2.6.3. Sequence diagrams

The sequence diagram presented in Figure 26 is associated with the example presented in section 2.6.2.2.

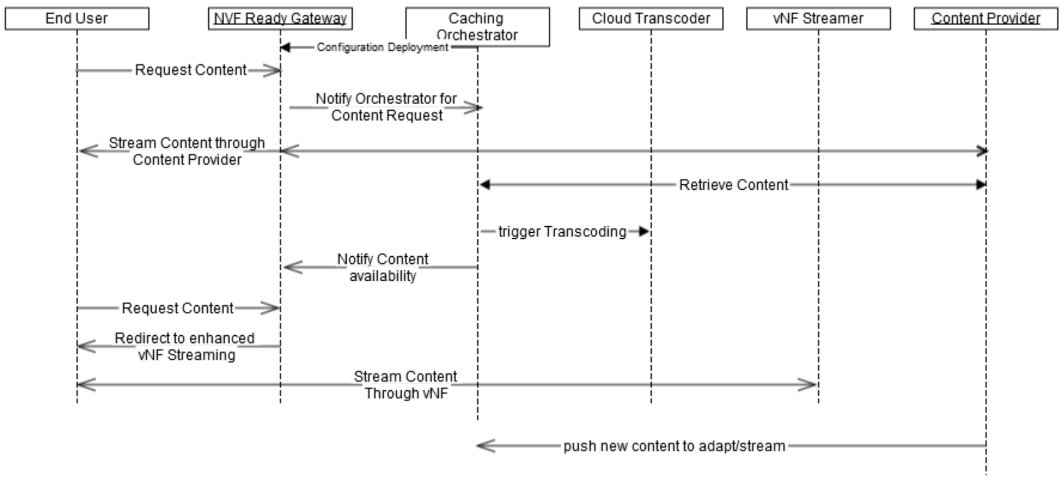

End User
NVF Ready Gateway
Caching Orchestrator
Cloud Transcoder
vNF Streamer
Content Provider
Configuration Deployment
Request Content
Notify Orchestrator for Content Request
Stream Content through Content Provider
Retrieve Content
trigger Transcoding
Notify Content availability
Request Content
Redirect to enhanced vNF Streaming
Stream Content Through vNF
push new content to adapt/stream

Figure 26 Sequence diagram for the transcode/stream VNF example.

2.6.4. Technology

2.6.4.1. Netty: a Java Non-Blocking Network Framework

Netty is *an asynchronous event-driven network application framework* [Netty] for rapid development of maintainable high performance protocol servers and clients.

One of the most striking features of Netty is that it can access resources in a non-blocking approach, meaning that some data is available as soon as it gets in the program. This avoids wasting system resources while waiting for the content to become available; instead a callback is triggered whenever data is available. This also saves system resources by having only 1 thread for resource monitoring.

Netty is one of the building blocks to be used to implement the OSGi bundle Proxy composing the vHG.

2.6.4.2. Restful architecture

End user applications, Gateways and Front-end need to interact though secured connection on the internet.

A Java Restful architecture can be implemented for those reasons:

- Architecture is stateless, which means that the servers that expose their resources do not need to store any session for the client. This greatly eases scaling up, since no real time session replication needs to be performed, therefore a new server will be deployed for load balancing purposes.
- Architecture is standard and well supported by the industry, allowing us to leverage tools for service discovery and reconfiguration.
- Authentication methods are well documented and widespread among web browsers and servers.

Regarding the technical details, we will consider the standards of the Java SDK, by using JAX-RS and its reference implementation, Jersey. This framework can be integrated on any servlet container, JEE container or lightweight NIO HTTP server like Grizzly which is used on the vHG.

2.6.4.3. Transcoding workers

One of the key features of cloud computing is its ability to produce on-demand compute power at a small cost. To take advantage of this feature, we plan to implement the most computing intensive tasks as a network of workers using a Python framework called Celery. Celery is an asynchronous task queue/job queue based on distributed message passing.

Every Celery worker is a stand-alone application being able to perform one or more tasks in a parallelized manner. To achieve this goal, a general transcoding workflow has been designed to be applied on a remote video file.

Having a network of workers allows us to scale-up or scale-down the overall compute power simply by turning a virtual machine up or down. Once the worker is up, it connects to the message broker, and picks up the first task available on the queue. Frequent feedback messages are pushed to the message broker, allowing us to present the results on the gateway as soon as they are available on the storage.

If the compute capacity is above the required level, active workers are decommissioned, leaving the pool as their host virtual machine turns off.

Note that workers only carry out software transcoding, leaving room for optimization through the use of hardware. The virtual Trancoding Unit (vTU) is an excellent drop-in replacement for the transcoding vNF. However, as hardware transcoding may not be available everywhere, we keep the slow software transcoding as a fall-back option.

2.6.4.4. Scalable Storage

We need to have caches able to store the massive amount of data needed by a CDN. These caches can be spread among several datacentres and must be tolerant to failure. They also need to scale, and must support adding or removing storage node as defined by the scaling policy.

To implement that, we decided to deploy [Swiftstack] which proposes to create a cluster of storage node to support Scalable Object storage with High availability, Partition Tolerance and eventual consistency.

Storage Nodes are accessed by external users using a Swift Proxy that handles the read and write operations. Swift has abstractions where nodes are stored inside zones and regions in Figure 25. We detail the mapping between swift abstraction and T-NOVA in Table 2-B

Swift	T-NOVA	Meaning
Region	NFVI-POP	Parts of the cluster that are physically separated
Zone	Compute Node	Zones to be configured to isolate failure
Node	VNFC	Server that run one or more swift process

Table 2-B Mapping between Swift and T-NOVA abstraction

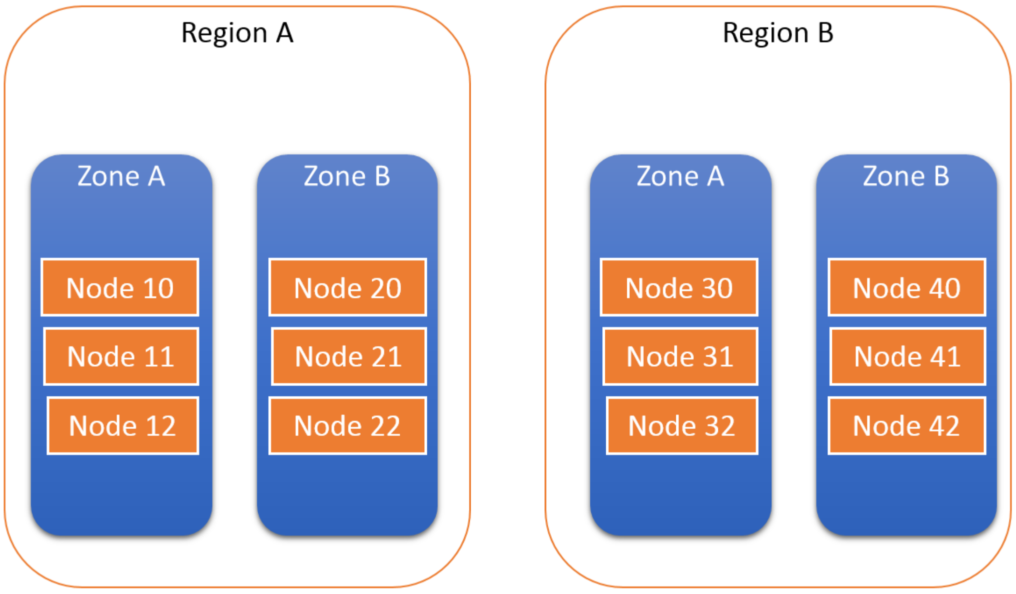


Figure 27 swift stack Region/Zone/Node.

We use swift abstraction to provide a reliable storage solution. For example, our vCDN spans over multiple datacentres to provide good connectivity. Each pop is associated to a region. With the same approach, we can have several compute nodes hosting our VNFC. For reliability reasons, we don't want all our nodes hosted on the same compute node, so that if the compute node goes down, part of the service will be still available. Finally, each VNFC hosts a swift Node.

Even if swift is an object storage, it allows users to access and push data over a standard HTTP API. It means that the streamer vNF feature can be implemented using swift as well.

2.6.4.5. Using Docker to provide safe, reliable and powerful application deployment

We decided to use [Docker] to support the implementation. Docker is an OS Virtualization technology that runs segregated applications and libraries on a common Linux kernel.

Docker can be run on major Linux Distribution like Debian or Fedora, but it can also run on smaller, custom distribution that provide an execution environment for container. CoreOS produces, maintains and utilizes open source software for Linux containers and distributed systems. Projects are designed to be composable and complementing each other in order to run container-ready infrastructure.[2]

The applications we build are based on vendor technologies (for example, the Java Docker image maintained by Oracle) that are kept updated on a regular basis. We implemented continuous deployment, meaning that whenever an upstream dependency gets updated, we re-package our software with the new image and run test to discover potential regression.

Our approach is safer. The traditional installation of a package on an OS since every container is walled from the other ones and the OS has the only responsibility of maintaining the container execution environment. Vendors usually provide a shorter delay to update their docker images that the Linux Distribution.

[2] https://coreos.com/docs/

Our approach is reliable in the sense that if a T-NOVA virtual machine goes down (except the VNF Controller which is not highly available for the moment) we are able to redeploy containers on the cluster on another available machine.

We also don't have to upload a new vnfd + vnf images every time we have a security update. All we need to do is to push the new release on our docker registry and the new image will be picked up automatically when configuring the VMs.

2.6.4.6. Orchestration and scaling

In order to ease the deployment of our vNFs, we use a configuration management tool named Salt Stack [Salt]. The necessity to use such a tool is developed in the next paragraphs; we then explain why we choose salt and finally conclude with an overview of the mechanisms we implemented.

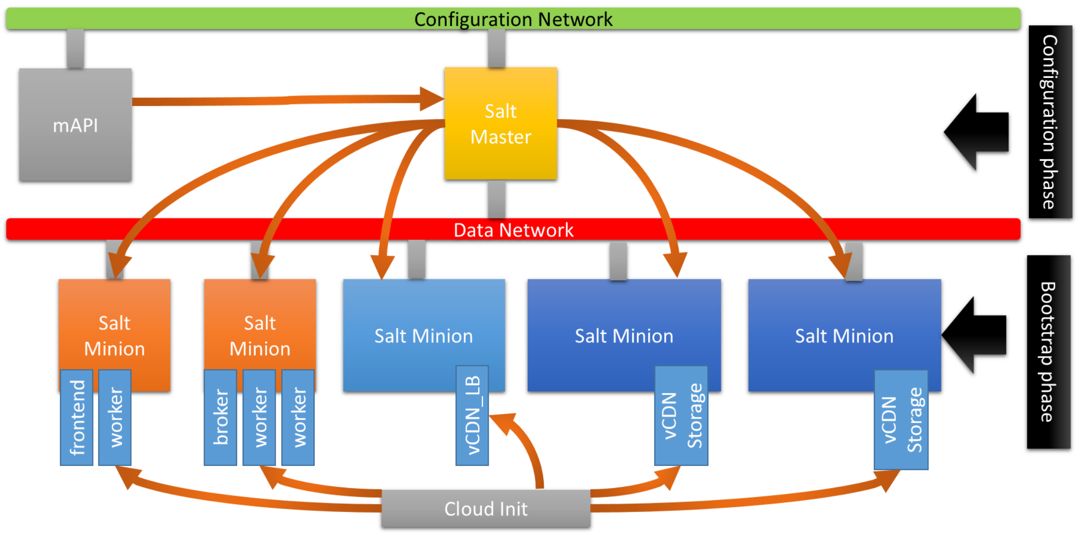


Figure 28 vCDN implementation for Software configuration Management.

(a) Why configuration management tool?

As mentioned in 2.1.1, only one VNFC is able to receive configuration commands from the Orchestration to support the whole VNF life cycle. This means that the information received on the configuration interface must be propagated to the other VNFCs.

When the VNF starts, some configuration need to be carried out to initialize the software components. For example, the storage nodes must be initialized with the DHT from the proxy, some block storage must be allocated to the node and so one. This non trivial configuration tasks must be carried out after the VM has booted, but also when scaling out or in. These tasks may fail, but the consistency of the whole system should be kept intact.

For those reasons, we decided to use an orchestration tool that create an abstraction level over the system to manage the software deployment, system configuration, middleware installation and service configuration with ease.

(b) Why Salt?

SaltStack platform or Salt is a Python-based open source configuration management software and remote execution engine. Supporting the "infrastructure-as-code" approach to deployment and cloud management, it competes primarily with Puppet, Chef, and Ansible.[3]

Salt Stack was preferred over other alternatives due to its scalability, ease of deployment, good support for Docker and python source code. We don't claim that what we designed would not have been possible with other alternative, but Salt was the solution we felt the more comfortable with at the end.

(c) Implementation of our configuration management

We implemented the configuration management in a two-phase. It is illustrated in Figure 28.

First during the bootstrap phase, each virtual machine is injected with cloud-init with the following data and programs.

- IPaddress of the salt master
- Certificates to assure a secure connection with the salt master
- Its role in the system.
- Salt-master or salt-minion service installed and launched.
- The "recipes" or desired infrastructure code deployed on the salt master.

Once the bootstrapping phase is over, we have a system comprised of VMs securely connected on the data network ready to take order from the master. Note that the OS could be pre-bundled with software in order to fasten the next phase, but this is not mandatory.

The second phase is launched when the start lifecycle event from TeNOR is received through the middleware API. This processes the infrastructure code and verifies the compliance of each minion with the desired infrastructure.

As we can see in Code listing 1, the yaml DSL used with Salt describes how the infrastructure should be configured. Salt allows us to "synchronise" the code infrastructure in yaml with the real infrastructure simply by calling the Salt API. This synchronization process installs, copies, configures, downloads the required missing software components and can even configure more low level aspects.

Providing the possibility for the system the scale-in is straightforward when having the infrastructure described as code. Installing, configuring and ramping up new VM is just a matter of "synching" the infrastructure state with the new resources available.

Our implementation use the Ubuntu OS for undockerized software (the storage) and CoreOS distribution to host application that are already dockerized (the rest of the software).

[3] https://en.wikipedia.org/wiki/Salt_%28software%29

```
#here we make sure that the latest worker docker image is present on the system
nherbaut/worker:
  #this command is equivalent to docker pull
  docker.pulled:
           #always use the latest version from our continus build system
    - tag: latest
    - require:
            #make sure that docker is installed before pulling the image
      - sls: docker
            #make sure that docker daemon is running
      - service.running: docker

# this set of jinja2 template file is here to provide the broker's IP address
{%- set minealias = salt['pillar.get']('hostsfile:alias', 'network.ip_addrs')
%}
{%- set addrs = salt['mine.get']('roles:broker', minealias,"grain") %}
{%- set broker_ip= addrs.items()[0][1][0] %}

# this set of instruction is there to provide the the swift proxy ip address
{%- set addrs = salt['mine.get']('roles:swift_proxy', minealias,"grain") %}
{%- set swift_proxy_ip= addrs.items()[0][1][0] %}


# now we are ready to cook our docker image
core-worker-container:
  docker.installed:
    - name: core-worker-container
    - image: nherbaut/worker:latest
     # now we are ready to cook our docker image
    - environment:
      - "CELERY_BROKER_URL" : "amqp://guest@{{ broker_ip }}"
      - "ST_AUTH" : "http://{{ swift_proxy_ip }}:8080/auth/v1.0"
      - "ST_USER" : "admin:admin"
      - "ST_KEY" : "admin"
    - watch:
      # trigger this event whenever the image is done being pulled
      - docker: nherbaut/worker
```

Code listing 1 an example of infrastructure code

2.6.5. Dimensioning and Performances

The results reported here are not derived from the results from the Pilot testbed, which is not available to VNF developer for performances purposes so far. We plan to perform the same tests on the real pilot and update our results in the next deliverable. We thought it would be valuable to have a first vision on some results through our related work [Herbaut2015].

We described the role of the vHG+vCDN deployed on the server-side infrastructure composed of various vNFs: Streamers vNFs deployed in regional PoPs, Caching and Transcoding Orchestrator and Transcoding vNFs deployed in regular data-centers.

As our proposal aims at showing how vHG can play a role in a vNF architecture, our first focus for the evaluation is devoted to assessing vHG side performance, CPU and memory footprint. Next, to evaluate the benefits of the overall vHG+vCDN system in above-cited use case, we made extensive simulations with hypothesis conforming to a French network of Service Provider.

2.6.5.1. vHG

	Web Traffic			File Transfer		
	CPU baseline = 1	free Memory baseline = 693 MB	Throughput baseline=11.5 MBps	CPU	Free Memory baseline = 708 MB	Throughput baseline = 11.5 MBps
Baseline	1x	100%	100	1x	100%	100
Squid	14x	99%	≃100	17x	99.3%	≃100
SvNF	**18x**	98%	99.5	**20x**	99.8%	≃100
SvNF with rules	**19x**	99%	99.7	**21x**	99.4%	≃100

Table 2-C vHG performances

We deployed the vHG as an OSGi bundle in the Apache Karaf OSGi runtime on a VM running Debian Jessie. It uses 2Gb or Ram and 1 vCore. The gateway connects the test operator and a PC-based file server.

JMeter was used to capture the network metrics of our solution. While generating HTTP requests, it reports on specific performances metrics. Each experiment consisted of 10 agents continuously downloading target resources on the HTTP file server, 1000 times.

We considered two different validation use cases: Web Traffic and File Transfer. First, the agents had to download a 192 MB video file, then a single HTML page which linked to 171 static resources composed of Javascript files, CSS and images of average size 16 KB.

We evaluated our solution with two different routing rules settings deployed in the vHG. In the first one, we did not deploy any rule, assessing only the overhead linked to the application network framework. In the second one, however, we deployed 10.000 rules, causing the vHG to process both requests and responses wrt those rules thereby assessing its ability to perform pattern matching in a timely manner. Both cases reflect the no operation scenario of our use case.

We decided to assess the overhead caused the vHG by comparing it to the well-known Squid 3 HTTP proxy with cache deactivated. To have a better grasp of the amount of resources consumed by the vHG, we also reported CPU and memory consumption for each settings as well.

From Table 2-C, we can see that the performances in term of throughput is globally the same across all settings, with the maximum deviation from the baseline setting 1 (simple IP forwarding) being less than 0.3%. In settings 2-4, we also see that a significant share of CPU power is dedicated to processing the requests for both Squid 3 and the vHG, with the vHG consuming up to a 25% extra CPU time in Settings 4 wrt Settings 1, but with no drop in throughput. This can be explained by the fact that the vHG runs on top of a JVM, while Squid is a native application with limited overhead wrt a Java application.

This experiment does not intend to mimic real life Internet usages but to stress the system up its limits. We conclude that even with the extra CPU involved, our solution does not significantly penalize the End-User, validating the possible deployment of vHG.

2.6.5.2. vCDN

We simulated the network deployment presented in Figure 28, with the hypotheses presented in Table 2-D

Settings	**Scenario A**	**Scenario B**
CP Bandwidth	2 Gbps	1 Gbps
Regional PoP Bandwidth	0 Gbps	1 Gbps
Regional PoP latency	25ms	
CP latency	50ms	
Video Distribution	Normal	
Video Size	Pareto with average video size=10Mb	
Requested bitrate	320kbps	
# of gateways	200	
# of video (cruising/peak)	200/100	
Mean time between request (cruising/peak)	0.1s/0.05s (Poisson))	
SLA Violation criteria	less than 75% of the target bitrate 10s after the request	
Video Caching	after 4 requests	

Table 2-D HYPOTHESIS USED FOR SIMULATION

The objective is first to assess the feasibility of having a vHG+vNF approach and second to highlight its benefits. The use case considers the deployment of vHG and vCDN in the SP's regional PoPs, for video delivery. Let us thus investigate the potential of such a solution.

Streamed video needs good bitrate to avoid re-buffering and improve QoE. That is the reason why we defined an SLA violation as the failure to deliver the proper average bitrate on time to a client. Our goal in this simulation is to reduce the SLA violations over time.

The regional PoP being located near the End-User, its latency is reduced wrt the CP network (backed by CDN), hence a possible higher throughput for HTTP traffic like video streaming. For our simulation, we included two types of patterns. The first one is composed of video requests emitted regularly by the clients, which generate the cruising phase traffic. The second consists of peak traffic at 25s which is characterized by a greater request arrival rate as well as a more concentrated distribution of videos. Consumption peaks usually occur when a viral video is posted, most of the time on the landing page of the Content Provider. Being able to cache this kind of video and to serve it as close as possible to the users is a key indicator of success for the vNF.

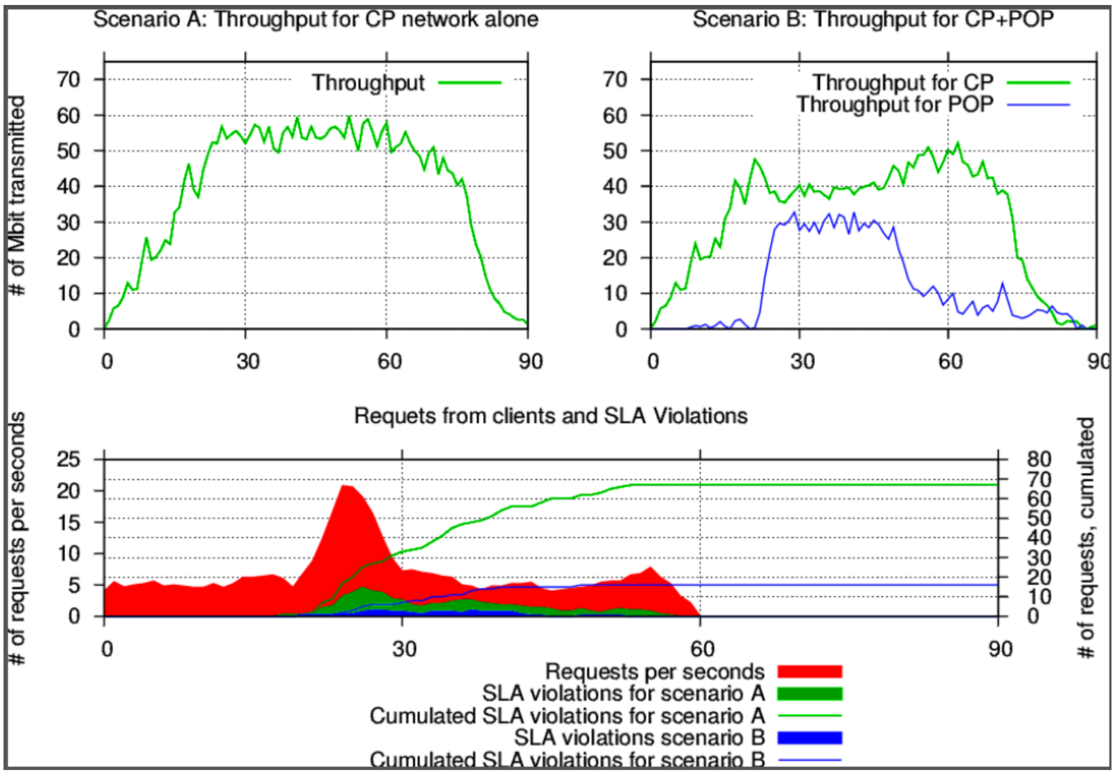


Figure 29 Evaluation of the benefits of the vHG+vCDN system.

Figure 29 depicts two scenarios. In (A), we only rely on CP network to deliver the media while in (B) a single regional PoP is added to the solution. Note that global bandwidth remains the same, as we took some bandwidth from the CP to allocate it to the regional PoP. This is also essential for the CP for comparing at the end the solutions in terms of bandwidth cost.

We can see that the cruising phase does not generate any SLA violation and the CP alone is able to handle the traffic load. However, when the peak occurs, SLA violations increase dramatically, causing a lot of requests to be dropped. In scenario B, however, the presence of the regional PoP as an alternative, low latency data source, mitigate the peak effect and reduces up to 70% of SLA violations on the overall simulation period.

Having a regional PoP with lower network latency to serve highly redundant requests, benefits both the End-User and the content provider. As the former sees an increase in QoE, the latter reduces its costs by avoiding the over provisioning of network capabilities. It's important however, to reserve regional PoP bandwidth to serve only highly popular videos, so as to maximize its benefits, while keeping the mean latency low between clients and regional PoPs by spreading PoPs over the territory.

2.6.6. Future Work

The next step for vHG+vCDN in WP5 is to deploy on a fully functional testbed, assess the performance of the overall solution and implement scaling based on metric analysis.

We may also consider a use case where hardware acceleration is used through vTU in order to accelerate the availability of cached content on the vStreamer.

2.7. ProXy as a Service VNF (PXaaS)

2.7.1. Introduction

A Proxy server is a middleware between clients and servers. It handles requests, such as connecting to a website or service and fetching a file, sent from a client to a server. In the most cases a proxy acts as a web proxy allowing or restricting access to content on the World Wide Web. In addition, it allows clients to surf the Web anonymously by changing their IP address to the Proxy's IP address.

A proxy server can protect a network by filtering traffic. For instance, a company's policies require that its employees are restricted to access some specific web sites, such as Facebook, during working hours but they are allowed to access them during break times or are restricted to access adult-content sites at all times. Furthermore, a proxy server can improve response times by caching frequently used web content and introduce bandwidth limitations to a group of users or individuals. Traditionally, proxy software resides inside users' LANs (behind NAT or Gateway). It is deployed on a physical machine and all local devices can connect to the Internet through the proxy by changing their browser's settings accordingly. However, a device can bypass the proxy. A stronger alternative deployment is to configure the proxy to act as a transparent proxy server so that all web requests are forced to go through the proxy. In this scenario the gateway/router should be configured to forward all web requests to the proxy server.

The Proxy as a Service VNF (PXaaS VNF) aims to provide proxy services on demand to a Service Provider's subscribers (either home users e.g. ADSL subscribers or corporate users such as company subscribers). The idea behind the PXaaS VNF is to move the proxy from the LAN to the cloud in order to be used "as a service". Therefore, a subscriber (e.g. LAN administrator) will be able to configure the proxy from a web-based user friendly dashboard and according to their needs so that it can be applied to the devices within the LAN.

2.7.2. Requirements

The table below provides the major requirements that the VNF will need to fulfill.

Table 2-E: PxaaS VNF requirements

Requirement ID	Requirement name	Description	Priority level
1	Web caching	The PXaaS VNF should be able to cache web content.	High
2	User anonymity	The PXaaS VNF should allow for hiding the user's IP address when accessing web pages. The proxy VNF's IP should be shown instead of the user's real IP.	High
3	Bandwidth rate limitation per user	The PXaaS VNF should allow for setting bandwidth rate limitations on a group of users or individual users by creating ACLs based on their account.	High
4	Bandwidth rate limitation per	The PXaaS VNF should allow for setting bandwidth rate limitations on a group of	Low

	service	services or individual services. For example, the PXaaS VNF should limit the bandwidth used for torrents.	
5	Bandwidth throttling on huge downloads	The PXaaS VNF should allow for reducing the bandwidth rate when huge downloads are detected. It could be applied to all users or a group of users or individuals.	High
6	Web access control	The PXaaS VNF should allow for blocking specific websites by the users.	High
7	Web access control (time)	The PXaaS VNF should allow for blocking or accessing specific websites by the users based on the current time.	Medium
8	User Control and Management	The user should be able to configure the PXaaS VNF using a dashboard. The dashboard should be responsive in order to be accessible from multiple devices and easy to use.	High
9	Service availability	The Proxy VNF should be available as soon as the user sets the configuration parameters on the dashboard. Each time a user changes configuration, the service should be available immediately.	High
10	Service accessibly	The connection with the proxy should be transparent (transparent proxy). Users do not need to set the proxy's IP on their browser. The traffic should be redirected from the user's LAN to the proxy VNF.	Low
11	Service – user authentication	Only subscribed PXaaS VNF users should be able to access the service.	High
12	Monitoring	The proxy VNF should provide metrics to the T-NOVA's monitoring agent.	High
13	Service provisioning	The proxy VNF should expose an API to be used by the T-NOVA's middleware for service provisioning.	High

2.7.3. Architecture

The PXaaS VNF consists of one VNFC. The VNFC implements both the proxy server software as well as the web server software. The figure below provides a high level topology of the PXaaS VNF. The VNFC is located at the PoP which is found between the user's LAN and the Operator's backbone. Once a user is subscribed with the PXaaS VNF the traffic from the user's LAN is redirected to the PoP and then it passes through the PXaaS VNF. The traffic might pass through some other VNFs according to service function chaining policies. Finally, the proxy handles the requests accordingly and forwards the traffic to the Internet. The user is able to configure the proxy through an easy to use web-based dashboard which is served by the web server. The web server communicates with the proxy server in order to set up the configuration parameters which have been defined by the user.

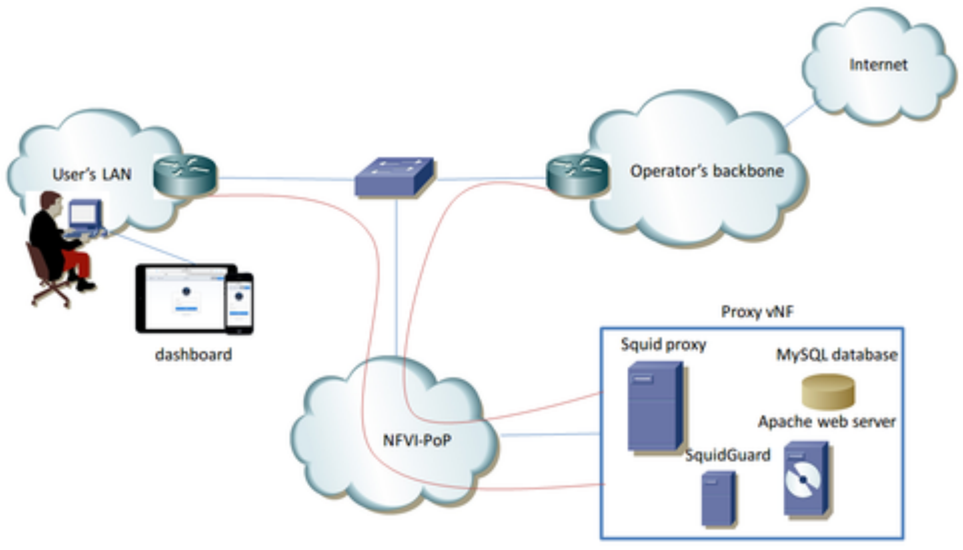


Figure 30. PXaaS high level architecture

2.7.4. Functional description

2.7.4.1. Squid Proxy server

Squid Proxy is a caching and a web proxy. Some of its major features include:

- Web caching;
- Anonymous Internet access;
- Bandwidth control. It introduces bandwidth rate limitations or throttling to a group of users or individuals. For example it allows "normal users" to share some amount of traffic and on the other hand it allows "admin users" to use a dedicated amount of traffic;
- Web access restrictions e.g. allow a company's employees to access Facebook during lunch time only and deny access to some specific web sites.

Bandwidth limitation examples

a) Bandwidth restrictions based on IP

The example below creates an Access Control List (ACL) with the name "regular_users" and is assigned a range of IP addresses. Requests coming from those IPs are restricted to 500KBps bandwidth.

```
acl regular_users src 192.168.1.10 - 192.168.1.20/32 # acl list based
on IPs
delay_pools 1
delay _class 1 1
delay_parameters 1 500000/500000 # 500KBps
delay_access 1 allow regular_users
```

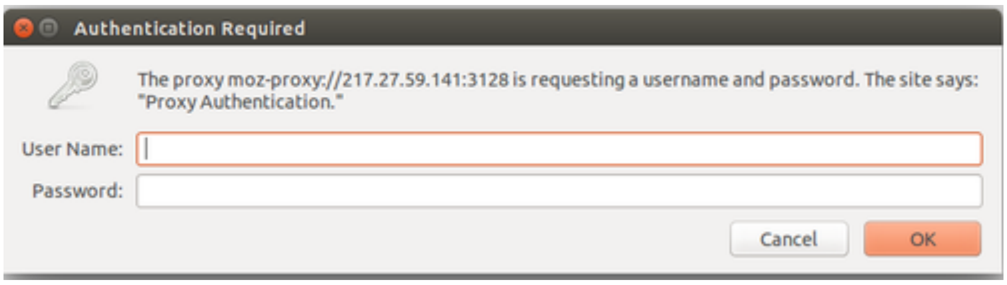
The limitation of this configuration is that Squid should be located inside the LAN in order to understand the private IP address space.

b) Bandwidth restrictions based on user

The following scenario performs the same bandwidth restrictions as the previous one except that the ACL is based on user accounts. Squid supports various authentication mechanisms such as LDAP, Radius and MySQL database. We consider MySQL database for authenticating with the PXaaS VNF.

```
acl regular_users proxy_auth george savvas # acl list based on
usernames
delay_pools 1
delay _class 1 1
delay_parameters 1 500000/500000 # 500KBps
delay_access 1 allow regular_users
```

The limitation of this configuration is that users must authenticate with the Proxy the first time they visit their browser. In this case the proxy is not considered as a transparent proxy. However, by using this scenario, Squid can be deployed on the cloud and can handle devices behind NAT as long as they authenticate with the proxy.



The proxy moz-proxy://217.27.59.141:3128 is requesting a username and password. The site says: "Proxy Authentication."

User Name:

Password:

Figure 31. Proxy authentication

2.7.4.2. Apache Web Server

Apache web server is used to serve the dashboard to the clients. The dashboard is responsible to allow users to configure and manage the Squid proxy. Therefore, Apache should have write permissions on Squid's configuration file. In addition, the LAN administrator is able to create user accounts which are stored in the MySQL database. The LAN administrator will be responsible to assign the user accounts to each device in order to achieve the limitations he envisions using the PXaaS.

The figure below presents the first version of the dashboard (version 1). In particular, the home page of the dashboard is presented. The current version supports the following features:

- **User management:** User accounts can be created with a username and password. Those accounts are used to access the proxy services;
- **Access control:** Users must enter their credentials in their browsers in order to surf the web;
- **Bandwidth limitations:** Group of users can be created with a shared amount of bandwidth. In this case bandwidth limitations can be introduced to a group of users;
- **Website filtering:** Group of users can be created with restricted access to a list of websites. Pre-defined lists with urls are provided;
- **Web caching:** Web caching can be enabled in order to cache web content and improve response time;
- **User Anonymity:** Users can surf the web anonymously.

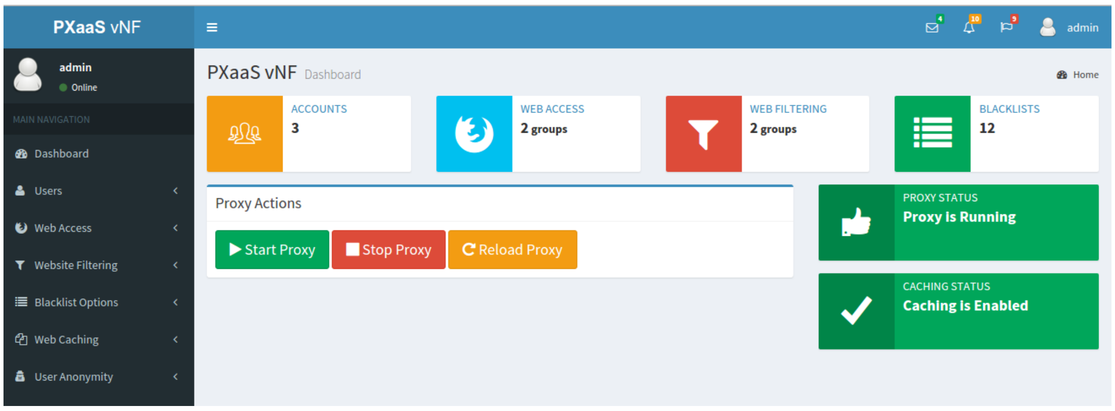


Figure 32. PXaaS Dashboard.

2.7.4.3. MySQL Database server

MySQL Database server maintains a list of user accounts that can be used for proxy authentication in the browser. In addition it stores all the required data needed by the dashboard.

2.7.4.4. SquidGuard

SquidGuard is used on top of Squid in order to block URLs for a group of users. It is used based on pre-defined black lists.

2.7.4.5. Monitoring Agent

The Monitoring Agent is responsible for collecting and sending monitoring metrics to the T-NOVA Monitoring component.

2.7.5. Interfaces

The figure below shows the VNFC in an OpenStack environment. It consists of 3 interfaces connected to 3 networks.

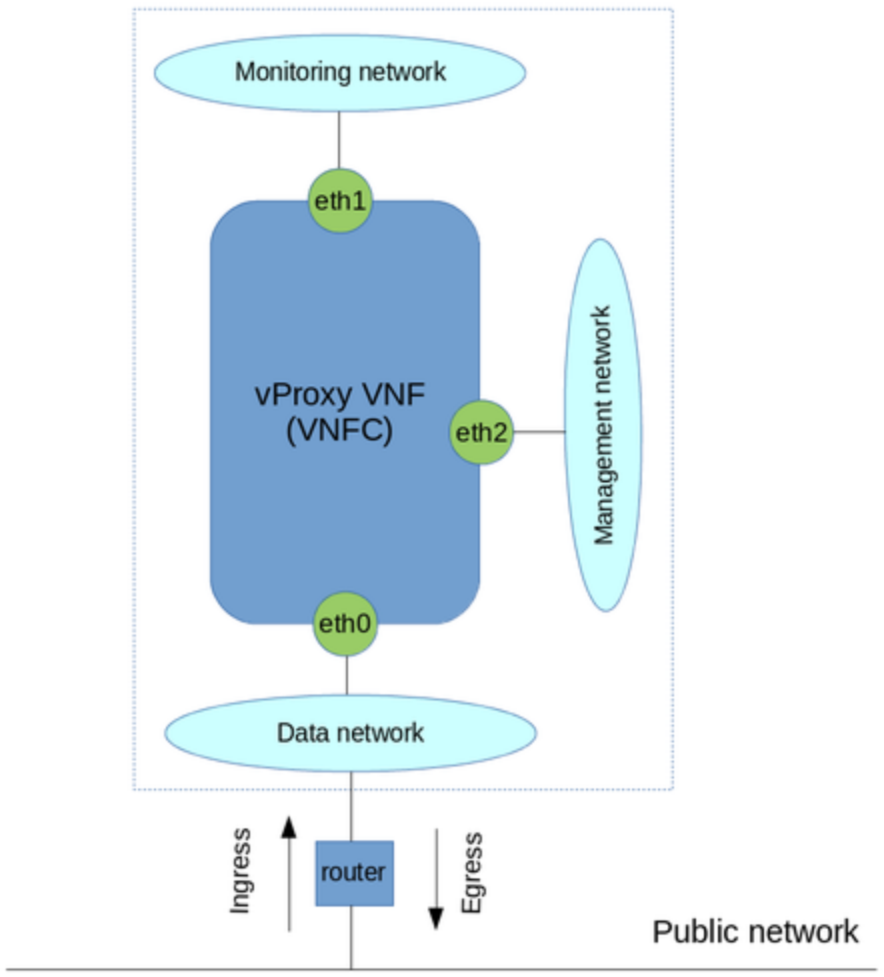


Figure 33. PXaaS in OpenStack

- **eth0**: This is the data interface. A floating IP is associated with this interface in order to send and receive data to/from the Public network.
- **eth1:** This is the monitoring interface which will be used to send metrics periodically to the Monitoring component.
- **eth2:** This is the management interface which will be used in order to communicate with the middleware API.

2.7.6. Technologies

The development environment used for the implementation and testing of the PxaaS is Vagrant with Virtualbox on an Ubuntu 14.04 Desktop machine. The VM itself runs Ubuntu 14.04 server OS.

As described in the Functional description section, Squid Proxy, SquidGuard, Apache Web server and MySql Database server are used. Specifically, the exact versions are:

- Squid Proxy 3.5.5
- SquidGuard 1.5
- Apache2 2.4.7
- Mysql 5.5.44-0ubuntu0.14.04.1

The Dashboard has been developed with the Yii framework (a PHP framework) for the server side and CSS, HTML, Jquery have been used for the client side.

As regards the monitoring agent two different components have been used:

1. Collectd. It collects system performance statistics periodically such as CPU and memory utilization.
2. A python script which collects PxaaS VNF specific metrics such as the number of HTTP requests received by the proxy and the cache hits percentage. The script analyses the results received by the squidclient, a tool which provides Squid's statistics, and send them to the T-NOVA Monitoring component periodically.

Mozilla Firefox is used for accessing Web through the proxy.

2.7.7. Dimensioning and Performance

Some preliminary tests were performed in order to verify whether the expected behavior is achieved. We assume that access to the PXaaS Dashboard is given to a user who acts as the administrator of his LAN in a home scenario. Therefore, the "administrator" sets up the Proxy service for his LAN via the dashboard and creates user accounts in order to allow other users/devices to access the Web via the Proxy. Specifically, the current version of the Dashboard was tested against the following test scenarios:

a) **Testing web access and bandwidth control**. This scenario aims to test if a newly created user is able to access the Web using their credentials and bandwidth limitation is achieved.

Execution: The administrator creates a new user by providing a username and password. Then he adds the newly created user under “research” group (the group was previously created by the administrator) which is restricted to 512Kbps bandwidth. The new user authenticates with the proxy from the browser and downloads a big file.

b) **Testing web site filtering**. The scenario tests whether a user is restricted to access some websites.

Execution: The administrator adds the user to the group “social_networks” (the group was previously created by the administrator and a pre-defined list of social networking websites was assigned to that group) in which all social networking websites are denied.

c) **Testing web caching**. This scenario tests whether web caching works properly.

Execution: Two different users access the same websites from different computers. For example “user1” accesses www.primetel.com.cy and then “user2” accesses the same website.

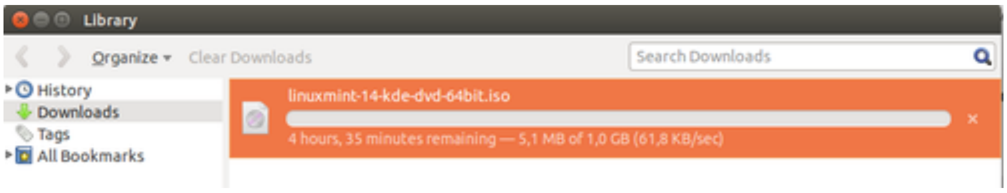
d) **Testing user anonymity**. This scenario checks whether a user is able to access the Web anonymously. In order to test this scenario and get meaningful results we deployed the PXaaS VNF on a server with public IP.

Execution: The administrator enables the user anonymity feature for a user. http://ip.my-proxy.com/ website is used in order to check whether user's real IP is publicity visible.

2.7.7.1. Test results

Below the results by executing the test scenarios are presented.

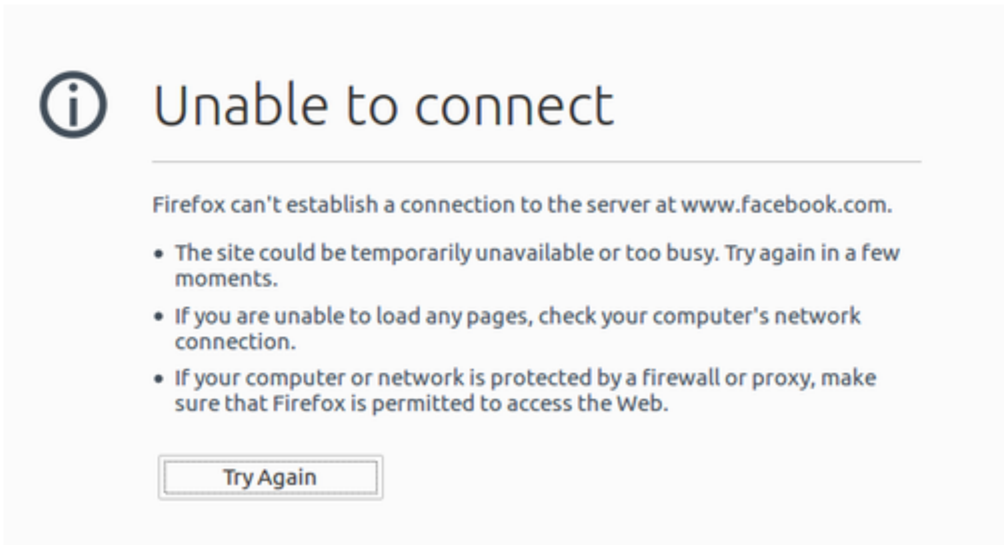
a) Once a user is authenticated with the Proxy he is able to access the Web. Then he starts to download an iso file. As we can see from the image below, the download speed is restricted to 61,8 KB/sec which is around to 500 Kb/sec (as we have expected). If another user starts to download a big file as well, then both users will share the 512Kb/sec bandwidth.



linuxmint-14-kde-dvd-64bit.iso
4 hours, 35 minutes remaining — 5,1 MB of 1,0 GB (61,8 KB/sec)

Figure 34. Bandwidth limitation

b) A user tries to access www.facebook.com with no success. The proxy denies access to the particular website.



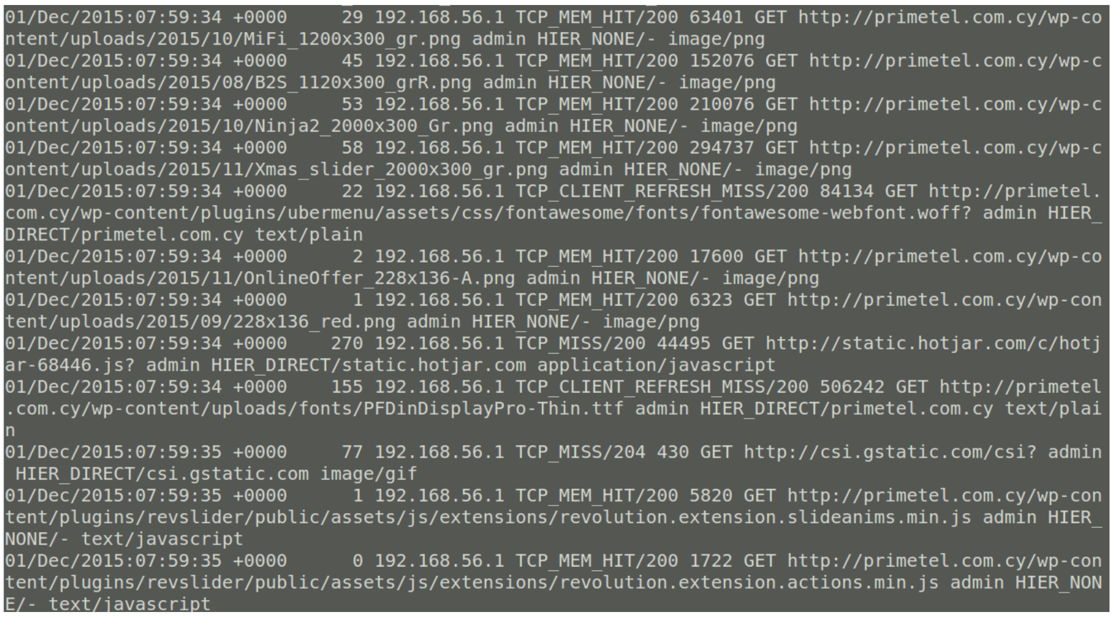
Unable to connect

Firefox can't establish a connection to the server at www.facebook.com.

- The site could be temporarily unavailable or too busy. Try again in a few moments.
- If you are unable to load any pages, check your computer's network connection.
- If your computer or network is protected by a firewall or proxy, make sure that Firefox is permitted to access the Web.

Figure 35. Access denied to www.facebook.com

c) The figure below shows the Squid's logs. In particular, it shows all the HTTP requests received by Squid from the clients and whether those requests result in cache hits. It can be observed that the particular requests were served from the Squid's cache. “TCP_MEM_HIT” shows that a request was served from Squid's Memory Cache (from the RAM).

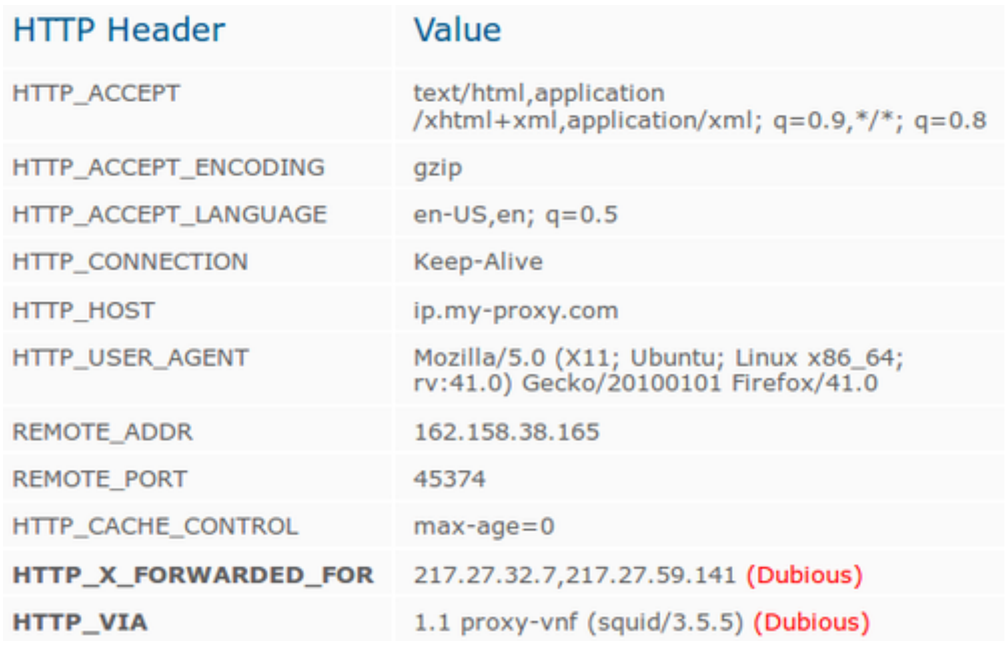


```
01/Dec/2015:07:59:34 +0000     29 192.168.56.1 TCP_MEM_HIT/200 63401 GET http://primetel.com.cy/wp-co
ntent/uploads/2015/10/MiFi_1200x300_gr.png admin HIER_NONE/- image/png
01/Dec/2015:07:59:34 +0000     45 192.168.56.1 TCP_MEM_HIT/200 152076 GET http://primetel.com.cy/wp-c
ontent/uploads/2015/08/B2S_1120x300_grR.png admin HIER_NONE/- image/png
01/Dec/2015:07:59:34 +0000     53 192.168.56.1 TCP_MEM_HIT/200 210076 GET http://primetel.com.cy/wp-c
ontent/uploads/2015/10/Ninja2_2000x300_Gr.png admin HIER_NONE/- image/png
01/Dec/2015:07:59:34 +0000     58 192.168.56.1 TCP_MEM_HIT/200 294737 GET http://primetel.com.cy/wp-c
ontent/uploads/2015/11/Xmas_slider_2000x300_gr.png admin HIER_NONE/- image/png
01/Dec/2015:07:59:34 +0000     22 192.168.56.1 TCP_CLIENT_REFRESH_MISS/200 84134 GET http://primetel.
com.cy/wp-content/plugins/ubermenu/assets/css/fontawesome/fonts/fontawesome-webfont.woff? admin HIER_
DIRECT/primetel.com.cy text/plain
01/Dec/2015:07:59:34 +0000      2 192.168.56.1 TCP_MEM_HIT/200 17600 GET http://primetel.com.cy/wp-co
ntent/uploads/2015/11/OnlineOffer_228x136-A.png admin HIER_NONE/- image/png
01/Dec/2015:07:59:34 +0000      1 192.168.56.1 TCP_MEM_HIT/200 6323 GET http://primetel.com.cy/wp-con
tent/uploads/2015/09/228x136_red.png admin HIER_NONE/- image/png
01/Dec/2015:07:59:34 +0000    270 192.168.56.1 TCP_MISS/200 44495 GET http://static.hotjar.com/c/hotj
ar-68446.js? admin HIER_DIRECT/static.hotjar.com application/javascript
01/Dec/2015:07:59:34 +0000    155 192.168.56.1 TCP_CLIENT_REFRESH_MISS/200 506242 GET http://primetel
.com.cy/wp-content/uploads/fonts/PFDinDisplayPro-Thin.ttf admin HIER_DIRECT/primetel.com.cy text/plai
n
01/Dec/2015:07:59:35 +0000     77 192.168.56.1 TCP_MISS/204 430 GET http://csi.gstatic.com/csi? admin
 HIER_DIRECT/csi.gstatic.com image/gif
01/Dec/2015:07:59:35 +0000      1 192.168.56.1 TCP_MEM_HIT/200 5820 GET http://primetel.com.cy/wp-con
tent/plugins/revslider/public/assets/js/extensions/revolution.extension.slideanims.min.js admin HIER_
NONE/- text/javascript
01/Dec/2015:07:59:35 +0000      0 192.168.56.1 TCP_MEM_HIT/200 1722 GET http://primetel.com.cy/wp-con
tent/plugins/revslider/public/assets/js/extensions/revolution.extension.actions.min.js admin HIER_NON
E/- text/javascript
```

Figure 36. Squid's logs

d) Figure 37 shows the results from http://ip.my-proxy.com/ when a user accesses the Web without having the user anonymity featured enabled. The most important fields are:

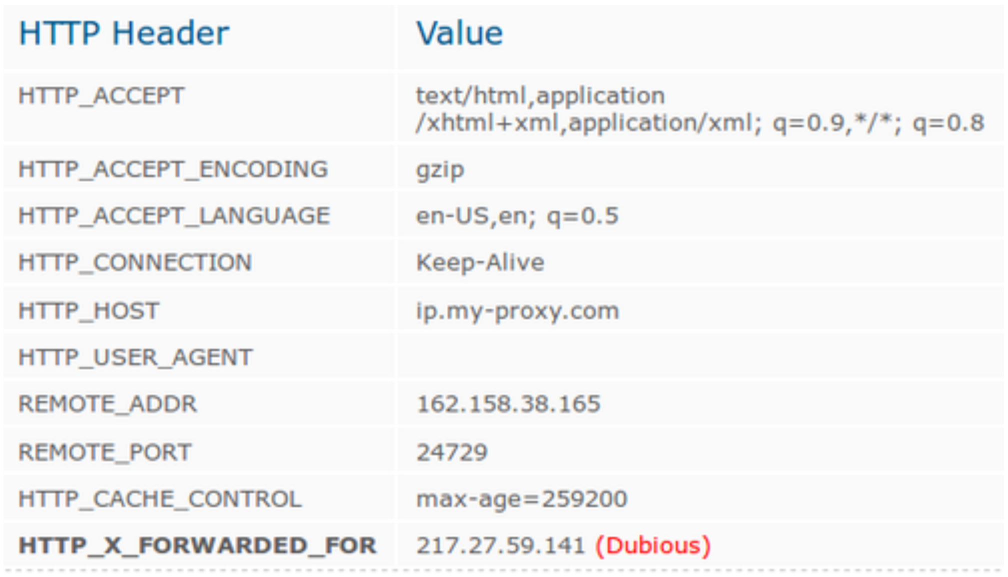
1. "HTTP_X_FORWARDED_FOR" . It shows the user's public IP (e.g. 217.27.32.7) address along with the Proxy's IP (e.g. 217.27.59.141)
2. "HTTP_VIA". It show the proxy's version (e.g. squid 3.5.5)
3. "HTTP_USER_AGENT". It shows the user's browser information (e.g. Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:41.0) Gecko/20100101 Firefox/41.0).



HTTP Header	Value
HTTP_ACCEPT	text/html,application /xhtml+xml,application/xml; q=0.9,*/*; q=0.8
HTTP_ACCEPT_ENCODING	gzip
HTTP_ACCEPT_LANGUAGE	en-US,en; q=0.5
HTTP_CONNECTION	Keep-Alive
HTTP_HOST	ip.my-proxy.com
HTTP_USER_AGENT	Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:41.0) Gecko/20100101 Firefox/41.0
REMOTE_ADDR	162.158.38.165
REMOTE_PORT	45374
HTTP_CACHE_CONTROL	max-age=0
HTTP_X_FORWARDED_FOR	217.27.32.7,217.27.59.141 (Dubious)
HTTP_VIA	1.1 proxy-vnf (squid/3.5.5) (Dubious)

Figure 37. Results taken from http://ip.my-proxy.com/ without user anonymity

Figure 38 shows the results while a user accesses the Web anonymously. It can be observed that the user's real IP is hidden and instead the Proxy's IP is shown. In addition the information about the proxy and the user's browser information are hidden.



HTTP Header	Value
HTTP_ACCEPT	text/html,application /xhtml+xml,application/xml; q=0.9,*/*; q=0.8
HTTP_ACCEPT_ENCODING	gzip
HTTP_ACCEPT_LANGUAGE	en-US,en; q=0.5
HTTP_CONNECTION	Keep-Alive
HTTP_HOST	ip.my-proxy.com
HTTP_USER_AGENT	
REMOTE_ADDR	162.158.38.165
REMOTE_PORT	24729
HTTP_CACHE_CONTROL	max-age=259200
HTTP_X_FORWARDED_FOR	217.27.59.141 (Dubious)

Figure 38. Results taken from http://ip.my-proxy.com/ with user anonymity.

3. Conclusions and Future Work

3.1. Conclusions

In this document, general guidelines for the development of Virtual Network Functions in the T-Nova framework have been provided. Specific descriptions of the Virtual Network Functions under development in the project have been given, together with some preliminary test results. The six VNF's under development have been described in detail, and the different technologies as well as the contributions coming from the open source community used in the development have been discussed. The obtained results clearly show how VNF development can greatly benefit both from open source software, as well as from the latest hw development. Also, some computational intensive functions have been implemented, without any significant loss in performance due to the adoption of virtualization technologies.

The presented VNF's cover a wide spectrum of applications, and can thus provide useful guidelines to new developers who are willing to create new functions to be used in the T-Nova framework. To this end, the initial paragraph of this document provide general information that can be useful to new developers through all the development phase of a new VNF.

3.2. Future work

In the next steps, the integration of the developed Virtual Network Functions has to be finalized, considering aspects such as scaling or high availability. The combination of different Virtual Network Functions in order to create new attractive Network Services will be addressed.

4. References

[Abgrall] Daniel Abgrall, "Virtual Home Gateway, How can Home Gateway virtualization be achieved?," EURESCOM, Study Report P055.

[Ansibl] http://en.wikipedia.org/wiki/Ansible_%28software%29

[Chef] http://en.wikipedia.org/wiki/Chef_%28software%29

[Chellouche2012] S. A. Chellouche, D. Negru, Y. Chen, et M. Sidibe, « Home-Box-assisted content delivery network for Internet Video-on-Demand services », in 2012 IEEE Symposium on Computers and Communications (ISCC), 2012, p. 000544-000550.

[Cruz] T. Cruz, P. Simões, N. Reis, E. Monteiro, F. Bastos, and A. Laranjeira, "An architecture for virtualized home gateways," in 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), 2013, pp. 520–526.

[Collectd] https://collectd.org/

[CUDA] NVIDIA, CUDA C Programming Guide, 2013.

[D2.1] T-NOVA: System Use Cases and Requirements

[D2.21] T-NOVA: Overall System Architecture and Interfaces

[D2.41] T-NOVA: Specification of the Network Function framework and T-NOVA Marketplace

[D4.01] T-NOVA: Interim Report on Infrastructure Virtualisation and Management

[D5.01] T-NOVA: Interim report on Network functions and associated framework

[D6.01] T-NOVA: Interim report on T-NOVA Marketplace implementation

[DITG] D-ITG, link: http://traffic.comics.unina.it/software/ITG/

[django] https://www.djangoproject.com/

[Docker] Docker https://www.docker.com/

[DPDK] Data Plane Development Kit, on-line: http://dpdk.org

[DPDK_NIC] DPDK Supported NICs, on-line: http://dpdk.org/doc/nics

[DPDK_RN170] INTEL, "Intel DPDK Kit", http://dpdk.org/doc/intel/dpdk-release-notes-1.7.0.pdf

[DSpace] DSpace http://www.dspace.org/

[Duato] Duato, J.; Pena, A.J.; Silla, F.; Fernandez, J.C.; Mayo, R.; Quintana-Orti, E.S., "Enabling CUDA acceleration within virtual machines using rCUDA," High Performance Computing (HiPC), 2011 18th International Conference on , vol., no., pp.1,10, 18-21 Dec. 2011 doi: 10.1109/HiPC.2011.6152718

[Egi] Egi, Norbert, et al. "Evaluating xen for router virtualization." Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on. IEEE, 2007.

[Eprints] Eprints http://www.eprints.org/

[ES282.001] ETSI ES 282 001: Telecommunications and Internet converged Services and Protocols for Advanced Networking (TISPAN); NGN Functional Architecture

[Fedora] http://www.fedora-commons.org/

[Felter] Felter, Wes, et al. "An Updated Performance Comparison of Virtual Machines and Linux Containers." technology 28: 32.

[Gelas] J.-P. Gelas, L. Lefevre, T. Assefa, and M. Libsie, "Virtualizaing home gateways for large scale energy reduction in wireline networks," in Electronics Goes Green 2012+ (EGG), 2012, 2012, pp. 1–7.

[GetA] http://docs.openstack.org/developer/heat/template_guide/hot_spec.html#hot-spec-intrinsic-functions

[GlueCon14] "Containers At Scale. At Google, the Google Cloud Platform and Beyond". Joe Beda. GlueCon 2014.

[Gupta] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-accelerated virtual machines," in 3rd Workshop on System-level Virtualization for High Performance Computing. NY, USA:ACM, 2009, pp. 17-24

[H2] H2 http://www.h2database.com/html/main.html

[Halon] http://www.halon.se/

[Herbaut2015] Herbaut N. ; Negru, D.; Xilouris G, Chen Y." in Network of the Future (NOF), 2015 International Conference and Workshop on the , 1-2 Oct. 2015, doi: 10.1109/NOF.2014.7119778

[IETF2013] https://datatracker.ietf.org/documents/LIAISON/liaison-2014-03-28-broadband-forum-the-ietf-broadband-forum-work-on-network-enhanced-residential-gateway-wt-317-and-virtual-business-gateway-wt-328-attachment-1.pdf

[Invenio] Cdsware http://cdsware.cern.ch/invenio/index.html

[IPERF] Iperf, link: https://iperf.fr/

[IPTraf] http://iptraf.seul.org/

[ITG] http://traffic.comics.unina.it/software/ITG/

[JBoss] JBoss http://www.jboss.org/

[Jetty] Eclipse http://eclipse.org/jetty/

[Karimi] K. Karimi, N.G. Dickson, F. Hamze, A Performance Comparison of CUDA and OpenCL, arXiv:1005.2581v3, arxiv.org.

[KeanMohd] http://core.kmi.open.ac.uk/download/pdf/11778682.pdf

[Kirk] D.B. Kirk, W.W. Hwu, Programming Massively Parallel Processors, 2nd ed., Morgan Kaufmann, 2013.

[Lauro] Di Lauro, R.; Giannone, F.; Ambrosio, L.; Montella, R., "Virtualizing General Purpose GPUs for High Performance Cloud Computing: An Application to a Fluid Simulator," Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on , vol., no., pp.863,864, 10-13 July 2012 doi: 10.1109/ISPA.2012.136

[LR_CPE] http://www.lightreading.com/nfv/nfv-elements/huawei-china-telecom-claim-virtual-cpe-first/d/d-id/710980

[LXC] https://linuxcontainers.org/

[IPTR] Iptraf, link: http://iptraf.seul.org/

[m0n0wall] http://m0n0.ch/wall/

[Maurice] C. Maurice, C. Neumann, Olivier Heen, and A. Francillon, "Confidentiality Issues on a GPU in a Virtualized Environment," Proceedings of the Eighteenth International Conference on Financial Cryptography and Data Security (FC'14),

[Mei] Mei Hwan Loke t al. (2006) Comparison of Video Quality metrics on multimedia videos

[Mikityuk] Mikityuk, A., J.-P. Seifert, and O. Friedrich. "The Virtual Set-Top Box: On the Shift of IPTV Service Execution, Service Amp; UI Composition into the Cloud." In 2013 17th International Conference on Intelligence in Next Generation Networks (ICIN), 1–8, 2013. doi:10.1109/ICIN.2013.6670887

[Modig] Modig, Dennis. "Assessing performance and security in virtualized home residential gateways." (2014).

[Murano] Murano https://murano.readthedocs.org/en/latest/

[MySQL] MySQL http://www.mysql.com/

[Nafaa2008] A. Nafaa, S. Murphy, et L. Murphy, « Analysis of a large-scale VOD architecture for broadband operators: a P2P-based solution », IEEE Communications Magazine, vol. 46, n° 12, p. 47-55, déc. 2008.

[Nec] http://www.nec.com/en/press/201410/global_20141013_01.html

[Netmap] http://info.iet.unipi.it/~luigi/netmap/

[Netty] netty.io

[NFV001] "Network Functions Virtualisation (NFV); Use Cases," ETSI GS NFV 001 V1.1.1, Oct. 2013.

[NFVMAN] ETSI NFV ISG, ETSI GS NFV-MAN 001: "Network Functions Virtualisation (NFV); Management and Orchestration" v1.1.1, ETSI, Dec 2014

[NFVSWA] ETSI NFV ISG, ETSI GS NFV-SWA 001: "Network Functions Virtualisation (NFV); Virtual Network Functions architecture" v1.1.1, ETSI, Dec 2014

[Nvidia] Nvidia, Cuda C Programming Guide, 2015

[OpenCL] AMD, Introduction to OpenCLTM Programming, 2014.

[OpenNF] http://opennf.cs.wisc.edu/

[OpenStack] OpenStack http://www.openstack.org/

[OpenVZ] http://openvz.org/Main_Page

[OPNFV] https://www.opnfv.org/

[Ostinato] https://code.google.com/p/ostinato/

[PFRing] http://www.ntop.org/products/pf_ring/

[pfSense] https://www.pfsense.org/

[PostgreSQL] PostgreSQL http://www.postgresql.org/

[Puppet] http://en.wikipedia.org/wiki/Puppet_%28software%29

[RD048] "HG REQUIREMENTS FOR HGI OPEN PLATFORM 2.0," HGI - RD048, May 2014.

[reddit] http://www.reddit.com/r/networking/comments/1rpk3f/evaluating_virtual_firewallrouters_vsrx_csr1000v/

[REFnDPI] ntop, "Open and Extensible LGPLv3 Deep Packet Inspection Library", on-line: http://www.ntop.org/products/ndpi/

[REFPACE] IPOQUE, "Protocol and Application Classification with Metadata Extraction", on-line: http://www.ipoque.com/en/products/pace

[REFWind] Wind River, "Wind River Content Inspection Engine" on-line: http://www.windriver.com/products/product-overviews/PO_Wind-River-Content-Inspection-Engine.pdf

[RFC1157] A Simple Network Management Protocol (SNMP)

[RFC2647] Benchmarking Terminology for Firewall Performance

[RFC3511] RTP Profile for Audio and Video Conferences with Minimal Control

[RFC4080] Next Steps in Signaling (NSIS)

[RFC4540] NEC's Simple Middlebox Configuration (SIMCO)

[RFC4741] NETCONF Configuration Protocol

[RFC7047] The Open vSwitch Database Management Protocol

[Salt] http://www.saltstack.com/

[SBC_ACME] http://www.acmepacket.com/products-services/service-provider-products/session-border-controller-net-net-session-director, Retrived Nov 2014

[SBC_ALU] http://www.alcatel-lucent.com/solutions/ip-border-controllers, Retrived Nov 2014

[SBC_Audiocodes] http://www.audiocodes.com/sbc, Retrived Nov 2014

[SBC_Italtel] http://www.italtel.com/en/products/session-border-controller, Retrived Nov 2014

[SBC_Metaswitch] http://www.metaswitch.com/products/sip-infrastructure/perimeta, Retrived Nov 2014

[SBC_Sonus] http://www.sonus.net/en/products/session-border-controllers/sonus-session-border-controllers-sbc, Retrived Nov 2014

[Shi] Lin Shi; Hao Chen; Jianhua Sun; Kenli Li, "vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines," Computers, IEEE Transactions on , vol.61, no.6, pp.804,816, June 2012 doi: 10.1109/TC.2011.112

[Sigcomm] http://www.sigcomm.org/sites/default/files/ccr/papers/2012/October/2378956-2378962.pdf

[Silva] R. L. Da Silva, M. A. C. Fernandez, L. E. I. Gamir, and M. F. Perez, “Home routing gateway virtualization: An overview on the architecture alternatives,” in Future Network Mobile Summit (FutureNetw), 2011, 2011, pp. 1–9.

[SNORT] Snort, link: https://www.snort.org/

[TheAge] http://www.theage.com.au/it-pro/business-it/telstra-and-ericsson-testing-virtual-home-gateway-20140721-zv6rh.html

[Tomcat] Tomcat http://tomcat.apache.org/

[TR-124] Broadband Forum, “Functional Requirements for Broadband Residential Gateway Devices, ”TECHNICAL REPORT TR-124”, Dec. 2006.

[TS23.228] 3GPP TS 23.228 IP Multimedia Subsystem (IMS)

[TS29.238] 3GPP TS 29.238 Interconnection Border Control Functions (IBCF) - Transition Gateway (TrGW) interface, Ix interface

[VBOX] VirtualBox, link: https://www.virtualbox.org/

[vSwitch] INTEL, “Intel DPDK vSwitch”, on-line: https://01.org/sites/default/files/downloads/packet-processing/329865inteldpdkvswitchgsg09.pdf

[Vuurmuur] http://www.vuurmuur.org/trac/

[Vyatta] http://www.vyatta.com

[Walters] J. P. Walters, A. J. Younge,D.-I. Kang, K.-T. Yao, M. Kang, S. P. Crago, and G. C. Fox, “GPU-Passthrogh Performance: A Comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL Applications,” in Proceedings of the 7th IEEE International Conference on Cloud Computing (CLOUD 2014), IEEE. Anchorage, AK: IEEE, 06/2014

[web2py] http://www.web2py.com/

[Wfirewalls] http://en.wikipedia.org/wiki/Comparison_of_firewalls

5. List of Acronyms

Acronym	Explanation
3GPP	Third Generation Partnership Project
API	Application Programming Interface
BGF	Border Gateway Function
CAPEX	Capital Expenditures
CLI	Command Line Interface
CRUD	Create, Read, Update, Delete
DDoS	Distributed Denial of Service
DFA	Deterministic Finite Automaton
DHCP	Dynamic Host Configuration Protocol
DoS	Denial of Service
DPDK	Data Plane Development Kit
DPI	Deep Packet Inspection
DPS	Data Plane Switch
DSP	Digital Signal Processor
DUT	Device Under Test
EMS	Element Management System
ETSI	European Telecommunications Institute
FW	Firewall
HGI	Home Gateway Initiative
HPC	High Performance Computing
HTTP	Hyper Text Transport Protocol
IBCF	Interconnection Border Control Function
IDS	Intrusion Detection System
IETF	Internet Engineering Task Force
IOMMU	I/O Memory Management Unit
ITU-T	International Telecommunication Union – Telecommunication Standardization Bureau
JSON	JavaScript Object Notation
KVM	Kernel-based Virtual Machine
LB	Load Balancer
MANO	Management and Orchestration

Acronym	Explanation
MIB	Management Information Base
NAT	Network Address Translation
NDVR	Network Digital Video Recorder
NETCONF	Network Configuration Protocol
NF	Network Function
NFaaS	Network Function as a Service
NFVI	Network Function Virtualization Infrastructure
NIO	Non Blockio I/O
NN	Neural Network
NPU	Network Processor Unit
NSIS	Next Steps In Signaling
O&M	Operating and Maintenance
OPEX	Operational Expenditures
OSGI	Open Service Gateway Initiative
OTT	over-the-top
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
POC	Proof of Concept
PSNR	Peak Signal-to-Noise Ratio
QoE	Quality of Experience
RAID	Redundant Array of Independent Disks
REST	Representational State Transfer
RFC	Request For Comments
RGW	Residential Gateway
RTP	Real-time Transport Protocol
SA	Security Appliance
SBC	Session Border Controller
SIMCO	Simple Middlebox Configuration
SIP	Session Initiation Protocol
SNMP	Simple Network Management Protocol
SOM	Self Organizing Maps
SQL	Structured Query Language
SR-IOV	Single Root I/O Virtualization

Acronym	**Explanation**
SSH	Secure Shell
STB	Set Top Box
TSTV	Time Shifted TV
UTM	Unified Threat Management
vCPE	virtualized customer premises equipment
VF	Virtual Firewall
vHG	Virtual Home Gateway
VIM	Virtual Infrastructure Manager
VM	Virtual Machine
VNF	Virtual Network Function
VOD	Video On Demand
VQM	Video Quality Metric
vSA	Virtual Security Appliance
vSBC	Virtual Session Border Controller
XML	Extensible Markup Language